Reducing the Discard of MBT Test Cases using Distance Functions

Thomaz Diniz
thomaz.morais@ccc.ufcg.edu.br
Federal University of Campina Grande

Everton L.G. Alves
everton@computacao.ufcg.edu.br
Federal University of Campina Grande

Anderson G.F. Silva
andersongfs@splab.ufcg.edu.br
Federal University of Campina Grande

Wilkerson L. Andrade
wilkerson@computacao.ufcg.edu.br
Federal University of Campina Grande

ABSTRACT

Model-Based Testing (MBT) is used for generating test suites from system models. However, as software evolves, its models tend to be updated, which often leads to obsolete test cases that are discarded. Test case discard can be very costly since essential data, such as execution history, are lost. In this paper, we investigate the use of distance functions to help to reduce the discard of MBT tests. For that, we ran a series of empirical studies using artifacts from industrial systems, and we analyzed how ten distance functions can classify the impact of MBT-centred use case edits. Our results showed that distance functions are effective for identifying low impact edits that lead to test cases that can be updated with little effort. Moreover, we found the optimal configuration for each distance function. Finally, we ran a case study that showed that, by using distance functions, we could reduce the discard of test cases by 15%.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

MBT, distance functions, test suite evolution, agile development

ACM Reference Format:
Thomaz Diniz, Everton L.G. Alves, Anderson G.F. Silva, and Wilkerson L. Andrade. 2019. Reducing the Discard of MBT Test Cases using Distance Functions. In *XXXIII Brazilian Symposium on Software Engineering (SBES 2019), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3350768.3350790

1 INTRODUCTION

Software testing plays an important role in a project since it helps developers to gain confidence that software works as expected [29]. Moreover, testing is fundamental for reducing risks and assessing software quality [29].

SBES 2019, September 23–27, 2019, Salvador, Brazil

ACM ISBN 978-1-4503-7651-8/19/09…$15.00
https://doi.org/10.1145/3350768.3350790

On the other hand, testing activities are known to be complex and costly. Studies have found that nearly 50% of a project's budget is related to testing [18]. Moreover, in practice, a test suite can combine manually and automatically executed test cases [14]. Although automation is desired, manually executed test cases are still very important. Itkonen et al. [14] state that manual testing plays an important role in the software industry and cannot be fully replaced by automatic testing. For instance, a tester that runs manual tests tents to better exercise a GUI and find new faults.

In this sense, several works guide efforts on trying to reduce the costs of testing. For instance, Model-Based Testing (MBT) [5, 36] is a strategy where test suites are automatically generated from specification models (e.g., use cases, UML diagrams). By using MBT, sound tests are extracted before any coding, and without much effort.

In the context of agile methodologies (e.g., eXtreme Programming [1] and Scrum [34]), where requirement changes are frequent to cope with client demands, test suites are used as safety nets for avoiding feature regression. In this scenario, an always updated test suite is mandatory. A recent work has proposed lightweight specification artifacts for enabling the use of MBT in agile projects [24], CLARET. With CLARET one can both specify requirements using use cases and generate MBT suites from them.

However, a different problem has emerged. As software evolves (e.g., bug fixes, change requirements, refactorings), both its models and test suite need revisions. In practice, since MBT test suites are generated from requirement models, as requirements change, the requirement artifacts are updated, and new test suites are generated. The new suites replace the old ones. Therefore, test cases that were impacted by the edits, instead of updated, are discarded [8]. Silva et al. [32] ran a study with MBT test suites which found that 86% of the test cases turn obsolete between two consecutive versions of a requirement file, and therefore are discarded. An obsolete test case is a test that includes at least one step that differs from the updated version of the specification document. Although one may find easy to just generate new tests, regression testing is based on a stable test suite that evolves. Test case discarding implies on important history data that are missed (e.g., execution time, the link faults-to-tests, fault-discovering time). In a scenario where development are guided by previously detected faults, missing tests can be a huge loss. Moreover, discard and poor testing are known to be signs of bad management and eventually lead to software development waste [31].

Silva et al. [32] also mention that part of the obsolete test cases could be easily reused with little effort and consequently reducing testing discard. However, manual analysis is tedious, costly and time-consuming, which often prevents its applicability in the agile context. In this sense, there is a need for an automatic way of detecting reusable test cases.

Distance functions [3] map a pair of strings to a real number that indicates the similarity level between the two versions. In a scenario where manual test cases evolve due to requirement changes, distance functions can be an interesting tool to help us classify the impact of the changes into a test case.

In this paper, we report a series of empirical studies using real industrial projects aiming at analyzing the efficiency of using distance functions to reclassify reusable test cases from the obsolete set. We mined the evolution of two projects that use MBT suites, and we use ten distance functions to classify edits between *low impact* (e.g., rewording, typo fixing) –require little test case updating– and *high impact* (specification and functionality changes) –require much test updating. Our results have found that all ten functions perform well on classifying edit impact. Moreover, we found the optimal configuration for using each function. Finally, we ran a case study in which our strategy was able to reduce the test case discard by at least 15%.

This paper is organized as follows. In section 2, we present a motivational example. The needed background is discussed in Section 3. Sections 4 and 5 present the performed empirical study and case study, respectively. In Section 6, some threats to validity are cleared. Finally, Sections 7 and 8 present related works and the concluding remarks.

2 MOTIVATIONAL EXAMPLE

Suppose that Ann works in a project and wants to benefit from MBT suites. Her project follow an agile methodology where requirements updates are expected to be frequent. Therefore, she decides to use Jorge et al.'s approach [24] (CLARET) for specifying requirements and generating test suites.

The following requirement was specified using the CLARET's DSL (Listing 1): "*In order to access her email inbox, the user must be registered in the system and provide a correct username and password. In case of incorrect username or password, the system must display an error message and ask for new data.*". In CLARET, an ***ef [flow #]*** mark refers to a possible exception flow, and a ***bs [step #]*** mark indicates a returning point from an exception/alternative to the use case's basic flow.

From this specification, the following test suite can be generated: S1 = {tc1, tc2, tc3}, where tc1 = [bs:1 → bs:2 → bs:3 → bs:4], tc2 = [bs:1 → bs:2 → bs:3 → ef[1]:1 → bs:3 → bs:4], and tc3 = [bs:1 → bs:2 → bs:3 → ef[2]:1 → bs:3 → bs:4] .

Suppose that in the following development cycle, the use case (Listing 1) was revisited and updated due to both requirement changes and for improving readability. Three edits were performed: (i) the message in line 9 was updated to "displays a successful message"; (ii) system message in line 12 was updated to "alerts that username does not exist"; and (iii) both description and system message in exception 3 (line 14) were updated to "Incorrect username/password combination" and "alerts that username and/or password are incorrect", respectively.

Since steps from all execution flows were edited (basic, exception 1, and exception 2), Ann discards S1 and generates a whole new suite. However, part of S1's tests were not much impacted and could be easily reused with little or no update. For instance, only edit (iii), in fact, changed the semantic of the use case, while (i) and (ii) are updates that do not interfere with the system's behavior. Therefore, only test cases that exercise the steps changed by (iii) should be in fact discarded (tc3). Moreover, test cases that exercise steps changed by (i) and/or (ii) could be easily reused and/or updated (tc1 and tc2).

We believe that an effective and automatic analyzer would help Ann to decide when to reuse and to discard test cases, and therefore reduce the burden of losing important testing data.

3 BACKGROUND

This section presents the use case notation considered in the paper along with the tool support and an example of a distance function.

3.1 CLARET and LTS-BT

CLARET [23, 24] is a DSL and tool that allows the creation of use case specifications using natural language. It was designed to be used as a central artifact for both requirement engineering and and MBT practices. Its toolset works as a syntax checker for use cases description files and provides visualization mechanisms for use case revision. Listing 1 presents a use case specification using CLARET.

From the use case description in Listing 1, CLARET generates its equivalent Annotated Labeled Transition System (ALTS) model [35] (Figure 1). Transition labels starting with *[c]* indicate pre or post conditions, while the ones starting with *[s]* and *[e]* are regular and exception execution steps, respectively.

CLARET's toolset includes a test generation tool, LTS-BT (Labeled Transition System-Based Testing) [2]. LTS-BT is an MBT tool that uses as input LTS models and generates test suites by traversing them. The generated tests are reported in XML files that can be

```
1  systemName "Email"
2  usecase "Log in User" {
3    actor emailUser "Email User"
4    preCondition "There is an active network connection"
5    basic {
6      step 1 emailUser "launches the login screen"
7      step 2 system "presents a form with username and password
         fields and a submit button"
8      step 3 emailUser "fills out the fields and click on the
         submit button"
9      step 4 system "displays a message" ef[1,2]
10   }
11   exception 1 "User does not exist in database" {
12     step 1 system "alerts that user does not exist"  bs 3
13   }
14   exception 2 "Incorrect password" {
15     step 1 system "alerts that the password is incorrect" bs 3
16   }
17   postCondition "User successfully logged"
18 }
```

Listing 1: Use Case specification using CLARET.

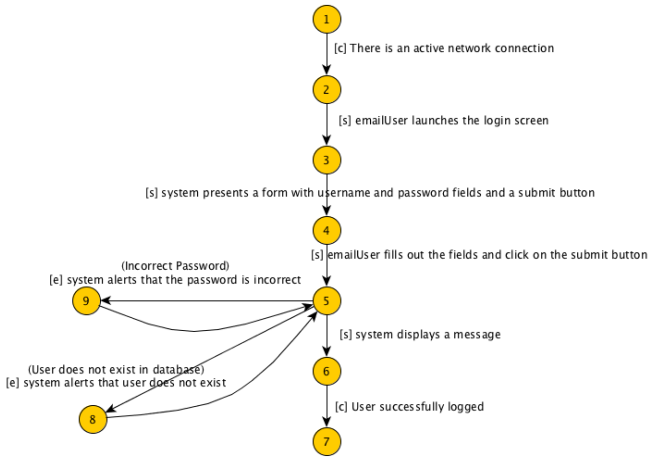

Figure 1: ALTS model of the use case from Listing 2.

directly imported to a test management tool, TestLink[1]. The test cases reported in Section 2 were collected from running LTS-BT.

3.2 Distance Functions

Distance functions are metrics for evaluating how similar, or different, are two strings [4]. Distance functions have been used in different contexts (e.g., [21, 27, 30]). Moreover, there are several different distance functions (e.g., [7, 10, 11, 13, 19]). For instance, the Levenshtein function [17, 19] (equation described below) compares two strings (a and b) and calculates the number of required operations to transform a into b, and vice-versa; where $1_{ai \neq bj}$ is the indicator function equal to 0 when $a_i \neq b_j$ and equal to 1 otherwise, and $lev_{a,b}$ is the distance between the first i characters of a and the first j characters of b.

For instance, consider a = "kitten" and b = "sitting", the Levenshtein distance is three, since three operations are needed to transform a to b: (i) replacing 'k' by 's'; (ii) replacing 'e' by 'i'; and (iii) inserting 'g' at the end.

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if min(i,j) = 0} \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{ai \neq bj} \end{cases} & otherwise \end{cases}$$

4 EMPIRICAL STUDY

We ran an empirical study for analyzing the use of distance functions for classifying changes in use case documents that could impact MBT suites.

4.1 Subjects and Functions

For that, we selected two industrial systems (SAFF and BZC) that were developed in the context of a cooperation between our research lab and two different companies, Ingenico do Brasil Ltda and Viceri Solution Ltda. The SAFF project is an information system that manages reports on payment terminals, and BZC is a system for optimizing e-commerce logistic activities. Both projects used agile methodologies to guide the development and updates in the requirement artifacts were frequent. Moreover, their teams used both CLARET [23], for use case case specification, and LTS-BT [2] for generating MBT suites.

Both projects use manually executed system level black-box test cases for regression purposes. In this sense, test case history data is very important since can help to keep track of the system evolution and to avoid functionality regression. However, the teams reported that often discard test cases when the related steps on the system use cases are updated in any form, which they refer as a practical management problem.

As our study focuses on the use of distance functions, we selected a set of ten of the most well-known functions that have been used in different contexts: Hamming [10], LCS [11], Cosine [13], Jaro [7], Jaro-Winkler [7], Jaccard [20], Ngram [16], Levenshtein [19], OSA [6], and Sorensen Dice [33]. To perform systematic analyzes, we normalize their results in a way that their values range from zero to one. Values near zero refer to low similarity, while near one values indicate high similarity. We reused open-source implementations of all ten functions[2][3]. To customize and analyze the edits in the context of our study, we created our own tool and scripts that were verified through a series of tests.

We mined the projects' repository and collected all use case edits. Each of these edits would then impact the test cases. We call "impacted" any test case that include steps that were updated during model maintenance. However, we aim to use distance functions to help us to classify these edits and avoid the test case discard.

4.2 Research Questions

To guide our investigation, we defined the following research questions:

- RQ1: Can distance functions be used to classify the impact of edits in use case documents?
- RQ2: Which distance function presents the best results for classifying edits in use case documents?

4.3 Study Setup and Procedure

Since the all use case documents were CLARET files, we mined the projects' repositories and collected, for each file *f*, its history of edits in a time frame. We consider an use case edit any update performed between two consecutive versions (*v1* and *v2*) of *f*. A total of 79 pairs of use case versions were analyzed in our study, with a total of 518 edits. Table 1 summarizes the data collected in our study considering the number of use cases, the number of versions, and the number of edited steps.

After that, we manually analyzed each edit and classified them between *low impact* and *high impact*. A **low impact** edit refers to changes that do not alter the system behavior (a pure synthetic edit), while a **high impact** edit refers to changes on the system expected behavior (semantic edit). Table 2 exemplifies this classification. While the edit in the first line changes the semantics of the original requirement, the next two refer to edits performed for improving

[1]http://testlink.org/

[2]https://github.com/luozhouyang/python-string-similarity

[3]https://rosettacode.org/wiki/Category:Programming_Tasks

Table 1: Summary of the artifacts used in our study.

	#Use Cases	#Versions	#Edits
SAFF	13	42	415
BZC	15	37	103
Total	28	79	518

Table 2: Classification of edits.

Steps Description		
Version 1	**Version 2**	**Classification**
"Extract data on offline mode."	"Show page that requires new data."	high impact
"Show page that requires new data."	"Show page that requires new terminal data."	low impact
"Click on Edit button"	"Click on the Edit button"	low impact

readability and fixing typos. During our classification, we found 399 *low impact* and 27 *high impact* edits for the SAFF system, and 92 *low* and 11 *high impact* for BZC. This result shows that use cases often evolve for structural improvement, which may not justify the great number of discarded test cases in MBT suites.

After that, for each edit (original and edited versions), we ran the distance functions using different configuration values and observed how they classified the edits compared to our manual validation. Figure 2 presents an overview of our study's procedure for a single project.

Figure 2: Study overview for a single subject.

4.4 Metrics

To help us evaluate the results, and answer our research questions, we used three of the most well-known metrics for checking binary classifications: *Precision*, which is the rate of relevant instances among the found ones; *Recall*, calculates the rate of relevant retrieved instances over the total of relevant instances; and *Accuracy*, which combines Precision and Recall. These metrics have been used in several software engineering empirical studies (e.g., [9, 12, 25]). Equations 1, 2 and 3 present those metrics, where TP refers to the number of cases a distance function classified an edit as low impact and the manual classification confirms it; TN refers to the number of matches regarding high impact edits; FP refers to when the automatic classification reports low impact edits when in fact high impact edits were found; and FN is when the automatic classification reports high impact when in fact should be low impact edits.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

4.5 Results and Discussion

To answer RQ1, we first divided our dataset of use case edits into two (low and high impact edits), according to our manual classification. Then, we ran the distance functions and plotted their results. Figures 3 and 4 show the box-plot visualization of this analysis considering found low (Figure 3) and high impacts (Figure 4). As we can see, most low impact edits, in fact, refer to low distance values (median lower than 0.1), for all distance functions. This result gives us evidence that low distance values can relate to low impact edits and, therefore, can be used for predicting low impact changes in MBT suites. On the other hand, we could not find a strong relationship between high impact edits and distance values. Therefore we can answer RQ1 stating that distance functions, in general, can be used to classify low impact.

RQ1: Can distance functions be used to classify the impact of edits in use case documents? Low impact edits are often related to lower distance values. Therefore, distance functions can be used for classifying low impact edits.

As for automatic classification we need to define an effective *impact threshold*, for each distance function, we run an exploratory to find the optimal configuration for using each function. By impact threshold we mean the distance value for classifying an edit as low or high impact. For instance, consider a defined impact threshold of x% to be used with function *f*. When analyzing an edit from a specification document, if *f* provides a value lower than *x*, we say the edit is *low impact*, otherwise it is *high impact*. Therefore, we design an study where, for each function, we vary the defined *impact threshold* and we observed it would impact Precision and Recall. Our goal with this analysis is to identify the more effective configuration for each function. We range the impact threshold between [0; 1].

To find the optimal value, we consider the interception point between the Precision and Recall curves, since it reflects a scenario with less mistaken classifications (false positives and false negatives). Figure 5 presents the analysis for Jaccard and highlights its best configuration (impact threshold of 0.33) – the green line

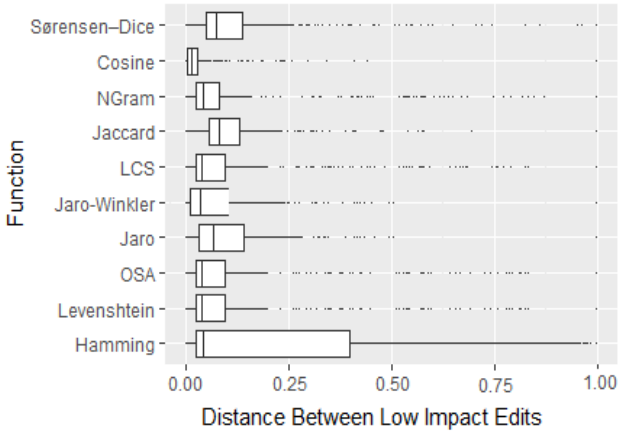

Figure 3: Box-plot for low impact distance values.

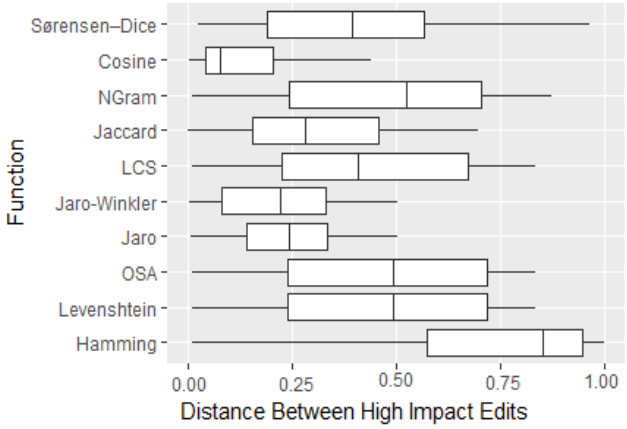

Figure 4: Box-plot for high impact distance values.

refers to Precision curve, the blue line to the Recall curve, and the red circle shows the point both curves meet. Figure 6 presents the analysis for all other functions.

Table 3 presents the optimal configuration for each function and the respective precision, recall, and accuracy values. These results reinforce our evidence to answer RQ1 since all functions presented accuracy values greater than 90%. Moreover, we can partially answer RQ2, since now we found, considering our dataset, the best configuration for each distance function. To complement our analysis, we went to investigate which function performed the best. First, we run proportion tests considering both the functions all at once and pair-to-pair. Our results show, with 95% of confidence, could not find any statistical differences among the functions. This means that distance function for automatic classification of edits impact is effective, regardless of the chosen function (RQ2). Therefore, in practice, one can decide which function to use based on convenience aspects (e.g., easier to implement, faster).

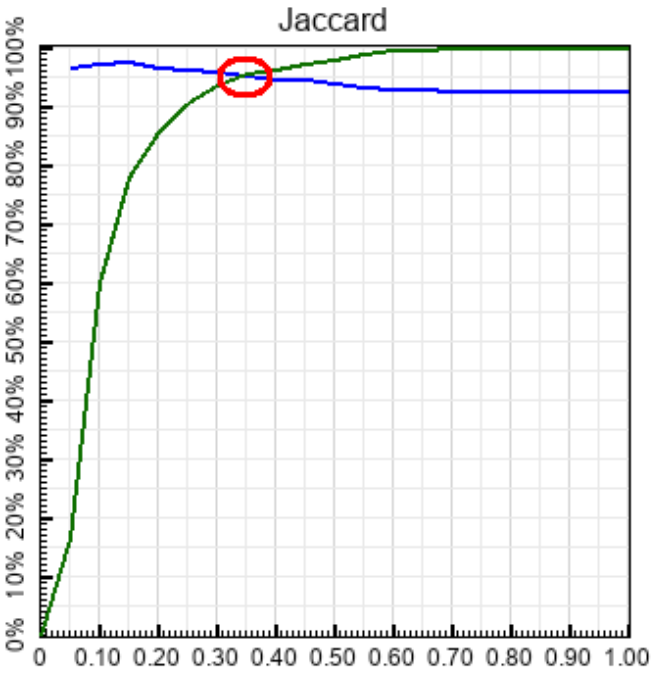

Figure 5: Best impact threshold for the Jaccard function.

RQ2: Which distance function presents the best results for classifying edits in use case documents? Statistically, all ten distance functions performed similarly when classifying edits from use case documents.

Table 3: Best configuration for each function and respective precision, recall and accuracy values.

Function	Impact Threshold	Precision	Recall	Accuracy
Hamming	0.91	94.59%	94.79%	90.15%
Levenshtein	0.59	95.22%	95.42%	91.31%
OSA	0.59	95.22%	95.42%	91.31%
Jaro	0.28	95.01%	95.21%	90.93%
Jaro-Winkler	0.25	95.21%	95.21%	91.12%
LCS	0.55	94.99%	94.79%	90.54%
Jaccard	0.33	95.22%	95.42%	91.31%
NGram	0.58	95.41%	95.21%	91.31%
Cosine	0.13	95%	95%	90.73%
Sørensen–Dice	0.47	94.99%	94.79%	90.54%

5 CASE STUDY

To reassure the conclusions presented in the previous section, and to provide a more general analysis, we ran new studies considering a different object, TCOM. TCOM is an industrial software also developed in the context of our cooperation with the Ingenico Brasil Ltda that controls the execution and manages testing results of a

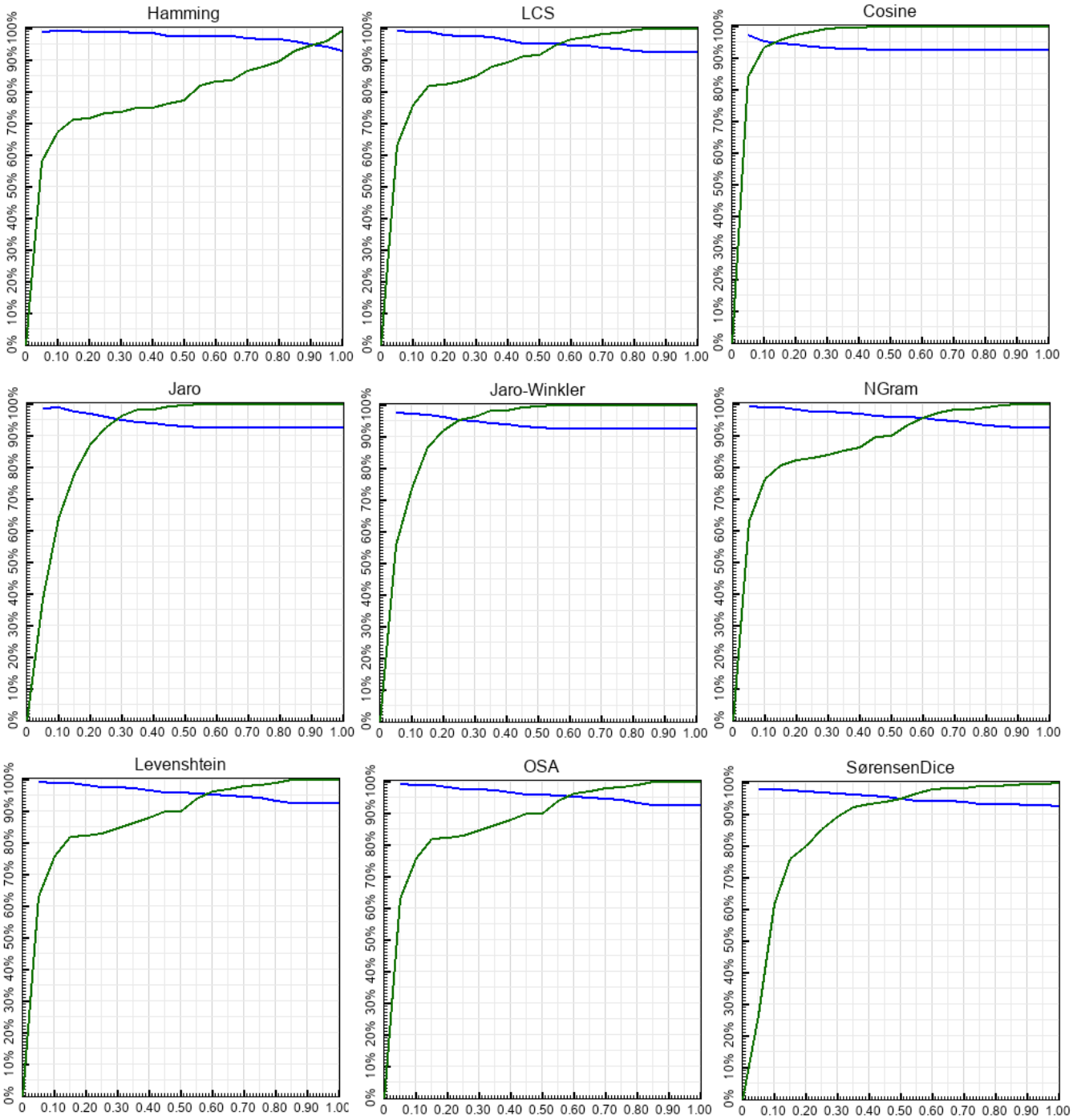

Figure 6: Exploratory study for precision and recall per distance function.

series of hardware parts. It is important to highlight that a different team ran this project, but they used a similar environment: CLARET use cases for specification and generated MBT suites. The team also reported similar problems concerning volatile requirements, and frequent test case discards.

In the first study, similar to the procedure applied in Section 4.3, we mined TCOM's repository and collected all versions of its use case documents and their edits. Table 4 summarizes the collected data from TCOM. Then, we manually classified all edits between

Table 4: Summary of the artifacts for the TCOM system.

	#Use Cases	#Versions	#Edits
TCOM	7	32	133

low and high impact to serve as validation for the automatic classification. Finally, we run all distance functions considering the optimal *impact thresholds* (Table 3 - second column) and calculated Precision, Recall and Accuracy for each configuration (Table 5).

Table 5: TCom - Evaluating the use of the found impact threshold for each function and respective precision, recall and accuracy values.

Function	Impact Threshold	Precision	Recall	Accuracy
Hamming	0.91	87.59%	94%	84.96%
Levenshtein	0.59	87.85%	94%	85.71%
OSA	0.59	87.85%	94%	85.71%
Jaro	0.28	89.52%	94.00%	87.22%
Jaro-Winkler	0.25	94.00%	89.52%	87.22%
LCS	0.55	89.62%	95%	87.97%
Jaccard	0.33	89.52%	94%	87.22%
NGram	0.58	87.85%	94%	85.71%
Cosine	0.13	88.68%	94%	86.47%
Sørensen–Dice	0.47	88.68%	94%	86.47%

As we can see, the found impact thresholds presented high precision, recall, and accuracy values when used in a different system and context (all above 84%). This result gives as evidence that, distance functions are effective for automatic classification of edits (RQ1) and that the found impact thresholds performed well for a different experimental object (RQ2).

In a second moment, we run a case study to evaluate how our approach (using distance functions for automatic classification) can help reducing test discards. For that, we also used TCOM's CLARET artifacts, and we defined the following research question:

- RQ3: Can distance function be used for reducing the discard of MBT tests?

To answer RQ3, we considered TCOM's MBT test cases generated from its CLARET files. Since all distance functions behave similarly (Section 4.5), in this case study we used only Levenshtein's function to automatically classify the edits and to check the impact of those edits in the tests. In a common scenario, which we want to avoid, any test case that contains an updated step would be discarded. Therefore, in the context of our study, we used the following strategy *"only test cases that contain high impact edits should be discarded, while test cases with low impact edits are likely to be reused with no or little updating"*. The rationale behind this decision is that low impact edits often imply on little to no changes to the system behavior. Considering system level black-box test suites (as the ones from the projects used in our study), those tests should be easily reused. We used this strategy and we first applied Oliveira's et al.'s classification [8] that divided TCOM's tests among three sets: *obsolete* – test cases that include impacted steps; *reusable* – test cases that were not impacted by the edits; and *new* – test cases that include new steps.

Table 6: Example of a low impacted test case.

... step 1: operator presses the terminal approving button. step 2: system goes back to the terminal profiling screen. ...	... step 1: operator presses the terminal approving button. step 2: system redirects the terminal to its profiling screen. ...

Table 7: Example of a highly impacted test case.

... step 3: operator presses camera icon. step 4: system redirects to photo capture screen. ... step 9: operator takes a picture and presses the Back button. ...	... step 3: operator selects a testing plan. step 4: system redirects to the screen that shows the selected tests. ... step 9: operator sets a score and press Ok. ...

A total of 1477 MBT test cases were collected from TCOM's, where 333 were found *new* (23%), 724 *obsolete* (49%), and 420 *reusable* (28%). This data reinforces Silva et al.'s [32] conclusions showing that, in an agile context, most of an MBT test suite became obsolete quite fast.

In a common scenario, all "obsolete" test cases (49%) would be discarded throughout the development cycles. To cope with this problem, we ran our automatic analysis and we reclassify the 724 obsolete test cases among *low impacted* – test cases that include unchanged steps and updated steps classified by our strategy as "low impact"; *highly impacted* – test cases that include unchanged steps and "high impact" steps; and *mixed*, test cases that include at least one "high impact" step and at least one "low impact" step. From this analysis, 109 test cases were *low impacted*. Although this number seems low (15%), those test cases would be wrongly discarded when in fact they could be easily turned into reusable. For instance, Table 6 shows a simplified version of a "low impacted" test case from TCOM. As we can see, only step 2 was updated to better phrase a system response. This was an update for improving specification readability, but it does not impact on the system's behavior.

The remaining test cases were classified as follows: 196 "highly impacted" (27%), and 419 "mixed" (58%). Table 7 and 8 show examples of highly impacted and mixed tests, respectively. In Table 7, we can see that steps 3, 4, and 9 were drastically changed, which infer to a test case that requires much effort to turn it into reusable. On the other hand, the test in Table 8, we have both an edit for fixing a typo (step 2) and an edit with a requirement change (step 7).

Table 8: Example of a mixed test case.

... step 2: operator presses button CANCEL to mark there is no occurrence description. ... step 7: operator presses the button SEND. ...	... step 2: operator presses the button CANCEL to mark there is no occurrence description. ... step 7: operator takes a picture of the hardware. ...

Table 9: Confusion Matrix.

	Predicted			
	Low	High	Mixed	
Actual Low	69	4	21	94
Actual High	3	37	27	67
Actual Mixed	37	155	371	563
	109	196	419	724

To check whether our classification was in fact effective we present its confusion matrix (Table 9). In general, our classification was 66% effective (Precision). A smaller precision was in fact already expected, when compared to the precision classification from Section 4, since here we consider all edits that might affect a test case, while in Section 4 we analyzed and classified each edit individually. However, we can see, our classification was highly effective for *low* and *highly impacted* test cases, and most mistaken classification was relate to the *mixed* one (test that combine low and high impact edits). Those were, in fact, test cases that were affected in a great deal by different types of use case editions.

Back to our strategy, we believe that only *highly impacted* test cases indicate test cases likely to be discarded since they refer to test cases that would require much effort to be updated. Therefore, 15% of the first "obsolete" set would be reclassified as reusable. Moreover, we believe this rate can get higher when we analyze the *mixed* tests. A mixed test combines low and high impact edits. However, when we manually analyzed those cases, we found several examples where, although *high impact* edits were found, most test case impacts were related to *low impact* edits. For instance, there was a test case composed of 104 execution steps where only one of those steps needed revision due to a *high impact* use case edit, while the number of *low impact* edits was seven. In a practical scenario, although we still classify it as a mixed test case, we would say the impact of the edits was still quite small, which may indicate a manageable revision effort. Thus, we state that mixed tests need better analysis before discarding. The same approach may also work for *highly impacted* tests when related to a low number of edits.

Finally, we can answer RQ3 by saying that an automatic classification using distance functions can, in fact, reduce the number of discarded test cases by at least 15%. However, this rate tent to be higher when we consider *mixed* tests.

RQ3: Can distance function be used for reducing the discard of MBT tests? The use of distance functions can reduce the number of discarded test cases by at least 15%.

6 THREATS TO VALIDITY

Most of the threats for validity to the drew conclusions refer to the number of projects, use cases and test cases used in our experimental studies. Those numbers were limited to the artifacts created in the context of the selected projects. Therefore, our results cannot be generalized beyond the three projects (SAFF, BZC, and TCOM). However, it is important to highlight that all used artifacts are from real industrial systems from different contexts.

As for conclusion validity, our studies deal with a limited dataset. Again, since we chose to work with real, instead of artificial artifacts, the data available for analysis were limited. However, all used data were validated by the team engineers and by the authors.

Regarding internal validity, we collected the changed set from the project's repositories, and we manually classify each change according to its impact. This manual validation was performed by at least two of the authors and, when needed, the project's members were consulted. Moreover, we reused open-source implementations of the distance functions[4]. These implementations were also validated by the first author.

7 RELATED WORK

Guldali and Mlynarski [15] discuss how MBT can support agile development. For that, they emphasize the need for automation aiming that MBT artifacts can be manageable and with little effort to apply.

Silva et al. [32] gave the first steps on investigating issues related to MBT in the context of agile development. They ran an empirical study on the evolution of specification models and their impact on generated MBT suites and found that 86% of a test suite is often impacted, however, more than half those tests were impacted due to syntactic model edits (low impact). Those findings greatly motivated this current research. Based on them, here we propose the use of distance functions for automatically classify the test cases that are little impacted and could be reused.

Oliveira Neto et al. [8] discuss a series of problems related to keeping MBT suites updated during software evolution. To cope with this problem, they propose a test selection approach that uses test case similarity as input when collecting test cases that focus on recently applied changes. Oliveira Neto et al.'s approach refer as obsolete all test cases that are impacted in any way by an edit in the requirement model. However, as our study found, a great part of those tests can be little impacted, and could be easily reused, avoiding the discard of testing artifacts.

The test case discard problem is not restricted to CLARET artifacts. Other similar cases are discussed in the literature (e.g.,[8, 26]). Moreover, this problem is even greater with MBT test cases derived from artifacts that use non-controlled language [28].

[4]https://github.com/luozhouyang/python-string-similarity

Other works also deal with test case evolution (e.g.,[22, 28]). They discuss the problem and/or propose strategies for updating the testing code. Those strategies do not apply to our context, since we work with MBT test suite evolution generated from use case models.

The use of distance functions in the context of software engineering is not new. Several works have used distance functions in different scenarios (e.g., [21, 27, 30]). For instance, Runkler and Bezdek [30] use the Levenshtein function for automatically extract keywords from documents. In the context of MBT, Coutinho et al. [4] investigated the effectiveness of a series of distance functions when used combined with strategies for suite reduction based on similarity. Although in a different context, their results go according to ours where all distance functions performed in a similar way.

8 CONCLUDING REMARKS

In this paper, we describe a series of empirical studies ran on industrial systems intending to evaluate the use of distance functions for automatically classify the impact of edits in use case files. Our results showed that distance functions are effective to identify low impact editions.

We also found that low impact editions often refer to test cases that can be easily updated without any effort. We believe those results can help testers to better work with MBT artifacts in the context of software evolution and avoid the discard of test cases.

As future work, we plan to expand our study with a broader set of systems. We also consider developing a tool that, using distance functions, can help testers to identify and update low impact test cases. Finally, we plan to investigate the use of different approaches (e.g., machine learning) to help testers when updating highly impacted test cases and reuse history data of obsolete MBT test cases.

ACKNOWLEDGMENTS

This research was partially supported by a cooperation between UFCG and two companies Viceri Solution LTDA and Ingenico do Brasil LTDA, the latter stimulated by the Brazilian Informatics Law n. 8.248, 1991. Second and fourth authors are supported by National Council for Scientific and Technological Development (CNPq)/Brazil (processes 429250/2018-5 and 315057/2018-1). First and third authors were supported by UFCG/CNPq and CAPES, respectively.

REFERENCES

[1] Kent Beck and Erich Gamma. 2000. *Extreme programming explained: embrace change*. addison-wesley professional.

[2] Emanuela Cartaxo, Wilkerson Andrade, Francisco de Oliveira Neto, and Patrícia Machado. 2008. LTS-BT: A tool to generate and select functional test cases for embedded systems. In -. 1540–1544. https://doi.org/10.1145/1363686.1364045

[3] William W Cohen, Pradeep Ravikumar, Stephen E Fienberg, et al. 2003. A Comparison of String Distance Metrics for Name-Matching Tasks.. In *IIWeb*, Vol. 2003. 73–78.

[4] Ana Emília Victor Barbosa Coutinho, Emanuela Gadelha Cartaxo, and Patrícia Duarte de Lima Machado. 2016. Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing. *Software Quality Journal* 24, 2 (2016), 407–445.

[5] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. 1999. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 285–294. https://doi.org/10.1145/302405.302640

[6] Fred J. Damerau. 1964. A Technique for Computer Detection and Correction of Spelling Errors. *Commun. ACM* 7, 3 (March 1964), 171–176. https://doi.org/10.1145/363958.363994

[7] Xavier De Coster, Charles De Groote, Arnaud Destiné, Pierre Deville, Laurent Lamouline, Thibault Leruitte, and Vincent Nuttin. 1. Mahalanobis distance, Jaro-Winkler distance and nDollar in UsiGesture. (1).

[8] Francisco G de Oliveira Neto, Richard Torkar, and Patrícia DL Machado. 2016. Full modification coverage through automatic similarity-based test case selection. *Information and Software Technology* 80 (2016), 124–137.

[9] Karim O Elish and Mahmoud O Elish. 2008. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software* 81, 5 (2008), 649–660.

[10] Richard W Hamming. 1950. Error detecting and error correcting codes. *The Bell system technical journal* 29, 2 (1950), 147–160.

[11] Tae Sik Han, Seung-Kyu Ko, and Jaewoo Kang. 2007. Efficient subsequence matching using the longest common subsequence with a dual match index. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer, 585–600.

[12] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Sundaram. 2005. Text mining for software engineering: how analyst feedback impacts final results. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 1–5.

[13] Anna Huang. 2008. Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, Vol. 4. 9–56.

[14] J. Itkonen, M. V. Mantyla, and C. Lassenius. 2009. How do testers do it? An exploratory study on manual testing practices. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. 494–497. https://doi.org/10.1109/ESEM.2009.5314240

[15] Mika Katara and Antti Kervinen. 2006. Making model-based testing more agile: a use case driven approach. In *Haifa Verification Conference*. Springer, 219–234.

[16] Grzegorz Kondrak. 2005. N-Gram Similarity and Distance.. In *SPIRE* (2005-10-31) *(Lecture Notes in Computer Science)*, Mariano P. Consens and Gonzalo Navarro (Eds.), Vol. 3772. Springer, 115–126. http://dblp.uni-trier.de/db/conf/spire/spire2005.html#Kondrak05

[17] Joseph B Kruskal. 1983. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM review* 25, 2 (1983), 201–237.

[18] Divya Kumar and KK Mishra. 2016. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science* 79 (2016), 8–15.

[19] Vladimir Iosifovich Levenshtein. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 10, 8 (feb 1966), 707–710. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[20] Jiaheng Lu, Chunbin Lin, Wei Wang, Chen Li, and Haiyong Wang. 2013. String similarity measures and joins with synonyms. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 373–384. https://doi.org/10.1145/2463676.2465313

[21] Andre Hasudungan Lubis, Ali Ikhwan, and Phak Len Eh Kan. 2018. Combination of levenshtein distance and rabin-karp to improve the accuracy of document equivalence level. *International Journal of Engineering & Technology* 7, 2.27 (2018), 17–21.

[22] Mehdi Mirzaaghaei. 2011. Automatic test suite evolution. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 396–399.

[23] Dalton N. Jorge, Patrícia Machado, Everton L. G. Alves, and Wilkerson Andrade. 2017. CLARET - Central Artifact for Requirements Engineering and Model-Based Testing. / In Proceedings of the 24th Tools Session / 8th Brazilian Conference on Software: Theory and Practice. Fortaleza, CE, BR, 41–48.. In -. https://doi.org/10.1109/RE.2018.00041

[24] Dalton N. Jorge, Patrícia Machado, Everton L. G. Alves, and Wilkerson Andrade. 2018. Integrating Requirements Specification and Model-Based Testing in Agile Development. 336–346. https://doi.org/10.1109/RE.2018.00041

[25] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. 2008. The influence of organizational structure on software quality. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 521–530.

[26] Sidney Nogueira, Emanuela Cartaxo, Dante Torres, Eduardo Aranha, and Rafael Marques. 2007. Model based test generation: An industrial experience. In *1st Brazilian Workshop on Systematic and Automated Software Testing*.

[27] Teruo Okuda, Eiichi Tanaka, and Tamotsu Kasai. 1976. A method for the correction of garbled words based on the Levenshtein metric. *IEEE Trans. Comput.* 100, 2 (1976), 172–178.

[28] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 33.

[29] Roger Pressman. 2005. *Software Engineering: A Practitioner's Approach* (6 ed.). McGraw-Hill, Inc., New York, NY, USA.

[30] Thomas A Runkler and James C Bezdek. 2000. Automatic keyword extraction with relational clustering and Levenshtein distances. In *Ninth IEEE International Conference on Fuzzy Systems. FUZZ-IEEE 2000 (Cat. No. 00CH37063)*, Vol. 2. IEEE,

636–640.

[31] Todd Sedano, Paul Ralph, and Cécile Péraire. 2017. Software Development Waste. In -.

[32] Anderson G.F. Silva, Wilkerson L. Andrade, and Everton L.G. Alves. 2018. A Study on the Impact of Model Evolution in MBT Suites. In *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing (SAST '18)*. ACM, New York, NY, USA, 49–56. https://doi.org/10.1145/3266003.3266009

[33] Thorvald Sørensen. 1948. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons. *Biol. Skr.* 5 (1948), 1–34.

[34] Jeff Sutherland and JJ Sutherland. 2014. *Scrum: the art of doing twice the work in half the time.* Currency.

[35] Jan Tretmans. 2008. Model based testing with labelled transition systems. In *Formal methods and testing*. Springer, 1–38.

[36] Mark Utting and Bruno Legeard. 2007. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.