GenTest
April 3, 2020

Software Testing Plan

ALEX GROCE, ASSOCIATE PROFESSOR, SICCS
JOSHUA JOHNSON ZANE FINK SILU SHEN TRISTAN MILLER
MENTORED BY FABIO SANTOS

Version 1

Table of Contents

Introduction

It was August, 2009 when a car crash in southern California ended in disaster, resulting in the loss of a family of four. The cause: uncontrollable acceleration. In the coming months, investigators would find that negligent coding practices produced a defect in the car's electronic throttle system [2] which led to uncontrollable acceleration in Toyota vehicles. After nine more reports of similar uncontrollable acceleration, Toyota would be forced to recall over 8 million cars suspected of containing these software defects. At its peak, the recalls cost the company $54 million a day in lost sales revenue [2] and brough one of the world's most prominent car companies to the brink of financial collapse.

Instances like the one with Toyota demonstrate the enormous responsibility software has in modern life. From cars to toaster ovens, it is hard to find devices and services which do not rely on some form of internal software. Many of the issues found in software have enormous consequences for companies, their employees, and customers; more often than not the impact of a software issue, or bug, is not quantifiable. Creating well-crafted software has never been more important. Just as an editor prepares a manuscript for publication, software testing is the gate which ensures well-crafted, fully functioning software. Its main goal is to ensure that each module and system in a software product works effectively, in-line with requirements. Effective software testing can mean the difference between profit or litigation, and in some cases life or death.

Still, not many developers are aware of the techniques available to implement effective software testing [1]. The GenTest software hopes to make these techniques more available to developers through the creation of a pseudo-compiler which takes C++ unit tests from the testing framework DeepState and creates a standalone test in another C++ unit testing framework. When we say unit test, we are referring to a branch of software testing concerned with testing only individual modules and the code there within. Similarly, when we reference standalone tests, we refer to unit tests which can be compiled, and run with the same advanced techniques that DeepState employs, without the need for DeepState to be referenced at compile time.

The ultimate end-goal of this translation is to provide a simple and comprehensive method for a developer using DeepState to export its main advantages into an external unit testing library of their choice, efficiently.
Given GenTest is a project motivated by testing, we would be remiss not to test our own product. This Software Testing Plan outlines the policies and structure of testing for the GenTest product,

which is now in an alpha stage. To prove software reliability, GenTest plans to implement tests for the following categories: **unit, integration, and usability**. These categories are broken down by number of tests per module or component in the system below:

Unit Testing **-** Independent tests on core module functionality without the inclusion of any other system modules.

- *Translation Engine* - 15 tests
- *Binary Parser - 12 tests*
- *File Assembler - 17 tests*

Integration Testing - Tests designed to ensure proper functionality between components in the system.

- *File System Communication - 4 tests*
- *Component Communication - 2 tests*

Usability Testing - Tests designed to ensure the end-product meets requirements based on feedback from testers.

- *"Out-loud" Testing - 1 test*
- General Usability Test - *1 test*

The distribution of these tests is skewed toward unit testing. This is because most of the important features of the pseudo-compiler rely on the correct functioning of each individual component. In the event one of these components does not supply correct values, or provides an erroneous output, the main use case of the product--translating unit tests--would be unfulfilled. While integration and usability testing are important, less tests are allocated here because the product does not have a sophisticated GUI which requires intensive testing. We are confident that as long as individual modules function, the usability and interaction between them will not greatly impact operation.

In the following sections, we will discuss more details about these tests and the strategy for completing them. In Section 1, we will discuss unit testing, in Section 2 integration testing will be outlined, and finally in Section 3 details about usability testing will be provided.

Unit Testing

Unit testing is a sub-branch of software testing which performs focused, individual tests on single modules, their functions, or individual lines of code without any interactivity between other pieces of the system. Its goal is to ensure that each component in the system functions stably and to prove in the general case that there are as few software issues as possible. GenTest is using the C++ testing library Catch2 to complete its unit testing because it has two major advantages: 1) it is header file only and does not require installation and 2) it can be incorporated into the same working environment as the GenTest product to provide easy-to-access tests at any point in the future.

The testing coverage that our unit tests currently acheive is about 83.8% across all major modules of the system. Major modules typically refer to the *Translation Engine, File Assembler, and Binary Parser*, though if any deviations are made from this they will be mentioned. This metric was calculated by taking the total number of lines we have to test for major modules (1172 lines of code) and dividing the number of lines being tested (983 lines of code) by it. By major module, the lines of code being tested are:

- **Translation Engine -** (447/559)
- **File Assembler -** (415/415)
- **Binary Parser -**(121/198)

From the above metrics, the testing coverage by module is: Translation Engine -- 79%, File Assembler -- 100%, Binary Parser - 61%. The rationale for the distribution of testing between these major modules is established by the type of code included in each. Since the File Assembler is responsible for the construction of all individual solutions to subproblems in our problem space, it is important to ensure we have a close to 100% coverage of the functionality held within this module. In contrast, the Translation Engine and Binary Parser modules have code which merely aid in linking the modules together. This code should be focused on during integration testing. Therefore, these lines have been excluded during the unit testing phase.

While our test coverage metrics are computed per-line, the testing “unit” that will be used is by module function. This means that the test coverage listed above is incorporated through these functions. In the following discussion, we will outline by major module more details about the tests that will be performed, their boundary cases, and the justifications for why each test is actualized.

General Procedures

GP-01

Name: GenTest Installation

Overview: Outlines the necessary steps to install DeepState if it is not already.

Execution:

Step 1. If not already installed, checkout the GenTest software and DeepState from: https://github.com/ZwFink/deepstate.git via the command-line by performing the following:

1. *sudo apt update && sudo apt-get install build-essential gcc-multilib g++-multilib cmake python3-setuptools libffi-dev z3*
    a.
2. *git clone* https://github.com/ZwFink/deepstate.git

Step 2. Once the software has been checked-out, navigate to the directory containing the /deepstate folder located on your local machine in the directory where the code was checked out. Do not enter the /deepstate directory yet.

Step 3. Make, build, and install DeepState by running the following code:

1. *mkdir deepstate/build && cd deepstate/build*
2. *cmake ../*
3. *make*
4. *sudo make install*

GP-02

Name: Building GenTest Test Executable

Overview: Outlines the necessary steps to create the executable containing unit tests for the GenTest software.

Execution:

Step 1. Perform the steps outlined in **GP-01**.

Step 2. From the base deepstate directory, navigate to the directory containing the GenTest portion of the DeepState software repository.

1. *cd GenTest*

Step 3. Make, and build the GenTest test executable:

5. *cd build*
6. *cmake ../ -DCMAKE_BUILD_TYPE=Debug*
7. *make gentest_test*

Translation Engine

The tests performed on the Translation Engine module are outlined below. For each test performed, the test id, name, description, boundary values, and process are provided.

U-01

Name: Can Place Multi-Line
Overview: Tests whether or not the system can appropriately place multi-line comments that are parsed by ANTLR4 into the appropriate structure and position in the output abstract syntax tree (AST). Further ensures for different syntax in C++ files that the format, placement, and recognition of multi-line comments is not affected.
Partition Values: Any syntax found in the input C++ file that begins with the keyword /* and ends after any number of lines in */.
Boundary Values: Syntax found in the input C++ file that begins with any reserved keyword in a multi-line comment as described in partition values or which uses this word within the comment.
Execution: **Step 1.** Navigate to /deepstate/GenTest and build the GenTest software by: 1. `cd ../` 2. `cd GenTest` 3. `cmake  .` 4. `make`
Step 2. Navigate to the examples directory by: 1. `cd ../` 2. `cd /build/examples`
Step 3. Run the GenTest software on the U01IntegerOverflow executable by writing the following in the command-line: `./U01IntegerOverflow --output_standalone U01Standalone.cpp --input_source_file ../../examples/U01IntegerOverflow.cpp --input_test_file /binary_files/U01.fai`

```
        --input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg
```

Step 4. After program execution ends, check the output file, U01Standalone.cpp for correct identification and propagation of all multi-line comments from the original file. There will be four major multi-line comments labeled via comment as U01A, U01B, U01C, and U01D.

If the multi-line comment was propagated and identified correctly the following should be true:

1. The multi-line comment is in the same relative position as in the original file.
2. The multi-line comment retains the same number of lines, spacing, and formatting as in the original file.

Multi-Line Comments by U01 Code

```
U01A = /* This is a sample multi-line comment on one line */
U01B = /* This is a sample multi-line comment across
              multiple lines. */
U01C = /*
          * Copyright (c) 2019 Trail of Bits, Inc.
          *
          * Licensed under the Apache License, Version 2.0 (the "License");
          * you may not use this file except in compliance with the License.
          * You may obtain a copy of the License at
          *
          *     http://www.apache.org/licenses/LICENSE-2.0
          *
          * Unless required by applicable law or agreed to in writing, software
          * distributed under the License is distributed on an "AS IS" BASIS,
          * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
         implied.
          * See the License for the specific language governing permissions and
          * limitations under the License.
          */
U01D = /* For all if */
```

Goal per Multiline Comment:

U01A - Check if multi-line comments are identified and propagated on only one line.
U01B - Check if multi-line comments are identified and propagated across lines >= 2.
U01C - Check that hyperlinks, typical characters, special characters, and lexer keywords such as “Deepstate” are appropriately identified and propagated across a complex, multi-line comment.
U01D - Check that C++ keywords such as if and for at the beginning and within the comment do not affect the propagation or identification of multiline comments.

U-02

Name: Can Propagate Single-Line Comments

Overview: Tests whether or not the system can appropriately place single-line comments that are parsed by ANTLR4 into the appropriate structure and position in the output AST. Further ensures for different syntax in C++ files that the format, placement, and recognition of multi-line comments is not affected.

Partition Values: Any syntax found in the input C++ file that begins with the keyword //and ends with a newline.

Boundary Values: Syntax found in the input C++ file that begins with any reserved keyword in a comment as described in partition values, one which uses this word within the comment, or a comment which is placed directly above a line of code.

Execution
Step 1. Navigate to /deepstate/GenTest and build the GenTest software by:

1. `cd ../`
2. `cd GenTest`
3. `cmake  .`
4. `make`

Step 2. Navigate to the examples directory by:

1. `cd ../`
2. `cd /build/examples`

Step 3. Run the GenTest software on the U02IntegerOverflow executable by writing the following in the command-line:

```
./U02IntegerOverflow --output_standalone U02Standalone.cpp --input_source_file
../../examples/U02IntegerOverflow.cpp --input_test_file /binary_files/U02.fai
--input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg
```

Step 4. After program execution ends, check the output file, U02Standalone.cpp for correct identification and propagation of all single-line comments from the original file. There will be four major single-line comments labeled via comment as U02A, U02B, U02C, and U02D.

If the single-line comment was propagated and identified correctly the following should be true:

1. The single-line comment is in the same relative position as in the original file.
2. The single-line comment retains the same number of lines, spacing, and formatting as in the original file.

Multi-Line Comments by U01 Code

U02A = // This is a simple comment.
U02B = // This is a comment above a line of code.
U02C = // For all the steps in this loop...
U02D = /* If this condition is evaluated...

Goal per Multiline Comment:

U02A - Check if single-line comments are identified and propagated.
U02B - Check if single-line comments do not absorb code when placed above a code line.
U02C - Check that C++ keywords such as “for” or “if’ at the beginning of the comment do not affect the propagation or identification of comments.
U02D - Check that C++ keywords such as “if” and “for” within the comment do not affect the propagation or identification of comments.

U-03

Name: Can Identify Namespace
Overview: Tests whether namespace expressions can be correctly identified and propagated into the standalone output file.
Partition Values: Any syntax found in the input C++ file that begins with the keywords using namespace and ends with a semicolon.
Boundary Values: Any namespace declaration which includes lexer keywords such as “DeepState”.
Execution **Step 1.** Navigate to /deepstate/GenTest and build the GenTest software by: 1. cd ../ 2. cd GenTest 3. cmake . 4. make
Step 2. Navigate to the examples directory by:

1. `cd ../`   2. `cd /build/examples`
Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line: `./U03IntegerOverflow --output_standalone U03Standalone.cpp --input_source_file ../../examples/U03IntegerOverflow.cpp --input_test_file /binary_files/U03.fai --input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg`
Step 4. After execution, inspect the U03Standalone.cpp file to ensure that the namespace “using namepsace std;” has been placed in the same relative position and propagated in the Standalone file.
Step 5. Change the U03Standalone.cpp file to have the namespace “using namespace Deepstate;”. Make the examples directory under build by typing “make”. Then execute step 5 again.
Step 6. After execution, inspect the U03Standalone.cpp file to ensure that the namespace “using namespace Deepstate;” has been placed in the same relative position and propagated in the Standalone file.

U-04

Name: Can Identify Include
Overview: Tests whether include expressions can be correctly identified and propagated into the standalone output file.
Partition Values: Any syntax found in the input C++ file that begins with the keyword #include.
Boundary Values: Any namespace declaration which includes lexer keywords such as “DeepState”.
Execution: **Step 1.** Navigate to /deepstate/GenTest and build the GenTest software by: 1. `cd ../`   2. `cd GenTest`   3. `cmake .`

```
4. make
```

Step 2. Navigate to the examples directory by:

```
1. cd ../
```

```
2. cd /build/examples
```

Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line:

```
./U04IntegerOverflow --output_standalone U04Standalone.cpp --input_source_file
../../examples/U04IntegerOverflow.cpp --input_test_file /binary_files/U04.fai
--input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg
```

Step 4. After execution, inspect the U04Standalone.cpp file to ensure that the include statement has been placed in the same relative position and propagated in the Standalone file.

U-05

Name: Can Identify Define

Overview: Tests whether define expressions can be correctly identified and propagated into the standalone output file.

Partition Values: Any syntax found in the input C++ file that begins with the keywords #define.

Boundary Values: Any namespace declaration which includes lexer keywords such as “DeepState”.

Execution:
Step 1. Navigate to /deepstate/GenTest and build the GenTest software by:

```
1. cd ../
```

```
2. cd GenTest
```

```
3. cmake .
```

```
4. make
```

Step 2. Navigate to the examples directory by:

```
1. cd ../
```

2. cd /build/examples

Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line:

```
./U05IntegerOverflow --output_standalone U05Standalone.cpp --input_source_file
../../examples/U05IntegerOverflow.cpp --input_test_file /binary_files/U05.fai
--input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg
```

Step 4. After execution, inspect the U05Standalone.cpp file to ensure that the define statement has been placed in the same relative position and propagated in the Standalone file.

U-06

Name: Can Identify Structure Header

Overview: Tests whether Structure Headers are appropriately pulled in and configured.

Partition Values: Any syntax found in the input C++ file that begins with the keyword struct.

Boundary Values: Single-line struct definitions.

Execution:
Step 1. Navigate to /deepstate/GenTest and build the GenTest software by:

1. cd ../
2. cd GenTest
3. cmake .
4. make

Step 2. Navigate to the examples directory by:

1. cd ../
2. cd /build/examples

Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line:

```
./U06IntegerOverflow --output_standalone U06Standalone.cpp --input_source_file
../../examples/U06IntegerOverflow.cpp --input_test_file /binary_files/U06.fai
```

```
--input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg
```

Step 4. After execution, inspect the U06Standalone.cpp file to ensure that the structure header has been propagated into the AST in the same position as the original file and is not missing any content.

U-07

Name: Can Identify Function Headers

Overview: Tests whether function header can be appropriately identified and placed into the AST.

Partition Values: Any syntax found in the input C++ file that begins with keywords, a type, and a name, followed by a parameter list and an open bracket either on the same line or the next line.

Boundary Values: Any function header that includes a library call such as ASTListener::myFunction(){

Execution:
Step 1. Navigate to /deepstate/GenTest and build the GenTest software by:

1. cd ../
2. cd GenTest
3. cmake .
4. make

Step 2. Navigate to the examples directory by:

1. cd ../
2. cd /build/examples

Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line:

```
./U07IntegerOverflow --output_standalone U07Standalone.cpp --input_source_file
../../examples/U07IntegerOverflow.cpp --input_test_file /binary_files/U07.fai
--input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg
```

Step 4. After execution, inspect the U07Standalone.cpp file to ensure that all contained functions have been identified correctly and placed in the AST.

U-08

Name: Can Identify Statement
Overview: Tests whether statements, such as int x = 5;, are correctly identified and propagated into the AST.
Partition Values: Any syntax found in the input C++ file that ends with a semicolon and does not begin with reserved keywords for other constructs.
Boundary Values: Any namespace declaration which includes lexer keywords such as “DeepState”.
Execution: **Step 1.** Navigate to /deepstate/GenTest and build the GenTest software by: 1. cd ../ 2. cd GenTest 3. cmake . 4. make
Step 2. Navigate to the examples directory by: 1. cd ../ 2. cd /build/examples
Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line: `./U08IntegerOverflow --output_standalone U08Standalone.cpp --input_source_file ../../examples/U08IntegerOverflow.cpp --input_test_file /binary_files/U08.fai --input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg`
Step 4. After execution, inspect the U08Standalone.cpp file to ensure that the statement line has been propagated correctly in the AST.
Step 5. Repeat steps 6 and 7 for U08Euler.cpp, U08Crash.cpp, and U08Primes.cpp.

U-09

Name: Can Identify Typedef
Overview: Tests whether typedef expressions can be correctly identified and propagated into the standalone output file.

Partition Values: Any syntax found in the input C++ file that begins with the keyword typedef.

Boundary Values: Any typedef declaration which includes enumerations.

Execution:
Step 1. Navigate to /deepstate/GenTest and build the GenTest software by:

1. cd ../
2. cd GenTest
3. cmake .
4. make

Step 2. Navigate to the examples directory by:

1. cd ../
2. cd /build/examples

Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line:

```
./U09IntegerOverflow --output_standalone U03Standalone.cpp --input_source_file
../../examples/U09IntegerOverflow.cpp --input_test_file /binary_files/U09.fai
--input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg
```

Step 4. After execution, inspect the U09Standalone.cpp file to ensure that the typedefinition header has been propagated into the AST and standalone file in the same position and with the same syntax.

U-10

Name: Can Identify Test Header

Overview: Tests whether test header expressions can be identified and propagated into the output standalone file.

Partition Values: Any syntax found in the input C++ file that begins with the keyword TEST.

Boundary Values: Any TEST declaration which includes lexer keywords such as "DeepState".

Execution:
Step 1. Navigate to /deepstate/GenTest and build the GenTest software by:

1. cd ../ 2. cd GenTest 3. cmake . 4. make
Step 2. Navigate to the examples directory by: 1. cd ../ 2. cd /build/examples
Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line: `./U10IntegerOverflow --output_standalone U10Standalone.cpp --input_source_file ../../examples/U10IntegerOverflow.cpp --input_test_file /binary_files/U10.fai --input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg`
Step 4. After execution, inspect the U10Standalone.cpp file to ensure the all tests in the original file were placed correctly into the AST and standalone file.

U-11

Name: Can Identify While Loops
Overview: Tests whether While loop headers can be correctly identified.
Partition Values: Any syntax found in the input C++ file that begins with the keyword while.
Boundary Values: Any while loop header with complex statements such as (x + z) != y/5
Execution: **Step 1.** Navigate to /deepstate/GenTest and build the GenTest software by: 1. cd ../ 2. cd GenTest 3. cmake . 4. make
Step 2. Navigate to the examples directory by:

1. cd ../ 2. cd /build/examples
Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line: `./U11IntegerOverflow --output_standalone U11Standalone.cpp --input_source_file ../../examples/U11IntegerOverflow.cpp --input_test_file /binary_files/U11.fai --input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg`
Step 4. After execution, inspect the U11Standalone.cpp file to ensure that the while loop header was correctly propagated into the AST and standalone file.

U-12

Name: Can Identify For Loops
Overview: Tests whether for loops are identifiable and propagate correctly to the output standalone ifle.
Partition Values: Any syntax found in the input C++ file that begins with the keyword for.
Boundary Values: Any for loop header with complex statements such as (x + z) != y/5
Execution: **Step 1.** Navigate to /deepstate/GenTest and build the GenTest software by: 1. cd ../ 2. cd GenTest 3. cmake . 4. make
Step 2. Navigate to the examples directory by: 1. cd ../ 2. cd /build/examples
Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line:

./U12IntegerOverflow --output_standalone U12Standalone.cpp --input_source_file ../../examples/U12IntegerOverflow.cpp --input_test_file /binary_files/U12.fai --input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg
Step 4. After execution, inspect the U12Standalone.cpp file to ensure that the for loop header has been correctly propagated into the output standalone file and the AST.

U-13

Name: Can Identify Conditionals
Overview: Tests whether Conditional expressions are identifiable and propagate into the output standalone file.
Partition Values: Any syntax found in the input C++ file that begins with the keyword if.
Boundary Values: Any conditional headers with complex expressions.
Execution: **Step 1.** Navigate to /deepstate/GenTest and build the GenTest software by: 1. cd ../ 2. cd GenTest 3. cmake . 4. make
Step 2. Navigate to the examples directory by: 1. cd ../ 2. cd /build/examples
Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line: ./U13IntegerOverflow --output_standalone U13Standalone.cpp --input_source_file ../../examples/U13IntegerOverflow.cpp --input_test_file /binary_files/U13.fai --input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg
Step 4. After execution, inspect the U13Standalone.cpp file to ensure that the conditional header was propagated into the standalone output file correctly and in the same position.

U-14

Name: Can Identify ASSERT/CHECK/ASSUME
Overview: Tests whether assert/check/assume statements can be identified and placed into the AST.
Partition Values: Any syntax found in the input C++ file that begins with the keywords assert, check, assume.
Boundary Values: Any syntax that also includes Deepstate before the keyword.
Execution: **Step 1.** Navigate to /deepstate/GenTest and build the GenTest software by: 1. cd ../ 2. cd GenTest 3. cmake . 4. make
Step 2. Navigate to the examples directory by: 1. cd ../ 2. cd /build/examples
Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line: `./U14IntegerOverflow --output_standalone U14Standalone.cpp --input_source_file ../../examples/U14IntegerOverflow.cpp --input_test_file /binary_files/U14.fai --input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg`
Step 4. After execution, inspect the U14Standalone.cpp file to ensure that the statement has been placed in the same relative position and propagated in the AST.
Step 5. Change the U14Standalone.cpp file to contain a different version of check/assume/assert clauses. Make the examples directory under build by typing “make”. Then execute steps 6 and 7 again.
Step 6. Repeat steps 6, 7, and 8 until all permutations of check/assert/assume clauses are covered.

U-15

Name: Can Identify Symbolic
Overview: Tests whether symbolic declarations are propagated correctly into the AST.
Partition Values: Any syntax found in the input C++ file that begins with the keyword “Symbolic” or “symbolic”.
Boundary Values: Any syntax that invokes a direct call to DeepState for a value such as Deepstate_int().
Execution: **Step 1.** Navigate to /deepstate/GenTest and build the GenTest software by: 1. cd ../ 2. cd GenTest 3. cmake . 4. make
Step 2. Navigate to the examples directory by: 1. cd ../ 2. cd /build/examples
Step 3. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line: `./U15IntegerOverflow --output_standalone U15Standalone.cpp --input_source_file ../../examples/U15IntegerOverflow.cpp --input_test_file /binary_files/U15.fai --input_translation_config ../../GenTest/test/test_data/gtestTranslation.cfg`
Step 4. After execution, inspect the U15Standalone.cpp file to ensure that symbolic_int, Symbolic<int>, and Deepstate_int() are all placed in the AST correctly.

File Assembler

The tests performed on the File Assembler module are outlined below. For each test performed, the test id, name, description, boundary values, and process are provided. We note that each test for the Binary Parser module is located in the unit test file of the GenTest software repository. Each test case lists the steps that will be performed by the code in a unit test. The FIle Assembler relies on several sub modules. Each module or submodule will get it’s own individual test case, and each test will have its own section.

Building

Because the unit tests for the Binary Parser are located within the unit test file of the GenTest software repository, a one-time build process must be performed. Once performed, the build process does not need to be repeated before any tests are run. The steps for this process are outlined in **GP-02**.

Translation Dictionary

The translation dictionary is a submodule of the file assembler. The translation dictionary handles loading of a translation configuration, as well as translating that configuration into a form that the File Assembler can utilize.

U-01

Name: Correct loading of a file
Overview: The Translation dictionary is able to load a correct configuration file
Partition Values: A file containing the translation configuration
Boundary Values: Files which are missing from the file system
Step 1. The name of the configuration file (*../test/gtestTranslation.cfg*) is stored in a string.
Step 2. An attempt to load the configuration file into that of the Translation Dictionary
Step 3. Assert that the return from the file load is true, meaning that it was correctly loaded

U-02

Name: Identify Incorrect loading of file
Overview: The Translation dictionary is able to identify an incorrect load
Partition Values: A file containing the incorrect translation configuration
Boundary Values: Files which are missing from the file system
Step 1. The name of the configuration file (*../test/gtestTranslationMissingVital.cfg*) is stored in a string.
Step 2. An attempt to load the configuration file into that of the Translation Dictionary
Step 3. Assert that the return from the file load is false, meaning there was an error during load

U-03

Name: Ensure values can be correctly searched and accessed
Overview: Make sure that given a correctly loaded TranslationDictionary, an entry can be searched
Partition Values: A correctly loaded TranslationDictionary
Boundary Values: A NTerminal without a translation
Step 1. Load the TranslationDictionary for access
Step 2. Call findTranslationFromNTerminal with a valid NTerminal
Step 3. Assert that the values located inside the returned TranslationEntry are correct

U-04

Name: Ensure invalid translations are identified
Overview: Make sure that given a correctly loaded TranslationDictionary, an entry can be searched
Partition Values: A correctly loaded TranslationDictionary
Boundary Values: None
Step 1. Load the TranslationDictionary for access
Step 2. Call findTranslationFromNTerminal with an invalid NTerminal
Step 3. Assert that the returned TranslationEntry is new, meaning the translation doesn’t exist

StructHandler

The Struct Handler is another submodule utilized in the File Assembler. The Struct Handler will take in the current AST and develop a data structure which can then later be used to insert values on instantiation of that particular struct.

U-05

Name: Ensure correct loading of the structHandler
Overview: Ensures that the structHandler can correctly load the structs in an AST
Partition Values: An AST containing Structs

Boundary Values: Invalid or unintended AST
Step 1. Initialize a struct handler and a predefined AST
Step 2. Call the struct handler function lookForSymbolic using the AST
Step 3. Assert that the Struct Handler correctly loaded the data

U-06

Name: Ensure correct generation of struct values
Overview: Ensure that given a correct Struct handler, it can generate values for a given struct initialization
Partition Values: A struct initialization string and a binary iterator
Boundary Values: Invalid Structs
Step 1. Correctly Load a Struct Handler
Step 2. Call writeStatementFor with the partition values
Step 3. Assert that the generated statement is correct

Helper Functions

This subsection of the file assembler unit tests covers that of helper functions utilized by the File Assembler process. These helper functions perform actions based on a single type of translation.

U-07

Name: Ensure Symbolic Lines Can be correctly generated
Overview: The function symbolicLine will be run with every available type, and ensure that the process is loaded correctly
Partition Values: A name for the variable in the line, a correctly loaded iterator for getting the value, and a string containing the type in the line
Boundary Values: A type not yet implemented
Step 1. Load temporary partition values for the test
Step 2. Call symbolicLine for each type

Step 3. Assert that the output from symbolicLine is correct.

U-08

Name: Ensure deepstateTypeReturn works correctly
Overview: The function deepstateTypeReturn will be run with every available type, and ensure that the process is loaded correctly
Partition Values: An example node, one of every type, the string for the current line, and a binary iterator
Boundary Values: A type not yet implemented
Step 1. Load temporary partition values for the test
Step 2. Call deepstateTypeReturn for each type
Step 3. Assert that the output from deepstateTypeReturn is correct.

U-09

Name: Ensure questionConversion works correctly
Overview: The function questionConversion takes a question statement with a boolean operator and converts it to a question using a boolean operator inside the function
Partition Values: A string containing a question, the NTerminal for that question, and a translation dictionary
Boundary Values: non question NTerminals
Step 1. Load temporary partition values for the test
Step 2. Call questionConversion for each type of question that can be converted
Step 3. Assert that the output from questionConversion is correct

U-10

Name: Ensure questionTranslation works correctly
Overview: The function questionTranslation takes a question statement with a valid translation and translates it
Partition Values: A string containing the question and a translation dictionary
Boundary Values: non question NTerminals

Step 1. Load temporary values for the test
Step 2. Call questionTranslation for a translation
Step 3. Assert that the output from questionTranslation is correct

U-11

Name: Ensure symbolicValHandle works correctly
Overview: The function symbolicValHandle takes a symbolic line and handles conversion from a potential multi variable line to many single variable lines
Partition Values: A string containing a symbolic line, a binary iterator, and a string containing the data type
Boundary Values: non symbolic NTerminals and unimplemented types
Step 1. Load temporary values for the test
Step 2. Call symbolicValHandle for the symbolic line
Step 3. Assert that the output from symbolicValHandle is correct

U-12

Name: Ensure questionHandle works correctly
Overview: The function questionHandle takes a question statement and decides whether to convert it or translate it
Partition Values: A string containing the question, a translation dictionary, and the question NTerminal
Boundary Values: non question NTerminals
Step 1. Load temporary values for the test
Step 2. Call questionHandle for a translation
Step 3. Assert that the output from questionHandle is correct

U-13

Name: Ensure deepstateQuestionHandle works correctly
Overview: The function deepstateQuestionHandle takes a question statement and translates it

Partition Values: A string containing the question and a translation dictionary
Boundary Values: non DeepState_question NTerminals
Step 1. Load temporary values for the test
Step 2. Call deepstateQuestionHandle for a translation
Step 3. Assert that the output from deepstateQuestionHandle is correct

U-14

Name: Ensure deepstateQuestionHandle works correctly
Overview: The function deepstateQuestionHandle takes a question statement and translates it
Partition Values: A string containing the question and a translation dictionary
Boundary Values: non DeepState_question NTerminals
Step 1. Load temporary values for the test
Step 2. Call deepstateQuestionHandle for a translation
Step 3. Assert that the output from deepstateQuestionHandle is correct

U-15

Name: Ensure deepstateTypeHandle works correctly
Overview: The function deepstateTypeHandle takes a DeepState_Type statement and translates it
Partition Values: A string containing the type statement, a binary iterator, and the current node
Boundary Values: non DeepState_Type NTerminals and non implemented DeepState_Types
Step 1. Load temporary values for the test
Step 2. Call deepstateTypeHandle for a translation
Step 3. Assert that the output from deepstateTypeHandle is correct

U-16

Name: Ensure structHandle works correctly

Overview: The function structHandle takes a non translated line, determines whether or not it contains a struct initialization, and then inserts the values if so
Partition Values: A string containing the current line, a struct handler, and a binary iterator.
Boundary Values: non DeepState_Type NTerminals and non implemented DeepState_Types
Step 1. Load temporary values for the test
Step 2. Call deepstateTypeHandle for a translation
Step 3. Assert that the output from deepstateTypeHandle is correct

BuildFile

The build file function is the backbone of the File Assembler. The build file function utilizes all aforementioned helper functions to generate the standalone file.

U-17

Name: Ensure buildFile works correctly
Overview: The function buildFile takes in an ast as well as several file locations, then builds all that into a completed file.
Partition Values: An AST, a path for a binary file, an output path, and a path for the configuration file
Boundary Values: Unintended or unimplemented values
Step 1. Load predetermined ASTs for the tests
Step 2. Call buildFile on those ASTs
Step 3. Assert that the output from buildFile is correct

Binary Parser

The tests performed on the Binary Parser module are outlined below. For each test performed, the test id, name, description, boundary values, and process are provided. We note that each test for the Binary Parser module is located in the unit test file of the GenTest software repository. Each test case lists the steps that will be performed by the code in a unit test.

Building

Because the unit tests for the Binary Parser are located within the unit test file of the GenTest software repository, a one-time build process must be performed. Once performed, the build process does not need to be repeated before any tests are run. The steps for this process are outlined in **GP-02**.

U-01

Name: Can Open Files on Disk from Their Name
Overview: The Binary Parser is able to open a binary data that is found on this disk.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An attempt to open the test file is made via the BinaryParser's interface. Specifically, BinaryParser.parse is called with the string filename passed in.

U-02

Name: Can Open Files on Disk an Input Stream
Overview: The Binary Parser is able to process binary data from an input stream.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in step 1.
Step 3. An attempt to open the test file is made via the BinaryParser's interface. Specifically, BinaryParser.parse is called with the input file stream passed in.

Step 4. Assert that no error was thrown in Step 3.

U-03

Name: Complete File Contents Read Into Memory
Overview: The Binary Parser is able to read the entire contents of a binary file into memory.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser's interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. Another input filestream is read into a variable used by the test
Step 5. Assert that the data pointed to by BinaryParser.getData is equal to the data that was parsed directly from the file.

U-04

Name: Iterator Can Read Characters
Overview: The Binary Iterator is able to read a single character from the Binary Parser's data.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser's interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. An iterator for the BinaryParser is retrieved with BinaryParser.getIterator
Step 5. One character is retrieved from the iterator using BinaryIterator.nextChar.

Step 5. Assert the character is equal to the byte “0x00”, which is consistent with the string that is stored in the test file.

U-05

Name: Iterator Can Read All Data
Overview: The Binary Iterator is able to read all of the data that was parsed from the BinaryIterator.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser’s interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. An iterator for the BinaryParser is retrieved with BinaryParser.getIterator
Step 5. One character is retrieved from the iterator at a time using BinaryIterator.nextChar.
Step 6. Assert that the characters are retrieved in the expected order, according to the string above.

U-06

Name: An Integer Can Be Read
Overview: The Binary Parser is able to retrieve an integer from the binary data.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser’s interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. An iterator for the BinaryParser is retrieved with BinaryParser.getIterator

Step 5. An integer is retrieved from the iterator’s data with BinaryIterator.nextInt()
Step 6. Assert the bytes of the integer are equal to the first 4 bytes in the binary data file.

U-07

Name: A std::size_t Can Be Read
Overview: The Binary Parser is able to retrieve a std::size_t value from the binary data.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser’s interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. An iterator for the BinaryParser is retrieved with BinaryParser.getIterator
Step 5. A std::size_t is retrieved from the iterator’s data with BinaryIterator.nextSize_t()
Step 6. Assert the bytes of the retrieved value are equal to the first sizeof(std::size_t) bytes in the binary data file.

U-08

Name: A long Can Be Read
Overview: The Binary Parser is able to retrieve a long value from the binary data.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser’s interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. An iterator for the BinaryParser is retrieved with BinaryParser.getIterator

Step 5. A long is retrieved from the iterator’s data with BinaryIterator.nextLong
Step 6. Assert the bytes of the retrieved value are equal to the first 4 bytes in the binary data file.

U-9

Name: A float Can Be Read
Overview: The Binary Parser is able to retrieve a float value from the binary data.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser’s interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. An iterator for the BinaryParser is retrieved with BinaryParser.getIterator
Step 5. A float is retrieved from the iterator’s data with BinaryIterator.nextFloat
Step 6. Assert the bytes of the retrieved value are equal to the first 4 bytes in the binary data file.

U-10

Name: A double Can Be Read
Overview: The Binary Parser is able to retrieve a double value from the binary data.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser’s interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. An iterator for the BinaryParser is retrieved with BinaryParser.getIterator

Step 5. A double is retrieved from the iterator’s data with BinaryIterator.nextDouble
Step 6. Assert the bytes of the retrieved value are equal to the first 8 bytes in the binary data file.

U-11

Name: A std::uint64_t Can Be Read
Overview: The Binary Parser is able to retrieve a std::uint64_t value from the binary data.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser’s interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. An iterator for the BinaryParser is retrieved with BinaryParser.getIterator
Step 5. A std::uint64_t is retrieved from the iterator’s data with BinaryIterator.nextUint64_t
Step 6. Assert the bytes of the retrieved value are equal to the first 8 bytes in the binary data file.

U-12

Name: A std::uint32_t Can Be Read
Overview: The Binary Parser is able to retrieve a std::uint32_t value from the binary data.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser’s interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. An iterator for the BinaryParser is retrieved with BinaryParser.getIterator

Step 5. An std::uint32_t is retrieved from the iterator’s data with BinaryIterator.nextUint32_t
Step 6. Assert the bytes of the retrieved value are equal to the first 4 bytes in the binary data file.

U-13

Name: A short Can Be Read
Overview: The Binary Parser is able to retrieve a short value from the binary data.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The name of the test file (*../test/test_data.dat*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser’s interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. An iterator for the BinaryParser is retrieved with BinaryParser.getIterator
Step 5. An short is retrieved from the iterator’s data with BinaryIterator.nextShort
Step 6. Assert the bytes of the retrieved value are equal to the first 4 bytes in the binary data file.

Integration Testing

While unit testing is good for testing individual subcomponents of a system, Integration testing analyzes how the major components of a system interact. Our team plans on utilizing integration testing by testing functions where the individual modules interact. These tests will include the communication between the Translation Engine and the File Assembler, between the Binary Parser and the File Assembler, as well as between the file system and our system

I-01

Name: Can Return AST
Overview: Tests whether the AST provided to the File Assembler by the Translation Engine is populated with nodes and is the same as the internal Translation Engine AST.
Execution: **Step 1.** If not already installed, checkout the GenTest software and DeepState from: https://github.com/ZwFink/deepstate.git via the command-line by performing the following: 1. *sudo apt update && sudo apt-get install build-essential gcc-multilib g++-multilib cmake python3-setuptools libffi-dev z3* 2. *git clone* https://github.com/ZwFink/deepstate.git
Step 2. Once the software has been checked-out, navigate to the directory containing the /deepstate folder located on your local machine in the directory where the code was checked out. Do not enter the /deepstate directory yet.
Step 3. Make, build, and install DeepState by running the following code: 1. *mkdir deepstate/build && cd deepstate/build* 2. *cmake ../* 3. *make* 4. *sudo make install*
Step 4. Navigate to /deepstate/GenTest and build the GenTest software by: 1. cd ../

2. cd GenTest 3. cmake . 4. make
Step 5. Navigate to the GenTest directory by: 1. cd ../ 2. cd /GenTest
Step 6. Run the GenTest software on the U03IntegerOverflow executable by writing the following in the command-line: ./GenTestTest
Step 7. After execution, inspect the pass fail results for I-01 returned from Catch2.

I-02

Name: Error Thrown on File With Bad Input Stream
Overview: The Binary Parser is able to recognize that an input file stream is not able to be read correctly.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The incorrect name of the test file (*../test/test_data.d*) is stored in a string.
Step 2. An input stream (std::ifstream) is opened on the test file in Step 1.
Step 3. An attempt to open the test file is made via the BinaryParser's interface. Specifically, BinaryParser.parse is called with the input file stream passed in.
Step 4. Assert that a std::runtime_error is thrown in Step 3, and that ifstream.good returns false.

I-03

Name: Error Thrown on File With String Name not Found
Overview: The Binary Parser is able to recognize that a file specified by its string name was not found on the disk and throw a std::runtime_error accordingly.
Partition Values: A file containing binary data generated by DeepState.
Boundary Values: Files that are not found on the disk.
Step 1. The incorrect name of the test file (*../test/test_data.d*) is stored in a string.
Step 2. An attempt to open the test file is made via the BinaryParser's interface. Specifically, BinaryParser.parse is called with the string filename passed in.
Step 3. Assert that a std::runtime_error is thrown in Step 2.

I-04

Name: Error Thrown on File With String Name not Found
Overview: The Translation Dictionary is able to recognize that a file specified by its string name was not found on the disk and throw a std::runtime_error accordingly.
Partition Values: A configuration file.
Boundary Values: Files that are not found on the disk.
Step 1. The incorrect name of the test file (*../test/gtestNotReal.cfg*) is stored in a string.
Step 2. An attempt to open the test file is made via the Translation Dictionary's interface. Specifically, TranslationDictionary.loadFile is called with the string filename passed in.
Step 3. Assert that a std::runtime_error is thrown in Step 2.

I-05

Name: Error Thrown on save file when issue occurs
Overview: The File Assembler is able to recognize that an output file path specified by its string name is either non-existent, or there are invalid permissions.
Partition Values: An output path
Boundary Values: Invalid Permissions or File Location
Step 1. The incorrect name of the output filepath (*../test/invalid_folder/output.cpp*) is stored in a string.

Step 2. An attempt to save the file for output from the File Assembler.
Step 3. Assert that a std::runtime_error is thrown in Step 2.

I-06

Name: Ensure the File Assembler can call and use Binary Parser Values
Overview: The File Assembler is able to utilize the Binary Parser for inserting values, this functionality needs to be tested to ensure proper operation.
Partition Values: A binary data file
Boundary Values: Invalid Permissions or File Location
Step 1. The correct name of the binary data file (*../test/test_data/test.fail*) is stored in a string.
Step 2. An attempt to load a new value from the Binary Parser
Step 3. Assert that the value is correct, or throws an error if erroneous behavior

Usability Testing

As described in the introduction, Gentest is a system that gets unit tests in DeepState and transforms them into independent unit tests in an external framework chosen by developers. The purpose of developing GenTest is to provide a simple and comprehensive method for developers using DeepState, so that their main advantages can be efficiently exported to the external unit test library of their choice, and users can test multiple times in this independent unit test.

So we will consider the practicability of this product based on the following points:

1. Is GenTest usable and efficient?

2. Does GenTest software provide a user interface in the form of a command line interface?

3. Is the description of the use process clear in the documentation?

4. Have independent unit tests been generated in the external unit test library selected by the user after running?

5. Does GenTest support nested, non-nested structures and all low-level types (including strings)?

6. If the path of the harness file entered by the user is incorrect or does not exist, does GenTest return information to inform the user of the error?

7. Does the output of GenTest contain any references to DeepState?

8. Is the output of the independent file generated in GenTest basically the same as that in DeepState?

Why do we choose these checkpoints?

1) While DeepState is very useful for producing tests, there is some complexity and overhead involved, so many projects will prefer not to depend on such a complex system for their core unit testing. That's why GenTest was born. GenTest should sample enough

and save time for users. So first of all, we must pay attention to the availability and effectiveness of GenTest.

2) Command line interface is a very familiar operation interface for computer filed workers, and it will greatly improve work efficiency after familiarity with command statements. Moreover, DeepState is also a framework that operates through command-line interface and command statements, and the user interface provided by GenTest in the form of command-line interface will retain the user's usage habits. So this is also an important monitoring point of usability.

3) Because most users use GenTest for the first time, the description of the usage of GenTest in documentation is particularly important. A good document should be concise, clear, and have sufficient introduction to the whole project and each step. This can increase the user's good feeling when using.

4) The ultimate goal of GenTest is to create independent unit tests that allow users to do unit tests repeatedly. Therefore, it is very important to detect whether independent unit tests are generated in the external unit test library selected by users after running.

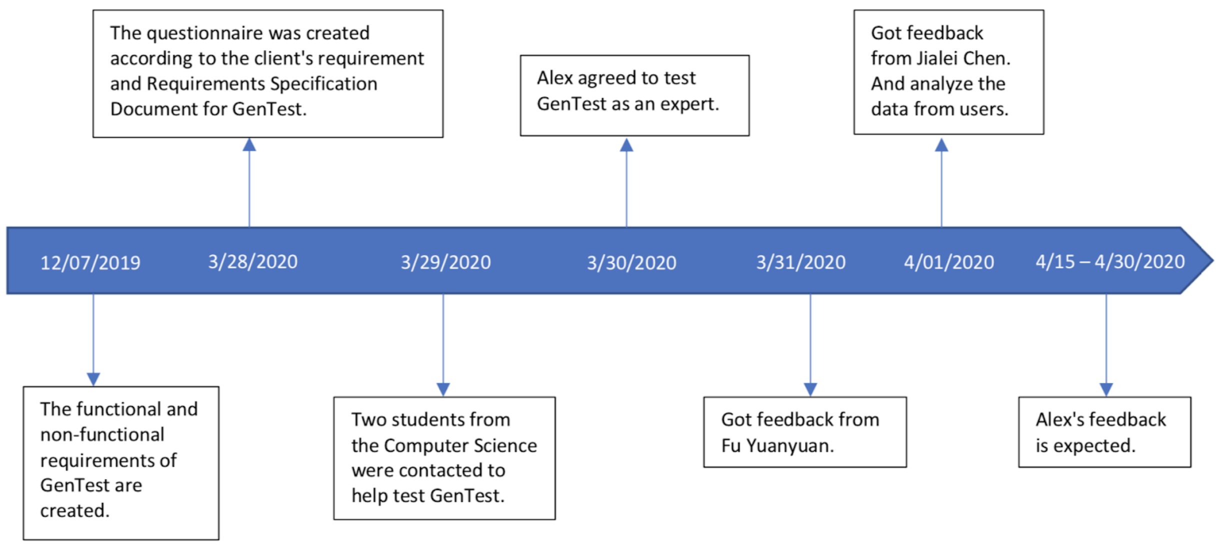
5) Because of the project requirements, our client requires that GenTest can support all the low-level types (including strings) that DeepState can generate, and create simple non recursive, possibly nested structures and codes. And as a unit test generation framework, these basic conditions should be met. So this is also one of the test points.

6) Because the user uses the command line interface to input commands, sometimes the input path is wrong or does not exist. GenTest should be able to detect error conditions and feedback to users. This kind of software is more humanized.

7) Because the client asked us that the output of GenTest should not contain any reference to DeepState, this is also one of the qualification points.

8) Finally, we need to ensure the stability and accuracy of our products, so it is important to check whether the output of the independent file generated in GenTest is basically consistent with the output in DeepState.

Timeline


The questionnaire was created according to the client's requirement and Requirements Specification Document for GenTest.
Alex agreed to test GenTest as an expert.
Got feedback from Jialei Chen. And analyze the data from users.
12/07/2019
3/28/2020
3/29/2020
3/30/2020
3/31/2020
4/01/2020
4/15 – 4/30/2020
The functional and non-functional requirements of GenTest are created.
Two students from the Computer Science were contacted to help test GenTest.
Got feedback from Fu Yuanyuan.
Alex's feedback is expected.

General Usability Test

In order to get more convenient information, we made a questionnaire to collect the user's experience. We get user samples from DeepState and general computer science students to test the product. Then the collected information is organized into a table. Here are some descriptions of table information:

1. Is GenTest available and efficient?

   We set 100 levels (from 0 to 100). 0 means very difficult to use and not efficient at all. 100 means it's very easy to use and extremely efficient.

2. Does GenTest software provide a user interface in the form of a command line interface?

   YES or NO

3. Is the description of the use process clear in the documentation?
   YES or NO

4. Have independent unit tests been generated in the external unit test library selected by the user after running?

   YES or NO

5. Does GenTest support nested, non-nested structures and all low-level types (including strings)?

   YES or NO

6. If the path of the harness file entered by the user is incorrect or does not exist, does GenTest return information to inform the user of the error?

   Always Usually Sometimes Rarely Never

7. Does the output of GenTest contain any reference to DeepState?

   YES or NO

8. Is the output result of the independent file generated in GenTest basically consistent with that in DeepState?

   Always Usually Sometimes Rarely Never

9. How satisfied are you with the look and feel of GenTest?

   Extremely satisfied
   Very satisfied
   Somewhat satisfied
   Not so satisfied
   Not at all satisfied

10. Do you have any thoughts on how to improve this software

Name: Alex David Groce	Job: Associate Professor
Is GenTest usable and efficient?	Waiting for reply
Does GenTest software provide a user interface in the form of command line interface?	Waiting for reply
Is the description of the use process clear in the documentation?	Waiting for reply
Have independent unit tests been generated in the external unit test library selected by the user after running?	Waiting for reply

Does GenTest support nested, non-nested structures and all low-level types (including strings)?	Waiting for reply
If the path of the harness file entered by the user is incorrect or does not exist, does GenTest return information to inform the user of the error?	Waiting for reply
Does the output of GenTest contain any reference to DeepState?	Waiting for reply
Is the output result of the independent file generated in GenTest basically consistent with that in DeepState?	Waiting for reply
How satisfied are you with the look and feel of GenTest?	Waiting for reply

Name: Yuanyuan Fu	Job: Computer Science Student
Is GenTest usable and efficient?	90
Does GenTest software provide a user interface in the form of command line interface?	YES
Is the description of the use process clear in the documentation?	YES
Have independent unit tests been generated in the external unit test library selected by the user after running?	YES
Does GenTest support nested, non-nested structures and all low-level types (including strings)?	YES(most of them)

If the path of the harness file entered by the user is incorrect or does not exist, does GenTest return information to inform the user of the error?	Always
Does the output of GenTest contain any reference to DeepState?	NO
Is the output result of the independent file generated in GenTest basically consistent with that in DeepState?	Always
How satisfied are you with the look and feel of GenTest?	Very satisfied

Name: Jialei Chen	Job: Computer Science Student
Is GenTest usable and efficient?	98
Does GenTest software provide a user interface in the form of command line interface?	YES
Is the description of the use process clear in the documentation?	YES
Have independent unit tests been generated in the external unit test library selected by the user after running?	YES
Does GenTest support nested, non-nested structures and all low-level types (including strings)?	YES(not sure)
If the path of the harness file entered by the user is incorrect or does not	Always

exist, does GenTest return information to inform the user of the error?	
Does the output of GenTest contain any reference to DeepState?	NO
Is the output result of the independent file generated in GenTest basically consistent with that in DeepState?	Always
How satisfied are you with the look and feel of GenTest?	Extremely satisfied

“Out-loud” Testing

We sent the questionnaire to the people who were interviewed and got some feedback from them. Here are the details of the feedback:

Yuanyuan Fu： I think this project you are working on is rather difficult to understand. And as a student, I can’t really touch this software. When I usually do my homework or work on a project, if the compiler's compilation and operation are passed, it is deemed as correct and successful. I never thought about doing unit testing. However, I do think that this should be quite important for some large projects. Also, I think this command-line interface is hard to use, and also, I didn’t learn very well in the command line statement. However, you got very detailed documentation, which directed me on a smooth installation and operation, though I didn't know what I was doing during the whole process. The only problem was, I installed Python, but when I was installing GenTest, the software suggested that the Python.h file could not be detected. It is my suggestion that you can develop a graphical interface, which could make us a better coding environment.

Jialei Chen：The software you made is very useful, although it is beyond my knowledge. The running speed of the software is normal. I'm not sure if GenTest can run nested structures, but I'm sure it can run non-nested structures, so I chose yes in question 5. I think the command line interface is very simple and convenient to use. It's hard to imagine how you can make this part so perfect. The documentation is also very clear. I downloaded and installed the software smoothly according to the process. Because I don't know the software you've made, I can't provide any professional advice. But I think you're really great.

Feedback Analysis

Due to the limited knowledge of testers, our testers are not familiar with unit testing so far. And they didn't use DeepState before they tested GenTest. So it's normal for them to feel strange and incomprehensible. Through communication with them, we know that it took them a long time to complete the whole processes including download, installation and testing (including the DeepState test), and the testers basically spent 1-2 days (should not be continuous work). But they feedback that the running speed of GenTest is very good. The biggest difficulty in the test is their understanding of the software. From the messages they replied, we can know that GenTest has met the basic functional and non-functional requirements. Of course, for some test points, their feedback is very vague, such as whether GenTest can handle nested structures, and whether GenTest can handle all basic types. I think this part still needs to wait for the test results from experts(Alex David Groce). We will continue to improve the function of GenTest and provide users with better experience in the future.

Conclusion

After the reading, users can clearly know that we divided this project into three parts. And this process is divided into two stages to complete the creation and integration of these three parts. Before the creation of the three components, the client has a detailed description of our requirements for each part. And in the completion process of GenTest, the core engineers of our group strictly control every detail of the code, communicate with the client in time to avoid problems, try to achieve full coverage of the functions required by the customers, keep the code neat and output correctly. And in each module, basically every important function has a separate test file. The module of File Assembler has nearly 100% coverage. This can largely avoid program errors and save the time needed to maintain the software later. We are confident that our outlined testing procedures will enable the delivery of a functional, fault-tolerant piece of software to our client.

We are trying to optimize our products, and hope that through our efforts, DeepState can help more people. We hope that accidents like the one in Southern California will not happen again, and we can embrace a world surrounded by safe and reliable software in the future.

References

[1] Groce, Alex, and Peter Goodman. “DeepState: Symbolic Unit Testing for C and C++.” *Github*, DeepState Repository, 2018, agroce.github.io/bar18.pdf.

[2] Evans, Scott. “The Toyota Recall Crisis - A Chronology of the Toyota Pedal, Floormat Recall - Motor Trend.” *Motor Trend*, Motortrend, 27 Jan. 2010, www.motortrend.com/news/toyota-recall-crisis/.