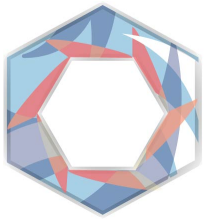
GenTest
November 8th, 2019



Technological Feasibility Document

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

Table of Contents

Section 1 Introduction

1.0.0.0 Project Description

Efficiently creating quality software is a major hurdle for a wide-range of commercial industries. In 2014, the Standish Chaos Report [8] 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 [8] in 1995. While development strategies have evolved since [11], [12], 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 [10], [11], [12].

TDD (Test-Driven Development) is one such method which has seen success in recent years [9] and has led to an increased adoption of unit testing by major software companies as a tool to address this problem [9]. 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 [10], while unsuccessful projects typically have 7 or more defects [10]. 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 [9]. 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 [13]. 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]

1.1.0.0 Standard DeepState Workflow

To execute the DeepState workflow, 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.

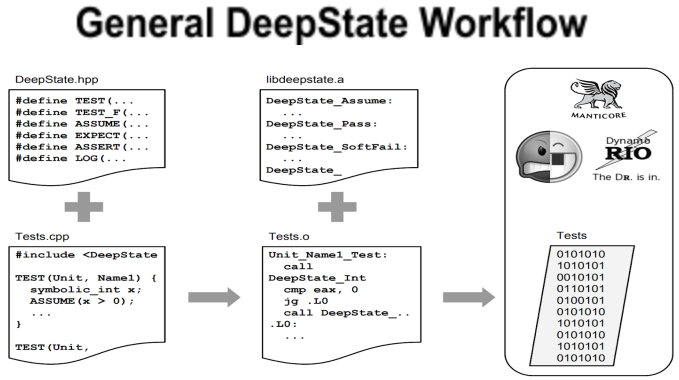


Figure 0 - The generalized workflow of the DeepState framework.

1.2.0.0 Problems and Solutions

For most developers, transitioning from their current framework to the above process can be difficult. 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.

To solve these problems, team GenTest has been contracted to make a solution that will:

- Provide an easy-to-use method for creating standalone unit tests from DeepState in traditional testing frameworks such as GoogleTest.
- Allow for the auto-generated binaries from Manticore, Angr, and Dr. Fuzz to be used in the developer’s traditional framework.

- Does not need to make use of the DeepState library to benefit from its automatic testing techniques.

If successful, the solution created by GenTest will provide an easier way for developers in companies like Google, Facebook, and Bloomberg to make use of DeepState without the need to migrate testing suites or rewrite previous unit tests.

1.3.0.0 Document Overview

The main goal of this document is to discuss the technical challenges and main design considerations for our solution at this stage in development. **Section 1** will introduce major technical challenges facing the project, **Section 2** will analyze these challenges, and finally, **Section 3** will discuss how these approaches will be combined to create a well-crafted, effective solution for our client's problem.

Section 2 *Technical Challenges*

Introduction

We have identified four main technical challenges that must be overcome for the delivery of a successful product. Broadly speaking, the challenges are in the parsing of data produced by DeepState, the subsequent transformation of the data into human-readable format, and the generation of standalone framework-specific tests to output. In the following subsections, we introduce these challenges and motivate their solutions:

- **Harness File Translation**
- **Binary Type Conversion**
- **Decision Capture**
- **Nested Structure Support**

2.1.0.0 Harness File Translation

Recall that developers write test harnesses containing abstract tests that DeepState then concretizes with the help of fuzzers and symbolic execution engines. A successful solution would need to translate these tests written in the syntax of the DeepState testing framework into the syntax of another testing framework, such as Google Test. Because testing frameworks rise and fall in popularity, the developed solution should be flexible enough that other testing frameworks can be supported without needing to change any code written as part of this project, maximizing the longevity of GenTest's software.

2.2.0.0 Structure Support

Typically, a programmer using DeepState will not only want to test primitive types such as *char,* and *int*, but will want to test the types they define. It is common that developers will build complex data-types out of a combination of the primitive types and custom structures. To develop a product with maximal utility, support for the export of these arbitrarily-nested data types will need to be implemented.

2.3.0.0 Binary Type Conversion

DeepState generates values for each test and writes them to a binary file in a pre-defined format for later use. Because standalone tests need to be generated, it is imperative that these values are decoded into a human-readable format. To run a test, DeepState must parse these files and insert the values found therein into the tests in a test harness. Similarly, our solution must be capable of parsing the values from the test files and inserting them into the standalone unit tests.

2.4.0.0 Decision Capture

DeepState supports “Decision Capture”, where some number of permutations of a sequence of function calls are generated, allowing the programmer to test that execution order of function does not affect the correctness of the program. Because this allows programmers to test behavior that would otherwise be laborious to test, GenTest needs to export this functionality into the standalone tests. Additionally, the option to exhaustively generate permutations or a subset of permutations should be retained.

2.5.0.0 Conclusion

In addition to overcoming the aforementioned challenges, GenTest needs to develop a solution with efficiency as a priority. The solution to each challenge needs to be efficient with respect to both time and space. By designing with efficiency at the forefront of our efforts, we hope to develop a product that will see the most utility in the workflow of programmers.

Section 3 *Technical Analysis*

3.0.0.0 Introduction

Given the technical challenges presented in the last section, we will now discuss the potential solutions, approaches, and tradeoffs associated with addressing these issues. This section presents each technical challenge in greater detail and provides additional information on each micro-solution of the system. As mentioned before, the technical challenges facing the GenTest software are:

- **Harness File Translation**
- **Auto-Generated Binary Encoding**
- **DeepState Decision Capture**
- **Engine Accessibility Support**

3.1.0.0 Harness File Translation

The main objective of the GenTest software is to create a usable, equivalent test case in a framework external to DeepState. This requires the translation of preexisting DeepState testing files into their non-DeepState counterparts and, as previously mentioned, is a major technical hurdle for the GenTest project. To auto-generate unit tests in developer-friendly frameworks, the GenTest software must be able to:

- *Parse the native C++ DeepState test harnesses and translate these structures into similar or exact constructs in the corresponding framework.*
- *Efficiently read and associate commands in the DeepState harness into a pointer structure for later export.*

In accordance with these main objectives, several metrics are discussed below which measure how well proposed solutions align with the needs of this technical challenge and why these metrics are vital to gauging solution quality.

3.1.1.0 Metrics

A truly well-made solution would need to create a parser which supports the following metrics:

- **Efficient Capture**
- **Translational Flexibility**
- **Maintainability**
- **Minimal Dependency**
- **Efficient Pointer Translation**

The next two pages provide a discussion of each of these metrics in more detail.

3.1.1.1 Efficient Capture and Translation Flexibility

Developers using our solution are primarily concerned with *how fast* a harness can be translated and into *what* frameworks. This means that they are focused on the efficient capture of harness details and the translational flexibility of our parsing capability. The ideal and most usable solution to file harness translation for the user would be seemingly instantaneous: the developer provides a test harness and all tests, functions, and fuzzed values are immediately available for use in *most* frameworks a developer would want. This is why a parser which solves test harness translation must be concerned foremost with **efficient capture** and **translational flexibility**.

3.1.1.2 Maintainability

From the perspective of the DeepState team, ensuring that the product is usable for years to come is a vital metric for the solution. Over the course of development, changes will likely be made to the DeepState software and its syntax. An important quality of any approach attempting to solve this technical challenge is its adaptability to those changes. To support the continuing development of DeepState, the solution must be stable through major system updates and be easily **maintainable** despite them.

3.1.1.3 Minimal Dependency

While supporting development and maintenance involves creating a system which is easy-to-understand and simple to use, it is similarly dependent on the libraries that it employs. For DeepState to be able to support and extend the GenTest solution easily and effectively, the parser must have **minimal dependencies**. It should not have main components rely on libraries which may become outdated or unusable in the future. This will ensure the stability and quality of the solution even beyond the duration of the GenTest project.

3.1.1.4 Efficient Pointer Translation

Another contributing factor for both developers and the DeepState team is efficient and simplistic data storage. After the parser reads and sorts data, it will need to store it in a data structure, most likely made of pointers. Optimally the parser would provide a straightforward method of storing this data which is both efficient to meet the needs of developers wanting instantaneous access to other frameworks and simple to support maintenance and development by the DeepState team. Balancing these two objectives means creating a system with **efficient pointer translation**.

Based on these metrics, one can imagine that a well-implemented parser would provide a standalone header file to include in other GenTest software modules, with an easy to use, maintainable algorithm to translate harness files for export into a data structure.

3.1.2.0 Approach 1 *Standard Parsing*

One of the first and simplest approaches to creating such a solution is to implement a standard parsing algorithm. This would involve creating an in-house parser in C++ using built-in functions such as strtok or other similar standard library tools. All parsing functions, constants, and search items would be designed to target specific DeepState syntax (i.e. TEST, Symbolic<Type>) and would generate a pointer structure from these results with all pertinent information about the DeepState test harness.

3.1.2.1 Efficient Capture

Such a design would allow for the parser to be concise: it looks for and records only what is necessary to enable important DeepState functionality. This would, over time, provide a simpler solution and better-documented tool if created well because it does not conflate the purpose of the software module. Instead of focusing on how to translate both the C++ syntax and the DeepState unique syntax, an approach using the standard parsing algorithm would be focused solely on its main function: reading only what is essential for making DeepState tests functional in other frameworks and leaving the interpretation of C++ syntax to the frameworks within which they are placed.

3.1.2.2 Concise Pointer Translation

Similarly, because the parser in a standard parsing approach is concise, the storage of data in pointer structures is efficient. All data read from the DeepState test harness can be directly stored in a pointer structure without the need for any intermediary interpolation of the data. This makes storage consistent and fast, while minimizing complexity when tackling other concerns such as translation. What this creates is an approach which can balance the needs of both developers and the DeepState team.

3.1.2.3 Challenges with Maintainability

The major advantage of focusing only on important DeepState syntax is the solution better aligns with the problem space. However, it also creates problems for software longevity because it contains close coupling between the current implementation of DeepState syntax and parsing. Research by the Institute of Electrical and Electronics Engineers (IEEE) indicates that coupling is one of the three major indicators of reduced system longevity and architectural instability [15]. Consider that over time, currently used syntax may be deprecated in favor of new syntax, and these changes require the

need for continual updates to the GenTest solution. Such continual updates are considered in industry to be burdensome, as maintenance typically accounts for 67% of all project development lifecycle costs [14]. Thus, this problem could negatively impact the DeepState framework long past the development cycle of the GenTest project.

3.1.2.4 Ineffective Translational Flexibility

An additional consequence of high coupling between the parser and current DeepState syntax is implementation dependent translation. By ignoring C++ syntax altogether and focusing only on the DeepState Syntax, exporting DeepState tests to frameworks requires they support C++ and makes the translation of data highly inflexible. Should DeepState want to support other languages in the future (such as C), the standard parsing approach would provide additional challenges for extensibility.

3.1.2.5 Metrics Summary

Development Metrics Supported by the Standard Parsing Approach

Supported Qualities	Standard Parsing Approach
Minimal Dependency	O
Pointer Translation	O
Efficient Language Capture	O
Maintainability	X
Translational Flexibility	X

Figure 1 - The metric qualities supported by the standard parsing approach.

These tradeoffs help to better understand how a solution using this approach would support the main objectives of this technical challenge, while likewise anticipating some of its pitfalls. *Figure 1* on the previous page contains a summary of the discussed pros and cons for the standard parsing approach. The approach provides effective support for Pointer Translation, Efficient Language Capture, and Minimal Dependencies due to its concise method of solving harness file translation. It suffers, however, from coupling

between the parser and implementation specifics details, making maintainability and translational flexibility difficult because implementation specifics could and most likely will change.

3.1.3.0 Approach 2 Abstract Syntax Trees

Another approach to translating harnesses would be Abstract Syntax Trees. A major drawback of building a normal parser specific to DeepState, as mentioned before, is a lack of maintainability and translational flexibility. Using an Abstract Syntax Tree (hereon referred to as AST) helps generalize the approach to parsing harnesses, providing a standard interface that can model code functions, types, and dependencies without the need for implementation specific knowledge.

3.1.3.1 How it Works

Using an Abstract Syntax Tree approach, the parser opens and provides the C++ harness file to an Abstract Syntax Tree Generator (i.e. Clang) and this harness file is then translated into a tree which contains all data relating to the harness structure. The GenTest parser is then responsible for managing this Syntax Tree to translate and identify important pieces of the syntax which need to be placed into the file in the new framework. This information is then provided to an external module which handles the generation of syntax and the creation of the standalone file.

3.1.3.2 Maintainable

This approach greatly improves maintainability because it is not tied to the implementation specifics directly. Since it will parse all types, variables, functions, and constants with no need for formatting restrictions or understanding DeepState specific syntax, the parsed structure will not be as greatly impacted by updates to the DeepState code base. The solution therefore is more flexible from a maintenance standpoint and does not require continuous updates to be effective.

3.1.3.3 Ineffective Pointer Translation

While the abstraction of implementation specifics does provide easier maintenance, it similarly creates challenges in the generated data structure. Since DeepState test harnesses use a special syntax that does not always coincide with the general syntax of C++, Abstract Syntax Trees will produce errors when reading harnesses because some code is unknown or unexpected. Using libraries like Clang on test harnesses may produce

overly complex ASTs or cause warnings, errors, and incorrect information to be generated in or with the AST. These errors add another level of interpolation for an AST over a standard parsing approach because the GenTest software would need to implement a wrapper that could manage and identify these potential pitfalls.

3.1.3.4 Inefficient Capture

Using AST library Clang in C++ also indicates that the abstract syntax tree generated captures extraneous data. Libraries like Clang are designed to recursively find all related references to a definition in code. This means that when DeepState uses the Symbolic<Int> macro to define a randomly generated integer value, the AST will include all preprocessor and other class definitions that support it. Such recursion includes extraneous data in the tree that is generated, making this approach more inefficient than its counterparts and inconcise.

3.1.3.5 Increased Dependencies

Due to the complexity of implementing an AST generator, using the AST approach requires the use of AST libraries to be effective. An in-house AST generator implementation would be time-consuming to create and the ROI for writing such a system would be low due to the need for additional wrappers to make the approach functional. At the present, the only major library that supports AST generation in C++ is Clang. While Clang is robust and has many years of development, updates, and stable releases, it does add another level of dependencies. This is an issue as GenTest plans to provide long term support for standalone test generation in the DeepState environment, and adding additional dependencies could cause compilation errors or unintended behavior if the Clang codebase changes.

3.1.3.6 Metrics Summary

Development Qualities Supported by ASTs

Supported Qualities	Abstract Syntax Tree Approach
Minimal Dependency	**X**
Pointer Translation	**X**
Efficient Language Capture	**X**

Maintainability	O
Translational Flexibility	X

Figure 2 - The metric qualities supported by the AST approach.

Figure 2 summarizes the advantages along with the disadvantages for the AST approach. While the pro of an AST approach is maintainability, the solution generally suffers from inefficiency, external dependencies, and difficulties with extraneous data being stored in pointers because of the implementation specifics it abstracts. Due to these issues, the AST approach struggles to support most of the major metrics for solving harness file translation and would need substantial rework to be effective.

3.1.4.0 Approach 3 Context-Free Grammar

Previously, the approaches discussed revolved around interpreting the parsed code from a harness file at the parser-level. An approach using context-free grammar assigns the responsibility of the parser differently. Instead of having the parser deal with the entirety of parsing, interpretation, and storage of code representation, a context-free grammar instead removes the job of interpretation of stored data from the parser and uses the grammar contained in a pointer structure to generate the translated file.

Consider a harness file that is to be parsed. Using an AST or standard parsing approach, every line of code must be interpreted by the parser itself into the output because either there are references which must be filtered from the vector that are unnecessary, as is the case with the AST approach, or the parser itself has to store the lines literally as in standard parsing approach.

3.1.4.1 How It Works

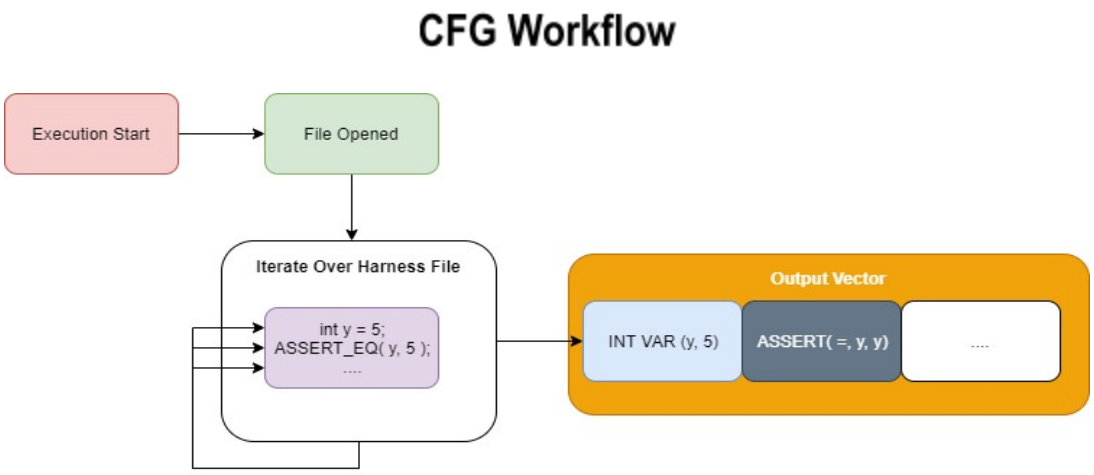


Figure 3 - The generalized workflow for the cfg approach.

The general process of the CFG workflow is as follows. Using the Context-Free Grammar (CFG) approach, the GenTest parser starts at execution time by opening the harness file specified by the user as depicted in *Figure 3.* After, the parser iterates over each line in the file as represented by the white container in the figure. The lines extending from the “Iterate Over Harness File” box indicate that the module is iteratively looking at each line of code. These lines are analyzed, translated to a CFG representation, and then stored sequentially in the output vector for later interpretation by other modules.

3.1.4.2 Improved Maintenance Flexibility

Using this method solves one of the fundamental problems facing translating harness files: implementation coupling. As discussed in the previous approaches, tying a solution to the current form of DeepState syntax would cause maintainability issues if updates were made in the future. A context-free grammar approach sidesteps this almost entirely by focusing only on abstract, common coding structures. In this way, the parser is not coupled with the implementation details of the DeepState harness, greatly improving maintainability should the need for updates arise.

3.1.4.3 Ease of Translation into Other Frameworks

Additionally, the abstraction of C++ constructs into a grammar assists in creating a system which can have translational flexibility. Unlike other methods where all

translation definitions would have to be changed for the system to export into a framework other than the one initially configured, a system that uses context-free grammar can change the export framework by simply changing the terminal values that are generated by the context-free language.

3.1.4.4 Sufficient Capture

Unlike with the AST approach, a context-free grammar does not need to interpret the lines of the code file using implementation specific knowledge to function. Preliminary configurations of a simple CFG indicate that parsing and representing code as terminal or non-terminal variables is simple and only needs knowledge of C++ syntax. This creates a system that is, as a whole, more concise than the AST approach but slightly less concise than the standard parsing approach because it incorporates any value in the translation regardless of its connection to DeepState.

3.1.4.5 Increased Response Times

The consequence of less concision is that a CFG approach is less efficient in response times than with a normal approach. Since this method requires more steps in the translation phase than in other approaches, it does take more resources to operate. However, this drawback is balanced by reasonably optimized translation schemes using recursion to generate and place translated code.

3.1.4.6 Metrics Summary

Development Qualities Supported by the CFG Approach

Supported Qualities	Context-Free Grammar Approach
Minimal Dependency	O
Pointer Translation	O
Efficient Language Capture	O
Maintainability	O
Translational Flexibility	O

Figure 4 - The metric qualities supported by the CFG approach.

Figure 4 depicts the support CFGs provide for vital solution qualities. Overall, a CFG approach supports all of the development qualities to a sufficient level because of its efficient implementation de-coupling and removal of translation responsibilities from the parser. It does, however, require more steps for translation than that of a standard parsing approach and thus may inflate response times without proper optimization.

3.1.5.0 Chosen Approach

Based on the above analysis, team GenTest has decided to utilize **context-free languages** to implement a solution for test harness translation. This decision was motivated by the simplicity that context-free grammars and languages provide to system implementation and its ability to balance critical solution qualities. We believe that a solution using context-free grammar will better support maintainability, longevity, and flexibility than with other approaches.

3.2.0.0 *Structure and Nested Structure Support*

In section 3.1.0.0, we considered how a solution could effectively solve DeepState harness file translation. Through this discussion, it was determined that a context-free grammar implementation would be an efficient solution for accomplishing translation objectives. Structure and Nested structure support is a continuation of this thought, revolving around how to accurately describe and record developer-defined structures in harness files. Unlike the primitives that were previously discussed, structures pose a greater technical challenge for the GenTest project because the need for introspection [1], [2].

Introspection is the ability for a language to analyze the objects, values, methods, and other defined constructs at runtime. Other programming languages such as Java and Python provide tools to analyze the details about classes and structures. For example, Java supports the *class.getDeclaredFields()* function among others to allow for easy analysis of code. Supporting structures, be they nested or non-nested, requires the use of introspection to be effective. This means that the GenTest solution must be able to create a pre-parser module which can:

- *Analyze, record, and understand the types, both primitive and non-primitive, defined in a structure within test harnesses.*

The metrics discussed below qualify the alternative methods suggested in this section and provide a basis for measuring solution quality.

3.2.1.0 Metrics

A successful implementation of the parser extension module would need to support the following qualities:

- **Run-Time File Invocation**
- **Nested Structure Relation**

The details of these metrics and why the are used to gauge solution quality are discussed on the next page.

3.2.1.1 Run-Time Introspection

Most C++ commentary will indicate that introspection is not supported natively by the compiler [1], [2] , [3]. While from C++ version 11 onward the compiler supports the RTTI (Run Time Type Information) interface, it is rigid, incomplete, and only supports classes [1]. Other workarounds do exist, however, which impact the problem space of this technical challenge. Consider a normal structure without any additional non-primitives pointers. There are two main approaches one could take when attempting to get type information: compile-time introspection and run-time introspection. Using a compile time approach, preprocessor command could be used to parse the struct and store its type definitions. With a run-time approach, functions like *decltype* or templates could be used to return the type of structure members x and y [1], [2], [16]. While both of these methods would be effective in a normal scenario, compilation faces major problems due to the current implementation of DeepState.

A key difficulty for a module enabling structural support is the current workflow of DeepState. The DeepState framework is designed to assist in the compilation of test-harnesses when importing DeepState.hpp. After compilation of the test harness, the DeepState libdeepstate.a works to interface with supported engines to execute test cases. When GenTest is executed, the harness file it must introspect is already compiled and similarly the object file has already been created for the harness. The consequence of these operations is that most compilation and run-time introspection methods are not a viable choice because the harness is already compiled before GenTest is run, or code needed by a run-time approach such a *decltype* [16] cannot be accessed in the object file.

Currently, the only way to interact with the DeepState library is through a command-line-interface (CLI). GenTest is designed to extend this CLI to include a command for test harness translation, which is then used to call the GenTest software. Since the invocation of the solution would need to happen during DeepState run-time, as that is when GenTest is called, this limits any solution to run-time introspection. As mentioned previously, standard run-time introspection tools will not work in this context, however this is not to say that a custom-made solution cannot handle introspection at run-time. Thus, any effective solution must be able to support **run-time introspection**.

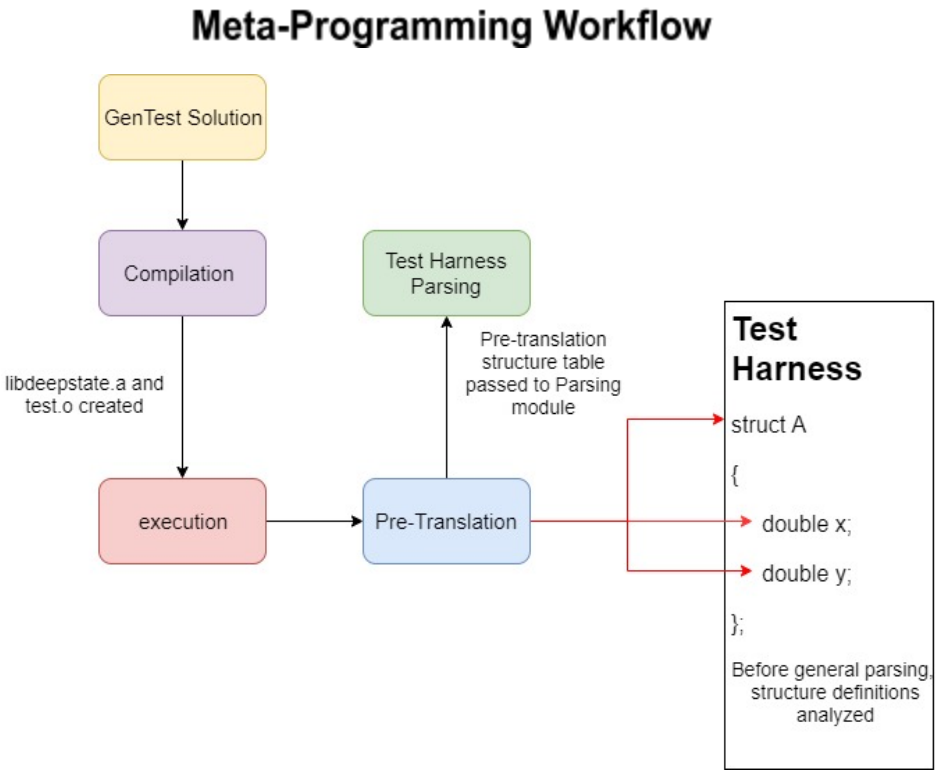
3.2.1.2 Nested Structure Relation

It is common that structures hold pointers to other defined type definitions or structs. Facilitating the analysis of such structures requires that the system be able to relate initialized definitions in the test harness to other structures read by the parser. Without the

ability to match previously defined structures to current ones, it would be impossible to implement a nested structure approach. As such, any solution for this technical issue must enable **Nested Structure Relation**.

3.2.2.0 Approach 1 Meta-Programming

Broadly, meta-programming can be defined as a program that is designed to generate, read, or modify other programs. Implementing a meta-programming solution is a well-known method for bypassing the limitations of C++ introspection ([1] contains an example study for the meta-programming approach). It can be used both at run-time and compile-time,



making it a flexible method for analyzing program structure. The general strategy of the meta-data approach is to record definitions that aid the parser in identifying and storing information about structure definitions.

3.2.2.1 How It Works

Consider the metaprogramming approach in *figure 5* on the previous page. Employing this technique revolves around analyzing the contents of a code file before parsing. After the execution of the DeepState library, the GenTest software enters a preparsing phase

named Pre-Translation. During this stage of execution, all meta-data (structure definitions, etc) is analyzed and recorded in a table. This table is then referenced by the parser when analyzing and storing structure declarations in the test harness code.

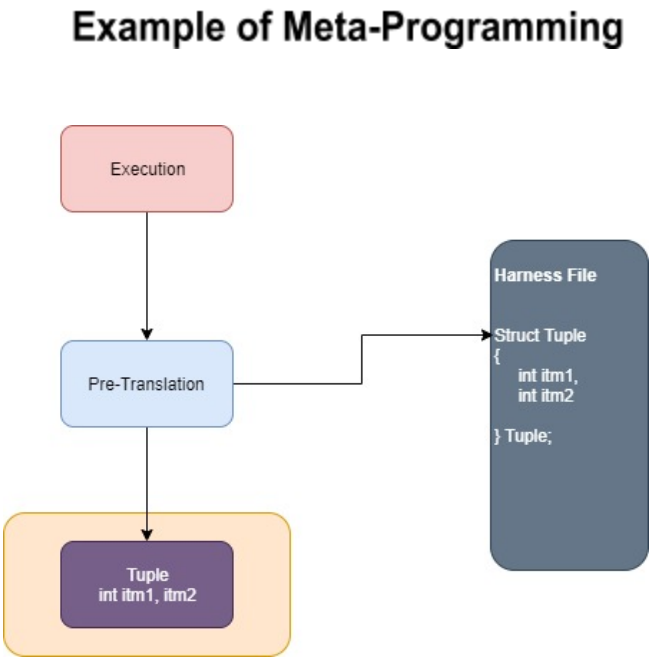


Figure 6 - A basic example of metaprogramming on a test-harness.

Say that a harness file supplied to the GenTest program had a Tuple structure definition as shown by *Figure 6* in the dark gray rectangle. When the GenTest solution starts, it transitions into the Pre-Translation phase. During this step, the parser will look at the harness file and try to find any structure definitions. In this example, the parser will find the definition of the Tuple structure with two int-type field variables. Once this definition is found, the parser will assemble the structure definition in a vector, as shown in the purple box, representing a bin in the vector, and the orange box symbolizing the vector itself. This vector can later be used by the parser during the parsing stage after pre-translation to store CFG non-terminal representations of structures referenced on lines.

3.2.2.2 Effective Introspection Workarounds

Depending on the implementation of DeepState and the testing harness, some functions, structures, and objects may be unreachable due to constraints on scope (such as private functions) [17] or the inability to reach the instantiated objects during run-time (the struct or object is encapsulated in a function). A meta-programming approach bypasses these issues altogether, providing support for run-time introspection. By exporting structure data into a table before parsing, the meta-programming approach circumvents the problems of accessing private qualifiers at run-time because it does not need to invoke the function, class, or file using a call in C++. Similarly, since it does not need to invoke any functions, this approach also does not need access to the internal lines in a function. Thus, the approach better supports run-time introspection because it does not need to do either of these operations.

3.2.2.3 Simple Nested Structure Relation

A benefit of the table structure passed to the parser is that it is flexible. For nested structures especially, this means that the approach of recording information for objects can be encoded in the table in a nested fashion. When the pre-parser responsible for meta-programming reads the file, if one struct has a reference to another, it can easily substitute the corresponding meta-data associated with the referenced struct into the field for data types. This effectively enables the system to relate previously referenced structs to the current.

Say for instance that we had two structure definitions, structure A and structure B. Assume that structure A holds a pointer reference to an object of structure B as a member. The ordering of these structure definitions can lead to potential problems for a meta-programming approach. If structure B were to be instantiated before structure A, or if structure B was defined in another header file, the pre-parser would have no way of knowing how to relate the found type because it does not yet exist. This accounts for one potential problem associated with this algorithm: the sequence of structure definitions in a given file.

While the first scenario in a normal environment should be limited by the compiler, the second creates a limitation for the meta-programming approach: it must be guaranteed that all structure definitions are in-file. This limitation may restrict the use of DeepState by developers wanting to incorporate multiple modules and structures because of the requirement that all definitions are in-file. However, the meta-data approach, despite this restriction, can effectively support nested-structure relation.

3.2.2.4 Metrics Summary

Accounting for the drawbacks associated with nested structure relation, the meta-programming approach remains a strongly viable alternative for this technical issue. With proper planning or documentation for in-file limitations, the approach can overall accomplish the main objective of extended pre-parser module.

Development Qualities Supported by the Meta-Programming Approach

Supported Qualities	Meta-Programming Approach

Run-Time Introspection	O
Nested Structure Relation	O

Figure 7 - The metric qualities supported by the meta-programming approach.

Figure 7 summarizes the supported metrics for the meta-programming approach. Due to its ability to facilitate introspection for the parser module and its easy placement of structure definitions into other structures, the approach can support all needed metrics for a quality solution.

3.2.3.0 Approach 2 Run-Time Invocation

Another approach to solving this technical solution is run-time invocation. Run-time invocation is based on linking the testing harness to the GenTest parser at compilation time. Instead of approaching the problem from a parsing perspective, this approach instead attempts to invoke the executable object (test_harness.o) and call its functions to obtain data about its members using templates to provide introspection (a feasibility analysis of templates as a solution for introspection can be found in [2]).

3.2.3.1 How it Works

Run-Time Invocation requires that the GenTest parser is linked in the makefile with the inputted test harness. After this link occurs, the parser invokes function calls to the harness file and analyzes the results using templates. These results are, similarly to the last alternative, stored in a data structure, however unlike with the other approach, this data is already native to the main parser module.

Example of the Invocation Approach

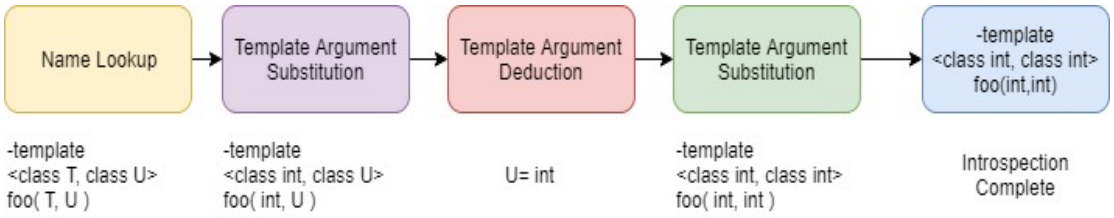


Figure 8 - An example using templates for Introspection.

Say for instance that we had a function named foo() with two arbitrary class parameters T and U as depicted in the text under the Name Lookup box in *Figure 8.* To use the invocation method, this function foo will be fed into a predefined template, which will then use its own definitions and knowledge of members to substitute the first class T into the function. Once this is complete, the process continues to the Template Argument Deduction phase as demonstrated in the figure, where the second class U is deduced to be Int as well based on the implementation of the template. Once placeholders U and T are replaced with int, int, the system has fully introspected the parameters of the function foo.

3.2.3.2 Improved Nested-Structure Relation

This approach provides an easy and effective way to analyze related structures. Using templates, the parser can relate all information about a structure by passing it into a module-defined template *type_description<t>*. Similarly, because the template can be called recursively, nested structure support is efficient. Unlike the previous approach, which required the use of a table data structure, this approach only needs one, more simplistic data structure to capture all structures that are nested. Thus it is a faster approach from the perspective of computation time, than its alternatives and more simplistic for nested structure support.

3.2.3.3 Impossible Introspection

One of the difficulties of the invocation approach is support for introspection. While templates can be used to analyze structures and nested structures effectively, one problem that severely restricts its

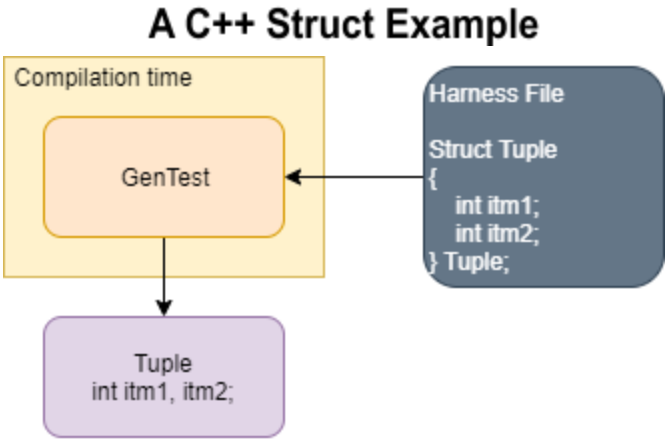


Figure 8 - A basic Example of a Run-Time Invocation

use is the ability to access encapsulated structures. Say for instance that we had the structure definition depicted in the harness file of *Figure 9.*

Most structures used in code are encapsulated either in classes or functions. These local variables in C++ would not be accessible due to their implementation [4]. Thus, any solution to the problem would either need to make a function variables global in the test harness, which would be impractical and severely limiting for the users of DeepState.

Figure 9 - A sample structure definition.

3.2.3.4 Metrics Summary

Development Qualities Supported by the Array Approach

Supported Qualities	Run-Time Invocation Approach
Run-Time Introspection	X
Nested Structure Relation	O

Figure 10 - The metric qualities supported by the array approach.

As a result of the above problems, the run-time invocation approach appears to be severely limiting for DeepState users. Its need for access to local variables defined in other functions or classes makes the implementation impossible in its current incarnation. *Figure 10* summarizes the supported qualities for a run-time invocation approach. While the ability to use templates would increase overall nested-structure relational efficiency, the limitations born from impossible variable access means that the approach cannot support one of the major components of a successful solution to this technical challenge.

3.2.4.0 Chosen Approach

As a result of the above analysis, team GenTest believes that using a **Meta-programming approach** is the better alternative due to shortcomings in the invocation which lead to needs impossible to implement in the C++ programming language.

3.3.0.0 Binary Type Conversion

The translation of DeepState test harnesses would be worthless without the ability to encode the binary values generated by DeepState. When DeepState is executed, it uses fuzzers like Dr. Fuzz to generate what are referred to as “tests”. The files (foo.test) hold the binaries that were produced by the fuzzing engines in DeepState. A major technical challenge for GenTest is the ability to track and encode these values into external testing files.

Exporting the values generated by fuzzing requires GenTest to be able to translate the primitives in DeepState’s engine into hard-coded values in the auto-generated unit tests. This means building a module that can:

- *Translate binaries within the binary file (foo.test) to their appropriate primitive types in the external testing framework.*
- *Transform nested structures with primitive types into appropriate structures in the unit testing framework.*

Accomplishing these goals requires that the implemented solution meets the metrics specified below.

3.3.1.0 Metrics

An effective solution to the problem of auto-generated binary encoding can be qualified by the following traits:

- **Little Endian Support**
- **Primitive Coverage**

The lines below contain a brief description of each listed metric.

3.3.1.1 Primitive Coverage

Most developers using the GenTest solution will need access to as many primitives as are supported in languages such as C and C++. Since tests can use any number of type values, developers will receive the most benefit from a GenTest solution that can support this wide range of primitives and structures for test [7]. From a user perspective then, it is vital that **primitive coverage** be complete and well-rounded. Any solution implemented

by GenTest to the problem of binary conversion must take into account how well it can cover the range of necessary primitives and basic structures.

3.3.1.2 Little Endian Support

From the view of the DeepState team, support for current algorithms of binary encoding is paramount. Currently, the DeepState software encodes binaries in a little endian fashion. To correctly convert binary values to their type equivalents, it is important that any implemented solution to the problem supports **little endian** binary encoding natively. This means that an effective solution to the problem would use a technology that can conveniently interpret binary values in little endian without the need to reverse the order of the bitstring. Such a technology would support efficiency, while also providing easier integration for the DeepState team and GenTest.

A useful solution for both the DeepState team and developers using DeepState would need to include these qualities to be efficient. The alternative technologies below are analyzed and compared using these traits as a benchmark for solution quality.

3.3.2.0 Approach 1 *Bitsets*

The first approach to solving binary type conversion is to use bitsets. Bitsets are a class native to the C++ standard library which converts the String representation of binary values into a data structure holding up to the specified number of bits.

Example of Using Bitsets in C++

```
std::string bit_string = "110010";
std::bitset<8> b1(bit_string);
```

Figure 11 - Sample C++ code using bitsets.

3.3.2.1 How it Works

Figure 11 demonstrates how the bitset class in C++ is used. When a programmer supplies the constructor of the bitset class with a bit string, as shown on line two, it will read each character into the bitset as a boolean value representing one or zero. Conceptually, a bitset is simply a wrapper for an array of booleans, which holds the binary values in an easy and convenient to access structure.

3.3.2.2 Little Endian Binaries Supported

One of the foremost benefits of using a bitset as depicted above is built in support for little endian binaries. Take for instance the bit string 110010. The standard bitset class in C++ would treat the last position in the string as the first index of the bitset. Thus, the bitset will read the binary string from the right-most position until the left-most position, which is essential for supporting little endian values. Without this property, the GenTest software would have to reverse the order of the bitstring from 110010 to 010011 before providing the binary string to the bitset class.

3.3.2.3 String Conversion to Unsigned Numeric Values

Another key advantage of bitsets is their support for conversion from binary to unsigned long and unsigned long long primitive types. From C++ version 11 onward, the bitset class has its own functions for returning these values without the need for any external conversion. This would enable the GenTest solution to support all numeric values that are unsigned (using type casting) and most numeric values without any additional interpretation of the bitset.

Natively Supported Primitives by the Bitset Approach

Natively Supported Primitives	Bitset Approach
Signed Short, Int, Long, Long Long	x
Unsigned Short, Int, Long, LongLong	O
Characters, Strings	x
Floats, Doubles	x

Figure 12 - The supported primitive types for the bitset approach.

3.3.2.4 Lack of Coverage for Other Needed Primitives

Despite these advantages, bitsets do not support char, signed numeric, floats, or doubles natively as demonstrated in *figure 12*. This is a major drawback for the solution because each of these types would have to be handled by the GenTest solution and likely transformed using conversion techniques. These hurdles would increase the complexity of the solution and add potential maintenance hurdles later in development.

3.3.2.5 Metric Analysis

However, given the wide range of needed conversions, bitsets can overall support a vital subset easily and without modification. The drawbacks of the bitset approach are balanced by built-in functionality which could be utilized with separate algorithms to support other types.

Development Qualities Supported by the Bitset Approach

Supported Qualities	Bitset Approach
Little Endian Support	O
Primitive Coverage	O

Figure 13 - The metric qualities supported by the DeepState approach.

Figure 13 summarizes the conceptual pros and cons of the bitset approach. While bitsets provide native little endian binary support and convenient primitive coverage due to built-in functions, it does not provide a full set of primitive coverage. Though these drawbacks would need to be overcome by GenTest to implement this approach, they would not be a major blocking factor for the software. Thus, because of its ability to support Little Endian notation and Primitive coverage, the bitset approach remains a well-rounded solution to the challenge of binary encoding type conversion.

3.3.3.0 Approach 2 Stoi/Stol

An Example Using Stoi

Another way of converting binaries to primitives in C++ is to use the built-in library function stoi or stol. Stoi is used to convert any string into its integer counterpart.

```
int main()
{
    int i = std::stoi( "01000101", nullptr, 2 );
    printf( "%d\n", i );
}
```

Figure 14 - An example of using stoi in C++.

3.3.3.1 How it Works

While typically stoi is used for converting strings such as “53” to the integer number 53, it can convert binary values as well. *Figure 14* demonstrates binary conversion using the stoi library function. In this example, the stoi function is passed the bitstring “01000101” which in little endian represents the number 162 and in big endian represents 69. The stoi function is also supplied the parameters for a pointer which will store the first value (which in this case is unnecessary) and the base of the conversion. Stoi will then convert the binary value into the corresponding decimal value as demonstrated in the next image.

3.3.3.2 Little Endian Not Supported

Output of Figure 14

```
69


...Program finished with exit code 0
```

Figure 15 - The result of running stoi on the bitstring in Figure 14.

Figure 15 depicts one of the downfalls of the stoi approach: it converts big-endian values instead of little endian values natively. The value 69 is the big endian value of the binary conversion for a binary string “01000101”. If this binary string were to have been supplied from a DeepState foo.test file, it would actually represent the number 162. Thus, whenever a binary value needs to be converted, the bit string would first need to be reversed before it is supplied to the stoi function.

3.3.3.3 Native Handling of Signed Numeric Values

The approach, however, benefits from its native ability to handle numeric values that are unsigned. Unlike bitsets, the stoi function can take and translate negative binary values if they contain the unary operator minus (“-”) at the beginning of the bitstring. This

provides extra functionality to the stoi approach which would be helpful in supporting primitive coverage. Yet one hurdle to using this functionality is that binaries in the foo.test files are encoded using an extra sign bit. The consequence of this is that even though stoi supports negative binaries, its native implementation would still need some handling of the bitstring by the GenTest software for negative numeric values to be read.

3.3.3.4 Lack of Other Basic Primitive Support

Like the bitset approach, the stoi function also cannot natively support strings, chars, floats, doubles, and due to the previously mentioned problems, signed numeric values. Thus, the stoi approach does not provide any additional support for primitives above its other alternatives. *Figure 16* summarizes the natively supported types by the Stoi/Stol approach. Based on this table, the tradeoffs of using the stoi or stol function seem to outweigh the benefits.

3.3.3.5 Metrics Summary

As a result of similar primitive support and a big-endian focus, the stoi/stol approach lags somewhat behind that of its bitset counterpart. *Figure 17* on the next page summarizes the tradeoffs discussed thus far for the stoi/stol approach. Though a major benefit of the Stoi/Stol approach is its ability to natively support signed numeric values, it is blocked by the current implementation of DeepState binaries. Thus, it does not provide an advantage over other methods for primitive coverage and requires additional steps by GenTest due to its big-endian focus. Due to these drawbacks, the Stoi/Stol approach supports only one of the three main qualities needed for an effective solution to this technical challenge.

Natively Supported Primitives by the Stoi/Stol Approach

Natively Supported Primitives	Stoi/Stol Approach
Signed Short, Int, Long, Long Long	x

Unsigned Short, Int, Long, LongLong	O
Characters	x
Strings	x
Floats	x
Doubles	x

Figure 16 - The primitive types natively supported by the stoi/stol approach.

Development Qualities Supported by the Stoi/Stol Approach

Supported Qualities	Stoi/Stol Approach
Little Endian Support	x
Primitive Coverage	O

Figure 17 - The metric qualities supported by the stoi/stol approach.

3.3.4.0 Chosen Approach

While neither of the above approaches support structures natively, they do provide a basis for binary-to-numeric conversion which can be used by GenTest to translate other types such as strings, characters, doubles, and more. While bitsets and the stoi/stol functions have similar coverage for primitives, bitsets provide a better solution for our needs because of its little endian support. Thus, GenTest will use **bitsets** to solve the technical challenge of binary type conversion.

3.4.0.0 DeepState Decision Capture

In addition to Fuzzing and Symbolic Execution, DeepState supports tool-decided function calls using the *oneOf* operator [7]. Generally, when a developer writes a unit test, they decide in what order the function to test will be called. DeepState automates this process in a non-deterministic way by allowing for these methods to be called with varying permutations either randomly or exhaustively.

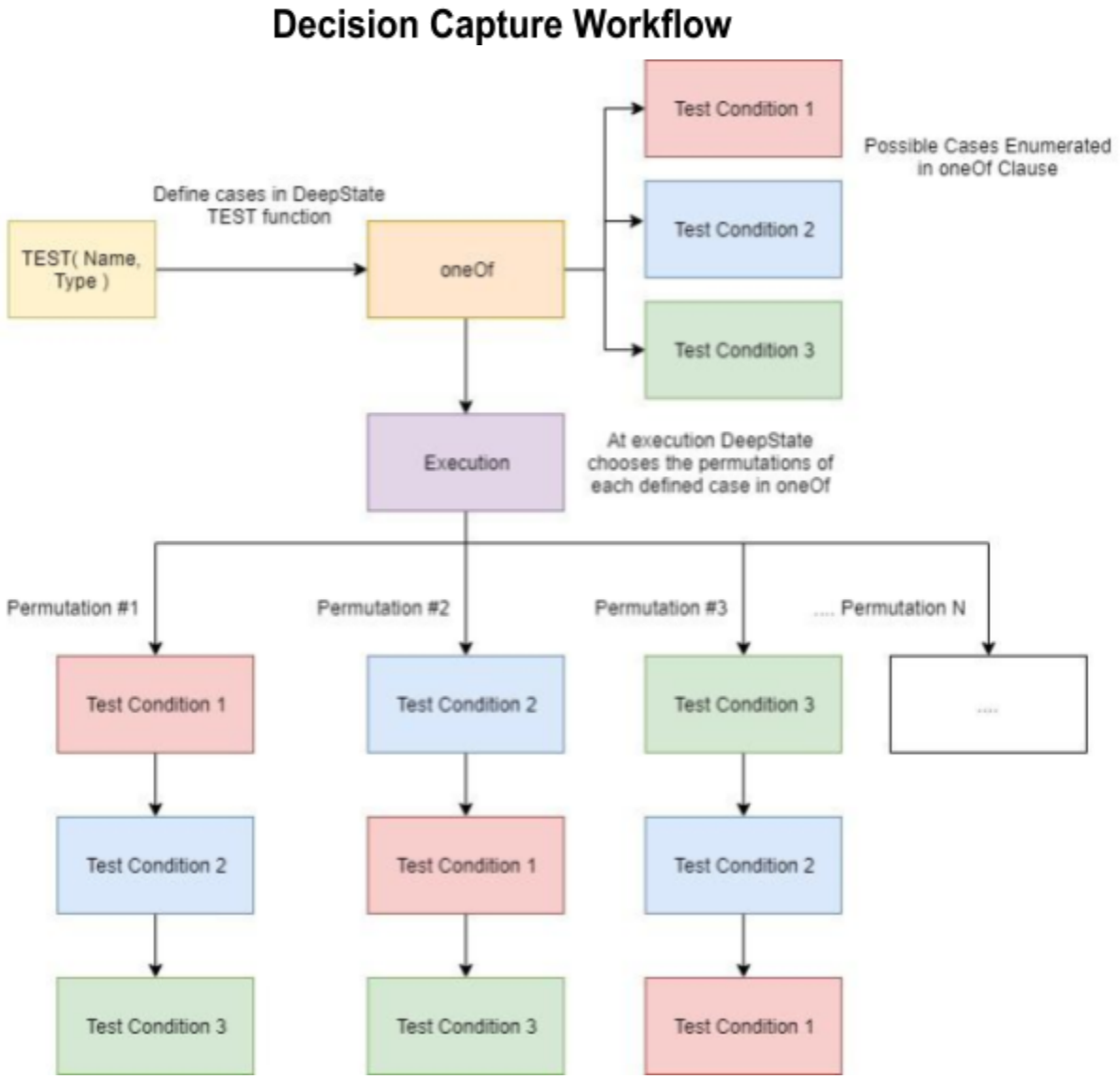


Figure 18 - The generalized workflow of decision capture.

The workflow of deepstate using this processes is depicted by *Figure 18* on *page 34.* When the test called the oneOf function, it will select a permutation of the different test conditions (in this case, the 3 different test conditions). Afterwards, it will execute each permutation of test conditions, and determine if the output is correct, or if that permutation fails. To use DeepState-decided function calls, the developer must use the oneOf clause in the TEST function of the harness file to declare possible cases. For example, if the developer enumerated three different cases that could happen for a given function to test (Conditions 1, 2, and 3), the DeepState software at execution would processes these cases using fuzzed values in either a random order or exhaustively from the first permutation until the Nth permutation.

This process creates another major technical challenge for the DeepState solution because it is a basic component of the DeepState framework commonly used by developers. Since developers may want to use a test enumerated in this fashion in a standalone format, the GenTest software must be able to capture the function call decisions of the DeepState software. To accomplish the goal of capturing these decisions, a solution must be created that:

- *Can capture and store the decisions made by DeepState at execution time.*
- *Allow for any number of permutations to be stored.*

The metrics below discuss how a proposed alternative will meet these needs and what qualities support an effective solution.

3.4.1.0 Metrics

The quality of a solution challenging DeepState decision capture can be judged by the following metrics:

- **Random Permutation Support**
- **Exhaustive Permutation Support**
- **Decision Capture Efficiency**

These metrics will be discussed in more detail on the next page.

3.4.1.1 Random and Exhaustive Permutation Support

Making the solution to this technical challenge robust and complete for the developers using DeepState requires that any solution support **random permutations and exhaustive permutations**. This means that any technical alternative discussed should be able to capture any number of permutations either to a limit or to effectively an infinity. Such flexibility is important for providing users the entire functionality of DeepState and its ability to generate permutations for function calls.

3.4.1.2 Decision Capture Efficiency

Likewise, the storage and encoding of these decisions creates another factor for consideration in any solution: **efficiency**. Depending on the data structure and algorithm used to store captured decisions, the solution will be limited in efficiency and storage capacity. Thus, any solution must take into account the efficiency of its implemented approach from the perspective of:

- *Its ability to efficiently store and read data*
- *The amount of data it can store.*

Considering these two points will assist in ensuring that any module created for the technical challenge will have maintainable performance. Using these metrics, we will analyze and discuss the proposed alternatives below.

3.4.2.0 Approach 1 *Linked Lists*

One possibility for solving the issue of DeepState decision capture is to use a linked list. A linked list is a common data structure in computer science which uses nodes and pointers to nodes to form a list of structures containing data [5]. Using a linked list approach, each permutation can be recorded, as they come, in nodes and added to the list of all calls. Take for example the *Figure 19* on the previous page. The current implementation of

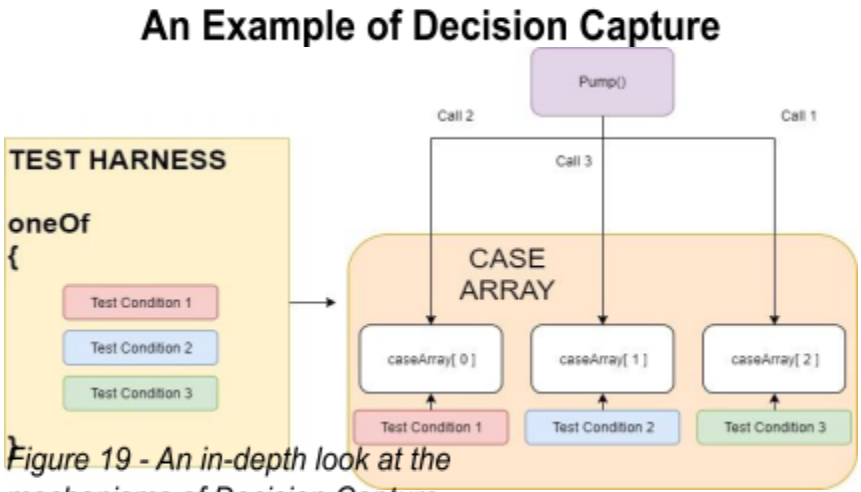


Figure 19 - An in-depth look at the mechanisms of Decision Capture.

DeepState when using the *oneOf* syntax, as depicted in the yellow box labeled “Test Harness”, will parse each enumerated condition from the beginning of the oneOf curly braces to the end.

These cases are then extracted and placed by the DeepState framework into a case array as demonstrated by the arrow extending from the test harness box to the orange box. A macro will call the DeepState function Pump() to create random calls or exhaustively create all permutations for accessing the array of test cases. These calls represented by the arrows coming from the purple box representing the pump() function. At any given time, the pump() function generates random indices from 0 until N, where N is the size of the array, and uses these indices to create random and exhaustive case execution through array access.

3.4.2.1 Exhaustive and Random Permutations

The process of creating random and exhaustive function calls leads to the first major advantage of a linked list approach: infinite size. A linked list has the flexibility needed to store as many calls as executed by the DeepState software. Due to this, the linked list approach can support both exhaustive and random permutations of test functions.

3.4.2.2 Decision Capture Efficiency

One foreseeable problem for efficiency, however, is the time required to traverse a linked list of length N, where N is the size of the linked list. Say, for instance, that the linked list contained one million calls. Using a normal linked list adding function which traverses from the head node until the end node, the linked list would have to perform at least a million operations before adding a new node. This could be a huge hurdle for system performance because to place an item at position N-1, the system must traverse N-2 values.

These constraints may cause long response times if executed on a long linked list. However, since the solution must store each function call in the order that they came, the addNode() function can instead be modified to only add to the end of the linked list, guaranteeing that every new node will take O(1) processing time, where O is the Big-O approximation of system time and O(1) indicates that the execution time does not increase in response to the number of elements in the list, to be added to the list [5], [6]. Thus, the linked list approach is efficient when storing nodes.

3.4.2.3 Problems with Access Computation Times

Yet for data access the chained nature of a linked list is a constant problem for system performance because it has no way of finding nodes except through traversing the list

until the seek node's position. Since we must read every node in the linked list to encode calls, the access time is directly proportional to the number of nodes in the list. In essence, a linked list structure will always have an O(N) access time [5], [6] meaning that access times will grow as the list becomes larger. If N grows by 100%, so will the computation time. When dealing with permutations that could reach into the millions or multi-millions, the computation time using a linked list will inflate dramatically as a result.

3.4.2.4 Metrics Summary

As a result of these problems, linked lists do not handle capture efficiency particularly well. However, the approach does allow for exhaustive and random permutations effectively and efficiently. These strengths and weaknesses are reflected in *figure 20* Overall, the linked list approach can support a majority of the needed metrics for a solution to decision capture. Still, to be functional it would require strategies to overcome some efficiency limitations.

Development Qualities Supported by the Linked List Approach

Supported Qualities	Linked List Approach
Random Permutations	O
Exhaustive Permutations	O
Capture Efficiency	x

Figure 20 - The metric qualities supported by the linked list approach.

3.4.3.0 Approach 2 *Vectors/Managed Arrays*

Another solution to decision capture is managed arrays or vectors. From an access perspective arrays are regarded as one of the most efficient data structures at a speed of O(1) [5] and are one of the simplest data structures to implement. While normal arrays can suffer from issues in insertion and search times, managed arrays provide a convenient solution to some of these problems.

3.4.3.1 Difference between Standard and Managed Arrays

A standard array is generally thought to be static, meaning that it is fixed in size, and cannot keep track of the next position in a sequence without modification [5]. Managed arrays circumvent the limitations of a standard array by implementing wrapper classes which serve as management systems. A vector, for instance, is simply a wrapper which monitors both the current position in an array and how many items are in the array to allow for potential resize operations if needed.

3.4.3.2 Increased Efficiency

Due to this management functionality, vectors provide increased insertion times over its array counterparts. While a standard array only operates at a linear computation time O(N) [5], a vector can operate at a speed of O(1) due to its tracking of the next placement index. This is faster than linked lists in terms of insertion speeds because arrays do not have to perform the costly memory operations that are associated with pointer retrieval. Likewise, access times are greatly improved over other approaches because all arrays have an O(1) access time [5], [6]. As a result, vectors increase the efficiency of program computation speed.

3.4.3.3 Potential Hurdles with Search Speed

Similarly to other discussed methods, arrays normally have an O(N) search speed [5], [6]. This limitation could prove costly in the event that values need to be retrieved out of the order in which they were stored. However, unlike with other approaches, there are methods of improving these speeds. Since the internal arrays in vectors have finite bounds and direct access, search methods can be parallelized to be more efficient. The parallelization of array search would allow for multiple parts of the array to be covered simultaneously and could speedup overall decision capture. Assuming that 50% of the module's code was parallelizable and that four threads were utilized, Amdahl's law states

that the maximum speedup the program could accomplish using parallelism is 1.6x the normal speed. While this is not a dramatic speedup, it would outperform other methods.

3.4.3.4 Removal of Size Limitations

Any solution that support random and exhaustive permutation support needs to have an infinite size. Normal arrays suffer in this metric because they are finite and cannot be resized if necessary. However, vectors provide resize functionality that allows the structure to take on any number of values if needed. This removes the traditional size limitations that would otherwise inhibit an approach using arrays.

3.4.3.5 Metric Summary

Overall, arrays would be a more simplistic approach to solving DeepState decision capture. *Figure 21* on the next page summarizes the characteristics supported by a vector/managed array approach. While search speeds do still provide an issue for this approach, it can be better mitigated using parallelism. As such the increased efficiency in the capture of data provides more benefit for the solution with little drawbacks to make the solution ineffective.

Development Qualities Supported by the Array Approach

Supported Qualities	Vector Approach
Random Permutations	O
Exhaustive Permutations	O
Capture Efficiency	O

Figure 21 - The metric qualities supported by a linked list approach.

3.4.4.0 Chosen Approach

Given the analysis for vectors and linked lists, GenTest believes that vectors, while suffering from some efficiency problems in search times, will be better suited to a solution for the capture

of DeepState decisions due to an increase in computation efficiency. Thus, GenTest will use **vectors** to build a solution for the technical challenge at hand.

3.5.0.0 Proving Feasibility

This section will prove how each individual tool or technique will increase the feasibility of the project.

3.5.1.0 Harness File Translation

In order to prove the feasibility of the team's chosen harness file translation solution, a small simulator of the module will be created to prototype the solution before the demo. Likewise, a full-fledged module containing most of the functionality outlined in the requirements gathering phase will be implemented for the demo of the system. These two prototypes will serve as the basis for actually proving that the solution chosen is valid and sufficient for the needs of the team and its clients.

3.5.2.0 Structure and Nested Structure Support

To prove the feasibility of the structure and nested structure support solution, a demo will be created which includes basic functionality for the pre-parser. In the demo, the team will demonstrate that the module can handle basic structures with no nesting and work effectively integrated with other parts of the system. Independent of other system modules, the demo will also prove that the solution can be used for nested-structures at this stage.

3.5.3.0 Binary Type Conversion

To prove the feasibility of the binary type conversion solution, we have created several basic functions that can already convert sample bit strings from the foo.test binary files generated by DeepState. In addition to these results, the GenTest team is working on several preliminary functions that can model basic structure binary conversions. By the end of the demo implementation phase, we will create software that can demonstrate most if not all of the required primitives as specified in the requirements document. These steps will help to prove the feasibility of using bitsets as a solution.

3.5.4.0 DeepState Decision Capture

To prove the feasibility of the DeepState decision capture solution, GenTest will create a simulation using the DeepState software to verify that the capture of these decisions is possible

using a linked list and model the efficiency using timing functions to ensure that edge cases (such as millions of permutations) will not cause drastic slowdown to automatic unit test generation as a whole. Similarly, the functionality will be implemented in the demo phase to verify that it is indeed operable with the entire system.

3.6.0.0 Conclusion

In this section, the major technical challenges facing GenTest were reviewed and analyzed. Through this analysis, one major strategy for each technical challenge was chosen. To verify the choices and decisions made above, an overview of how these challenges will be proven was provided. GenTest is confident that these decisions represent a complete and full analysis of the given challenges for the solution.

Section 4 *Technology Integration*

Introduction

Using the tools and techniques presented in section 2, we will now analyze how these individual pieces will fit together, as well as looking at how these individual pieces will work together to create a usable standalone test. A few different tools and techniques that will be looked at in this section will be:

- **Translation Engine**
- **Binary Value Parsing**
- **Storage of Generated Tests and Values**
- **Standalone Test File Structure**

An additional section 3.1 will give an understanding of how each individual system will work together to generate a standalone test file.

4.1.0.0 System Overview

The Goal of GenTest is to take DeepState test files (example.cpp) as well as the binary files (foo.test) and translate them into a standalone test file. In order to accomplish this, the tools analyzed in **section 3** will be implemented in multiple parts of the system. The translation engine discussed in **section 4.2.0.0** will utilize a context free grammar, as well as a meta-programming approach. **Section 4.3.0.0** will break down how the binary value parser will work, as well as how bitsets will play a role in the system. In **section 4.4.0.0,** the usage of vectors to pass values through to the standalone file assembler will be discussed. Finally, **section 4.5.0.0** will discuss the standalone file assembler and how it functions. Further detail on the operation and implementation will be expanded upon in the following sections.

4.2.0.0 Operation

During GenTest operation, the translation engine will take in the DeepState harness file, and convert it to a format that the standalone file assembler can utilize. The binary parser will generate values for the standalone file assembler to insert into the standalone tests. Finally, the standalone file assembler will take the values stored in the vector and create a standalone test file by utilizing the values and tests stored inside the vector. *Figure 22* gives a general overview of

the system design. At the top of the figure, are the inputs into GenTest. Everything inside the GenTest box will be pieces that will be implemented, with the output being a standalone test file.

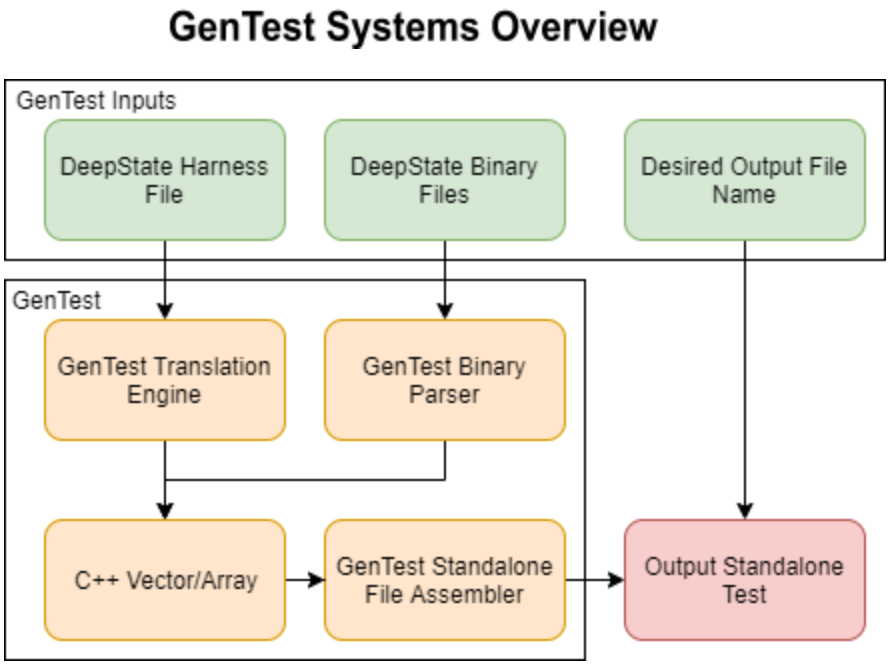


Figure 22 - An overview of GenTest systems.

In further sections, we will discuss how each of the individual parts of the system will accomplish generation of standalone tests.

4.2.0.0 Translation Engine

This section will discuss GenTest's translation engine. The basic operation of this engine will take a DeepState file and translate it into a test file utilizing a similar testing library's conventions and grammar (Eg. GoogleTest). *Figure 23* shows the general flow of how the translation engine will function. In order to understand how the DeepState test is executed, GenTest will require an understanding of runtime execution of the DeepState test file.

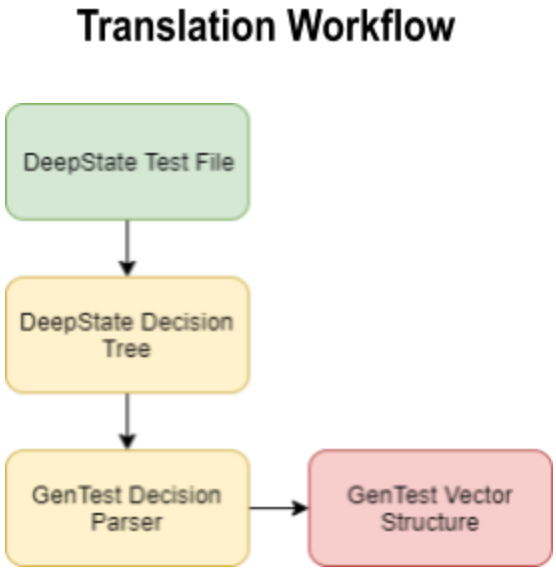


Figure 23 - Translation System Flow

Meta-programming techniques will allow GenTest to handle this. When replacing the DeepState tests with equivalent tests of another testing library, the context free grammar give this functionality.. With a combination of these two techniques, the translation engine will convert tests as well as structs into a basic form that the standalone file assembler can construct into standalone tests.

4.2.1.0 Operation

During runtime, GenTest will capture the execution process generated by DeepState and store it into the LinkedList class. This linked list will contain a linear flow of tests generated by DeepState, as opposed to the abstract nature of DeepState itself. DeepState is abstract in nature because during runtime it can randomly generate a set of tests, meaning that each runtime of DeepState could contain a unique set of tests. On creation of the translation engine, it will require knowledge of the Testing library to convert to, as well as the DeepState file that it needs to convert. It will then utilize this knowledge to convert the DeepState test file into a format that can be run using the new testing library.

4.3.0.0 Binary Parser

Another critical component of GenTest is a binary parser. This parser will take in binary files (foo.test) generated by DeepState and parse them into primitive data types by using the tools discussed in **section 3.3.0.0**. This will allow for the use of values generated by the fuzzing techniques found in DeepState to be applied to the standalone test file, and utilize those values in each individual test.

4.3.1.0 Operation

GenTest will take a binary file, or directory containing binary files, using the C++ Standard Library class bitsets, and converts them to primitive data types to be used in the standalone tests. These values will now be casted into the correct data type. Afterwards, these values are stored in the vector class alongside the tests generated in **section 4.4.0.0**. An example of the binary parser flow is in *Figure 24* on the next page. The Figure shows the flow of the binary parser, starting with the conversion. The conversion is required as the values passed into the binary parser is unstructured binary data. After conversion, the values will then be casted (if necessary) into the correct data type. After casting, this value will be stored into the Vector, for later use in the Standalone file assembler seen in Section 3.5. On creation of the binary parser, it will require knowledge of the binary test files, as well as the LinkedList created in the translation engine.

After finding the desired value, the parser will store the value into the linked list node containing the test that the value is for.

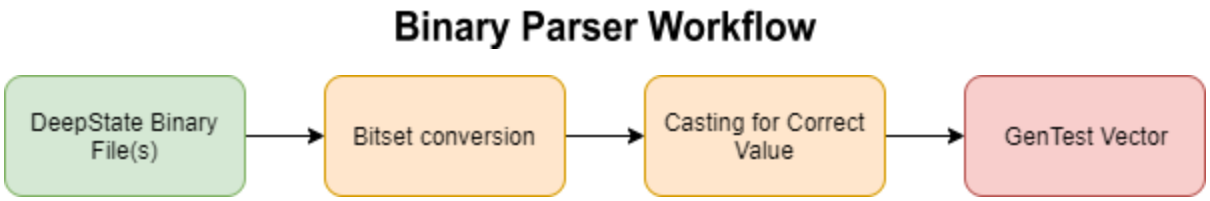


Figure 24 - Binary Parser Flow

4.4.0.0 Vector

GenTest's vector class is a vital step in the translation from DeepState test files into a standalone test file. In our implementation, the vector will be used as a device to pass data from the binary parser and the translation engine into the standalone file assembler. This will allow for a linear construction of the standalone test file, while also giving the data for each individual test.

4.4.1.0 Operation

On creation of the vector, The first test will be passed in, along with its values. For each index of the vector object, it will store a data type containing a list of values, as well as the test that those values are for. Afterwards, this vector object will be passed into the standalone file assembler. Figure 25 is an example of how the vector class and each individual index will be structured. Each test case will be a struct, which will allow for structuring of the data that comes from the binary parser, as well as the translation engine.

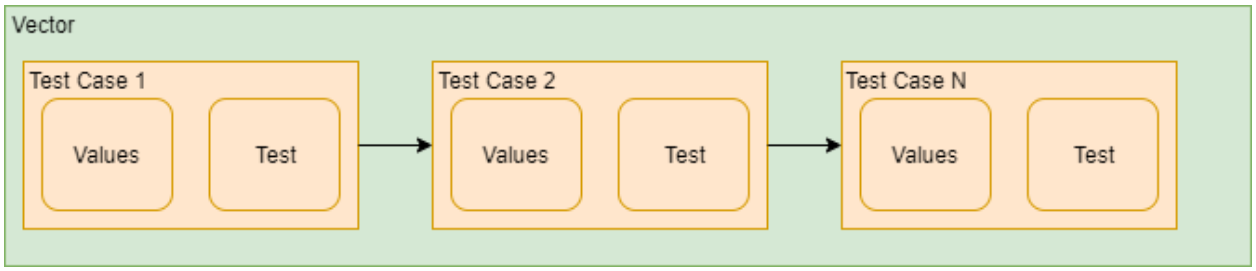


Figure 25 - Layout of vector class including test case struct.

4.5.0.0 Standalone File Assembler

At this point, all the required knowledge to generate a standalone test is stored in the vector class discussed in **section 4.4.0.0**. The File Assembler will take in the vector generated and populated by the translation engine and binary parser and output it to the standalone test file. This will be performed iteratively through each individual index of the vector class.

4.5.1.0 Operation

The standalone file assembler will iteratively crawl through the vector object, pulling the tests and values out and constructing a standalone test. When converting the tests, the standalone file assembler will need knowledge of what testing library is being converted to. During conversion, the standalone file assembler will also need to construct potential structs or data types for proper testing. After all this, the standalone file assembler will write these tests to a standalone test file with the name being given at runtime. *Figure 26* shows how this iterative process will proceed.

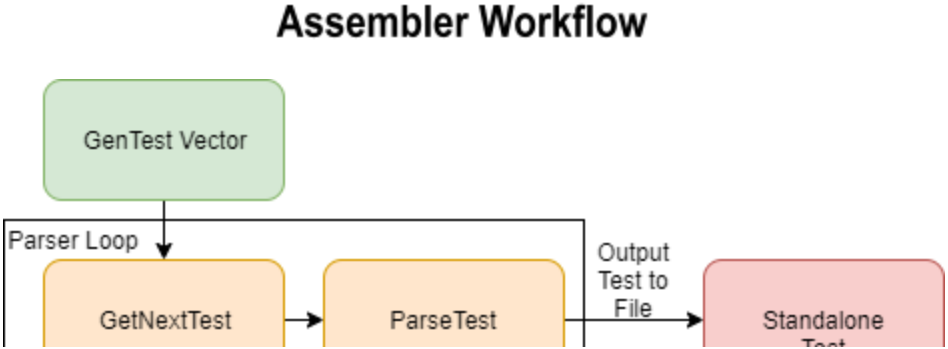


Figure 26 - Standalone test assembler operation.

4.6.0.0 Conclusion

At the present point, GenTest faces several major technical challenges. While each problem has specific knowledge-areas, it is important that each of these domains can be effectively integrated into a working system architecture. This section covered some of the issues that GenTest may encounter while undergoing such a process, as well as discussed what modules would be needed to support a working system. Through these modules, GenTest is confident that a functional and maintainable solution for DeepState's problem can be created.

Section 5 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 analyze the major technical challenges of this project and discuss potential solutions to the identified technical problems. Through in-depth research and analysis of said challenges, viable solutions were identified for these technical needs. Below is a table which summarizes these findings by technical challenge, identified solution, and confidence level:

Summary of Technical Challenges and their Solutions

Technical Challenge	Solution	Confidence Level
Harness File Translation	Contex-Free Grammars	**Moderate**
Auto-Generated Binary Encoding	Bitsets	**Confident**
Structure and Nested Structure Support	Meta-Programming	**Moderate**
DeepState Decision Capture	Vectors	**Confident**

GenTest is confident that the proposed solutions to the above technical challenges will be effective. More questions need to be answered for Structure and Nested Structure support because of the lack of introspection in C++. Similarly, the use of context-free grammar as a method of syntax generation warrants more prototyping. However, GenTest is firmly convinced that the proposed solutions above will provide an effective means of building a solution to our client’s problem.

References

---

[1] Chuang, Tyng Ruey, et al. “Non-Intrusive Object Introspection in C : Architecture and Application.” *Proceedings of the 20th International Conference on Software Engineering*, 19 Apr. 1998, pp. 312–321. *IEEE*, doi:10.1109/icse.1998.671360.

[2] Singh, Hermanpreet. “Introspective C++.” *Virginia Tech*, 2004.

[3] Koskinen, Johannes. “Metaprogramming in C++.” *Penn State University*, 9 Mar. 2004, citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.664.1801&rep=rep1&type=pdf.

[4] Agarwal, Harsh. “Scope of Variables in C++.” *GeeksforGeeks*, 1 Aug. 2018, www.geeksforgeeks.org/scope-of-variables-in-c/.

[5] Parlante, Nick. “Linked List Basics.” *Stanford University* , 1998, cslibrary.stanford.edu/103/LinkedListBasics.pdf.

[6] Rowell, Eric. “Know Thy Complexities!” *Big*, www.bigocheatsheet.com/.

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

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] Alija, Nexhati. “Justification of Software Maintenance Costs.” *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 7, no. 3, 2017, pp. 15–23., doi:10.23956/ijarcsse/v7i2/01207.

[15] Le, Duc Minh, et al. “Relating Architectural Decay and Sustainability of Software Systems.” *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016, doi:10.1109/wicsa.2016.15.

[16] Guegant, Jean. “Compile-Time Type Introspection Using SFINAE.” *Jeguegant.github.io*, Jet Brains, 2017, jguegant.github.io/blogs/tech/images/SFINAE-compile-time-introspection.pdf.

[17] Kariya, Abhirav, and Harsh Agarwal. “Access Modifiers in C++.” *GeeksforGeeks*, GeeksforGeeks, www.geeksforgeeks.org/access-modifiers-in-c/.