

DOE FY18.2 SBIR Topic 30d (Modeling and Simulation)

Phase I Final Report: DRAFT

An Extensible Verification and Validation Library with NEAMS Workbench Integration

Contract # DE-SC0018728

Ben O'Neill, PI Gerald Sabin,

¹ RNET Technologies

240 W Elmwood Dr

Dayton, OH 45459-4296

boneill,gsabin@rnet-tech.com

These SBIR/STTR data are furnished with SBIR/STTR rights under Grant No. DE-SC0018728. For a period of four (4) years after acceptance of all items to be delivered under this grant, the Government agrees to use these data for Government purposes only, and they shall not be disclosed outside the Government (including disclosure for procurement purposes) during such period without permission of the grantee, except that, subject to the foregoing use and disclosure prohibitions, such data may be disclosed for use by support contractors. After the aforesaid four-year period, the Government has a royalty-free license to use, and to authorize others to use on its behalf, these data for Government purposes, but is relieved of all disclosure prohibitions and assumes no liability for unauthorized use of these data by third parties. This Notice shall be affixed to any reproductions of these data in whole or in part.

Contents

1	Identification and Significance of the Problem or Opportunity, and Technical Approach	3
1.1	Significance and Identification	3
2	Phase I Work and Feasibility Study	4
2.1	Task 1: Develop Routines for Data Collection with ADIOS.	5
2.1.1	Injection Points	6
2.1.2	The Testing Interface	8
2.1.3	Variable Transform	11
2.1.4	The VnV Runtime Module	11
2.2	Task 2: Prototype Several VnV Tools	14
2.3	Task 3: Prototype the Graphical User Interface.	16
3	Summary and Conclusions	20
4	Publications and Presentations	21

1 Identification and Significance of the Problem or Opportunity, and Technical Approach

This is the final report for contract DE-SC0018728, a Phase I DOE SBIR entitled “An Extensible Verification and Validation Library with NEAMS Workbench Integration”. The work was performed by RNET Technologies Inc.

1.1 Significance and Identification

Numerical simulations are an essential component of the R&D pipeline, with new and more powerful algorithms and packages being developed every year. For numerical simulation codes being used to design real world nuclear reactors, erroneous simulations can result in design errors that can be extremely expensive to fix, damage the environment, and ultimately result in loss of human life. As such, the verification and validation of all numerically obtained solutions is essential, especially when those solutions are going to be used in the design of real-world products. One only needs to look to the catastrophic Sleipner platform accident, where an offshore platform collapsed due to failures in the finite element simulation, to get an idea of the consequences associated with using unverified numerical simulations.

V&V is usually seen as a one-off event that occurs once the computational model has been finished. This results in developers delaying V&V, which can increase the effort and cost required to fix errors. Additionally, it is rarely the case that a computational model is not under continuous, or at least incremental, development. In that case, each time the computational model changes (be it a change in the user driven simulation or the underlying computational toolkit) all previous verification and validation of the model becomes void. Without the updated V&V the improved simulation code is often not adopted by the end users. As such, a streamlined approach to V&V is an essential component of any design pipeline that uses numerical simulations to influence real world designs.

One of the goals of the NEAMS program is to equip end-users with a robust set of high-fidelity multi-physics capabilities that can be used to inform lower-order models for the design, analysis and licensing of advanced nuclear systems and experiments. Given the high stakes nature of nuclear power generation, it is essential that all NEAMS code, including the core tools and end-user driven and written simulations, are verified and validated using industry best practices. The NEAMS group uses internal tools and processes to verify and validate its core tools (see the NEAMS Software Verification and Validation Plan Requirement (Version 0) specification [?]), but, as of yet, there is limited support for end-user driven verification and validation of simulations. The proposed V&V toolkit will be designed to full this gap, providing end-users with an automated framework for verifying and validating numerical simulations based on industry best practices.

The goal of the Phase I/II project is to develop a framework that facilitates the development of *explainable* numerical simulations. Here, the term *explainable* is borrowed from the field of artificial intelligence, where researchers are looking to address issues associated with trust in AI algorithms. In the context of V&V, we define an *explainable* numerical simulation to be a simulation that, in addition to the final solution, provides a detailed report as to how the solution was calculated and why it can be trusted. To do this, the proposed framework will provide all the functionality required to create such a simulation, including support for:

- Writing a detailed V&V plan.
- Performing Mathematical and algorithmic testing (convergence analysis, mesh refinement studies, method of manufactured solutions, etc.).

- Verifying and Validating a broad benchmark testing suite.
- Performing Uncertainty quantification and sensitivity analysis.
- Comparing of simulation results with experimental data and results from third party simulations.
- Automatic documentation of the V&V effort.

The Phase I project focused on prototyping an automated V&V framework for numerical simulations that are written and driven by end-users. In particular, the Phase I effort was directed toward developing effective techniques for injecting the required functionality into general purpose numerical simulation packages. To that end, this report will introduce the reader to *VnV*: a self describing testing framework that facilitates in-situ V&V in advanced numerical simulations with the following functionality:

- Cross library and multi-lingual support for run-time test injection in numerical simulations.
- Efficient data output and analysis in HPC settings.
- Support for modular, generic testing libraries that can be configured at runtime.
- A simple XML configuration file that allows users to fully control the V&V process without re-compiling.
- An Automated system for generating V&V reports with support for advanced data visualization techniques.

2 Phase I Work and Feasibility Study

The result of the Phase I project is a prototype of the *VnV* framework. As we will show, this framework provides all the core functionality required to facilitate in-situ V&V and automated documentation generation in numerical simulations.

The requirements set by the project team for the Phase I prototype were:

- Cross library support. Most modern numerical simulations rely on a deep hierarchy of numerical simulation tools. For example, BISON, the fuel performance code in the NEAMS toolkit, is built using MOOSE; MOOSE uses libMesh; libMesh uses PETSc; PETSc uses hypre; and hypre uses BLAS. Thus, it is a requirement of the prototype that the system should provide a single interface for in-situ testing in any library linked to the final executable.
- Multilingual support: The big three computing languages in high performance computing are C, C++ and FORTRAN. To ensure the system has wide applicability, it is essential that users be able to use the framework in software written in any one of these languages.
- Run time configuration: Users should be able run V&V tests without needing to recompile either the executable or any of the linked libraries. The tests themselves should be completely independent from the source code and configurable at runtime.
- Simple integration: The injection point system should be simple to integrate into existing applications.
- The framework should automatically generate a highly customizable, publication grade V&V report for each simulation.

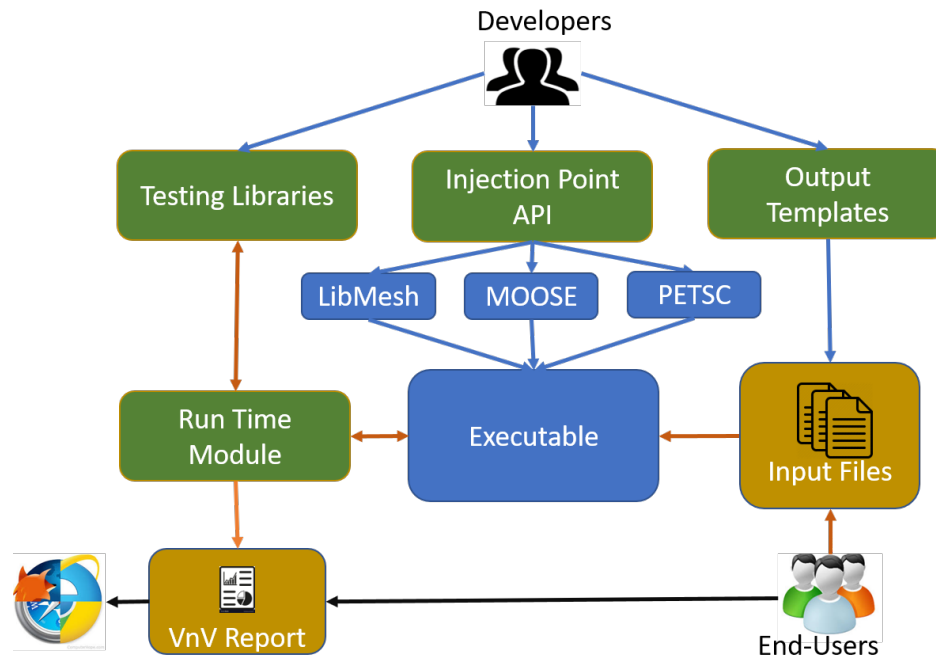


Figure 1: The VnV toolkit development lifecycle. Here, green boxes represent core VnV functionalities. Developer interactions are shown in blue, runtime interactions are shown in orange and post-processing interactions are shown in black.

These requirements were chosen because they represent the core functionality required to facilitate the development of explainable numerical simulations. That is to say, in demonstrating this functionality, the project team will be able to prove the feasibility of the proposed approach in equipping general purpose numerical simulation toolkits with built in support for end-user solution verification and validation.

Figure 1 provides an example of how developers and end-users will interact with the VnV toolkit. In this case, we show an example of how the toolkit functionality might be implemented in the MOOSE toolchain. The first step in the VnV development life-cycle is to specify and describe the injection points. These injection points will be placed at key locations of the code where testing can and should take place. In the Phase II product, inserting an injection point into a code will be as simple as annotating some variables for inspection and calling an injection point function. In addition to adding the code, developers will also complete an output template that will be used in the final V&V reports to describe the state of the simulation each time that injection point is met. The VnV tests are developed in external libraries and hence, can be developed either by the developer of the simulation or by the end-user of the code. The core framework will also include a robust set of general purpose V&V tests. We envision that the developers of a numerical simulation package will ship the library with hard-coded injection points, a set of custom V&V tests and a number of VnV configuration files.

2.1 Task 1: Develop Routines for Data Collection with ADIOS.

Work on task one began with a through investigation into the optimal approach for allowing injection points to be inserted into existing code bases. The goal of the task was to provide a simple mechanism for facilitating data collection and testing that required minimal changes to the build system, had small overheads when testing was turned off, and that which could be removed from the final executable during compilation if need be. The project team investigated several approaches to doing this including the development of a custom C pre-processor and an investigation into binary instrumentation with the Dyninst

API.

In the end, it was decided that the best approach for performing V&V data collection and testing inside existing simulations was to produce a framework that allowed the developer to hard-code injection points into the original source code. This approach is maintainable (the injection point specifications exist in the code-base), portable (it uses only standard C functions) and efficient (pointers are used to avoid copying the data). In what follows, we describe the framework developed to support data collection and V&V testing in existing simulations.

2.1.1 Injection Points

At the core of the framework are injection points. Injection points represent locations in a code where V&V testing can take place. Inserting an injection point into an existing function is a simple, three step process; (1) include the “vv-runtime.h” header file, (2) place injection points in the code and (3) write the injection point specification.

Declaring an injection point is as simple as calling the INJECTION_POINT macro. The format for this macro is:

```
INJECTION_POINT(<name>, <stage>, <type> <variable>, ...)
```

The macro is expanded to a variadic C function during preprocessing. Here, the unique name represents the id that will be used to define the injection point in the configuration files and the final reports. This name must be unique across all injection points in an executable.

The stage parameter is an integer value that defines the step that this injection point belongs to. Using this parameter, the developer can set up multi-staged injection point testing. Tests defined on staged injection points stay in scope across all stages allowing tests to collect data across multiple code locations. The Phase I prototype supports up to 8000 stages at each injection point, although we have yet to come across a reason for an injection point with more than three or four stages. There are a few restrictions on the stage parameters to ensure efficient data output. First, a single stage injection point should be specified with a stage parameter of -1. A stage parameter less than 1000 should only be used to represent the starting point of a staged injection point. Likewise, a stage parameter greater than 9000 should be used to indicate that this is the last stage in the staged injection point. This allows a staged injection point to have multiple entry and exit points, while also allowing us to automatically handle recursion of injection points.

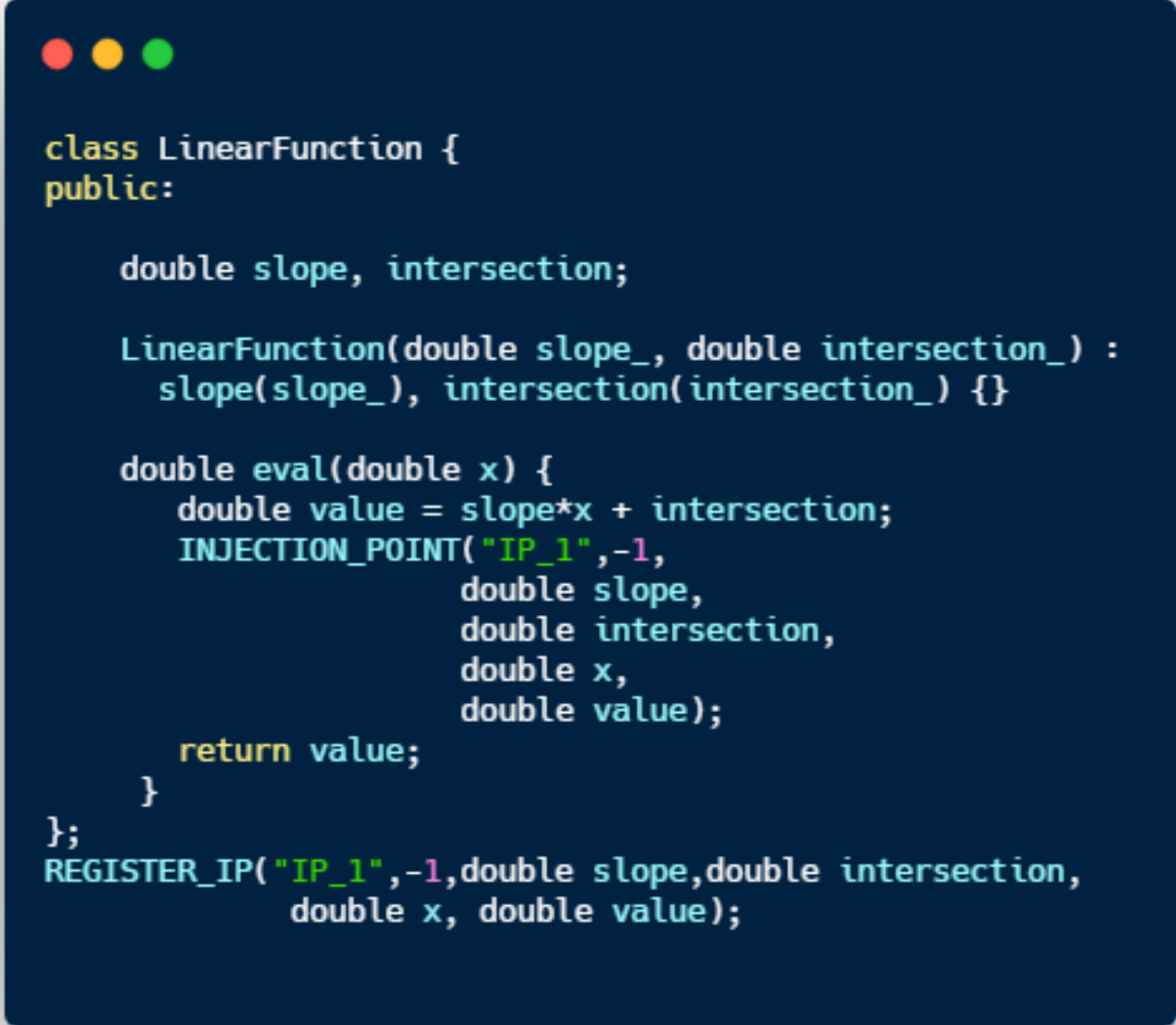
The macro supports up to 25 separate variables; although we have not come across cases where more than 5 variables are required. Specifying a variable is as simple as writing its type and name. During preprocessing, the macro expands each variable-type declaration as

```
..., <type> <name> , ... --> ..., "<type>", (void*) &name , ...
```

The runtime module of the framework then maps that void* pointer and the string based type specification to the user specified tests for processing. This is an obviously risky approach that, when implemented incorrectly, could lead to memory corruption errors. For example, it would be easy for a developer to change the type of the variable in the code, but forget to update the string in the injection point declaration. A key goal of the Phase II work will be to develop a custom pragma directive that automatically detects the correct type for each variable.

Figure 2 shows a class for evaluating a linear function that has been enhanced with a single stage VnV injection point (the stage parameter is -1).

In C++ codes, the developer can also opt to pre-register the injection point with the runtime module using the REGISTER_IP macro. This macro uses a feature of object orientated programming languages



```
class LinearFunction {  
public:  
  
    double slope, intersection;  
  
    LinearFunction(double slope_, double intersection_) :  
        slope(slope_), intersection(intersection_) {}  
  
    double eval(double x) {  
        double value = slope*x + intersection;  
        INJECTION_POINT("IP_1",-1,  
                        double slope,  
                        double intersection,  
                        double x,  
                        double value);  
        return value;  
    }  
};  
REGISTER_IP("IP_1",-1,double slope,double intersection,  
            double x, double value);
```

Figure 2: A code Snippet showing a member function enhanced with a single stage injection point called “IP_1”. This injection point is declared inside a member function designed to evaluate a linear function at a particular value of x . This injection point provides access to the member variables describing the slope and intersection point of the linear function, the input parameter x that defines where the function should be evaluated and to the result that will be returned. A user could add a simple test to verify the accuracy of the value being returned and/or to assert that the slope and intersection are correct in accordance with the physical model they are attempting to simulate.

that allows code in the constructors of static variables to be executed prior to the main function. In this way, one can register certain code elements by simply defining a static variable. Self registration is particularly nice because it allows for run-time detection of the injection points present in the call-graph. That information allows for the automatic generation of a customized test configuration file for each executable. All calls to the REGISTER_IP macro should be made outside of function calls and classes, as shown in the last line of the code snippet in Figure 2. Unfortunately, static variables can only be initialized with constant expressions known at compile time in C and FORTRAN, so instead, information about injection points must be determined either by running the full simulation, or by pulling information from the injection point specification files. The custom compiler extension developed in Phase II will address this issue by hardwiring information about the included injection points into the meta-data of the executable or library.

2.1.2 The Testing Interface

The second facet of the framework is the V&V testing interface. The development of the test interface was based on the idea that tests should be loaded at runtime and defined independently of the source code. To achieve this, the test interface was built using an C++ plugin pattern. This pattern allows users to develop tests in separate testing modules that can be loaded and configured at runtime using an XML configuration file. This allows the users to add or remove tests from injection points located in any linked library without ever needing to recompile the executable or any of the libraries.

The first step in the development of a new VnV test is to create a testing library. The framework includes a library generation script that will automatically build the directory structure and makefiles required to build this library. Once the library has been initialized, the user can begin to develop individual tests.

Figure 3 shows the declaration for the IVVTest interface that needs to be implemented when developing a new test. To simplify the process of writing tests, the VnV framework includes a test generation script. If the library generation script was used, this script will be available in the src directory of the new testing library. To run this script, the user should provide a unique name for the test and a list of the names and types of parameters that will be supported by the new test. Using this information, the script generates all the boiler plate code required to implement the IVVTest interface and to handle all the required type-casting.

Implementing the interface is a three step process. First, the developer should implement the declareParameters function. In this function, the developer will list the parameter names and types that will be supported by the test. For example, the test shown in Figure 4 supports the declaration of two parameters, a double named slope and a double named intercept. The user will map the injection point parameters to the test parameters at runtime using the test configuration file.

The main function of the test is the "runTests" function. The test generation script does all the dirty work required to cast the injection point parameters to their correct object types. As shown in Figure 4, the user is then required to implement a single function of the form

```
TestStatus runTests(adios2::Engine &engine, int testStage, ...);
```

Here, testStage is an integer representing the stage that the test is currently in. A test can support as many stages as there are integers. As will be shown below, the test stages are mapped to injection point stages at runtime using the test configuration file. The ... represents pointers to the variables requested by the test in the declareParameters function.

Data output can occur at any point in the runTests functions. The core runtime module ensures that each test stage is completed in a unique ADIOS2 step. This allows for efficient compartmentalization of the test output and makes post-processing significantly easier. Data output is completed directly in



```
class IWVTest {
public:
    // Pure virtual function that must be implemented by Base Class. This function
    // defines what happens during the test. All data output should also occur
    // inside this function.
    virtual TestStatus runTest( adios2::Engine &engine, int stage , NTV &params) = 0;
    virtual void declareParameters(std::map<std::string,std::string>&parameters) = 0;

protected:
    WVTestConfig m_config;
    NT m_parameters;

    IWVTest(WVTestConfig &config);
    virtual ~IWVTest();
    TestStatus _runTest( adios2::Engine &engine, int stageVal, NTV &params);

    template <typename T>
    T* carefull_cast(int stage, std::string parameterName, NTV &parameters );
};
```

Figure 3: The IVVTest Interface. The framework includes a test generation script that generates all the boiler plate code required to implement this interface. All the developer needs to do is implement the "declareParameters" function and the "runTests" function.

```

#include "injection.h"
#include <math.h>

class LinearTest : public IVVTest {

    bool valid = false;

    void declareParameters(std::map<std::string, std::string> &parameters) {
        parameters.insert(std::make_pair("slope", "double"));
        parameters.insert(std::make_pair("intersect", "double"));
    }

    // The actual testing code. In this case we check if the slope and intersection
    // point are valid (>0) and write the values to file.
    TestStatus runTest(adios2::Engine &engine, int stage, double* slope, double* intersect) {

        valid = ( slope > 0 && intersect > 0 ) ? 0 : ( slope <= 0 ) ? -1 : 1;
        engine.Put("slope", *slope);
        engine.Put("intersect", *intersect);
        engine.Put("valid", valid);
    }

    void declareIO(adios2::IO &io) {
        io.DeclareVariable<double>("slope");
        io.DeclareVariable<double>("intersect");
        io.DeclareVariable<int>("valid");
    }

    // BoilerPlate Code Automatically generated by the test generation script.

    TestStatus runTest(adios2::Engine &engine, int stage, NTV& parameters ) {
        double* d0 = careful_cast<double>(stage, "slope", parameters);
        double* d1 = careful_cast<double>(stage, "intersect", parameters);
        int testStage = m_config.getStage(stage).testStageId;
        return runTest(engine, testStage, d0, d1);
    }
}

//More boiler plate code generated by the test generation script. This code registers
// the test with the runtime module when the shared library is loaded.

extern "C" {
    IVVTest* LinearTest_maker(VVTestConfig &config) {
        return new LinearTest(config);
    }
    void LinearTest_DeclareIO(adios2::IO& io) {
        LinearTest::DeclareIO(io);
    }
};

class LinearTest_proxy {
public:
    LinearTest_proxy(){
        // Register the test with the factory
        VV::test_factory["LinearTest"] = std::make_pair(LinearTest_maker, LinearTest_DeclareIO);
    }
};

LinearTest_proxy lt_p;

```

Figure 4: An example of a custom VnV test. In this case we implement a test that checks that the slope and intersection point of the linear function are positive and writes the result to file.

ADIOS through the ADIOS2 read/write API and the `adios2::Engine`. For example, the test shown in Figure 4 writes the slope and intersect points received from the injection point to file along with an integer representing the validity of those values.

The next step is to pre declare the IO variables that the test will utilize. This is optional, however, it is considered best practice because it allows for optimizations in the handling of the meta-data inside ADIOS. The final step in writing the test is to register the test. Registration provides the core runtime module with the information necessary to initialize the test at runtime. The test generation script automatically generates the code required to do this, hence, no further input is required from the developer on this part.

2.1.3 Variable Transform

In addition to tests, the VnV framework also provides support for pluggable variable transforms that can be used to map injection point parameters into formats that can be consumed by the tests. These test modifiers are developed using the same plugin based C++ pattern used to define the tests and can be included in separate modifier libraries or as separate components in existing testing libraries. Figure 5 shows a modifier that extracts a PETSc PC object from a KSP object prior to testing. This allows the developer to call tests that work with the PC object from injection points that only pass the KSP.

2.1.4 The VnV Runtime Module

The VnV runtime module is the driving force behind the framework. This module contains all the functionality required to detect the injection points, parse the configuration file, setup the ADIOS IO engine, load the external testing libraries and run all the tests.

Configuring a simulation to use the VnV framework is a simple, four step task;

- Include the "vv-runtime.h" header file.
- Call the `VVInit` function prior to the first injection point
- Call the `VVFinalize` function before exiting the main function
- Link the VV library to the executable.

The users primary interaction with the VnV runtime module is through the XML configuration file. The full XSD specification for the input files is shown in Figure 6.

A full XML parser was built for the input file specification using `XSD2Cpp`. The parser allows for reading and writing input files based on the XSD specification and has built in support for input file validation. This makes it incredibly easy to parse and validate the XML input files.

Figure 7 shows a snippet of the XML file required to setup a test at an injection point. The first step to injecting a test is to load the test library. This is as simple as specifying the path to the library in the "testLibrary" element. This element supports the specification of multiple paths, making it very easy to include multiple libraries. In this case, we include a test library located in the tests directory called "testLib.so".

The second step to injecting a test is to define the injection point. The Phase I prototype can automatically generate an input file that contains all registered injection points; however, including a non-registered injection point is a simple task. The only required parameters for an injection point are the "name" and "markdown" attributes. The "name" attribute represents the unique name given to the injection point in the code. The markdown attribute should be the path to a file that contains the YAML based specification for the injection point (described below).

```
//Transforms allow for the transformation of injection point
//parameters prior to being passed to the testing routines. This
//minimizes the number of tests that need to be written.

//This modifier expects a petsc ksp and returns a petsc PC.
class TestModifier : public IVVTransform {

    PC pc;

    //Declare the input and output type supported by this Modifier
    std::pair<std::string, std::string> declareExpects() {
        return std::make_pair("KSP", "PC");
    }

    // The main transform function
    void* Transform(std::pair<std::string, void*> ip, std::string tp) {
        if ( ip.first.compare("KSP") != 0 || tp != "PC" )
            throw "Parameter types not supported"

        KSP *ksp = (KSP*) ip.second;
        KSPGetPC(*ksp, &pc);
        return (void*) pc;
    }

};

//Boiler Plate code generated using the IVVTransform generation
//script.

extern "C" {
    IVVTransform* TestModifier_modifier() {
        return new TestModifier();
    }
};

class TestModifier_proxy {
public:
    TestModifier_proxy(){
        VV::trans_factory["TestModifier"] = Test_Under_Sample_modifier;
    }
};

TestModifier_proxy p;
```

Figure 5: An example of a custom VnV test. In this case we implement a test that checks that the slope and intersection point of the linear function are positive and writes the result to file.

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="vv" xmlns:t="vv"
elementFormDefault="qualified">

  <xs:element name="testLibrary">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="path" type="xs:string" minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="testparameter">
    <xs:attribute name="from" type="xs:string"/>
    <xs:attribute name="to" type="xs:string"/>
    <xs:attribute name="trans" type="xs:string" default="default"/>
  </xs:complexType>

  <xs:complexType name="configparameter">
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="value" type="xs:string"/>
  </xs:complexType>

  <xs:element name="testStage">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="parameter" type="t:testparameter" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="ipId" type="xs:integer" />
      <xs:attribute name="testId" type="xs:integer" />
    </xs:complexType>
  </xs:element>

  <xs:element name="test">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="t:testStage" minOccurs="0" maxOccurs="unbounded" />
        <xs:element name="config" type="t:configparameter" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" />
      <xs:attribute name="markdown" type="xs:string" />
    </xs:complexType>
  </xs:element>

  <xs:element name="injectionPointStage">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="desc" type="xs:string" maxOccurs="1" />
        <xs:element name="parameters" type="xs:string" maxOccurs="1"/>
      </xs:sequence>
      <xs:attribute name="stageId" type="xs:integer" />
    </xs:complexType>
  </xs:element>

  <xs:element name="injectionPoint">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="desc" type="xs:string" maxOccurs="1" />
        <xs:element ref="t:injectionPointStage" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="t:test" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" />
      <xs:attribute name="markdown" type="xs:string" />
    </xs:complexType>
  </xs:element>

  <xs:element name="scope">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="t:scope" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="t:injectionPoint" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" />
    </xs:complexType>
  </xs:element>

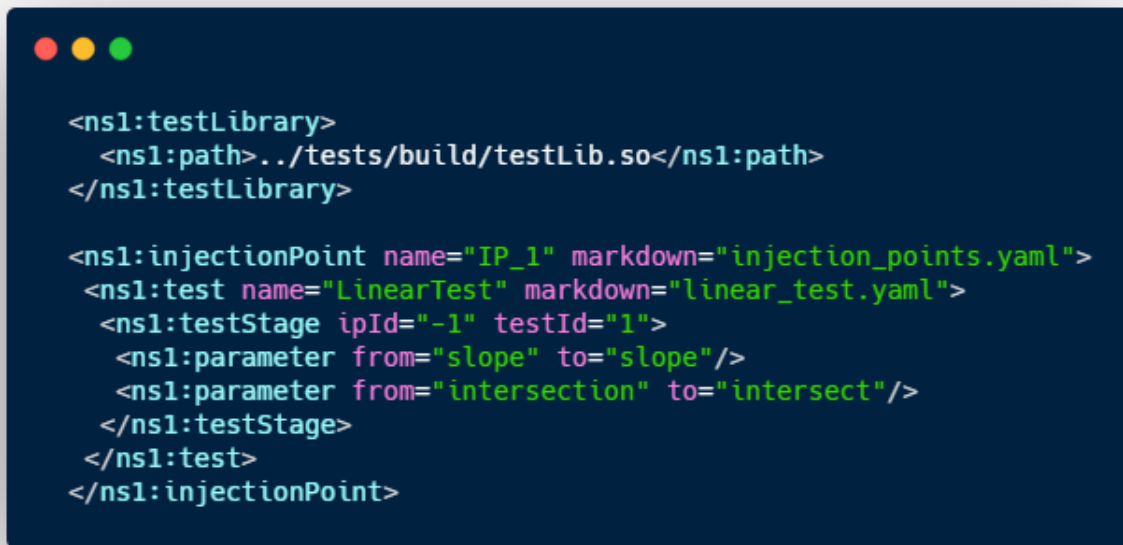
  <xs:element name="intro">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="markdown" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="outro">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="markdown" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="exe">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="path" type="xs:string" minOccurs="1" maxOccurs="1" />
        <xs:element ref="t:intro" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="t:outro" minOccurs="0" maxOccurs="1"/>
        <xs:element ref="t:testLibrary" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="t:scope" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

Figure 6: The XSD specification for the input configuration file. The VnV framework uses this specification in conjunction with Xsd2Cpp to automatically generate a fully featured XML Parsing library with support for reading, writing and validating XML files.



```
<ns1:testLibrary>
  <ns1:path>../tests/build/testLib.so</ns1:path>
</ns1:testLibrary>

<ns1:injectionPoint name="IP_1" markdown="injection_points.yaml">
  <ns1:test name="LinearTest" markdown="linear_test.yaml">
    <ns1:testStage ipId="-1" testId="1">
      <ns1:parameter from="slope" to="slope"/>
      <ns1:parameter from="intersection" to="intersect"/>
    </ns1:testStage>
  </ns1:test>
</ns1:injectionPoint>
```

Figure 7: An example of the XML configuration file for attaching test to injection points at runtime. In this case, we attach the LinearTest test shown in Figure 4 to the "IP_1" injection point shown in Figure 2.

To specify a test, the user must provide the unique name of the test and the path to the YAML based specification file for the test.

The testStage element allows the user to map the injection point stages to the test stages. This is completed using the ipId and testId parameters. In Figure 7, we attach the LinearTest test shown in Figure 4 to the single stage "IP_1" injection point shown in Figure 2.

Once the input file has been created, turning on testing using the VnV framework is as simple as calling the Initialization function with the correct filename. Runtime validation of the input file is completed at this time. For registered injection points, this includes checks to ensure the parameter mapping specified in the input file are valid. For un-registered injection points, this validation must be completed at runtime. In that case, if a parameter mapping is found to be invalid, the runtime module prints out an error and skips that test.

2.2 Task 2: Prototype Several VnV Tools

Using the framework defined above, the project team developed a small VnV testing library for inspecting the values of data stored in C++ vectors.

Figure 8 shows an example of one of the tests supported in this prototype library. In this case the test calculates the euclidean error between two C++ vectors. The output out the test is a double representing the error.

Other tools included in the library include a tool for calculating vector norms, a tool for asserting that all values in the vector are positive, and a tool for plotting the vector on a line chart. A key goal of the Phase II project will be to extend this library to support evaluations on distributed arrays. Distributed arrays are particularly tricky because the testing algorithms will have no knowledge of how the data is

```
#include "injection.h"
#include <math.h>

class EuclidianError : public IVWTest {
public:

    void declareParameters(std::map<std::string, std::string> &parameters) {
        parameters.insert(std::make_pair("data0", type));
        parameters.insert(std::make_pair("data1", type));
    }

    TestStatus runTest(adios2::Engine &engine, int testStage,
        std::vector<double> *data0, std::vector<double> *data1 ) {

        double res = 0.0;
        if ( data0.size() == data1.size() ) {
            for ( int i = 0; i < data0.size(); i++ ) {
                res += powf(data0[i] - data1[i], 2);
            }
            engine.Put("error", sqrt(res));
            return SUCCESS;
        }
        return FAILURE;
    }

    static void DeclareIO(adios2::IO &io) {
        io.DefineVariable<double>("error");
    }

    // Autogenerated Boiler Plate code
    ...

};

// Autogenerated Registration code.
...
```

Figure 8: The testing code for the beginnings of a VnV testing library for making statistical assertions regarding the data stored in distributed arrays. Amongst other things, the Phase I prototype supports calculating the Euclidean error between two vectors. The Phase II effort will look to extend this support to include a range of efficient statistical methods that can be applied to data stored in distributed arrays.

distributed across the processors.

2.3 Task 3: Prototype the Graphical User Interface.

The final task of the Phase I project was to develop a GUI for the VnV toolkit. In particular, the Phase I effort focused on developing the software interfaces required to visualize the data output during VnV testing.

After accessing the strengths and weaknesses of multiple different approaches, the project team decided to develop the VnV data visualization interface using HTML and Javascript. The primary benefit of this approach is portability - the report can be displayed in any web browser - but other benefits include interactive components, non-linear data presentation and high levels of customization. Moreover, the server-less nature of the HTML web-page allows for direct publication on any static web hosting service (github.io, AWS S3, etc.).

The goal of the Phase I project was to create an interface that allowed for automated, informative post-processing of the data output during VnV testing. To do this, the project team developed the VnV automated documentation generation system.

This system is built around the fact that simulations are built up of a large number of smaller functions, each with a specific pre-defined task and an expected result. This compartmentalization makes it very easy to write generic templates that describe what is happening at each function and during each injection point. Likewise, while the inputs to VnV tests might change, the overall structure of the outputs is constant. For example, the LinearTest shown above always outputs the slope and intersection points to file. This makes it very easy to write template specifications that can be populated with data during post-processing to produce an informative, customized report.

The VnV framework implements this functionality using a combination of the YAML file format and a custom markdown specification. Figure 9 gives an example of this YAML markdown file that provides templates for the “IP_1” injection point shown in Figure 2 and for the LinearTest test in Figure 4. The fields supported in the YAML specification for each test and injection point are:

- **title:** A descriptive title for the injection point.
- **content:** A markdown formatted string representing the content to be displayed for this injection point in the final report.
- **sections:** A map containing the content for any subsections to be displayed under the original content description. Each subsection is displayed as a collapsible child of its parents content panel and is included in the overall index of the final report.

During compilation, the automatic documentation generation script loops through every injection point recorded in the VnV output file. For each injection point, the script locates the injection point specification using the markdown attribute provided in the input configuration file, renders it in HTML, and adds it as a new section in the final report. Similarly, for each test, the script locates the test specification and renders it as a subsection in the injection point section.

The defining feature of the specification files is the support for writing the content using a custom markdown format that provides support for automating data post-processing and visualization. In particular, this custom markdown format is designed to facilitate direct interaction with the data collected during the VnV testing Phase. In this way, users can write generic markdown specifications that are automatically populated with data during rendering.

The Extended markdown format for numerical simulations, MD-XNS, was developed for the VnV project as a highly customizable system for automatically processing the results obtained from numerical

simulations into interactive, informative HTML/JS web pages. The extension itself was built using py-markdown, an open source python library for converting markdown files into HTML. In addition to standard markdown commands, the MD-XNS format supports custom post-processing commands of the form

```
[VV::<name>={...},...],
```

where name is the name of the component the user would like to insert and ... represents a dictionary of configuration options. The key feature of the MD-XNS format is that it supports direct interaction with data stored at each ADIOS step. In the markdown specification, this data is accessed through the following syntax:

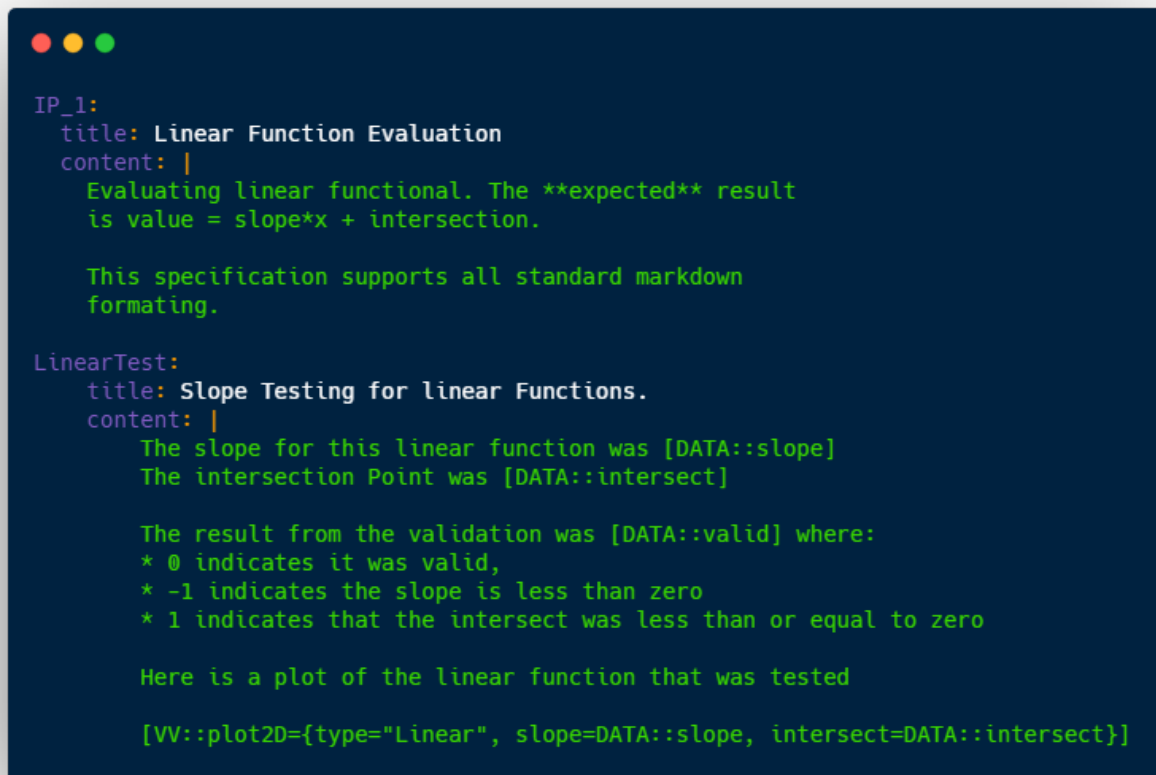
```
[DATA::<name>={...}]
```

Here, the name represents the name of the data element to be inserted. The following library allows for specific type formatting information, although that support has not been implemented in the Phase I prototype yet. The data that is available in a markdown specification depends on the injection point and test that the specification was written for. For example, the slope, intersect and valid parameters written by the LinearTest test could be queried when writing the test specification for the LinearTest test. In this way, developers of tests can write simple markdown specifications that render into highly informative, interactive data visualizations without ever having to process the data. This allows for the creation of living, breathing VnV reports that evolve as the simulation is developed. Figure 9 shows an example test specification for the “IP_1” injection point and the LinearTest test.

The Phase I prototype has limited support for a number of custom data visualization routines including

- **Table:** The table command inserts an interactive, sortable and search-able table into the final HTML document. The user can populate the table by entering the information manually, by providing the name of a csv file, and/or by using data generated at runtime.
- **Chart2D:** The Chart2D command inserts an interactive Charts.JS chart into the document. The entire array of Charts.JS charts are available through this component, including bar, line, scatter and pie charts. In each case, the chart is configured using a python dictionary entered directly into the markdown.
- **VTPView:** The VTPView command uses VTK.js to insert interactive 3D visualization of a .VTI files in the final document.
- **PostPro:** The PostPro function allows the user to set up post-processing scripts for execution during the report generation phase. In this way, users can write simple scripts that parse the data into formats more suitable for use in any of the other data visualization components.
- **ThreeJs:** The ThreeJS command provides another approach for integrating three dimensional visualization in the final report, in this case using three.js for rendering. This component is particularly useful for viewing meshes.

Figure 10 shows a screenshot of a VnV report generated using this approach. The main layout consists of two components; the index and the content. The index is generated directly from the VnV output file. Each entry in the index represents an injection point that was reached during the execution of the simulation. If a new injection point was reached during the first and last steps of another staged injection point, that injection point is listed as a child node in the index. In this way, the index represents a coarse grained view of the simulations call stack, whereby the injection points represent the nodes. The content



```
IP_1:
  title: Linear Function Evaluation
  content: |
    Evaluating linear functional. The expected result
    is value = slope*x + intersection.

    This specification supports all standard markdown
    formating.

LinearTest:
  title: Slope Testing for linear Functions.
  content: |
    The slope for this linear function was [DATA::slope]
    The intersection Point was [DATA::intersect]

    The result from the validation was [DATA::valid] where:
    * 0 indicates it was valid,
    * -1 indicates the slope is less than zero
    * 1 indicates that the intersect was less than or equal to zero

    Here is a plot of the linear function that was tested

    [VV::plot2D={type="Linear", slope=DATA::slope, intersect=DATA::intersect}]
```

Figure 9: An example YAML specification file. In this case, the test LinearTest uses the data API to extract the values for the slope and intersect provided by the injection point. The test specification then uses the support for two dimensional plotting to generate a linear plot with the appropriate slope and intersect point.

Introduction
▼ Linear Function Constructor
▼ Test 1a Function Point
▼ Test Two Function Point
▼ Conclusion

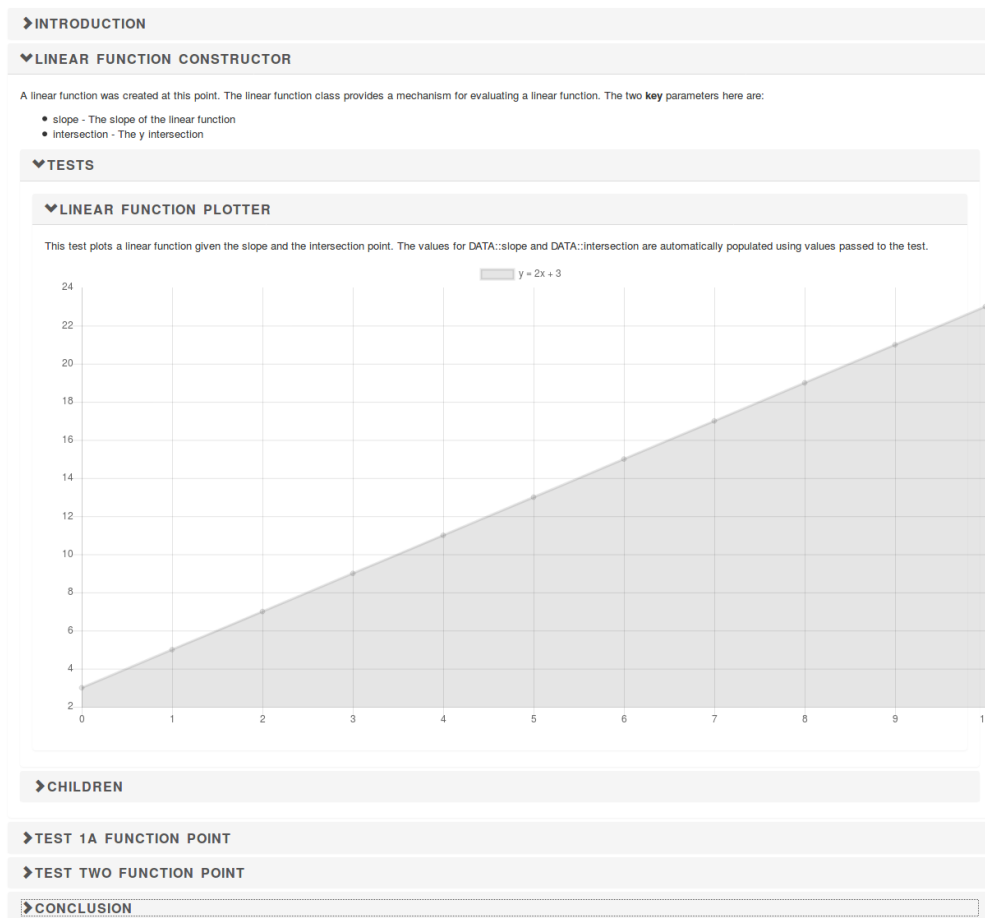


Figure 10: An example of the VnV report generated automatically from the VnV output files and the injection point and test specification files.

section is generated automatically from the YAML specification files. In this case, the user has completed a test titled “Linear Function Plotter” inside an injection point called “Linear Function Constructor”.

The key point to note here is that this interface was generated automatically from the VnV output file. Once the test and injection point specifications have been written, no user action is required to build this fully interactive VnV report. As stated earlier, we envision that developers of simulation libraries (e.g., PETSc, libMesh, hypre) would ship their codes with the injection points hard coded into the source code along with any custom VnV testing libraries. Those developers would also provide their users with the injection point specification files, the test specification files, and the input configuration files required to support end-user V&V for their library.

Using these files, the end-users of those packages, i.e., the developers of actual numerical simulations, can then form a single input file that configures V&V testing in every VnV equipt library linked to the final executable. In this way, the V&V toolkit provides a mechanism for generating explainable numerical simulations that not only provide the solution, but also a detailed report on how the solution was obtained and why it can be trusted.

The Phase I proposal had initially stated that the Phase I GUI prototype would include support for setting up input files. However, after developing the framework, and using the input specifications, it became clear that a “click” style user interface would not be all that useful for setting up the configuration files. Rather, text-editor with context aware auto-completions and validations as is present in the NEAMS workbench is likely the best option. As discussed above, the project team developed a XSD specification for the input file format. This specification contains the majority of the information required to setup context aware auto complete when building the input file. However, due to the evolving nature of the inputs; and with the further changes likely to occur in the Phase II project, it would have been highly inefficient to develop the input file GUI support in Phase I. As such, the development of a GUI for editing input files was delayed until Phase II.

The VnV report generation system was written using HTML and JS to ensure portability and applicability for a wide range of users. However, this does not limit, in any way, the ability to view the VnV reports inside the NEAMS workbench. To prove this fact, the project team created a QT application that uses the QWebEngineView to display the HTML VnV report. This QWebEngineView allows for direct interaction with the report in any QT application. In Phase II, the project team will look to improve on this integration by allowing for input files to be specified directly in the MOOSE input files using the NEAMS support for context aware auto-complete and input validation.

3 Summary and Conclusions

During Phase I of the project, RNET have tackled various technical challenges with regard to developing a modern framework for facilitating in-situ end-user verification and validation in general purpose numerical simulation packages. This capability, when fully developed, will provide a convenient interface for creating explainable numerical simulations that not only provide the user with a solution, but also a detailed report on how the solution was obtained and why it should be trusted.

In order to demonstrate feasibility of developing this product, RNET has accomplished the following.

- Developed cross library support for defining and registering injection points in general purpose numerical simulation packages.
- Created an interface for writing and integrating custom V&V tests that can be configured at runtime.
- Developed a custom markdown format that allows for automated post-processing and visualization of testing data.

- Implemented an automated documentation generation script that creates a server-less, interactive VnV report that can be displayed in any web browser.
- Demonstrated how the VnV reports could be viewed in the NEAMS workbench through the QWebEngineView.

The above evaluations demonstrate the technical feasibility of the proposed approach. This platform also has great commercial potential. RNET intends to promote the VnV framework by initially supporting relevant open-source computational tools to leverage their existing customer base.

4 Publications and Presentations

None

References