Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

How to speed up BDD automated acceptance testing for safety-critical systems

Daniel Ryan Degutis

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Yang Wang
Commenced:	August 1, 2017
Completed:	February 1, 2018
CR-Classification:	D.2.4, D.2.5, D.2.8

Abstract

An important aspect of developing safety-critical systems is testing, and in some cases an agile development and testing approach is desirable. To reflect and test safety requirements, a process based on Behavior Driven Development (BDD) is considered in this work. The goal is to have an as efficient as possible process for BDD automated acceptance testing.

The original process for this, used in an earlier experiment, is examined and automatable parts are identified. Based on this, improvements to the process are proposed and implemented. This results in an updated process, that utilizes a newly implemented command line tool written for the purpose of producing test cases. These can then be used for the BDD automated acceptance testing process.

Finally, an evaluation with students BDD acceptance testing a sample system is conducted, to determine the effectiveness of the improved process. The results of the evaluation show benefits in productivity when using the improved process.

Kurzfassung

Beim Entwickeln sicherheitskritischer Systeme ist das Testen ein wichtiger Aspekt. Es kann sinnvoll sein, hier auf agile Methoden wie BDD zurückzugreifen. In dieser Ausarbeitung wird ein Prozess basierend auf BDD betrachtet, der das Testen von Sicherheitsanforderungen erleichtert. Das Ziel ist es, diesen Prozess zu modifizieren um möglichst effizient ein System, mittels BDD Akzeptanztests, testen zu können.

Der ursprünglich betrachtete Prozess stammt aus einem früheren Experiment. Im Rahmen dieser Ausarbeitung werden Verbesserungen gesucht, indem Teile des Prozesses mit Hilfe eines neu entwickelten Kommandozeilen-Werkzeugs automatisiert werden.

Zum Abschluss wird eine Evaluation durchgeführt, bei der Studenten den neuen Prozess anwenden, um ein Mustersystem zu testen. Das Ergebnis der Evaluation zeigt Verbesserungen bei der Produktivität mit Hilfe des neuen Prozesses auf.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AT Acceptance Test. 18

ATDD Acceptance Test Driven Development. 18

BDD Behavior Driven Development. 3

CAST Causal Accident Analysis based on System Theory. 17

SPM System Process Model. 18

SPMV System Process Model Variable. 18

STAMP Systems-Theoretic Accident Model and Processes. 17

STPA Systems-Theoretic Process Analysis. 15

TDD Test Driven Development. 18

UAT User Acceptance Testing. 15

UCA Unsafe Control Action. 15

XSTAMPP eXtensible STAMP Platform As Tool Support for Safety Engineering. 15

1 Introduction

Safety analysis and verification of safety requirements are of the highest importance for safety-critical systems. It is therefore desirable for these processes to be a continual part of the agile development process. This can be difficult, as safety standards and regulations that make up safety requirements are different from other requirements. They tend to define constraints for system behavior rather than solely typical functional requirements for singular units in a system. Development, especially testing aspects, should take these differences into account.

In this case, an approach like Systems-Theoretic Process Analysis (STPA) to describe a system and produce good safety requirements is useful. Many accidents in safety-critical systems are caused by Unsafe Control Actions (UCAs), rather than single component failures. Modeling system processes and UCAs is possible with the open-source tool eXtensible STAMP Platform As Tool Support for Safety Engineering (XSTAMPP).

As UCAs are unsafe system behavior, testing system behavior automatically is a suitable testing method for safety requirements.

Automated acceptance testing can be accomplished by applying BDD techniques and tools, as they show acceptable results for testing safety requirements, according to an experiment performed in the EST 17 lecture at the University of Stuttgart (section 6.1).

However, the experiment also showed that writing BDD scenarios and test cases for automated acceptance testing does not offer enough notable benefits in terms of speed over manual User Acceptance Testing (UAT).

The purpose of this thesis is therefore to improve upon the current process of writing BDD acceptance tests for safety-critical systems, by an approach that automates as much as possible of that process.

The concepts and tools mentioned and required are explained in chapter 2.

1.1 Motivation

The goal is to integrate safety requirements testing into the agile development process in a more efficient manner.

1.2 Problem Statement

Currently BDD acceptance tests for safety requirements are written throughout a large number of manual steps. Hence, the current process should be improved by automating certain repetitive tasks. For this the current process must be examined, automatable steps identified and a solution implemented. In chapter 4 improvements are proposed and in chapter 5 the improvements are implemented.

1.3 Research Objective

The research objective is to determine whether there is an increase in productivity with the improvements made to the process of writing BDD acceptance tests. For this, an evaluation will be conducted with a group of students where the improved process is used to BDD acceptance test a sample system. To evaluate changes in productivity, the results are then compared to the results of the EST 2017 agile experiment (section 6.1) that used the old process. Details are discussed in chapter 6.

Structure

This thesis is structured in the following manner:

Chapter 2 – Background: The fundamentals for discussing BDD automated acceptance testing for safety requirements are explained.

Chapter 3 – Related Work: Some related work is introduced.

Chapter 4 – Concept: The current BDD safety requirements testing process and the planned improvements to it are described in detail.

Chapter 5 – Implementation: The improvements to the BDD safety requirements testing process are implemented.

Chapter 6 – Evaluation: An Evaluation to examine the effectiveness of the improvements is conducted.

Chapter 7 – Conclusion

2 Background

The prerequisite concepts and tools relevant to understand and discuss BDD automated safety requirements testing are introduced. BDD, as well as the BDD testing tool JBehave [JBW17] are heavily relied on throughout this work.

2.1 Safety-critical Systems

Safety is an emergent property of a system and depends on context and environment [LecQSW17]. A safe system is one that does not endanger its environment, thus, it should not reach failure states that are harmful to its environment. This is in contrast to security, where the main question is whether the environment can endanger the system.

A safety-critical system is one where safety is of the highest priority, thus, where a harmful failure state may result in loss of life or other significant damage to its environment.

Safety analysis and testing methods are therefore very important, and the requirements for system safety should be based on desired and undesired behavior of the entire system.

Since individual components of a system are not inherently safe or unsafe, but depend on interaction and context, a way to consider the system as a whole is required. Systems-Theoretic Accident Model and Processes (STAMP)/STPA (section 2.2) is such an approach based on system theory.

2.2 STPA with STAMP and XSTAMPP

STAMP [LecQSW17] is an accident model, which views the system it is modeling in its entirety. The system theoretic approach of STAMP allows to model complex relationships between system components.

STPA and Causal Accident Analysis based on System Theory (CAST) are the main methods of STAMP, where CAST is used for investigating the causes of an accident. STPA tries to discover possible hazards in a system, and as a result of the STPA process refined safety requirements, as well as other related artifacts, are produced. Safety, as an emergent system property, can therefore be considered within STPA.

STAMPs perspective on safety is that of a control problem. This means that it models accidents as failures of component interactions due to unsafe control actions (UCAs). They are violations of safety constraints.

The main idea is that controllers control processes via control actions according to an internal process model (System Process Model (SPM)). It receives feedback from the controlled process and determines the next control action based on that feedback and its SPM. The SPM includes process variables (System Process Model Variable (SPMV)) and their possible values.

XSTAMPP [Abdulkhaleq15] is an open-source safety engineering software written in Java. With XSTAMPP it is possible to model system processes using STAMP methodologies.

Resulting SPMs, which model system behavior, provide a guideline for creating acceptance tests and define safety requirements.

For BDD safety requirements testing according to the scheme used in section 4.1, the ability to define SPMVs and possible values within the SPM, as well as UCAs is of interest.

2.3 Agile Testing

In a traditional, phased software development approach, testing activities occur at the end of the development. One of the problems with such an approach is that testing time gets shortened when coding takes longer than expected [CG08]. Additionally, tests might be more difficult to write and when problems are discovered, fixing them might just become too costly.

As an alternative, an agile approach is iterative and incremental. What an iteration looks like might be different between projects and teams. Each new feature is tested as soon as it is finished, which means coding can not really get ahead of testing, as both activities need to be fulfilled for the feature to be completed [CG08].

This is where agile development methods such as Test Driven Development (TDD) come in. In the TDD process tests are written first, made to fail and then the required functionality is implemented in a minimal way to make the tests pass [TDD17]. Functional test cases, ideally based on real-world working examples, are written before any coding begins [CG08].

The idea is that testing happens as early as possible, consequently allowing it to drive design and coding. Moreover, errors get caught a lot earlier in comparison to traditional approaches for testing.

An Acceptance Test (AT) is a formal and often technical description of the behavior of a software [AT17]. Acceptance tests can also be used for TDD, as opposed to the usual component or functional tests, resulting in Acceptance Test Driven Development (ATDD) [ATDD17]. This kind of test is separate from a UAT, in which real-world users test the software in a real-world scenario.

Agile testing is often realized through the use of continuous integration with the support of automation and testing tools.

2.4 BDD and JBehave

BDD [BDD17] is a collection of agile development methods that focus on software behavior. It is based on TDD and ATDD (section 2.3).

BDD also follows a TDD style process. The difference to TDD lies in the fact that tests de-emphasize implementation details. They focus more on testing system behavior rather than single components. This also means that tests are less susceptible to frequent change when implementation changes.

BDD is suitable for testing safety requirements since they, as an emergent system property, tend to describe system behavior.

BDD testing tools provide a structured notation based on natural language that improves communication between all stakeholders. This notation can specify desired behaviors, in this case safety requirements. A behavior specification follows the *given, when, then* template:

- Given: an initial context is defined, this is a set of preconditions or inputs.
- When: some event or action occurs within the context.
- Then: an expected outcome is defined.

Such a test, also called scenario, can be used to test for behavioral requirements. Step definitions that map to such a scenario must then be written. They are the actual code of the test.

Scenarios are grouped into stories which represent some core functionality of the system, whereas a scenario represents an example of the system behavior. This is also called *specification by example*.

Automated acceptance criteria testing via a set of scenarios, can be accomplished using BDD test automation tools [Sma14a] such as JBehave [JBW17] or Cucumber (section 3.2). These automated tests are considered executable specifications and can be used as both acceptance and regression tests [Sma14b]. JBehave features integrations for, among others, Eclipse, IntelliJ and Maven [JBW17]. When using JBehave to support the BDD process, there are some steps to follow:

- Behavior specifications are written in *.story* files according to the established *given, when, then* template as scenarios.
- Java step definitions are created that map to the *stories*. They are the code that is executed when JBehave runs scenarios.

- Creation of a test suite or multiple test suites for use with a testing framework, such as JUnit.

The sample scenario (Listing 2.1)

```
Given a countdown counter (two step) that has a value of 20
When the countdown counter is decreased
Then the countdown counter value is 18
```

Listing 2.1: Sample Scenario

and its corresponding Java mapping (Listing 2.2), the step definitions,

```
public class DecreaseCountdownSteps {

   private int countdownCounter;

   @Given("a countdown counter (two step) that has a value of $value")
   public void countdownCounterHasValue(int value) {
      countdownCounter = value;
   }

   @When("the countdown counter is decreased")
   public void decreaseCountDownCounter() {
      countdownCounter -= 2;
   }

   @Then("the countdown counter value is $value")
   public void theValueOfTheCounterMustBe1Greater(int value) {
      assertTrue(value == countdownCounter);
   }

}
```

Listing 2.2: Sample Step Definitions for the Scenario

show that JBehave maps the *given, when, then* steps in the scenario to the corresponding Java methods with the help of the *@Given, @When, @Then* annotations. The values in the scenario substitute for the marked parameters in the Java mapping. This demonstrates the *Specification-by-Example* style of BDD, where the scenarios specify the behavior by sample values.

While the current BDD testing tools are suited for automated acceptance testing, the tests for safety requirements must still be written manually for the most part as BDD *scenarios*.

To improve upon the BDD automated acceptance testing process, existing assets should be used. For example, a subset of BDD acceptance tests could be automatically generated based on existing SPMs.

3 Related Work

Some related work that is not directly relied on or part of this work, is briefly introduced.

3.1 FMEA

FMEA or Failure Modes and Effects Analysis is a system safety analysis technique [PPG04]. During FMEA, lists of component failure modes are compiled and attempts are made to find the effects of the failure modes on the system. Simple diagrams are used in understanding how component failures affect the entire system [PPG04].

FMEA can be regarded as less effective, considering safety when compared to STAMP/STPA, as FMEA has difficulties handling modern and complex systems [LecQSW17].

3.2 Cucumber

Cucumber [CC17] is a testing tool that enables BDD (section 2.4) style automated testing. It features its own language, Gherkin, to write behavior examples that act as specification, mostly in plain English. Gherkin is supposed to improve communication and collaboration between all stakeholders. Then Cucumber is used to run the examples as automated acceptance tests and reports back the results [CC17].

Cucumber supports many different target languages for the automated acceptance tests as well as integrations for different software platforms.

As an alternative to JBehave, it could also be used for this work, however, due to prior use of JBehave in the EST 2017 agile experiment (section 6.1), JBehave will be prioritized.

3.3 BDD Security - Continuum

BDD-Security is an open source automated security testing framework that uses the BDD Gherkin syntax to specify security requirements. It is developed by Continuum Security [Conti18].

The goal is to make security testing part of the development cycle and ready to run on continuous integration servers, as well as enabling the other advantages of BDD. This

means security requirements are defined in one place, and can be tested and automated in typical BDD manner.

BDD-Security runs tests against a deployed application over the network. It does not require access to application source code. This means that an application under test can by written in any language and framework, as long as it supports HTTP/S [Conti18].

BDD-Security supports integrated reporting, so for example reports can be displayed in HTML format via the Jenkins Cucumber reporting plugin when running on a continuous integration server [Conti18].

4 Concept

When testing safety requirements in safety critical systems, automated acceptance tests are often desired. BDD is a suitable approach integrating and utilizing them during the agile development and testing process (section 2.4). A safety critical system under test is assumed to have the artifacts SPM, SPMVs, possible values as well as UCAs available. The SPMVs correspond to real process variables in the system under test. These artifacts can be created with the STAMP hazard analysis method, STPA (section 2.2) and they can be used to write the BDD acceptance tests. The tests should find the occurrence of UCAs, by testing SPMVs value combinations with the software system's functions.

However, the speed of writing BDD acceptance tests is too slow, as per the results of a student experiment conducted at the University of Stuttgart (section 6.1). Based on this, the main goal of this work is to speed up writing BDD automated acceptance tests. The BDD testing tool considered is JBehave.

4.1 Current Workflow: Writing Automated BDD Acceptance Tests

Currently, when writing automated BDD acceptance tests, there are two major steps, *Step 1* and *Step 2*.

In *Step 1* the BDD scenarios are written, which can later be used to acceptance-test the real software system. The scenarios are based on information of UCAs and SPMs.

In *Step 2* the Java test cases that map to the scenarios written in *Step 1* are created. For this, step definitions stubs must be written that match the scenario steps. Then the actual test code, which calls the software under test, is written into the stubs.

In an agile process, we repeat *Step 1* and *Step 2* for each iteration for a selection of UCAs.

The end goal is to produce a directory structure (for an example see figure 4.1) that contains the BDD scenarios as *stories*, the Java test cases (step definitions) and the test suites. Such a structure can then be directly used in a software system for testing with JBehave.

Step 1 of the current workflow considers only a subset of all scenarios with the following format:

- Given: A list of all SPMVs from the SPM, excluding one, set up the context.

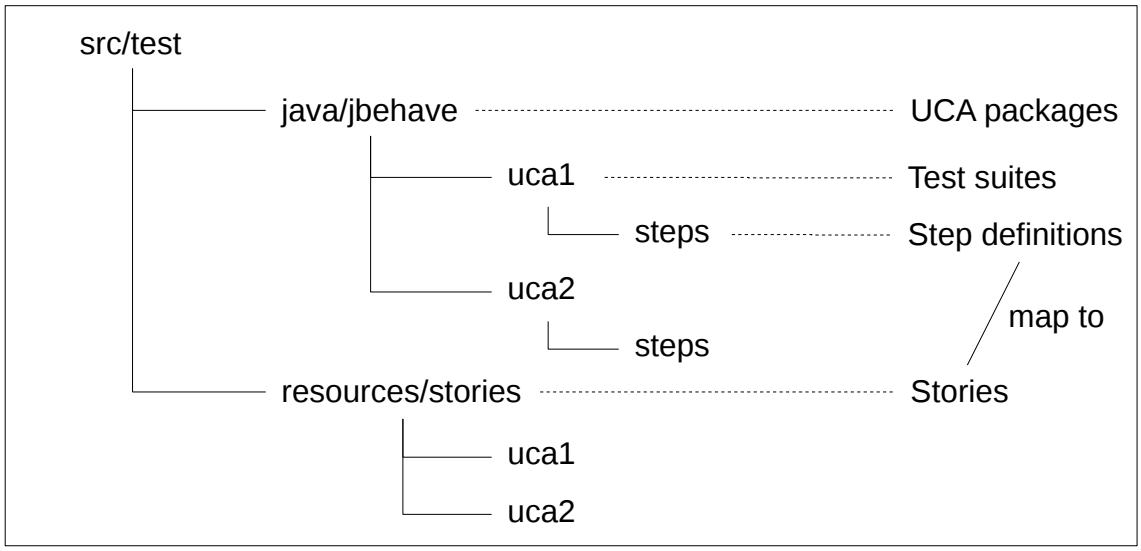

Figure 4.1: Target Directory Structure

- When: A single excluded SPMV from the list, that could causes a UCA to happen, as well as some related actions.
- Then: A UCA occurs.

Only scenarios of this type are considered in this work. By isolating a single SPMV for every scenario, an attempt is made to find variable-value combinations that cause UCAs to happen. This strategy and format of writing unsafe scenarios to discover UCAs was originally introduced and used for the EST 2017 agile experiment (section 6.1).

Such a scenario is created for each UCA, excluded SPMV, as well as possible value for every SPMV. The *Step 1* process can be seen in figure 4.2, it produces BDD scenarios according to the considered scheme. The information required for *Step 1*, UCAs and SPMVs, can be manually supplied or exported from XSTAMPP (section 2.2).

In *Step 2* the Java test cases with the step definitions are created and executed one after another to detect conditions that would cause UCAs. The pool of BDD scenarios from *Step 1* is used here. For each unsafe scenario a corresponding Java test case is created and executed. Depending on the result, the process proceeds differently:

- The test case passes: A UCA was found, the process continues with the next unsafe scenario.
- The test case fails: The test case is either rewritten until it passes or the unsafe scenario is adjusted or removed.

The *Step 2* process is illustrated in figure 4.3.

In more detail the workflow of *Step 1* and *Step 2* may look like this for a Maven project:

- *Step 1* starts.

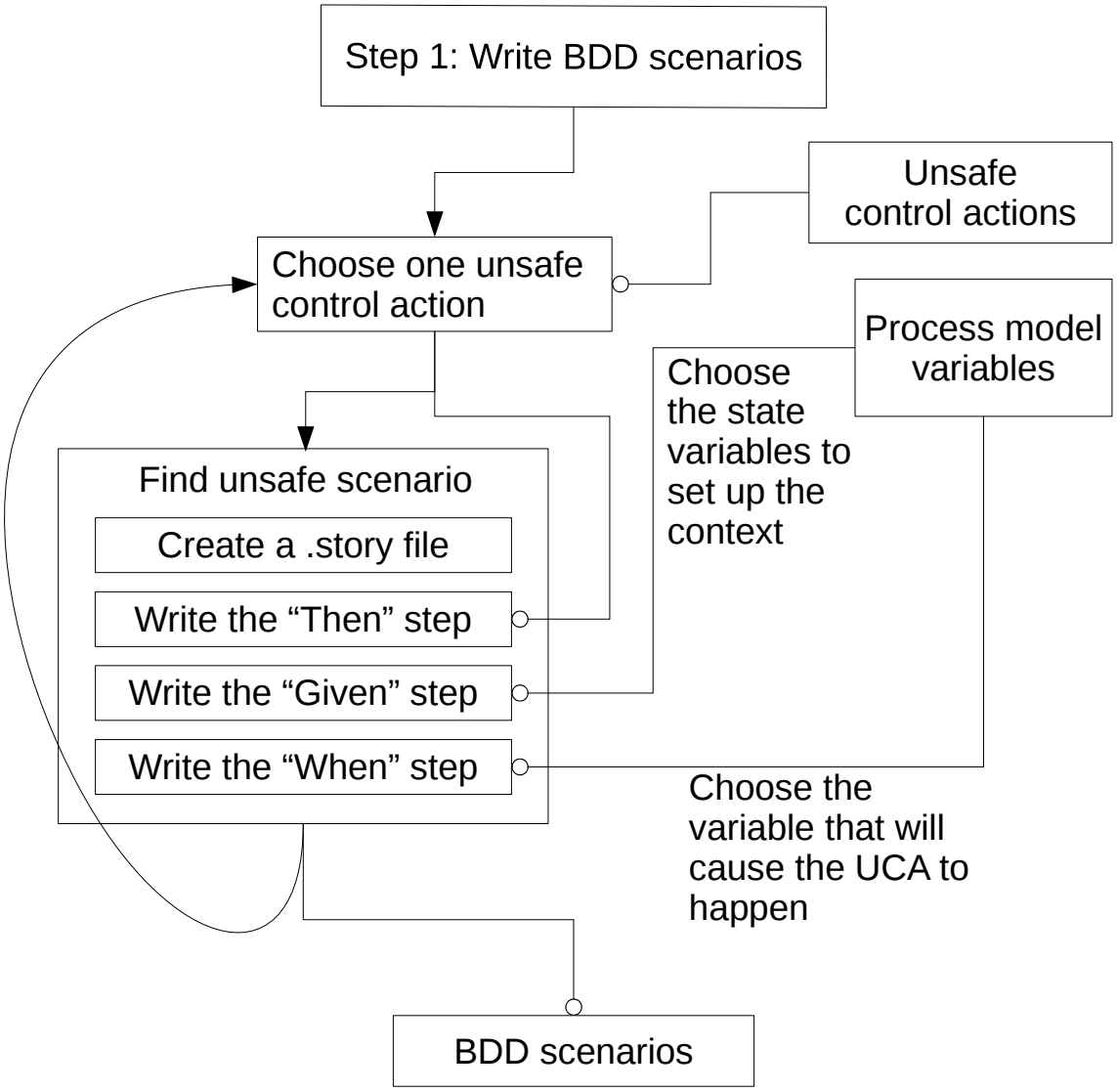

Figure 4.2: Step 1 in the Current Workflow

- Creating a folder for each UCA, for example *uca1*, in the *src/test/resources/stories* folder.
- Writing one or more scenarios, in a new file, for example *the_uca_that_occurs.story*, into the folder *uca1*, according to the scheme described for *Step 1*.
- Writing more scenarios in separate *.story* files.
- *Step 2* starts.
- Creating a new test suite for executing the Java test cases with JUnit for each *.story* in the *src/test/java/jbehav/* folder.
  A sample test suite could be named *TheUcaThatOccurs.java*, derived by removing '_' from the *.story* file's name, camel casing it and capitalizing the first letter.

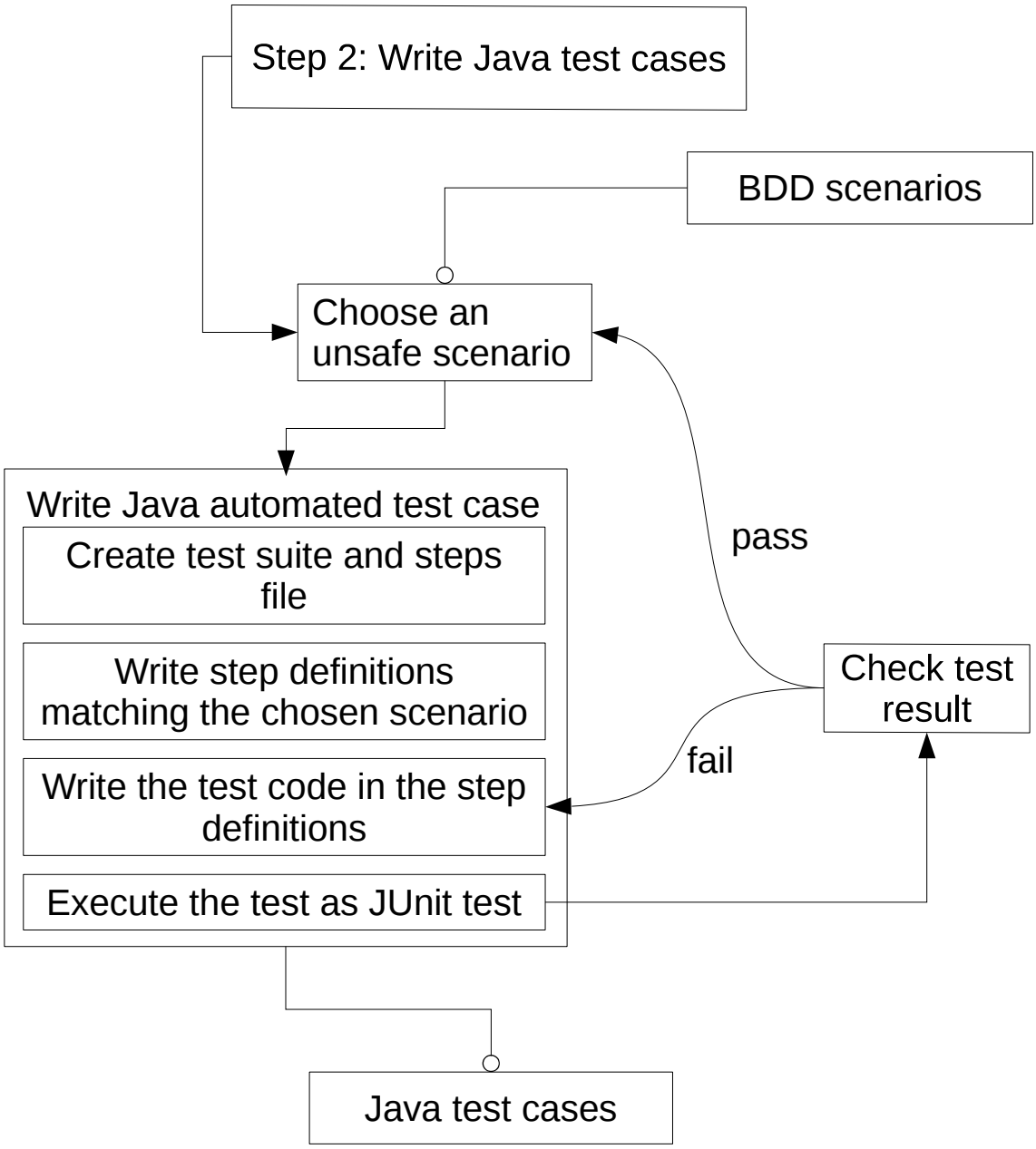

Figure 4.3: Step 2 in the Current Workflow

- Writing a new Java test case for a *.story* file into the folder *src/test/-java/.../jbehave/steps*, for example *Uca1Steps.java*.
  - The Java file name is derived by capitalizing the first letter of the folder name of the UCA's *.story* files, then adding 'Steps' as suffix to it.
  - The location is in the package or a subpackage of the test suites.
  - The Java test cases contain the step definitions that map to the scenario's steps.
  - The actual test code is in the step definitions.
- The test is executed with JUnit and the test outcome is observed.
- If the test passes, the next Java test case for the next scenario is written.
- If the test fails, the code is modified until the test passes or the scenario dropped.

All the files are created manually or copied from existing files and adjusted accordingly. They are named in line with a defined scheme and placed at locations, also conforming to a defined scheme. This is for test management primarily due to the large quantity of possible scenarios. There are also some technicalities involved, such as that steps written in the *.story* files must be globally unique and directly match the step tags in the Java step definitions.

Looking at all the substeps involved, it is clear, that writing these types of scenarios and test cases, as well as manually naming and placing them, is a tedious process.

This workflow was used in the the EST 17 lecture agile experiment (section 6.1).

4.2 Improvements to the BDD Testing Process

Looking at *Step 1*, there is potential for automation. As the type of BDD scenarios considered in this work depends directly on a set of UCAs, SPMVs and their values, it is possible to automate scenario generation. This eliminates large parts of *Step 1*. *Step 1* can be reduced:

- Select some UCAs.
- Generate unsafe scenarios for the selected UCAs with *tests4jbehave*, a conceived tool for this purpose.
- Remove unsafe scenarios that are not required or do not make sense.

Step 2 can be partially automated by automatically generating the Java test cases and their step definition stubs, as this information is available. It is however not possible to automate *Step 2* entirely, as the actual test code, that calls the software system under test must still be manually written.

For the identified areas of improvement, the tool *tests4jbehave* is written. It automates as much as possible of the workflow, thereby improving the speed of the largely manual process of writing BDD safety requirements tests. *tests4jbehave* is designed as command line tool.

In detail for the current manual workflow, the following parts will be automated:

Directory structure A directory structure of *stories*, Java step definitions and test suites is automatically generated and it can be used directly by JBehave and the tested software. All generated files are automatically named according to a sensible scheme and placed at appropriate places in an output directory structure.

Scenarios Scenarios are automatically generated according to the format:

- Given: a list of SPMVs excluding one.
- When: the excluded SPMV from the list that might cause a UCA to occur.
- Then: a UCA occurs.

All possible scenarios of this format are generated, given a list of UCAs and a list of SPMVs. The scenario format attempts to find what combinations of SPMVs values and what SPMV causes a UCA to happen. This set of scenarios was also used in the EST 2017 agile experiment (section 6.1). Scenarios are written to one *.story* file each, where each *.story* file's name is unique and also describes the contained scenario. The file names contain the *stories' then* UCA and *when* SPMV as well as a unique identifier. Naming and location conventions are followed (chapter 5).

Examples For different combinations of possible SPMV values, a scenario according to (paragraph 4.2) could be tested, to discover which value combinations trigger a UCA. This is where the BDD technique *specification-by-example* is applied. With JBehave, the *Examples Table* [Sma14c] feature is utilized to hold the possible value combinations for the SPMVs for a scenario.

tests4jbehave can generate *examples* for each scenario according to some strategy. One strategy for generating examples is to generate all possible combinations of values.

As alternative to this, the pairwise strategy is available. Pairwise values, or 'all-pairs' are guaranteed to be a set of value combinations where every single value is paired with every other single value. The total number of tests generated in this way is a lot less compared to exhaustive testing. Pairwise testing can be a relatively effective alternative in certain situations, due to the observation that most faults are caused by the interaction of a small number of variables [PWT17].

Other than that, it should be possible to choose not to generate any examples by setting them manually or forgoing this option. The strategy to generate examples can be set with the *-e* command line option (chapter 5).

Java step definitions and test suites The Java step definition stubs corresponding to each scenario are automatically generated. It is possible to include additional data such as imports and some declarations in the Java test cases. While nothing can be done to generate Java method bodies, at least the parameters in the step definition's declaration can be generated, according to a sensible variable type. Such a SPMV type mapping can be supplied as *.csv* file with the appropriate command line option (chapter 5).

The Java test suites necessary for execution as JUnit test by JBehave are generated as well.

All generated files follow naming and location conventions for consistent use with JBehave. Package names and subdirectories to place all these generated Java files can be further specified with the appropriate command line options (chapter 5).

Usage of *tests4jbehave* in the Improved Workflow The Java command line tool *tests4jbehave*'s intended usage is depicted in figure 4.4.

The required inputs, the list of UCAs and the list of SPMVs, can be either manually created and supplied by the user or exported from existing XSTAMPP projects. Their input formats are specified in section 5.2.

First, the *tests4jbehave* Java sources are built with Maven.

Then the resulting *.jar* artifact with included dependencies is called from the command line with the desired arguments. The arguments are the required inputs of UCA list, SPMV and possible values list as well as SPMV to Java data type mapping. They are the minimum arguments for scenario generation and immediate usable output. If the variable types mapping is not supplied, a sensible default is chosen. The other arguments shown in figure 4.4 are optional and specify what the output directory structure should look like. *tests4jbehave* offers a number of other options. If they are not specified, sensible defaults are chosen. The available options are listed in chapter 5.

The output is a directory structure with generated *stories*, Java step definitions and test suite, ready to be used in an existing Java project and executed with JBehave.

The workflow is now changed. We consider for one iteration in an agile development process:

- The directory structure for JBehave is generated for a chosen subset of UCA for that iteration.
  It contains the unsafe scenarios, Java step definition stubs and test suites.
- An unsafe scenario is chosen and the step definitions are written.

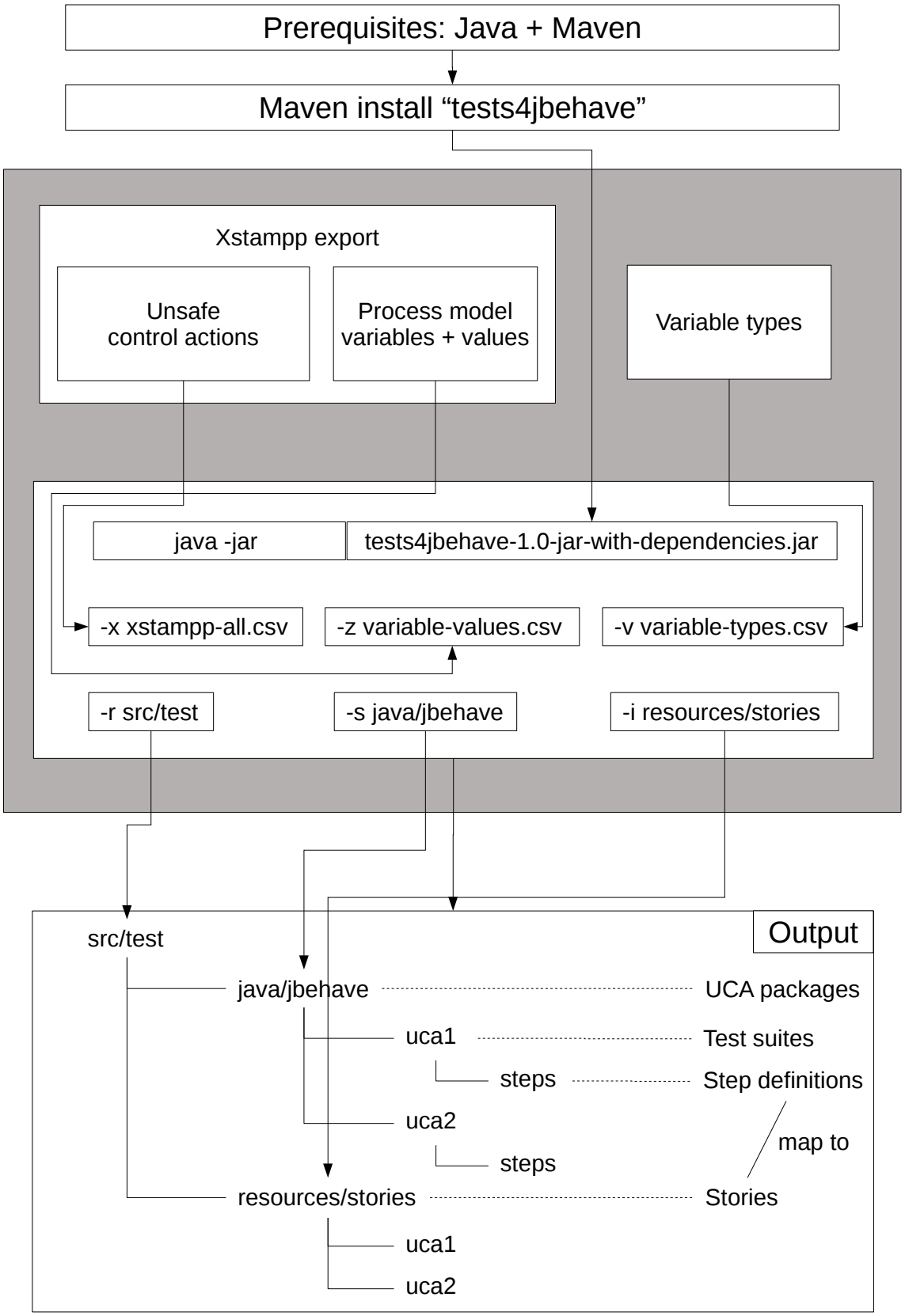

Figure 4.4: tests4jbehave Setup and Usage

- The test case is executed as JUnit test. If the test
  - passes: An unsafe scenario is found. The process continues by writing more step definitions for other test cases and potential unsafe scenarios.
  - fails: The goal is to make the test case pass by rewriting step definitions or removing undesired *examples* to find unsafe scenarios for this test case.
    Alternatively, the test case and scenario is deemed unsuitable and is deleted.
- The next unsafe scenario is chosen and the corresponding test case is written and tested with JUnit.
  This is repeated for all required unsafe scenarios.

This process is illustrated in figure 4.5.

A more detailed sample application can be found in section 5.4.

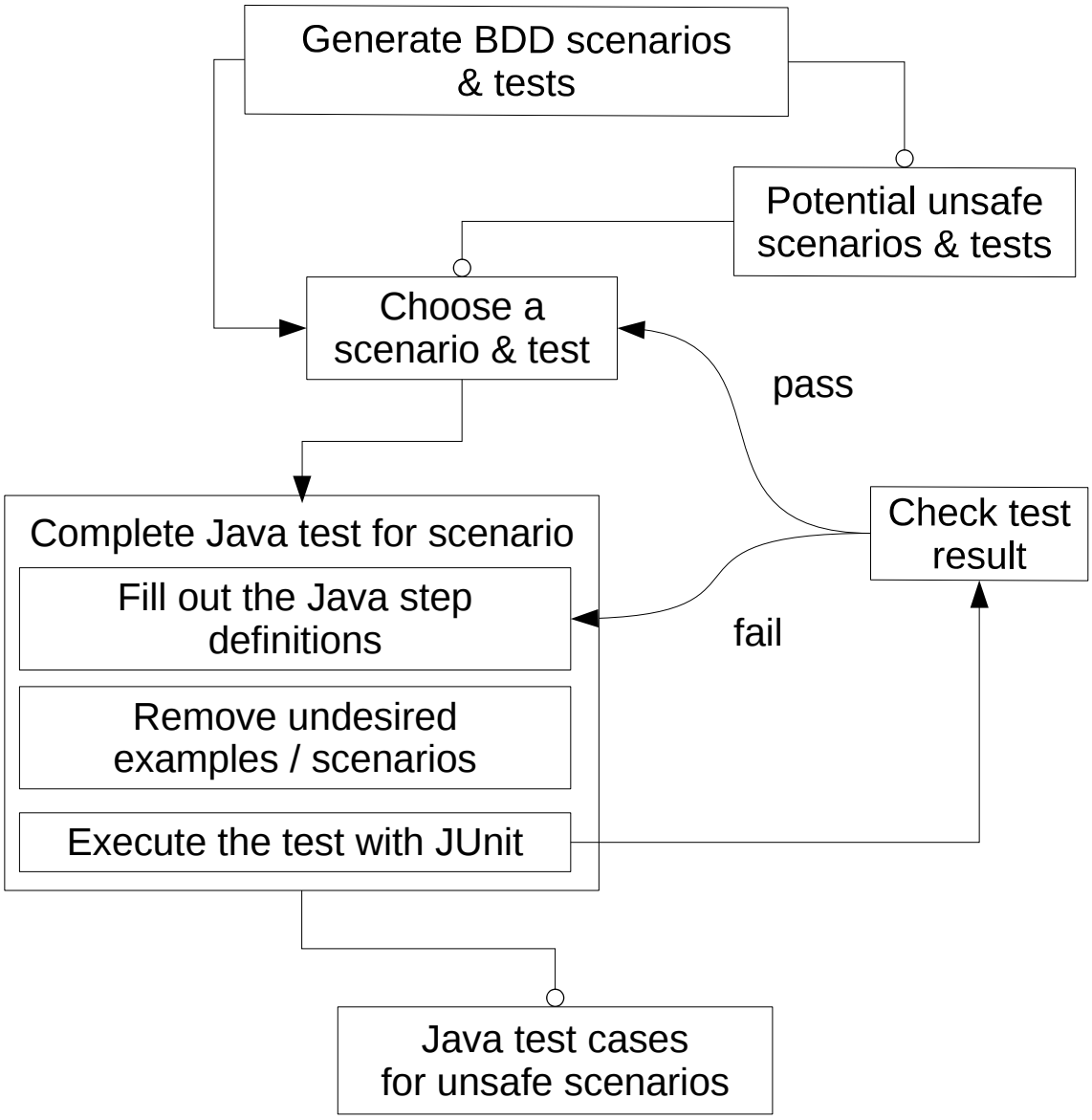

Figure 4.5: New Workflow

5 Implementation

The improvements discussed in chapter 4 are implemented as the *tests4jbehave* command line tool, written in Java. The source code written and build instructions are supplied in the *implementation* folder alongside the *.pdf* version of this document.

The main functionality is the generation and output of a directory structure containing *.story* files with the generated test scenarios and value combinations (examples), the corresponding Java step definition stubs, as well as the Java test suites to launch all tests.

The resulting artifacts are intended for usage with the BDD test automation tool JBehave and Thucydides [Thucy17], now known as Serenity [Seren17] for reporting. JBehave features, such as *Examples* tables are relied on by the generated test cases.

A number of configuration options are available for *tests4jbehave* as command line options:

Option	Description
-c	The format the generated scenarios should have. Currently only the default one (section 4.2) is available.
-d	The *.csv* delimiter to assume for the input *.csv* files.
-e	The type of examples that should be generated. Either all possible combinations, pairwise combinations or no examples.
-f	The file encoding assumed for all files used.
-g	The default variable type to use in Java step definitions, if no appropriate type can be found.
-h	Show a help text describing all options.
-i	The subdirectory for *.story* files.
-j	The path to a file for imports to include in generated tests. Will be copied as-is into each Java file.
-k	The path to a file for declarations to include in generated tests. Will be copied as-is into each Java file.
-l	The line separator to use for all files.
-m	Manually specify the initial scenario id in case of conflicts when resources are reused unchanged in subsequent executions.
-n	Manually specify the initial step id in case of conflicts when resources are reused unchanged in subsequent executions.
-p	The package name for the generated test suites.

-r	The root directory of the output directory structure.
-s	The test suite subdirectory in the output directory structure.
-u	List of UCA ids to specify the subset of ucas to use with matching ids (alphanumeric) as *.csv* file.
-v	The list of UCAs as *.csv* file.
-x	The SPMV to Java data type mapping as *.csv* file.
-z	The SPMVs and possible values list as *.csv* file.

Table 5.1: tests4jbehave Command Line Options

These options and their default values can be viewed when calling *tests4jbehave* with the *-h* argument.

Not all of these options are necessary though, as for useful output the required inputs are:

- A list of UCAs as *.csv* file.
- A list of SPMVs and their possible values as *.csv* file.
- A SPMV to Java data type mapping as *.csv* file.

While the list of UCAs can be exported from XSTAMPP, the SPMV to Java data type mapping must be manually created and supplied by the user, since XSTAMPP simply does not have this information. If the mapping is not supplied, a sensible default value is chosen.

The list of SPMVs and their possible values is currently not available as XSTAMPP export. However, XSTAMPP has access to this information. A possibility is to add an appropriate export option to XSTAMPP.

5.1 Extending XSTAMPP

After examining the XSTAMPP source code, the necessary steps to implement the missing export functionality become apparent:

- A new class "ProcessValuesWizard" is added to the "xstampp.astpa.wizards.stepData" package.
- An entry for "ProcessValuesWizard" is added to the "xstampp.astpa/plugin.xml", so that it is registered and available to the user.
- "StpaCSVExport" in the "xstampp.astpa.util.jobs" package is modified so that a new method "writeProcessModelValues" is called when the "ProcessValuesWizard" is selected according to "ICSVExportConstants" by the user for exporting data from XSTAMPP.

- "writeProcessModelValues" takes the "DataModelController" and leverages "getValuesTOVariables" to retrieve the possible values for each SPMV and writes them in a *.csv* file.

5.2 Input File Format

The larger inputs are *.csv* files. All *.csv* files must use the same *.csv* delimiter. The expected file formats for the respective inputs are:

UCAs list The UCAs list can be potentially supplied from XSTAMPP or manually created. It is supplied with the *-v* command line option. The only requirement to be understood by *tests4jbehave* is that the *.csv* file must contain a section as follows:

- The section has a row where the second field has the contents "Unsafe Control Actions".
- All second fields beneath this row are now taken as one UCA each. It is recommended to use only alphanumeric characters.
- All first fields in the row have a globally unique lowercase alphanumeric identifier.
- The section is terminated by a row with only empty fields.

SPMV to Java data type mapping This *.csv* file is very simple and must be manually created. It is supplied with the *-x* command line option. The format is:

- The first row has two fields: "column" and "type".
- All rows after the first row have two fields, one for the variable and one for the type.
- The mapping assigns one SPMV named "column" the type "type".

SPMV and possible values This *.csv* can be either manually created or exported from XSTAMPP. It is supplied with the *-z* command line option. It has two rows at the beginning that are reserved for XSTAMPP information. Their values are not considered in any way. After that:

- In the third row each subsequent field is one SPMV. It is recommended to use lowercase alphanumeric characters.
- In the following rows, each field's content is a possible value for the SPMV above it in the third row.

UCA id list A list of unique lowercase identifiers separated with a *.csv* delimiter or line break. If not supplied, *tests4jbehave* will generate test cases for all UCAs specified with the *-v* command line option. If supplied, only test cases for the UCAs from the UCA list are generated, of which the id matches one of the ids from this UCA id list.

5.3 Limitations

Due to constraints by Thucydides and JBehave for test suite to *.story* and step matching the following limitations to inputs for *tests4jbehave* should be respected:

- SPMVs names must be lowercase.
- UCAs are lowercase and should be limited to alphanumeric characters.
- In any input there must not be an underscore followed by a number.
- There must not be underscores directly after an underscore.
- All UCA ids consist only of lowercase alphanumeric characters and should be globally unique.

If *tests4jbehave* encounters such a violation, it is removed or replaced. Due to this, in certain cases, name collisions between different UCAs, SPMVs or UCAs ids may occur. For example, if UCAs are identical except for a violation of the limitations:
`this uca is_fine` and `this uca is_1fine` will both be converted to the *.story* file name `this_uca_is_fine`. As such, only one of the two will be in the final output, as JBehave will otherwise encounter a conflict. It is therefore best practice to stick to the limitations described above.

5.4 Sample Usage Details

Examining the improvements (section 4.2), *Step 1* (figure 4.2) was changed considerably. For *Step 2* (figure 4.3) only the actual test code must be written into the generated step definition stubs in the Java test cases.

A sample usage scenario will serve to illustrate the potential viability of the implemented improvements, using the improved workflow. The workflow is executed with the sample system used in the EST 2017 agile experiment(section 6.1).

Ensuring that the prerequisites are met An initial context is assumed:

- The current working directory is *..../ReplicationPackage*.
- There is a *res* folder containing:
  - A *declarations.txt* file containing the declarations for each of the test cases.
  - An *imports.txt* file containing the imports for each of the test cases.
  - An *uca-id-list* file containing the UCA ids to generate test cases for.
  - A *variables-values.csv* file containing mapping of SPMVs and possible values to use in test case generation.
  - A *variable-types.csv* file mapping each SPMV to a Java type.
  - A *xstampp-all.csv* file containing all UCAs.
- There is a *tests4jbehave* folder containing the executable *.jar*.
- There is a *workspace* folder containing a prepared Eclipse Maven workspace.

Step 1 *tests4jbehave* is used to generate test cases by executing it with the following options:

```
java -jar tests4jbehave/tests4jbehave-1.0.0-SNAPSHOT-jar-with-dependencies.jar
-j res/imports.txt -k res/declarations.txt -r workspace/autonomous-parking-system/src/test/
-s java/com/bddexperiment/jbehave/ -p com.bddexperiment.jbehave -v res/variable-types.csv
-x res/xstampp-all.csv -z res/variables-values.csv -u res/uca-id-list.csv -e pairwise.
```

The resulting test cases are generated into the prepared Eclipse workspace.

For each UCA there is a package containing the Java test suites and a *steps* subpackage containing Java test cases with the stub definitions.

In the *resources* folder all scenarios for each UCA are in the *.story* files in the corresponding subdirectory.

With this, the considered scenarios for *Step 1* are generated and *Step 1* is completed. The Eclipse workspace could look similar to the sample found in figure 5.1.

Step 2 In *Step 2* all that needs to be done is to write the test code in the generated step definitions in the Java test cases.

When a test case is written, the testing process in *Step 2* should be applied (figure 4.3). This means that a scenario and corresponding test case is examined for its viability and either kept or discarded. If it is kept, *examples* must be either manually written, or if some were generated, unfit ones must be removed to make the test case pass and thus discover unsafe scenarios that cause a UCA to occur.

Step 2 is the more time consuming step, compared to *Step 1*.

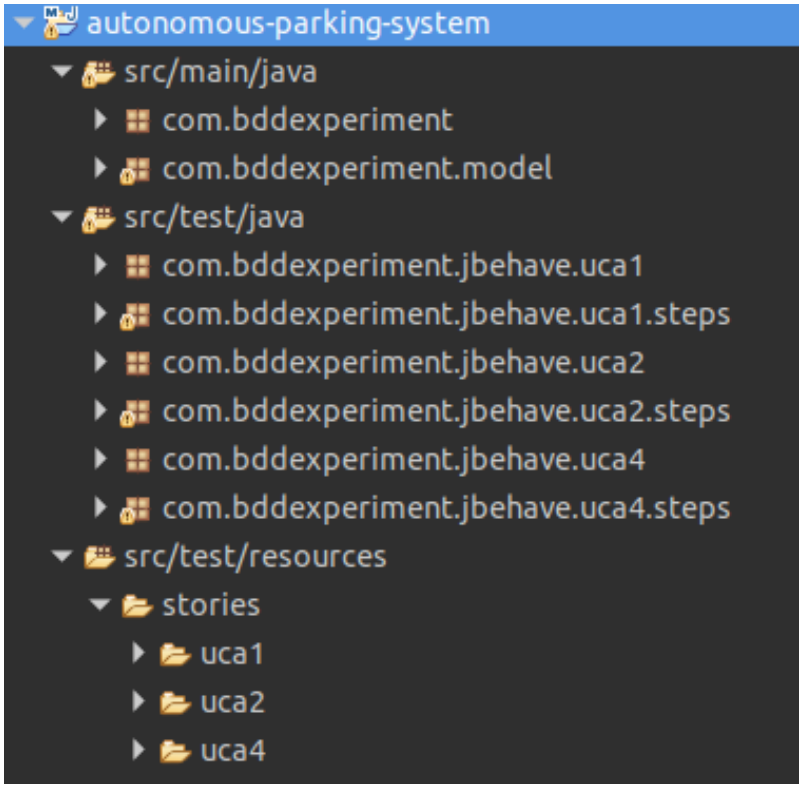

Figure 5.1: Sample Eclipse Workspace

Results The improved workflow has considerably less manual interaction than the old workflow, leading to a faster output of test cases.

As an added benefit, different value combinations can be generated as *examples* by *tests4jbehave* for each scenario, matching and assisting the BDD approach *specification-by-example*.

JBehave will execute each scenario once for each value combination supplied for that scenario. This can potentially lead to large numbers of test executions, depending on the number of scenarios kept and number of *examples* supplied with each scenario.

The results look fairly promising:

- *Step 1*:
  - Number of UCAs written into unsafe scenarios: 9.
  - Number of unsafe scenarios: 72 (1944 with examples).
  - Time used: 10 min.
- *Step 2*:
  - Number of passed test cases: 13 test cases with a total of 137 passed scenarios.
  - Line coverage: 59 / 167 (35%).

- Mutation coverage: 6 / 157 (4%).
- Time used: 30 min.

An evaluation of the improved workflow in comparison to the old workflow can be found in chapter 6.

6 Evaluation

The improved workflow from chapter 4 and its implementation are evaluated in comparison to the old workflow of writing automated BDD acceptance tests.

For this, the EST 2017 agile experiment resources and results will be relied upon. The sample system and testing process used in the EST 2017 agile experiment will also be used when evaluating the effectiveness of the improvements to the old process.

6.1 EST 2017 Agile Experiment

To determine if automated acceptance testing with BDD and JBehave offers benefits in terms of productivity over UAT and manual testing, an experiment comparing the two approaches was conducted in the EST 2017 Summer lecture at the University of Stuttgart.

The strategy used for unsafe scenario creation, to isolate single SPMV to discover those that cause UCAs, is also basis of this work.

The results indicate that automated acceptance testing with BDD and JBehave shows promise, but improvements in productivity are desired.

6.2 Evaluation

The research objective is to determine if there is an increase in productivity with the improvements made to the process of writing BDD acceptance tests for safety-critical systems over the old process. Both processes are described in chapter 4.

For this, an evaluation is conducted with a group of students where the improved process is used to BDD acceptance test a sample system under test.

The results are then compared to the results of the EST 2017 agile experiment (section 6.1), which used the old workflow as well as UAT, to check for changes in productivity.

The details of the improved workflow used here can be found in section 5.4.

This evaluation mostly follows the *Steps in the Experiment Process* described in *Experimentation in Software Engineering* [WRH+12].

Effectively the evaluation design should be, "one factor with two treatments". The factor is the workflow choice and the treatments are either the old or improved workflow. Section 6.2.3 mentions some details regarding this.

6.2.1 Scoping

The goal of this evaluation is defined as follows:

```
Analyze the difference between the old process and the improved process of writing
BDD acceptance tests for safety critical systems
for the purpose of evaluation
with respect to productivity
from the point of view of the researcher
in the context of M.Sc. students writing BDD acceptance tests for a sample system.
```

In addition to productivity, fault detection capability is considered as well. This is for the purpose of showing that a possible positive change in productivity does not negatively impact the quality of the tests produced by the new workflow.

6.2.2 Hypotheses

For productivity the following hypotheses are considered:

- The null hypothesis, H_0, is that there is no difference in productivity using the improved process over using the old process.
  The alternative hypothesis, H_1, is that there is a positive difference in productivity using the improved process over the old process.

For fault detection capability the following hypothesis are considered:

- The null hypothesis, H_0, is that there is no difference in fault detection capability using the improved process over the old process.
  The alternative hypothesis, H_1, is that there is a positive difference in fault detection capability using the improved process over the old process.
- The above null and alternative hypotheses are repeated with a different variable for improved confidence.

6.2.3 Independent Variable

The independent variable in this case is the use of *tests4jbehave* with the improved process. During this evaluation the improved process is always used. The case where the old process is used is covered by the results of the EST 2017 agile experiment (6.1), which are merged with the results of this evaluation.

6.2.4 Dependent Variables

The dependent variables should measure the effects of the independent variables in terms of productivity as well as fault detection capability. For this we measure the following variables:

- Number of UCAs written into unsafe scenarios. (NIUS)
  NIUS measures productivity.
- Line coverage of all tests written. (LC)
  LC measures fault detection capability.
- Mutation coverage of all tests written. (MSI)
  MSI also measures fault detection capability.

6.2.5 Subjects

The subjects are 11 students of the software technology course and chosen based on convenience of availability.

6.2.6 Object

The object of the evaluation will be the sample system the subjects write BDD acceptance tests for in their assigned workflow.

6.2.7 Operationalization

The evaluation follows the EST 2017 agile experiment (section 6.1) in its procedure. This means that the evaluation is split into 2 phases, lasting 30 minutes each. A third phase as used in EST 2017 agile experiment is irrelevant in this case. Phase 1 will mirror *Step 1* in the improved process and phase 2 will mirror *Step 2*.

Using *tests4jbehave* unsafe scenarios are generated and then sifted for suitable scenarios during phase 1. After the participant is finished, or after 30 minutes, phase 2 begins.

The participant now writes the step definitions for the scenarios created in phase 1 and writes test code in such a way that the acceptance tests pass, thereby discovering which scenarios cause UCAs and are truly unsafe.

In general, the procedure from section 5.4 is followed.

Guidelines

For the evaluation the following guidelines are followed:

- The evaluation takes place in the same room under conditions that are as constant as possible for every participant.
- The participants receive an explanation of the material to prepare for the evaluation:
  - The sample system that is to be tested.
  - Writing BDD scenarios and Java test cases.
  - Using JBehave with Eclipse.
  - The workflow that must be followed.

  Printouts are supplied as well.
- The variables and time used are measured and the data collected by means of survey sheets, which are filled out by hand after each phase in the evaluation and at the end.
- The evaluation will be conducted over the time period of 1 week.
- The participants are guaranteed anonymity.
- Each participant is awarded the same monetary compensation.

6.3 Evaluation Results

The results of the evaluation are viewed together with the earlier EST 2017 agile experiment (section 6.1). In total, there are three experiments. The experiments "UAT" and "BDD 1" are the previous experiments results, where "UAT" was a manual UAT testing process and "BDD 1" was the old workflow. "BDD 2" is the repeat of the experiment with the improved workflow, namely this evaluation.

Based on the results (figure 6.1 and figure 6.2) all three null hypotheses (section 6.2.2) are rejected. This means there is an increase in productivity, measured by the variable NIUS, and fault detection capability, measured by the variables LC and MSI, when using the improved workflow for BDD automated acceptance testing over the more manual alternatives. This is a positive result, that shows the improved workflow is effective without obvious drawbacks in test quality.

Measure	Experiment	Mean	St.Dev	St. Error	Max	Median	Min	95% CI lower	95% CI upper
NIUS	UAT	**0.58**	0.22	0.02	1.00	0.57	0.33	0.45	0.71
	BDD 1	0.52	0.24	0.02	1.20	0.45	0.26	0.37	0.66
	BDD 2	**3.72**	1.16	0.11	5.67	4.00	1.20	3.03	4.40
LC	UAT	**3.44**	1.78	0.16	7.70	3.06	1.53	2.39	4.49
	BDD 1	4.83	2.26	0.21	11.00	4.00	2.64	3.50	6.17
	BDD 2	**7.85**	1.86	0.17	11.43	6.92	5.64	6.74	8.95
MSI	UAT	**0.89**	0.36	0.03	1.56	0.88	0.42	0.67	1.10
	BDD 1	0.90	0.38	0.03	1.33	1.00	0.36	0.67	1.13
	BDD 2	**1.95**	0.67	0.06	3.63	1.98	0.99	1.56	2.34

Figure 6.1: Results - Descriptive Statistics

	Hypothesis Testing	Mann-Whitney U	P-value	Z	Wilcoxon W	ANOVA F	d	Rejected H_0 or not
Productivity	NIUS	0	0.00004	-3.940	0	70.512	3.761	Rejected H_0
Fault detection capability	LC	6	0.00019	-3.546	0	29.270	2.422	Rejected H_0
	MSI	8	0.00032	-3.415	1.5	19.672	1.971	Rejected H_0

Figure 6.2: Results - Hypothesis Testing

6.4 Threats to Validity

The threats to validity considered are of the four types: *conclusion, internal, construct and external validity*.

6.4.1 Conclusion Validity

- Reliability of measures: Data might be falsely recorded by the participants, as each participant is responsible for recording his or her data. This is mitigated by double checking results after the experiment and re-running the tests on the artifacts produced by the participants.
- Reliability of treatment implementation: While the same explanation and conditions are presented to each participant for this evaluation, this can not necessarily be said about the EST 2017 agile experiment. As it was conducted months back, there are

surely some differences to this evaluation's procedure. There is nothing that can really be done about this, except sticking to known information as much as possible.

- Random heterogeneity of subjects: As 11 students from the software technology course are selected, it is reasonable to assume that the impact of individual difference in capabilities is limited. However, the group selected is rather specific and not necessarily representative and this could pose problems for external validity.

6.4.2 Internal Validity

- History: The evaluation was carried out about 4 months after the EST 2017 agile experiment. Although effort was expended to replicate conditions as much as possible, it is not realistic to expect perfect replication, partly since some details will not be documented. There is not really anything that can be done about this though.
- Maturation: As the tasks for the participants are somewhat repetitive, as in writing tests in the same manner, the risk for negative effects is quite small, since the time allotted for each phase is only 30 minutes. This is also not enough time for large improvements through learning in this small time frame. Due to this the threat of maturation is not deemed too large. Small improvements during phase 2 of the evaluation are somewhat expected though, as immediate feedback by executing the written test is gained.
- Instrumentation: The same survey and printouts for the material are supplied as in the EST 2017 agile experiment for as close as possible condition replication. If there are problems with the documents, they must have also existed in the EST 2017 agile experiment, and thus will not impact the comparison of results. This is why instrumentation will not be considered a threat that can be countered.
- Selection: Only volunteers are selected for this evaluation. While it is true that volunteers may not be representative and show above average motivation, there are reasons to ignore this as a threat. Firstly, the EST 2017 agile experiment was also conducted on volunteer basis, and to keep the conditions in this evaluation as close as possible, volunteers based on availability are chosen. Secondly there is no other way to gain participants in the first place in this case.
- Diffusion or imitation of treatments: This evaluation is carried out over the course of one week, meaning theoretically participants could learn from each other. To combat this, no participant is informed who else takes part in the evaluation. Participants are not informed how good their results are in comparison. A small risk that participants learn of each other remains though. Additionally participants could learn from the EST 2017 agile experiment if they have acquaintances that took part there. This is not something that can be controlled though and can therefore only be ignored.

6.4.3 Construct Validity

- Interaction of testing and treatment: This is a real risk, as subjects fill out the data collection form themselves. They are aware of this form at all times and might purposefully try to increase LC, MSI or NIUS, as they must fill out the form with their results after each phase. Also, as the number of passed test cases is measured but not considered for this evaluation. The subjects are not aware of this and might possibly decrease natural results in considered variables in favor of artificially higher scores in ignored ones. To try to counter this, no incentives for good results are offered or implied. The explanation for the participants glosses over the data collection sheet, in hopes that it is only considered by any given participant at the end of a phase.
- Restricted generalizability across constructs: This is a real risk, as the main focus is productivity. However the variables LC and MSI are also considered to represent fault detection capability. An increase in productivity that causes a decline in test quality can therefore be detected.
- Hypothesis guessing: The participants are aware of the data collection form before they begin. They may guess which of the recorded values are considered for which hypothesis and what that hypothesis might be. However they are not informed what the comparison standard is. The only information participants receive is that the evaluation will determine the effectiveness of the workflow they follow. While it is possible to purposefully maximize the values of NIUS by not following proper procedure of deleting unchecked or unwanted test cases, the values of LC and MSI require real testing code to be written. The requirement that each test must pass for MSI to even be measurable means that manipulation of this type is a lot more difficult, especially considering the short time frame participants are allotted. The risk for manipulation is certainly present for NIUS. Aside from checking for abnormally high values, not much can be done.
- Evaluation apprehension: This is a potential risk, especially as the subjects record the values themselves. For this reason values are double checked by collecting the artifacts produced by the subjects and executing the produced test cases again. Mitigation also comes from the fact that participants are not informed against which group, method or individual they compete.

6.4.4 External Validity

- Interaction of selection and treatment: Only students were selected, on the basis of available volunteers, just as in the EST 2017 agile experiment. The participants might not represent the entire spectrum, however, due to the conditions of EST 2017 agile experiment being replicated this point is rendered mute.
- Interaction of setting and treatment: This could be the case, as the sample system used is considered a toy system with limited size. As participants must be able to test

it in a short amount of time, the restricted size corresponds to the limitation of the evaluation process.

7 Conclusion

BDD acceptance testing is a fitting method to test safety critical systems for safety requirements. An existing BDD testing approach based on artifacts produced by STPA was examined with the goal of finding inefficiencies and improving upon them. That approach was used in the earlier EST 2017 agile experiment, showing its viability. It is based on a test case scheme centered around isolating a process variable that would cause UCAs to occur with the goal of discovering unsafe scenarios. Writing tests according to this scheme and ready to use with JBehave as BDD acceptance tests is quite repetitive and tedious. During the course of this work improvement potential was identified and the process partially automated.

Improvements to the process were implemented in the form of a command line tool, automating and thereby alleviating repetitive parts of the process. The command line tool generates test cases according to a given scheme and respects naming and location conventions, so that JBehave can directly be used with the generated test cases to BDD acceptance test a system.

Finally, an evaluation of the command line tool was conducted that utilized the results from the EST 2017 agile experiment for comparison. During the evaluation a sample system was BDD acceptance tested, according to an improved workflow that relies on the new command line tool. The effectiveness of the improvements was determined by the evaluation, and the results show that the improved workflow has increased productivity and fault detection capability over the processes used in the EST 2017 agile experiment.

7.1 Future Work

As for future work, the command line tool for automation could be improved. One possibility is to support Cucumber in addition to JBehave. Also the limitations imposed by JBehave are still present in the tool and it might be desirable to remove them and make the tool more flexible. This is also an implicit requirement for an extension that would add support for Cucumber. Additional improvements could also be capability for different naming, location and generation schemes.

Bibliography

[Abdulkhaleq15] A. Abdulkhaleq, S. Wagner. "XSTAMPP: An eXtensible STAMP platform as tool support for safety engineering." eng. In: *2015 STAMP Workshop, MIT, Boston, USA*. Universität Stuttgart, 2015. URL: http://elib.uni-stuttgart.de/opus/volltexte/2015/9987 (cit. on p. 18).

[AT17] *Acceptance Test*. 2017. URL: https://www.agilealliance.org/glossary/acceptance/ (cit. on p. 18).

[ATDD17] *Acceptance Test Driven Development*. 2017. URL: https://www.agilealliance.org/glossary/atdd/ (cit. on p. 18).

[BDD17] *Behavior Driven Development*. 2017. URL: https://www.agilealliance.org/glossary/bdd/ (cit. on p. 19).

[CC17] *Cucumber*. 2017. URL: https://cucumber.io/ (cit. on p. 21).

[CG08] L. Crispin, J. Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2008, pp. 12–14. ISBN: 9780321534460 (cit. on p. 18).

[Conti18] *BDD Security - Continuum*. 2018. URL: https://www.continuumsecurity.net/bdd-security/ (cit. on pp. 21, 22).

[JBW17] *JBehave*. 2017. URL: http://jbehave.org (cit. on pp. 17, 19).

[LecQSW17] *Qualitätssicherung und Wartung - Sommer 2017, University of Stuttgart*. 2017. URL: http://www.iste.uni-stuttgart.de/se/lehre/fruehere-semester/sommersemester-2017/qualitaetssicherung-und-wartung-qsw.html (cit. on pp. 17, 21).

[PPG04] Y. Papadopoulos, D. Parker, C. Grante. "Automating the failure modes and effects analysis of safety critical systems." In: *Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings.* Mar. 2004, pp. 310–311. DOI: 10.1109/HASE.2004.1281774 (cit. on p. 21).

[PWT17] *Pairwise Testing*. 2017. URL: http://pairwise.org (cit. on p. 28).

[Seren17] *Serenity BDD*. 2017. URL: http://thucydides.info/docs/serenity/#introduction (cit. on p. 33).

[Sma14a] J. F. Smart. *BDD in Action*. Manning, 2014, p. 14. ISBN: 9781617291654 (cit. on p. 19).

[Sma14b] J. F. Smart. *BDD in Action*. Manning, 2014, p. 21. ISBN: 9781617291654 (cit. on p. 19).

[Sma14c] J. F. Smart. *BDD in Action*. Manning, 2014, p. 125. ISBN: 9781617291654 (cit. on p. 28).

[TDD17] *Test Driven Development*. 2017. URL: https://www.agilealliance.org/glossary/tdd/ (cit. on p. 18).

[Thucy17] *Thucydides*. 2017. URL: https://github.com/thucydides-webtests/thucydides (cit. on p. 33).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012. ISBN: 9783642290442 (cit. on p. 41).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature