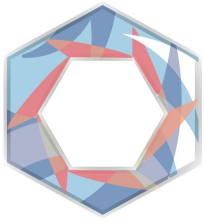
GenTest
December 7, 2019



Requirements Specification Document

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

Client: _____ **Project Team Lead:** _____

Revision History

Date	Person	Change
11/6/19	Joshua Johnson	Document created and the introduction/problem statement was added.
11/10/19	Joshua Johnson	Solution vision and functional requirements added to document.
11/11/19	Silu Shen	Environmental requirements added to document.
11/11/19	Joshua Johnson	Project plan added to document.
11/13/19	Tristan Miller	Project risks added to document.
11/13/19	Silu Shen	Environmental requirements edited to equipment and language requirements.
11/14/19	Zane Fink	Non-functional requirements added to the document.
11/14/19	Joshua Johnson	Functional requirements edited based on suggestions from Mentor. Removed table of functional requirements and slimmed functional requirements three core.
11/16/19	Silu Shen	Environmental requirements edited to include external library use requirement.
11/17/19	Tristan Miller	Conclusion added to document.
12/2/19	Tristan Miller	Added table for risks, their severity, and probability to the potential risks section.
12/2/19	Joshua Johnson	Modified project plan with new Gauntt chart including project demo, expanded upon project demo explanation, added table of example tests to project plan testing and verification, and modified process DeepState impact sub-section.
12/4/19	Joshua Johnson	Added command-line interface requirement to environmental requirements section and fixed formatting.
12/5/19	Joshua Johnson	Modified process DeepState impact and functional reqs.

Table of Contents

Section 1: Introduction

1.1 Project Description

Efficiently creating quality software is a major hurdle for a wide-range of commercial industries. In 2014, the Standish Chaos Report [1] measured the statistical success of software projects and found that only a mere 16.2% of all software was completed on-time and on-budget. Similarly, delayed and canceled software cost companies and the government a combined total of $150 billion dollars [1] in 1995. While development strategies have evolved since[4], [5] , software success rates remain a great challenge for the software industry that loses billions. To increase project success, companies need more effective strategies for quality control and project development [3], [4], [5].

TDD (Test-Driven Development) is one such method which has seen success since 2009 [2] and has led to an increased adoption of unit testing by major software companies [2] . The impact of unit testing on project performance is related to code defects. According to research by Software Productivity LLC, most successful projects have around 4 defects per function [3], while unsuccessful projects typically have 7 or more defects [3]. From this research, it follows that the reduction of defects in a system directly correlates to an increase in project success. Studies conducted at Microsoft found that unit testing reduces defects in a system from 62% to 91% using automatic techniques and 20.9% using normal [2] . Thus, unit testing can be connected with an increase in the probability of project success.

Despite the increased performance automatic unit testing can provide over its normal counterpart, developers typically do not understand how to use automatic testing techniques such as fuzzing and symbolic execution [6] . Associate Professor Alex Groce at Northern Arizona University and cybersecurity company Trail of Bits have created DeepState to fix this problem. DeepState is a unit testing framework which acts as an easy-to-use interface providing access to fuzzing and symbolic execution engines [7]. Currently, DeepState provides the following advantages and more for Developers wanting to create unit tests:

- Tests look like Google Test, but can use fuzzing/symbolic execution to generate data (parameterized unit testing)
- Supports Manticore, Angr, libFuzzer, file-based fuzzing with e.g., AFL or Eclipser
- Provides high-level strategies for improving symbolic execution/fuzzing [7]

1.2 Problem Statement

Understanding the problems DeepState currently has requires an understanding of the DeepState workflow. To execute DeepState, there are two primary concerns:

- *Harness Files*
- *Binary Tests*

The harness file contains the test and evaluation functions for a unit test just like in other frameworks such as GoogleTest or JUnit [7]. In the DeepState framework, the word “test” refers to the randomly generated binary values that are placed in the harness file at DeepState run-time. “Test Cases” refers to the enumeration of these binaries in files or in memory. Running the DeepState framework requires these two concerns to be met either by specifying the binary test case to use in the command-line or by using built-in Fuzzers to generate the binary values at execution.

Take for example the process modeled in *Figure 0*. To use the DeepState framework, the developer must first link the DeepState.hpp file to the test harness. At compile time, the harness file is linked with the DeepState library, which references the executable Test.o file in the figure to output binary test cases in thousands of binary files with pass/fail conditions using fuzzing engines.

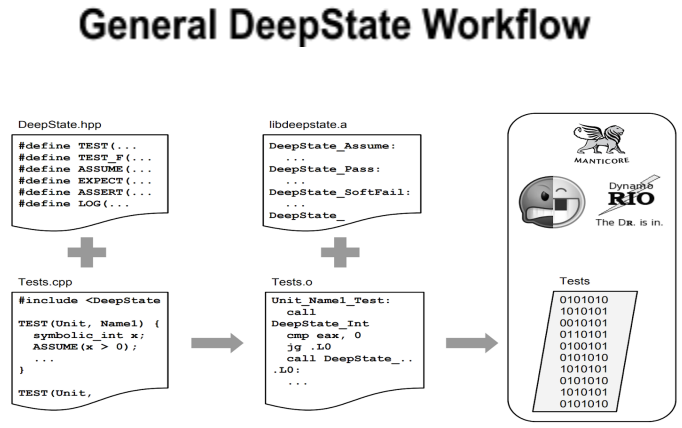


Figure 0 - The generalized workflow of the DeepState framework.

Given the above process, most developers have difficulty transitioning from their current framework to the DeepState tool. Due to the need for testing harnesses native to the DeepState framework, the integration of DeepState into large-scale projects is limiting for existing technologies because:

- Projects may already have previously unit testing frameworks (such as GoogleTest) which contain unit tests that would need to be rewritten.
- The cost of installing the DeepState framework across a company’s systems may be inhibiting.
- Maintaining two testing frameworks is painful and takes more resources.

1.3 Solution Vision

To solve these problems, GenTest has been contracted to create a system that can provide developers access to the DeepState framework without incorporating DeepState itself into the project. Our solution envisions a pseudo-compiler which will read test cases defined in the DeepState framework and translate them to a standalone unit test file in an external framework of the developer's choice. The system will provide the following features for the developer:

- *Use of Fuzzers and Symbolic Execution without including any DeepState library*
- *Creation of compilable, executable, standalone tests in an external-unit testing framework*
- *Support for DeepState test case execution capture (i.e. in what order test cases are called) using the oneOf operator.*

There are three major components to the GenTest solution that make this happen: **the Translation Engine, the File Assembler,** and **the Binary Parser**.

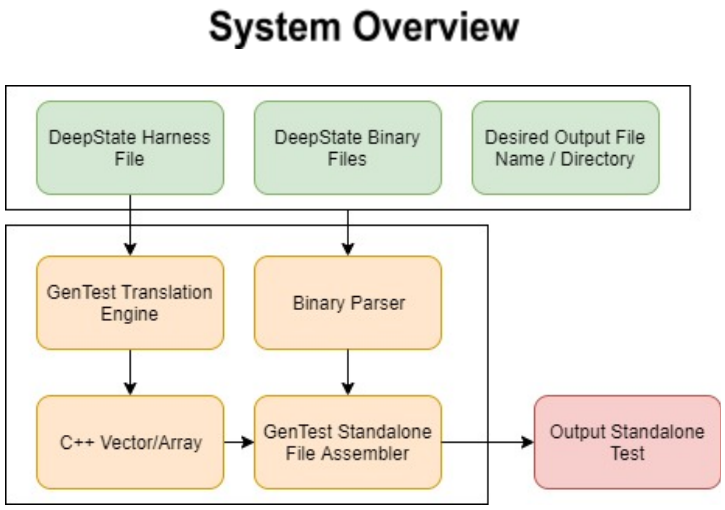


Figure 1 - An abstract overview of the GenTest solution.

The use of these components requires that several inputs are provided to the command-line before execution. These inputs are depicted by the green boxes in *Figure 1* and are:

- The harness file to translate.

- An output file name or directory path

Optionally, developers may also provide binary files as an input.

1.3.1 How are the Inputs Used?

Inputs provided by the developer to the GenTest software are used to translate test cases to another framework in a standalone unit testing file. There are three major computational processes which take the supplied harness file from its native representation to the external:

- Translation
- Binary Parsing
- File Assembly

Translation is the process of taking a harness file native to DeepState, parsing it, and placing an abstract representation of the code sequentially in a C++ vector. Referring back to *Figure 1*, this process is encapsulated in the component (all orange boxes are system modules) labeled “Translation Engine”. This vector is then passed to the File Assembly computation stage. At this point, the File Assembler depicted to the right of the vector component invokes calls from the Binary Parser to populate each test with data to a specified number of test cases or until a test is found that fails. Finally, as the File Assembler conjoins the abstract representation of the harness file with randomly generated values, it uses grammar production rules to generate a new standalone unit test at every iteration.

1.3.2 How will this process impact DeepState?

The main goal of GenTest is to translate harness files correctly using the same workflow currently used by DeepState. While the GenTest product strives to impact this workflow as minimally as possible, there is a possibility that harness translation times may increase the execution time of the DeepState software. When the GenTest software is invoked via its optional command-line arguments, the used DeepState test must be translated into the target framework before it is executed by DeepState. A consequence of this workflow is that the longer a harness file is, the more time it will take to translate it into the target external framework. For instance, if a DeepState harness file contains fifteen test functions, the parsing and translation of this file will take longer than a file containing only one as a result of the translation module requiring more parsing and

translation actions. This creates longer execution times for the GenTest product as a whole without optimizations.

1.4 Conclusion

In this section, the DeepState framework, its problems, and the GenTest solution were discussed. While DeepState is already a valuable resource for any developer, there are some ways it can be more accessible to software projects. GenTest hopes to create a solution which despite small performance trade-offs will provide a more accessible way for developers to make use of DeepState. Next in the document, the requirements of the system will be explained in more detail.

Section 2: Functional Requirements

To support the solution vision detailed in the previous section, there are several functional requirements that must be discussed. These requirements were gathered through talking with the client, analyzing the project description, and exploring the DeepState GitHub and Slack channel.This section provides an overview of the main details which constitute a finished product for the client.

2.1 Requirements Terminology

The following subsections will explain these functional requirements in more detail. Within each requirement description statement, the following language is used to further demonstrate priority:

Shall *- Vital for project success.*
Must - *Important but not the foremost priority*
Will - *Would be nice to have, but is not vital.*

2.1 Jargon and Project Terminology

Project-specific terminology is used within the functional requirements to express domain related needs. Each of these terms is defined in this section to further clarify the expectations of each requirement with respect to the project domain.

- **Symbolic Execution** - A process which analyzes a program to determine its normal inputs and interpret them as an abstract symbolic value. This symbolic value, λ (lambda), can be thought of as a variable (like in mathematics) which can assume any value. λ is used by a language interpreter in-place of the actual value it represents in computations and calculations. DeepState has a built in Symbolic Execution engine which is used to assume symbolic values in test files rather than expressing the actual values of variables at compile-time.

- **Fuzzers** - Tools which produce automatically generated invalid, unexpected, or random data as inputs to a program. DeepState includes multiple fuzzing engines which can be used by developers to automatically generated data values for test-cases written in DeepState test files.

- **Test Replay -** Test replay refers to the process of capturing, storing, or monitoring the automatically generated values by DeepState built-in fuzzers (see above) for later encoding into translated standalone test files in external C++ testing frameworks (i.e. GoogleTest, Catch2).

2.3 Domain Level Functional Requirements

The domain level functional requirements for GenTest are listed below:

1. The GenTest solution ***shall*** translate DeepState test files into standalone files in an external framework.
2. The GenTest solution ***shall*** replay the generated data by DeepState fuzzing and symbolic execution.
3. The GenTest solution ***shall*** translate nested and non-nested structures in the standalone output file.

Each domain level functional requirement is broken down further in the following section.

2.4 Expansion of Domain-Level Functional Requirements

1. The GenTest solution *shall* translate DeepState test files into standalone files in an external framework.

    1.1. The user shall be able to provide the GenTest solution with a DeepState test file.
    1.2. The GenTest software shall accept a supported framework from the user to translate into.
    1.3. The user shall be able to run the GenTest solution through the standard DeepState Command-Line-Interface.
    1.4. The user shall be able to specify to the GenTest solution an output file name or directory.
    1.5. The GenTest software shall display for the user a list of received inputs at the start of execution. This includes the following information:
        1.5.1. Chosen test harness
        1.5.2. Selected output file or directory
        1.5.3. Selected output framework

1.6. The GenTest software shall provide appropriate error messages for potentially common mistakes.
    1.6.1. It shall inform the user if path to the harness file is incorrect.
    1.6.2. It shall inform the user if the path to the harness file was not specified.
1.7. The GenTest solution shall output standalone files that are compiliable by the compiler used by the target framework.
1.8. The GenTest solution shall translate all DeepState syntax in the output file to the syntax of the target external framework.
1.9. The output of the GenTest solution shall not contain any reference to the DeepState framework.
1.10. The test results of the GenTest solution shall as much as possible match the results of the DeepState test.

2. The GenTest solution shall replay the generated data values by fuzzing and symbolic execution.
    2.1. It shall capture values generated by test replay.
    2.2. It shall encode these values in the translated file.
    2.3. It shall place all generated values sequentially in the order they are produced by DeepState.
    2.4. It shall support the following basic primitives:

        2.4.1. Char, String
        2.4.2. Short
        2.4.3. Int
        2.4.4. Unsigned
        2.4.5. Long
        2.4.6. Int8_t
        2.4.7. Uint8_t
        2.4.8. Int16_t
        2.4.9. Uint16_t
        2.4.10. Int32_t
        2.4.11. Uint32_t
        2.4.12. Int64_t
        2.4.13. Uint64_t
        2.4.14. Double, Floats

3. The GenTest solution shall translate nested and non-nested structures in the standalone output file.

3.1. The GenTest solution shall be able to record and encode automatically generated values into nested or non-nested structures.

Section 3: Performance Requirements

With the functional requirements outlined in Section 2, a viable solution will also meet several performance requirements. These requirements outline what is necessary for the creation of a usable product. In the sections that follow, we discuss the impact of two different performance requirements:

- Usability
- Functional Correctness

Section 3.1 Usability

The solution developed by GenTest will take the form of a command-line interface accessible through the DeepState software. Users of DeepState will invoke GenTest's software with the inclusion of an additional argument to DeepState. Further arguments will be provided to GenTest's software through the command-line. To avoid frustrating new users, no more than 5 command-line arguments should be required to run GenTest's software. Each argument's meaning will be outlined in both the help message shown when the '--help' flag is included and in the software's user manual. Additionally, a user familiar with DeepState usage should be able to start GenTest's software within 3 minutes of their first encounter with the software; while subsequent encounters should require no more than 1 minute to start the software.

Section 3.2 Functional Correctness

Any mistranslated tests may result in the false identification of bugs or may cause critical bugs to be missed. The misidentification or non-identification of bugs may result in errors in critical software, with the potential for disastrous ramifications. Therefore, the software produced by GenTest must be able to correctly translate tests with supported features 100% of the time. To ensure the the GenTest solution is functionally correct and does not propagate these errors, the GenTest team will perform comparison testing, where we take a sample of three different test files native to DeepState, translate them, and compare the results between a sample of the original DeepState pass/fail outputs and those generated by the standalone file.

Section 4: Environmental Requirements

Every project has environmental requirements which limit the project in some way. The following requirements detail the environmental constraints placed on the GenTest team when developing their solution for the DeepSate problem. In order to meet the expectations of users, Gen test must meet the following development conditions:

1) **Equipment Requirements**
2) **Programing Language Requirements**
3) **External Library Use**

Section 4.1 Equipment Requirements

Currently, DeepState only supports the MacOS and Linux operating systems. The GenTest solution must be able to provide cross-platform support for these operating systems as to provide developers and users with the maximum availability possible. This does not extend to the Windows operating system. This means creating a system that can run on both operating systems with the DeepState framework.

Section 4.2 Programming Language Requirements

The GenTest solution will deal with C/C++ code and translate this code into C/C++ code with the syntax of external frameworks. It is expected that the scope of external frameworks is limited to the external frameworks which support this programming language requirement.

Section 4.3 External Library Use

No external libraries will be used by GenTest except for the STL in C++. This requirement has been placed upon GenTest to prevent dependencies which may later stop working and cause the system to be inoperable.

Section 4.4 Command-Line-Interface

The GenTest software must provide an user-interface in the form of a Command-Line-Interface (CLI). Said interface must be integrated with the pre-existing DeepState CLI as optional command parameters for use by developers.

Law is an important way to protect the intellectual property of software development. Before entering the production cycle of the project, we must have sufficient legal awareness and discuss the details of cooperation with customers to avoid legal disputes on the way of cooperation or delivery of finished products.

- In the process of cooperation with users, Team GenTest should fully communicate and implement the complete information of the project. Information content includes: project name, content, description, development function, and customer requirements for the project.

- When the product is delivered after the development, the developer shall make a good agreement with the client on the ownership of the intellectual property. The party who obtains the intellectual property shall not disclose the technical product, technical data, documents, etc. to the third party without the other party's knowledge or consent.

- During the cooperation, all documents of both parties shall be kept confidential and shall not be disclosed to a third party without the other party's knowledge.

- If the project needs to be maintained and upgraded by the original development team (Gen test) after the completion of the development, this requirement should be stated before the formal development of the software and recognized by both parties.

- GenTest should post the free source code on GitHub. However, any third party may not use the project resources for commercial purposes without the approval of users and developers.

Section 5: Potential Risks

With every project comes inherent risks, and GenTest is no different. This section will cover not only the risks involved, but the mediation tactics we will use to prevent them from being an issue. Some of the risks involved with this project are as follows:

- Tests generated by GenTest fail to compile.
- Test output from GenTest differs from that of DeepState
- Incomplete or sparse support for generic data types

The Table below contains the Probability and severity of each of these issues

Risk	Probability	Severity
Failure To Compile		
Semantic Inequivalence		
Unsupported Data Types and Structs		

Section 5.1 Failure to Compile

On runtime, GenTest will be generating C++ testing code. One main issue with this approach is that generation of the C++ testing file could fail to compile. If our project has issues during compile time, developers looking to utilize GenTest in their development workflow could grow frustrated, and are looking for a tool that just works. In GenTests case, deconstruction of the original test file will be done using a context-free grammar or CFG. This context-free grammar will allow for thorough construction of the C++ test file from scratch, which will prove useful as failure to compile the code will mean that there is an error in the CFG. use of this CFG will make debugging this issue easier, as opposed to converting a file and inserting values into the converted test, and allowing team GenTest to mitigate this risk.

Section 5.2 Output Differs from Original Tests

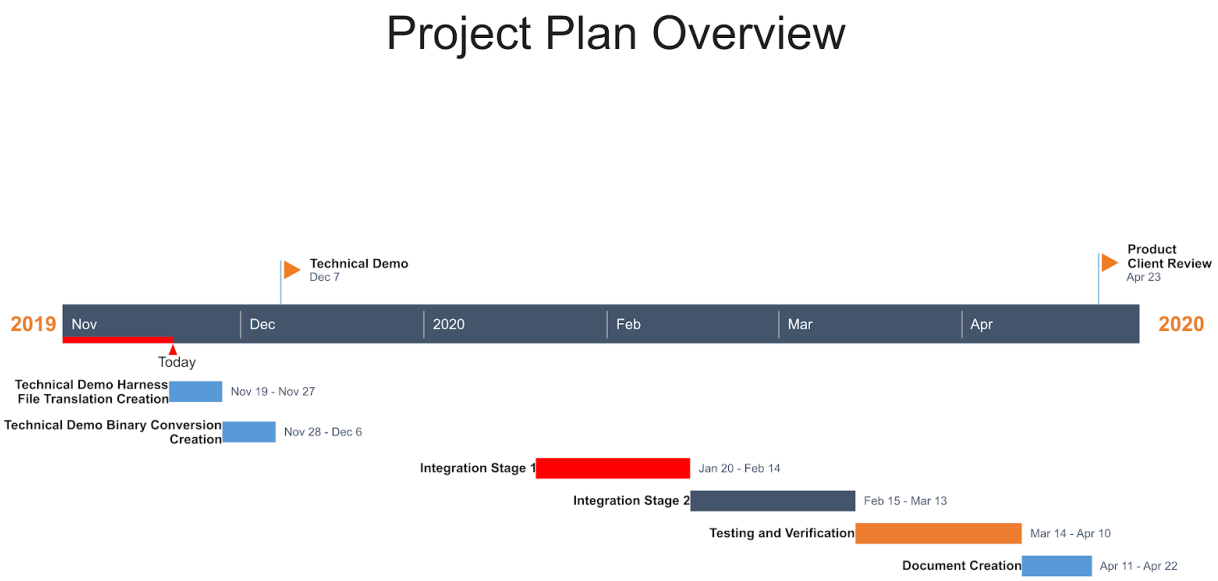
Another risk GenTest faces is invalid tests. If a test is parsed incorrectly, the output from the test could differ from that of the output from the DeepState test. With a differing test outcome, developers using DeepState could potentially spend additional time attempting to debug a bug that is not even created by their code. This could lead developers to not utilize GenTest in their developmental flow as GenTest could create more problems than it solves itself. The GenTest team plans to prevent this using extensive testing with a wide variety of example testing harnesses. Each different test harness will test an individual implementation or multiple implementations described in section 2. With extensive testing, the GenTest team will be able to work out potential issues inserting complex values and functions, as well as mitigate the risks involving

Section 5.3 Incomplete or Sparse Support

Due to the nature of the binary values generated by DeepState, utilizing these values could prove to be difficult, as GenTest will receive unstructured binary data. The goal at team GenTest is to implement all the types brought forth in Section 2 Subsection 9 Requirement ID F5. In order to implement these generic data types, GenTest will pass the unstructured binary data to DeepState, and observe and extract the created generic types. Another potential issue regarding support is that of nested structures, and pointers to these nested structures. Due to the nature of nested structures, GenTest will need to understand how to break down and rebuild these structures, while also utilizing the concrete data types inserted at runtime. GenTest plans to mitigate these risks by being explicit in documentation with what is currently being supported. If GenTest fails to support these features, the usage of GenTest would fail to reach the goal of widespread use, as the tests that GenTest can support would be required to be basic in nature.

Section 6: Project Plan

Before now, most of the work relating to the project has been around documentation and the gathering of requirements. Now, the project is beginning to head towards the implementation of the software product for our customer. *Figure 3* is a Gantt chart representing the phases that are planned for the future development of the GenTest product. A brief summary of each from the today marker will be provided below.



Section 6.1 Technical Demo

The technical demo milestone includes creating a prototype for the GenTest solution as a proof-of-concept exercise. For this milestone, GenTest plans to create two independent modules: the Binary Parser and Translation module. GenTest will demonstrate that these two modules can work independently and together to achieve file export and basic value recording.

There are two main demos that team GenTest will present to prove feasibility for system design: **Harness File Translation** and **Translation with Binary Conversion**.

The first demo will, *Harness File Translation*, will use the Translation module to demonstrate GenTest's ability to translate DeepState harness files into a standalone file

in an external framework. GenTest will prove this capability by translating three example DeepState test harness files into standalone Google Test files. The translated harness files will include one of the following basic primitives: integer and character. For this demo, the values propagated in the Google Test standalone files will be hard-coded.

The second demo is *Translation with Binary Conversion*. GenTest will prove its ability to convert binary values by showing the translation process outlined in the *Harness File Translation* demo using values pulled from binary files. All values propagated in the standalone Google Test file will not be hard-coded in this demo.

Section 6.2 Integration Stage 1

Once the next semester starts, GenTest will proceed onto integration stage 1. During this phase, we will take these two basic prototypes discussed above and incorporate them into one general system that can produce output standalone tests. During stage 1, it is expected that most primitives will be supported, but nested and non-nested structures will not be supported by system execution.

Section 6.3 Integration Stage 2

During this stage, the GenTest team will take the integrated systems from Stage 1 and add nested-structure support to the translation and value generation stages. This will be the last step in the integration stage for the GenTest system before the overall system is completed.

Section 6.3 Testing and Verification

During this stage, GenTest will perform several unit, integration, and requirements tests to ensure that requirements have been met by the overall solution and to prevent potential risks identified in this document. During this phase, unit, integration, and requirements testing will be performed to verify that:

- The system does not encounter potential risks cases such as failing to compile or having different outputs.
- Non-functional requirements are met.
- GenTest operates as expected and outlined in the functional requirements.

To ensure that potential risk cases do not happen, a series of tests will be performed to compare the outputs of DeepState and translated tests. Similarly, translation will be run on every example DeepState test harness to prove that translation will provide a compilable output for multiple types of harnesses. For non-functional testing, the team will introduce the GenTest solution to users of DeepState to ensure that using the solution is fast and easy to learn. A sample set of the tests that will be run by GenTest for product verification can be found in the table below.

Table of Sample Tests for the GenTest Product

Test Name	Test Description	Test Type
Symbolic Primitive Translation Test	The Symbolic Primitive Translation test verifies the Translation module's ability to translate symbolic variables of primitive type as specified in the functional requirements into the proper primitive type declaration in the standalone external framework. An appropriate sample size of the example DeepState harness files will be run using the GenTest product 15 times and analyzed for correct compilation and expected output.	**Unit**
Compilation Test	To verify that the GenTest software can correctly translate DeepState harness files, an appropriate sample size of the example DeepState harness files will be selected, run through the GenTest software using multiple target frameworks, and compiled to ensure correct syntax generation.	**Integration**
Value Generation Test	To ensure that the correct values are propagated in standalone file by the GenTest software, an appropriate sample size of example DeepState harness files consisting of most required primitives will be run through the GenTest software. The values propagated in the standalone files will be compared against those generated by	**Integration**

	DeepState to verify correct replay of generated values.	
Usability Test	To verify that the DeepState system is usable by its target users, two to three DeepState users will be enlisted to use the GenTest software via the DeepState command-line interface. For each user, the start-up time will be recorded and compared with the specifications outlined in the non-functional requirements to ensure start-up times meet required targets.	**Requirements**

Section 6.4 Documentation

Once the system has been confirmed to work as expected, the GenTest team will focus on documenting how to use the system to meet non-functional requirements. The team will write documentation for GenTest on the GenTest GitHub repository.

Section 6.5 Product Client Review

At this stage, all core functionality for the GenTest solution will be finished, tested, and complete. The GenTest team will hold a product review with the client to verify that it meets acceptance criteria and it is to the client’s expectations.

Conclusion

Currently, integrating the DeepState software with pre-established products is difficult due to migration and integration costs for software teams. GenTest plans to develop a thorough solution that will allow for standalone test generation to mitigate these issues and allow for a more accessible way to utilize DeepState. The features team GenTest Plan to implement are as follows:

- Test File Translation
- Standalone Unit Test Generation
- Harness File Translation
- Binary Value Insertion
- Low-Level Primitive Support
- Nested Structure Support
- Capture and Response of DeepState Decisions
- C compatibility

Using these requirements, GenTest plans to allow DeepState to generate standalone unit tests for C and C++ while also utilizing an external testing library. Using a mix of python and C++ will allow GenTest to thoroughly break down and rebuild the DeepState test files, as well as the Harness files. The requirements and time frame laid out in this document provides concrete goals and milestones for the team as well as laid out an estimated development cycle timeframe.

References

[1] “The Standish Group Chaos Report.” *Project Smart*, 2014, www.projectsmart.co.uk/about-us.php.

[2] Williams, Laurie, et al. “On the Effectiveness of Unit Test Automation at Microsoft.” *2009 20th International Symposium on Software Reliability Engineering*, 2009, doi:10.1109/issre.2009.32.

[3] Jones, Capers. “Software Project Management Practices: Failure Versus Success.” *Academia.com*, 2001.

[4] Lee, Gwanhoo, et al. “Ambidextrous Coping Strategies in Globally Distributed Software Development Projects.” *Communications of the ACM*, vol. 49, no. 10, 2006, p. 35., doi:10.1145/1164394.1164417.

[5] Subramanian, Girish H., et al. “Software Quality and IS Project Performance Improvements from Software Development Process Maturity and IS Implementation Strategies.” *Journal of Systems and Software*, vol. 80, no. 4, 2007, pp. 616–627., doi:10.1016/j.jss.2006.06.014.

[6] M. Zalewski, “american fuzzy lop (2.35b),” http://lcamtuf.coredump.cx/afl/, November 2014.

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