Behave Nicely! Automatic Generation of Code for Behaviour Driven Development Test Suites

Ruxandra Bob
School of Computing Science
University of Glasgow
Glasgow, United Kingdom
2126189B@student.gla.ac.uk

Tim Storer
School of Computing Science
University of Glasgow
Glasgow, United Kingdom
timothy.storer@glasgow.ac.uk

***Abstract*—Behaviour driven development (BDD) has gained widespread use in the software industry. System specifications can be expressed as test scenarios, describing the circumstances, actions and expected outcomes. These scenarios are written in a structured natural language (Gherkin), with each step in the scenario associated with a corresponding step implementation function in the underlying programming language.**

A challenge recognised by industry is ensuring that the natural language scenarios, step implementation functions and underlying system implementation remain consistent with one another, requiring on-going maintenance effort as changes are made to a system. To address this, we have developed behave_nicely, a tool, for automatically generating step implementation functions from structured natural language steps, with the intention of eliminating the need for maintaining step implementation functions.

We evaluated our approach on a sample of 20 white box and 50 black box projects using behaviour driven development, drawn from GitHub. Our results show that behave_nicely can generate step implementation functions for 80% of the white box and 17% of black box projects. We conclude that (a) there is significant potential for automating the process of code generation for BDD tests and (b) that the development of guidelines for writing tests in Gherkin would significantly improve the results.

***Index Terms*—Behaviour driven development, code generation**

I. Introduction

Behaviour driven development (BDD) was first introduced by North in 2006 [1] as a means of expressing system specifications as test scenarios in a structured natural language, Gherkin. Each scenario describes the circumstances of the test from the perspective of a user role, the action that the user takes on the system and the expected observable changes. Each of the statements within the scenario can be linked to a step implementation function (step function) in the target system's programming language. This linking enables the collection of scenarios written for the system specification to be used as an executable a test suite. Several authors have advocated the benefits of this approach. Carrera et al [2] advocate for BDD as a means of fostering communication between customers, users and developers. Keogh [3] argues that BDD encourages developers to focus on delivering features of value to users, whilst also forcing the elaboration of poorly understood requirements. Solis & Wang [4] argue that the combination of features enables a traceable integration between a readable system specification on the one hand and executable acceptance tests and implementation on the other.

BDD has been widely adopted in the software industry. Software tools to support the process have been developed for a wide variety of programming languages, including Python [5], Java [6], Ruby [7] and .NET [8] amongst many others. A recent survey estimated that more than a fifth of teams were employing behaviour driven development [9]. A second survey reported that approximately three quarters of the (self-selecting) respondents planned to use BDD as either a key tool or as an optional tool in future projects. Half of the respondents also described BDD as either very important or important as a tool within their projects [10].

Despite this popularity, there is growing recognition that adoption of BDD incurs significant maintenance costs. The same survey of software industry practitioners described above reported that 20% of respondents reported that use of BDD reduced team productivity. The respondents also stated that BDD tests can be hard to change as the system requirements evolve, as one respondent observed "Changes required to be done in more than one place." This problem arises because existing BDD practice requires development teams to maintain two versions of their tests (the Gherkin scenario and the corresponding step functions) and ensure they remain consistent with one another. This may be further complicated if a step is used in more than one test scenario, as changes to the step to satisfy a change to one scenario may cause another to fail. The authors of the survey conclude that there is a significant opportunity to "Investigate automated test repair for BDD specifications".

This paper addresses this maintenance challenge. To do so, rather than focusing on test case repair, we investigate whether step functions can be automatically *generated* from the structured natural language steps in the source Gherkin scenario. If step functions can be automatically generated from the scenarios, then this would reduce the need for the step functions to be manually defined and maintained alongside the Gherkin scenarios. Our hypothesis is therefore:

> *Hypothesis* Test code step definition functions can be correctly generated from scenarios expressed in the Gherkin structured natural language.

Our approach to investigating this hypothesis exploits the use of Natural Language Processing (NLP) to extract the core features of a step function from a Gherkin scenario. We anticipate that NLP will be of use this purpose for two reasons. First, the steps in a Gherkin scenario are explicitly labelled according to their purpose, either to establish the pre-conditions for a test (Given steps), perform a user action on that fixture (When steps) or confirm that a post-condition has been satisfied by the system (Then steps). Therefore, we anticipate that if a Gherkin scenario has been expressed correctly, the purpose of each step in the corresponding step functions should be explicit and restricts the possible code that should need to be implemented to realise it. Second, individual scenario steps are typically terse and explicit, allowing for the identification of critical information.

Contribution: Three contributions are made in this work. First, we develop and present an implementation of our approach, behave_nicely [11], targeting the Python language and `behave` BDD package. We demonstrate that behave_nicely can be used to successfully generate step functions for Python based only on the content of Gherkin language scenarios. Previous efforts at code generation from Gherkin focused on Java and leveraged the static type system to infer the details of step functions. Second, we develop a novel method for evaluating the reliability of code generation for test suites using mutation testing [12]. Finally, we perform the first significant evaluation of step function generation. To date, attempts at code generation from Gherkin scenarios have been limited to case studies of single projects [13]. In contrast, we evaluate behave_nicely against a sample of 20 white box and 50 black box open source projects employing `behave` and retrieved from the GitHub open source project repository [14]. We demonstrate that behave_nicely was able to successfully generate step functions for 80% of the white box and 17% of black box projects.

The paper is structured as follows: Section II presents an overview of the BDD workflow, including an illustrative example of both Gherkin and a suite of step function implementations. Section III describes related previous work on test code generation from formal requirements specifications and other approaches, and code generation from natural language requirements. Section IV provides an overview of the tool we developed and Section V describes the design of our experiment to evaluate behave_nicely using our novel approach to comparing generated code using mutation testing. Section VI presents the results of our experiments and Section VIII section VII discusses the implications of the results and identifies potential future work.

II. Behaviour-Driven Development and Gherkin

This section provides a brief tutorial on the BDD workflow. The examples are presented using the `behave` package for Python [5], although the approach is similar for other target programming languages. The overall workflow is shown in Figure 1. In some respects, Behaviour Driven Development mimics the workflow for Test Driven Development [15], in that developers write tests that initially fail, before proceeding to implementation.

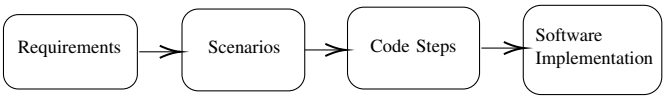

Fig. 1: Conventional Behaviour driven development workflow

```
Feature: balance management
  As a customer
  I want to perform transactions on my bank
    account
  So that I can manage my finances

  Scenario:
    Given a bank account with initial balance of 0
    When I deposit an amount of 100 into the bank
      account
    And I withdraw an amount of 50 from the bank
      account
    Then the balance of the bank account should
      be 50
```

Fig. 2: Example Gherkin Scenario

The first stage in BDD is for system specifications to be expressed as user stories with associated test scenarios using the structured natural language, Gherkin. An example of a user story and associated test scenario is given in Figure 2. The figure shows a user story for managing money in a banking application. In addition, a single test scenario is shown including some transactions on a bank account. The scenario comprises three types of steps:

- **`Given`** steps describe the pre-conditions that must be satisfied before any user action can proceed, including the fixture to the tested that must be created.
- **`When`** steps describe the actions taken by a user on the system.
- **`Then`** steps describe the post-conditions that should be satisfied if the system meets the specification.

In addition, **`And`** steps are continuations of the previous type of step.

Gherkin scenarios are stored in a plain text file with .feature extension within the project test suite. `Behave`, like many BDD implementations, searches for these files during the execution of the test suite. Once all the scenarios have been discovered, the tool then searches the project code base for functions that have been annotated with the names of scenario steps contained in the Gherkin file. In `behave` these functions are conventionally stored in a steps.py file within the test suite. In the second step in the BDD workflow, this code must be implemented and maintained by the developer, although many BDD tools support automatic generation of empty stub functions.

Figure 3 illustrates the implementation of step functions given in 2. The figure shows four `step_impl` step functions, annotated with the steps defined in the scenario. Notice that the implementation is characteristic of the Gherkin statement

```
from behave import given, when, then
from banking import BankAccount

@given('a bank account with initial balance of 0')
def step_impl(context):
    context.bank_account = BankAccount(0)


@when('I deposit an amount of 100 into the account')
def step_impl(context):
    context.bank_account.deposit(100)


@when('I withdraw an amount of 50 from the account')
def step_impl(context):
    context.bank_account.withdraw(50)


@then('the balance of the bank account should be 50')
def step_impl(context):
    assert context.bank_account.balance == 50
```

Fig. 3: Step functions implemented in `behave` for Python

```
class BankAccount(object):

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

    def deposit(self, amount):
        self.balance = balance + amount

    def withdraw(self, amount):
        self.balance = balance - amount
```

Fig. 4: Complete implementation of bank account example

type. **`Given`** steps are implemented by the creation of a fixture (in this case a bank account). **`When`** steps perform actions on the fixture (credits and withdrawals) and **`Then`** steps contain assertions. Each Gherkin statement in a test scenario with a matching function is executed in turn within the test. If no matching function is found for a statement, or the statement has been stubbed then the test scenario fails.

Finally, the developer creates the implementation of the application code to satisfy the test scenarios. Figure 4 illustrates the implementation of the example BankAccount class from the banking application. At this stage the suite of test scenario can be executed again using `behave` and should pass. Development may now proceed to new features.

Note, however, that should detail of the implementation change, then the step functions and possibly the Gherkin scenarios may also need to be updated. Similarly, if the Gherkin scenario changes then so may the step functions and application. For example, if a decision is made to refer to 'savings account' specifically, then the nomenclature throughout the Gherkin scenario, step functions and implementation all need to be manually updated. This work is more difficult than other forms of refactoring because the work must be undertaken across languages (Gherkin and Python). In the next sections we detail related work concerning the mitigation of this maintenance task and then our approach to test code generation.

III. Related Work

Test case generation and maintenance has been recognised as a source redundancy by a number of researchers, requiring the manual translation of user requirements for a system into executable acceptance tests.

Much of the work on test code generation has concerned deriving test suites from formal specifications of system behaviour. Dick and Faivre [16] attempt test-case generation from formal mathematical specifications. They describe a process of detecting individual operations in order to automate them by creating a sequence of operations that handles all the tests needed for a system. Their process used the results of mathematical analysis of the specification to construct a Finite State Automaton (FSA). Tests were scheduled by identifying paths in the FSA. Kim et al [17] suggest generating test cases from state-diagrams in Unified Modelling Language (UML). They generate test-cases that use flow analysis to verify the correctness of the implementation code by checking if newly generated classes have the same control and data flow as it was shown in the UML diagram specification. Ismail et al [18] proposed a technique that uses intelligent searching techniques to identify which previously stored keywords match use-case diagrams generated from textual specifications, and generates test-cases based on those keywords. While these approaches seem promising, they still introduce an additional intermediate step of diagrams, which would still need to be changed if changes would occur to the requirements themselves.

Other test-case generation approaches include property-based testing and crowd-sourcing tests. Property-based testing involves making assumptions about the output of a piece of code based on the input. Tests are run repeatedly with different generated input and their output is validated against the expected output to determine if they were passed successfully or not. Fink et al [19] introduce property-based testing as a new testing approach meant to help developers keep track of common mistakes that occur during the design and implementation stages of the software development process in the form of formal specification that would map to tests and be generic enough to be automated

Crowd-sourcing of tests [20] is a technique used in the software development industry to bring together the knowledge of developers from different teams and different locations with the aim of taking advantage of multiple perspectives and various levels of expertise to improve test coverage by generating new test-cases. While it does show promising results, one potential problem may be that organisations may be reluctant to reveal user stories and scenarios that describe commercially valuable features to a crowd of workers.

Several researchers have investigated the potential for natural language processing to be used for requirements analysis and application code generation. Fatwanto [21] proposed a method of transitioning natural language requirements to a

scenario-based formal language through different stages of parsing, in order to use the scenario representation to potentially generate various other representations. However, the resulting method did not have a clear way of mapping between, natural language elements and formal language concepts. Other research in the field includes using NLP techniques for defect detection in textual requirements, to detect ambiguity or to check if the requirements match a certain template [22], [23].

Groen et al [24] investigated whether there was an actual need for NLP in requirements engineering as opposed to taking a usual manual approach. They performed an expert-based manual analysis of natural language requirements and compared the performance with that of an automated analysis on the same data set. Their results showed that automated requirements analysis using NLP scaled significantly better than manual analysis. This suggests that there is indeed a need to make use of NLP techniques in requirements engineering.

With recent improvements in NLP techniques and a growth in interest from industry, Ferrari [25] outlined the issues that could arise from applying these techniques in industry, such as a need for large data sets for the NLP algorithms to work properly, a need for individuals with domain experience to take part in the process which might be a problem due to confidentiality and the fact that the context differs among companies so a functioning general solution might not exist.

To our knowledge, test code generation directly from natural language has not appeared in the peer-reviewed literature. However, Wang et al [26] explored automatic test case generation from structured formal specifications, with a focus on extracting behavioural information from the specification text in the context of safety critical systems. They achieve this by combining NLP techniques with constraint solving. In order to generate the test cases, they enforce the use of a formal specification language called RUCM. Their research is pursuing a similar goal to the work presented here, however the formal language they propose to use for requirements representation lacks the natural language aspect of BDD scenario descriptions.

In the most closely related previous work, Kamalakar [13] proposed eliminating step implementation entirely by using NLP techniques to analyse BDD user stories, then mapping them to existent implementation code and generating the test implementation by using properties from the project code the user story refers to. There are two limitations to Kamalakar's work. First the approach is limited to projects that have already been implemented, thus not allowing for the full use of a BDD approach. Second, the evaluation was limited to a single case study in which the author made modifications to a software project to assist the code generation mechanism. Thus, to date, there has not been a study of the reliability of test code generation techniques for multiple and/or unseen projects.

IV. Design of behave_nicely

This section describes the design of our approach to test code generation for `behave`. Figure 5 illustrates the desired workflow for the tool. In the approach, user stories and test scenarios are still written and maintained in Gherkin. Similarly, application code is still developed in Python. However, in contrast to the workflow illustrated in Figure 1 for conventional workflows, step functions are generated automatically whenever the BDD test suite is executed.

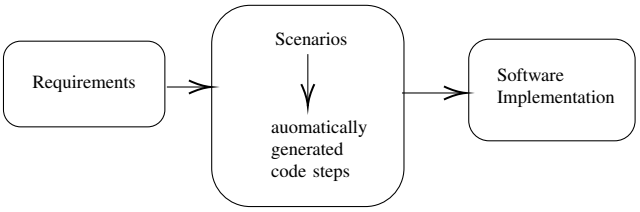

Fig. 5: Desired BDD Workflow

Figure 6 illustrates the overall architecture for behave_nicely. The tool comprises of three main components:

- The *Change Detector*: detects changes made to the project API or the scenario specifications and notifies that the step functions need to be regenerated again. Natural language processing can be computationally expensive, so it is convenient to limit the number of times this task must be performed, particularly if code generation is incorporated into a continuous integration pipeline, in which it is desirable to keep processes fast [27].
- The *Parsing component*: performs part-of-speech tagging and shallow semantic parsing on each statement in the Gherkin scenarios. This component employs a combination part-of-speech (POS) tagging implemented using NLTK [28] and Semantic role labelling (SRL) implemented using AllenNLP [29].
- The *Generating component*: generates step function code using the information from the parser and stores them into a steps.py file. Currently, the generating component targets the `behave` BDD library for the Python programming language.

POS provides syntactical information about the structure of a sentence, whilst SRL provides greater information about the meaning of a sentence. When generating test cases, we take a

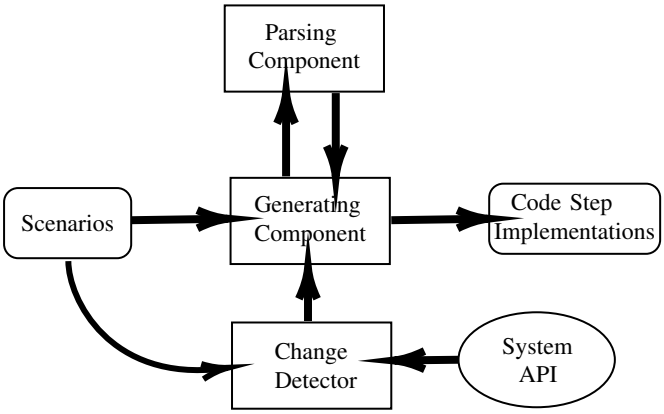

Fig. 6: Overview of the architecture of behave_nicely

different approach for each type of code step - **`Given`**, **`When`** or **`Then`**:

- **`Given`** statements should identify a test *fixture* to be constructed. Therefore, the first noun in a given statement is considered to be a fixture. Subsequent nouns are parsed as arguments to the fixture. In the example given in Figure 2, the 'bank account' is parsed as a fixture and the balance of '0' is parsed as an argument.
- **`When`** statements are expected to perform an action on a fixture by invoking one of its methods. Good software development practice states that methods should be identified by verbs [30]. In the two example **`When`** statements given in Figure 2, the verbs 'deposit' and 'withdraw' indicate the methods to be invoked in the step function. A scenario may comprise more than one fixture, so all the nouns in the scenario are extracted and compared to fixtures already declared in **`Given`** steps. In this case, the 'bank account' is again identified as the fixture to be executed. Finally, any remaining noun phrases are treated as parameters to the method to be executed in the order they are declared in the statement.
- **`Then`** statements are expected to verify that a post condition holds. The fixture to be checked is extracted as for **`When`** statements. Similarly, the property to be checked is the first non-literal (number, quoted string etc.) noun in the statement. Finally the expected value of the property is the last literal (number, quoted string etc.) in the statement.

A complicating factor for behave_nicely is that Gherkin supports the use of parameters and tables of examples inputs and expected outputs. Parameters are indicated by identifiers enclosed in angle brackets, for example `<balance>`. As well as being incorporated into the body of the step function, these parameters must also be declared in the function parameters.

V. Experimental Design

This section presents our experimental design for evaluating behave_nicely. The experimental design comprises the construction of a data set of open source projects that employ BDD using the `behave` framework. Separately, the mechanism for comparing the step functions generated by behave_nicely with the reference functions in the original projects is described.

A. Open Source Project Data Set

Two data sets of open source projects were created. The data sets were created by searching GitHub for projects that used the `behave` framework using the search term "bdd" and "behave". The search was filtered for forks to reduce the possibility that duplicate or similar features would be included in the data set. The top 70 search results were cloned and divided into two data sets as follows.

- A *white box* data set of 20 projects was used to support the development of behave_nicely. When behave_nicely failed to generate the correct step function for a scenario in this data set, the project code was inspected to understand the reason why. Hence, we worked to maximise behave_nicely's support for these projects and we anticipated that final overall performance for this category of project would be high.
- A *black box* data set of 50 open source projects (again taken from GitHub). These projects were used to provide a measure of behave_nicely's performance against unseen Gherkin feature files. Hence, we anticipated that behave_nicely's performance against this data set would be substantially lower than for the white box data set.

All projects were cloned from GitHub between the 30th and 31st of January 2019. Tables I and II in Section VI list the projects used for white box and black box testing, respectively. The two data sets both provide insights as to the feasibility of generating set functions from Gherkin scenarios. On the one hand, black box testing provides an assessment of the reliability of behave_nicely as it is currently implemented. Conversely, undertaking a white box evaluation gives an indication of the likely *future* reliability of this code generation approach, as more examples of variations in uses of Gherkin are incorporated into the tool.

B. Evaluation Using Mutation Testing

The step functions generated by behave_nicely were compared with the implementations found within the open source projects in the data sets. Our goal was to determine whether each generated step function performed the same as the equivalent step function in the original project. Determining whether two programs are functionally equivalent is, in general, undecidable. To address this, an approximate measure of equivalence between step functions was developed using mutation testing [12].

Mutation testing is a technique to measure the effectiveness of test suites in detecting bugs. In the approach, a number of 'mutants' of an application are created using code manipulation tools. Examples of mutations include the replacement of logical and mathematical operators with alternatives, negation of logical expressions and replacement of arguments to function call with null references. The intention is to simulate the introduction of defects to a project. The test suite for the project is then executed against each mutant. Mutants that pass all the tests in the suite are said to be 'survivors', with a high mutant survival rate indicative of a test suite with poor coverage.

Mutation testing was used slightly differently for the purposes of evaluating behave_nicely. The mutation tool Mutmut [31] was used to generate mutants for the target project. The original test suite of Gherkin scenarios and step functions was then used to test the mutant and the scenarios that passed or failed as a result was recorded.

Next, behave_nicely was executed on the Gherkin scenarios to generate new step functions. The generated functions were then inspected and minor inconsistencies such as variable name mismatches were corrected manually. The test suite was then re-run using the step functions generated by behave_nicely and the scenarios that passed or failed were

```
@given('br-init is run with {location}')
def step_impl(context, location):
    context.location_param = None if \
      location == 'no parameters' else location
    project_folder = context.location_param or \
      'features'
    context.project_created_dir = \
      os.path.join(os.getcwd(), project_folder)
```

Fig. 7: Example of non-explicit step function from `behave-restful`.

also recorded. The outcome for each scenario was compared against the outcome for the original project.

The code generation by behave_nicely was considered to be successful for a Gherkin scenario if step functions could be successfully generated for the scenario *and* the outcomes of the execution of the test scenario on the mutated project matched for the original step function implementation and the generated step function (i.e. pass and pass, or fail and fail). Overall results for the comparison are reported in the next section.

VI. Results

A. White Box Projects

Table I summarises the results for the white box testing projects used during the development of behave_nicely. The table lists the project title, URL on GitHub and the number of Gherkin scenarios found in each project.

In addition, the table shows the number of scenarios for which step functions were successfully generated by behave_nicely. As described above, a scenario was considered to be successful if all step functions were generated *and* the same test/fail behaviour was observed for the test scenario as compared to the original scenario when subjected to mutation testing. As can be seen, all scenarios were successfully generated for 16 (80%) of the 20 projects and 74 (82%) of the 91 scenarios that they contained. On further inspection, where behave_nicely could successfully generate step functions, the resulting scenario would also pass the mutation testing evaluation.

behave_nicely was unable to generate step functions for 4 of the 20 white box projects. On inspection, these projects contained step functions that did not conform with the expected uses of the different Gherkin step types, or were too ambiguous to be translated into code. For example, the project `behave-restful` contains the step function shown in Figure 7. The figure shows that much of the necessary functionality for the code step is not explicit in the corresponding Gherkin statement. The trade off between implicit and explicit detail expressed in Gherkin is discussed further in Section VIII.

B. Black Box Projects

Table II summarises the results of the same evaluation procedure for the black box projects. As was anticipated, behave_nicely was able to successfully generate step functions for far fewer projects in the black box data set compared to those used for white box testing. In total, it was possible to generate step functions for 15 (30%) of the 50 projects. Of these, step functions were successfully generated for all scenarios for 13 (26%) projects and partially successful for 2 projects. Overall, behave_nicely was able to successfully generate step functions for 92 (17%) of the 530 scenarios found in the black box data set of projects.

A lightweight review of the black box projects was conducted *after* the evaluation in order to understand factors for the success and failure of step generation. As for the white box projects, it was evident that where features contained explicit descriptions of actions to be undertaken on the system API it was possible for behave_nicely to automatically generate the step function. However, where a Gherkin step abstracted multiple actions undertaken in the step function (i.e. the Gherkin step does not explicitly describe a single action on the application API), behave_nicely failed to generate the code correctly. Separately, one project, `transcend` (47) was discovered after evaluation to be implemented in Portuguese, meaning that the scenarios could not be parsed. The results from the white and black box evaluation are discussed further in Section VIII.

C. Limitations

There are two limitations to the mechanism for evaluation of behave_nicely that we adopted.

First, both the white and black box evaluation is limited to a data set of 70 open source projects drawn from GitHub. The size of the data set, whilst substantial, is still limited relative to the extent of practice of BDD [9]. The size of the evaluation suite was restricted by practical considerations of conducting the experiment. Further, the decision to focus on open source projects may have introduced bias. Many of the software projects in the data set appear to be 'toy' or 'tutorial' projects. Although more than 600 Gherkin scenarios were evaluated with behave_nicely, only 3 of the white box and 10 of the black box projects had test suites containing 10 or more Gherkin scenarios. A focus on open source projects was necessitated by the need for a sizeable set of projects that use BDD in order to undertake an empirical evaluation. However, this focus may have led to the selection of projects that were less complete or less well maintained than projects developed and maintained within commercial settings.

Second, the need to make minor manual adjustments to generated step functions introduced an element of subjectivity into the evaluation. As described, minor adjustments were made where there was a mismatch between the naming conventions within the Gherkin statement and the implementation details in the step function. Adjustments were *not* made if the step function required significant alteration such as the introduction of control structures or other alteration to the function structure. Without the manual adjustments, the performance of behave_nicely would have inevitably been lower, however, the adjustments were necessary in order to assess the extent

TABLE I: List of projects comprising the white box set and mutation testing comparison results. Shaded projects contain at least one unsuccessfully generated scenario.

	Project	URL	Scenarios	Successful Generation
1	algorithms	https://github.com/downquark/algorithms	1	1
2	Banking-BDD	https://github.com/rbob96/Banking-BDD	3	3
3	BDD-Behave-Example	https://github.com/rifferreinert/BDD-Behave-Example	3	3
4	Behave	https://github.com/mmguzman/Behave	4	4
5	behave-calc	https://github.com/AlberTajuelo/behave-calc	2	2
6	behave-demo	https://github.com/rsavalagi/behave-demo	2	0
7	behave-example	https://github.com/j-bennet/behave-example	6	0
8	behave_parallel_demo	https://github.com/vishalm/behave_parallel_demo	15	15
9	behave-restful	https://github.com/behave-restful/behave-restful	6	0
10	behave-testlink	https://github.com/jframos/behave-testlink	3	3
11	behave_web2	https://github.com/timbortnik/behave_web2	3	0
12	ci-jenkins-tox-example	https://github.com/Enforcer/ci-jenkins-tox-example	1	1
13	cucumber-python	https://github.com/dev-lusaja/cucumber-python	5	5
14	DataSet Verification	https://github.com/VONLY-LLC/DataSetVerification	6	6
15	Python Gherkin Demos	https://github.com/AndreasAugustin/Gherkin-Demos-python	5	5
16	Learning Selenium and BDD in python	https://github.com/rajadavidh/learning-selenium-behave-python	2	2
17	PythonBDD	https://github.com/kazune-br/PythonBDD	3	3
18	Python_BehaveBDD	https://github.com/NVKPAVANKUMAR/Python_BehaveBDD	4	4
19	vending-machine	https://github.com/SpenserHardin/vending-machine	13	13
20	Testing Strategy	https://github.com/peter-evolent/python-testing-examples	4	4
			91	**74**

to which step functions could be generated within a realistic context.

VII. Discussion

The results from both the white box and black box evaluations demonstrate that the ability of behave_nicely to successfully generate step functions that are equivalent to the original implementations in the data sets is strongly influenced by three factors:

- The extent to which a step function implementation in the reference project conforms with the expected pattern for the step function type. There were multiple examples of violations of this assumption, particularly in the black box data set. For example, some step functions had not been fully implemented in the reference project, containing a single pass statement, or raising an unimplemented step exception. Table III summarises the number of steps in the black box projects that fell within these categories.
- The extent to which the naming conventions within the Gherkin feature files matched those within the underlying application. For example, if a feature file was discovered to refer to a 'bank account' as a fixture within a `Given` statement, then behave_nicely assumes that a class called BankAccount exists within the project API. However, if the class has been named SavingsAccount, or Account, then behave_nicely will not generate the step function correctly. This problem is perhaps exacerbated by the limited support for cross-language refactoring in many development environments.
- The explicitness of the Gherkin step relative to the corresponding implementation in the step function. For example, projects contained examples of substantial implicit work undertaken in the step function that was not explicitly described in the corresponding Gherkin step. This meant it was not possible to generate step functions in these cases.

All three of these issues limited the ability of behave_nicely to correctly generate step functions, particularly for the black box suite. A consequence of these difficulties was the need to make subjective decisions as to what constituted a 'minor' alterations to generated step functions in order to perform mutation testing comparison.

There appears to be little that can be done about the first issue and its impact on the results for behave_nicely, since in these cases, it seems unlikely that the original step functions in the data sets represent the 'correct' implementation as intended by the authors. It may be argued that these steps should have been excluded from the evaluation, however, this would have had the effect of reducing the 'realism' of the approach to testing and would have required a more detailed review of the black box data set in particular before commencing the evaluation.

The second and third issue in particular raise questions as to the establishment of good practice guidelines for BDD and for describing specifications and scenarios in Gherkin. There is an extensive guidance discussing different aspects of code hygiene for application programming languages, with several texts being very popular on the topic [32]–[34]. However, practical guidance for maintaining clean test code is rather limited [35], and we were unable to discover any research concerning best practice for maintaining acceptance tests in Gherkin. Existing applicable best practice is largely focused on API naming conventions, emphasising that these should reflect the domain of concern for an application, rather than implementation details [32]. However, these do not address cross-language consistency and code generation would be eased if this was encouraged.

The guidance that is available for writing maintainable tests

TABLE II: List of projects comprising the black box data set. Dark shaded rows are for projects that behave_nicely did not generate any scenarios. Light shaded rows are for partially successful projects.

	Project	URL	Scenarios	Successful Generation
1	59_effective	https://github.com/Jyotiranjan767/59_effetctive	2	2
2	behave	https://github.com/stonedMoose/behave	150	0
3	BehavePractice	https://github.com/huyenuet/BehavePractice	89	0
4	Brewerslabng	https://github.com/allena29/brewerslabng	2	2
5	brightness-zoo	https://github.com/banan3k/Intel-Galileo-Mraa	6	0
6	chess	https://github.com/maxtheman/chess	2	2
7	cli-tools	https://github.com/yamaszone/cli-tools	2	0
8	Cloudonyms	https://github.com/MarkAufdencamp/cloudonyms	8	0
9	code_lib	https://github.com/przemekkot/code_lib.git	3	0
10	configure	https://github.com/hotgloupi/configure	4	0
11	demo	https://github.com/7ep/demo	2	2
12	D - I - N - K - Y	https://github.com/kowalcj0/dinky	3	3
13	Docker Start to Finish	https://github.com/machzqcq/docker-for-all	1	1
14	health monger	https://github.com/twonds/healthmonger	4	0
15	House Price Demo	https://github.com/odoko-devops/houseprice-demo	3	0
16	Hyperledger	https://github.com/nikileshsa/hyperledger_pluggable_datastore	4	0
17	KalibroClient	https://github.com/mezuro/kalibro_client_py	1	0
18	liste-della-spesa	https://github.com/slug-it/liste-della-spesa	10	10
19	Malunas	https://github.com/normantas/malunas	2	0
20	ManuTironis	https://github.com/Winnetou/ManuTironis	1	0
21	MMP	https://github.com/mas15/MMP	8	6
22	MP2ASCII	https://github.com/chrdavid/superlists	3	0
23	NYU Devops Class	https://github.com/ilanasufrin/nyu-devops-homework-1	8	8
24	prj_bdd	https://github.com/nzabus/prj_bdd	4	0
25	prog-strat-game	https://github.com/mc706/prog-strat-game	8	0
26	PyBazaarBot	https://github.com/marcoplaisier/PyBazaarBot	12	12
27	pynetlib	https://github.com/migibert/pynetlib	8	0
28	license-generator	https://github.com/walterdolce/python-package-license-generator	6	0
29	python-amq-multi-deploy-example	https://github.com/pcarney8/python-amq-multi-deploy-example	2	0
30	Python_restapi_Demo	https://github.com/Vikramsingh01/python_restapi_demo	4	0
31	python_workspace	https://github.com/mauriciochaves/python_workspace	1	0
32	quic-perf-metrics	https://github.com/kazuho/quic-perf-metrics	12	0
33	RaceAutomationPython	https://github.com/Grigorevskiy/RaceAutomationPython	2	0
34	rdopkg	https://github.com/softwarefactory-project/rdopkg	23	0
35	room5 StudyCase	https://github.com/kmlsim/room5StudyCase	4	4
36	Satpy	https://github.com/pytroll/satpy	4	0
37	shoptest	https://github.com/mihelanjelo/shoptest	5	5
38	smoke_test_webstation	https://github.com/ivanori1/smoke_test_webstation	5	0
39	software_testing	https://github.com/fatalruin/software_testing	3	3
40	Speedwagon	https://github.com/UIUCLibrary/Speedwagon	10	6
41	Sprinkle_Test	https://github.com/singhkalpik/SprinkleTest	2	0
42	suit-test	https://github.com/kate777k/suit-test	1	0
43	superlists	https://github.com/bbvch/brightness-zoo	15	0
44	tdd	https://github.com/jeanmichelem/tdd.git	1	0
45	tinylog	https://github.com/anaplian/tinylog	1	0
46	tradecore	https://github.com/Vojvodic1/tradecore	39	0
47	transcend	https://github.com/WELDISSON/transcend	10	0
48	trtc	https://github.com/pcn/trtc	26	26
49	Veryfay	https://github.com/florinn/veryfay-python	1	0
50	WanderLust-TheTravelApp	https://github.com/PrateekVachher/WanderLust-TheTravelApp	3	0
			530	**92**

in general recommends that they should be both explicit and short [32]. For the Gherkin test scenarios these requirements present a trade-off. If Gherkin scenarios are kept short, this often necessitates abstracting much of the detail of the implementation of the test within the corresponding step function, limiting test explicitness. On the other hand if all the details of the test are maintained within the Gherkin feature, the description of the scenario may become very long, making the purpose of the test harder to understand.

A further limitation of behave_nicely is that the code generation method assumes that step functions target a Python language API. However, some projects employ Gherkin to drive user acceptance testing via other interfaces such as a user interface or a REST API. In these cases, the current implementation of behave_nicely will not correctly generate the step functions. This limitation is more feasible to address directly within the implementation of behave_nicely, since all that is required is an alteration to the expectations of how to use the information gathered during step parsing. The extent to which BDD specifications should support user interface testing and concepts is a subject of some debate amongst practitioners (see North for example [36]), although we are unaware of investigation in the research literature, beyond the survey of Binamungu et al [10].

TABLE III: Summary of code step implementations that behave_nicely could not correctly generate within the black box set of 50 projects.

Category	Count
All Steps reviewed	941
Steps that only contained "pass" in the implementation	44
Given or when steps containing asserts	122
When and Then steps containing fixture constructor calls	21
Steps only containing not implemented exceptions	58

VIII. Conclusions

This paper has described and evaluated behave_nicely, a step function generation tool using natural language processing (NLP). The tool exploits the intended purpose of each of the different types of steps in BDD scenarios to ease the task of translating natural language statements into executable step functions, reducing the need for manual code maintenance. The paper also presents a novel method for evaluating code generation for test suites using mutation testing and uses it to evaluate the efficacy of the described approach. The results from the experiments are promising, suggest that automatic generation of step functions for BDD is feasible and establishes a baseline for future improvements.

There are several directions for further research resulting from the reported limitations of the experimental approach and the discussion of results. Most immediately, further work is needed to investigate the scalability of the code generation technique for larger behaviour driven development specifications. The largest project in our test set consisted of the 150 scenarios in the `behave` project itself, which may not be representative of scale specifications maintained for more larger applications with many different features. However, preliminary work will also need to be undertaken to extend the work of Binamungu et al [10] to understand what constitutes a typical scale behaviour driven development specification in commercial settings. In addition, to support a wider range of projects for evaluation, the set of target platforms and programming languages supported by behave_nicely could be expanded. This would extend the range of projects in the evaluation that could be supported by the approach.

There may also be a variety of approaches possible for increasing the fidelity of the NLP of the Gherkin scenarios. For example, inspection of the test data sets suggests that the range of sentence structures used for the different types of Gherkin statement could be relatively limited. This suggests that it might be possible to create a suite sentence structure templates using a tool such as Spacy [37]. Each template could be tagged with the likely location of information necessary to implement a code step. This suite of sentences could be extendable or even approximated using a machine learning approach.

A more radical approach to employing NLP in BDD would be for the purpose of enforcing or enhancing code hygiene between Gherkin scenarios and step implementations. Static analysis tools such as lint are used by practitioners to enforce program language conventions such as PEP8 for Python. These tools are useful because they can enhance the standardisation of implementation styles amongst different developers within a team or project, with the intention of improving readability and reducing maintenance costs over the long term. An alternative use case for behave_nicely would be to support the identification of inconsistencies between terminology used in Gherkin scenarios and that used in the target application API. Static analysis could also be used to identify where implementation details that have been included in a step function could be transferred to the target application, improving the overall design. As a final step in this direction, behave_nicely could be used to identify opportunities for refactoring of Gherkin scenarios and step functions, an area that remains unaddressed [10]. Further research is needed to investigate how users of BDD would interact with and benefit from this approach.

Behaviour driven development (BDD) represents a trade-off between approaches that enable test cases to be automatically generated from formal specifications at the expense of the ease of communication amongst non-expert stakeholders and the development team; and informal specifications that support ready communication, but lack a tight integration between system specifications and executable test suites. The compromise in BDD requires that the association between individual natural language statements in test scenarios and the corresponding step function that implements them be manually maintained by the development team. The research described in this paper therefore begins the investigation of the extent to which a combination of improved natural language processing and encouragement of consistency and formalisation of language in Gherkin specifications can mitigate the effort required to support this compromise.

References

[1] "Introducing bdd," https://dannorth.net/introducing-bdd/.

[2] A. Carrera, C. A. Iglesias, and M. Garijo, "Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development," *Information Systems Frontiers*, vol. 16, no. 2, pp. 169–182, April 2014.

[3] E. Keogh, "Bdd: A lean toolkit," in *Processings of Lean Software & Systems Conference*, 2010.

[4] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug 2011, pp. 383–387.

[5] "Behave," https://behave.readthedocs.io/en/latest/.

[6] "Jbehave," https://jbehave.org/.

[7] "Cucumber," https://docs.cucumber.io/.

[8] "SpecFlow," https://specflow.org.

[9] CollabNet VersionOne, "13th annual state of agile report," https://www.stateofagile.com, May 2019.

[10] L. P. Binamungu, S. M. Embury, and N. Konstantinou, "Maintaining behaviour driven development specifications: Challenges and opportunities," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 175–184.

[11] "Behave nicely repository," https://gitlab.com/rbob96/behave_nicely.

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[13] S. Kamalakar, "Automatically generating tests from natural language descriptions of software behavior," Master's thesis, Faculty of the Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 2013.

[14] Microsoft Corporation. http://github.com.

[15] K. Beck, *Test Driven Development by Example*, ser. Signature. Addison Wesley, November 2002.

[16] J. Dick and A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications," in *FME '93: Industrial-Strength Formal Methods*, J. C. P. Woodcock and P. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 268–284.

[17] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha, "Test cases generation from uml state diagrams," *IEE Proceedings - Software*, vol. 146, no. 4, pp. 187–192, August 1999.

[18] N. Ismail, R. Ibrahim, and N. Ibrahim, "Automatic generation of test cases from use-case diagram," in *Proceedings of the International Conference on Electrical Engineering and Informatics Institut Teknologi Bandung, Indonesia June 17-19, 2007*, January 2007.

[19] G. Fink and M. Bishop, "Property-based testing: A new approach to testing for assurance," *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 4, pp. 74–80, Jul. 1997.

[20] "Crowdsourced testing," https://www.rainforestqa.com/blog/2016-08-17-what-is-crowdsourced-testing/.

[21] A. Fatwanto, "Software requirements specification analysis using natural language processing technique," in *2013 International Conference on QiR*, June 2013, pp. 105–110.

[22] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer, "Automated checking of conformance to requirements templates using natural language processing," *IEEE Transactions on Software Engineering*, vol. 41, no. 10, pp. 944–968, October 2015.

[23] J. Kocerka, M. Krzeslak, and A. Galuszka, "Analysing quality of textual requirements using natural language processing: A literature review," in *23rd International Conference on Methods & Models in Automation & Robotics, MMAR 2018, Międzyzdroje, Poland, August 27-30, 2018*. IEEE, 2018, pp. 876–880.

[24] E. C. Groen, J. Schowalter, S. Kopczynska, S. Polst, and S. Alvani, "Is there really a need for using NLP to elicit requirements? A benchmarking study to assess scalability of manual analysis," in *Joint Proceedings of REFSQ-2018 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018), Utrecht, The Netherlands, March 19, 2018.*, ser. CEUR Workshop Proceedings, K. Schmid, P. Spoletini, E. B. Charrada, Y. Chisik, F. Dalpiaz, A. Ferrari, P. Forbrig, X. Franch, M. Kirikova, N. H. Madhavji, C. Palomares, J. Ralyté, M. Sabetzadeh, P. Sawyer, D. van der Linden, and A. Zamansky, Eds., vol. 2075. CEUR-WS.org, 2018.

[25] A. Ferrari, "Natural language requirements processing: From research to practice," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, May 2018, pp. 536–537.

[26] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 385–396.

[27] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality*, ser. Addison Wesley Signature Series. New Jersey: Addison Wesley, June 2007.

[28] "Natural language toolkit," https://www.nltk.org/.

[29] "Allennlp," https://allennlp.org/.

[30] T. C. Lethbridge, "Priorities for the education and training of software engineers," *The Journal of Systems and Software*, vol. 53, pp. 53–71, 2000.

[31] "Mutmut," https://mutmut.readthedocs.io/en/latest/.

[32] R. C. Martin, *Clean Code. A Handbook of Agile Software Craftsmanship*, ser. Robert C. Martin Series. Prentice Hall, 2009.

[33] A. Hunt and D. Thomas, *The Pragmatic Programmer*. Addison Wesley, October 1999.

[34] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, June 2004.

[35] A. van Deursen, L. M. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," CWI (Centre for Mathematics and Computer Science) Amsterdam, The Netherlands, Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 2001.

[36] D. North, "Whose domain is it anyway?" https://dannorth.net/2011/01/31/whose-domain-is-it-anyway/, January 2011.

[37] "Spacy," https://spacy.io/.