Behavior-Driven Requirements Traceability via Automated Acceptance Tests

Garm Lucassen, Fabiano Dalpiaz,
Jan Martijn E.M. van der Werf, Sjaak Brinkkemper
Utrecht University, The Netherlands
{g.lucassen, f.dalpiaz, j.m.e.m.vanderwerf, s.brinkkemper}@uu.nl

Didar Zowghi
University of Technology Sydney, Australia
didar.zowghi@uts.edu.au

***Abstract*—Although information retrieval advances significantly improved automated traceability tools, their accuracy is still far from 100% and therefore they still need human intervention. Furthermore, despite the demonstrated benefits of traceability, many practitioners find the overhead for its creation and maintenance too high. We propose the Behavior Driven Traceability Method (BDT) that takes a different standpoint on automated traceability: we establish ubiquitous traceability between user story requirements and source code by taking advantage of the automated acceptance tests that are created as part of the Behavior Driven Development process.**

***Index Terms*—User stories, automated acceptance tests, traceability, requirements, behavior-driven development**

I. Introduction

The benefits of maintaining requirements traceability include support for change impact analysis, increased program comprehension and faster software development [1]. Nonetheless, many industry practitioners do not adopt traceability practices [2]. Two main reasons are that (i) the stakeholders who need to create the links are not the ones who reap the benefits, the so-called *traceability benefit problem* [3], [4], and (ii) the lack of methods and tools supporting traceability [5] hamper the effective adoption of traceability practices [6].

Recognizing these problems, Cleland-Huang and colleagues put forward the grand challenge of "always there" *ubiquitous* traceability that is "built into the software engineering process" [6]. This is particularly relevant for the ad-hoc and just-in-time requirements engineering practices of agile and open source projects, which do not go beyond prefacing a commit message with an issue ID [7]. Many tools have been proposed [8]–[10] that largely rely on information retrieval (IR) algorithms. Although promising, those IR algorithms do not yield the 100% accuracy that ubiquitous traceability requires and their performance is highly dependent on the input data quality and type [11].

In this paper, we present the automated Behavior-Driven Traceability method to establish ubiquitous traceability by taking advantage of *automated acceptance tests*. Created as part of the software engineering process *Behavior-Driven Development* or BDD [12], these tests refine user story requirements into steps that mirror a user's interaction with the system. A typical acceptance test has at least three steps: (i) establishing a starting position by navigating to a specific interface, (ii) executing one user action such as clicking a button and (iii) asserting whether the action produced the expected effect.

Unlike unit tests, these steps do not directly execute the source code itself. Instead, a BDD framework launches a complete instantiation of the software system and then simulates how a user would interact with the interface. This creates an opportunity to leverage runtime tracers to identify all the source code invoked to realize a given user story without requiring imprecise information retrieval techniques.

After presenting the necessary background in Section II, we introduce the *Behavior-Driven Traceability* method (BDT) for generating execution traces that link user stories to code via automated acceptance tests in Section III. Throughout these sections, we illustrate our work with the running example of the fictitious EventCompany's software that enables event organizers to sell tickets to their visitors online. The discussion and outlook in Section IV concludes with the implications of BDT and opportunities for future work.

II. Background

We present the relevant literature for our approach: agile requirements via user stories, requirements traceability and its role in agile development, and behavior-driven development.

A. User Stories

User stories are a concise notation for capturing requirements whose adoption has grown to 50% thanks to the increasing popularity of agile development practices such as Scrum [13], [14]. A user story consists of three basic components: (1) a short text describing the user story itself, (2) conversations between stakeholders to exchange perspectives on the user story, and (3) acceptance criteria. In this paper, we are concerned with the first and the third elements.

The first component captures the essential elements of a requirement: *who* it is for, *what* is expected from the system, and (optionally) *why* it is important. An example of a user story that follows the popular Connextra format [15] is the following: *"As an Administrator, I want to receive an email when a contact form is submitted, so that I can respond to it"*.

B. Requirements traceability

Requirements traceability has been studied for almost three decades [16] and already in 2003 Lee et al. emphasized the

importance of early and non-obtrusive methods for traceability to become successful in agile software development [17]. This is echoed by the vision for *ubiquitous traceability* that is "always there" and "built into the engineering process" [6].

Recent works contribute towards this vision with varying levels of success. Ranging from lightweight just-in-time traceability [18] to sophisticated approaches that combine IR methods to achieve state-of-the-art accuracy [19], [20]. Unfortunately, industry has yet to embrace these advances.

The software industry adopts simpler agile traceability practices, such as prefacing code commit messages with an issue ID to manually create an issue-code segment trace [7]. However, this method is vulnerable to human negligence and requires processing large amounts of historical data.

C. Behavior-Driven Development

Behavior-Driven Development (BDD) [12] augments Test Driven Development (TDD) [21] in two ways: (a) teams should formulate a simple "ubiquitous language" for capturing automated acceptance tests that any team member can read and (b) these acceptance tests should specify user *behavior* for the system to fulfill. See Listing 1 for an example acceptance test utilizing the industry-standard Gherkin syntax, which describes a series of steps using **given** *some initial context*, **when** *an event occurs*, **then** *ensure some outcome* [22]. While not ubiquitous, it is recommended practice to annotate a BDD test with the user story it tests [12]. For illustration purposes, we refer to requirements for the EventCompany scenario (see Section I).

Listing 1 Example acceptance tests for EventCompany us1

```
1:  Feature: Contact Form
2:    As a Visitor
3:    I want to use the contact form
4:    so that I can contact the organizer.

5:  Scenario: Submit contact form
6:    Given I go to the "contact" page
7:     visit path_to(contact_page)
8:    When I enter "Hello World" in the "Question" field
9:     fill_in('Question', with: 'Hello World')
10:   And I submit the form
11:    click_button('Submit')
12:   Then the organizer receives a message
13:    open_email('organizer@webcompany.com')
14:    expect(current_email).to have_content 'Hello World'
```

Each scenario step captures a single *behavior* for the system to fulfill. If these tests are maintained and extended to support new functionality, a comprehensive collection of BDD tests functions as a growing, accurate and up-to-date documentation source for the entire software system [22].

Unlike unit tests, BDD tests do not directly execute methods or code parts. Instead, a BDD test framework launches the entire software system and simulates how a real user would interact with the software's interface step by step. As a consequence, the execution trace of a BDD test includes all the source code invoked to realize its annotated user story. This creates an opportunity to trace user story requirements without requiring probabilistic—thus, imprecise—IR techniques. While the tracing is still only as good as the human-created BDD tests, unlike other approaches BDD output does not require any additional effort to establish ubiquitous traceability.

Although BDD is still a fringe development process, it has evolved and grown considerably in the past ten years. Community-driven initiatives have resulted in a mature development approach with ample reference literature and strong tool support. Nowadays, the Cucumber tool and its associated *The Cucumber Book* are the de-facto standard for BDD [22].

Research that uses or extends the approach is limited to automatically analyzing BDD acceptance tests using NLP to either (i) suggest source code or test code [23], [24], or (ii) reconcile BDD with Business Process Modeling [25], [26].

III. Behavior-Driven Traceability

We propose the Behavior-Driven Traceability Method or *BDT* (see Fig. 1) that automatically establishes ubiquitous traceability on top of the well-established BDD process. BDT relies on two key features of BDD: (i) its detailed decomposition of each user story in brief scenario tests that describe end-user interaction in a stepwise fashion, and (ii) the practice of operationalizing these steps on the system's interface instead of the source code. We explain how the BDT Method takes advantage of these characteristics and introduce the *BDT Tracer* for building the so-called *BDT Matrix* that records the source code and methods called for each user story. We illustrate each step using EventCompany as an example.

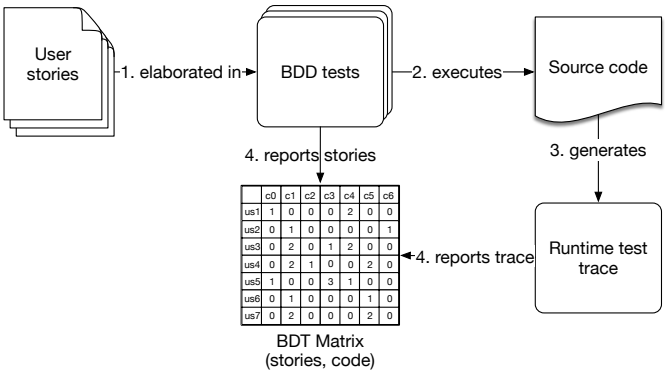

Fig. 1. Behavior-Driven Traceability Method

As an extension of BDD, BDT first requires the formulation of user stories, which the developers subsequently manually *(1) elaborate in* a collection of BDD tests that validate whether the developed features satisfy the customer's requirements. We expand upon EventCompany's initial user story and associated BDD test in Listing 1 by introducing us2: *"As an Organizer, I want to register my event, so that I can sell tickets"*. As the development of both user stories completes, successfully running the BDD test suite *(2) executes* the source code by mimicking how a real person would use the system's interface. To *(3) generate* runtime test traces for the BDD tests we need to integrate with and extend a project's source code so it logs

Listing 2 Snippet of the output of the BDT Runtime Tracer for the Contact Form user story (us1)

```
    File                    Class:method          us#  BDD scenario                BDD test step
1   classes/user.rb:10      User:current_user     us1  features/contact.feature:5  features/contact.feature:6
2   classes/user.rb:22      User:track_visitor    us1  features/contact.feature:5  features/contact.feature:6
3   classes/form.rb:15      Form:process          us1  features/contact.feature:5  features/contact.feature:10
4   classes/user.rb:22      User:track_visitor    us1  features/contact.feature:5  features/contact.feature:10
```

all invoked methods. To achieve this, we leverage the availability of runtime tracers for programming languages: Python has traceback, .Net comes with Environment.StackTrace and Java supports getStackTrace. Since BDD tests decompose each user story into small steps, these tracers can relate lines of code to BDD artifacts at *three levels of granularity*: an entire user story, a scenario that tests part of the user story, or a single scenario step. Our BDT Tracer on GitHub[1] shows how we configure Ruby's TracePoint to record traces for all relevant method calls in a Ruby on Rails project.

Listing 2 shows a partial runtime trace generated by the Ruby BDT Tracer for the Contact Form feature of Listing 1. Each line lists the location of the executed source code file, the invoked class and method, the user story in the description, the BDD scenario and the corresponding BDD test step that triggered the executed source code. Note that the numbers 5, 6 and 10 after the BDD scenarios and test steps refer to the line numbers of Listing 1. This is formalized in Definition 1.

Def. 1 [Runtime test trace] Given a set of user stories $US = \{us_1, \ldots, us_n\}$, a runtime test trace rtt_{us} is a list built by sequentially executing all the BDT tests of all user stories in the set, where every list element is a tuple $\langle loc, cl, meth, us, scen, step\rangle$ such that *loc* is the source code location of the class *cl* whose method *meth* was executed as part of *step* of the scenario *scen* of the user story *us*.

The BDT method then *(4) combines the reported* runtime test trace with the BDD test's annotated user stories into the BDT Matrix. This matrix records user stories on the Y axis, source code methods on the X axis and the number of times each user story invokes each method in the cells. Table I shows the BDT Matrix for the two EventCompany's user stories introduced earlier in this paper. Definition 3 formalizes the notion of a BDT Matrix after Definition 2 presents the preliminary concept of invocation frequency.

Def. 2 [Invocation frequency] Given a user story us_i and a runtime trace $rtt_{\{us_i\}}$, the invocation frequency for a method m of class c (denoted as *c:m*) $invFreq_{us,c:m} \in \mathcal{N}$ indicates the number of tuples $\langle loc, cl, meth, us, scen, step\rangle$ in $rtt_{\{us_i\}}$ such that $cl = c$ and $meth = m$.

Def. 3 [BDT Matrix] Let $US = \{us_1, \ldots, us_n\}$ be a set of user stories, $M = \{m'_1, \ldots, m'_q\}$ be a set of methods (each included in some class). A BDT Matrix *BDT* has size $n \times q$, and each cell indicates the invocation frequency of a method in the BDD tests of a user story. Formally, $\forall i, j \in \mathcal{N}^+$ such that $i \leq n, j \leq q$, then $BDT_{i,j} = invFreq(us_i, m'_j)$.

When a software development team creates individual BDD tests for each user story, applying BDT results in a BDT Matrix that allows a developer to request all the source code invoked to realize a given user story. By applying smart filtering techniques the BDT Matrix can then be used to produce a variety of reports, such as methods that are never called in the entire test suite to identify dead code, or all the classes involved in a specific user story to inform developers modifying or refactoring a user story's code.

TABLE I
PARTIAL BDT MATRIX FOR THE EVENTCOMPANY CASE

	User:		**Form:**		**Event:**
	current_user	track_visitor	process	encode_media	check_pricing
us1	1	2	1	0	1
us2	1	0	1	1	1

Note, however, the significant overlap in the BDT Matrix: 3 out of 5 methods are called when running the test for both user stories. This is typical for high quality software that relies on code reuse. Thus, BDT is prone to including omni-present code: classes and methods that are called in almost every test step. These create noise in the output, making it more difficult to identify the unique code for that user story.

To reduce this effect, we take inspiration from *feature location* techniques that automatically identify which part of the source code implements a given functionality [27]. Our situation is comparable to *software reconnaissance* [28]: runtime traces produced by running test scenarios contain a lot of shared source code, thereby hiding uncommon and feature-specific methods. They identify the *"uniquely involved components"* for a feature by taking the set of components exercised as part of the test cases related to that feature and then excluding any components exercised in test cases unrelated to the feature. We go beyond this approach and look at the relative importance of the uniquely involved components.

To identify all uncommon, relevant methods and retain their relative importance, we normalize the BDT Matrix by dividing the number of method calls for a user story by the total number

TABLE II
NORMALIZED PARTIAL BDT MATRIX FOR THE EVENTCOMPANY CASE

	User:		**Form:**		**Event:**
	current_user	track_visitor	process	encode_media	check_pricing
us1	0.5	2	0.5	0	0.5
us2	0.5	0	0.5	1	0.5

[1]https://github.com/gglucass/BDT/blob/master/bdt_tracer.rb

of user stories, scenario's or steps the method is called in, resulting in the normalized BDT Matrix in Table II. Note for example that the User:current_user method is called in both user stories, resulting in a normalized BDT output of $1/2 = 0.5$. User:track_visitor is only called in us1, however, leading to a normalized BDT output of $2/1 = 2$, emphasizing the important role of this method for us1.

Def. 4 [Normalized BDT Matrix] Given a *BDT* Matrix of size $n \times q$, a normalized BDT Matrix $NBDT$ has size $n \times q$ and its cells denote the relative frequency for a user story to invoke a method with respect to how many other user stories also invoke that method. Formally, let ite be the if-then-else operator, and $\forall i, j \in \mathcal{N}.i \leq n, j \leq q$,

$$NBDT^{us}_{i,j} = \begin{cases} 0 & \text{if } BDT_{i,j} = 0 \\ \dfrac{BDT_{i,j}}{\sum\limits_{1 \leq k \leq n} ite(BDT_{k,j} > 0, 1, 0)} & \text{otherwise} \end{cases}$$

It is also possible to obtain a more fine-grained normalization with respect to the number of test scenario ($NBDT^{sc}$) or steps ($NBDT^{st}$), instead of the number of user stories. In those two variants, the numerator stays the same ($BDT_{i,j}$), while the denominator is the number of scenarios and the number of steps that invoke method j, respectively.

IV. Discussion and Outlook

This paper proposed the Behavior-Driven Traceability method (BDT) that, building on the agile software development process *Behavior-Driven Development*, attempts to establish *ubiquitous* traceability [6]. We have shown how to trace user story requirements to source code by processing runtime traces of automated acceptance tests, and how normalized versions of the BDT matrix emphasize user-story-specific classes and methods.

Initial testing of our prototype on the open source project *Archive of Our Own* (https://archiveofourown.org) produces promising output that motivates us to continue investigating this domain (available at https://goo.gl/LmDvQi). Both raw and normalized BDT output have merit: the former captures the prevalence and importance of a small number of classes and methods, while the latter highlights classes and methods that are important to a single user story.

In future work we intend to evaluate BDT output's accuracy by testing our prototype on industry projects and by involving practitioners. In particular, we want to understand whether the classes and methods in the NBDT matrix are relevant for software engineering tasks such as refactoring and resolving bugs. We are currently applying BDT at one industry partner to demonstrate to the Central Dutch Bank that its ability to accurately pinpoint which code is relevant for which requirement at any given time. Another interesting direction is automated change impact analysis: how to use BDT for identifying the impact of adding a new user story. Finally, we plan to experiment state-of-the-art feature location techniques as part of our normalization method.

References

[1] P. Mäder and A. Egyed, "Do developers benefit from requirements traceability when evolving and maintaining a software system?" *Empirical Software Engineering*, vol. 20, no. 2, pp. 413–441, 2015.

[2] F. Blaauboer, K. Sikkel, and M. N. Aydin, "Deciding to adopt requirements traceability in practice," in *Proc. of CAiSE*, 2007, pp. 294–308.

[3] P. Arkley and S. Riddle, "Overcoming the traceability benefit problem," in *Proc. of RE*, Aug 2005, pp. 385–389.

[4] D. M. Berry, K. Czarnecki, M. Antkiewicz, and M. Abdelrazik, "The problem of the lack of benefit of a document to its producer (PotLoBoaDtiP)," in *Proc. of SWSTE*, June 2016, pp. 37–42.

[5] E. Bouillon, P. Mäder, and I. Philippow, "A survey on usage scenarios for requirements traceability in practice," in *Proc. of REFSQ*, 2013, pp. 158–173.

[6] J. Cleland-Huang, O. C. Z. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: Trends and future directions," in *Proc. of FOSE*. ACM, 2014, pp. 55–69.

[7] N. A. Ernst and G. C. Murphy, "Case studies in just-in-time requirements analysis," in *Proc. of EmpiRE*, Sept 2012, pp. 25–32.

[8] M. Borg, P. Runeson, and A. Ardö, "Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1565–1616, 2014.

[9] A. M. D. Duarte, D. Duarte, and M. Thiry, "TraceBoK: Toward a software requirements traceability body of knowledge," in *Proc. of RE*, Sept 2016, pp. 236–245.

[10] T. Vale, E. S. de Almeida, V. Alves, U. Kulesza, N. Niu, and R. de Lima, "Software product lines traceability: A systematic mapping study," *Information and Software Technology*, 2016.

[11] T. Merten, D. Krämer, B. Mager, P. Schell, S. Bürsner, and B. Paech, "Do information retrieval algorithms for automated traceability perform effectively on issue tracking system data?" in *Proc. of REFSQ*, 2016, pp. 45–62.

[12] D. North, "Behavior modification," *Better Software Magazine*, Jun. 2006.

[13] L. Cao and B. Ramesh, "Agile requirements engineering practices: An empirical study," *IEEE Software*, vol. 25, no. 1, pp. 60–67, 2008.

[14] M. Kassab, "The Changing Landscape of Requirements Engineering Practices over the Past Decade," in *Proc. of EmpiRE*, 2015, pp. 1–8.

[15] G. Lucassen, F. Dalpiaz, J. M. E. M. van der Werf, and S. Brinkkemper, "The Use and Effectiveness of User Stories in Practice," in *Proc. of REFSQ*, ser. LNCS, vol. 9619. Springer, 2016, pp. 205–222.

[16] J. Cleland-Huang, "Traceability in Agile Projects," in *Software and Systems Traceability*. Springer, 2012, pp. 265–275.

[17] C. Lee, L. Guadagno, and X. Jia, "An agile approach to capturing requirements and traceability," in *Proc. of TEFSE*, 2003.

[18] J. Cleland-Huang, M. Rahimi, and P. Mäder, "Achieving lightweight trustworthy traceability," in *Proc. of FSE*, 2014, pp. 849–852.

[19] Y. Zhang, C. Wan, and B. Jin, "An empirical study on recovering requirement-to-code links," in *Proc. of SNPD*, May 2016, pp. 121–126.

[20] M. Rahimi, W. Goss, and J. Cleland-Huang, "Evolving requirements-to-code trace links across versions of a software system," in *Proc. of ICSME*, Oct 2016, pp. 99–109.

[21] K. Beck, *Test-driven development: By example*. Addison-Wesley, 2003.

[22] M. Wynne and A. Hellesoy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.

[23] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *Proc. of TOOLS*, C. A. Furia and S. Nanz, Eds., 2012, pp. 269–287.

[24] N. Gao and Z. Li, "Generating testing codes for behavior-driven development from problem diagrams: A tool-based approach," in *Proc. of RE*, Sept 2016, pp. 399–400.

[25] R. A. de Carvalho, F. L. de Carvalho e Silva, and R. S. Manhães, "Mapping business process modeling constructs to behavior driven development ubiquitous language," *CoRR*, vol. abs/1006.4892, 2010.

[26] D. Lübke and T. van Lessen, "Modeling test cases in bpmn for behavior-driven development," *IEEE Software*, vol. 33, no. 5, pp. 15–21, 2016.

[27] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[28] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.