Modelling Realistic User Behaviour in Information Systems Simulations as Fuzzing Aspects

Tom Wallis and Tim Storer

University of Glasgow, Glasgow, Scotland.
`w.wallis.1@research.gla.ac.uk`
`timothy.storer@glasgow.ac.uk`

Abstract In this paper we contend that the engineering of information systems is hampered by a paucity of tools to tractably model, simulate and predict the impact of *realistic* user behaviours on the emergent properties of the wider socio-technical system, evidenced by the plethora of case studies of system failure in the literature. We address this gap by presenting a novel approach that models ideal user behaviour as workflows, and introduces irregularities in that behaviour as aspects which fuzz the model. We demonstrate the success of this approach through a case study of software development workflows, showing that the introduction of realistic user behaviour to idealised workflows better simulates outcomes reported in the empirical software engineering literature.

1 Introduction

Information systems are operated within a wider organisational context, characterised by the needs, demands and behaviours of individual users, interpersonal relationships, organisational structures, business processes, legal and regulatory standards, and cultural norms [? ?]. The influence of this *socio-technical* interplay on system behaviour (and often failure) has been described in multiple and diverse case studies of information systems, including automated emergency vehicle dispatch [?], electronic voting [?] and stock market trading systems [?]. In each case, system failure cannot be attributed to either purely user behaviour or the information system(s), but instead to an interplay between both factors within the overall socio-technical system.

The contention in this paper is that these failures arise because systems engineers lack the tools and methods to efficiently model and accurately simulate the interaction between the information systems and their organisational context, which comprise a socio-technical system. Without such facilities, systems engineers cannot predict potential socio-technical system behaviour during information system design. Simulating this interaction between the information system and users is hard because user behaviour is heterogeneous, contingent and evolutionary, leading to irregularities in workflows envisaged by systems engineers. Different users have different abilities, training and experiences and can experience phenomena such as distraction, misjudgements, exhaustion and confusion that can have significant influence on how and when a user completes a task. For example, a novice developer working within a large software team may be unaware of software development best practices described in a workflow, perhaps forgetting to make frequent commits to a version control server. Conversely, an experienced developer may omit steps that they consider unnecessary, such as peer reviews of their code, to optimise their performance.

Information system users may also adapt their behaviour due to contingencies that arise from faults in the information system or the behaviour of other users in the environment [**?**]. Continuing the example scenario, a software development team may be tempted to begin reducing quality assurance efforts as a deadline for a release approaches, in an effort to ensure all required features are completed [**?**]. Behaviour is also continually evolving, as the users of a system adapt to new circumstances, discover optimizations to their workflows, adapt the workflow to suit local organisational priorities or take shortcuts [**?**]. In the example scenario, it is reasonable to anticipate that a novice developer's behaviour will gradually evolve into that of an expert as they gain more experience. As a consequence, the de facto behaviour exhibited within a system may differ from that envisaged by system architects in idealised workflows.

Despite the potential impact on system performance, the complexity of user behaviour is cumbersome to model and therefore predict using conventional systems engineering notations for describing workflows, such as BPMN [**?**], activity diagrams [**?**] and YAWL [**?**]. Attempting to model all the potential sequences of actions that result from realistic user behaviour using these approaches inevitably results in models that are either too abstract or narrow to be informative, or too complex to be tractable. Approaches that abstract the complexity of user behaviour, such as i* [**?**], Kaos [**?**] and Responsibility Modelling [**?**] lack sufficient detail to generate executable simulations.

The key insight of this paper is that the same behavioural irregularities can affect many different idealised workflows. Conversely, a given user's behaviour can be modelled as the combination of expected ideal behaviour and the irregularities that affect how that user exhibits their behaviour. Therefore, *we propose and demonstrate the separate modelling of behavioural irregularities from workflows themselves, showing that they can be represented as cross-cutting concerns which can be applied to an idealised workflow model.*

The two contributions of this paper are as follows:

- A method for simulating user interactions with information systems that allows for the separate modelling of idealised descriptions of workflows and the effects of irregularities on those workflows. The approach is implemented in a framework comprising of aspect oriented programming [**?**] and dynamic code fuzzing [**?**]. This combination allows for irregularities to be introduced into workflow models without sacrificing readability or simplicity.
- A demonstration of the efficacy of the approach in a case study comparison of Test Driven Development (TDD) and Waterfall software development workflows. The case study demonstrates that models of idealised behaviour

in software development do not conform with the empirical evidence which suggests that TDD outperforms Waterfall [**? ? ?**], but that models which incorporate realistic user behaviour when interacting with information systems do.

The rest of this paper is structured as follows. Section 2 discusses related work, covering existing techniques for modelling socio-technical workflows and the limitations encountered in the literature. Section 3 introduces the case study problem domain selected to evaluate our approach, and presents the method for constructing models of information systems and associated workflows. Section 4 describes the development of aspect oriented fuzzing of user behaviour and the software tool developed for this purpose. Section 5 presents our evaluation of the case study and Section 6 discusses conclusions and future work.

2 Related Work

This section presents a literature review of approaches to modelling the interaction between users and information systems. Graphical notations have received considerable attention, perhaps due to their perceived efficacy in communicating specifications between users, customers and system architects. Workflow description languages, such as as UML activity diagrams [**?**], BPMN [**?**] and YAWL [**?**] can be used to denote workflows visually as directed graphs composed of activities, as well as forking and merging workflows. Although activity diagrams are more widely adopted than BPMN, the latter provides a richer modelling notation, allowing for discrimination 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. YAWL 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.

Describing realistic user behaviour in workflow notations can be difficult, because of the basic assumption that all paths in a workflow can be completely described at a given level of granularity, and that more complex details can be encapsulated within coarser grained modules. As argued in Section 1, user behaviours are inherently complex, making such refinement based techniques difficult to apply. As **?**] 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 complexity in behaviour. Both i* [**?**] and KaOS [**?**] are goal oriented notations for modelling socio-technical systems [**?**]. In contrast to workflows, goal oriented approaches primarily capture what system actors 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. **?**] 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. **?**] introduced techniques for annotating goal oriented system models with vagueness. The annotation allows for the distinction between consistent vagueness (due to abstraction) and inconsistent vagueness due to omission. In principle, this approach allows for the identification of aspects of a model that may be subject to irregularity. However, the notation is not accompanied by a formal semantics, or other means of supporting automated analysis.

Other authors have extended goal oriented approaches to provide greater flexibility. **?**] argued that stakeholders often struggle to express their behaviour within a socio-technical system in terms of goals. Instead, they 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 contingencies. Various notational features have been provided for extending responsibility modelling, including for linking between responsibilities and required resources and for associating indicative workflows [**?**]. However, these are not accompanied with a means of evaluating the effects.

The contributions made by **?**] and **?**] demonstrate the recognition of the need for methods which capture the complexity of user interactions with information systems. However, to the best of our knowledge there has been no attempt to go further and develop methods which can simulate the effects of this behaviour on the overall system. The observation made in Section 1 that the causes of irregularities in workflow execution are separate concerns from the model of the workflow itself invites a novel approach to the problem of simulating realistic user behaviour. This approach combines dynamic software fuzzing, which provides variation in the workflow model, with aspect oriented programming (AOP), to decouple this behavioural variation from the idealised model.

The representation of cross-cutting concerns, such as access control, logging and concurrency as program aspects has been extensively studied in the literature [**?**]; and the use of code fuzzing techniques have been employed extensively in quality assurance practices, to assess the impact of unexpected events on software systems, including fuzzing of software configurations at runtime [**?**], mutation testing of test suites [**?**] and fuzzing of inputs test data to search for software security vulnerabilities [**?**]. However, we have not discovered any similar work on the application of fuzzing techniques to dynamically inject variation into a simulation model at runtime in order to mimic complex user behaviour. Similarly, we are unaware of any research that utilises AOP techniques in simulation models, either for socio-technical user behaviours or elsewhere.

3 Team Based Software Development Case Study

In this section we demonstrate our approach to modelling information systems and associated idealised workflows in socio-technical systems through a case study. We have chosen a case study of team based software development, in which we will explore the efficacy of the workflows of two software development lifecycles (SDLC): Waterfall and Test Driven Development (TDD). The case study was chosen as representative of a socio-technical system, combining different user roles (developer, project manager), information systems (version control server and client, and the software development effort itself) and a variety of well documented idealised workflows (implementation, testing, debugging etc.). Further, there is a growing consensus amongst software development professionals and in the academic literature that TDD is a more resilient SDLC than Waterfall to realistic human behaviour [**? ? ?**]. The case study therefore provides an opportunity to test whether the modelling approach can be used to distinguish between the effects of idealised and realistic user behaviours in a socio-technical system. The model is implemented in an agent oriented framework, Theatre_Ag, written in Python. Fragments of the case study are provided as examples, but the full source code is also available for inspection [**?**].

The first step in the modelling approach is to represent the state and functions of the information systems in the case study. The domain is modelled as a collection of Python classes, as shown in Figure 1, following an object oriented approach [**?**]. The core information system is a version control server (VCS) which hosts the software development effort. Software developers interact with the server via a VCS client. Both the server and clients maintain copies of the software system, which must be coordinated through a VCS update, conflict-resolve, commit cycle. Software systems are composed of features, representing user-facing specifications of the system's functionality. Features may be of varying size, requiring more or fewer code chunks to be implemented in order to be operational. Chunks may have dependencies on other chunks in the system. 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. Features can be operated, causing some of the underlying chunks and any dependencies to also be operated. Bugs may become manifest during chunk operation, or through the execution of tests. Features can be debugged, resulting in the removal of detected bugs. Consequently, the more tests created for a feature, the greater the probability of detecting a given bug, easing the process of debugging. All of the information system behaviours described above and shown in the diagram are implemented as methods in Python classes.

The second modelling step is to define user behaviour workflows that will be followed by users, represented by agents in the simulation environment. These are also implemented as Python classes, collating related tasks to operate on a common state (the information systems). Figure 2 shows classes for two of the workflows created for the case study: change management and implementation. The state for the change management workflow is the VCS server and a client that manages the working copy. Separate task methods are provided for checking

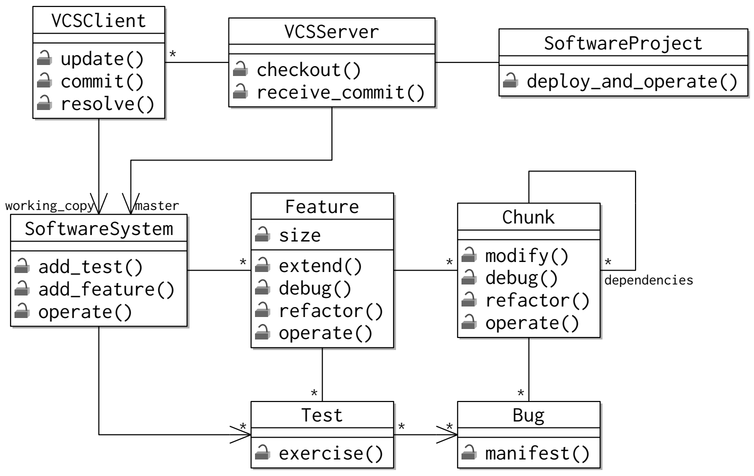


Figure 1: The software development case study domain model, showing information system artefacts and functions that can be accessed by user workflows.

out the client, committing working copy changes to the server and resolving conflicts that arise during a commit task.

Note that many of the functions of the information system have side effects that are modelled stochastically, such as the possibility that an update from the version control server will result in conflicts with another client's commit. The workflows therefore explicitly pass a source of randomness to the information system model when these actions are invoked.

Further workflows were implemented for common software development activities, including the specification and implementation of features in the system, debugging and refactoring (reducing dependencies), and testing. Since workflows are modular they can also be organised hierarchically. Figure 2 shows how this modularity is exploited for the implementation workflow. The other workflows for modifying the structure of a software system (for adding chunks or tests and so on) also depend on the change management workflow in order to coordinate the distribution of changes within a team.

Two further workflows, Waterfall and Test Driven Development (TDD) were implemented to investigate the performance of different team based software development lifecycles (SDLC). The Waterfall SDLC [**?**] describes a linear staged approach to system development, in which specification of all features is followed by system implementation, testing, debugging, refactoring and final deployment. Conversely, TDD [**?**] prescribes the definition of tests for each feature, before proceeding to implementation and refactoring. Advocates of TDD argue that such an approach results in software that is delivered quicker and of higher quality because tests are explicitly linked to the specification and features are not committed without a passing test suite.

```
class ChangeManagement(object):
    def __init__(self, vcs_server):
        self.vcs_server = vcs_server
        self.vcs_client = None

    @default_cost(1)
    def resolve(self, conflict, rand):
        self.vcs_client.resolve(conflict, rand)

    @default_cost(0)
    def commit_changes(self, rand):
        while True:
            try:
                self.vcs_client.commit()
                self.vcs_client.update(rand)
                break
            except CentralisedVCSException:
                self.vcs_client.update(rand)
                for conflict in self.vcs_client.conflicts:
                    self.resolve(conflict, rand)

    @default_cost(0)
    def checkout(self):
        self.vcs_client = self.vcs_server.checkout()


class Implementation(object):
    def __init__(self, change_management):
        self.change_management = change_management

    @property
    def working_copy(self):
        return self.change_management.vcs_client.working_copy

    @default_cost(1)
    def add_chunk(self, chunk_logical_name, feature, rand):
        feature.extend(chunk_logical_name, rand)

    @default_cost()
    def implement_feature(self, logical_name, rand):
        self.change_management.checkout()

        feature = self.working_copy.get_feature(logical_name)

        while not feature.is_implemented:
            self.add_chunk(len(feature.chunks), feature, rand)
            self.change_management.commit_changes(rand)

    @default_cost()
    def implement_system(self, rand):
        self.change_management.checkout()
        for feature in self.working_copy:
            self.implement_feature(feature, rand)
```

Figure 2: Python code for workflows describing change management and software implementation.

Once the models of the information systems and workflows are complete, simulations can be executed in Theatre_Ag, which are initiated by allocating a set of tasks (individual methods in a workflow) to agents in the system. This may lead to the creation and execution of further tasks by agents, or to the allocation of tasks to other agents. In the case study, simulations were configured to represent a team of software developers working on a software project following both types of SDLC. For Waterfall, the initial configuration specified an agent in the team with the role of project manager. This agent is directed to allocate tasks for each stage of the software development process (specification, implementation, testing, debugging, refactoring) to other agents in the team and waits for all tasks in each stage of the development process to be complete before allocating tasks in the next stage. For TDD, all members of the software development were configured to draw features to be implemented from the project backlog and implement them following TDD.

All task execution is coordinated relative to a single simulation clock, enabling both the explicit representation of problem domain time for controlling task execution and concurrent execution of these tasks by agents. Task costs are denoted using the `@default_cost` decorator in a workflow as shown in Figure 2. The agent pauses execution of a task until sufficient ticks from the clock have been received and then resumes execution. This mechanism is implemented transparently with respect to workflow classes, easing the modelling burden. The execution of tasks and their duration is logged in a tree-like structure by each agent, supporting later inspection and analysis of behaviour.

4 Modelling Irregularity in Behaviour as Fuzzing Aspects

In this section the approach to modelling irregularities in user behaviour is demonstrated. To achieve this, an irregularity is modelled as a *dynamic fuzzing aspect* using the library developed for this purpose, PyDySoFu [**?**]. PyDySoFu interrupts the invocation of workflow task methods in the background during the execution of a simulation. When a workflow task method that can be subject to irregularity is invoked, PyDySoFu pauses the simulation thread of execution and passes the method context and structure (represented as the abstract syntax trees of the statements in the method body) to a modeller defined fuzzing function. The fuzzing function then manipulates the structure of the method AST and context as desired and returns the AST to PyDySoFu. In a final stage, PyDySoFu compiles the altered AST and resumes simulation execution by invoking the altered method. Further implementation details are omitted for brevity, but the full source code of the PyDySoFu library and documentation is available for inspection [**?**].

In order to evaluate the feasibility of the approach, a fuzzing aspect that mimics *distraction* was implemented using PyDySoFu. The distraction aspect represents irregularities in expected behaviour when a user engaged in one task becomes distracted by another, irrelevant activity and so does not complete later steps in the intended workflow. The probability of distraction increases as the

```
def default_pmf(concentration=1.0):

  def _probability_distribution(remaining_time, probability):
    adjusted_probability =\
          probability ** ((remaining_time + 1.0) * concentration)

    return sys.maxint if adjusted_probability == 1.0 \
        else int(1.0 / (1.0 - adjusted_probability) - 1)

  return _probability_distribution

def incomplete_procedure(random, pmf=default_pmf()):

  def _incomplete_procedure(steps, context):
    clock = context.agent.clock
    remaining_time = clock.max_ticks - clock.current_tick

    n = pmf(remaining_time, random.uniform(0.0, 0.9999))

    fuzzer =\
      recurse_into_nested_steps(
        target_structures={ast.While, ast.For, ast.TryExcept},
        fuzzer=filter_steps(
          choose_last_steps(n, reapply=False),
          replace_steps_with(
            replacement='self.agent.idling.idle()'
      )))
    return fuzzer(steps, context)

  return _incomplete_procedure
```

Figure 3: An aspect which dynamically truncates the execution of a workflow to represent a distracted user operating a system.

simulation proceeds, i.e. the probability of moving to an irrelevant task increases the longer a simulation has been executing. For example, a software development team may be distracted from the implementation of tests and debugging as the pressure to complete a set of features increases, due to an impending release deadline. Conversely, the aspect also allows the probability of distraction to be reduced by increasing the concentration of a simulated user (i.e. users with better concentration do not get distracted as easily). The implementation of the `incomplete_procedure` fuzzing aspect is shown in Figure 3.

The fuzzing aspect is parameterised, so the actual aspect is defined as the inner function `_incomplete_procedure`. The first four lines of the inner function configures a probability mass function (PMF) for choosing how many steps should be removed by the aspect, `n`, based on the remaining time in the simulation and the agent's concentration. The default PMF is shown in the figure, although this can be configured by the modeller.

PyDySoFu is accompanied by a library of aspects from which more complex aspects can be constructed and parameterised. This includes functions for recursing into control structures and selecting and replacing steps based on their position in the AST. These features are exploited in the second part of the fuzzer, which recurses into any nested control structures (while, for and try) found in the method and applies a filter to the bodies of these statements, tail first. The `filter_steps` aspect selects up to `n` steps to remove from the body and replaces each of them with an invocation of `idle()`, causing the agent executing the workflow to wait one tick for each replaced step. The `filter_steps` aspect is then applied successively up through the target AST. The `choose_last_steps` filter is configured to track how many steps it has already removed, so that the value of `n` is calculated across the entire body of the fuzzed task method, rather than at each level in the recursion.

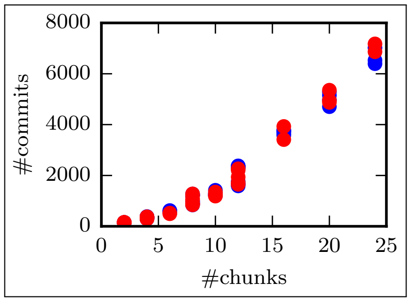
Notice that the fuzzing aspect models the effects of distraction *independently* of its application to a particular workflow. This allows the effects of this irregular behaviour to be applied transparently to any of the different user workflows in the simulation. In the current work, it is not claimed that the implementation of the aspect is empirically accurate, rather that it is representative of a qualitatively recognisable phenomena that is better modelled separately from the workflow; and that the approach allows for the effect of the phenomena on a workflow at different degrees of severity to be assessed. Also, it is not claimed that distraction is the *only* cause of irregularities in behaviour in software development workflows. Rather, we present distraction as an example of a cause of irregularity that may affect many different workflows in the same way, and demonstrate the approach to modelling it separately as a cross-cutting concern.

5 Evaluation

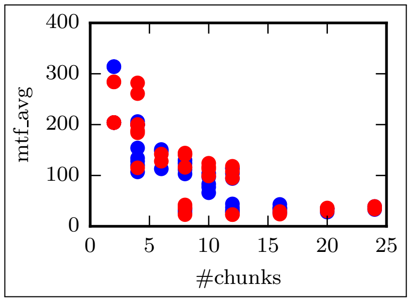
This section presents the results of applying the user behaviour fuzzing method to the case study. Evaluating the correctness of simulation techniques intended for large scale systems is notoriously difficult, since the usual rationale for developing the simulation is the lack of available empirical observations of the real world phenomena [**?**]. However, in the context of the case study, limited empirical evidence exists that suggests TDD outperforms Waterfall in terms of features delivered and software quality [**? ? ?**]. The evaluation will therefore adopt a hybrid strategy.

Following **?**], the model of the problem domain will first be tested for *plausibility*. A comparison of simulations of user behaviour with and without the application of dynamic aspect fuzzing will then be made. The first comparison will test whether two SDLC workflow models perform equivalently when executed in ideal circumstances, i.e. that simulations of ideal behaviour do not conform with results presented in the literature. The second comparison will test whether the TDD workflow simulation out-performs Waterfall when user behaviour varies from the ideal workflow due to the effects of distraction, in conformance with the literature.

Simulations of the problem domain were configured as follows. A single source of randomness was initialised for each configuration with a constant seed to provide for repeatability of the experiment and to provide for comparison between configurations. All configurations were executed for a team of 3 software developers, with simulations executing for a maximum of 500 ticks. Each



(a) Commits against project size.



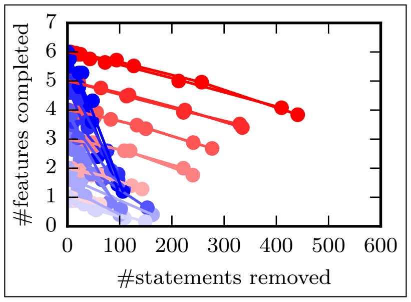
(b) Mean time to failure.

Figure 4: Scatter plots of software development simulations without fuzzing for Waterfall (blue) and TDD (red) projects.

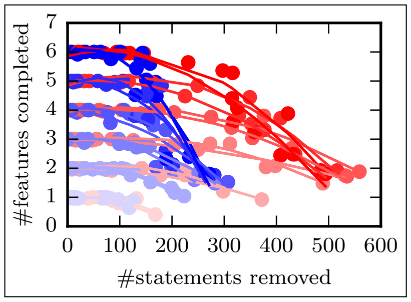
configuration was run for 25 different projects. The system 'built' by the simulation was operated 10 times, in order to measure software quality. Operation of the simulated system entails random selection and operation of features up to 750 times. If a bug is manifested during operation then the system operation halts. The length of the average trace of the operations of the system was recorded as the system's mean time to failure (MTF). Software projects consisted of up between 1 and 6 features, with projects divided into small (2 chunks per feature) and large (4 chunks per feature). Both the Waterfall and TDD SDLC workflows were simulated. In either case, the incomplete procedure distraction aspect was applied to either the high level software development workflow, or to the set of lower level development activity workflows (implementation, change management etc.). The aspect was applied using concentration values between 0.001 and 5.0. Raw simulations results are available for inspection [**?**].

In the first part of the evaluation, simulations of the software development workflow were considered for ideal behaviours, without the effects of distraction applied. Figure 4 illustrates two comparisons of simulations to establish problem domain plausibility. Figure 4a shows that there is an exponential increase in the number of commits to the version control server as project size increases. This result is to be expected due to the exponential increase in the number of additional modifications, tests, debugging and refactoring activities that would be required as a project grows in scale. Similarly, Figure 4b shows a decline in project quality (a reduction in the MTF from 300 to just above 0) as project size increases. Again, the additional complexity of larger projects increases the probability of a bug causing a system failure. These results strengthen the claim that the model of the problem domain is plausible.

A further observation from the figures is that TDD and Waterfall workflows perform equivalently when workflows are not subject to user distraction. When ideal workflows are executed for both Waterfall (blue) and TDD (red), the results show similar metrics for each. This confirms the expectation that



(a) Fuzzing of Waterfall and TDD.



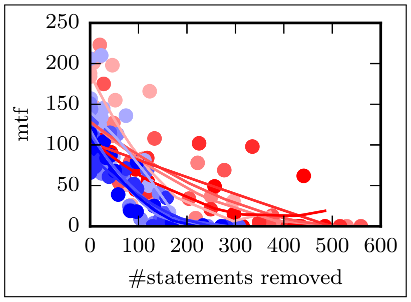
(b) Fuzzing of low-level workflows.

Figure 5: Scatter plots and trend lines of feature completion for Waterfall (blue) and TDD (red) for distraction applied to selected workflows.

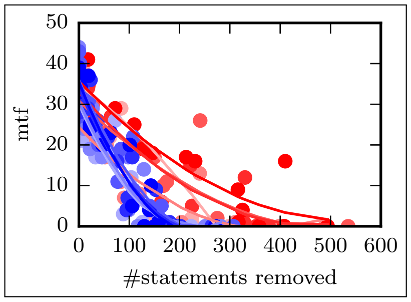
the simulations of idealised workflows, while behaving plausibly, do not conform with expectations from the literature that TDD should outperform Waterfall.

We now proceed to compare the performance of the two strategies when subject to behavioural irregularity caused by distraction. Figure 5 shows scatter plots for the effect of distraction for subsets of workflows on productivity. In both plots, the number of features implemented is plotted against the total number of workflow statements removed by fuzzing (Waterfall in blue, TDD in red). Figure 5a shows the effect when only the high level software development workflow (Waterfall or TDD) is fuzzed, whereas Figure 5b shows the effect of fuzzing lower level activities (change management, implementation, testing, debugging and refactoring). Both figures clearly demonstrate that increasing distraction reduces the number of completed features. However, in both configurations, TDD is more resilient to distraction fuzzing than Waterfall, with a larger proportion of features implemented as steps are removed. In Figure 5, both SDLC workflows appear to be largely immune to distraction until around 100 steps are removed, after which feature completion declines, with steeper declines for Waterfall.

Figure 6 shows the effect of distraction on MTF, again distinguishing between Waterfall (blue) and TDD (red) SDLCs. In these plots, both workflow fuzzing configurations described above are plotted together, as their effects are similar. Instead, small and large feature projects are plotted separately, as these have different orders of magnitude MTF. Similar to the plots of feature completion, the plots of MTF suggest that TDD is more resilient to fuzzing than Waterfall, which experiences a much more dramatic decline as more statements are removed. In summary, the two comparisons of productivity and quality demonstrate that the application of distraction fuzzing results in more realistic simulations of user behaviour in software development workflows, as found in the available literature.



(a) Small projects.



(b) Large projects.

Figure 6: Scatter plots and trend lines of MTF for Waterfall (blue) and TDD (red) for distraction applied to all workflow combinations.

6 Conclusions

This paper has presented and evaluated a novel approach to modelling and simulating realistic user behaviours when interacting with information systems, through the use of model fuzzing aspects. The novelty of this approach is the ability to model the causes of realistic user behaviour (such as distraction) separately as aspects that can be applied flexibly to many different workflows in a simulation. The efficacy of the approach is demonstrated in the introduction of realistic user behaviour to an example system simulation, which better conforms with results in the literature without altering the original workflows.

This proof of concept presents the opportunity for a substantial research agenda in the modelling and simulation of user interaction with information systems, in order to better predict emergent system properties. Further research is necessary to test the viability of the approach in more complex, large scale simulations, such as the election e-counting system described by **?**], comprising whole information infrastructures and many diverse users. This will provide opportunities to experiment with a variety of other causes of irregularity in user behaviour, such as subjective misjudgements, exhaustion, reordering of steps and shortcut taking. Interdisciplinary research is required to develop and validate the realism of the fuzzing aspects that implement these user behaviours, and a library of pre-validated aspects could be made available to ease modelling. There is a need to link empirical evidence of the causes and occurrences of these behaviours to their impact on a workflow's simulation and real-world execution.

The overall aim of this research agenda is towards simulation techniques that can have predictive capabilities suitable for informing systems engineering decisions, before resources are allocated toward them. Such tools would do much to progress the current craft of large scale information systems engineering.

Acknowledgements

The authors would especially like to thank Gordon Baxter, Phil Trinder and Russell Lock for their help when revising this paper, and OBASHI Technology for helping to fund the work.