University of Glasgow | School of Computing Science

Code Fuzzing for Sociotechnical Variance

Tom Wallis

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — 25th March 2016

Abstract

Currently, testing a model of a sociotechnical system often does not include the uncertainty of action that comes with unreliable human actors. Human actors can be unreliable because their behaviour can be non-deterministic as a result of stresses that act upon them. This non-deterministic approach can be time consuming and produces models that are made particularly complicated by having this uncertain nature built into the model's mechanics.

A better approach might be to automate the human uncertainty involved by taking a deterministic "blueprint model", composed from the expected behaviour of the human actors, and made from some procedures that represent the actors' workflow. Then, when the human actors perform incorrectly, some code mutation technique is employed to enforce some random changes to their workflow, representing sociotechnical stresses. In this way, the concerns of the model's workflow and the actors' unreliability may be separated, and the model can be simple and human-readable while retaining the accuracy of more complex non-deterministic modelling techniques.

This report will explore the feasibility of mutating a sociotechnical workflow, so as to produce a "blueprint model" without this uncertainty built in, and will explore what such a model might look like.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

Chapter 1

Preliminaries

1.1 Introduction

When a person wants to make a model of some real-world situation, lots of things need to be accounted for. Moreover, different models might be concerned with very different things. For example, a model of some physical system and the forces acting on it can be relatively easy to model, compared to a street on a busy day or the inner workings of a pancreas. There are also different types of complexity: where the pancreas plays host to several rather complex physiological interactions, other systems are less physical in nature. The busy street requires a degree of social modelling to model accurately, for example.

A model of a system with interactions between social and technical things, which also interact with their environments, is termed a *sociotechnical* model. A sociotechnical model has unusual complexities involved in its modelling, such as people sharing information, or how they go about achieving a goal should a piece of equipment break. Unlike physical modelling problems, where the physics and mathematical laws governing how things act are well known, sociotechnical modelling has another degree of complexity in understanding the rules and guidelines that shape the behaviour of a person or group with their technology or environment.

1.2 Context

As a result of this high degree of complexity in sociotechnical modelling, the practice of creating sociotechnical models is notoriously difficult. This is especially true when trying to model the way that humans in these models act under stress. This is important to model for accuracy's sake, because these stresses can subtly affect the way people operate, and even small changes to sociotechnical models can greatly affect the product of a model[1].

As a result of the rapidly increasing degree of complexity, and the difficulties involved in simulating very important subtleties, sociotechnical modelling is a very difficult thing to do well. Regardless, sociotechnical modelling also has its uses, from mapping out the UK's National Health Service's IT systems[2] to ethnography[1] and even risk analysis[3].

1.3 Problem

When sociotechnical modelling gets very complex, introducing the unreliability of its human actors becomes very difficult. It would be very useful to be able to model sociotechnical systems simply, and then introduce the unreliability of the human actors in the model as imperfections in the simulation.

There already exists a technique for introducing imperfections into programming code: *code fuzzing*. Code fuzzing introduces changes to code to confirm that a programmer has written their tests appropriately. If the tests pass even when the code has been made bad, then the original code may contain bugs even if the tests pass.

This technique is interesting, because it suggests that it may be possible to produce these imperfections in a sociotechnical model, if the model is written in programming code that can be edited using existing methods. If those imperfections can be introduced in such a way that they are representative of the imperfections in human activity, then this might simplify the model created and help us to create more complex models without the overhead complexity of actor unreliability.

A method for creating this mutated model, representing human-like inconsistency in the code, and creating models suitable for mutation must therefore be made to determine whether code fuzzing can really simulate these inconsistencies, and whether this technique can be feasibly used.

1.4 Terminology

For clarification of terms within this report, some definitions of terms used have been provided:

Sociotechnical System: A sociotechnical system is a system that includes complex interactions between humans, machines, and the environment around them.[4]

Sociotechnical Variance: Sociotechnical variance is the degree of change or non-determinism within a sociotechnical system.

Sociotechnical Environment: The environment a sociotechnical system exists within, which can affect its emergent phenomena and its behaviour in general. A sociotechnical system typically affects its environment to at least some small degree.

Emergent Phenomena: A perceived activity in a sociotechnical system that emerges out of the system's complexity, rather than being a part of the system explicitly added to the model.

Sociotechnical Stress: Some social or technical force influencing the activity of a sociotechnical system. This would often be either built into a model, or added after the fact, if a model is concerned with stresses.

Code Fuzzing: Altering source code or inputs to some black-box system to test what happens, often at random[5]. For example, one might alter a sociotechnical system to make some changes at random to the model and perceive the outcome.

Mutation Testing: Use of code fuzzing to verify tests. If a test passes code that has been mutated, the test might have a flaw, as it is passing code that should no longer perform its original purpose.

Process Fuzzing: Altering some process, as one might some code, to perceive the effect the change results in. This report focuses specifically on process fuzzing, but because of the similarities with Mutation Testing and Code Fuzzing, these terms are sometimes used here to mean Process Fuzzing. Therefore, if a fuzzing/mutation system is mentioned, and no meaning is explicitly stated, it should be assumed that Process Fuzzing is meant.

Procedural Model: A model made using procedures and programming code, rather than photographically or in a markup format.

Actor: Anything which has agency within a sociotechnical system. Actors perform activities within the sociotechnical systems they operate in, and anything that has activity associated with it is an actor.

Directed Acyclic Graph: A Directed Graph is a graph[6] with edges that point from some source node on the graph to some target node on the graph, and can only be traversed in the direction of the source to the target. A Directed Acyclic Graph is a Directed Graph with no Cycles, i.e. from any node it is impossible to traverse the graph such that that node is reached again.

DAG: Abbreviation for Directed Acyclic Graph.

Chapter 2

Motivations

Sociotechnical modelling is important: it can give insight into organisations as small as startups and as large as governments and multinational corporations. It has many applications, too, as different groups use it with different aims in mind. Some use sociotechnical modelling to make more efficient workflows[7], while others use it to create more effective end-products or to analyse the potential products of a system.

Sociotechnical systems have been widely adopted for these purposes. For example, adoption of ISO standards for state algebras like Z[8] and modelling techniques made by and for government use such as SSADM show that there is a concerted international push for sociotechnical modelling techniques. Clearly, sociotechnical systems modelling is regarded as a very important practice for all sorts of organisations. From the commercial end of the development spectrum, BPMS has been created for modelling business processes and workflows[9].

Unfortunately, despite its usefulness, sociotechnical modelling remains very difficult. There are many aspects to this difficulty. For example, a sociotechnical system contains lots of detail, not all of which can be modelled all of the time. Vagueness in sociotechnical models is an example of something that remains difficult to accurately capture in a working model[10].

Particularly difficult to model can be the minutia of a system. Small details can have a big impact on the system's state[1], which can make sociotechnical systems modelling difficult from an ethnographic perspective: how does one capture the uncertainty of human actors in a sociotechnical system? We might be able to create non-deterministic models of sociotechnical systems, or to create detailed sociotechnical systems that attempt to account for all possible actions, but it can be difficult to capture the smaller details about a system, on the level of human behaviour. Human behaviour can be uncertain in many different ways, and crucially, alters depending on the stresses the actors experience at any one time.

Some people are modelling human uncertainty to some degree, but this often centres around the concept of vagueness of model instead of the uncertainty surrounding human behaviour. These approaches attempt to leave elements out of a model, rather than inserting a non-deterministic randomness that represents sociotechnical stress. Hermann & al.[10] give an example of vagueness in a model which allows for flexibility in the model's creation, but ultimately gets us no closer to modelling human unreliability. The concept of taking a model of the human behaviour and tweaking it seems to be unrealised as a research area.

This begs the question: what does a model of sociotechnical stress look like? Moreover, how might one compose the model and specify that sociotechnical stress?

This inspired a project to take a model of behaviour as intended as a blueprint of the realistic sociotechnical model, which is then made imperfect *during* the simulation using some technique to alter the model. A technique to alter code already exists in the software engineering community – process fuzzing, or code mutation[5] – and so utilising this for introducing alterations to a model might prove effective.

Code fuzzing is interesting because those edits made might represent *changes in behaviour*, given we would be altering a program that represents the behaviour of some actors. However, this approach also raises some questions. For example, would it be feasible to represent an actor's workflow as a regular procedural program in a commonly known language? Furthermore, would this code fuzzing approach to simulating sociotechnical stress be particularly effective, or is another approach necessary? These questions construct the hypothesis of the project:

Can code fuzzing feasibly be used to simulate stress within a sociotechnical model?

2.1 Aims

To test this hypothesis, three components will be required:

1. Some model to fuzz
2. Some mechanism by which fuzzing can happen appropriately
3. Some examples of fuzzing that might represent sociotechnical stress

The fuzzing should be done in such a way as to represent sociotechnical stress on an ideal system, so attention should be paid to the type of fuzzing implemented and how it affects the code that makes the model. Many fuzzing systems employ fuzzing to source code which is then run, but because some procedural code will represent actions an actor performs multiple times, it may be the case that the fuzzing should occur during the running of the code, such that an "action" might have a different effect on the system when performed many times. The fuzzing technique might also want to account for the innumerable ways human actors can be unreliable, so a requirement of that fuzzing system might be that it be extensible to account for any arbitrary change to the workflow. An exploration of the tools that already exist and how they meet these requirements can be found in the research chapter of this dissertation (§3).

The models made to test the hypothesis should be made in a way that permits fuzzing in some meaningful way, and also makes the effect of the fuzzing on the system clear to some degree. The construction and constitution of the models made therefore affect the testing of our hypothesis. These models will therefore become a necessary component of the research, as they cannot reliably be taken from other

sources, to ensure that their content and construction is suitable for the fuzzing methods implemented. We must be able to confirm that the fuzzing makes sense in our sociotechnical environment.

Some other aspects of this project will require discretion. An example would be code fuzzing and the Halting Problem[11]. Will it be possible to guarantee, after mutation, that a model will execute? This might be the case if one is to assume an appropriately constructed model to begin with. A sociotechnical model which does not initially break might guarantee a code mutation that is valid. Moreover, we might want to build into our models some construct that allows for monitoring the sociotechnical environment such that we might alter the system's execution automatically, possibly fixing some halting problem issues if our simulation is getting notably out of hand.

As can be seen, lots of details affect the outcome of this project. Therefore, the main focus will be process fuzzing and the properties of the experiment that affect this, so as to more directly answer the hypothesis, which also has process fuzzing as its primary concern.

2.2 Outline

This report will be broken down into several chapters:

3. Research into existing platforms
   This will serve to lay out the requirements of the system that needs to be made, as it is important to know what research must be done before carrying it out. The requirements of the tools are identified.

4. Modelling System Created
   Here, the structure of the modelling system created is laid out, so as to define a type of model that adequately meets the requirements of our project.

5. Fuzzing Library Created
   Here, the code fuzzing library is documented, and its implementation is discussed, so as to show how the requirements of our fuzzing library were adequately met.

6. Evaluation
   The tools are used to create a model and run it, testing the platforms created in the previous chapter and then using this to evaluate the hypothesis.

7. Future Work
   Once the hypothesis has been evaluated, potential next steps for this line of enquiry in sociotechnical modelling is discussed.

8. Conclusion
   The outcome of the experiment is discussed, as well as the potential for code fuzzing for sociotechnical variance and some personal reflection on the project.

Chapter 3

Research

With regards to the hypothesis of the project, there was not much literature revolving around the introduction of variance to a sociotechnical model, rather than taking variance into account when creating the model. For example, Herrmann & Loser[10] discuss sociotechnical modelling with inherent vagueness, and while this might be shown to be a useful construct, it doesn't address the problem that sociotechnical modelling with uncertainty of action is difficult to model. Instead, Herrmann & Loser create sociotechnical models using pictorial representations – but these may be difficult to mutate programmatically. Therefore, a comparison of current methods for representing sociotechnical systems and introducing variance was necessary.

3.1 Modelling System

Modelling System Requirements

The modelling system desired had some properties that could not be found in the available alternatives. For example, the models had to be procedural, but also easy to write and maintain. It was decided that, with the work done in software engineering to maximise the maintainability of code, a mainstream and well-documented programming language should be chosen. To that end, and for other reasons discussed below, Python was the candidate of choice for this attempt.

The programming language chosen was also to be powerful, because potentially rather complex models would be created with this modelling system. However, Python allowed for this, as can be seen by the large software development projects undertaken in Python, such as components of PayPal[12]. The fact that Python is a very powerful, very portable language that allows for readable codebases at scale was a strong influencing factor when deciding what language would suit the construction of a sociotechnical system.

Another feature of the models that needed to be implemented was that they must be modular. With Python being a language often chosen for large codebases, and being a language often used to deploy webapps using frameworks such as Django[13], Python allows for the creation of modules and packages that makes it easy to bring a workflow written for a subsection of a system and integrate it into a larger model of that system.

It is worth noting that there are problems with this approach. For example, when introducing two systems together, their interaction is a part of the atoms the system defines. However, with each component being created separately, neither system would initially be able to interface to another. Middleware layers for sociotechnical systems are discussed in the future work section.§7.2

A lesson learned from research on i* (found at §3.1.3) was that the interactions between different people makes a big difference to the end result of the sociotechnical simulation to be built. Therefore, to simplify the end-product, only one actor should be simulatable at a time in the modelling system that is ultimately created. Any more would add complexity to the project, and one actor should be sufficient for modelling a workflow that can exhibit sociotechnical variance. This limitation will mean that any sociotechnical stress that occurs as a result of interplay between actors cannot be simulated.

3.1.1 Currently existing platforms

As a result of the desire to create programmatic models to mutate, it was decided that a purely pictographic representation of a sociotechnical system could introduce some complications to the research. In particular, with the project's focus being on the affect of code fuzzing and whether it represents sociotechnical variance, it was decided that a modelling system that lended itself to code fuzzing was the biggest concern. If code fuzzing did lend itself to better modelling social variance in sociotechnical models, then the technique could be applied to other modelling platforms, too.

3.1.2 KaOS

KaOS is a goal-oriented modelling technique for sociotechnical systems modelling[14]. This seemed like an appropriate place to begin, as KaOS is heavily used and cited in the academic community. The actors we model are attempting to achieve or fulfil some goal, so surely this is an appropriate place to start researching current modelling techniques.

However, there are some issues with KaOS for the purposes of our models. For one, KaOS is useful if one must elicit software engineering requirements from a client, but in terms of modelling human actors, KaOS is less widely used. Requirements engineering techniques would be inappropriate for a model we intend to fuzz, even if a representation of the goals might also be mutatable.

Ideally these models should be easily verifiable by a human, so either a graphical or human-readable textual format would be good to represent these, but KaOS lends itself well to graphical representations that seemed hard to execute as some simulation. Therefore, while widely taught and used, KaOS doesn't meet our requirements for modelling.

3.1.3 i*

i* is another technique that, while useful for gaining an insight into some sociotechnical system, is ill-suited to our needs as a result of its requirements engineering background[14]. In i*, actors rely on each other for goals and responsibilities to be fulfiled. i* therefore represents a better social model for a system than some of its competitors, and is widely cited and popular in the academic community, similar to KaOS. Both are used in situations as different as software engineering[15], artificial intelligence[16], and even comes up in journals for infectious diseases[17].

As a result of i*'s better social modelling than KaOS, we get a slightly different look a sociotechnical modelling and requirements engineering. However, as widely cited and interesting as i* might be, it suffers the same fundamental flaws that KaOS does, the largest of which being its lack of workflow modelling. It is clear to see that interactions between people is an important component of any sociotechnical modelling platform, however. Due consideration should be given as to the social implications in a sociotechnical model and whether any sociotechnical stresses that arise from the social interaction would be useful for this proof of concept project.

3.1.4 YAWL

YAWL was particularly interesting because of its focus on workflow. YAWL, which stands for Yet Another Workflow Language[18], is a workflow modelling platflorm with its own tools, which are sophisticated and multi-platform. The creators of YAWL also have interests in areas such as π-calculus[19], which makes YAWL interesting for the additional reason that it has some mathematical backing in its representation of workflows. With this said, the lead authors have also published work stating that workflows are much more than simple π-calculus processes[19].

While workflow is our main concern when searching for candidates to perform fuzzing on, YAWL is largely constructed with a graphical tool, making it difficult to find things to mutate. Ergo, YAWL would be inappropriate as a candidate for fuzzable systems. However, it does get us closer to the modelling system we require, and shows that there is some academic interest in workflow-oriented modelling systems.

3.1.5 OBASHI

OBASHI[20] is a modelling system that focuses on business processes instead of sociotechnical models. However, OBASHI's business process modelling techniques focus on dataflow, meaning that systems have states that get changed as a simulation occurs. OBASHI also focuses on simplicity in its models, which is something that would be required by a sociotechnical modelling framework to ensure that the models created for this project would be at least moderately accurate representations of the systems as a whole.

While little academic literature is available on OBASHI due to its strong commercial focus, it supports an idea of hierarchy between different components of a system with its layered Dataflow Analysis Views[20]. Given that dataflow and workflow have clear correlations, it might be prudent for a fuzzable sociotechnical platform to seperate its own concerns by using layer models.

Indeed, focusing away from the academic perspective, many modelling systems are used commercially that are not open to scientific analysis. Few pieces of academic literature exist at all for OBASHI, despite its 15 year existence. Therefore, it might be prudent to look to the commercial sector and analyse what works in the practically minded business world to create mappings of systems. One might rely on the common market to select mapping systems that are reliable and functional, so long-lasting methods with interest from industry might give insights into better sociotechnical models in the future. For the purposes of this project, such a detailed modelling system would not be worth the time required to build it, as the hypothesis to be tested centres around code fuzzing.

3.1.6 Code Fuzzing for Sociotechnical Variance

Little academic research appears to have been done on programmatically inserting sociotechnical variance. A little work has been done into systems for *Punctuated Socio-Technical Information System Change model*s[21], but this appears to refer more to the way information systems alter over time and less to do with the study of uncertainty in information systems.

Other research has been done into the management of organisational uncertainty[22][10], which appears to be ideal for the task at hand. Unfortunately, no sociotechnical simulations with uncertainty injected into the models created could be found.

Therefore, while sociotechnical modelling is now a well-established field with competing methodologies, and sociotechnical uncertainty has also been a research subject since as early as 1976[23], no programmatic sociotechnical modelling has been documented which models sociotechnical stress as a component of that model. Therefore, the hypothesis of the project is clear; that is, to construct such a system and attempt to verify that sociotechnical stress can be introduced to the system by means of code fuzzing.

3.1.7 Introducing Variance

Currently, variance must be introduced to a programmatic model by hand. That is, as a construction of a model, we must build the sociotechnical stress into the model rather than applying it after the model has been created.

3.1.8 Graphical representations and UML

Pictorial representations of sociotechnical systems can be very human-readable, which makes them suitable candidates for creating a sociotechnical model. Modelling systems such as OBASHI[20] use graphical representations of sociotechnical systems to build models of business and IT processes, but build these models using dataflow. To contrast, a UML Use Case diagram[24] models an actor within a larger system to visualise a sociotechnical system. Both models use graphical representations of the systems they model to make the complexity of the model easy for a human to parse, but these models can be difficult for a computer to parse.

UML tries to overcome this by being equally readable by a computer: frameworks like the Eclipse Modelling Framework can generate Java code from a UML class diagram, for example[25]. UML also succeeds in creating a standard for colloquial ”flowcharts” by creating well-defined specifications such as the UML Activity Diagram. Therefore, using UML as a format for laying out a sociotechnical system that can be easily created and read by humans, but is also easily computer-parsable, seemed to be an appropriate way to create the programmatic models.

3.2 Fuzzing Library

Having researched what was already available in the academic and commercial spaces, it was decided after some deliberation that the best option was to define our own modelling system and create our

own fuzzing library.

3.2.1 Fuzzing Library Requirements

The fuzzing library desired also had unique properties that made it difficult to find an equivalent for in a commercial or academic setting.

The fuzzing library was of particular importance because the main study of the project was on variance in sociotechnical systems. Therefore, the fuzzing library had a particularly strict set of requirements. the fuzzing library needed to have fairly precise control over the activity of the mutations, because the mutations it introduced needed to be representative of sociotechnical stress. Ideally, it would be possible that the mutation testing would not be a black box, so that the exact nature of the sociotechnical stress could be discerned. Therefore, if a custom library was not to be made, then the chosen library would need to be open sourced.

It transpired that the library would need to be built to spec anyway, because the library needed to generate different mutations on every call to the functions it was fuzzing. This meant that not only would specific parts of the procedural model need to be targeted, but the library would either have to intercept function calls somehow or fuzz the function to randomly select one of many fuzzed versions of itself at execution time. This functionality was *critical*, because human variance in a workflow is to change the behaviour in some random way every time that behaviour is executed. With flows and atoms acting as blueprints for behaviour, these blueprints would need to be altered when enacted, just as in real life, to make a meaningful simulation. No available code fuzzing library could be found that could do fuzzing during the execution of the fuzzed program, so a custom library needed to be built.

With the fuzzing library being such a critical part of the research, none of these points could be compromised on. In addition, a custom built library would allow fuzzing methods to adapt and grow as the modelling method matured. Therefore, a custom fuzzing library was made.

3.2.2 Existing Fuzzing Platforms

Because the systems we were interested in creating were to be self-documenting, we chose Python as a language to write our models in. Python's clarity combined with the ease of quickly writing working Python code meant that self-documenting models could be feasibly made. In addition, Python supplied language features such as decorators and a built in Abstract Syntax Tree library that made it an ideal candidate.

We therefore set about finding appropriate fuzzing libraries in Python, and found several:

Sulley

Sulley is a fuzzing library for Python which is popular for remote application and protocol fuzzing. Sulley alters the protocol by which it inserts data into an application or system, and observes how that application reacts to the altered input, to simulate problems with networks and human users. It is under active development, and is open sourced.

While using a popular, open-source library for inspiration when creating our own mutation system, Sulley was inappropriate for our needs.

Sulley fuzzes by way of changing protocols and inputs to some system it interacts with. We felt that altering inputs to that system lacked the degree of control that changing workflows directly would provide. Technically, it would be possible to get around this problem by creating models in some remote program that took instructions from some fuzzed input, but we felt that this additional degree of abstraction would lower the self-documenting nature of our models and would be an impractical modelling technique anyway.

Fusil

Fusil was another fuzzing library that was investigated. Fusil allows for fuzzing the environment of a python program, which made it appealing to us. While environmental factors impact the sociotechnical model however, these factors are unrelated to the properties Fusil is capable of monitoring.

Fusil is able to create mangled files as fuzzed/invalid input to a program, monitors and limits CPU and memory usage, and checks logs for system errors and other signs that a fuzzed program was not functioning correctly under adverse circumstances. This seemed appealing, because monitoring performance under adverse circumstances is precisely our goal in this project.

Unfortunately Fusil also did not meet our requirements: it monitors the effect of a change in computational environment on a program, where we are interested in the effect of the change of a social environment on a workflow. While our workflows are being modelled as programs, this does not mean that the workflows themselves would be being altered, so stress on a sociotechnical system still wouldn't be being modelled. Fusil was a further worse choice when taking into account the fact that we would not be able to pinpoint the errors that were being introduced by observing memory and CPU usage, and other computational concerns.

Therefore, we could not use Fusil as our fuzzing system of choice.

MutPy

From the MutPy package page[26]: MutPy is a mutation testing tool for Python 3.x source code. MutPy supports standard unittest module, generates YAML reports and has colorful output. It's apply mutation on AST level. You could boost your mutation testing process with high order mutations (HOM) and code coverage analysis.

It seemed that MutPy supplied *everything* required. This would be beneficial, but unfortunately, MutPy had a relatively large and complex codebase[27] that made it hard to modify for the specific requirements of the research. In addition, relying on a complex dependency that is designed for unit testing seemed to be a bad approach, as research interests might not align with MutPy's maintainers' in the future. As a result, it might have been necessary to fork their codebase at the time and maintain our own version for research purposes; this approach was not future-proof, and so an alternative approach was deemed necessary.

Working in MutPy's favour, however, was that it was a *code fuzzing* rather than protocol fuzzing library. Particularly, it fuzzed source code to a similar end as something like Java's PiTest[28]. This

was more in line with our goals, although MutPy still didn't deliver what was required from a code fuzzing library for this experiment. Therefore, it was decided to learn from MutPy's strong points and implement something similar, but from a fresh, maintainable codebase with interests more aligned with that of the research. This would also allow for potential improvements to be implemented in the future, which could not be determined before modelling began; as it happens, this decision turned out to pay off, as additional functionality was needed(§5).

PyMuTester

PyMuTester[29] proved to be the closest to the library required, though it still fell short in some areas. Particularly, PyMuTester did not appear to have many types of mutation it could enact.

PyMuTester did not have a large codebase[29], nor did it have very much functionality to offer, but it offered mutation via Abstract Syntax Tree editing. The decision was made to study the PyMuTester source as a template for a code fuzzing library of a new design. PyMuTester seemed to be much too immature as a library to base the whole project around, and was last updated in October 2014. However, it did show promise, and gave a basis around which writing a code mutation library could be built, while also taking cues from the more advanced source of MutPy[27].

Chapter 4

Modelling System Implementation

As previously discussed, the project's development lay in three key areas:

1. Some model to fuzz
2. Some mechanism by which fuzzing can happen appropriately
3. Some examples of fuzzing that might represent sociotechnical stress

The latter two being tools required to create the first. Therefore, to test the hypothesis, a fuzzing library and modelling system it could work on were created. Here, we will discuss the design of a suitable modelling system, and give examples of its constituent parts.

4.1 Modelling System

The modelling system was designed to make both creating and fuzzing models easy. As no modelling systems in the research phase were found to be adequate for the project, a new one had to be designed that was suitable.

This modelling system needed to have a few traits. As a result of it being written by hand, and being a full sociotechnical model in programming code, the language used had to be very human readable. In addition, the language needed to have easy and accessible tools for creating fuzzing libraries. Also, this language needed to be easy to create relatively complex models in, so the language itself needed to be easy to use. Lastly, a well known language was desirable, as people writing models would need to know the language. In this scenario, the models were being made for the dissertation project, and there was insufficient time to learn a new language for modelling over and above creating the fuzzing libraries. Only one language – Python – met these criteria, so Python was the language chosen to create models in.

Currently, tools such as Eclipse Modelling Framework[25] can create code from a UML diagram, so basing the models on UML seemed sensible; in the future, large parts of model creation in this format might be automated, because a graphical representation can be converted into a procedural model for the fuzzing technique. As a result of the project's heavy focus on workflow modelling, UML Activity

Diagrams[24] were chosen as a representation of workflow. UML Activity diagrams have long been a common way to represent workflow models[30]. As a result of the simplicity possible in the models created to test the hypothesis, a complete adaptation of the UML specification for Activity Diagrams to Python code was not pursued. Instead, determining how to adapt basic flow structures from Activity Diagrams was all that was required to create sufficiently complex models to test the hypothesis. More complex adaptations of the UML Activity Diagram specification, such as concurrent activity with fork and join notations, is discussed in the Future Work section.

4.1.1 Layer Model

To separate concerns and make fuzzing safer and easier, two critical concerns were separated in the model creation section: the flow of work, and that work's effect on its sociotechnical environment.

Flow Layers

It was identified that descriptions of work were generally given on a high level, and broken down to finer detail when models were considered in depth, at the beginning of the research. As a result, it made sense to regard certain patterns of work as sub-workflows that are chained together by some overarching workflow.

Furthermore, there was no reason why these sub-workflows could not be nested to a fine degree of detail. It was determined that any "flow" of work does not affect the sociotechnical environment in-and-of itself, but that it simply dictates the order in which the effects of sociotechnical work are introduced to a system. At the finest grain of detail for a given model, another component, called "atoms", affect the sociotechnical environment and these are chained together by the flows to create meaningful workflows.

A flow of work can also be considered a collection of connected nodes on a UML Activity Diagram. These areas of work are small operations that are, in real life, broken down into smaller tasks, but finer detail is still needed to accurately model the sociotechnical detail[1].

Where this fine detail becomes important is in creating a model structure appropriate for fuzzing. If we want to introduce code fuzzing when actors perform a particular activity, it is important to be able to target that particular activity as a subsection of the whole model and isolate that activity. Such scenarios could be the case where an actor is a newly trained employee and is likely to miss a step in a certain process they were badly trained in. In this case, we might expect them to successfully complete the rest of the model, but to occasionally skip a node in a sociotechnical flow and move straight into its successor. This can be done by isolating the activities a sociotechnical flow is made of, creating each minor activity (the *atoms* described below), and chaining them together into some code which is to be marked as fuzz-able.

Flows are to be made from a high-level perspective, and then broken down into sub-components until a model of sufficient detail is constructed. This way, one can construct a model in a natural way: considering the activities people perform on a day-to-day basis and drilling down to finer detail over time. To take the example of the model implemented later in this report, one might describe the agile method of software engineering, after requirements gathering, as:

- Implementing features

- Unit testing
- Integration testing
- User acceptance testing

Which, as a flow, might look like this:

Code sample 4.1: Example high-level flow from a series of functions

```
def implement_feature():
    make_new_feature()
    create_test_and_code()
    unit_test()
    integration_test()
    user_acceptance_test()
```

When we break down one of these steps in the larger workflow, we get a more fine-grained view of what ”unit testing” entails:

Code sample 4.2: Fine grained view of the earlier top-level workflow

```
def implement_feature():
    make_new_feature()
    create_test_and_code()
    unit_test()
    integration_test()
    user_acceptance_test()

# Descriptions of model created and environment module below
def unit_test():
    run_tests()
    while not environment.resources["unit tests passing"]:
        fix_recent_feature()
        run_tests()
```

Now, if one wanted to mutate unit testing but wanted to assume that feature implementation, integration testing, and user acceptance testing were to be reliably executed, the modeller would only need to mutate the unit_test function.

In this way, the models are suitable for fuzzing, while keeping the flow of activity easy to read. In addition, models can be edited and maintained easily, as any large blocks of code are indicators that a flow can be broken down to smaller components.

Atom Layers

Atoms are the finest grain of a model’s detail. Therefore, if an action cannot be broken down into smaller activities, that action is an atom — so named because they cannot be broken down further — and are called by flows to enact change on the sociotechnical environment.

The distinction between atoms and flows that makes them separate layers is that flows make calls to other flows or atoms, but *cannot* change the sociotechnical environment. This is because the order itself of an activity does not affect a sociotechnical environment; the actions of actors in a sociotechnical system do. Therefore, atoms are the only level of detail that should affect the sociotechnical environment.

When changing a sociotechnical environment, we alter its state, which internally in a simulation is to alter values held – to write to a variable. However, to read the values in those variables is to make decisions which are informed by the current state of the sociotechnical system. This is necessary for flows. Therefore, we make a distinction between reading from and writing to a variable: *only atoms are allowed to write to sociotechnical state*. Both atoms and flows may read the sociotechnical state. (The above example of a flow does this in evaluating `environment.resources[''unit tests passing'']`.)

While atoms can change the sociotechnical environment, they should never call another atom. This is because that atom's affects on its environment are specific to the activity it represents; to run another atom after the current atom would represent a small sub-workflow, which is the domain of flows.

Doing this separates the concerns of the two layers, which, while useful for fuzzing purposes, also serves to make the systems more maintainable. Any problems with sociotechnical environment would be an issue with the atoms, and any problem with the ordering of activity or flow of action would be an issue with the flows. The concerns can also be split across multiple files, further improving maintainability and making the creation of a system which represents the real world easier.

As a result of this distinction, another should be drawn: atoms cannot be mutated. This is because the mutation we enact is on a sociotechnical flow or *workflow*, whereas atoms are *activities*. The distinction is similar to that between actions we choose to do and physics itself: we are bound by the laws of physics and cannot change this element of our own environment. It is possible, however, to change the things we do within these laws of physics, which are our own workflows. Similarly, sociotechnical atoms are the physics of the workflow: they shouldn't be changed, because if an actor was to do things in the wrong order or forgets to act in certain ways, this does not change the effect that their individual actions have on their environment. Rather, the actions they actually perform are changed. Therefore, every action an actor can perform is that actor's own sociotechnical physics, and an action's effects on its environment do not change. An example of this would be unit testing: if a programmer writes a test, that test is written and is a component of their sociotechnical environment. They may forget to write a test though; this is a change to the programmer's workflow, so it is represented as a mutation in the workflow layer of the model that they run within.

To extend the above example model to include an atom, create_test_tdd, too, one might write:

Code sample 4.3: Sample from a model including flows and atoms

```
#An example flow, does not affect environment
def implement_feature():
    make_new_feature()
    create_test_and_code()
    unit_test()
    integration_test()
    user_acceptance_test()

#An example flow, does not affect environment
def unit_test():
```

```
    run_tests()
    while not environment.resources["unit tests passing"]:
        fix_recent_feature()
        run_tests()

#An example flow, does not affect environment
def create_test_and_code():
    create_test_tdd()
    add_chunk_tdd()

#An example Atom, directly affects environment
def create_test_tdd():
    environment.resources["time"] += 1
    test = dag.Test()
    environment.resources["tests"].append(test)
```

As can be seen, the atom in this example affects the environment of the model in two ways: the activity takes time, incrementing a counter, and a test is created and appended to a list of tests the model is aware of. However, the flows in the model never affect sociotechnical state, and the atom never calls another atom.

Creating a model

The final component of a model is the environment itself. This completes the sociotechnical system, as we only distinguish between the sociotechnical environment and the workflow that exists within it. As a result, the larger sociotechnical environment should not be subject to mutation either.

The sociotechnical model created can, of course, be anything and of arbitrary complexity. A benefit of using Python for model creation is that creating complex information systems in it is relatively easy due to its simple syntax and clean formatting. While a domain specific language might be more useful for this section of the code, as previously discussed, an ordinary procedural language was required for building the fuzzing library and models to test it.

Because the environment itself will be driven entirely by the atoms created, it should not be required to create any sections of the environment programmatically. Instead, classes and variables needed are represented in some format that is global to all files the model is composed of. This way, the same environment is in the scope of all atoms and all flows involved in a model. In the example model created, this was the ”environment” module. Classes it uses are kept in a module called ’dag’, because the model created has a component which is a Directed Acyclic Graph.

As an example, a model could be created from a flowchart like this:

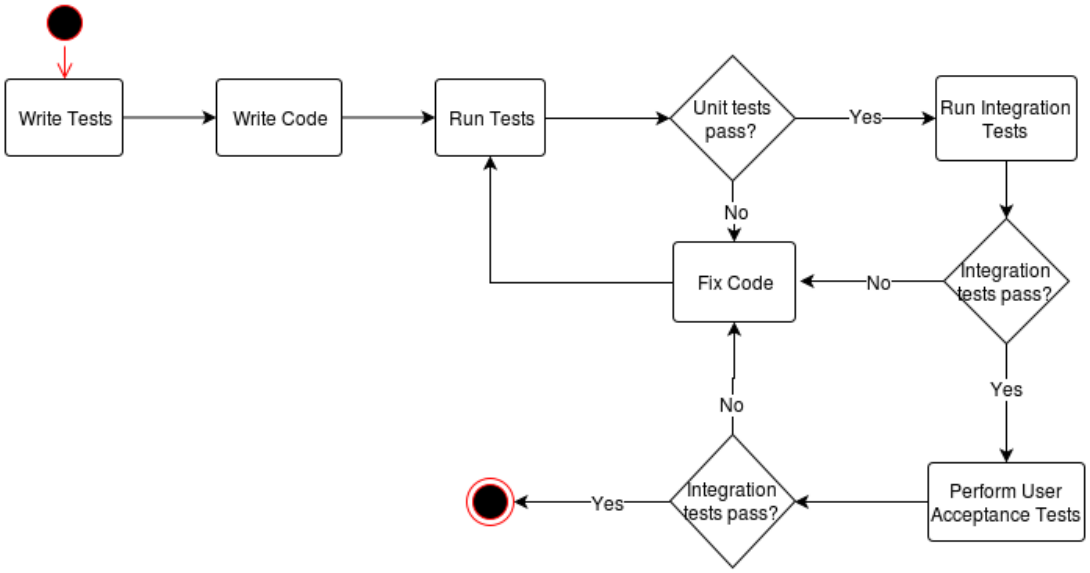

Figure 4.1: A flowchart describing an example Test-Driven Development workflow

And turned into a series of flows like this:

Code sample 4.4: Sample adaptation from flowchart to Python flows

```
def implement_feature():
    create_test_and_code()
    unit_test()
    integration_test()
    user_acceptance_test()


def make_new_feature():
    create_feature()


def create_test_and_code():
    create_test_tdd()
    add_chunk_tdd()


def unit_test():
    run_tests()
    while not environment.resources["unit tests passing"]:
        fix_recent_feature()
        run_tests()


def fix_recent_feature():
    for feature in environment.resources['features']:
        for chunk in feature:
```

```
            fix_chunk(chunk)


def integration_test():
    perform_integration_tests()
    while not environment.resources["integration tests passing"]:
        unit_test()
        perform_integration_tests()


def user_acceptance_test():
    perform_user_acceptance_testing()
    while not environment.resources["user acceptance tests passing"]:
        unit_test()
        integration_test()
        perform_user_acceptance_testing()
```

For the sake of brevity, atoms and environment have been omitted from this example, as these are much larger than the flows. Functions not defined here are components of the atom or environment layers.

Chapter 5

Fuzzing Library Implementation

The implementation of the fuzzing library was broken down into two distinct parts.

The first was a wrapping function that would allow a modeller to determine which of their functions were to be mutated. This was done by employing *decorators*. *Decorators* are a feature of Python that allow for a clean notation to wrap a function in another function. The wrapping function (the decorator) is passed a function object when the original function is called by a program, and is expected to run the original function around some other work or to conditionally run the function if some preconditions were met. This is often employed for URL to controller function routing in web app frameworks like Django[13].

For our purposes, the decorator would run a mutated version of the original function, meaning that every time the function was called, the decorator would generate a mutation of that function from some source code. Upon generating this fuzzed version of the function, the decorator would call this rather than the original function, thereby running a variant that simulates the unreliability of human actors in a sociotechnical system, as per the library's requirements.

The second section of the library would be the mutation class itself. To do this, inspiration was taken from PyMuTester[29], which creates an Abstract Syntax Tree from source code and modifies the source for the purposes of analysis. Another reason for continuing to use Python was the availability of an Abstract Syntax Tree library built in to Python[31], which PyMuTester also used. Where PyMuTester used a Python NodeVisitor to collect information about source, however, we utilised a Python NodeTransformer to edit an abstract syntax tree and return a mutated version.

5.1 A note on functions and Function objects

In Python, a function is really a Function object. This is because, while Python is interpreted, it will compile a function definition for reasons of efficiency and operations internal to the Python interpreter; Python uses the Function objects it has compiled internally, but because that function object is also in the scope of any program with that function definition, the Function object is available to the programmer, too. Therefore, when we make a function call, like so:

Code Sample 5.1: A basic Python example

```
def chunky():
  print 1
  print 2
  print 3

chunky()
```

Python uses syntactic sugar here, an addition to the language that makes it easier for humans to read and write. Internally, Python changes this to a `.__call__()` method:

Code Sample 5.2: Demonstrating Python's __call__() syntactic sugar

```
def chunky():
  print 1
  print 2
  print 3

chunky.__call__()
```

Furthermore, we might have a function that we want to parameterise with the operations of another function, for which we might pass in a Function object as a parameter:

Code Sample 5.3: Python function objects can be passed and run, like a function pointer

```
def add(number1, number2):
  return number1 + number2

def multiply(number1, number2):
  return number1 * number2

def operate(func, number1, number2):
  return func(number1, number2)

print operate(add, 4, 5)
print operate(multiply, 4, 5)
```

This would use the Function object passed as an argument to calculate and print:

```
--> 9
--> 20
```

One might think of a python Function object as the closest thing you might get to an actual function pointer in Python. This is important, because when we use Python decorators, they will wrap a function call in the decorator. That is to say,

Code Sample 5.4: A basic decorator example

```
@myDecorator
def myFunction():
  print 1
  print 2
  print 3

myfunction()
```

Will be interpreted by Python as:

Code Sample 5.5: How Python interprets a decorated function call

```
@myDecorator
def myFunction():
  print 1
  print 2
  print 3

myDecorator(myfunction)()
```

Here, myDecorator is a function that receives a Function object as a parameter, and *returns another function object*. In Python, we can write a function that defines a function, because internally Python is simply creating a new object. Therefore, we can return function objects in the same way we might list objects or vectors or File objects. Python uses this to fulfil its decorator pattern, which we employ extensively for readable mutation systems, and for configuring our mutations themselves, as described below(§5.2.2).

5.2 Decorator

5.2.1 Code and functionality

The code for a decorator was fairly simple. Python allows us to write decorators in two ways: as a function that receives a function object and returns a function object, or as a class with a `__call__` method that returns a function object and a constructor that receives a function object. Using the class definition allowed us to create a decorator which accepted parameters, which we use for parametrising the mutation functionality(§5.2.2). Therefore, the mutator could be constructed as simply as:

Code Sample 5.6: The ”mutate” decorator

```
class mutate(object):

    cache = {}

    def __init__(self, mutation_type):
        self.mutation_type = mutation_type

    def __call__(self, func):
```

```
        def wrap(*args, **kwargs):

            if environment.resources["mutating"] is True:

                # Load function source from mutate.cache if available
                if func.func_name in mutate.cache.keys():
                    func_source = mutate.cache[func.func_name]
                else:
                    func_source = inspect.getsource(func)
                    mutate.cache[func.func_name] = func_source
                # Create function source
                func_source = ''.join(func_source) + '\n' + func.func_name + '_m
                # Mutate using the new mutator class
                mutator = Mutator(self.mutation_type)
                abstract_syntax_tree = ast.parse(func_source)
                mutated_func_uncompiled = mutator.visit(abstract_syntax_tree)
                mutated_func = func
                mutated_func.func_code = compile(mutated_func_uncompiled, inspec
                mutate.cache[(func, self.mutation_type)] = mutated_func
                mutated_func(*args, **kwargs)
            else:
                func(*args, **kwargs)
        return wrap

    @staticmethod
    def reset():
        mutate.cache = {}
```

This is, in fact, the entirety of the mutate decorator used in the fuzzing library. The code is explained below.

Now, because the class has a `__call__` method, Python's syntactic sugar will work; that is to say, Python will interpret `mutate()()` as valid Python, even though we are technically calling a Class object. This will construct the class (by initialising it), and then running the function associated with the call with `mutate.__call__()`.

However, our mutation function has a parameter, which is the mutation function used by the mutator to alter the source code. The functionality here is explained later, but this means that we now have a decorator that can take arguments, too. So, usage of this decorator would look like this in a model:

Code Sample 5.7: Usage of the mutate decorator

```
import random

def deadline_stress(lines):
  lines.remove(random.choice(lines))
  return lines

@mutate(deadline_stress)
```

```
8  def unit_testing_flow():
9    print "I "
10   print "am "
11   print "mutated!"
```

It is evident from the decorator notation above the definition of `unit_testing_flow()` that this flow is being mutated, and that the format of its mutation is according to the `deadline_stress` function. Internally, `mutate()` is being initialised with the Function object `deadline_stress` as a parameter, the usage of which is described shortly.

The decorator's code

The mutate decorator used is identical to that above. Its operation is fairly simple:

We define a function, `wrap`, that will be returned as per the requirements of a decorator in Python. `__call__` takes the function object of the function being decorated as an argument, and `wrap` is passed any parameters to that function. As per the definition of a Python decorator, we return the `wrap` function.

We only create a mutation if the sociotechnical environment has a flag for mutation turned on. Therefore, if that flag is turned off, this decorated function should simply return the original function and exit.

Assuming that the function is instructed to mutate, it loads source code from the original Function object using Python's built in `inspect` module, which contains a function called `getsource`. `getsource` will return source code for the definition of a function as found in the program's source code, complete with decorators. Unfortunately, `getsource` returns source code for the wrong function if an attempt is made to retrieve source code for a function the library has already mutated, so we cache the source code upon retrieval and load from that cache if we have seen the original Function object before. This way, each mutated Function object has its source code retrieved precisely once.

Having retrieved the source code, the source is collected together into a string to be analysed with a call to the function being defined at the end. Notably, this function call is to a function with the original name and "_mod" at the end. This is because, if a function with the same name as that of the original Function object is defined, Python's internal cache of function objects is overwritten with a mutated version of the function. Therefore, to avoid overwriting Python's internal cache, a new name is given to the function. The function definition itself is given an altered name in the same fashion when the mutation itself occurs(§5.3).

We create a new mutator and pass it the mutation function given to the `mutate` decorator at initialisation. The purpose of this is explained in the following sections. Once the mutator is made, it creates an Abstract Syntax Tree of the Function object, traverses the tree with the `visit` method, and an uncompiled Function object is returned.

Once an uncompiled Function object is retrieved, a copy of the original Function object is made so that it can be modified. The compiled bytecode in the copy of the original function object is replaced with the mutated version, compiled using Python's built-in `compile` function. It should be noted that this mutated compiled bytecode is created from the source traversed by the `Mutator` class, which received source code with a function call at the end. Therefore, when the new source code is run, it

will define a new function with the original name appended by "_mod", defines this new function, and subsequently executes the Function object which contains this mutated bytecode.

The `mutate` class also contains a static method, `reset`, used if the class' internal cache of function source should need to be reset when running multiple simulations from their initial state.

5.2.2 Mutation functions

`''Mutation functions''` Have been mentioned multiple times through this documentation so far. A mutation function is a function a modeller makes to define the activity of a mutation, and is used by the `Mutator` class as seen in the next section(§5.3). When a mutator is created by the decorator, it is passed a function that is passed to the decorator; this function is any function in Python that will accept a list containing Python AST Line objects and returns a list containing Python AST Line objects.

This is done because the mutation represents sociotechnical variance, and the mutator operates on the definition of some workflow. Therefore, for the modeller to denote the variance that they want to impose on the system, they must denote the changes made to the workflow's definition when under that sociotechnical stress. This function is then used by the `Mutator` class to alter the workflow in whatever way they require.

This function, when passed to the mutator, is called from the function object to alter lines of code within the definition. Therefore, a modeller has complete control over both the model and the variance from it. This is also one of the reasons why the `mutate` decorator must follow the python Class format rather than a nested function definition, as the nested function definition method cannot accept parameters when used as a decorator.

5.3 Mutator

The Mutator class is actually slightly less complex than the decorator class, but it performs the mutations on the function we have retrieved the source code from. The class itself inherits from the built-in Python AST module class, `ast.NodeTransformer`[32]. A NodeTransformer will traverse an Abstract Syntax Tree and modify it as it goes. As a result of a NodeTransformer traversing the tree recursively, it will "visit" each child of a node passed to `self.generic_visit`. That node has a type according to what exists at that node on the tree. If the NodeTransformer has a method defined, `visit_NodeType`, Python will visit using that function. Otherwise, it will simply call `self.generic_visit` on that child node, which in turn visits all of its children.

The definition of the `Mutator` class is as follows:

Code sample 5.8: Mutator class

```
1 class Mutator(ast.NodeTransformer):
2
3     mutants_visited = 0
4
5     # The mutation argument is a function that takes a list of lines
6     # and returns another list of lines.
```

```
    def __init__(self, mutation=lambda x: x, strip_decorators=True):
        self.strip_decorators = strip_decorators
        self.mutation = mutation

    def visit_FunctionDef(self, node):

        # Fix the randomisation to the environment
        random.seed(environment.resources["seed"])

        # Mutation algorithm!

        # so we don't overwrite Python's object caching
        node.name += '_mod'

        # Remove decorators if we need to so we don't re-decorate
        # when we run the mutated function, mutating recursively
        if self.strip_decorators:
            node.decorator_list = []

        # Mutate! self.mutation is a function that takes a list of
        # line objects and returns a list of line objects.
        node.body = self.mutation(node.body)

        # Now that we've mutated, increment the necessary counters
        # and parse the rest of the tree we're given
        Mutator.mutants_visited += 1
        environment.resources["seed"] += 1
        return self.generic_visit(node)
```

Two pieces of configuration can be passed into the `Mutator` class upon construction:

1. A mutation function object. This will be used to enact sociotechnical stress if required.
   This option defaults to an identity function, so that we can visit the abstract syntax tree without mutating if no option is provided.

2. An option to strip a function definition of its decorators (this is a boolean value).
   For the purposes of this experiment, we want to run mutated versions of our functions. We do this by altering the function definition and using this to define new functions. If we define that function with the mutate() decorator associated, we would then run this newly defined function and re-mutate it. Therefore, by default, decorators should be stripped from the mutated function. This is left as an option, however, in case further research wants to mutate undecorated functions, or leave some decorators attached to the function definition (perhaps selectively). The flag is provided for these future work scenarios.

As a result of the need to mutate a function definition, and because `inspect.getsource()` returns the function definition of the flow given to mutate, we can visit and process this function definition by defining a method, `self.visit_FunctionDef(self, node)`. Here, `node` is passed as an argument that represents the function definition in the Abstract Syntax Tree.

When visiting a function definition, we will be mutating the source according to some random values, to introduce variance. To make experiments repeatable, the random seed is set according to a value stored in the models.

Once the random seed is set, mutation can begin. The name of the node is altered so that the function definition will compile to a function with the original name appended with "_mod". This is the same name created in the decorator and appended to the end of the source being parsed by the `Mutator` class, so that the newly defined function will execute when this source is recompiled by the decorator.

According to the flag provided in the constructor, we might set the list of decorators on the function definition to an empty list; this removes all decorators from the function definition. If future researchers were to mutate functions and retain some decorators, this list would be probed and the required decorators can be kept, rather than simply stripping all decorators from the definition.

We alter the body of the function according to the mutation function provided by the modeller. If no mutation function is provided, the `Mutator` class would run the default identity function against the body, altering nothing. Otherwise, the body of the function is altered according to the function `Mutator` has been parameterised by. Worth noting is that, because the mutator runs this Function object indiscriminately, the safety of the mutation is left in the hands of the modeller.

After this has done, all necessary mutations have been made. The necessary counters are altered, including the random seed; this means that the next mutation's operation is also predictable yet different to the present one, so that the experiments are left predictable. Variance is preserved by the change in the seed, so that the next generated flow is different to the last, as an actor performing a workflow several times would perform that flow subtly differently each time.

To finish the mutation, all child nodes are visited. This technically should not be necessary, as no other function definitions should exist within a flow. Considering this, it was determined that traversing the rest of the tree might be useful if future work was to be done extending the library, as this function would behave in a predictable way, and would align more closely with the ordinary activity of a Python `NodeTransformer`.

5.4 Example use

An example of use of the fuzzing library might look like this:

Code sample 5.9: Example use of the mutation library

```
1  import random
2
3  # An example mutator function for an actor
4  # being inattentive, and so forgetting a step in a workflow.
5  def inattentive(lines):
6      if random.choice([True, False]):
7      lines.remove(random.choice(lines))
8      return lines
9
10 # A flow for implementing a feature according to an agile methodology
```

```
11 def implement_feature():
12     make_new_feature()
13     create_test_and_code()
14     unit_test()
15     integration_test()
16     user_acceptance_test()
17
18 # A mutated flow where the unit testing actor is inattentive
19 @mutate(inattentive)   # Mutation is specified here
20 def unit_test():
21     run_tests()
22     while not environment.resources["unit tests passing"]:
23         fix_recent_feature()
24         run_tests()
25
26 # Another flow
27 def create_test_and_code():
28     create_test_tdd()
29     add_chunk_tdd()
30
31 # An example atom for implementing a test
32 def create_test_tdd():
33     environment.resources["time"] += 1
34     test = dag.Test()
35     environment.resources["tests"].append(test)
```

Here, the mutation of the flow using the library can be seen in an example taken from a subset of a model used for testing the hypothesis. The flow randomly removes a line from the unit testing flow, but will only do so 50% of the time due to the `if` statement on line 6.

The library, while small, allows for great flexibility and power. It operates quickly as a result of using the Python built-in `ast` library. It also appears very cleanly in real-world models, taking one unobtrusive line to specify that a flow exists under stress, as a component of the flow's definition. Usage of the `mutate` decorator is therefore somewhat self-documenting too. A function under sociotechnical stress is decorated with the fact that it will be mutated, and the type of mutation it is, which if given a sensible name (as above) will describe by name exactly what the mutator function represents. This should also have meaning to the modeller, who wrote the function.

A large organisation wishing to create very large models with many types of mutation might create their own library of mutation functions. This way, creation of a varied model is as simple as importing the organisation's library and getting access to (presumably well-tested) sociotechnical variance functions for easy non-deterministic modelling.

Chapter 6

Evaluation

6.1 Building an example model

To confirm that the modelling system was adequate for the task, and to test the hypothesis laid out, the third required component of the project was constructed: an example model to be used to test the hypothesis.

Figure 6.1 shows how a workflow would begin to be developed using very simple Activity Diagrams as flows that are linking together. After all of the flows needed are broken down, any node on any flow that does not represent another flow (or the current flow itself, if the flow is recursive) would be an atom with activity that should be obvious by its name.

Two example models were built for the purposes of testing this hypothesis. They both modelled software development teams that adhered to some commonly used software development methodology. However, while one is a traditional Waterfall software development strategy as defined by the US Department of Defence in their original DOD-STD-2167A specification[33], the other is an Agile methodology, designed to fix many of the issues perceived to exist in the old Waterfall standard. Particularly, the Test Driven Development (TDD) methodology intends to fix the problem that exists when many bugs exist in a Waterfall system. The TDD method writes tests that nominally represent the functionality of a piece of code, and subsequently write that code until the code passes the test, as can be seen in the full Activity Diagram for the TDD methodology in Figure 6.2.

6.1.1 Building flows

Constructing the flows was as simple as following the understanding of the system from a high-level, not particularly detailed perspective, and drilling down on each individual node of the high-end view until more detail was extracted. In this way, the full TDD workflow that can be found in Figure 6.2 was built from the first flow which can be found at Figure 6.1. This allowed very fast prototyping of the sociotechnical system in a manner which could be ported from the Activity Diagram to Python code very quickly.

As was noted in §3.1.8, UML diagrams can be programmatically transformed into executable skeleton code with ease. In this way, converting the model to code that is ready for fuzzing would be even

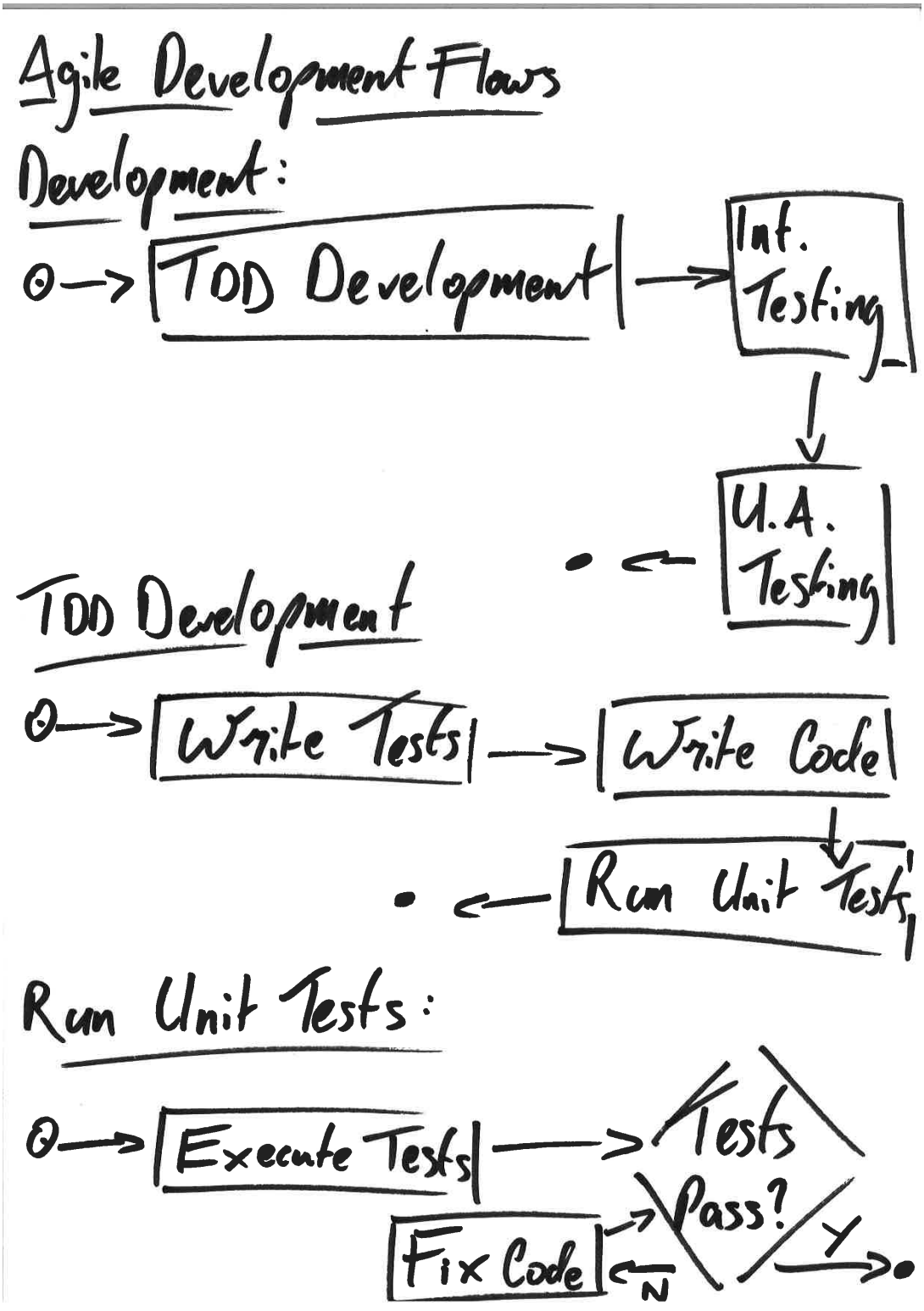

Figure 6.1: Some example Agile flows

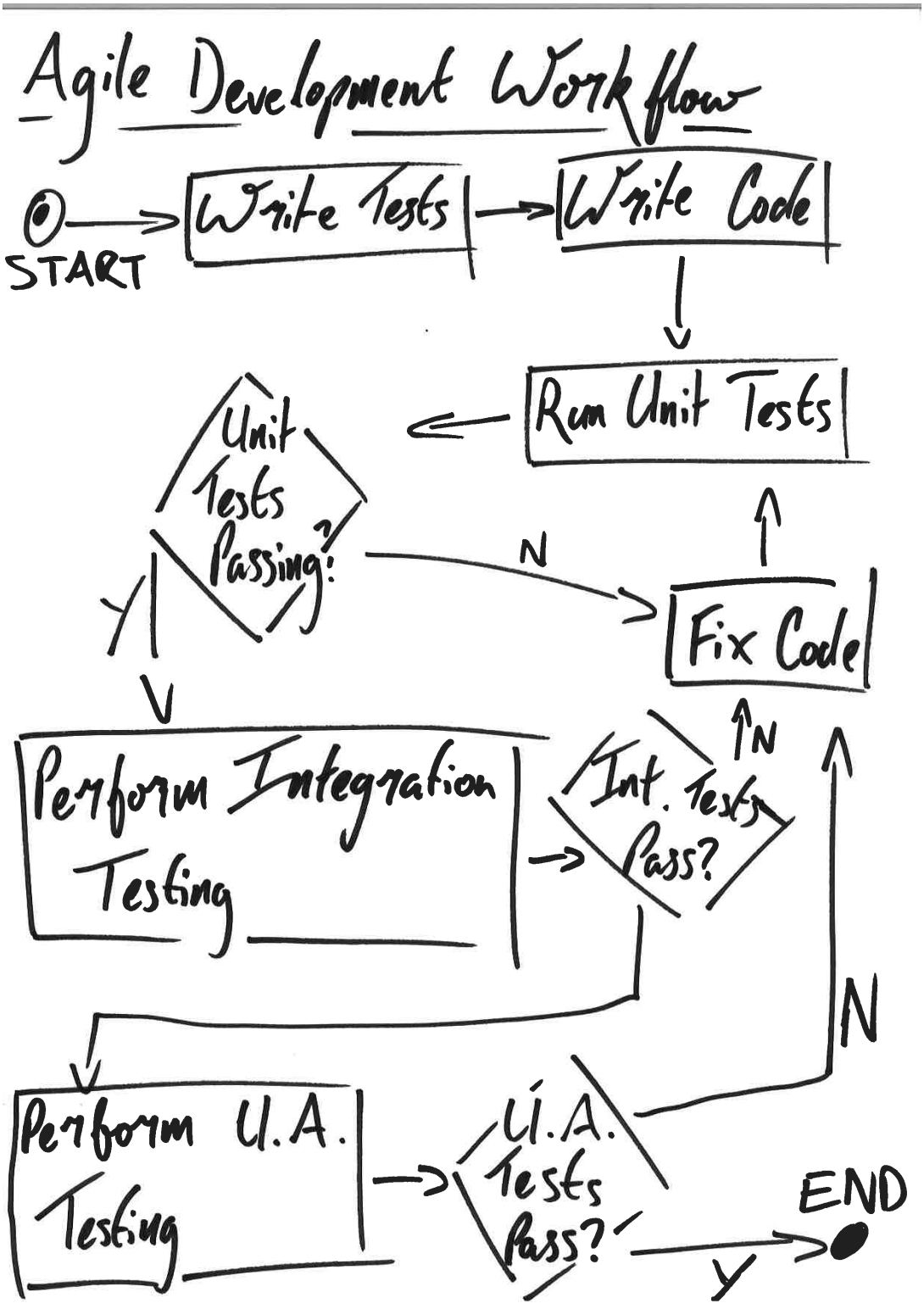

Figure 6.2: A full workflow for the development portion of a software product in Test-Driven Agile Development

easier than it already is in the future.

This procedure was carried out for both the agile and waterfall workflows. The agile flows used for the experiments can be found in §4.1.1, for example. Further code examples can be found in the Appendices (Appendix A, Appendix B).

6.1.2 Building atoms

Once flows were successfully built, all nodes on the Activity Diagrams that did not have a definition as another must be defined as atoms. Therefore, after the flows had been written as python programs, every function call that had no definition was indicated by the IDE, making the remaining work easy to spot. The errors in the flows were solved by creating the relevant atoms, together with the environment necessary to facilitate their effects, and the sociotechnical models were complete.

The sociotechnical environment was implemented as a Directed Acyclic Graph in the form of four different class definitions. Objects constructed from these classes were kept in a dictionary alongside some boolean flags and values which were used throughout the model, for modifying the program's behaviour and signalling things like tests passing to the rest of the model.

6.1.3 Mutating the models

Mutating the models was very easy. A fully mutated form of a code example from §4.1.1 was written like so:

```
from base import mutate
import random

#Removes an action from a behaviour if the actor is stressed (50% chance)
def stressed(lines):
    if random.choice([True, False]):
        lines.remove(random.choice(lines))
    return lines


def implement_feature():
    make_new_feature()
    create_test_and_code()
    unit_test()
    integration_test()
    user_acceptance_test()

@mutate(stressed)
def unit_test():
    run_tests()
    while not environment.resources["unit tests passing"]:
        fix_recent_feature()
        run_tests()
```

Mutating specific functions proved very easy with the system derived. The mutations could be directed to a specific workflow, was easy to write, and was also human-readable, particularly where it mattered in the flow definition.

Once concern was that, if a mutation function was convoluted or required complex logic to construct, it might be very easy to break the mutation library with a small bug. Another mutation function was then written to see whether anything complex would arise from deadlines being hard to meet, rather than actors feeling stressed:

```
from base import mutate
import random

def cannot_meet_deadline(lines):
    if random.choice([True, False]):
        lines = lines[:random.randint(1, len(lines)-1)]
    return lines

@mutate(cannot_meet_deadline)
def unit_test():
    run_tests()
    while not environment.resources["unit tests passing"]:
        fix_recent_feature()
        run_tests()
```

As can be seen, the additional mutation function was exactly as long and no more complex than the previous. Therefore, any simplification could satisfactorily be left for future work, as the mutation library implemented was adequate for the purposes of this project.

6.2 Running a model

Running a model was as simple as writing simple unit tests that call the highest-order function of the models created, because of the construction of the models according to the modelling system.

This is a result of the top-down approach to the sociotechnical model design. A side-effect of the full workflow being the first thing to be defined was that this first function can be executed to simulate the whole sociotechnical system. An additional flow was added for testing purposes which implemented 50 features according to each methodology rather than just one, as this way the effects of the variance in the system would be more pronounced and therefore easier to detect. Naturally, this was very easy to implement, as it was simply a repetition of the main workflow.

6.2.1 Carrying out experiments on the models

The flows were run within a series of unit tests that confirmed that the sociotechnical environment was being affected in some way that made sense for the models that were constructed, with a flag to trigger mutations turned off. To confirm that the outcomes were not the results of the random seed at the time, the random seed was set as a variable in the sociotechnical environment.

Similar unit tests were then run with the environment turned off and turned on separately, so as to compare the effect of mutating each model. A method was implemented in each model to reset its sociotechnical environment to some base state, which meant that a model could be run indefinitely many times in the same unit test.

Once the mutation library was verified to be working, a unit test was constructed that ran each model twice — once mutated, and once in its unmodified state — and calculated the difference in the time each model took to implement 50 features when put under stress in its unit testing phase. This would be done four times with four different (but predetermined) random seeds. The difference was then compared between the two executions of the models.

If the delta in time taken was less for the agile model than it was for the waterfall model, it would be confirmed that influencing the models using code fuzzing was having the intended effect, as an agile flow should operate better than a waterfall method when put under sociotechnical stress. The unit test was thus written so that it would pass if this was true, indicating that the experiment's hypothesis was validated.

When the test was run, it passed, validating the hypothesis. Some graphs of the results were then produced.

6.2.2 Results of these experiments

The data produced by the unit test was plotted using the `pyplot` library[34]. As in the unit tests, seeds were varied and an average of the time taken by each team modelled was compared. The data produced by the models can be found in the graphs in Figure 6.3 and Figure 6.4.

As can be seen, the varied agile flow outperforms the varied waterfall flow in almost every scenario. This would indicate that the hypothesis was accurate, as one would expect Agile's iterative nature to catch bugs early and fix them when fixing a bug in Agile did not cost much time.

The similarities between the two graphs indicate that more complex variance might produce more pronounced or interesting results; however, this more complex variance would require a more complex and/or detailed model, and insufficient time was available to implement this.

6.3 Evaluating the tools used

The tools performed well in reducing complexity and applying variance to the sociotechnical system. They also made the creation of the sociotechnical system straightforward, as the activity diagram is a very natural way to represent workflow.

The representation of workflow also felt quite natural as a format for constructing a sociotechnical system. This was likely a side effect of the fuzzing library's focus on the mutation of sociotechnical workflow, as the project was being approached from the perspective that workflow was the most important component of the sociotechnical model.

Other models, such as responsibility modelling, would not fare as well being represented as a UML Activity Diagram. However, as a result of code fuzzing working to inject sociotechnical variance into

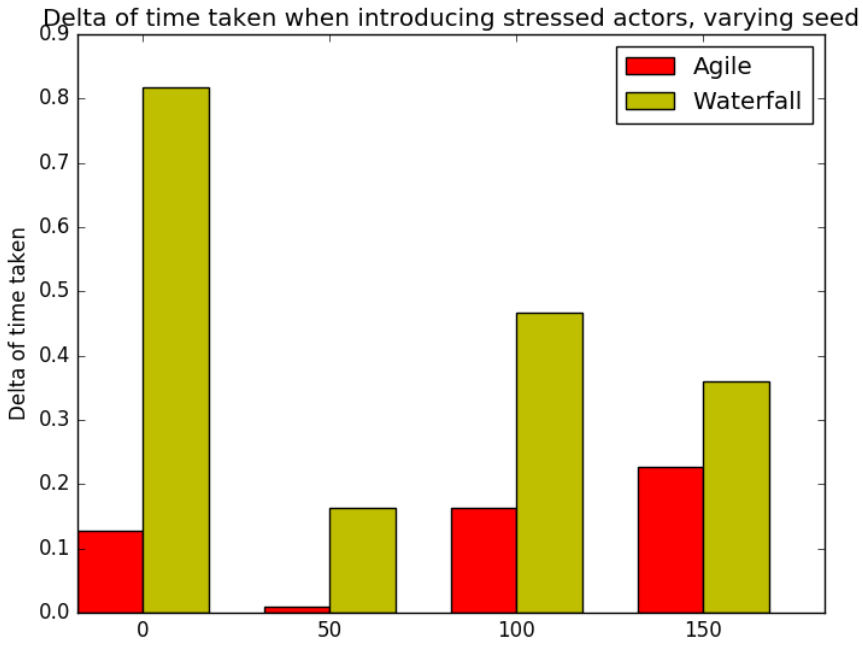

Figure 6.3: A graph of the degree of change in a model undergoing sociotechnical stress created by stressed actors, presented as a decimal proportion of an unvaried flow.

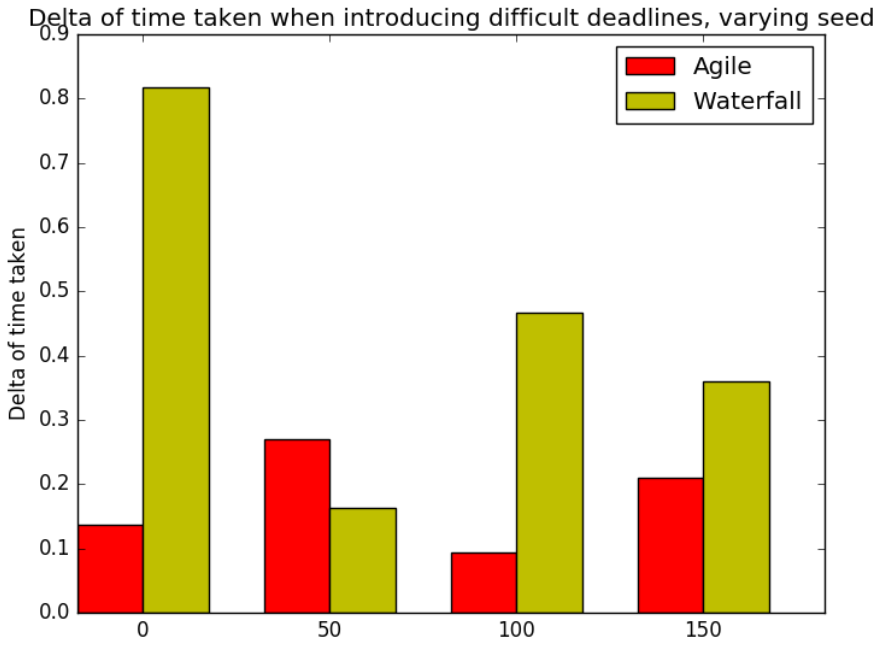

Figure 6.4: A graph of the degree of change in a model undergoing sociotechnical stress created by hard to meet deadlines, presented as a decimal proportion of an unvaried flow.

a workflow model, it might be that a similar tactic would work for injecting variance into alternative models such as responsibility modelling. Another angle that may allow the code fuzzing technique to apply to modelling systems like Responsibility Modelling would be to generalise the procedural modelling system to represent a wider range of model focuses. However, this angle may be a more difficult approach than writing another fuzzing system for that type of model specifically.

Building the models did not take much time. After the fuzzing system was created the original model created needed rewriting for additional detail; this was little effort, as a result of the separation of concerns between atoms and flows, as it meant that details of the modelling problem could be separated and solved individually and recombined later for a better overall model.

On a similar note, a benefit of this sociotechnical modelling tactic is that the atoms and flows can be removed from each other. Therefore, a large library of well tested atoms could be kept in an organisation that built sociotechnical models, and these atoms could be reused for any model which used the same atoms. This is another corollary of the atoms-as-physics idea discussed earlier, as the activities an actor might perform cannot change from model to model, because the actions are the same. Therefore, with a reconfigured environment and with different flows connecting the atoms together, a sociotechnical system could be built even easier by simply creating UML Activity Diagrams and representing them as flows that connect pre-written models. From the experience of writing and rewriting the models from the experiments, it could be confirmed that the creation of the models' atoms was the most time-consuming part of the exercise, even considering that the models shared many atoms.

One issue with the modelling system as it stands was that only one actor could be modelled from the UML diagram. A more robust methodology for moving a UML Activity Diagram to a procedural workflow, such as taking into account Fork and Join operations, would help with this. This is discussed in the chapter on future work (§7.3).

Chapter 7

Future Work

7.1 A mathematical basis

Some sociotechnical mathematics already exists. For example, π-calculus has been suggested as a tool for modelling workflows[35] (though it should be noted that there is some resistance to this idea[19]). The potential for π-calculus to be suitable as a workflow modelling system might encourage one to do modify the system such that, rather than mutating Python, π-calculus was mutated instead.

This approach appeals for a few reasons. For example, we might want to then perform model checking on the sociotechnical model, which would be possible with currently existing tools[36], adding another reason why researchers and sociotechnical modellers might want to use and improve this tool.

However, π-calculus can be complex, and the tool and methods are not yet fully complete, as will be seen. Moreover, the modelling system itself has more pressing concerns than a need for a mathematically focused engine, but this work might be valuable if the tools were to be maintained long-term.

Another direction to take sociotechnical mathematics might be responsibility modelling. An adaptation from the current Activity Diagram/workflow-as-blueprint model might be to use fuzzing on responsibility or obligation logics, such as deontic logic[37]. Responsibility modelling is a growing field of research, and here variance might be found as misunderstandings or uncertainties when responsibilities are communicated between actors. This method could therefore be used to model the communication of responsibilities within responsibility modelling, another mathematical basis for code fuzzing as sociotechnical variance.

7.2 Middleware layers for submodel interaction

The interaction of models currently mostly relies on every detail of the model being worked into the system we're testing, because importing one set of activities into another doesn't account for the interplay between the two systems.

One must account for this interplay between systems, because missing even small details from a model leaves that model functionally incomplete: emergent phenomena from that small detail would be

lost, and swathes of activity that should be modelled in a perfect case is lost in the incomplete one.

To combat this, one might create some middleware that allows two subsystems to interact by registering and referencing each other. In this way, activity from one system can influence another system, even though the two systems do not directly interact.

This middleware layer might also house the environment of the system, so that the environment is kept globally accessible to all procedures, and all subsystems that operate through the middleware interact with the same environment. This would keep the concerns of the modules bound together in one place, with all interactions with the modules of the larger system being together, whether those interactions took place directly between activities in the models or indirectly through changing their shared environment.

7.3 Concurrent modelling

A sociotechnical system can have many different types of stresses when modelled in a concurrent fashion. This is because multiple human actors greatly affect the complexity of a sociotechnical system increases as the complexity of the interactions on the part of human actors increases. This is necessarily true, because a sociotechnical system is a system that includes complex interactions between humans, machines, and the environment around them[4] — thus, more human interaction adds complexity to the model, as communication between actors might affect workflow.

A concurrent modelling system was mentioned when laying out the requirements of a sociotechnical system modelling platform(§3.1). While this was not implemented, the choice of Python as a language to create the models should facilitate this, as Python has support for concurrent processing[38]. Additionally, UML Activity Diagrams have support for Fork and Join methods[24], allowing for concurrent modelling in the planning phase of model creation.

The combination of these two factors should make concurrent modelling in this system possible and not too difficult to implement; work must be done to solve problems such as race conditions with the sociotechnical environment and multiple actors, or how resources in that environment are expended when multiple actions can be performed by an actor at once. Once solutions to these problems are solved, modellers should be able to employ Python and UML's respective concurrency systems to create concurrent models using this format.

7.3.1 Human interaction modelling

An import and difficult to model component of the sociotechnical system is direct social interaction. The workflow of a system is largely human interaction with technology; technological interactions and effects are largely modelled in the atom layer of our model; social interactions must be modelled through some alternative system, where knowledge can be exchanged between actors, or where actors can influence each other's environment, if knowledge is really a component of an actor's sociotechnical environment.

With the current modelling system, no solution to this problem immediately springs to mind. Therefore, development of a solution to this issue — and therefore development of a more complete sociotechnical system — are left to future research and development upon this proof of concept.

7.4 More case studies

Additional models, such as models of a barista in a café or a barber in a barbershop would be interesting to develop. These systems are subtle and difficult to model. Moreover, modelling these systems requires gaining an insight into the subtleties of these systems with inherant variance. They can be complicated by equipment not functioning properly in either case, customers being unhappy or interacting in unexpected ways with baristas and barbers in either environment, or actors in the sociotechnical model interacting in unreliable ways with both the technology they are in charge of and each other.

Therefore, cafés and barbershops make for interesting case studies, as they are extreme sociotechnical systems that are also mundane: as we interact with them daily, we have a basic understanding of their complexities without really knowing their subtleties. Moreover, both cases have a degree of subtlety but aren't seriously complex sociotechnical systems compared to other potential case studies, such as a university's payroll system or the UK's National Health Service. They also may contain unreliabilities and complexities that lead to more detailed and interesting mutations, particularly when both are made concurrent.

7.5 Fuzzing more models

A component of the functionality of this method is that it relies on workflow modelling. This is necessary, because we introduce sociotechnical stress as alterations to the workflow — the workflow therefore needs to be modelled. However, one method to solve the social interaction modelling would be to mutate actor or intent modelling platforms, often used for requirements engineering.

Other interesting models to mutate would be goal-oriented models such as i*[39], where alterations to the model would be changes to goals actors might work towards. This could be a useful way to demonstrate differences in communication or misunderstandings between actors, effectively goal-oriented sociotechnical stress.

Furthermore, we might even want to mutate dataflow based models like OBASHI[20], where mutations might occur as alterations to the flow of information. Impact analysis[40] could therefore be modelled by observing the effect that change of Business & IT system state has on the running of that system. Using techniques to effect randomised states, such as fuzzing, is functionality which is currently missing from tools such as OBASHI. However, running simulations with fuzzed models might produce very different outlooks on the critical parts of Business & IT systems.

7.6 More sophisticated fuzzing types

Creating these case study models would require creating more sophisticated fuzzing techniques. While the fuzzing library has been tested and has been shown to successfully introduce sociotechnical variance to a system, the fuzzing techniques used to test the systems have been relatively simple. It would therefore be prudent to test the system with more complicated fuzzing techniques, to see whether the system holds up against this.

One reason for this is that a more complex mutation function might introduce bugs in the system that are hard to detect, because the affect the mutation function would have might be harder for the modeller to predict. They might also get quite hard to write, as a result of being a component of the model that feels abstract to write. Having more complex mutation functions would therefore indicate more whether this system is a feasible modelling system in its own right, and whether code fuzzing can be successfully introduced to a system using the models already created.

Another reason is that more complex fuzzing techniques may well test some of the future work addressed above: variance through social interaction, alternative case studies where more complex fuzzing is appropriate, or putting a mathematical modelling system through some more intense tests. It should also show whether more complex fuzzing systems can stay reasonably safe in terms of the mutated code and Turing's Halting Problem[11], or whether some more safe sandbox for this mutated code should be constructed to aid the modeller.

7.7 Multiple fuzzing options in one area of workflow

Currently, the system allows one to apply some sociotechnical stress to a part of a workflow, by using a decorator which accepts a mutation function defined by the modeller. While this system adequately reflects sociotechnical stress, in the real world stresses and unreliabilities come from myriad stresses in a variety of ways. Real-world stress isn't just one stress, so much as different stresses compounding.

An extension to the fuzzing library, therefore, would be to create some way of combining stresses on a workflow for larger, more realistic systems than the one tested. These systems would need the added complexity of different stresses combined to accurately reflect real-world variance on their workflows.

This addition to the current library might be easier if a more mathematical back-end were implemented, as composing mathematical functions might prove much easier than somehow combining procedures. Some other concerns also need to be addressed before this can be implemented, however. One concern that needs to be addressed is that different stresses might act in different amounts on a system. It might be that a workflow is altered by hard-to-meet deadlines only a little, while variance is often introduced by bad training of new staff. Accurately simulating this variance might mean defining a procedure for deadline stress and another for bad training, then a separate method for defining the degree to which each stress affects the system. Retaining the simplicity of the approach while introducing this additional complexity to the fuzzing layer might be difficult to do, and much thought should be given here to the implementation in terms of both accuracy of simulation and ease of modelling.

7.8 Markov Chains from simulations

Using the tools in their current state, we can now step through a simulation which has sociotechnical variance inserted, as per the requirements of the project. However, every simulation executed takes time, and building a full view of what that sociotechnical stress looks like after many iterations of a project could take some time for a large model with many mutated areas.

A better system might be to model the variance probabilistically, and alter a mathematical model by composing the function of the workflow being mutated with the mutation function in a mathematical way. This would allow us to build variance into the model programmatically using the current notation,

with the mathematical foundations described earlier. However, that mathematical model could be hard to make and will likely require lots of research and further development to create.

A more immediate solution might be to create a Markov Chain of the varied simulations by running them many times and creating a graph from the simulated workflows. This would give us a graphical representation of the workflow.

Markov Chains can be analysed using graph theory. They are very well understood, being necessary in practises such as weather prediction[41], and because they are representable as a directed graph with weighted edges, they are also appropriate as a representation of a flowchart. Therefore, a modeller might want to take their procedural model of a workflow and transform it back into a graph; this is easily done using a Markov Chain.

A Markov Chain might be generated from a simulation by using decorators for atoms that catch function calls to atoms, and run the atom as they would normally while accounting for preconditions and aftereffects of atoms running if needed. After running the atom, the atom's execution is logged in some chronologically ordered list. After the simulation has finished, a chronological log of every action taken in the simulation is produced. If features of the sociotechnical state are also logged at the time of the atom execution, then we can say, given sociotechnical state, what the next action of an actor in a model is likely to be. We can use this to build our Markov Chain, by weighting each edge between nodes according to the probability that, if the source node of the edge is the action just completed, then the target node of the edge is a possible next action given the actions taken in the simulation, with a weight according to the probability of traversing that edge with the current sociotechnical environment's state.

If many simulations were done, generating a large corpus of data collected with sociotechnical stress present, then a reasonably accurate Markov Chain could be produced which could then be analysed in a mathematical way. Traversing this Markov Chain would not take much less time than a regular workflow on account of the fact that most of the time spent by the computer in running the models is in alterations to the sociotechnical environment, but a mutated model might be different. This is because no mutations need to be created, as the variance is built into the Markov Chain, so for models that heavily employ stress modelling or employ relatively complex mutation functions, no additional work is needed to alter the workflow.

Chapter 8

Conclusion

8.1 Discussion

The modelling system works as intended, and allows the fuzzing library to show that sociotechnical stress can indeed be simulated using code fuzzing. The library worked particularly well in that configuring and using it is very easy, and the mutation creation itself is simple and fast.

Another pleasant note on the fuzzing library is that creating a code fuzzing representation of any sociotechnical stress is easy to do. A single function that receives and returns a list of line objects is all that is required for the fuzzing library to operate successfully. In this way, infinitely many versions of sociotechnical variance can be made and implemented. This makes the library very flexible, and allows future work to be implemented much easier as a result of not relying on built-in options that need to be modified for a researcher's purposes.

Realistically, any system which has a need for sociotechnical variance and models using workflows should be able to use a variant of this tool, rewriting it to traverse ASTs of the desired language. Any non-AST version of the code might not be safe, as ASTs are the commonly used format for code fuzzing and mutation testing in the wild, as seen in MutPy[26] and PyMuTester[29].

8.2 Conclusion

The result of testing the hypothesis was that the experiment was a success. Sociotechnical stress can feasibly be modelled using code fuzzing.

The needs for a sociotechnical system which can have stress inserted in a clean, readable way was met by the combination of a modelling system made for workflow modelling, and a fuzzing system that was very flexible.

The tools made seem genuinely useful, particularly the fuzzing library. The fuzzing library's flexibility should make it useful for future work on this and related projects. Making a full mutation testing system for Python code using the technique would also be interesting to do, as no mutation testing library could be found that mutated code upon each function call in the way that this library does.

Fortunately, the models used to test the system showed that variance was introduced in a way we might expect from the sociotechnical variance we introduced. This conclusively prove the hypothesis.

However, just because the technique *can* be feasibly used doesn't mean that it can be used in its *current state*. More research is needed now to produce a fully fledged modelling system from this technique, as discussed in the previous chapter (§7).

8.3 Personal Reflection

It is interesting to me that sociotechnical modelling can learn so much from software engineering. Truth be told, the two fields aren't terribly far apart — closer than, say, sociotechnical modelling and knitting — but this still goes to show that differing fields can learn from each other in surprising ways.

The success of the technique is satisfying, because it appears to be very useful. Variance as its own concept seems to be something that many different modelling frameworks in many different fields need for a range of purposes. Variance has a different purpose depending on the focus of the model. Ultimately, it appears to be a very useful tool for introducing uncertainty and non-determinism into an otherwise static, deterministic model. The fact that Python allows this to be done in a clean, readable way is pleasing.

I am very happy with the result of the project.

Bibliography

[1] Andy Crabtree et al. “Ethnomethodologically informed ethnography and information system design”. In: *Journal of the American Society for Information Science and Technology* 51.7 (2000), pp. 666–682. ISSN: 15322882. DOI: 10.1002/(SICI)1097-4571(2000)51:7<666::AID-ASI8>3.0.CO;2-5.

[2] Sean Brennan. “The biggest computer programme in the world ever! How’s it going?” In: *Journal of Information Technology* 22.3 (2007), pp. 202–211. ISSN: 02683962. DOI: 10.1057/palgrave.jit.2000104.

[3] R. Lock, T. Storer, and I. Sommerville. “Responsibility modelling for risk analysis”. In: September 2009 (2009). URL: http://eprints.gla.ac.uk/71594/.

[4] Gordon Baxter and Ian Sommerville. “Socio-technical systems: From design methods to systems engineering”. In: *Interacting with Computers* 23.1 (2011), pp. 4–17. ISSN: 09535438. DOI: 10.1016/j.intcom.2010.07.003. URL: http://dx.doi.org/10.1016/j.intcom.2010.07.003.

[5] Barton P. Miller. “CS 736 Fall 1988 Project List”. In: *University of Wisconsin* (1988). URL: http://research.cs.wisc.edu/adsl/Publications/starbox-hotstorage13.pdf.

[6] Nicos Christofides. *Graph Theory: An Algorithmic Approach (Computer Science and Applied Mathematics)*. Orlando, FL, USA: Academic Press, Inc., 1975. ISBN: 0121743500.

[7] W M P van der Aalst and A H M ter Hofstede. “YAWL: Yet Another Workflow Language”. In: *QUT Technical report* (2002).

[8] Z Standards Panel and Consensus Working Draft. “Formal Specification - Z Notation - Syntax , Type and Semantics”. In: 19 (2000), p. 183. URL: http://www.open-std.org/jtc1/sc22/open/n3187.pdf.

[9] *BPMS Specification - Business Process Model and Notation* . URL: \url{http://www.bpmn.org/},urldate={\date},author={ObjectManagementGroup},month=March,year={2016}.

[10] Thomas Herrmann and Kai-Uwe Loser. “Vagueness in models of socio-technical systems”. In: *Behaviour & Information Technology* 18.5 (1999), pp. 313–323. ISSN: 0144-929X. DOI: 10.1080/014492999118904.

[11] Alan Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem - A Correction”. In: *Proceedings of the London Mathematical Society* (1937). ISSN: 00664138. DOI: 10.1016/B978-0-12-386980-7.50002-2. arXiv: arXiv:1011.1669v3.

[12] *Enterprise Overhaul: Resolving DNS — PayPal Engineering Blog*. https://www.paypal-engineering.com/2015/12/16/enterprise-overhaul-resolving-dns/. (Accessed on 03/23/2016).

[13] David Azzopardi Leif Maxwell. “How To Tango With Django 1.7 — How to Tango with Django 1.7”. In: (2016). URL: http://www.tangowithdjango.com/book17/index.html.

[14] Vera Maria Benjamim Werneck, Antonio de Padua Albuquerque Oliveira, and Julio Cesar Sampaio do Prado Leite. “Comparing GORE Frameworks: i-star and KAOS”. In: *Wer* (2009), pp. 1–12. URL: http://wer-papers.googlecode.com/svn-history/r71/trunk/dataset/wer09/WER09{_}4.pdf.

[15] Faisal Almisned and Jeroen Keppens. “Requirements Analysis: Evaluating KAOS Models”. In: *Journal of Software Engineering and Applications* 03.09 (2010), pp. 869–874. ISSN: 1945-3116. DOI: 10.4236/jsea.2010.39101.

[16] Jurriaan Van Diggelen et al. “Implementing collective obligations in human-agent teams using KAoS policies”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 6069 LNAI. M4D. 2010, pp. 36–52. ISBN: 3642149618. DOI: 10.1007/978-3-642-14962-7{_}3.

[17] Kaos Tutorial. “A KAOS Tutorial”. In: *International journal of infectious diseases IJID official publication of the International Society for Infectious Diseases* 12.5 (2007), pp. 1–46. ISSN: 12019712. DOI: 10.1016/j.ijid.2008.03.002. URL: http://www.objectiver.com/fileadmin/download/documents/KaosTutorial.pdf.

[18] Arthur H M Ter Hofstede et al. *Modern business process automation: YAWL and its support environment*. Vol. 7. 11. 2010, pp. 1–676. ISBN: 9783642031205. DOI: 10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3.

[19] Wil M P Van Der Aalst. “Why workflow is NOT just a Pi-process”. In: *BPTrends* (2004), pp. 1–2.

[20] Fergus Cloughley Paul Wallis. *The OBASHI Methodology*. 1st ed. Vol. 1. The offical manual for the OBASHI Methodology. The Stationery Office, 2010. ISBN: 9780117068575.

[21] Kalle Lyytinen and Mike Newman. “Explaining information systems change: a punctuated socio-technical change model”. en. In: *European Journal of Information Systems* 17.6 (2008), pp. 589–613. ISSN: 0960-085X. DOI: 10.1057/ejis.2008.50. URL: http://www.palgrave-journals.com/ejis/journal/v17/n6/abs/ejis200850a.html.

[22] Gudela Grote. “Uncertainty management at the core of system design”. In: *Annual Reviews in Control* 28.2 (2004), pp. 267–274. ISSN: 13675788. DOI: 10.1016/j.arcontrol.2004.03.001.

[23] Gerald I. Susman. *Autonomy at Work: A Sociotechnical Analysis of Participative Management*. Praeger, 1976, p. 230. ISBN: 0275561402. URL: https://books.google.co.uk/books/about/Autonomy{_}at{_}Work.html?id=y{_}qZAAAAIAAJ{\&}pgis=1.

[24] Omg. *OMG Unified Modeling Language TM (OMG UML), Superstructure v.2.3*. Vol. 21. May. 2010, p. 758. ISBN: formal/2010-05-03. DOI: 10.1007/s002870050092. URL: http://www.omg.org/spec/UML/2.3/.

[25] Marcelo Paternostro Ed Merks Dave Steinberg Frank Budinsky. *EMF: Eclipse Modeling Framework*. 2nd ed. Vol. 1. Addison-Wesley Professional, 2009. ISBN: 978-0-321-33188-5.

[26] *MutPy 0.4.0 : Python Package Index*. https://pypi.python.org/pypi/MutPy. (Accessed on 03/25/2016).

[27] *khalas / mutpy / source / — Bitbucket*. https://bitbucket.org/khalas/mutpy/src. (Accessed on 03/25/2016).

[28] *PIT Mutation Testing*. http://pitest.org/. (Accessed on 03/25/2016).

[29] *GitHub - miketeo/PyMuTester*. https://github.com/miketeo/PyMuTester. (Accessed on 03/23/2016).

[30] Martin Gogolla and Cris Kobryn. *UML 2001 – The Unified Modelling Language*. 2001, pp. 1–523. URL: http://download.springer.com/static/pdf/748/bok{\%}253A978-3-540-45441-0.pdf?originUrl=http{\%}3A{\%}2F{\%}2Flink.springer.com{\%}2Fbook{\%}2F10.1007{\%}2F3-540-45441-1{\&}token2=exp=1458602790{~}acl={\%}2Fstatic{\%}2Fpdf{\%}2F748{\%}2Fbok{\%}25253A978-3-540-45441-0.pdf{\%}3ForiginUrl{\%}3Dhttp{\%}253A{\%}252F{\%}252Flink.springer.com{\%}252Fbook{\%}252F10.1007{\%}252F3-540-45441-1*{~}hmac=3a800ca2b0548104e3bf3bb9fdb3e479b1e700865f2e3b7ed8e4a7a9504

[31] *32.2. ast — Abstract Syntax Trees — Python 2.7.11 documentation*. https://docs.python.org/2/library/ast.html. (Accessed on 03/23/2016).

[32] *32.2. ast — Abstract Syntax Trees — Python 2.7.11 documentation*. https://docs.python.org/2/library/ast.html. (Accessed on 03/27/2016).

[33] UNITED STATES OF AMERICA DEPARTAMENT OF DEFENSE. "Defense System Software Development Dod-Std-2167". In: (1984), p. 11.

[34] MatPlotLib development team. *Matplotlib 1.5.1 documentation*. http://matplotlib.org/contents.html. (Accessed on 03/25/2016).

[35] Howard Smith and Peter Fingar. "Workflow is just a Pi process". In: *BPTrends* (2003), p. 28.

[36] A. Tiu. "Model checking for pi-calculus using proof search". In: *Lecture notes in computer science* 3653 (2005), p. 36. URL: http://www.springerlink.com/index/98291nbtf19ytxdp.pdf.

[37] David Hutchison and John C Mitchell. *Software-Intensive Systems and New Computing Paradigms. Challenges and Visions*. Vol. 5380. 2008, pp. X, 266. ISBN: 978-3-540-89436-0, 978-3-540-89437-7. DOI: 10.1007/978-3-540-89437-7. URL: http://link.springer.com/10.1007/978-3-540-89437-7$\backslash$nhttp://www.springer.com/978-3-540-89436-0.

[38] *16.2. threading — Higher-level threading interface — Python 2.7.11 documentation*. https://docs.python.org/2.7/library/threading.html?highlight=threading#module-threading. (Accessed on 03/25/2016).

[39] Eric S. Yu. "Social modelling and i*". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5600 LNCS.c (2009), pp. 99–121. ISSN: 03029743. DOI: 10.1007/978-3-642-02463-4{_}7.

[40] Shawn A. Bohner. "Software Change Impact Analysis". In: (1996).

[41] *49708837511_ftp.pdf;jsessionid=15C412359F0A77F9FD92DD24B703095F.f01t02*. http://onlinelibrary.wiley.com/store/10.1002/qj.49708837511/asset/49708837511_ftp.pdf;jsessionid=15C412359F0A77F9FD92DD24B703095F.f01t02?v=1&t=im9clifp&s=6bc6cccb372d6b20f66da1bda0c67b9535efb2d4. (Accessed on 03/25/2016).

Appendices

Appendix A

Agile Flows

Following are the flows used to test the Agile methodology:

Code Listing A.1: Flows used to test the Agile methodology

```
@flow
def make_new_feature():
    create_feature()


def create_test_and_code():
    create_test_tdd()
    add_chunk_tdd()


@flow
def unit_test():
    run_tests()
    while not environment.resources["unit tests passing"]:
        fix_recent_feature()
        run_tests()


@flow
def fix_recent_feature():
    for feature in environment.resources['features']:
        for chunk in feature:
            fix_chunk(chunk)


@flow
def integration_test():
    perform_integration_tests()
    while not environment.resources["integration tests passing"]:
        unit_test()
```

```
        perform_integration_tests()


@flow
def user_acceptance_test():
    perform_user_acceptance_testing()
    while not environment.resources["user acceptance tests passing"]:
        unit_test()
        integration_test()
        perform_user_acceptance_testing()


@flow
@mutate(stressed)  # This gets changed to cannot_meet_deadlines for additional t
def implement_feature():
    create_test_and_code()
    unit_test()
    integration_test()
    user_acceptance_test()


@flow
def implement_50_features():
    for i in range(50):
        make_new_feature()
        implement_feature()
```

Appendix B

Waterfall Flows

Following are the flows used to test the Waterfall methodology:

Code Listing B.1: Flows used to test the Waterfall methodology

```
@flow
def write_code():
    add_chunk_waterfall()


@flow
def unit_test():
    for chunk in environment.resources["features"]:
        if chunk.test is None: create_test_for_chunk(chunk)
    run_tests()
    while not environment.resources["unit tests passing"]:
        fix_codebase()
        run_tests()


@flow
def fix_codebase():
    for test in environment.resources["tests"]:
        if not test_passes(test):
            fix_chunk(test.chunk)


@flow
def integration_test():
    perform_integration_tests()
    while not environment.resources["integration tests passing"]:
        fix_codebase()
        perform_integration_tests()


```

```
@flow
def user_acceptance_test():
    perform_integration_tests()
    while not environment.resources["integration tests passing"]:
        fix_codebase()
        perform_integration_tests()
        integration_test()


@flow
@mutate(stressed) # This gets changed to cannot_meet_deadlines for additional te
def create_product():
    for i in range(environment.resources["size of product in features"]):
        write_code()
    for i in range(environment.resources["size of product in features"]):
        unit_test()
    for i in range(environment.resources["size of product in features"]):
        integration_test()
    for i in range(environment.resources["size of product in features"]):
        user_acceptance_test()


@flow
def implement_50_features():
    environment.resources["size of product in features"] = 50
    create_product()
```

Appendix C

Mutator Functions

Following are the mutator functions used in this report's experiments.

Code Listing C.1: Mutator functions used in experiments

```
def random_boolean():
    return random.choice([True, False])


def stressed(lines):
    if random_boolean():
        lines.remove(random.choice(lines))
    return lines


def cannot_meet_deadline(lines):
    if random_boolean():
        lines = lines[:random.randint(1, len(lines)-1)]
    return lines
```

Appendix D

Software Engineering atoms

Following and the sociotechnical atoms used to power the previous flows.

Code Listing D.1: Atoms used in experiments

```
@atom
def create_feature():
    environment.resources["features"].append([])
    environment.resources["current feature"] = len(environment.resources["featur


@atom
def add_chunk(testing=False):
    environment.resources["time"] += 2
    feature = environment.resources["current feature"]
    chunk = dag.Chunk()

    # Add a test if it's necessary
    if testing:
        test = environment.resources["tests"][-1]
        chunk.test = test
        test.chunk = chunk
        environment.resources["tests"].append(test)

    # Record this chunk of code
    environment.resources["features"][feature].append(chunk)

    # Conditionally add a bug to the chunk
    if random.randint(3,5) is 4:
        bug = dag.Bug(chunk)
        environment.resources["bugs"].append(bug)

    # We're now working on the chunk we just created, so...
    environment.resources["current chunk"] = chunk

```

```
31
32  @atom
33  def add_chunk_waterfall(testing=False):
34      environment.resources["time"] += 2
35      chunk = dag.Chunk()
36
37      # Add a test if it's necessary
38      if testing:
39          if len(environment.resources["tests"]) is not 0:
40              test = environment.resources["tests"][-1]
41              if test is not None:
42                  chunk.test = test
43                  test.chunk = chunk
44                  environment.resources["tests"].append(test)
45
46      # Record this chunk of code
47      environment.resources["features"].append(chunk)
48
49      # Conditionally add a bug to the chunk
50      if random.randint(3,5) is 4:
51          bug = dag.Bug(chunk)
52          environment.resources["bugs"].append(bug)
53
54      # We're now working on the chunk we just created, so...
55      environment.resources["current chunk"] = chunk
56
57
58  @atom
59  def add_chunk_tdd(testing=True):
60      environment.resources["time"] += 2
61      feature = environment.resources["current feature"]
62      chunk = dag.Chunk()
63
64      # Add a test if it's necessary
65      if testing:
66          if len(environment.resources["tests"]) is not 0:
67              test = environment.resources["tests"][-1]
68              if test is not None:
69                  chunk.test = test
70                  test.chunk = chunk
71                  environment.resources["tests"].append(test)
72
73      # Record this chunk of code
74      environment.resources["features"][feature].append(chunk)
75
76      # Conditionally add a bug to the chunk
77      if random.randint(3,5) is 4:
78          bug = dag.Bug(chunk)
79          environment.resources["bugs"].append(bug)
```

```
80
81      # We're now working on the chunk we just created, so...
82      environment.resources["current chunk"] = chunk
83
84
85  @atom
86  def create_test_for_chunk(chunk):
87      environment.resources["time"] += 1
88      if chunk.test is None:
89          test = dag.Test(chunk)
90          environment.resources["tests"].append(test)
91          chunk.test = test
92
93  def create_test_tdd():
94      environment.resources["time"] += 1
95      test = dag.Test()
96      environment.resources["tests"].append(test)
97
98
99  @atom
100 def fix_chunk(chunk=None):
101     if chunk is None: chunk = environment.resources["current chunk"]
102
103     # Iterate over bugs that affect the chunk and thrash until fixed
104     for bug in environment.resources["bugs"]:
105         while detects_bug(chunk.test, bug):
106             environment.resources["time"] += cost_of_bug(bug)
107             if random.randint(0, 5) is 4:
108                 remove_bug(bug)
109                 break
110
111
112 @atom
113 def perform_integration_tests():
114     if len(environment.resources["bugs"]) == 0:
115         environment.resources["integration tests passing"] = True
116         return True
117     number_of_messy_bugs = random.randint(0,len(environment.resources["bugs"]))
118     environment.resources["integration tests passing"] = number_of_messy_bugs ==
119
120     return environment.resources["integration tests passing"]
121
122
123 @atom
124 def run_tests():
125     environment.resources["unit tests passing"] = number_of_detected_bugs() == 0
126     return environment.resources["unit tests passing"]
127
128
```

```
@atom
def perform_user_acceptance_testing():
    environment.resources["time"] += 1
    environment.resources["user acceptance tests passing"] = environment.resourc
    return environment.resources["user acceptance tests passing"]
```

Appendix E

Environment Layer

Code Listing E.1: Environment layer contents

```
resources = {}
resources["time"] = 0
resources["seed"] = 0
resources["integration tests passing"] = False
resources["user acceptance tests passing"] = False
resources["unit tests passing"] = False
resources["tests"] = []
resources["successful deployment"] = False
resources["features"] = []  # To be a list of Chunk objects
resources["bugs"] = []  # To be a list of Bug objects
resources["mutating"] = True
resources["current chunk"] = None
resources["most recent test"] = None
resources["current bug"] = None
resources["current feature"] = 0
```

Code Listing E.2: Classes used by sociotechnical environment

```
import environment

class Vertex:
    id = 0

    def __init__(self):
        self.adj = []  # A list of vertices this vertex points to
        self.timeCreated = environment.resources["time"]
        self.id = Vertex.id
        Vertex.id += 1


    def add_adjacent(self, other_vertex):
        self.adj.append(other_vertex)
```

```
15
16      def remove_adjacent(self, other_vertex):
17          self.adj.remove(other_vertex)
18
19      def adjacences(self):
20          return self.adj
21
22      def is_adjacent_to(self, other_vertex):
23          return other_vertex in self.adj
24
25      def age(self):
26          return environment.resources["time"] - self.timeCreated
27
28
29
30  '''
31      The chunks a bug affects are its adjacent vertices
32      i.e. if a bug applies to a chunk, add the chunk as an adjacent vertex
33  '''
34  class Bug(Vertex):
35      def __init__(self, buggy_chunk=None):
36          Vertex.__init__(self)
37          self.chunks = []
38          if buggy_chunk is not None: self.chunks.append(buggy_chunk)
39
40      def affects(self, chunk):
41          for affectedChunk in self.chunks:
42              if chunk.id == affectedChunk.id:
43                  return True
44          return False
45
46
47  '''
48      The bugs a chunk's tests will detect are the adjacent vertices
49      i.e. if a chunk detects a bug, add the bug as an adjacent vertex
50  '''
51  class Chunk(Vertex):
52
53      def __init__(self, test=None):
54          Vertex.__init__(self)
55          self.test = test
56
57      def is_tested(self):
58          return self.test is not None
59
60
61  '''
62      A test that applies to a chunk or series of chunks
63      If the test applies to more than one chunk, you should find them in the test
```

```
64  '''
65  class Test(Vertex):
66
67      def __init__(self, chunk=None, works=True):
68          Vertex.__init__(self)
69          self.chunk = chunk
70          self.works = works
```

Appendix F

Fuzzing Library

Code Listing F.1: Complete fuzzing library

```
import environment, random, ast, inspect

class ResourcesExpendedException(Exception):
    pass


class Mutator(ast.NodeTransformer):

    mutants_visited = 0

    # The mutation argument is a function that takes a list of lines and returns
    def __init__(self, mutation=lambda x: x, strip_decorators = True):
        self.strip_decorators = strip_decorators
        self.mutation = mutation

    #NOTE: This will work differently depending on whether the decorator takes a
    def visit_FunctionDef(self, node):

        # Fix the randomisation to the environment
        random.seed(environment.resources["seed"])

        # Mutation algorithm!
        node.name += '_mod'    # so we don't overwrite Python's object caching
        # Remove decorators if we need to So we don't re-decorate when we run th
        if self.strip_decorators:
            node.decorator_list = []
        # Mutate! self.mutation is a function that takes a list of line objects
        node.body = self.mutation(node.body)

        # Now that we've mutated, increment the necessary counters and parse the
        Mutator.mutants_visited += 1
        environment.resources["seed"] += 1
```

```
33          return self.generic_visit(node)
34
35
36  class mutate(object):
37
38      cache = {}
39
40      def __init__(self, mutation_type):
41          self.mutation_type = mutation_type
42
43      def __call__(self, func):
44          def wrap(*args, **kwargs):
45
46              if environment.resources["mutating"] is True:
47
48                  # Load function source from mutate.cache if available
49                  if func.func_name in mutate.cache.keys():
50                      func_source = mutate.cache[func.func_name]
51                  else:
52                      func_source = inspect.getsource(func)
53                      mutate.cache[func.func_name] = func_source
54                  # Create function source
55                  func_source = ''.join(func_source) + '\n' + func.func_name + '_m
56                  # Mutate using the new mutator class
57                  mutator = Mutator(self.mutation_type)
58                  abstract_syntax_tree = ast.parse(func_source)
59                  mutated_func_uncompiled = mutator.visit(abstract_syntax_tree)
60                  mutated_func = func
61                  mutated_func.func_code = compile(mutated_func_uncompiled, inspec
62                  mutate.cache[(func, self.mutation_type)] = mutated_func
63                  mutated_func(*args, **kwargs)
64              else:
65                  func(*args, **kwargs)
66          return wrap
67
68      @staticmethod
69      def reset():
70          mutate.cache = {}
```