Simulating Variance in Socio-Technical Behaviours using Executable Workflow Fuzzing

Completed Research Paper

# Introduction

*Socio-technical systems* are large scale, complex models, representing the interactions between a diverse set of actors including individual technical artifacts, human operators, organisational structures (El-Hassan & Fiadeiro, 2007; Sommerville & Dobson, 2008; Susman, 1976). The behaviour of such systems are influenced by a complex interplay of factors, including formally defined business processses, legal or regulatory standards, technological evolution, organisational culture or norms and interpersonal relationships and responsibilities \citep{bade07structures,pentland05organisational}. Examples of such systems with complex workflows involving multiple actors include emergency vehicle dispatch \citep{robinson96limited}, electronic voting systems \citep{bryans04towards,lock07observations}, patient care in a neo-natal unit \citep{baxter07evaluating} and electronic stock exchange infrastructures \citep{cftc-sec10findings}. Systems of this form are not susceptive to systems engineering modeling methods for several reasons:

* Socio-technical systems are simultaneously *very large and heterogeneous*, comprising a a mix of autonomous actors, each with their own behaviours \cite{crabtree00ethnomethodologically}. Systems engineering has traditionally approached the problem of scale through the development of models that abstract complex behaviours and model them as emergent system properties \cite{vespignani11model}. However, these stochastic treatments do not capture the complex interactions that occur between heterogeneous actors, with interactions occurring across different scales of activity. For example, \citet{lock07observations} observed the disruptions caused to a national election in Scotland caused by a variety of small scale technical system defects.
* The behaviour is contingent on unpredictable circumstances, including both factors in the environment and concerning the system actors. For example, the time and manner in which a task, such as developing a new feature for a software system, is completed may vary considerably between actors with different training and experiences. Similarly, the decision to work on a task at all may depend on unpredictable and uncontrollable external circumstances (such as a power outage). In these circumstances, actors may also take it upon themselves to complete tasks outwith expected workflows in order to discharge their responsibilities, by working from a nearby cafe for example, even if this violates organisational security policies. As \citet{besnard03human} note, such adaptations often make the human actors the dependable parts of a socio-technical system.
* Behaviour is continually evolving, as the autonomous actors in a system adapt to new circumstances, discover optimisations to their workflows, adapt the workflow to suit local organisational priorities or take shortcuts \citep{bonen79evolutionary,Lyytinen2008,anderson04heterogeneous}. As a consequence, the *de facto* behaviour exhibited within a system may differ from that envisaged by system architects in idealised workflows. For example, a ward manager in a hospital may delay releasing beds for re-allocation by wider hospital management in the anticipation that these will be required by incoming patients later in the day \citep{dewsbury07responsibility}. This evolution of practice may quickly invalidate expected models of behaviour.

We contend that due to these challenges, modelling socio-technical system behaviours using conventional systems engineering methods will typically either result in a model that is tractable, but lacks the necessary detail of the underlying system to provide informative results; so narrow in scope as to be uninformative about the behaviour of the wider system of interest; or so large and complex as to be intractable for analysis, whether manual or automated. Consequently, the design and construction of systems at this scale is still very much a craft, lacking the methods and tools to support modelling and predictive simulation available in other engineering disciplines.

The research contribution of this paper is to present and evaluate a novel environment, Fuzzi Moss, for simulating complex and contingent behaviour in socio-technical systems which addresses this challenge. In our approach, we provide for a separation of concerns between the model of a problem domain, models of idealised socio-technical actor behaviour and the influence of contingent factors that complicate the actual execution of idealised workflows in practice. The separation of concerns is achieved by modelling:

* The problem domain as collection of classes implemented in the Python programming language.
* Idealised workflows descriptions as executable Python classes in the agent oriented modelling framework, Theatre\\_Ag \citep{theatreag}.
* Contingent behaviour as *dynamic fuzzing aspects* that can alter the flow of execution in workflow descriptions during the execution of a simulation, using the PyDySoFu library \citep{wallis2017pydysofu}.

Both the Theatre\_Ag framework and PyDySoFu libraries were implemented specifically for this work.

Critical to the approach is our hypothesis that:

**Hypothesis: Dynamic fuzzing of workflow descriptions can represent the effect of complex and contingent behaviour by actors in socio-technical systems, when following idealised workflows.**

To test this hypothesis, an example socio-technical case study of team based software development was developed. The case study compares the performance of different software development processes when a software development team follows idealised workflows that have been subject to contingent behaviour. Development processes are compared based on their effect on the emergent properties of the simulated system under development, specifically features implemented and mean time to failure.

The rest of this paper is structured as follows. Section \ref{sec:related} discusses related work, covering existing techniques for modelling socio-technical workflows and other applications of code fuzzing in software engineering. Section \ref{sec:fuzzi-moss} presents the method for constructing models of socio-technical systems, associated workflows and denoting desired fuzzings. Where relevant, this section also discusses details of the implementation details for Fuzzi Moss. Section \ref{sec:evaluation} presents the case study evaluation of the method and Section \ref{sec:conclusions} discusses conclusions and future work, as well as noting the potential for applying fuzzing to other forms of socio-technical models.

# Related Work

This section presents a literature review of the development of models and of behaviours in socio-technical systems. The difficulties of developing modelling techniques that accommodate the inherent scale, complexity, contingency and dynamism of socio-technical systems are highlighted. In addition, existing applications of software fuzzing are reviewed with respect to their relevance to the present work.

Graphical notations have received considerable attention, perhaps due to their perceived efficacy in communicating requirements between users, customers and system architects. These modelling languages include workflow based approaches such as UML activity diagrams \citep{omg2010omguml}, BPMN \citep{omg2011omgbpmn}, YAWL \citep{hofstede2010yawl} and OBASHI \cite{obashimethodology}; and goal based approaches such as KaOS \citep{Werneck2009}, *i\** \citep{yu1995} and responsibility modelling \citep{sommerville09responsibility}.

Activity diagrams are perhaps the most commonly known workflow language, due to incorporation in the UML standard \citep{omg2010omguml}. The notation supports the modelling of the flow of control across a directed graph of activities, with arcs representing transitions in control. Additional nodes are provided for denoting entry and exit points, as well as decision branches. The notation is based on the Petri Net formalism and includes support for concurrent flows through the chart, as well as workflow forking and merging. The semi-formal nature of the UML standard enable the automatic parsing of graphical models, using CASE tools such as the Eclipse Modelling Framework \citep{EMFManual}. An advantage of this approach is that models can be used for negotiation between project stakeholders, whilst also being used for simulations to predict system behaviour.

The Business Process Model and Notation (BPMN) is an alternative OMG standard for modelling workflows, with similar core notation and semantics for modelling workflows \citep{omg2011omgbpmn}. Unlike activity diagrams, however, BPMN provides a richer notation for expressing more complex aspects of activities, such as differentiating between tasks, activities and transactions; triggering and orchestrating concurrent activities using messages; the identification of information resources need to realise an activity; and the orchestration of activities across organisational boundaries \citep{White2004}. The notation is intended to support the generation of executable business processes expressed as web services, however, it can also be employed in other workflow contexts.

Yet Another Workflow Language (YAWL) provides similar capabilities to activity diagrams for modelling workflows, as well as being supported by CASE tools for graphical modelling \citep{hofstede2010yawl}. However, unlike activity diagrams, YAWL is based on the \picalc\citep{Aalst2004}. The notation also provides for a richer range of workflow requirements than activity diagrams, including sophisticated forking and merging rules, separation between workflow specifications and executions and resourcing and data requirements.

The OBASHI (Ownership, Business, Application, System, Hardware, Infrastructure) methodology and notation \citep{obashimethodology} is designed for modelling business processes across enterprise infrastructures. The notation is intended for capturing the movement of data through a business process and revealing the associated dependencies on underlying infrastructure such as software systems, servers and network communications. The language also provides a means for mapping these flows to higher level concerns, such as business rationale and ultimate organisational owner. In contrast to other workflow notations, flows are based on the movement of data rather than control.

Describing socio-technical behaviour using workflow notations can be difficult, because of the basic assumption that all contingencies in a workflow can be completely described at a given level of granularity, and that more complex details can be encapsulated within coarser grained activities with well defined interfaces. As argued in Section \ref{sec:introduction}, socio-technical behaviours are inherently complex, contingent and evolutionary, making such refinement based techniques difficult to apply. As \citet{israilidis13ignorance} have argued, the unknowns in a socio-technical system may be far more significant than the knowns. Several authors have therefore discussed alternative techniques for modelling socio-technical systems with support for contingent behaviour \citep{yu1995,dardenne93goal,Herrmann1999,sommerville09deriving}.

Both *i\**\citet{yu1995} and KaOS \citet{dardenne93goal} are goal oriented notations for modelling socio-technical systems \citep{Werneck2009}. In contrast to workflows, goal oriented approaches primarily capture the intents of actors (what they are seeking to achieve). Goals can be de-composed into a sub-goal hierarchy using logical operators to express the form of decomposition. Goals can also be annotated with strategies and/or resource requirements to support automated analysis. \citeauthor{yu1995} argued that socio-technical systems should be viewed as collections of collaborating actors, each with their own (potentially conflicting) objectives. Eliciting and analysing the actors intents allows the inter-dependencies between actors and the overall behaviour of the system to be understood, without the need for explicit models of individual workflows.

Other authors have extended goal oriented approaches to provide greater flexibility. \citet{sommerville09deriving} argued that stakeholders often struggle to express their behaviour within a socio-technical system in terms of goals. Instead, \citeauthor{sommerville09deriving} argue that the concept of *responsibilities*, the duties held by an actor in a system, are a more intuitive means of describing system behaviours that also capture a variety of contingent behaviours. A notation for expressing the relationships between responsibilities and resources in order to identify dependencies within a system is provided. Earlier work on responsibility modelling also provided mechanisms with annotating responsibilities with indicative workflows, expressing the means by which responsibilities *could* be executed \citep{dewsbury07responsibility}.

Despite providing for contingency, a limitation of the goal and responsibility approaches is the need for complete model descriptions. \citet{Herrmann1999} introduced techniques for annotating goal oriented system models in the SeeMe notation with vagueness. The notation enables a modeller to denote where vagueness may be present in a model due to abstraction (i.e. consistent vagueness) and due to omission (inconsistent vagueness). In addition, \citeauthor{Herrmann1999} provide notation for indicating that a model is thought to be complete, containing all pertinent details. However, the annotations are not accompanied by a formal semantics, or other means of supporting automated analysis.

We are not aware of other applications of fuzzing techniques to modelling contingent behaviours in socio-technical systems. However, software code fuzzing (or mutation) is employed in software quality assurance in order to automatically generate program variants. *Mutation operators* in such applications may alter the value of literals, swap arithmetic or other operators, or change the ordering of arguments to a function call, for example. Applying different combinations of mutation operators creates a population of mutants of the target program. One application of this technique is mutation testing, in which the generation of program variants is used to simulate the introduction of defects and evaluate the effectiveness of an application's test suite in detecting regressions \citep{demillo78hints}. A test is considered to have detected a mutant if the application of the test to the mutant fails. A test suite that detects a higher proportion of mutants is considered to have good coverage of the target program.

The effectiveness of mutant generation is significantly influenced by choice of mutant operators to apply, since the search space of potential mutants to be tested is very large and many mutants will reveal the same test suite deficiencies \citep{takanen08fuzzing}. Generation of mutants based on an understanding of a system's specification allows mutant generation to be focused on a system's intended behaviour. It can therefore be expected that applying code fuzzing to simulating socio-technical behaviours requires an understanding of the likely variants to behaviour that may occur in a workflow in order to generate realistic simulations.

There are a variety of existing tools that incorporate fuzzing functionality for mutation testing, including PiTest \citep{coles14pitest} for Java and MutPy \citep{mutpy26} and PyMuTester \citep{pymuttester} for Python. \citet{storer15ringneck-repos} has also developed a tool for mutation testing Maven component assembly specifications. All these tools work by constructing and then manipulating abstract syntax trees of target programs. The result is a population of statically generated mutant programs that can be evaluated using the target program's own test suite. A disadvantage of this mechanism (for the purposes of modelling socio-technical systems) is that the mutants are generated statically, prior to program execution. Our own implementation of code fuzzing is motivated by the desire to simulate dynamic contingent behaviour, that can vary from the idealised model each time a fuzzed step in a workflow is executed.

# Problem Domain Model

In this section we introduce our approach to modelling a problem domain and associated idealised workflows in socio-technical systems. We have chosen to present the approach through an example case study of team based software development, in which we will explore the efficacy of two development workflows: waterfall and test driven development.

We have adopted an object-oriented approach to modelling the elements of the problem domain \citep{bennett06object}, so that the elements of the domain are described as a collection of Python classes. Figure \ref{fig:feature-class-diagram} shows the class diagram for the case study. The diagram shows classes for:

* Features, representing user-facing specifications of the system's functionality.
* Code chunks, representing the implementation details of the features, which may have dependencies on other chunks in the system.
* Bugs introduced into chunks during the completion of features.
* Software systems which aggregate all the source artefacts of a software project, including features, chunks, bugs and tests.
* Version control servers and clients for coordinating distributed development of a project.

Methods are implemented for these classes that provide them with behaviours in the problem domain. Many of these behaviours have side effects which are modelled stochastically. For example:

* Features can be extended through the addition of code chunks. Each time a new chunk is added to a feature other chunks may also need to be modified, potentially creating further dependencies between chunks or introducing bugs.
* Tests can be exercised resulting in the detection of bugs. The more tests created for a feature, the greater the probability of detecting a given bug.
* Features can be debugged (resulting in the removal of bugs) or refactored (resulting in the reduction in the number of dependencies).
* Software systems can be operated, which may cause bugs in the system to manifest themselves, causing a system halt.

No restrictions are placed on the implementation of the problem domain classes within the Python language. For example, operations can accept a variety of arguments, modify object state, invoke operations on other problem domain classes and return values as desired. The @Property decorator can also be used to improve the readability of code, as normal.

Idealised socio-technical workflows are collections of task descriptions that operate on a common state. Tasks are implemented as Python methods, with all the tasks associated with the same workflow and operating on the same state collected together in a single Python class. For the purposes of the software development case study, workflows were created for interacting with a version control server in an update-merge-commit cycle; specification of new features in a system; implementation of features; development of tests to exercise features; and debugging tests that revealed bugs and refactoring of features. Workflows can also be organised hierarchically, so for example, the workflows for modifying

the system artefacts depend on the change management workflow in order to coordinate changes within a team. Further

workflows were implemented for coordinating the overall team activities by following the waterfall and test driven

development methodologies. Figure \ref{fig:debugging} shows the Python code for the debuggin workflow, while

\ref{fig:tdd} shows the Python code for the Test Driven Development workflow.

# References

El-Hassan, O., & Fiadeiro, J. L. (2007). Role-based Architectural Modelling of Socio-Technical Systems. *Electronic Notes in Theoretical Computer Science*, *181*, 5–17.

Sommerville, I., & Dobson, J. (2008). *Socio-technical systems engineering*.

Susman, G. I. (1976). *Autonomy at Work: A Sociotechnical Analysis of Participative Management*. Praeger.