Assuring Domain Software Quality through Workflow Testing and Specification

Melody L. Hammel, Lan Lin
Department of Computer Science, Ball State University, Muncie, IN 47306, USA
{mlhammel, llin4}@bsu.edu

Abstract

In this paper, we report our experience testing the GenericModelAgent *toolkit, a critical piece of the software developed for the NSF* CyberWater *project [1], to ensure that the workflows work properly without error when deployed to the user community. The many challenges we faced include a lack of developer-oriented specifications, a complex framework, complicated and domain-specific functionality, frequently changing requirements, domain developers' unfamiliarity with TDD best practices, and poor model-view separation in a legacy software used to construct visual workflows. We present a generalized testing strategy for approaching different modules in this toolkit, in an effort to understand every input, its relation to the main functionality of the module, and every output. We then built unit and integration test cases from bottom up to enable systematic and thorough exercises of all usage scenarios identified of every module as well as their integration. Our testing discovered problems in the toolkit with suggested fixes that no one was aware of before. It also led to the creation of rewritten specifications for each module, as a by-product, that are precise and developer- and tester- (rather than user-) oriented. Our specifications, test plan, test cases and results became reusable assets for future development, testing and maintenance of the toolkit.*

1. Introduction

CyberWater is a project in development funded by the National Science Foundation (NSF) and designed to reduce user time and effort required for hydrologic modeling studies. According to *CUAHSI* [1], *CyberWater* allows hydrologists with little to no coding knowledge to integrate their models into the system and begin running simulations, allowing fundamental discoveries to be made more frequently, more easily and with less effort than before. Data flow is automated and easy access to High-Performance Computing (HPC) is provided, allowing more efficient execution and study of hydrologic data models and simulations, significantly increasing the accessibility of open-data workflows and lowering the learning curve and barrier to entry [1]. The development team consists of developers from a wide array of backgrounds, but not all of them are familiar with software engineering best practices.

One crucial component of their delivered software is the *GenericModelAgent* toolkit [4], which is a collection of modules within the *CyberWater* framework on top of *VisTrails* [2]. *VisTrails* is a GUI-based software for building scientific workflows using blocks called *modules*, which connect to each other to pass outputs to another module's inputs. These are called *output ports* and *input ports* respectively. It is typically used for data visualization and running simulations. An example workflow is shown in Figure 1.

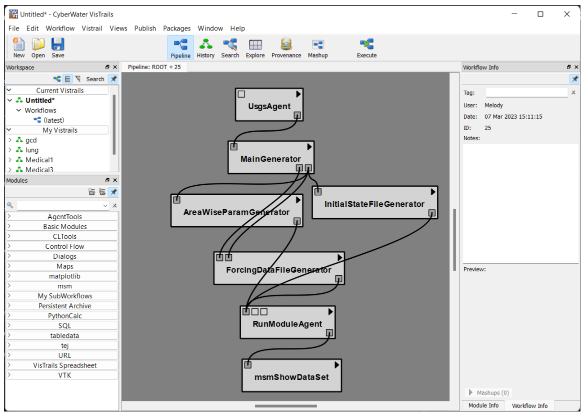


Figure 1. An example workflow in *VisTrails* using the *GenericModelAgent* Toolkit

GenericModelAgent's purpose is to allow easy integration of hydrologic simulation models that users already have into the *CyberWater* workflow without having to write any new code, with the directory structure auto-generated based on common user-derived models' requirements [4]. The modules in the toolkit we were testing have rather complex functionality; however, there was a lack of specification to explain it, an inter-connected and inter-reliant structure and the intention to be used by the end-user in *VisTrails* in a

DOI reference number: 10.18293/SEKE2023-063

combined workflow. Thus, they have functionality within each that relies on the proper functionality of the previous module in the chain, requiring a top-down approach to understand them – knowing what the previous module in the workflow does is almost always integral to understanding the functionality of the next. Though we faced this and plenty of other challenges in attempting to test these modules to ensure the quality of the final product, we were able to overcome said challenges and produce work products that became reusable testing assets, such as reworked specifications, fixed bugs, thorough unit and integration tests and test documentation, as well as insight as to how this could have been done better and made easier in the future.

2. Challenges of Testing

The *GenericModelAgent* toolkit is a critical component of the *CyberWater* framework software, providing major functionality of the advocated open-data open-model framework, and thus needs to be thoroughly tested and assured to be of good quality. When we were asked to test these modules, there were a few heuristic test cases written by the developer, but they are not documented, saved in the repository, automated, nor part of the CI/CD process.

However, with *GenericModelAgent* already deployed in users' hands and limited testing to be found, test cases had to be designed quickly and efficiently to reveal any hidden bugs in the software. Our job was to thoroughly test this toolkit, writing and automating both unit and integration tests in order to ensure the proper functionality of all the modules in this toolkit and integrate CI/CD into the workflow. In doing so, we have learned much about the design of this toolkit and have helped to improve it, both in code quality and in catching hidden bugs.

As the toolkit is already in use, any change made to the code will have an almost immediate effect. Unit testing is integral, in this scenario, to ensure that no broken or less-than-quality code makes its way into a part of the toolkit that end-users will have. Even beyond that fact, however, the toolkit is built as an extension of *VisTrails* [2], legacy software which is written in an older, deprecated version of Python (version 2.7), *and* lacks model-view separation in some areas, making testing without using the GUI somewhat difficult [5].

There were a few challenges that we faced at the beginning of testing:

1. **Lack of developer/tester-oriented specifications for the modules and their functionality**

   With a lack of both developer-oriented specifications and knowledge in the field of water science, for which this software is being developed, much of the testing process for each module involved attempting to understand what it was we were testing *for*. We needed to delve deep into the outputs of the module as compared to the inputs. This led us to find that the data and inputs of the modules were much more complicated than we had previously thought.

2. **A large, complex, inter-woven framework upon which these modules are built**

   This toolkit was designed heavily around the *MSM* package [6], legacy software and a predecessor upon which the *CyberWater* framework was built and which has a winding, interconnected set of modules and functionalities upon which *GenericModelAgent* was based. That means *fully* understanding *GenericModelAgent* requires also fully understanding *MSM* – though this was not within the scope of our testing.

3. **Connections between modules and the complication of their functionality**

   Similar to *MSM*, this toolkit involves a set of modules which are closely related to each other, given that their purpose is to be used in tandem in a workflow in *VisTrails*. Thus, they all assume functionality of each other, where a later module in the workflow will presume that the previous module has done something special – editing one module's functionality could potentially require changing *all* the modules' functionalities. The functionality in itself is complicated as well, given that it spans across modules – the high level idea is hidden unless one understands the domain specificity of the intended use of this toolkit.

4. **Extremely domain-specific functionality**

   The functionality of these modules is very domain-specific. Most of the functionality that does not revolve around setting up a directory structure involves processing hydrological data, with all the data being passed around in structures referred to as *datasets*. Given that data processing is unavoidable and the specifications we received didn't go deep into the data, understanding the structure of it and what each point of it meant was integral to testing these modules, as we needed to create consistent test data for the modules to work with.

5. **Frequently changing functionality in the midst of testing**

   Since this toolkit is both actively in development *and* deployed and in use by end-users, we also found that the functionality was changing, even if slightly, throughout the course of our testing of these modules – even in subtle ways, such as changing the way the output data is formatted, required us to rework our test cases around the new intended functionality. With a lack of specification as to what the output of these modules was really supposed to look like, the new, updated functionality was stated as a fact and our test cases and design had to be adapted to the new version of the modules.

6. **Lack of development using TDD, and what testing that did exist was heuristic and not thorough**

   The things we know as software engineers to be best practices such as Test-Driven Development (TDD) [3] and proper Git flow would have made the whole process smoother – tests would have been developed for the modules as the modules themselves were developed, essentially creating a specification for themselves. Testing that did exist was primarily heuristic. The developers were unfamiliar with unit testing, and often misunderstood what it was that we needed to perform black-box unit testing. Similarly, proper Git flow was rarely used, and the rapidly-changing functionality was made more difficult to keep track of.

7. **Lack of model-view separation in *VisTrails*'s module design**

   VisTrails is legacy software. Its design of the modules lacks proper model-view separation, making testing in the back-end difficult – similarly, the *GenericModelAgent* toolkit is primarily visual software, designed around processing data to be visualized by the *MSM* package. Thus, we were tasked with automating unit testing in the back-end for software designed to be a visual workflow – our test design had to be generalized to extend to any type of workflow, allowing us to ensure the quality of both our methods and of the software under test.

Despite all the challenges, we accomplished what we set out to do. We developed a systematic approach to testing every individual module in this toolkit, documented our solution and produced meaningful results to aid in the development as well as to improve the quality moving forward, which we elaborate in the next section.

3. Our Solution to the Testing Problem

The *GenericModelAgent* toolkit, as shown in Figure 2, consists of five modules:

- `MainGenerator`
- `AreaWiseParamGenerator`
- `InitialStateFileGenerator`

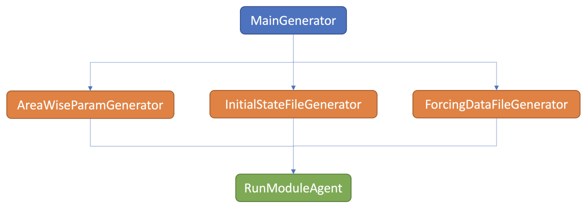


Figure 2. The modules in the *GenericModelAgent* toolkit

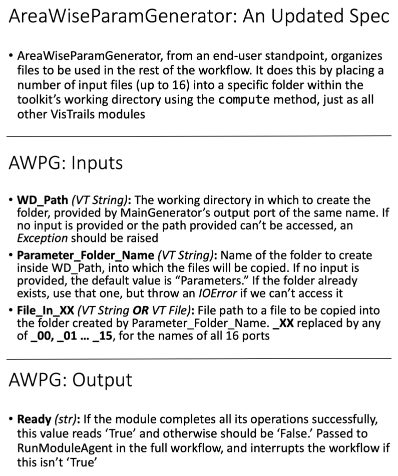
- `ForcingDataFileGenerator`
- `RunModuleAgent`

The workflow is designed around the concept of running a user's "model." Typically, this represents a compiled, executable file either downloaded or written by the user, which is designed to take in some input data and output new data into files with identical structure to the input files. The goal of *GenericModelAgent* is to simplify the process of setting up the environment needed to run the model, allowing the user to get their model up and running quickly and without having to write any glue code. The model is easily integrated using the *MSM* package to visually represent the data output by the model's simulation [4].

When beginning testing of this toolkit, we had already tested the first module in the workflow, `MainGenerator`, so we already had a decent idea of how testing of the rest of the workflow would go. `MainGenerator`, however, was a comparatively simple module – we consider it in greater detail in [5]. "Generator" modules typically have simpler but limited functionality, revolving around setting up the directory structure. A "Generator" module's output `WD_Path` will be used as the input to all other modules' inputs of the same name, providing the working directory of all the functionality in the workflow.

One by-product of our testing was the production of precise specifications, both in the form of an overview using one or more visual diagrams displaying the overall functionality and inputs/outputs of the module as well as detailed written developer documentation, as shown in Figure 3. In essence, it serves as a guide for future specifications to be written and a good reference for what to test when writing unit tests for these modules. These specifications were developer-focused and low-level, to enable black-box testing without exposing any implementation details.

We developed a strategy that we applied to all the modules in the toolkit in order to understand the environment in which they work and their functionality. We first began by carefully considering every input and output that the module can receive and generate – this extends beyond just input and output *ports* and to other things that the module



AreaWiseParamGenerator: An Updated Spec

- AreaWiseParamGenerator, from an end-user standpoint, organizes files to be used in the rest of the workflow. It does this by placing a number of input files (up to 16) into a specific folder within the toolkit's working directory using the `compute` method, just as all other VisTrails modules

AWPG: Inputs

- **WD_Path** *(VT String)***:** The working directory in which to create the folder, provided by MainGenerator's output port of the same name. If no input is provided or the path provided can't be accessed, an *Exception* should be raised
- **Parameter_Folder_Name** *(VT String)***:** Name of the folder to create inside WD_Path, into which the files will be copied. If no input is provided, the default value is "Parameters." If the folder already exists, use that one, but throw an *IOError* if we can't access it
- **File_In_XX** *(VT String **OR** VT File)***:** File path to a file to be copied into the folder created by Parameter_Folder_Name. **_XX** replaced by any of **_00, _01 … _15**, for the names of all 16 ports

AWPG: Output

- **Ready** *(str)***:** If the module completes all its operations successfully, this value reads 'True' and otherwise should be 'False.' Passed to RunModuleAgent in the full workflow, and interrupts the workflow if this isn't 'True'

Figure 3. A snippet from our rewritten `AreaWiseParamGenerator` specification

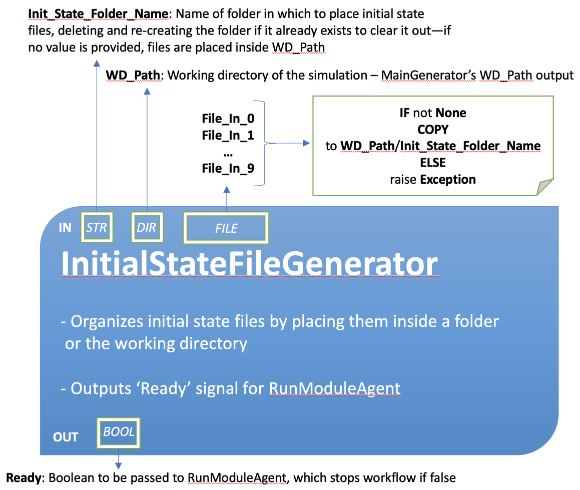

Init_State_Folder_Name: Name of folder in which to place initial state files, deleting and re-creating the folder if it already exists to clear it out—if no value is provided, files are placed inside WD_Path
WD_Path: Working directory of the simulation – MainGenerator's WD_Path output
File_In_0
File_In_1
…
File_In_9
IF not None
COPY
to WD_Path/Init_State_Folder_Name
ELSE
raise Exception
IN
STR
DIR
FILE
InitialStateFileGenerator
- Organizes initial state files by placing them inside a folder or the working directory
- Outputs 'Ready' signal for RunModuleAgent
OUT
BOOL
Ready: Boolean to be passed to RunModuleAgent, which stops workflow if false

Figure 4. The inputs, outputs, and functionality of `InitialStateFileGenerator`

can receive: directory structure, files in a location it expects them to be, a previous module's execution as inputs; and resulting directory structure, generated files, copied files as outputs. We then examined how each one of these inputs and outputs relates to the overall behavior (both intended and implemented) of the module, asking deeper questions such as how does this input affect the output? What is *done* with any given input that allows this output and behavior to be observed? How changes in inputs bring about different outputs?

3.1. `AreaWiseParamGenerator` and `InitialStateFileGenerator`

These are two more "Generator" modules in the toolkit and, as "Generator" modules, both are similar in function to `MainGenerator` – designed around setting up the environment for the user model to run. We designed tests for success and failure of each individual function, with various types of edge cases, such as file system permissions and an improper directory structure. Keeping consistent test data, such as making sure file names and folder names are always the same, helped drastically in making the tests cohesive. The "Generator" modules (`InitialStateFileGenerator`, as an example, shown in Figure 4) only have one output port, designed to connect them together in a *VisTrails* workflow. The outputs of the modules boil down to two main things: a directory structure and raised errors.

Since *VisTrails* modules are based around their `compute()` methods, our testing strategy was to define constant test data for things not being tested, call the `compute()` method and assert that the directory structure looked how we expected it to after the `compute()` method finished. With each test, we varied only the input to the one port or aspect of behavior, shown in Table 1. We considered every scenario that could go wrong and grouped tests based on input and output ports.

3.2. `ForcingDataFileGenerator` and `RunModuleAgent`

These two are a spike in complication, both in function and in testing: they start to integrate with the *MSM* package, involving small amounts of data processing. Thus, we needed to understand the data, the outputs and what inputs should cause what outputs. This was increasingly difficult with the lack of developer documentation and specification. This pushed most of our focus onto understanding the inputs and outputs.

In order to run automated unit tests in the back-end for workflows built in *VisTrails*, we utilized *object-method replacement*, (used in the testing of `MainGenerator` [5]). To apply it to testing the modules here, we had to revamp the `get_input` method to handle "compound input ports."

The *MSM* package uses a structure called `DaoDataSet`. These are objects in Python which, at their core, represent JSON files with additional methods to operate on the data contained within. These JSON files have a deeply nested structure, shown in Figure 5, with many keys and values required for the `DaoDataSet` constructor to function. In order to test these modules which use these *datasets*, we needed to know the structure of these datasets, to create test data that will result in predictable outputs in testing. It took much experimentation and effort

Table 1. Test design for `AreaWiseParamGenerator` and `InitialStateFileGenerator`. Scenarios marked with asterisk (*) are merged into other test cases in practice.

AreaWiseParamGenerator		InitialStateFileGenerator	
Param_Folder_Name	Folder exists with write permission*	Init_State_Folder_Name	Folder exists with write permission*
	No name provided		No name provided
	Folder doesn't exist		Folder doesn't exist
	Folder exists w/o write permission		Folder exists w/o write permission
			Previous folder contents removed
WD_Path	No path provided	WD_Path	No path provided
	Path doesn't exist		Path doesn't exist
	Path exists w/o write permission		Path exists w/o write permission
	Path exists with write permission*		Path exists with write permission*
File_In_XX	No file provided	File_In_X	No file provided
	File doesn't exist		File doesn't exist
	File exists w/o read permission		File exists w/o read permission
	File exists with read permission		File exists with read permission
Ready signal (Out)	True	Ready signal (Out)	True
	False		False

as little was well-defined.

In most instances, test case design for `ForcingDataFileGenerator` had to revolve around crafting various test datasets as inputs with varying structures of data that would cause different testable behavior and asserting that it behaved in an expected way. We considered every possible varying output of the module and chose specific structures out of the wide variety of datasets to ensure correct output was produced for the given scenario.

`RunModuleAgent`'s functionality, shown in Figure 6, is based around the concept of a "model." This is assumed to be a program which can be run from the command-line with arguments, which generates files of a specific name and format to be read by `RunModuleAgent` and placed into a *dataset*. We wrote a script to run the module in our tests that had predictable outputs and wasn't computationally intensive.

3.3. The Test Model

The test model developed for testing these modules is written in C, to be compiled at run-time by a `system()` call in the Python script that runs the tests and to be called by `RunModuleAgent`. It is programmed with multiple different outputs that it can generate to test the different responses of `RunModuleAgent` and the workflow, which can be specified by the argument with which the model is run. For example, if the model is run with `multiple_columns` as an argument, the file it generates will have multiple columns – however, if its argument is `alternate_separator`, the file it generates will be delimited by commas instead of tabs. The model generates an output file in different formats depending on the argument it is run with. This model, since compiled at runtime, is also customizable, and any of its parts can be switched in and out or ignored with ease, due to the use of arguments – simply add an extra else case with a string comparison to the code. A snippet of the model's code is shown in Figure 7.

3.4. Integration

After unit testing all of the modules thoroughly and individually, integration testing was an indispensable step in ensuring the functionality and code quality of this toolkit. Previously, we had only tested these modules in isolation, making sure they produce the outputs we expect given specific inputs, but integration testing allows us to test them in a similar environment to the one they will be used in: providing each other inputs through one another's outputs.

Using Python's *Unittest* library, we tested an integration of these modules, using one module's output as the input to another module, simulating how it would be done with connectors in the *VisTrails* GUI, shown in Figure 8.

We used the test model from earlier, extending it to account for every module's outputs. This then generates an output file with specific information in it depending on which modules failed, if any, and this file is read by the Python tests, interpreted and the user is notified of any failures and the exact module that was the culprit. For example, consider `ForcingDataFileGenerator` generates a file with incorrect content compared to what we would expect based on the dataset provided. The test model will read the file, compare it against the expected output and generate its own file, appending `"fdfg"` to the output. The code is shown in Figure 7.

The Python test code will then, upon raising an `AssertionError` that the output is not what was expected, read said file, as it is now aware an error has taken place and parse the content based on the included text in the file, see that `"fdfg"` is included and raise the caught `AssertionError` again with an er-

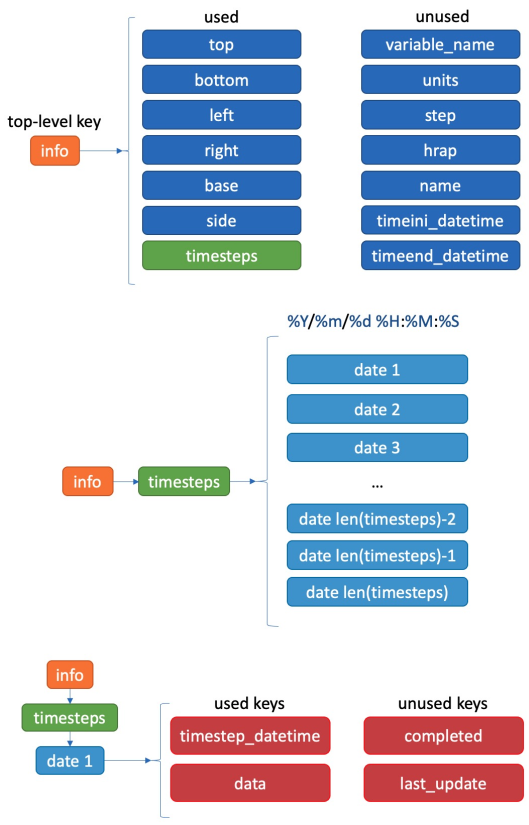


Figure 5. A snippet of the nested structure of the `DaoDataSet` JSON

ror message explaining to the end-user what went wrong: `ForcingDataFileGenerator` improperly generated its file. The code for this is shown in Figure 9.

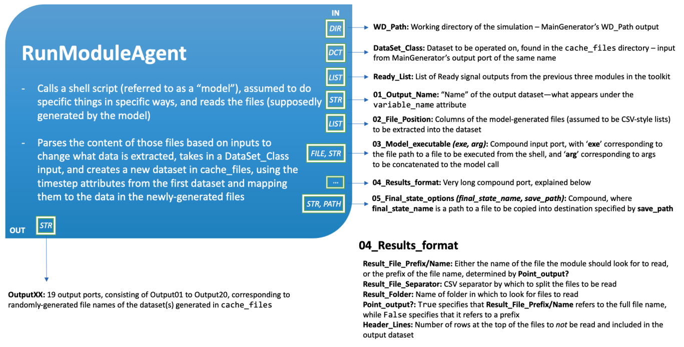
The integration testing ensures that all of the modules can "talk" to each other in ways that each module is expecting and successfully execute a workflow all working in tandem. This provides us an extra layer of confidence of the modules in their expected environment of use.

4. Testing Results

In this section, we report the major results we obtained through testing.

4.1. Test Cases and Discovered Errors

Table 2 shows the number of test cases we wrote and ran for each module in the toolkit, how many of them were successful and how many failed. These test cases allowed us to discover certain bugs that were present with all the modules. The developers were not aware of these bugs without systematic unit and integration testing described here.



- Calls a shell script (referred to as a "model"), assumed to do specific things in specific ways, and reads the files (supposedly generated by the model)
- Parses the content of those files based on inputs to change what data is extracted, takes in a DataSet_Class input, and creates a new dataset in cache_files, using the timestep attributes from the first dataset and mapping them to the data in the newly-generated files

WD_Path: Working directory of the simulation – MainGenerator's WD_Path output

DataSet_Class: Dataset to be operated on, found in the `cache_files` directory – input from MainGenerator's output port of the same name

Ready_List: List of Ready signal outputs from the previous three modules in the toolkit

01_Output_Name: "Name" of the output dataset—what appears under the `variable_name` attribute

02_File_Position: Columns of the model-generated files (assumed to be CSV-style lists) to be extracted into the dataset

03_Model_executable *(exe, arg)*: Compound input port, with '**exe**' corresponding to the file path to a file to be executed from the shell, and '**arg**' corresponding to args to be concatenated to the model call

04_Results_format: Very long compound port, explained below

05_Final_state_options *(final_state_name, save_path)*: Compound, where **final_state_name** is a path to a file to be copied into destination specified by **save_path**

OutputXX: 19 output ports, consisting of Output01 to Output20, corresponding to randomly-generated file names of the dataset(s) generated in `cache_files`

04_Results_format

Result_File_Prefix/Name: Either the name of the file the module should look for to read, or the prefix of the file name, determined by **Point_output?**
Result_File_Separator: CSV separator by which to split the files to be read
Result_Folder: Name of folder in which to look for files to read
Point_output?: `True` specifies that **Result_File_Prefix/Name** refers to the full file name, while `False` specifies that it refers to a prefix
Header_Lines: Number of rows at the top of the files to *not* be read and included in the output dataset

Figure 6. The inputs, outputs, and functionality of `RunModuleAgent`

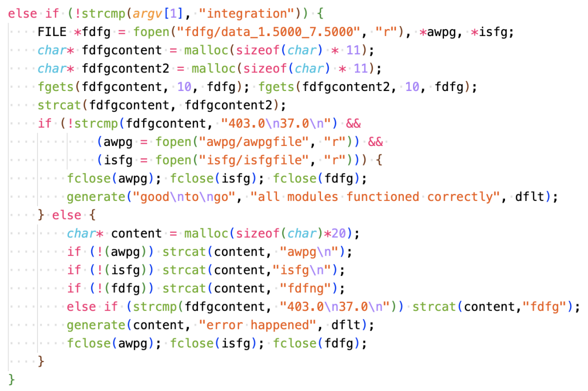

```
else if (!strcmp(argv[1], "integration")) {
    FILE *fdfg = fopen("fdfg/data_1.5000_7.5000", "r"), *awpg, *isfg;
    char* fdfgcontent = malloc(sizeof(char) * 11);
    char* fdfgcontent2 = malloc(sizeof(char) * 11);
    fgets(fdfgcontent, 10, fdfg); fgets(fdfgcontent2, 10, fdfg);
    strcat(fdfgcontent, fdfgcontent2);
    if (!strcmp(fdfgcontent, "403.0\n37.0\n") &&
            (awpg = fopen("awpg/awpgfile", "r")) &&
            (isfg = fopen("isfg/isfgfile", "r"))) {
        fclose(awpg); fclose(isfg); fclose(fdfg);
        generate("good\nto\ngo", "all modules functioned correctly", dflt);
    } else {
        char* content = malloc(sizeof(char)*20);
        if (!(awpg)) strcat(content, "awpg\n");
        if (!(isfg)) strcat(content,"isfg\n");
        if (!(fdfg)) strcat(content, "fdfng");
        else if (strcmp(fdfgcontent, "403.0\n37.0\n")) strcat(content,"fdfg");
        generate(content, "error happened", dflt);
        fclose(awpg); fclose(isfg); fclose(fdfg);
    }
}
```

Figure 7. The case for integration testing, where the model reads the files and compares them against expected outputs

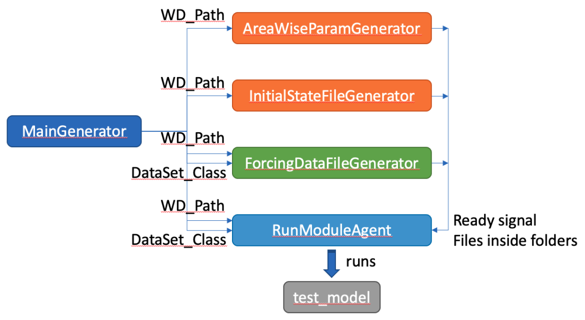
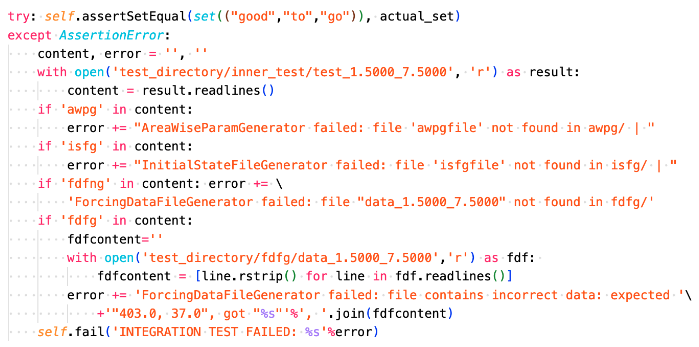


Figure 8. The module outputs used as inputs in integration testing

The most common errors that we found in the modules were ones that resulted from edge cases of the module usage, shown in Table 3. These primarily resulted from instances where a module would fail and the rest of the work-



```python
try: self.assertSetEqual(set(("good","to","go")), actual_set)
except AssertionError:
    content, error = '', ''
    with open('test_directory/inner_test/test_1.5000_7.5000', 'r') as result:
        content = result.readlines()
    if 'awpg' in content:
        error += "AreaWiseParamGenerator failed: file 'awpgfile' not found in awpg/ | "
    if 'isfg' in content:
        error += "InitialStateFileGenerator failed: file 'isfgfile' not found in isfg/ | "
    if 'fdfng' in content: error += \
        'ForcingDataFileGenerator failed: file "data_1.5000_7.5000" not found in fdfg/'
    if 'fdfg' in content:
        fdfcontent=''
        with open('test_directory/fdfg/data_1.5000_7.5000','r') as fdf:
            fdfcontent = [line.rstrip() for line in fdf.readlines()]
        error += 'ForcingDataFileGenerator failed: file contains incorrect data: expected '\
            +'"403.0, 37.0", got "%s"'%', '.join(fdfcontent)
    self.fail('INTEGRATION TEST FAILED: %s'%error)
```

Figure 9. The Python code that reads the file and informs the user of the error

Table 2. Test cases and results

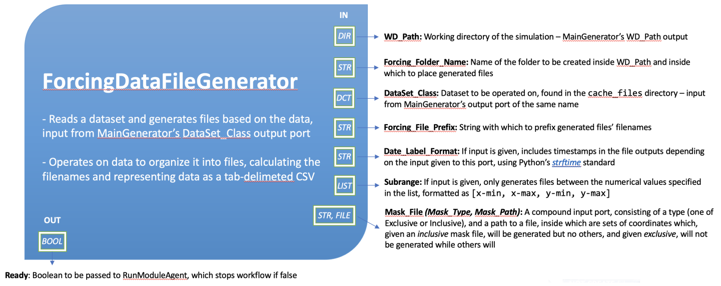
Module	Total	Failed	Passed
MainGenerator	13	0	13
AreaWiseParamGenerator	13	2	11
InitialStateFileGenerator	15	4	11
ForcingDataFileGenerator	37	5	32
RunModuleAgent	30	2	28

flow would continue, regardless, until the last module informed the user of the problem because the model it was trying to run would fail. Our test cases revealed flaws in error handling for nearly every module and, with code fixes, informed the user in a helpful way.

Our test cases allowed meaningful improvements to be made to the modules that will provide a better user experience to end-users of the software. Improved error handling will better inform the user about errors encountered in the modules (rather than cryptic Python errors). Our improved error handling in these modules allows them to better achieve their goal of simplifying the integration of a model into the *CyberWater* framework, thus improving the quality and usability of the final product.

4.2. Specifications as By-Products

A substantial, though often ignored, benefit of considering unit testing during the development of a project is the creation of good developer specifications. Specifications are an artifact that continues to be useful and evolves from iteration to iteration of the project. *Good* specifications can clear up, early in development, what features are to be in the software, and from a black-box tester's point of view, what the inputs and outputs are, how it will likely be used, all of its parameters, et cetera. Once a unit test written based off of these specifications fails, the developer immediately knows what part of the system failed, what part of the specification it violates, and what the output should look like, which can potentially kick-start bug fixing as less time is spent searching for what is *causing* the unintended behavior. For more complex aspects of these modules, the specifications included usage examples and, as a part of the specifications for each module, we also produced a diagram of its input ports and output ports with brief descriptions of their functionalities, as shown in Figures 4, 6, and 10. We developed these specifications to be, in essence, good examples of what would have been ideal to receive at the beginning of the project. They are by-products as well as reusable testing assets for future evolving iterations.



ForcingDataFileGenerator

- Reads a dataset and generates files based on the data, input from MainGenerator's DataSet_Class output port

- Operates on data to organize it into files, calculating the filenames and representing data as a tab-delimeted CSV

WD_Path: Working directory of the simulation – MainGenerator's WD_Path output

Forcing_Folder_Name: Name of the folder to be created inside WD_Path and inside which to place generated files

DataSet_Class: Dataset to be operated on, found in the `cache_files` directory – input from MainGenerator's output port of the same name

Forcing_File_Prefix: String with which to prefix generated files' filenames

Date_Label_Format: If input is given, includes timestamps in the file outputs depending on the input given to this port, using Python's *strftime* standard

Subrange: If input is given, only generates files between the numerical values specified in the list, formatted as `[x-min, x-max, y-min, y-max]`

Mask_File *(Mask_Type, Mask_Path)*: A compound input port, consisting of a type (one of Exclusive or Inclusive), and a path to a file, inside which are sets of coordinates which, given an *inclusive* mask file, will be generated but no others, and given *exclusive*, will not be generated while others will

Ready: Boolean to be passed to RunModuleAgent, which stops workflow if false

Figure 10. The inputs, outputs and functionality of `ForcingDataFileGenerator`

4.3. Improved Error Handling

The primary problems we found with the modules that we were able to discover and fix were lackluster or nonexistent error handling. In most instances, the modules were programmed with many assumptions as to the users' knowledge of the toolkit at play. It wasn't considered that a user could provide inputs in a different way; the system was unequipped to handle inputs of various structures.

Our testing was able to catch a problem with the "Ready" signal (an output) of every module, which tells the next module in the chain whether its function completed successfully. This was implemented as a way to connect the modules together in a *VisTrails* workflow, to ensure that the functionality happened in order. In some instances, the order of the code was written such that the exception would be raised before the signal could be set, but in some other instances, the signal was not considered when raising an error at all, and the signal could never be set to `False`.

Similarly, modules that created folders inside of `WD_Path` would, rather than raise an error if the folder didn't exist, indicating that something had gone wrong in the execution of `MainGenerator`, create the folder and continue execution as if nothing was wrong. This lack of error handling extended

Table 3. Software bugs revealed by testing

Module	Failed Test Case	Revealed Bug
AreaWiseParamGenerator	Ready signal outputs `False` when error occurs	Signal could never be set to `False`
	Non-existent WD_Path input	Folder was created instead of exception raised
InitialStateFileGenerator	Ready signal outputs `False` when error occurs	Signal could never be set to `False`
	Non-existent WD_Path input	Folder was created instead of exception raised
	Non-existent file input	User not informed of error
	No input files provided	
ForcingDataFileGenerator	Ready signal outputs `False` when error occurs	Signal could never be set to `False`
	Non-existent WD_Path input	Folder was created instead of exception raised
	Mask_File not formatted correctly	Exception not raised
	Input non-formatted date string	Exception not raised
	Non-default separator input	Separator remained default value
RunModuleAgent	Ready signal with `False` input stops execution	Input was not properly validated
	Non-existent WD_Path input	Folder was created instead of exception raised

to `ForcingDataFileGenerator`, where the dataset structure was very specific – despite this, the error handling for parsing the datasets was unhelpful, raising cryptic JSON errors about extracting keys that didn't exist. This was caught by our testing and improvements were made to the error handling to better inform the user of malformed datasets.

Lastly, when `RunModuleAgent` was provided a non-tab-delimited CSV file, it would fail to correctly parse columns, as it assumed the separator would always be a tab character, despite the input port designed to change the separator within the input file. The issue with the "Ready" signal also extended into this module – as it was designed to be the last module in the workflow, it takes in inputs corresponding to the signals of all the previous modules.

The test cases that we produced as a result of our work on this toolkit and the errors that they allowed us to detect in the modules' functionality will continue to provide value to the development of the *GenericModelAgent* toolkit moving forward. Our testing work has undoubtedly improved the quality of the toolkit in field use. Our test design strategy can be applied to testing other toolkits of comparable complex design and prove useful to scientists writing domain software.

5. Conclusion

Testing, without doubt, remains one of the most important and effective means to assure the quality of software, although in development for domain science the benefit of testing is easily overlooked, especially given the lack of expertise and experience. We report here our experience testing the `GenericModelAgent` toolkit, a critical component of the *CyberWater* framework, from scratch, applying unit and integration testing and back-end model-view separation in a CI/CD workflow. Our strategy in approaching workflow testing in the back-end, thorough test case design from functional requirements that addresses all the usage scenarios with edge cases, has led to a comprehensive testing documentation for this toolkit as well as bug fixes that no one on the development team would have discovered otherwise. Beyond improving the quality of the final product to be released, we also produced useful specifications of the modules under test that will benefit future iterations of development. The importance of such developer- or tester-oriented specifications cannot be overstated.

6. Acknowledgments

This work was generously funded by the National Science Foundation (NSF) under Grants 1835602 and 2209834. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF. Special thanks to Ranran Chen and Dr. Yao Liang at IUPUI for their helpful assistance in understanding the toolkit and the modules within it.

References

[1] CyberWater. `https://www.cuahsi.org/projects/cyberwater/`.
[2] VisTrails. `https://vistrails.org`.
[3] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2003.
[4] R. Chen, D. Luna, Y. Cao, Y. Liang, and X. Liang. Open data and model integration through generic model agent toolkit in CyberWater framework. *Environmental Modelling and Software*, 152:105384, 2022.
[5] L. Connelly, M. Hammel, B. Eger, and L. Lin. Automated unit testing of hydrologic modeling software with CI/CD and Jenkins. In *Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering*, pages 225–230, 2022.
[6] D. Salas, X. Liang, M. Navarro, Y. Liang, and D. Luna. An open-data open-model framework for hydrological models' integration, evaluation and application. *Environmental Modelling and Software*, 126:104622, 2020.