

University of Sheffield

A tool to specify testable causal models of software behaviour



Hao-Hsuan Teng

Supervisor: Dr Neil Walkinshaw

COM3610

A report submitted in fulfilment of the requirements
for the degree of MSc in Advanced Computer Science

in the

Department of Computer Science

May 5, 2022

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name : Hao-Hsuan Teng

Signature : Hao-Hsuan Teng

Date : 12/4/2021

Abstract

With the inventions of supercomputer with powerful processing power, a common way to utilize it is to run complex computational models that can be used to analysis the behavior or outcome of a certain event. These modules often involve large number of inputs and outputs and can be extremely time consuming to test. A tool name Causcumber that can be uses to execute the test has already been in development, but it still requires some functions to make the system user-friendly. The goal of this project is to allow users who don't have extend knowledge in software testing to be able to use the Causcumber system. This tool needs to be easy to understand, and accessible for people. A user interface with the ability to display results and create new test sets have been created, this streamed line the testing process, minimize the need for users to interact with the coding part of the testing.

Contents

1	Introduction	1
1.1	Aims and Objectives	2
1.2	Overview of the Report	2
2	Literature Survey	4
2.1	Computational models	4
2.1.1	Introduction to computational model	4
2.1.2	Purpose of computational model	5
2.2	Testing computational models	6
2.2.1	Why computational models are hard to test	6
2.2.2	Current testing methods and its disadvantage	7
2.3	Advanced testing techniques for computational models	7
2.3.1	Behavior-driven Development with Behave	7
2.3.2	Causal testing	9
2.3.3	Testing computational model with causal testing	10
2.4	Summary	10
3	Analysis	12
3.1	Project Requirements	13
3.2	Requirement analysis	15
4	Design	16
4.1	Design background	16
4.2	Design concept and process	17
5	Implementation and testing	20
5.1	Overview	20
5.2	Create and select scenario	21
5.3	Main screen	23
5.4	Edit .dot file	23
5.5	Setup background files	26
5.6	Edit .feature files	27

6	Evaluation	30
6.1	Testing influenza1918	30
6.1.1	Testing process for influenza1918	30
6.1.2	Testing result for influenza1918	35
6.1.3	Discuss influenza1918 result	35
6.2	Testing Covasim	35
6.2.1	Testing process for Covasim	35
6.2.2	Testing result for Covasim	38
6.2.3	Discuss Covasim result	38
6.3	A summary of results	38
6.4	How does the tool help user simplify the testing process	39
7	Conclusion	40
	Appendices	44
A	Testing process for influenza1918	45

List of Figures

2.1	A feature file example.	8
2.2	A feature file example.	9
3.1	A diagram analysis for the system.	12
4.1	Design concept for main menu.	17
4.2	Part 2.Design concept for edit .dot function.	18
4.3	Part 1.Design concept for edit .feature function.	19
4.4	Part 2.Design concept for edit .feature function.	19
5.1	Basic workflow for testing a model.	21
5.2	Start screen.	22
5.3	Create new scenario screen.	22
5.4	Main screen.	23
5.5	Graphviz example.	24
5.6	Add parameter to .dot file.	25
5.7	Edit parameter relation.	26
5.8	Edit first section of .feature file.	27
5.9	Edit scenario outline section of .feature file.	28
5.10	Edit scenario section of .feature file.	29
6.1	Edit graph for input parameters.	31
6.2	Edit relation for parameters.	32
6.3	Relations of parameters presented in Graphviz graph.	32
6.4	Edit initial parameters name, type and value/distribution in influenza1918.	33
6.5	Edit recorded parameters name and type in influenza1918.	33
6.6	Edit scenario for influenza1918.	34
6.7	Result produced by Causcumber display in result section.	34
6.8	Relation for parameters for testing Covasim.	36
6.9	Background for testing Covasim.	37
6.10	Result produced by using assisting tool.	37
A.1	A 95% confidence interval example with one scenario passed.	45

A.2	Creating new scenario for influenza1918.	46
A.3	Select “Edit dot files” function.	46
A.4	Graphviz for input influenza1918 parameters graph.	47
A.5	Drop-down menu for select parameter.	48
A.6	Edit background files for testing influenza1918.	49
A.7	Edit feature files for testing influenza1918.	49
A.8	Select “Select feature files” function.	50
A.9	Select newly created feature file.	50
A.10	Input and output graph for testing Covasim.	51
A.11	Select add additional edges.	51
A.12	Scenario outline for testing Covasim.	52

Chapter 1

Introduction

Throughout the years, companies have developed CPU with more and more powerful processing power, and these more powerful CPU comes the invention of supercomputer, these supercomputers have superior computational power compared to normal computers. With this computational power, people start to develop large and complex computational models. Computational model is a way to try and use computer system to simulate real life situation, these models use large amounts of parameters to determine the behavior or outcomes of a certain event, researchers can utilize these models to simulate experiments and real-life event [1] and aid their research with the results.

But these models aren't without problems, one problem is these models has large number of inputs and outputs, and it takes a lot of times to process them, this means in order to test them, it also requires a lot of time to test all inputs and outputs' accuracy with traditional method [6].

There are several advanced testing techniques or tools that have been developed, but these techniques or tools usually require specific knowledge in software testing. The solution provided by this project is to implement a user interface, this interface minimizes the need for user to interact with the programming part of the testing process. Instead of require user to interact with the code directly, it visualizes the code into a way that is easy to read for user, highlight the parts that require edit and prompt what to edit for user.

Causcumber is the tool this project mainly built upon. Causcumber is a tool that can determines the relationship between different input and output parameters and use this information to create graphs that focus on certain behaviours. Then just simply compare the difference between the test sets and the output of the tested models, people can know how accurate a certain interaction is between input and output parameters. With this method tester can saved a lot of time compared to traditional method where the best ways to test it is to run the model repeatedly with new data [19].

The problem with this method is that there's currently lack of a user-friendly way for people to access Causcumber, it requires a level of knowledge in software testing to understand how to operate this tool. With this interface, testing with Causcumber become more accessible for people who don't have extensive knowledge in software testing.

1.1 Aims and Objectives

This project will be primary based on Causcumber, a tool for testing computational model and is mainly based on a model called Covasim, Covasim is a computational model focus on COVID-19 analyses, it can determine the infection rate and how will the rate change based on different interventions (such as quarantine, social distance, etc.) [13]. Causcumber is a in developing tool and is aimed to provide a testing method for computational model like Covasim, it uses Cucumber specification, another tool reads executable specification in plain text and validates if the software does what those specifications say, to produce casual model graph.

Since Causcumber is a new system that is still in development, there isn't a convenient way for user to interact with the system. So, the primary goal of this project is to provide a user interface that can streamline the testing process, minimize the need to interact with programming code directly, allowed users to get a grasp of how to use this system more easily.

Another goal of this system is that it needs to be easy to read, and to achieve this, we use gherkin reference to help with it, by using a set of special keywords with a certain structure, it can give meaning to executable specifications. For the users who are not specialize in software engineering, this is a way to make the system easy to understand and can avoid a lot of confusion.

1.2 Overview of the Report

In the next chapter, Literature Survey, a list of literature will be providing information and reasons why there's a need for this type of testing tool to exist. It will start with explaining what scientific software is, and what are their purpose. Follow up with why they are hard to test and why they aren't usually tested in a proper method. Then provide some of the current testing methods and why those methods are insufficient. Then this will be followed up with introductions to Cucumber and Behave, two of the main factors of the Causcumber system. After that will be explain what Causal testing is and how computational model testing is could benefit from it. Finally, explaining the current flaw of this method and what

can be done to improve it.

In the third chapter, Requirements and Analysis, will be mainly talking about the requirements and objectives of the system. And what solution is possible based on these requirements

Part four, Design, will be discussing the design philosophy of the system, the design process and discuss what the system should do.

Part five, Implementation and testing, will be demonstrate the final system by introduce different parts of it and what functions do those parts have.

Part six, Evaluation, will provide two walkthrough of testing process with different models using the implemented user interface, each walkthrough will include the full testing process, the results, and a discuss about the result. Part seven, Results and discussion, will discuss what does the results mean and how does the user interface help simplify the testing process, and what can be improve with the user interface.

Part eight, Conclusions, will summarize what has been achieved in this project, provide a overview on what the final product performs.

Chapter 2

Literature Survey

This section will explain why there's a need for this system by exploring background literature related to this subject. Mainly the need of a proper system to test scientific software, start with explaining what a scientific software is, following up by explaining why most of them are not tested correctly. Then we will be discussing some of the current method in testing and why are they insufficient. Next we will be discussing the two main components of this system, Cucumber and Behave. Then finally, explaining what casual testing is, the main method used by this system to test other software, and exhibit why it lacks a proper interactive system for non software engineering users.

2.1 Computational models

Using a computational model for scientific research is common practice with today's technology, scientists use these models to predict the result of an experiment or the outcome of an event. But whether these models can accurately predict the result, is still a problem since most of them lack some form of proper testing. Below will be discussing what is a computational model, why they are poorly tested, and some of the current testing method.

2.1.1 Introduction to computational model

Computational model use computers to simulate and study real life events by using mathematics, statistics, physics, and computer science to study the mechanism and behavior of complex systems by computer simulation [1]. There are two main ways to process model's data, empirical and mechanistic, each have its own benefits.

For empirical, researcher don't need to know exactly how the system works, they just need to observe the outcome of certain scenario and predict what's going to happen in the future [10]. For example, by observing how tides are going to change in a year, people can predict how the tides will change in the upcoming years. They don't need to know the exact mechanism behind why and how the tides change, only require knowing the circumstance

and the outcome. By using this method, it simplifies the model since there's less complex calculation required. And it's better to comprehend compared to mechanistic. The problem for empirical is it require a lot more input data to have good accuracy in the model, there won't be much problem if the model is developed with big data and machine learning tools, but if the model don't have enough data at the start, it will affect the model's accuracy.

For mechanistic, it requires fewer input data to achieve the same accuracy as the empirical, but it is also more computationally intensive. Compared to empirical, it is easier to extrapolation, researchers can make predictions based on the previous input data.

In terms of the implementation of model, it is split into two types, agent based and equation-based model.

For agent-based model, is a system is modeled as a collection of autonomous decision-making entities called agents. Agents will act based on the situation its in and the abilities it has [3]. Agents with react appropriate to the rule defined for them, for example, in an agent-based model for honeybee, if a bee detect a flower near by with resource in it, the bee will move toward the flower and collect the resource. In this case, the bee agent is in a situation where there is flower with resource nearby, according to the rules define for the bee agent, it reacts by moving towards the flower and collect from it.

For equation-based model, instead of have agents with rules defined, it have a set of differential equations, the model uses equations to represent relationships between observables [23], the situation of observables will change depend on the equation. For example, modeler can define an equation on how fast a disease will spread, by define a start infected population, the equation can then calculate how many people are infected in each given timestep.

2.1.2 Purpose of computational model

By using models that contain a number of input variables, and algorithms that will define the system. Researchers use these models to simulate real-life situations and adjust these models by changing their variable and algorithm according to the results to make the model more conform to reality. Then researchers can use these models to see how one or more variables can affect the outcome of a certain event. Computational models provided some level of prediction of being able to calculate an anticipated result from a given set of variables [17]. This forecasting system can be used to predict complex systems, such as weather or disease spread, and help researchers or decision makers to decide their next move.

An example of this kind of model is Covasim [7], this model is used heavily in the implementation of Causcumber, a tool for testing in computational model that will discuss later. Covasim uses

agents to simulate individual people, the model mainly focused on one type of calculation, what is the probability of an individual in a given time step will change from not infected to infected, or badly ill to death.

The simulation starts from loaded the parameters, then it will start creating individual with different age, sex, and comorbidities based on the selected location (i.e., Different country). Then will be grouped into a social network depending on their attributes. After that the model will start looping, in each step, the model will apply various operations on the individual, then collect the result and apply analysis.

2.2 Testing computational models

Computational models are often very complicated, and require special knowledge related to the field, so is testing computational model. This often results in some issue in testing, some methods have been developed to help with the testing process, but most are either to complicate or insufficient. There's currently a more efficient way to test these models which is by using behavior-driven development (or BDD) and Cucumber. Combined with casual model testing, a way to draw relationships between variables, it will make testing these complex models a lot easier.

2.2.1 Why computational models are hard to test

Computational models are often developed by scientists themselves, but most scientists aren't software developers and may not have knowledge in some common software engineering practices, this may cost the quality of the scientific software [22]. Software testing is one of the aspects that is impacted, since the lack of knowledge in software development, lack of understanding in systematic testing is expected. Mistake can be made without notice and may affect the output of the system, causing the result to become inaccurate.

Testing a scientific software itself can be a difficult task to do, this may be the result of two types of challenges:

First is due to the software's main purpose is to predict something unknown or simulate areas even researchers have little knowledge in, but to test a software, it is crucial to know how the software should Behave, what kind of output should it show. Without these knowledge, it is hard to tell if the software is working the right way.

Furthermore, these computation models are often consisting of hundreds of parameters and complex algorithm, this means it would require lots of test case in order to test every aspect of the system, and this leads high execution time making the testing process become very

time consuming [22].

Second is that scientific software is mostly developed with scientists being the lead role instead of a software engineer, and the value of the software system is often underestimated [22]. And most scientists never went through the training in software design like software engineer [20], this means limited understanding of the testing process and not applying known testing methods. This will make the core design of the software unfriendly for effective testing.

2.2.2 Current testing methods and its disadvantage

After modeler complete the modeling process, they are required to verify the model. Despite the rarely carried out practice, verification is an important step in modeling, it helps making sure the results produced by the model is accurate enough to represent the reality or the theory of the model. Verification is done by comparing the predication of the model with actual data. The type of compared data provide will determine whether the models is validated at the pattern, point, distribution, or value level, or some combination of these [5].

This verification step can have some issue when it comes to models with multiple agents, modelers may be required to test the agent as an individual or in as a group, this is often determined by the requirement of the model. If the model's main goal is focus on the broader environment, then its agents need to be verified as a group, for example in a model for researching bee colony's behavior, the bee agents will be assessed as an entire hive instead of as individual bees. On the contrast, if the model's goal is focus on individual agent, it will need to be verify as an individual. The modelers will be required to collect data to compare it with the results of the model. This can make the verification process become very tedious and time consuming.

2.3 Advanced testing techniques for computational models

With the drawback of the current testing method, more advanced method is developed to make testing more efficient. But this isn't without any issue, advance method require tester to have specific knowledge in software testing, which is not common for scientist developed the computational model.

2.3.1 Behavior-driven Development with Behave

Behave is a python API based on behavior-driven development, where the goal is for people who don't really understand programming code to be able to participate in the testing process. Behavior-driven development is a development technique that encourages collaboration between participants in a software project [2]. Behavior-driven development or BDD, aims to have a clear understanding of the desired software behavior between stakeholders and software

developers, and communicate by writing test cases in a natural language that both sides can understand. Behave is a Python API created for this, it consists of tests written in Cucumber, another API written in natural language style.

When a test case is written, it requires a tool to read its specifications and see if the system works as the test-case specified. Cucumber is the tool that was developed for this purpose, it reads specifications written in plain text with a certain format and validates if the software does what those specifications say [15] and creates a report.

For the non-developers who want to participate in testing, they are required to create a feature file. The feature need be written in Gherkin language, Gherkin uses a set of grammar and special keywords to give plain text structure so that Cucumber could understand it, one of the pros for Gherkin is that keyword can be translated to other languages, making it usable for people who don't speak English [16]. A proper Gherkin grammar starts by giving a context, then describing an event, and after that what should happen after the event. With these grammars, testers can specify a situation in the system, describe a certain parameter, and what the results should be with those parameters. By writing a test using Cucumber and Gherkin grammar, tester can combine them into a .Feature file as display below:

```
Feature: Compare interventions
  Background:
    Given a simulation with parameters
      | parameter | value | type |
      | quar_period | 14 | int |
      | n_days | 84 | int |
      | pop_type | hybrid | str |
      | pop_size | 50000 | int |
      | pop_infected | 100 | int |
      | location | UK | str |
      | interventions | baseline | str |
    And the following variables are recorded weekly
      | variable | type |
      | cum_tests | int |
      | n_quarantined | int |
      | n_exposed | int |
      | cum_infections | int |
      | cum_symptomatic | int |
      | cum_severe | int |
      | cum_critical | int |
      | cum_deaths | int |
```

Figure 2.1: A feature file example.

For developers, they are required to create a step decorator the matches the steps, for example in the above feature file example, there is a “given” type, therefore in step decorator will need to match it as below:

```
@given(u"a simulation with parameters")
def step_impl(context):
    for row in context.table:
        var = Input(row["parameter"], locate(row["type"]))
        context.types[row["parameter"]] = locate(row["type"])
        var.distribution = eval(row["distribution"])
        context.scenario.modelling_scenario.variables[row["parameter"]] = var
```

Figure 2.2: A feature file example.

With the step file implemented, tester can execute the .feature file in terminal and Behave can read what needs to be tested and return a report. In this way, tester who don't have sufficient knowledge can involve in testing phase, therefore making collaboration between developers and non-developer in a project become a lot easier.

2.3.2 Causal testing

With the help of Behave and Cucumber, we have the ability to test computational models. It is possible to test the entire system by writing an enormous feature file, but the execution time will become really long, and since computational models are usually in big scale it is also easy to make mistakes. This is when causal model can help simplify the testing process.

Causal model is a way to represent causal relationship within a system through mathematical model. It helps testers monitor causal relationships in the data [11]. A causal module can predict the behavior of a system, by comparing different inputs and the outputs a system produces, it can explore the cause and influence of a certain relationship [4]. Tester can purposefully change a certain input to observe the change in behaviour, when change in input cause change in output, tester can observe causal relationship between the variables. When there's a difference between in executions of the system, whether its in input or the execution process, no matter how small it is, these differences can help testers identify any unusual behaviour in the system.[12]

With this method testers can reduce the time required to test a complicated system by only testing parts of it one at a time, then combine the results, and testers can have a full picture of how the system performs. This is unlike the traditional method, where it requires testing the entire system all at once, and may require lots of time if it is a complicated system.

2.3.3 Testing computational model with causal testing

Despite the existing implementation of combining Behave and causal testing, Causcumber, it still requires more assist in order to make this system more easy to approach. Causal testing can be a really useful testing method for software engineers [9], however computational models are often developed by scientists or researchers who have limited knowledge in software testing [2]. Tester who wants to use causal model to test system needs to have prior knowledge in how variables work in a software, this means people who may not have that much knowledge in software engineering, for example researchers in other fields, might not be able to conduct this method of testing in their software.

Causcumber is a system incorporate Behave and Cucumber into Causal testing, testing with Causal testing become more accessible for people who don't have specific knowledge in software testing. Without Behave, testers will need to either design inputs them self or using auto test value generation [5], but either option will require testers to have sufficient training in software testing, which are not common among scientists or researchers who developed computational model. And with Behave, user can specify the testing cases in a more natural language, making creating testing case more accessible. After creating test case for computational model, user can compare different test cases to identify the behaviour of the system and use the behaviour to identify if the subject system is behaving correctly.

But in the current version of Cucumber, it requires users to edit the test case file directly, which still require user to use a certain syntax to make sure the test cases work with the decorator. There's still a need to further streamline this process, to achieve this, we can implement a GUI to guild and assist tester to in the testing process. With this GUI, tester can use it to create test case without the need to interact with the coding part of the system. This approach can make the testing process even more accessible.

2.4 Summary

Testing a computational model isn't a easy task, not only because there can be lots of unknowns in the developing of computational model, but also because among the people who develop those software, most have limit experience in software testing. The current testing method is by gather lots of data and compare those data with the result produced by the model, this method may be useful, but can become tedious with more complex model. To make this process more convenient, several tools can be used to help simplify the process. With Behave, an API based on the behavior-driven development method, tester can specify the test case in a natural language, making test process a more approachable, encouraging people who do not specialize in software engineering to take part in testing. With causal model, testers can effectively test large and complex computational models. By using different test cases, tester can compare the inputs, outputs and the execution path of the test cases to identify the behaviour of the system. And with the behaviour, tester can verify if the

software is working as intended. But the problem is that when using causal model testing, it is inaccessible for most people, due to its requirement for user to have knowledge in software testing. To make it more user friendly, Behave is integrated to make it where users can create test case with a relatively natural language. But it still require a certain syntax to work, so a way to further simplify the testing process is needed, which leads to the development of a GUI to help streamline this process.

Chapter 3

Analysis

Computational models are often too large and complex when testing with traditional methods, this often causes developers of those models unwilling to test their models thoroughly [22]. Furthermore, computational model developers are often scientists or researchers that don't have extensive knowledge in software testing. Causal model testing combines with Behave can be a powerful tool to test computational model, but even with this method there's still require tester to learn a certain syntax. It may still be difficult to encourage people who don't have software engineering background to participate in testing.

To encourage more people to participate in testing, a way to simplify the process is needed. By reduce the need for tester to program and assist the process in creating test case, this can encourage people to test models with this tool.

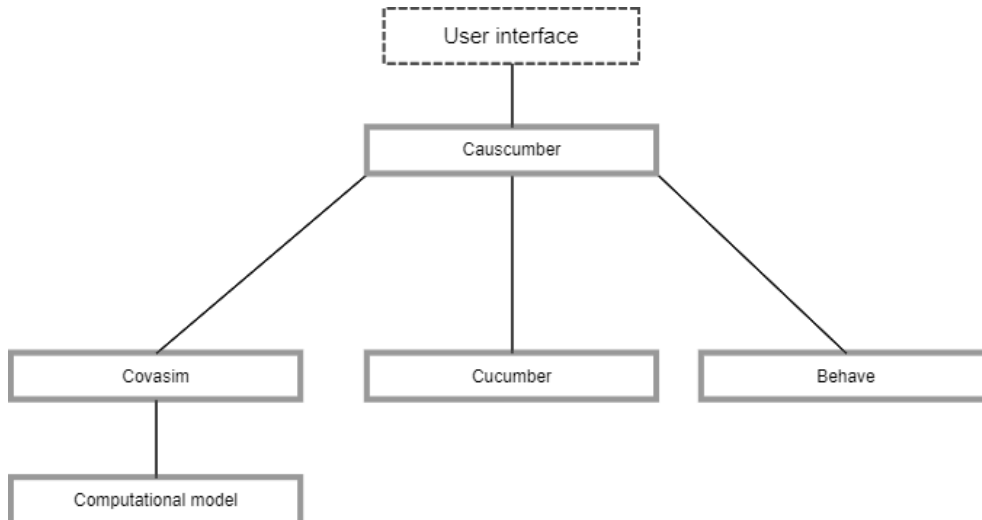


Figure 3.1: A diagram analysis for the system.

3.1 Project Requirements

Causcumber, is a tool that helps testers who aren't trained in software to test a computational model, the goal of this project is to develop a tool that simplify the testing process of using Causcumber. In order to achieve this goal, first will need to understand how Causcumber work, Causcumber itself is a complex tool, it consist of codes from other API(Behave, Cucumber), and it also require to user follow various steps to test a model, this reverse engineering process need to be done in order to start working on the assist tool itself.

To test a model using Causcumber, user will need to define five separate files. First is the *dags* file, in this file user will need to define edges of the model, each edge consists of two parameter that are related to each other in some way, user will need to list out all the parameters in the model, and define the which parameter is related to which.

Second is the environment file, this one is a bit more complicated. User will need to setup multiple functions to initiate the testing data, execute the test itself, set the format of the produced result, and most importantly, the function to execute the tested model. This file require user to have a decent understanding in how to initiate data, and how to setup test using Causcumber.

The third files and fourth file are *dag_steps* and *abstract*, these two files are python scripts for *Behave*. These files will need to define the step decorators for feature file, user will need to setup functions that can read the information in feature file and use the provided information to test the model and return the result. These two files will require user to understand how to setup decoder using the *Behave api*.

The fifth and final file is a feature file, in feature file user will need to set up test in the following order: 1. The background of the simulation (Value of parameter, etc.). 2. Is the edges user wish to test. 3. The scenario outline where user the define how will the change of a parameter should affect a group of parameters. 4. Scenario, where user define the expect behaviour of parameters if a parameter is altered.

As for the assisting tool, it should be able to assist tester in the process by reduce the need for tester interact with the coding part of testing directly. After that, this tool should be able to execute the feature file they have written, the system will examine the feature file and go through the computational model to check if the model performs as the test specifies. As a result, the system should produce a coherent result detailing the accuracy of the tested model. Thus, the systems need to accomplish the following:

R1. As a user, I want this system to be easy to understand, when I use the system, I want

to immediately know what I need to do to get the result I want.

R2. As a user, I want to use the system to aid my testing process by reducing the need to code directly.

R3. As a user, I want the system to go through the feature file I created and test the computational model with it.

R4. As a user, I want the system to produce a result where I can know the accuracy of the computational model.

Since Causcumber has been in development for a while, what this project aims to accomplish is adding more to this tool. Currently the state of Causcumber is capable of executing feature files to test models. And it is capable of returning a lot of useful information, but the information still requires some organization to make the result easier to read, also there isn't a way to execute the testing system without using the terminal to execute the command. It also required testers to manually code every part of the test such as the DOT file that define the relation of the parameter, or the feature file that define the value and expected result. This makes testing with Causcumber still fairly complex, so there's a need for a tool to assist people in operating the system.

To simplify and streamline the testing process, this tool should reduce the need for testers to interact with the code directly. Therefore, this tool for Causcumber should be able to accomplish the following:

1. As a user, I want the result produced by the system to be clean and easy to understand, focusing on the important part.
2. As a user, I want to have the ability to change to different scenarios, so I can test models under different settings.
3. As a user, I want to have the ability to create new scenarios with essential files, so I can modify these files to test the model.
4. As a user, I want to execute and interact with the system through a user interface.
5. As a user, I want to have a more convenient way to create a feature file, to avoid any mistake during the creation of the feature file.
6. As a user, I want to have a more simplify way to create a Dot file, to make the creating

process less confusing.

3.2 Requirement analysis

The first step of this project is to understand the workflow of Causcumber. To test a model, user will need to define several files to start the testing, in these files user will also need to define various aspect to test it. Organize and understand the purpose of these file will be the first step. Since the main theme for this project is simplify and ease of use, and therefore that's what most of the requirements are focused on. For requirement R 1 – 4, the goal is to make the testing process as simple as possible, with a simplified testing process, people will be more willing to test their computational model with this tool.

To enhance the user experience for Causcumber, requirement 1, 2, 3 are aimed to achieve this. These requirements make using Causcumber to test computational models a more convenient experience.

Since integrating the causal model testing can be difficult and confusing, requirements 4, 5, 6 are focused on simplifying this process, by making testing with Causcumber less code demanded, this should help mitigate a lot of errors and further simplify the testing process.

Chapter 4

Design

In order to satisfy the requirement, the design of the user interface will focus on making it easy to navigate and understand. It will also need to have the ability to help user create new testing scenario.

4.1 Design background

Since the basic version of Causcumber require user to manually coded in every step, this tool will be focus on reengineer it into a more accessible version. In Causcumber, user will need to define various file to test a model, first user will need to create a scenario folder, this folder should contain all the file required for user to test a model, then user will need to define four files. First is a *.dot* file, this file will define the relations of parameter for files. Second is the *environment.py*, in this file user will need to define a function that allows user to execute the tested computational mode. Next is *dag_steps.py* and *abstract.py*, user will need to define multiple methods so that when user execute a feature file, Causcumber can execute the correct test. After all these is done, user can then define feature file, in the feature file, user will first need to define the background, then the edges for the model, last is the scenario outline and scenario where user define parameter change and expected outcome.

This tool should modify the information provide by the tester into the files and format above so Causcumber can read. This system is under the assumption that its users have knowledge in programming, but isn't specialize in this area, especially not trained in software testing [20]. Another problem this tool needs to tackle is the lack of will to test a model due to how tedious it is, so testing step will need to be concise and straight to the point. To achieve this, the design of the system should focus on making testing part of the system as streamlined as possible, and the result is produced concise and easy to read.

4.2 Design concept and process

Since the tool need to satisfy the requirements, the design of the interface will need to accomplish the following:

1. The ability to create new scenario
2. The ability to select existing scenario
3. The ability to auto generate essential files for a scenario
4. The ability to create Dot file for scenario
5. The ability to create Feature file for the scenario

In order to make this system easy to use, all the function should be straight forward, and there should be guide provided for the user to refer to.

In the main screen, it should allow user to select which feature file they wish test and run the test and produce the result. This screen should also provide a portal to other function provided by this interface such creating dot and feature file, below is a design concept for this part:

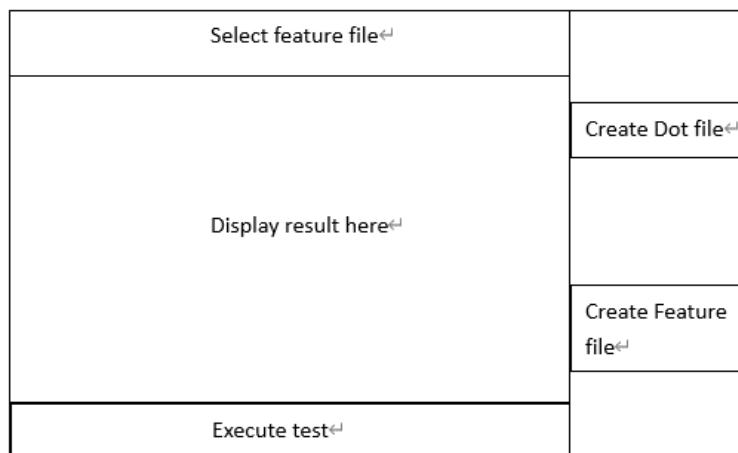


Figure 4.1: Design concept for main menu.

To test model, first user will need to create a .dot file that define the relation of the parameters. By select the create .dot file option in the menu, the user will be bought to a screen that can edit what parameters is in the model. To make this function even more convenient, there should be a way to visualize the relation of the parameters. By visualize the relation of the parameter, user can use it as a reference to aid them in the process. User will

also need to edit the relation between them, next screen will need to display the parameters the user previous entered. The user can select what a parameter is related to, and like the previous part, the interface will also have a visualization to assist user, below is a design concept for this part:

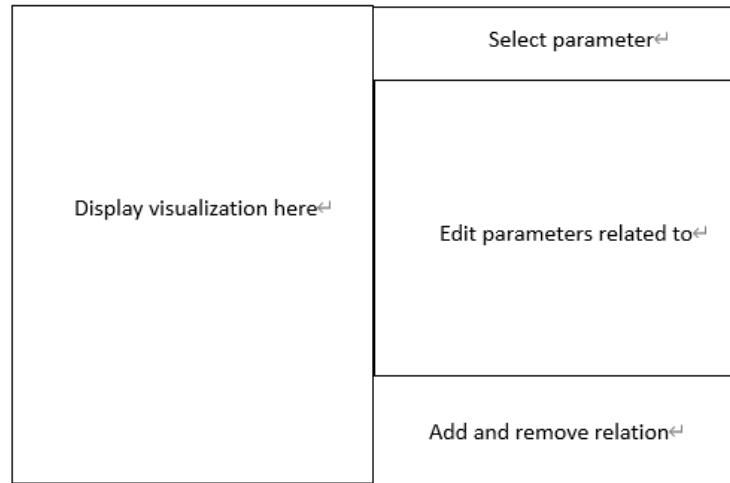


Figure 4.2: Part 2.Design concept for edit .dot function.

After edit the relation of parameter, user will need to edit some of the auto generated files to run the model and read the output, the purpose of these file is for Causcumber to execute the tested model and read the information provided by the model. This part assumed the user have certain level of knowledge in programming, that they have to ability to modify those files

When all the setup process is done, user can start creating feature file as test case, user can select the edit feature file function from the main screen edit it. This function will be structured in a way that is easy to navigate and mostly represent how the final feature file will look like. Since feature file are more complex compared to .dot file, there are more steps required to accomplished this process.

First, user need to define the parameters, parameter type and its value, this setup the tested parameter and their value. User will also need to define what parameter needs to be record. Below is a design concept for this part:

Parameter name↵	Value↵	Parameter type↵
And the following variables are recorded at the end ↵		
Parameter name↵	Parameter type↵	

Figure 4.3: Part 1.Design concept for edit .feature function.

Then, user will need to define edges for the parameters entered in the first part, this part should be similar to the previous edit .dot part to keep the consistence of the system, with some minor change.

Finally, user can start defining the scenarios. In scenarios, user can define a certain situation such as increase the value of a certain variable, and what the expect outcome of the system will be, visualization is also provide to assist user in the process. Below is a design concept for this part:

Display visualization here↵	Define scenario background↵
	Define how the scenario will change↵
	Define the expected outcome↵

Figure 4.4: Part 2.Design concept for edit .feature function.

Chapter 5

Implementation and testing

Following the design, the implementation aimed to fulfill it and the requirements. Some changes were made to the implementation according to the feed back of the Causcumber team. Since Causcumber are built with python, this tool will also be built with python to reduce the chance of any potential problem happening. The tool will primary be a front end for Causcumber, and it will be built with Kivy, an open source python library mainly for develop user interface and application [14].

5.1 Overview

This project will utilize the Screen and ScreenManager function in Kivy. This design is due to many steps in testing with Causcumber require different syntax, to make this clearer for user, different syntax is assigned to different screen. With this, different screen will have different composition to reflect each syntax, extra functions are added to some screens to assist user in editing. This design also allows the development process be more efficient and concise. Since if there's any new function required to be added, develop can simply create a new screen with new function and use ScreenManager to connect it to existing screen, and the new function can be organized into file(s) to keep the codes concise. Since Causcumber require several mandatory files to work, user will need to go through several steps before execute the test, below is the basic workflow of the tool:

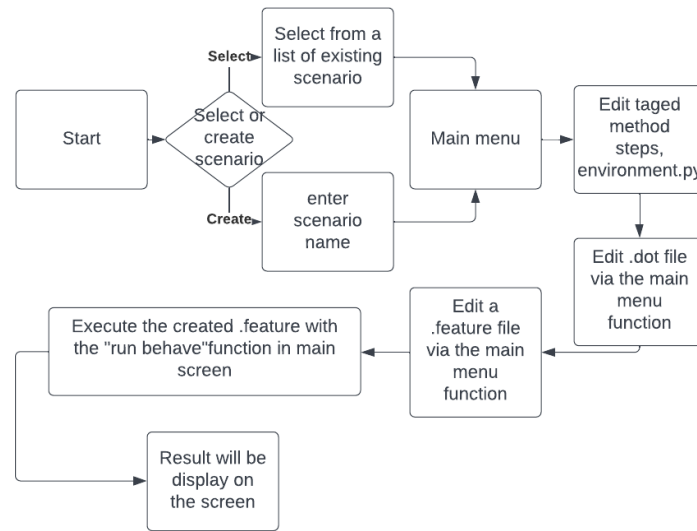


Figure 5.1: Basic workflow for testing a model.

5.2 Create and select scenario

In the beginning, Causcumber require user to create scenario directory, a scenario contains two mandatory sub-directories, *dags*, a directory contain causal graphs that defines the relations of parameter as *.dot* file. The second sub-directories is *features*, a directory where user can create *.feature* file that contain element for behave [8]. It also contain *environment.py* file, and a sub-directory *steps* that has scripts to implement step definitions for *.feature* file. The tool will allow user to create or select and existing scenario, upon create a new scenario, the tool will generate the basic files and sub-directory required for a scenario. Upon starting the system, the user will have two options, select, and create scenario.

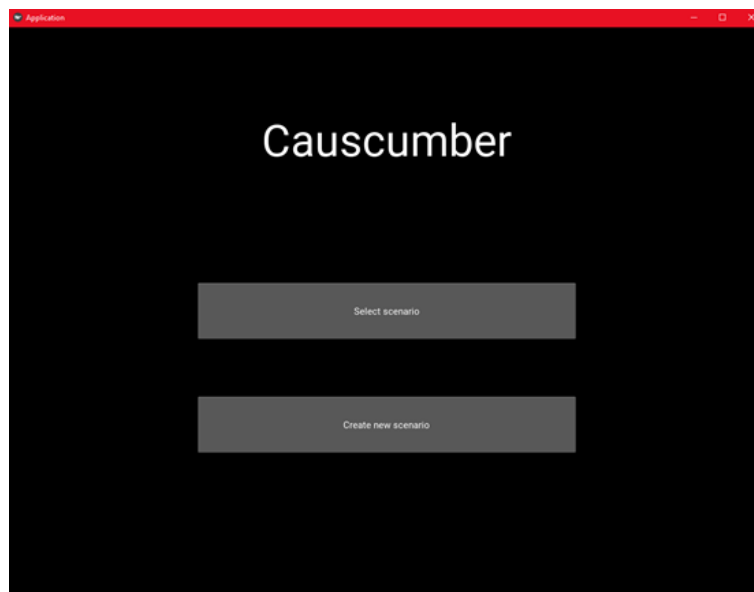


Figure 5.2: Start screen.

If user choose to select existing scenario, they will be presented with a list of existing scenarios to choose from. If user choose to create a new scenario, user will be prompt to enter the scenario name.

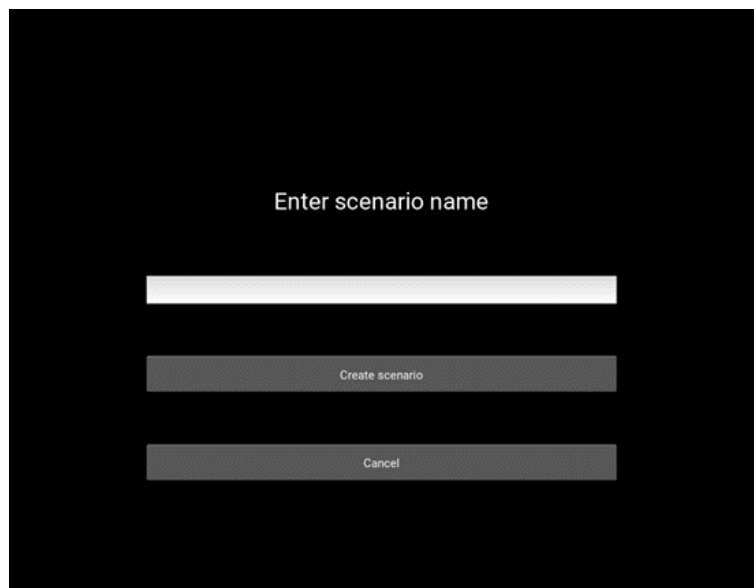


Figure 5.3: Create new scenario screen.

The tool will create files with custom file and method name based on the scenario name entered by the user.

5.3 Main screen

In the basic version of Causcumber, to execute test, user will need to use terminal and run the command “behave feature/feature filename”. In the main screen, user can achieve this by simple use the “Select feature file” to chose the target *.feature* file, and “run behave” to execute the test. Then the result will be display on the screen. But before this, user will need to go through some steps to setup the test.

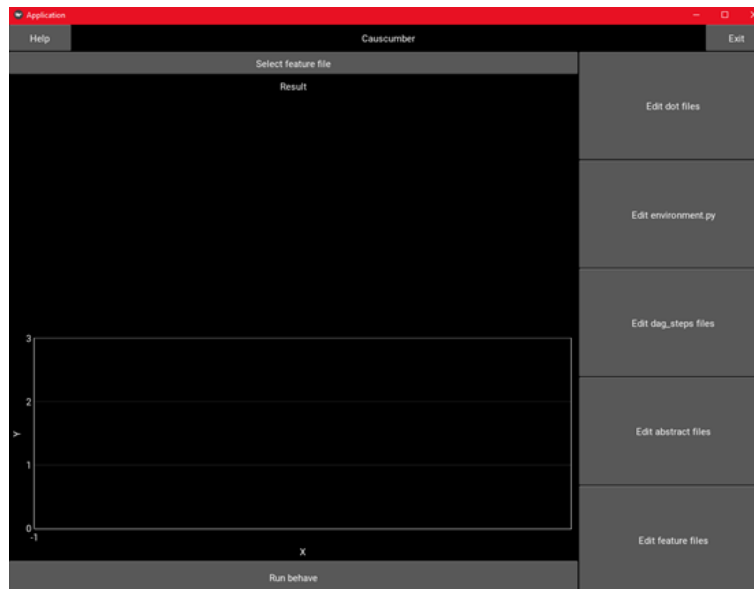


Figure 5.4: Main screen.

The steps required is placed from top to bottom at the right of the screen as shown above, user will need to finish all the steps in order to test a model. A help section on top left is also provided to guide user if needed.

5.4 Edit *.dot* file

In the *.dot* file, user will first need to define the parameter clusters, typically there will be two clusters, one for input parameters and another for output parameters. After that user will need to define the relation between the parameters, this is done by using the syntax *parameter1 -> parameter2*. One useful feature for *.dot* file is combine with Graphviz, an open source graph visualization software, user can generate a graph that help visualize the relation and cluster.

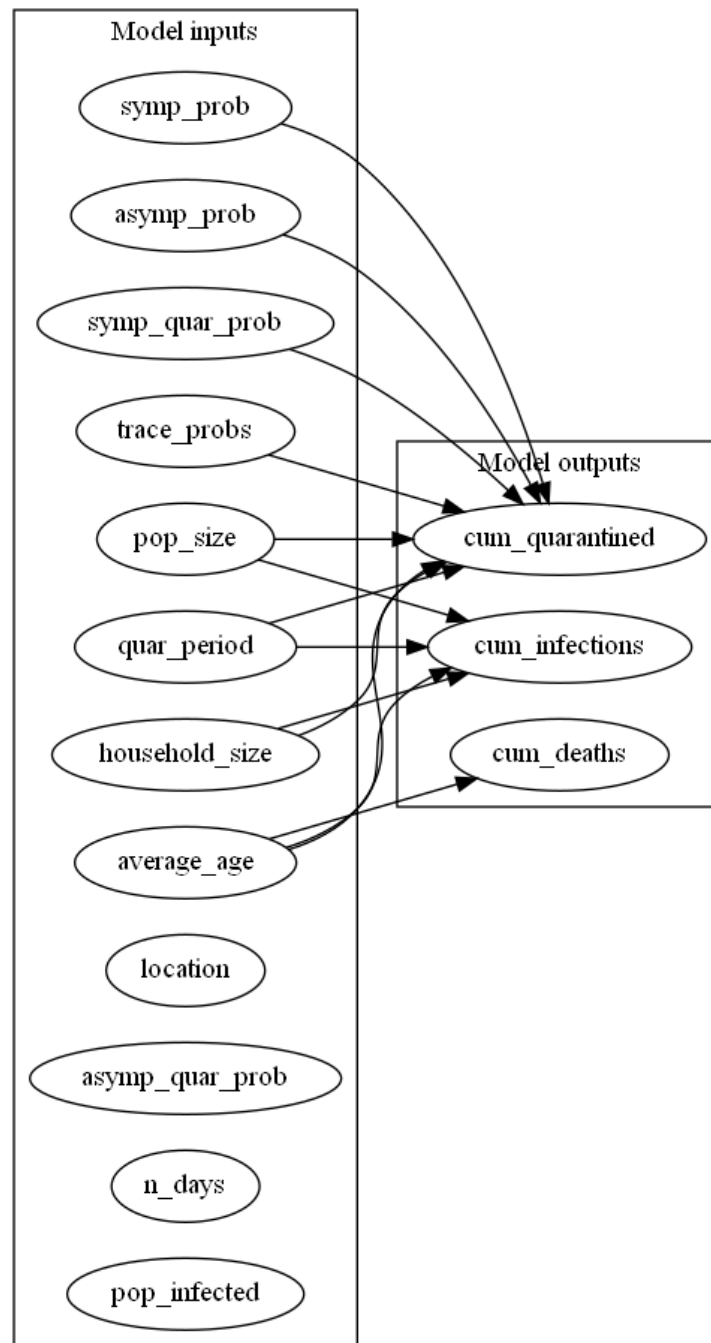


Figure 5.5: Graphviz example.

In the user interface, Graphviz is used to assist in the process. After click on the “Edit dot files” button in the menu, user will be brought to the screen for editing parameter clusters.

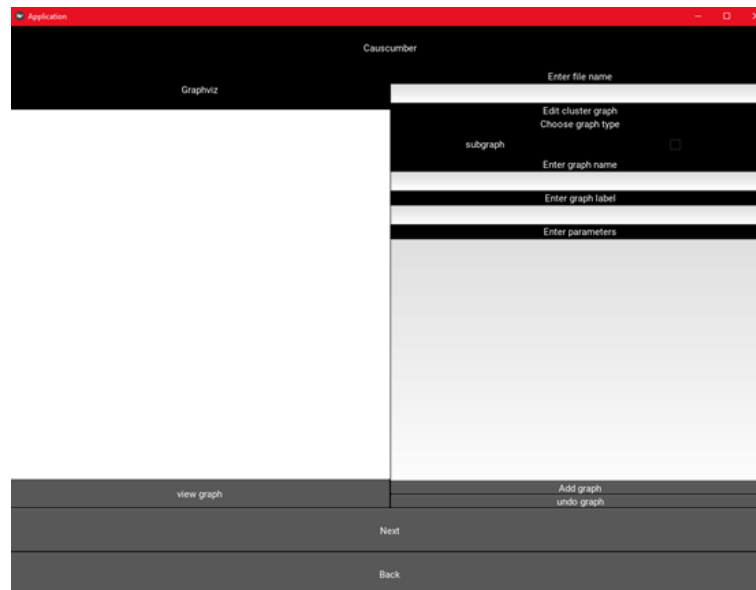


Figure 5.6: Add parameter to .dot file.

At the right side, user will need to enter the filename, this can either be an existing file or a new file. Next user can change the syntax of the graph by tick or untick the subgraph checkbox, after user will need to enter the graph name and label. Then in the last section, enter parameters, user can enter the parameter for the graph.

After user input all the parameters, user can click next to go to the next step to edit relation of the parameters. Same as before, user should enter the filename, after that user need to click on the “Update parameter relation”, then the interface will display parameters entered in the previous step. By clicking the “select parameter”, a drop down menu will open for user to select a parameter, by select a parameter, that mean the selected parameter is related to the parameters ticked below (Include itself if ticked). When finish editing relations, user can click the “add graph” button to input the graph. The result of the graph will be display at the left side of the screen to assist the user.

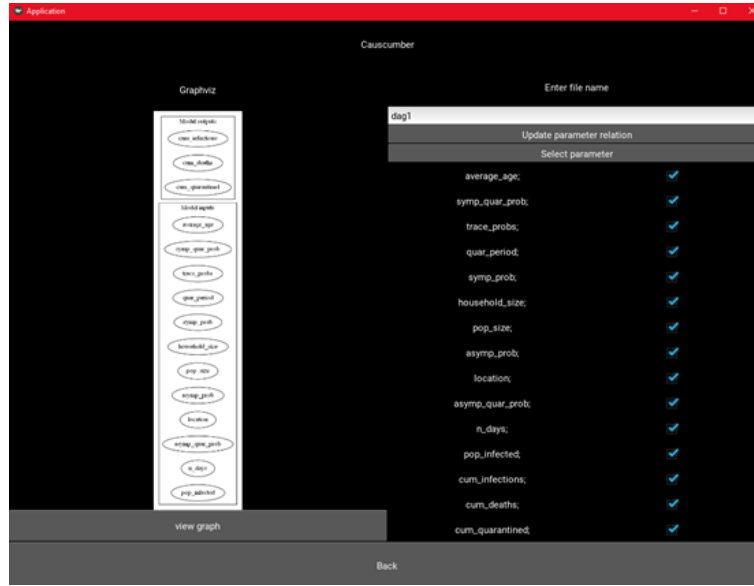


Figure 5.7: Edit parameter relation.

5.5 Setup background files

Next, there are three background files require to setup. First in *environment.py*, user need to edit a *run_[Model name]* method, the purpose of this file is to execute the testing model, user will need to modify it to able to run the model and return the result data. Second, is in *dag_steps.py*, there are two parts require to be edit, custom distributions and metavariables.

For custom distributions, user is requiring to define class for distributions so that Causcumber can picks it up. The reason for why this needed is because default distributions in Causcumber uses SciPy, an open-source software for mathematics, science, and engineering [24], to generate valid parameters values between 0 and 1, but not all the parameters are numerical. Custom distributions are for these types of parameters, for example *location* in covasim, custom distributions should generate strings from a given set.

For metavariables, user can define metavariables in the Background. Metavariables, or metasynthetic variable in computer science is a placeholder name that has no meaning and will be substituted [18]. User can use this to modify data in data sets.

The third file is *abstract.py*, in this file user can define custom constrains for parameter value. This limits the distributions for test data generation. For all three files, it requires users to manually define it since different models may work differently and require different codes, use uniform generated codes may cause problem. In the main screen, buttons are provided to help user open these file, by clicking those button, the files will be open with default editor set by the user.

5.6 Edit .feature files

Finally, user can start edit the feature file. In feature file, user will need to define background, list edges, scenario outline and scenario. In background, user will need to define the parameters and the distribution, user can also define metavariables, then user will need to define variables are recorded at the end, last user can also define extra conditions for the scenario with *And* syntax in Cucumber.

In the next part, user will need the list the edges of models, this part allow Causcumber to focus on relations of parameters in model define by the user. Next is *Scenario outline*, in this part user can define changes in parameter and the expected outcome, this require user to provide *Example*. Finally is *Scenario*, this is mostly same as *Scenario outline* but without *Example*.

All the parts mentioned above require user to formatted in a certain syntax, with the interface, the syntax is incorporated into the format of the interface's design. Starting with background, user need to edit the feature file's name, then user can start setting the parameters, parameters' type and parameters' value. Next user can add meta variables, this part isn't necessary for feature file, and can be left empty. Next is the recorded variables, this part is similar to the meta variables but will generate different syntax upon create feature file. Last is the for user to add extra condition to the background if needed. This part is structured very similar to the *.feature* file's syntax, but with some modification to make those syntax even more plain text.

The screenshot shows the Causcumber application interface for editing a .feature file. The window has a red title bar and a black header with the text "Causcumber" and "Enter file name". Below this is a section titled "Edit background" with the subtitle "Given a simulation with parameters". This section contains a table with three columns: "parameter", "value", and "type". Below the table are two sections: "Meta variable" and "Meta variable type". These sections are followed by a section titled "And the following variables are recorded" with a table with two columns: "variable" and "type". Below this is a section titled "More condition (Leave empty if not need)". At the bottom of the interface are two buttons: "Next" and "Back".

Figure 5.8: Edit first section of .feature file.

Next part is to define the edges, this part is mostly identical to the edit *.dot* function to keep the consistence of the system, the only new addition is a new option allow user to add new edges. This new function is added due to the potential need for user to add extra edges, user can choose to not add new edges if there's no need for that.

After this is scenario outline and scenario, both are also structured like background, where parts of the sentence that require user to edit being left blank. In scenario outline, the syntax for example is a table like structure, this is simulated in the interface where user can use the add column and add row to edit the example.

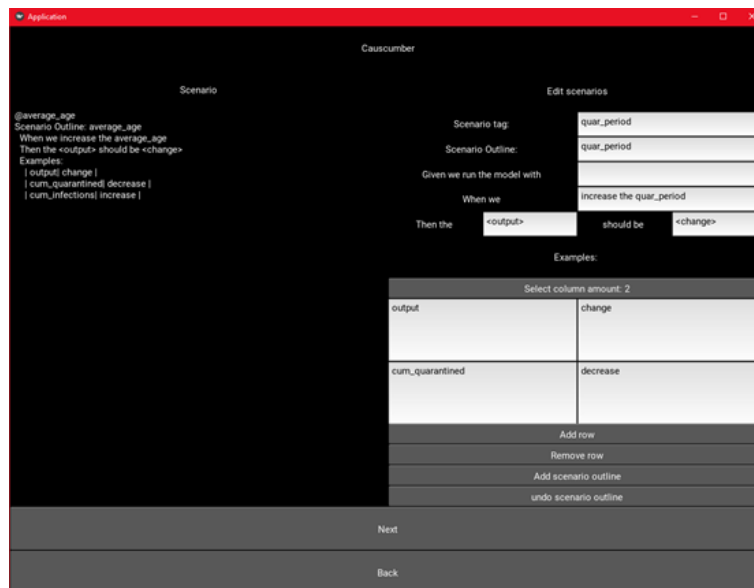


Figure 5.9: Edit scenario outline section of *.feature* file.

In scenario, instead of example, it uses “Then” and “And” structure in Cucumber to represent the expected result. The structure is similar to the previous page but with example replaced with the option to add “And”.

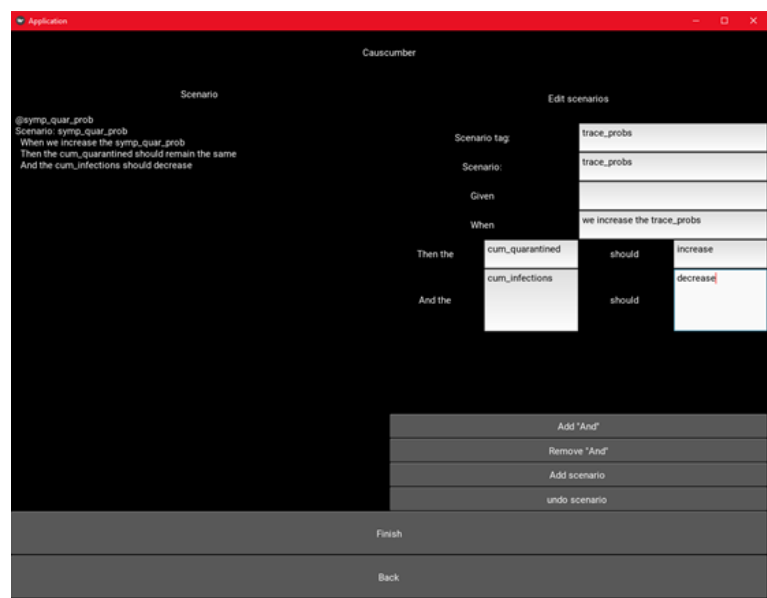


Figure 5.10: Edit scenario section of .feature file.

When done, user can just click “Finish” button to return to main menu, then select and run the feature file created.

Chapter 6

Evaluation

In this chapter, the tool will be put in to test with two computational model, influenza1918 and Covasim. Influenza1918 is a relatively simple model compare to Covasim, this chapter will go through the testing process more both model, and the result produced, then some discussion about the result.

6.1 Testing influenza1918

Influenza1918 is an equation-based model originally developed to for the need to design model validation strategies, this is a model that simulate epidemiological disease-spread of the course of the 1918 Influenza epidemic within the United States. This model has five state variables and six parameters [21].

6.1.1 Testing process for influenza1918

First, a scenario name influenza1918 is created, by creating this scenario, the basic files for testing is also auto generated. Since influenza1918 is implemented through OpenModelica in a *influenza1918.mo* file, so to test this model, first user need to put the model file into the `\scenarios\influenza1918\model`. Next use the “Edit dot files” button to start edit the relations of the parameters. A file named *influenza1918_abstract* is created, there are two graph needed, first is *cluster_inputs* with the input parameters, then *cluster_outputs* with the output parameters. In the right panel, enter the file name *influenza1918_abstract*, then select the type of graph, input the graph name and label, and the parameters for the selected cluster, click add graph to apply the change.

Enter file name	
influenza1918_abstract	
Edit cluster graph	
Choose graph type	
subgraph	<input checked="" type="checkbox"/>
Enter graph name	
cluster_inputs	
Enter graph label	
Model inputs	
Enter parameters	
EncounterRate; TransmissionProb; MortalityProb; IncubationTime; Infected; MortalityTime; RecoveryTime;	
Add graph	
undo graph	

Figure 6.1: Edit graph for input parameters.

After added the graph, the left panel will then display the graph via Graphviz. Next is edit the relations between these parameters.

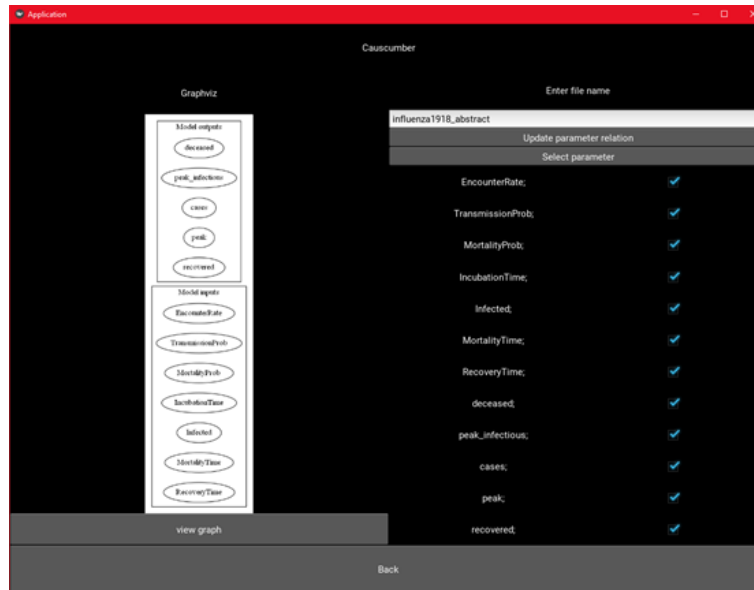


Figure 6.2: Edit relation for parameters.

First, select the file, then select the parameters that are going to be affected via the check-boxes, then select the parameters that are going to affect them via the drop-down menu, then the left panel will be updated.

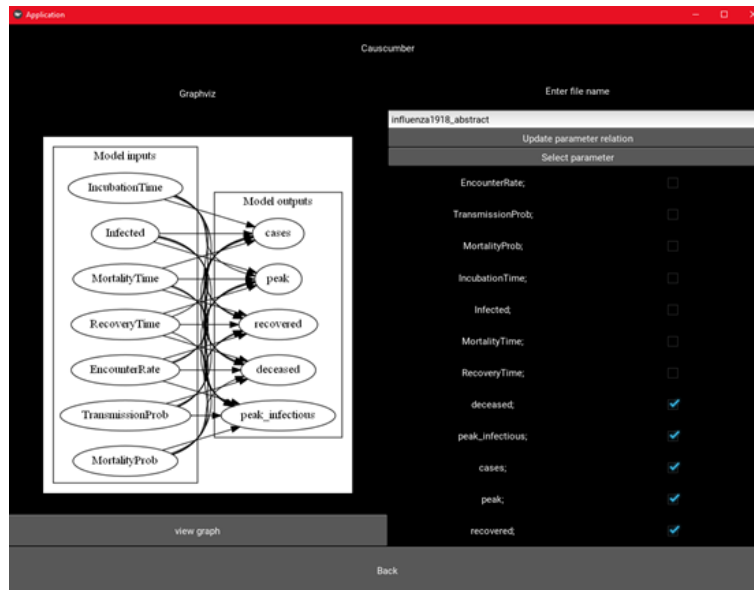


Figure 6.3: Relations of parameters presented in Graphviz graph.

After finishing editing the relationship, click back to return to the main menu. Next user will need to edit the three background files. In `environment.py`, edit the `run_influenza1918` function to execute the `influenza1918` model. Then for `dag_steps.py`, define metavariables in

the background, a metavariable called “m” needs a function “populate_m” define. Last is for abstract.py file, apply custom constrain to the parameters of the tested model.

Finally is the .feature file, use “Edit feature files” to create a feature file. Starting from the top user will need to define the feature file’s name, its parameters and its value/distribution and type,

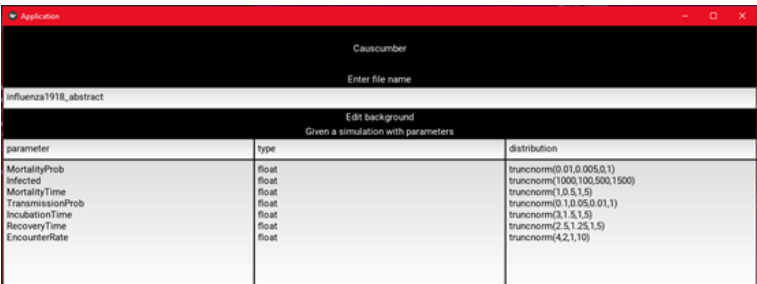


Figure 6.4: Edit initial parameters name, type and value/distribution in influenza1918.

Next step is to define what parameter to record and when to record, since influenza1918 doesn’t require any meta variable, that part will be left empty.

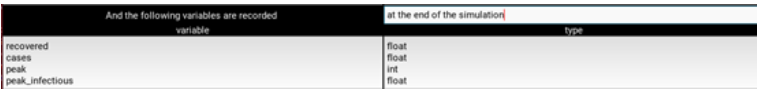


Figure 6.5: Edit recorded parameters name and type in influenza1918.

Next is to define the edges for the parameters, this process is same as edit dot file step. Since scenario outline is required in this case, next part will also be skip. Last is scenario, in the right panel, edit the change and expected outcome.

Edit scenarios

Scenario tag: MortalityProb_decrease

Scenario: P(Mortality) decrease

Given

When we decrease the MortalityProb

Then the deceased should decrease

recovered

And the should increase

should

should

Add "And"

Remove "And"

Add scenario

undo scenario

Figure 6.6: Edit scenario for influenza1918.

Upon finish edit all the scenario, user can then return to main menu and then select the feature file. After select feature file, user run behave to produce the result on the screen,

Application

Help Causcumber

Select feature file

Result

Given a simulation with parameters ... passed in 0.041s

parameter	type	distribution
MortalityProb	float	truncnorm(0.01,0.005,0.1)
Infected	float	truncnorm(1000,100,500,1500)
MortalityTime	float	truncnorm(1,0.5,1.5)
TransmissionProb	float	truncnorm(0.1,0.05,0.01,1)
IncubationTime	float	truncnorm(3,1.5,1.5)
RecoveryTime	float	truncnorm(2.5,1.25,1.5)
EncounterRate	float	truncnorm(4,2,1,10)

{'classname': 'compare_influenza1918_abstract.Compare influenza1918_abstract', 'name': 'Draw DAG', 'status': 'passed', 'time': 0.054938}

Given a simulation with parameters ... passed in 0.013s

parameter	type	distribution
MortalityProb	float	truncnorm(0.01,0.005,0.1)
Infected	float	truncnorm(1000,100,500,1500)
MortalityTime	float	truncnorm(1,0.5,1.5)
TransmissionProb	float	truncnorm(0.1,0.05,0.01,1)
IncubationTime	float	truncnorm(3,1.5,1.5)

Figure 6.7: Result produced by Causcumber display in result section.

6.1.2 Testing result for influenza1918

In the result produced by Causcumber, either with the assisting tool or without, all produce the same result. In the files produced by the tool, starting with *.dot* file, it is identical with manually created version with some minor difference in formats that doesn't matter in this case. With this assist of the tool, this process also become a lot more efficient. Next are the three background files, in those files, most of the parts are already pre-define with some parts that can only be manually define by user. Last, in *.feature* file, the file generated by the tool is no difference compare to the manually created file, and by splitting the files into different sections, and edit it one by one, it become more streamline and readable from a user's point of view.

6.1.3 Discuss influenza1918 result

With the assisting tool, creating test for influenza are relatively simplify because as a user, it is not necessary to understand all the syntax for all the files. And with the auto generated files, since not all the parts require manually input, the chance of error occur has been reduced. One of the problem originally user may encounter when define edges for files such as *.dot* file and *.feature* file, is that it is difficult for user to list all the edges without any sort of visualization. With the Graphviz's assist, this process become much clearer during the process. Unfortunately, for the three background files, part of it still require user to manually code those parts in due to the methods use by developers to code computational model maybe drastic different.

6.2 Testing Covasim

Covasim is an agent-based model pf COVID-19 dynamics and interventions, it simulate individual people as an agent, agents will be assigned different states such as infectious or recovered, agents can change to different states and in each states can provide different affects to the simulations. Agents will also be affected by the "Interventions" such as masks or physical distancing. Compare to influenza1918 models in the previous section, this model is a lot more complex with more parameters and methods.

6.2.1 Testing process for Covasim

First a scenario name "Covasim" is created, in this scenario, in this test, we focus on the interventions part of Covasim. Starting with *.dot* file, one graph for input parameter and another one for output is created, then the following relations are defined,

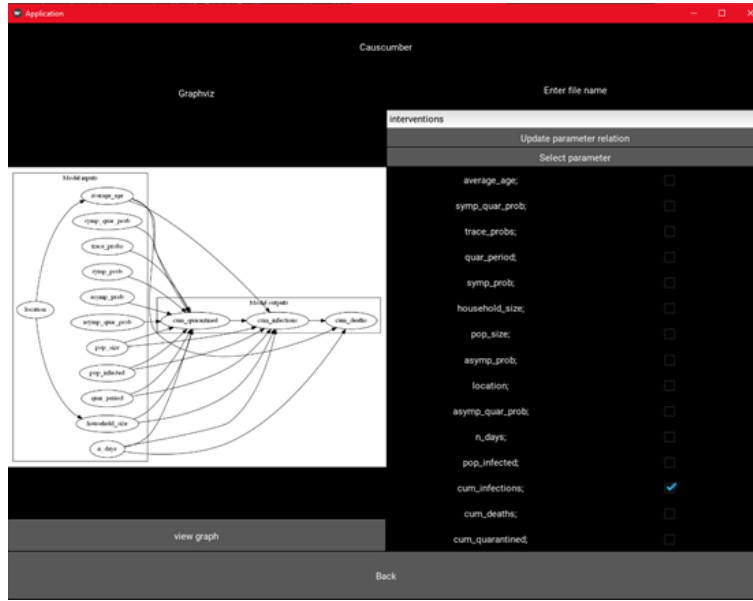


Figure 6.8: Relation for parameters for testing Covasim.

Next is to define the three-background file, in *abstract.py*, there are two custom constraints needs to be define, “average_age” and “household_size”. In covasim, one of the parameters requires string, therefore in *dag_steps.py* a custom constrain named “countries” is defined. Next, there are several meta variables in the background, these meta variables are also defined as functions in *dag_steps.py*.

Last is the feature file, in the first section, one of the parameters is location, since it only accepts string, its distribution will be using the custom distribution “countries” defined before. Also, there are meta variables require, so in this section meta variable are also required to be defined. There is also multiple extra initial condition require to be defined, this is also done in the first section.

The screenshot shows the 'Causcumber' application window. The 'Edit background' section is active, displaying a table of parameters and their values. The parameters are: quar_period (int, 60), pop_size (int, 25), pop_infected (int, 15), location (str, 1000), symp_prob (float, 1.5), asymp_prob (float, 0.8), symp_quar_prob (float, 1.5), asymp_quar_prob (float, 1.5), and trace_prob (float, 1.5). The table also shows the distribution for each parameter. Below the table, there are sections for 'Meta variable' (average_age, household_size), 'And the following variables are recorded' (cum_quarantined, cum_infections, cum_deaths), and 'More condition' (Leave empty if not needed). The 'Next' button is visible at the bottom.

parameter	value	type
quar_period	60	int
pop_size	25	int
pop_infected	15	int
location	1000	str
symp_prob	1.5	float
asymp_prob	0.8	float
symp_quar_prob	1.5	float
asymp_quar_prob	1.5	float
trace_prob	1.5	float

Figure 6.9: Background for testing Covasim.

In next section, same as influenza1918, edges are defined using the assisting tools, in Covasim, there are additional edges being added, so instead of choosing next, choose “And add edges” options to add more edges, this is similar function as define edges, but will return slightly different result in the generated feature file. Next, is to edit the scenario outline, by changing the column and row amount, we can provide information according to the requirement of the scenario outline. After this is scenarios, this process is same as influenza1918, with all these steps done, select the newly created feature file, and run Beheve.

The screenshot shows the 'Causcumber' application window. The 'Result' section is active, displaying a plot of the simulation results. The plot shows a line graph with 'X' on the x-axis and 'Y' on the y-axis. The y-axis ranges from 0 to 3, and the x-axis ranges from -1 to 3. The plot shows a line that starts at (0,0) and increases to (3,3). The plot is titled 'Result'. To the right of the plot, there are several options: 'Edit dot files', 'Edit environment.py', 'Edit dag_steps files', 'Edit abstract files', and 'Edit feature files'. The 'Run behave' button is visible at the bottom.

Figure 6.10: Result produced by using assisting tool.

6.2.2 Testing result for Covasim

In the testing result for Covasim, result produced by are similar to manually created version, in the manually created version of feature file, some Cucumber syntax can be used to reduce the length of the file, this isn't replicate the version created by the tool. As for the test result produced by Causcumber, it is mostly identical to the manually created version with some difference in indentation which doesn't affect the result.

6.2.3 Discuss Covasim result

In the testing process for Covasim, the tool provides a lot of assists in the process, especially in the process of defining relation of parameter, and the editing process of feature file. But with the background files, despite the guide provided, it still requires user to have decent programming skill to accomplish it. Overall, with the assist of the tool, the testing process has become more streamline and swift, but parts of the system still holds back by necessity to manually code the test.

6.3 A summary of results

In the result provided by testing on influenza1918, it shows that with the help of the tool, testing become more convenient. The process is overall simplified by reduce the need for user to directly interact with the coding part of testing. With the parts that is necessary for the process, it is mostly reduced by auto generate most of it, with some minor parts still require to be define. This is also the same when defining *.feature* file, without the need to deal with syntax and format, this process become a lot more smoother.

For covasim, the experience is mostly same when it comes defining relations for parameter and creating feature file, without the need to directly code these files, and with the help of the GUI, the testing process is a lot more understandable and streamline. But when it comes to the background files, despite the auto generated parts, there are still more required to be define. Consider that Covasim is a more complex model, this is to be expected, but this part also shows that if the user tends to add a lot of custom testing aspect, the testing process could still become lengthy.

Combine the information provide by two models, it shows that in terms of editing *.feature* file and *.dot* file. This tool can effectively assist user, but when it comes to the background file, if the user is intended to have many custom conditions, then the user will be required to do a decent amount of editing in the background files, this may discourage people to participate in testing.

6.4 How does the tool help user simplify the testing process

As shown by the result, the tool can effectively assist user by allowing user to ignore the required syntax to edit the dot and feature file, with the tool, user will only be required to input the data hinted by the GUI. In this way, user will not be required to learn the complete syntax of the file, make it easier to promote this system to people who aren't specialize in software testing.

In the files that require manually input, parts of the files will be auto generated. In these files, a total of 319 lines of code will be auto generated, most of it are essential function for the working of Causcumber and will require a decent understanding in Causcumber system to implement those lines. The auto generated parts of the files, together with the assist generated dot and feature file, lowers the skill requirement to use the Causcumber.

To summarize, two main difficulties in using Causcumber are tackled, first is the need to understand the syntax when edit dot and feature file, second is background files for Causcumber. With these two-problem tackled, user won't necessary need to understand how Causcumber work, but will only required to know where to input the parameters, which guides are provided to help user in the process.

Chapter 7

Conclusion

Computational models are important tools in modern days, researchers and decision makers can utilize these models to aid them during research or make important decision. But these models may often receive poor testing, this is because its developers are often not trained in software testing, and the testing process can be very complex and time consuming.

Advance testing method are developed to assist testing, but these methods are often too complex for people who are trained in software testing. One of these tools is Causcumber, a tool that use the behaviour of model to determine if the model meet the developer's expectation. But this tool still suffers from the same problem as other advance tool, where it maybe too complex for untrain user, therefore discourage people from using it.

A tool to improve user experience of Causcumber is created, this tool can automatically turn user's input into the format that can be accepted by Causcumber, and auto generate parts of background file to reduce user's workload. With this tool, user won't need to learn the syntax required by Causcumber, and the line of code required user manually input is also reduced.

Causcumber is still a in development tool, and there might be changes in the future, and there's also more to improve for the assist tool. First is in the viewing function of graphviz graph, in the current version, it's size is limited by the some technical issues, and will affect user experience, parts of these is due some limitation of Kivy api, this will require some time to improve.

Second is editing scenario outline and scenario in feature file, more hints or the ability to select how the parameter going to change (For example, increase, decrease) from a menu can be implemented. This is to be decide since this function might limit the options in testing.

Third is to further reduce the need for user to program, namely the three background files. In those files, a similar like edit scenario in feature can be added, but since these files are a lot

more complicated, the editing function for these files may not be as simple to implement, the way user implements their model may also affect this process, which adds more complexity to the implementation of this function.

Overall, this project set out to make a complex tool become more approachable, despite some technical challenges, this project provides some interesting results for more future work on this subject. With more functions implemented, testing will be more approachable in the future.

Bibliography

- [1] Computational modeling. *National Institute of Biomedical Imaging and Bioengineering* (May 2020), 1.
- [2] BENNO RICE, R. J., AND ENGEL, J. Behavior driven development.
- [3] BONABEAU, E. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences* 99, suppl.3 (2002), 7280–7287.
- [4] BRITTANY JOHNSON, YURIY BRUN, A. M. Causal testing: Finding defects’ root causes. 123 – 146.
- [5] CARLEY, K. M. Validating computational models.
- [6] CLARK, A. G., WALKINSHAW, N., AND HIERONS, R. M. Test case generation for agent-based models: A systematic literature review. *Information and Software Technology* 135 (2021), 106567.
- [7] CLIFF C. KERR, ROBYN M. STUART, E. Covasim: An agent-based model of covid-19 dynamics and interventions. *PLOS Computational Biology* (July 2018).
- [8] FOSTER, M., C. A. T. R., AND WILD, C. Citcom-project/causcumber.
- [9] GASKIN, P. B. L. J. Partial least squares (pls) structural equation modeling (sem) for building and testing behavioral causal theory: When to choose it and how to use it.
- [10] GLOBAL, C. All models are not born equal - empirical vs. mechanistic models - creme global.
- [11] HITCHCOCK, C. Causal models.
- [12] JOHNSON, B., BRUN, Y., AND MELIOU, A. Causal testing: Understanding defects’ root causes. 87–99.
- [13] KERR CC, STUART RM, M. D. E. Covasim.
- [14] KIVY.ORG. Introduction — kivy 2.1.0 documentation.
- [15] MARIT VAN DIJK, ASLAK HELLESØY, E. Cucumber.

- [16] MARIT VAN DIJK, ASLAK HELLESØY, E. Gherkin reference.
- [17] MUFFY CALDER, CLAIRE CRAIG, E. Computational modelling for decision-making: where, why, what, who and how. *Royal Society Open Science* (June 2018).
- [18] PROGCLUB.ORG. Metasyntactic variable.
- [19] ROBERT C WILSON, A. G. C. Ten simple rules for the computational modeling of behavioral data. *eLife* (November 2019), 1.
- [20] SEGAL, J. Scientists and software engineers: A tale of two cultures. *Open research Online*, 1 (2008), 2.
- [21] SUKUMAR, S. R., AND NUTARO, J. J. Agent-based vs. equation-based epidemiological models: A model selection case study. 74–79.
- [22] UPULEE KANEWALA, J. M. Testing scientific software: A systematic literature review. *ScienceDirect* (May 2014).
- [23] VAN DYKE PARUNAK, H., SAVIT, R., AND RIOLO, R. L. Agent-based modeling vs. equation-based modeling: A case study and users’ guide. 10–25.
- [24] VIRTANEN, P., GOMMERS, R., OLIPHANT, T. E., HABERLAND, M., REDDY, T., COURNAPEAU, D., BUROVSKI, E., PETERSON, P., WECKESSER, W., BRIGHT, J., VAN DER WALT, S. J., BRETT, M., WILSON, J., MILLMAN, K. J., MAYOROV, N., NELSON, A. R. J., JONES, E., KERN, R., LARSON, E., CAREY, C. J., POLAT, İ., FENG, Y., MOORE, E. W., VANDERPLAS, J., LAXALDE, D., PERKTOLD, J., CIMRMAN, R., HENRIKSEN, I., QUINTERO, E. A., HARRIS, C. R., ARCHIBALD, A. M., RIBEIRO, A. H., PEDREGOSA, F., VAN MULBREGT, P., AND SCI-PY 1.0 CONTRIBUTORS. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272.

Appendices

Appendix A

Testing process for influenza1918

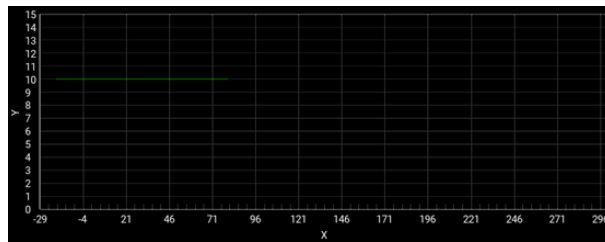


Figure A.1: A 95% confidence interval example with one scenario passed.

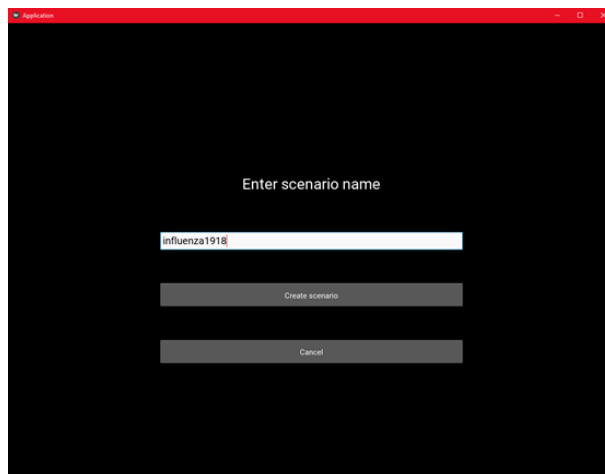


Figure A.2: Creating new scenario for influenza1918.

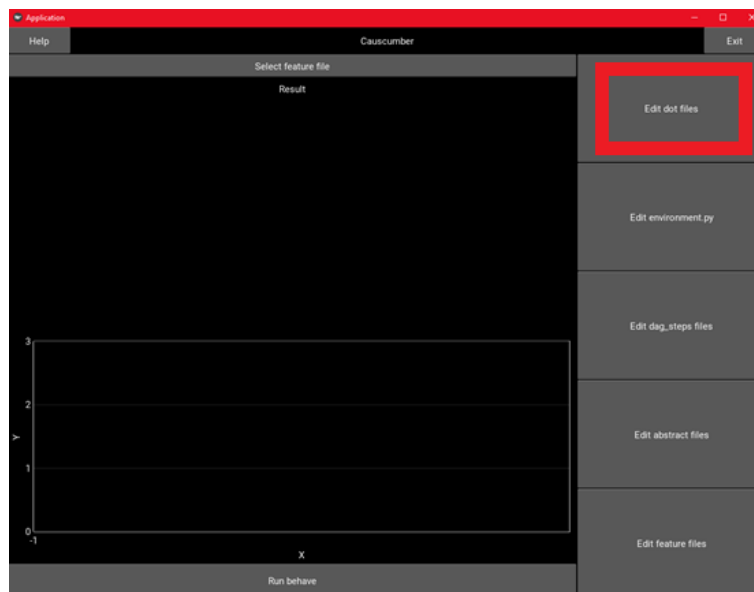


Figure A.3: Select “Edit dot files” function.

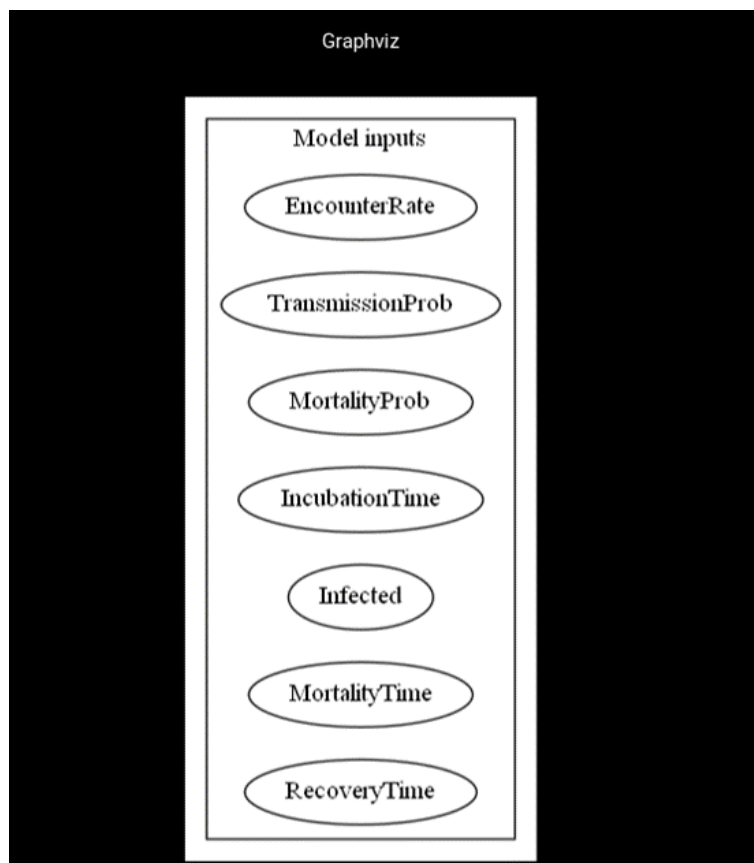


Figure A.4: Graphviz for input influenza1918 parameters graph.

Enter file name	
influenza1918_abstract	
Update parameter relation	
Select parameter	
EncounterRate;	
TransmissionProb;	
MortalityProb;	
IncubationTime;	
Infected;	
MortalityTime;	
RecoveryTime;	
deceased;	
peak_infectious;	
cases;	
peak;	
recovered;	
peak,	<input type="checkbox"/>
recovered;	<input checked="" type="checkbox"/>

Figure A.5: Drop-down menu for select parameter.

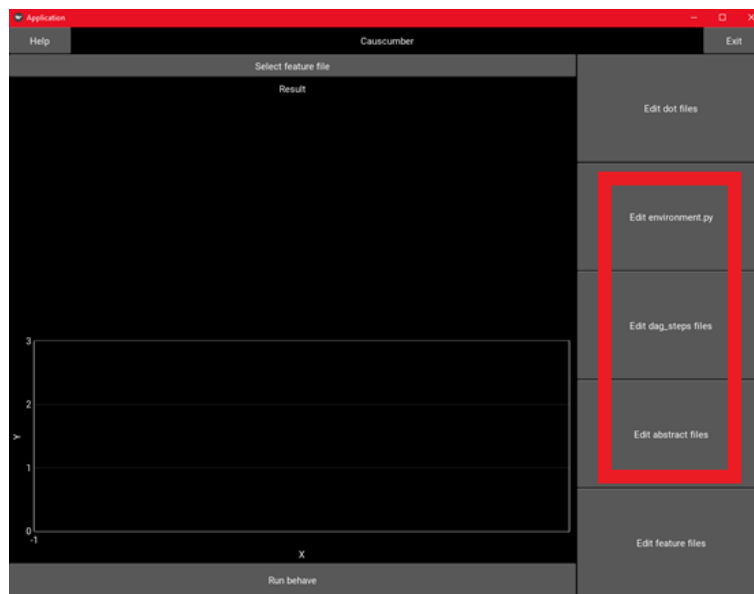


Figure A.6: Edit background files for testing influenza1918.

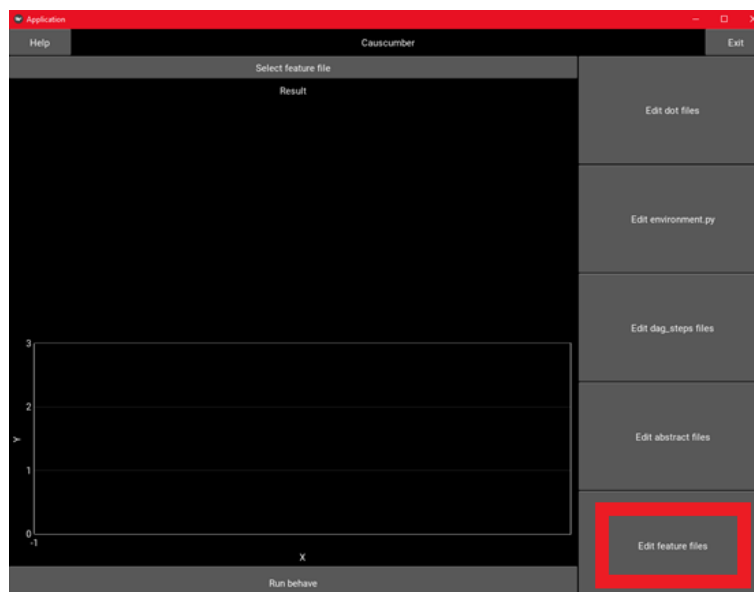


Figure A.7: Edit feature files for testing influenza1918.

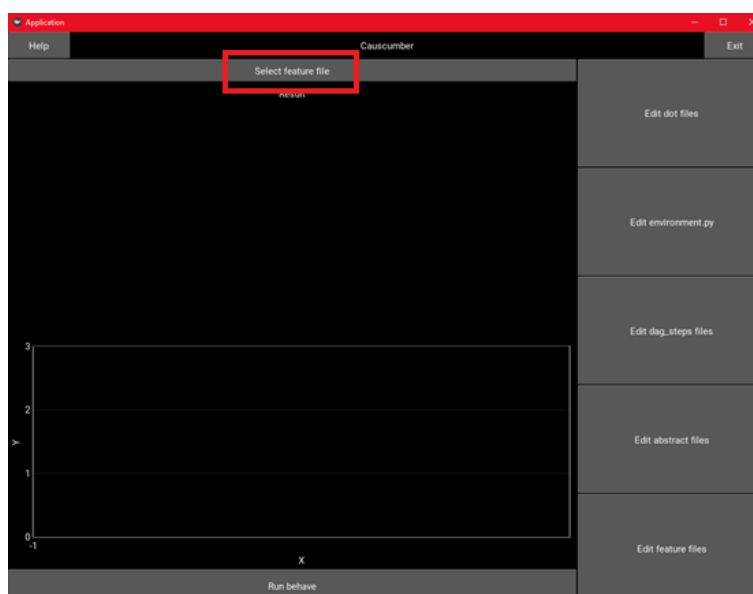


Figure A.8: Select “Select feature files” function.

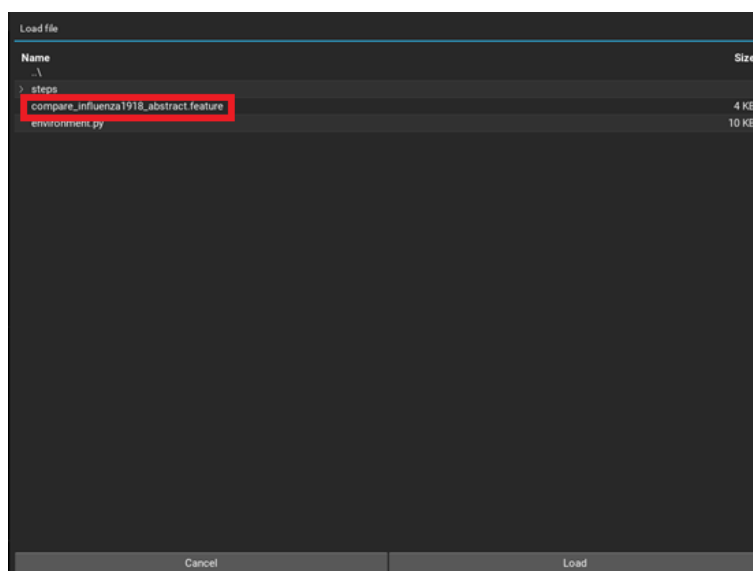


Figure A.9: Select newly created feature file.

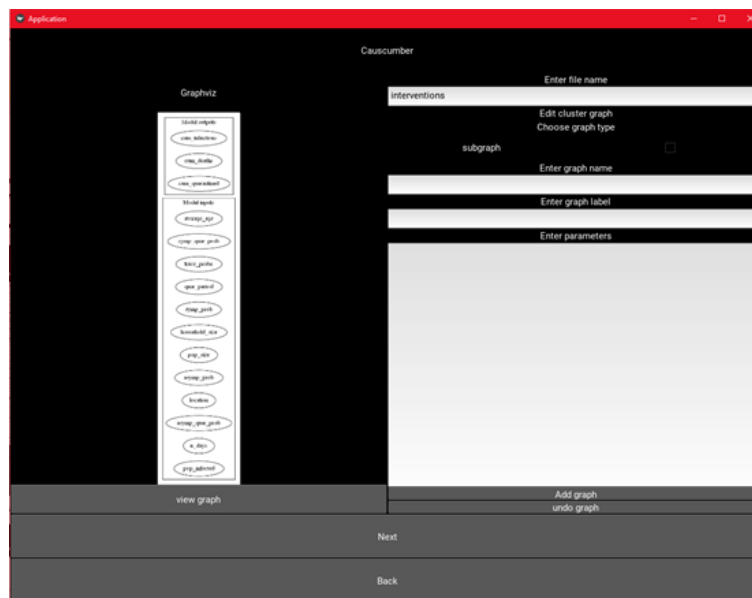


Figure A.10: Input and output graph for testing Covasim.

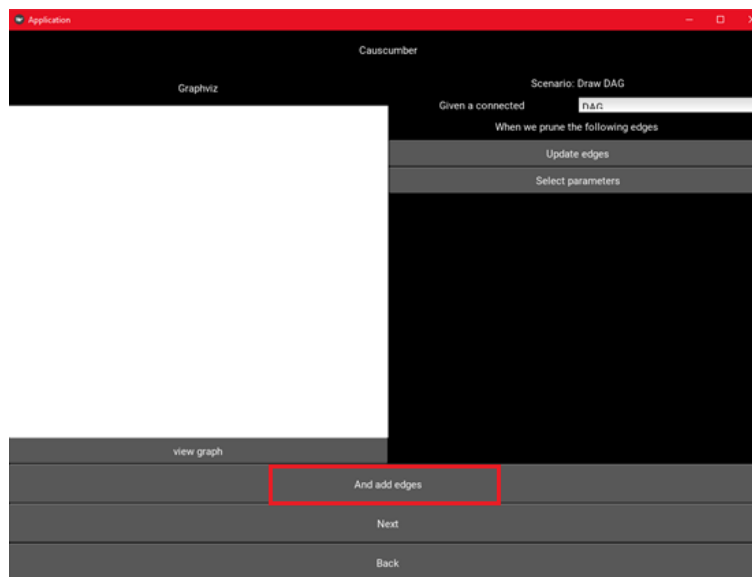


Figure A.11: Select add additional edges.

Figure A.12: Scenario outline for testing Covasim.