GenTest
February 7th, 2020

Software Design Document

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

Table of Contents

Section 1: Introduction

1.1 Project Description

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 [8] 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 [8] 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. As the role of software has increased in the products and services that are used, so too has the dangers of software bugs. While many of the bugs found in software have enormous consequences for companies and their revenue, more often than not their impact is not quantifiable. The responsibility of companies creating software in the modern world has transcended simply creating a usable product--it has become in many scenarios a matter of life or death.

If the scope of software has evolved to such a major extent, how do software companies create products efficiently, while still meeting the often intense and rigorous safety requirements for products? Even when safety requirements are not as strict, creating a bugless, well-rounded software product is an enormous issue for companies. Indeed, software success rates for companies have traditionally been low. 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], X[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].

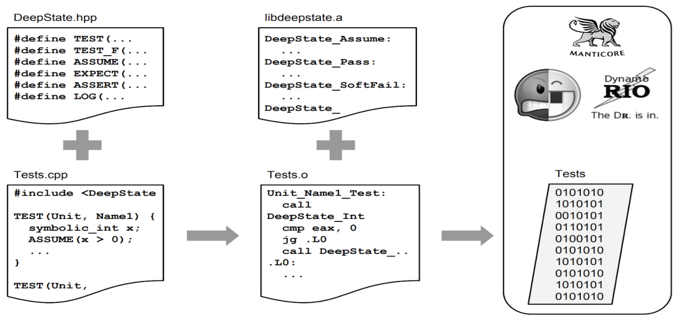
TDD (Test-Driven Development) is one such method which has seen success since 2009 [2] and has led to an increased adoption of unit testing by major software companies [2]. The impact of unit testing on project performance is related to code defects. According to research by Software Productivity LLC, 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 automatic unit testing techniques [7].

1.2 Problem Statement

Understanding the problems DeepState currently has requires an understanding of the DeepState workflow. 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.

General DeepState Workflow



DeepState.hpp
```
#define TEST(...
#define TEST_F(...
#define ASSUME(...
#define EXPECT(...
#define ASSERT(...
#define LOG(...
```

libdeepstate.a
```
DeepState_Assume:
  ...
DeepState_Pass:
  ...
DeepState_SoftFail:
  ...
DeepState_
```

Tests.cpp
```
#include <DeepState

TEST(Unit, Name1) {
  symbolic_int x;
  ASSUME(x > 0);
  ...
}

TEST(Unit,
```

Tests.o
```
Unit_Name1_Test:
  call
DeepState_Int
  cmp eax, 0
  jg .L0
  call DeepState_..
.L0:
  ...
```

Tests
```
0101010
1010101
0010101
0110101
0100101
0101010
1010101
0101010
1010101
0101010
```

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 and Requirements

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 is a pseudo-compiler which will read unit tests defined in the DeepState framework and translate them to a standalone unit test in an external framework of the developer's choice. For example, if a developer wrote a unit test in DeepState, they could use the GenTest software to translate it into the syntax of a widely used C++ testing framework, Google Test. To better understand the jargon of the project, please see Appendix A.

The solution implemented by GenTest must meet the following Domain-Level Functional Requirements:

1. *The GenTest solution shall translate DeepState unit tests into standalone unit tests in an external framework.*

2. *The GenTest solution shall replay the DeepState-generated test data produced by fuzzing and symbolic execution.*

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

In addition to these Domain-Level Functional Requirements, the solution must provide functional correctness and usability. **Functional correctness** refers to the ability of the GenTest solution to provide output unit tests which are functionally equivalent to its DeepState unit test representation. This means that if a developer writes a unit test in DeepState and translates it into, say, Google Test, they should reasonably expect the unit test to behave in a functionally equivalent way. **Usability** in this instance means that the provided solution is easy for DeepState users to adapt as defined by specific metrics. More on these metrics can be found in the *Requirements Specification Document*.

Guided by the above functional and performance requirements, the GenTest solution has some basic restrictions on what it can support. First, the GenTest software can only provide support for the MacOS and Linux operating systems. Second, it can only take C and C++ code for translation. Finally, the GenTest solution must take the form of a command-line interface. Incorporating these restrictions, GenTest is optimistic that its software product can provide a well-rounded solution for the client's problem.

Section 2: Implementation Overview

As previously mentioned, the GenTest product is a pseudo-compiler that is designed to tackle the main domain-level functional requirement of producing a compilable and executable external unit test from a DeepState specific test. In simpler terms, the purpose of the GenTest solution is to take one unit test in DeepState (using its library specific syntax) and translate it into a runnable unit test file in the language of another unit test library. "Pseudo-compiler" is used as a rough term to indicate that the GenTest software is not simply a compiler itself. As a whole, the GenTest solution provides the mechanisms, tools, and interpretations necessary to translate DeepState functionality into an external unit test. 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 the external unit testing library of their choice, efficiently. To achieve this goal, the software is divided into three concerns: interpretation, translation, and assembly.

Maps are used to relate the interpreted unit tests into translated values. The GenTest software uses the C++ Standard Library implementation of a map for these associations. For correct reference, the version of C++ used is C++ 11 or higher. More on the map data structure used by the interpretation approach can be found at: http://www.cplusplus.com/reference/map/map/. The map data structure provides the GenTest solution with an easily configurable, efficient manner of associating translation terms with translation literals. Similarly, the availability of maps means that the system can incorporate multiple maps relating to different C and C++ unit testing frameworks.

All other components and algorithms used by the GenTest software do not involve external libraries or will make use of DeepState functions. More on each of these concern areas will be described in the *Modules and Interface Descriptions* section.

Section 3: Architectural Overview

The *Architectural Overview* section is used to give detail on the overall system design. This section will introduce the components and subcomponents for the GenTest system. This section will contain two parts, The first giving a visual representation of the overall architecture, and the second part giving commentary regarding the architecture.

3.1 Architecture Design

This part will contain an overview of the overall system output, including a diagram with each individual part of GenTest, what it takes as inputs, and what it gives as outputs. The general layout of the GenTest system is laid out in *Figure 3*.

GenTest System Overview

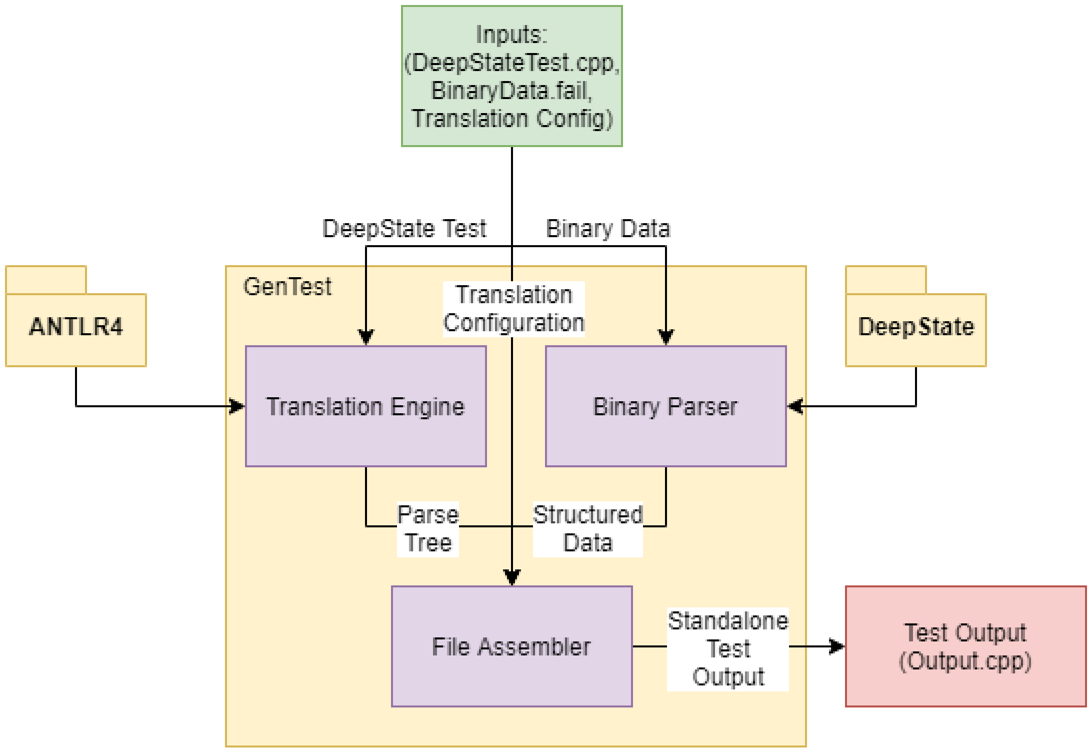


Figure 3 - GenTest System Architecture Overview.

3.2 Architecture Information

GenTest contains multiple components that come together to generate a standalone test utilizing DeepState values. This part will contain information on each individual GenTest system, laying out the key responsibilities and communication mechanisms for each individual part. Besides the use of ANTLR for the Translation Engine, this project is bare metal, using only default C++ standard template library for development. This section will give a broad understanding of the GenTest system, before going into more details in section 4.

3.2.1 Translation Engine

The Translation Engine's job is to deconstruct the DeepState test file. The Translation Engine will take in the file to be translated, then break it down into a parse tree utilizing ANTLR4. This parse tree will then be passed to the File Assembler for reassembly of the standalone test file.

3.2.3 Binary Parser

The Binary Parser's job is to make sense of the unstructured binary data generated by DeepState. The Binary Parser will take in an unstructured binary data file generated by DeepState and read the data contained in that file. The File Assembler will then utilize the Binary Parser to "fill in the blanks" where symbolic values once were.

3.2.3 File Assembler

The File Assembler's job is to reconstruct the file using the parse tree given by the Translation Engine, while also inserting values from the Binary Parser. This reconstructed file will then be outputted as a C++ file. The standalone file will be compilable and contain the generated values from DeepState.

Section 4: Module and Interface Descriptions

The *Module and Interface Descriptions* section is designed to provide further detail on each component listed in the software architecture. It provides additional information on the responsibilities, sub-components, connections, and interfaces which comprise each piece of the architecture. Additionally, the limitations of each will be explicitly stated as to provide a comprehensive overview of the capabilities and expectations of the components. As mentioned, there are three major concerns of the GenTest software: interpretation, translation, and assembly. In the following discussion, these concerns are related to the components of the system architecture in the following way: interpretation is provided by the Translation Engine, and translation and assembly are provided by the File Assembler and Binary Parser. While other components are necessary for the system, these components provide the basis for most system functionality.

4.1 Translation Engine

The responsibility of the Translation Engine is to provide interpretation for the GenTest system. When the word interpretation is used, we are generally referring to the analysis

Translation Engine Class Diagram

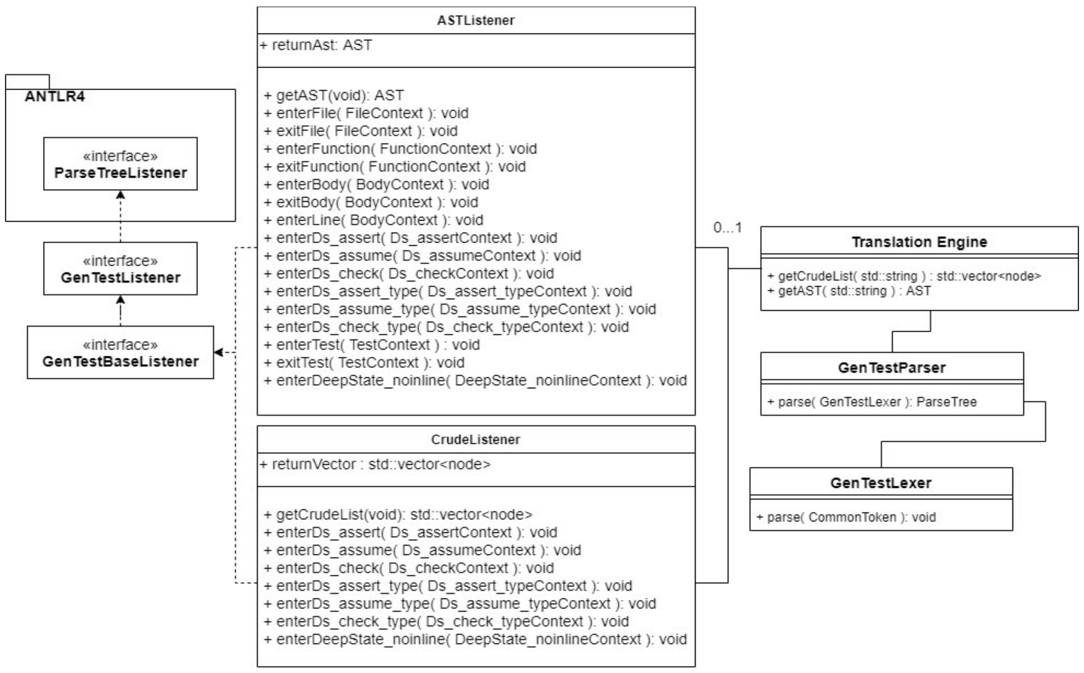

ASTListener
+ returnAst: AST
+ getAST(void): AST
+ enterFile(FileContext): void
+ exitFile(FileContext): void
+ enterFunction(FunctionContext): void
+ exitFunction(FunctionContext): void
+ enterBody(BodyContext): void
+ exitBody(BodyContext): void
+ enterLine(BodyContext): void
+ enterDs_assert(Ds_assertContext): void
+ enterDs_assume(Ds_assumeContext): void
+ enterDs_check(Ds_checkContext): void
+ enterDs_assert_type(Ds_assert_typeContext): void
+ enterDs_assume_type(Ds_assume_typeContext): void
+ enterDs_check_type(Ds_check_typeContext): void
+ enterTest(TestContext) : void
+ exitTest(TestContext): void
+ enterDeepState_noinline(DeepState_noinlineContext): void
ANTLR4
«interface»
ParseTreeListener
«interface»
GenTestListener
«interface»
GenTestBaseListener
0...1
Translation Engine
+ getCrudeList(std::string) : std::vector<node>
+ getAST(std::string) : AST
GenTestParser
+ parse(GenTestLexer): ParseTree
GenTestLexer
+ parse(CommonToken): void
CrudeListener
+ returnVector : std::vector<node>
+ getCrudeList(void): std::vector<node>
+ enterDs_assert(Ds_assertContext): void
+ enterDs_assume(Ds_assumeContext): void
+ enterDs_check(Ds_checkContext): void
+ enterDs_assert_type(Ds_assert_typeContext): void
+ enterDs_assume_type(Ds_assume_typeContext): void
+ enterDs_check_type(Ds_check_typeContext): void
+ enterDeepState_noinline(DeepState_noinlineContext): void

***Figure 4* -** Classes of the Translation Engine Component

of C or C++ code files containing DeepState unit tests which utilize DeepState syntax. Nominally, the Engine is expected to take as input a stream of characters and lines, representing the file to translate, and perform analysis which will A) identify translation terms and B) abstract these translation terms into a generalized representation. By *translation term*, we refer to any combination of characters which represents a string literal in DeepState syntax and which requires translation into another library. Further generalized, the objective of the Translation Engine is to take a file to translate (i.e. myUnitTest.cpp) and produce an abstract representation of its contents (i.e. conditionals, functions, classes) in a data structure which identifies special DeepState syntax, that the File Assembler can then use to deconstruct this syntax into its output translations in external frameworks.

This responsibility requires several classes which work together to produce the outputted data structure. As depicted in Figure *4,* the Translation Engine uses the ANother Tool for Language Recognition (ANTLR) library to implement this component's functionality. ANTLR is a library which provides a standard method for producing parsers capable of recognizing constructs defined by a language grammar (more on ANTLR can be found at https://www.antlr.org/). The parser generated by ANTLR identifies DeepState syntax as defined by a custom-made language grammar. What this provides is a comprehensive, fully-configurable parser capable of efficiently understanding program structure while minimizing the overhead required to create a custom-made parser capable of accepting a wide-range of programming syntax.

The first class to note in *Figure 4* is the Translation Engine class. It acts as the external interface for all other sub-components of the Engine and provides the utility functions needed by other modules for code file interpretation. The class makes use of two other major components of the Translation Engine, the GenTestParser class and GenTestLexer class, to provide these services. Both of these sub-components are auto-generated by the ANTLR4 library and make use of the ANTLR4 C++ run-time to function. At the present, the ANTLR4 runtime library is comprised of header files incorporated into the project.

The GenTestParser and GenTestLexer classes are the core of the Translation Engine parser. These classes serve to A) interpret the file to translate based on a grammar written by GenTest (GenTest.g4) and B) provide interpretations for the base string tokens in the file to translate. Together, the parser and lexer work to create a hierarchical structure of the file and to identify translation terms, as previously discussed.

However, the parser and lexer alone do not have the capability of creating the data structures needed to convey the abstracted information they have identified to other modules. ANTLR4 listeners are used to provide this capability. Listeners work by watching the GenTest parser module and waiting for specific events. When one of these events occur, the listener can use information provided by the parser at that time to construct the necessary structures to provide for translation. In the class diagram, the ASTListener and CrudeListener are two separate implementations of the common interface for a GenTestListener based on the GenTest grammar. One is specifically designed to create an AST from the information, while the other is meant to provide only a list of translation terms and their relative positions in the file to translate.

As stated, the main interface for the Translation Engine component is its Translation Engine class. A more detailed overview of this class is given below:

TranslationEngine Class

Field Parameters: N/A
Functions:

- **getCrudeList(std::string filePath) : std::vector<node>**
  The getCrudeList function takes in a string filePath representing the location of the file to translate and runs parsing operations on that file. For every translation term in the file, a node is added to the return vector containing the non-terminal representation of that translation term and its relative position in the file. After parsing operations are completed, the function returns a vector containing a set of nodes.

- **getAST(std::string filePath) : AST**
  The getAST function takes in a string filePath representing the location of the file to translate and runs parsing operations on that file. For every construct found in the file, an abstract representation is added in a hierarchical fashion to an Abstract Syntax Tree (AST). This AST is then returned at the end of execution and contains all found constructs with their relative positions in the file to translate.

The two methods described above provide the main services of the Translation Engine to other components in the GenTest architecture.

4.3 Binary Parser

The binary parser will transform the unstructured binary data created by DeepState into structured values that can be inserted into a test. This parser will provide support for both reading the binary values from a *foo.test* file generated by DeepState, and for extracting binary values from DeepState at runtime. At its core, DeepState supports the fuzzing of several primitive data types. These fuzzed values are then used by DeepState to run the tests in a harness.

DeepState supports fuzzer integration by exposing a public API that allows users to declare abstract values in their tests that are later filled in by DeepState. Typically, this API follows the "Deepstate_Type()" idiom, where "Type" is the type of the value to be returned. For example, "Deepstate_Int()" returns an integer that was generated by a fuzzer. Tables 1 and 2 detail the numeric and non-numeric types supported by DeepState, respectively. For each type, the size in bytes is also shown. The binary parser must provide support for the extraction of these values from unstructured binary data. Because binary values may come from different types of input stream, the Binary Parser's interface must be decoupled from any one form of input.

As discussed with our client, the binary parser will need to translate unstructured binary from multiple sources. Thus far, two main sources of unstructured binary data have been identified: (*i*) Test files (*foo.test*) generated by DeepState, and (*ii*) raw binary data provided by DeepState at runtime. To maintain functional parity with the DeepState runtime, the binary parser should internally use DeepState's public API for retrieving values.

Toward the fulfillment of the requirements outlined in above, we have identified the necessity for two sub-modules of the binary parser: (*i*) Binary Representation Transformer, and (*ii*) Binary Value Iterator. Broadly, these modules must (*i*) receive unstructured binary data from memory or from a binary file generated by DeepState, and (*ii*) provide an iterator interface to transform this unstructured data into a valid representation of one of the types outlined in Tables 1 and 2, respectively. To fulfill these requirements, the need for 2 classes has been identified.

BinaryParser Class

Figure 5 shows the interface provided by the BinaryParser. The BinaryParser class facilitates the parsing of binary data from arbitrary input streams and from an input file specified by its string filename. The BinaryParser stores the data it parses in a vector of bytes. This vector is copied to the internal data of the BinaryIterator returned by getBinaryIterator(). Because DeepState generates a maximum of 4096 bytes per test, we do not consider the cost of copying the data to be prohibitive.

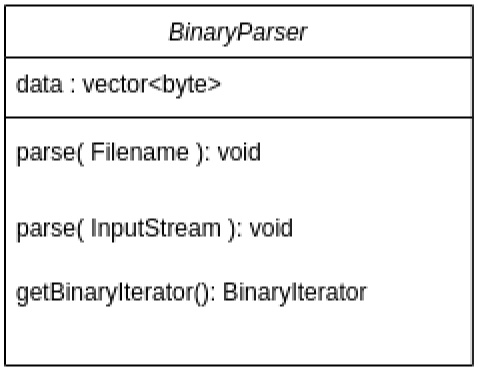

BinaryParser
data : vector<byte>
parse(Filename): void
parse(InputStream): void
getBinaryIterator(): BinaryIterator

Figure 5 *- The interface provided by the BinaryParser class, complete with member and method definitions.*

BinaryIterator Class

Once raw data has been parsed from the unstructured input data, it needs to be interpreted as the correct data types. To achieve this, we propose the *BinaryIterator* class that provides an iterator interface to this binary data. This interface is proposed to encourage separation of concerns: knowing the types the actual binary data represents requires knowledge of the structure of the test, while interpreting the binary data on-demand requires only knowledge of the requested type.

The BinaryIterator class provides an object-oriented interface to the existing DeepState API for returning values from the unstructured binary data. This interface is shown in Figure 6, where an abstract “nextT()” function that returns a value of type T is shown. Internally, a function “nextT()” will return the result of “DeepState_T()”. The BinaryIterator class contains a method of this form for each of the types shown in Tables 1 and 2. For example, the BinaryIterator contains the functions “nextInt” returning int, and “nextChar” returning a character. These functions use “DeepState_Int()” and “DeepState_Char()” respectively. For each item returned by the BinaryIterator, the internal index is incremented by the size of the type returned. As a concrete example, a call to “nextUInt” will advance index by 4 (sizeof(unsigned int)).

Type	Unsigned Type	Size (Bytes)
size_t		8
long		4
float		4
double		8
int64_t	uint64_t	8
int	unsigned int	4
int32_t	uint32_t	4
short	unsigned short	4

Table 1 *- For each numeric type that will be supported by GenTest, the size in bytes is shown. If available, an unsized type is also included. Note that for all type X where the type unsigned X exists, sizeof(X) = sizeof(unsigned X).*

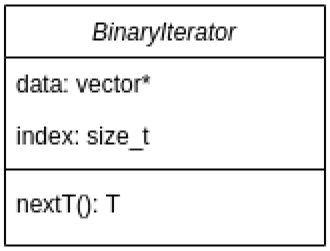


Figure 6 *- The interface provided by the BinaryIterator class. Instead of listing one method for each of the types recorded in Tables 1 and 2, we specify a generic "nextT()" method that returns a value of type T. In the actual implementation, for each type T specified in Tables 1 and 2, nextT() is defined.*

Type	Size (Bytes)
bool	1
string	[1,4096]

Table 2 *- For each non-numeric type that will be supported by GenTest, the size in bytes is shown. Because strings are often variable in length, a string may be requested with a minimum length of 1 (not including the terminating byte), and a maximum length of 4096 (the maximum supported by DeepState).*

The Binary parser and iterator will be used in conjunction with the translation engine by the file assembler to create standalone unit tests.

4.4 File Assembler

The responsibility of the File Assembler is to reconstruct the test file utilizing the Translation Engine's output as well as the Binary Parser to generate a standalone test file that no longer requires DeepState to run. The File Assembler has two main objectives. The first objective is the successful reconstruction of the standalone file. Successful implies the operation of the file is identical to that of the DeepState test. The second main objective is the correct insertion of DeepState generated values. The File Assembler will utilize the Binary Parser for getting the generated values, then insert those values into the parse tree for use in the construction process.

The structure of the File Assembler contains multiple parts, which are laid out in *Figure 7*.

File Assembler Class Layout

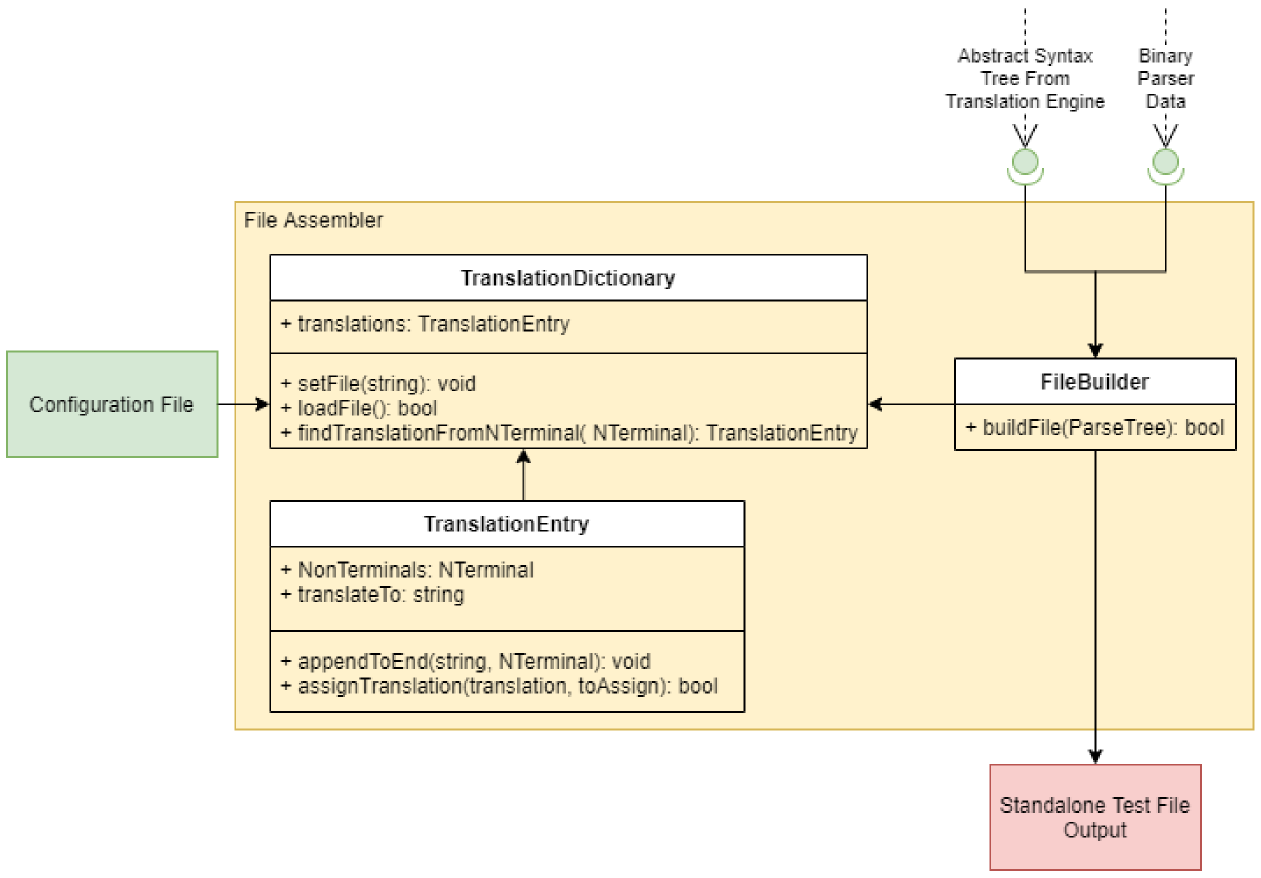
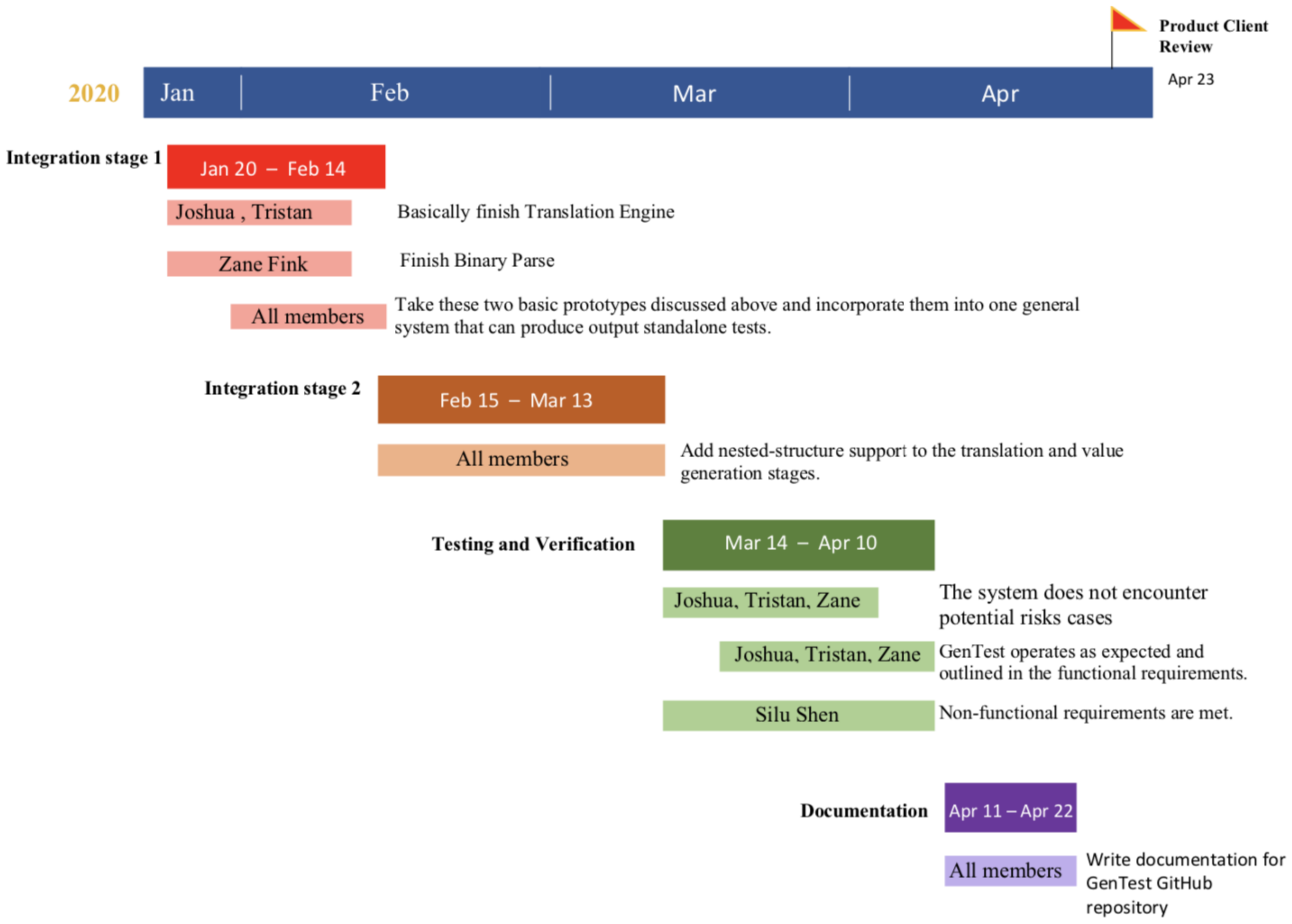

Abstract Syntax Tree From Translation Engine
Binary Parser Data
File Assembler
TranslationDictionary
+ translations: TranslationEntry
+ setFile(string): void
+ loadFile(): bool
+ findTranslationFromNTerminal(NTerminal): TranslationEntry
Configuration File
FileBuilder
+ buildFile(ParseTree): bool
TranslationEntry
+ NonTerminals: NTerminal
+ translateTo: string
+ appendToEnd(string, NTerminal): void
+ assignTranslation(translation, toAssign): bool
Standalone Test File Output

Figure 7 - UML Class Diagram for the File Assembler.

The first class in the diagram is the TranslationDictionary. The TranslationDictionary class is utilized by the FileBuilder to find a translation for a NonTerminal (The base unit for translation). The TranslationDictionary will take in a configuration file. This configuration file will contain a translation for certain NonTerminals. LoadFile will then be called, and the TranslationDictionary will build a translation list. This translation list is a linked list utilizing the TranslationEntry. The File Assembler contains no public facing parts, and is exclusively called from the GenTest process. The FileBuilder will handle the process, with subclasses making the process increasingly robust. Additionally, the Binary Parser will be called by the File Builder to acquire data to be inserted into the symbolic value locations of the test. The FileBuilder will traverse the parse tree given by the Translation Engine. Due to the tree structure, the FileBuilder will be able to intuitively understand file scope. At the end of the process, the FileBuilder will then output the standalone C++ test file not requiring DeepState to compile and run.

Section 5: Implementation Plan

This part is the project implementation timeline using a Gantt chart. The implementation phase and personnel allocation of the project are briefly described in the figure.



Section 5.1 Integration Stage 1

GenTest already gets onto integration stage 1. During this phase, we will incorporate Harness File Translation and Binary Parser 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 5.2 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 5.3 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.

Section 5.4 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 5.5 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.

Section 6: Conclusion

In its current form, DeepState provides an effective framework for accessing advanced, automated testing tools such as Fuzzers and Symbolic Executors. Still, the platform faces issues because of the costs associated with transitioning real-world projects from more mainstream frameworks. For companies and developers, pre-existing testing cases makes the incorporation of DeepState into testing suites difficult because it requires costly rewrites before the features of DeepState are usable. GenTest aims to provide a solution for developers, which enables them to maintain their testing suites in other frameworks, while providing an easy and convenient way to use DeepState functionality external to the library.

The purpose of this document was to provide an in-depth overview of the software required to implement our client's envisioned solution. By designing software that is modular, well-documented, and thoroughly-tested, we anticipate the production of a product that not only satisfies our client's needs, but helps bring DeepState into the toolbelt of every developer.

Although we are a small team working on a small problem, we envision a future where catastrophic software failures are a relic of a bygone era. GenTest hopes that our software, used in conjunction with DeepState, will help to provide a world where safe, reliable software is the exception, not the rule.

Appendix A

The basic jargon and keywords for the project are defined below:

- **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 generate 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).

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.

[8] 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/.