RI11014 5 September 2011 Computer Science

IBM Research Report

Automating Test Automation

Suresh Thummalapenta
IBM Research – India

Saurabh Sinha
IBM Research – India

Debdoot Mukherjee
IBM Research – India

Satish Chandra
IBM T.J. Watson Research Center

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

Abstract

Mention "test case", and it conjures up image of a script or a program that exercises a system under test. In industrial practice, however, test cases often start out as steps described in natural language. These are essentially directions a human tester needs to follow to interact with an application, exercising a given scenario. Since tests need to be executed repeatedly, such *manual* tests then have to go through *test automation* to create scripts or programs out of them. Test automation can be expensive in programmer time.

We describe a technique to automate test automation. The input to our technique is a sequence of steps written in natural language, and the output is a sequence of procedure calls with accompanying parameters that can drive the application without human intervention. The technique is based on looking at the natural language test steps as consisting of *segments* that describe actions on targets, except that there can be ambiguity in the action itself, in the order in which segments occur, and in the specification of the target of the action. The technique resolves this ambiguity by backtracking, until it can synthesize a successful sequence of calls.

Our technique has been surprisingly effective in automating manual test cases as written by professional testers. In our main experiment, in which we asked professional testers to write test cases for two web applications, the technique successfully automated 85% of the test cases end to end.

1 Introduction

What is Test Automation Developers write unit test cases in the form of little programs, typically in a framework such as JUnit [6]. Such a test suite can be run at the desired frequency, *e.g.,* before each code commit, once daily, etc., without any manual intervention. Fixing regressions in the unit test suite is often the first line of defence in ensuring code quality.

The story for system tests, or end-to-end tests for a complete application is different. By applications, we mean web applications that reside on a server and are accessed via a web browser. System tests are typically written by testers, not developers themselves. System tests start with identifying test scenarios from requirements and use cases, and creating test cases from the scenarios. A test case is just a sequence of test steps written in natural language; indeed they are intended for use by a human (the tester), who performs a test manually through the application's GUI. Figure 1 shows a sample test case written by a professional tester.

Manual test cases do not have the benefit of repeatable and efficient test execution. Therefore, automatically executable test scripts need to be created from manual test cases, so the advantages of repeatable and efficient test execution are clawed back.

Test automation, then, is the task of creating a mechanically interpretable representation of a manual test case. Such a representation could be a program in a general-purpose programming or scripting language (*e.g.,* Java or VBScript) . Alternatively, automation could be realized by formulating a test case as a precise sequence of "keywords" or calls to executable subroutines, along with the relevant arguments. This sequence can then be executed automatically by a test driver. Figure 2 shows this form for test case of Figure 1. These approaches entail different degrees of control over interaction with the application, and require different degrees of technical skill in carrying out test automation; writing in Java being more complex albeit more general that using keyword driven automation. Given that the number of manual test cases for an application can be in the thousands, test automation is an expensive part of the software testing effort. It is useful to look for ways to reduce this cost.

Alternatives to Test Automation It is natural to ask why manual test cases are so prevalent, if they are bound to have a latent cost of test automation. Why not write automated test cases to begin with, e.g. in the scripting language of a testing tool? Because system test cases are most commonly created by non-developers, who may not possess the technical skills required for "coding" test cases in a precise notation, the natural-language expression of system tests is the one used most often in practice. Natural language incurs no training overhead.

An alternative is to write manual test cases in a specialized test-scripting language that has the feel of natural language but allows mechanical interpretation. ClearScript [14] is an example of such a language. An example of how the test case of Figure 1 would look like in ClearScript is shown in Figure 3. ClearScript permits some elision of rigor of a full programming language, but is still fairly structured compared to natural language. Until such a language sees broader adoption, organizations will continue to have to pay the cost of test automation.

Another alternative is to use capture-replay tools, which record the tester's actions on the user interface to create an automatically executable test script. However, the task of demonstrating the manual test cases on the GUI still costs

```
1: Launch the application through the link  http://godel.in.ibm.com:8080/online-bookstore/Default.jsp
2: Enter the intended book search name as "MySQL" at the "Title" Edit field and select "Category"
     as "All" by Drop  down list and then Click "Search" Button
3: Select a title from the list of all Search Results  displayed and then click either on the image
     of the book or on Name of the Book
4: Enter login and password, and click login
5: Enter the Quantity "1" and Click on "Add to Shopping Cart" Button
6: Verify "User Information" and Following "Item"
     Details of Selected Book
       Details
       Order \#
       Item Price
       Quantity
       Total
```

Figure 1: An example manual test case for the bookstore application written by a professional tester.

```
 1: <goto,http://godel.in.ibm.com:8080/online-
     bookstore/default.jsp,>
 2: <enter,title,MySQL>
 3: <select,category,all>
 4: <click,search,>
 5: <select,title,>
 6: <enter,login,guest>
 7: <enter,password,guest>
 8: <click,login,>
 9: <enter,quantity,1>
10: <click,add to shopping cart,>
```

Figure 2: A keyword-based script, consisting of triplets of action, the target of the action, and any relevant data.

```
 1: go to "http://godel.in.ibm.com:8080/online-
      bookstore/Default.jsp"
 2: enter "MySQL" into the "Title" textbox
 3: select "All" from the "Category" listbox
 4: click the "Search" button
 5: click the "MySQL & PHP From Scratch" link
 6: enter "guest" into the "Login" textbox
 7: enter your password into the "Password" textbox
 8: click the "Login" button
 9: enter "1" into the "Quantity" textbox
10: click the "Add to Shopping Cart" button
```

Figure 3: The ClearScript [14] representation of the manual test case.

human time, and second, the generated scripts can be difficult to maintain—*e.g.,* the scripts contain monolithic code with no modularization.

Figure 4 illustrates the spectrum of the levels of formalism that can be followed in creating system test cases. At the one end of the spectrum, tests are stated in a "stylized" natural language; stylized, because as the reader can see in Figure 1, test cases are written in a certain pattern. At the other end, system tests are expressed in a programming language. In between these extremes, there are two approaches with intermediate degrees of formality: specialized test-scripting languages that attempt to retain the flavor of natural-language expression, and a sequence of keywords or names of executable subroutines, where for each call, the user-interface element and data values need to be identified. The keyword approach can also have a slightly less formal variant, where a keyword is a natural-language expression [11].

Automating Test Automation The motivation for our work is to have non-programmers not lose the comfort level that they would have with the degree of informality, flexibility, ambiguity (without exactness) that is inherent in natural-language expressions; and that would be closer to their thought processes than the rigor demanded by even the specialized scripting languages. Thus, our goal is to improve the efficiency of automating the natural-language manual tests—by *automating the automation task*.

In this paper, we address the problem of creating automated test scripts from manual test cases, specifically, in the context of keyword-driven automation. Given a manual test-case step s stated in the English language, our goal is to extract automatically a tuple $S = (a, t, d)$, consisting of the keyword or action a for s, the target user-interface element t to which the action applies, the data value d (if any) mentioned in s. We have built a tool to accomplish this goal. Figure 2 is, in fact, the result of our tool on the manual test case of Figure 1; note that Figure 2 is mechanically interpretable.

We are not proposing to understand natural language, at least not in advanced ways demonstrated recently in Watson [21]. Rather, we rely on the observation that the style in which testers write manual steps have a very predictable structure consisting of *segments*, where a segment is an instruction for the tester to perform an action on a user interface. Examination of over a thousand test steps taken from manual tests of real applications supports this observation. Moreover, the tests have a restricted vocabulary (*e.g.,* consisting of nouns that are pertinent for the application being tested and verbs that specify actions on user-interface elements); this is also borne out by our experiments (see Section 4.3).

The problem is not trivial, however. Although the manual test cases are written in a stylized form, the ambiguities

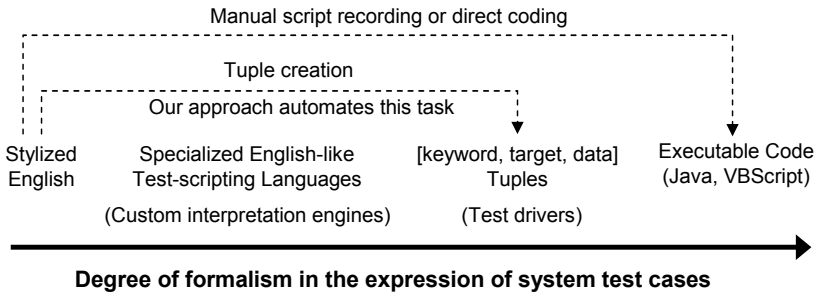

Figure 4: The spectrum of approaches for specifying system tests in increasing order of the degree of formalism and precision.

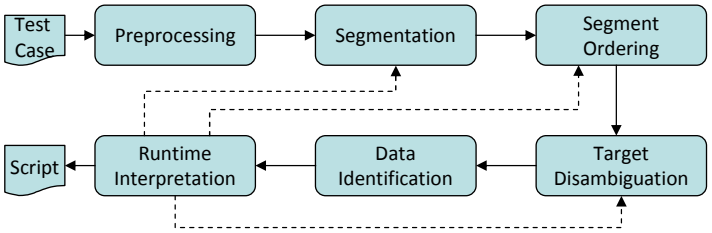

Figure 5: Overview of our approach.

inherent in natural language instructions, which may pose no challenges for human interpretation, can make mechanical interpretation difficult. Section 2 explains in detail the challenges of interpreting even stylized tester's English. As a relatively simple example, consider step 3 in Figure 1. At first read, it would appear that there are two segments of imperative commands, `select` and `click`, but there is only one action to be done at the browser.

There are many instances of such, and more complex, cases of ambiguity that we found by examining a large corpus of real manual test cases. There can be ambiguity in the action itself, in the order in which segments occur, and in the specification of the target of the action. Moreover, the ambiguities and the stylized form of expression can vary significantly from one application to the other. Although application-specific rules and heuristics are required to handle variations, a general approach that, to some extent, is variation-agnostic would be very valuable. In this paper, we present such an approach.

We can look at the problem of automating test automation as an instance of program synthesis [4] where the goal is to discover a program that realizes user intent. In our case, the user intent is stated in stylized natural language, as a sequence of manual test steps, and the goal is to synthesize a mechanically interpretable test script. Our work is also an instance of "end-user" programming [1], where the intent is to bridge the gap between natural-language and programming-language expressions, and lower the barriers to programming for end-users and novice programmers.

Our Approach The key idea underlying our approach is that the presence of ambiguities gives rise to alternative interpretations of a test action. Instead of resolving the ambiguities up-front, we can proceed by exploring an alternative until either the entire test case can be interpreted or the analysis reaches a point beyond which it cannot proceed. In the latter case, the analysis *backtracks* to the last decision point, and explores a different alternative. In this manner, the approach explores multiple potential flows, with the assumption that the selection of an incorrect alternative would cause the analysis to reach a dead-end eventually. The feature that makes our approach especially powerful is that, in exploring a flow, it actually determines whether the inferred tuple is executable by using the application under test as an oracle. Thus, if our tool terminates by computing a complete sequence of tuples, the tuples are guaranteed to execute.

Figure 5 presents an overview of our backtracking-based automation technique—it shows the different phases of the analysis. The first phase, Preprocessing, takes as input a manual test case (which could be specified in a plain-text document, a spreadsheet, etc.) and parses the test case to identify the individual test steps. The second phase, Segmentation, analyzes the test steps to identify the segments. The next phase orders the segments. The fourth phase identifies the potential targets for each segment. Each of the segmentation, ordering, and target-disambiguation phases

can encounter ambiguities, which result in decision points with multiple flows to be explored. The fifth phase tries to identify the data values, if any, that are mentioned in a segment. Finally, the technique attempts to execute the computed (a, t, d) tuple on the browser. If it is unable to execute the tuple, it backtracks to the last decision point, which may have occurred during segmentation, segment ordering, or target disambiguation.

Overview of Results We implemented the approach and conducted an evaluation of several aspects of it. Our first study measures the success of the approach in automatically creating keyword-based representation (as in Figure 2) for a suite of 23 end-to-end manual test cases for two open-source web applications. These tests were written by professional testers, who were unfamiliar with our approach or the tool that implements the approach. We were able to fully automate 19 out of these 23 test cases. In the remaining 4, the approach encountered some step that it could not automate; human intervention would be required to get the tool past it.

In a second study, we studied the success of segmentation and correct interpretation of each segment as an (a, t, d) tuple in a large test corpus containing over a thousand test steps. This corpus is proprietary, and also, the actual application was not available to us. However, we wanted to validate segment identification over a larger test corpus than the one used in the first study. The segmentation study showed that, in real test corpuses, about half of the individual test steps contained multiple segments, and across all these segments (which numbered well over a thousand), the tool was able to correctly reconstruct over 80% of them.

Contributions The key contributions of the work are the following:

- We present a novel and highly effective approach, based on backtracking and exploration of alternative flows, for automating manual tests in the context of keyword-driven automation.
- We present a detailed evaluation of our approach on two publicly available web applications, on manual tests created by *professional* testers. We also evaluate the effectiveness of segmentation on a larger corpus of manual test cases taken from a proprietary source.

The rest of the paper is organized as follows. Section 2 describes the challenges that manual test cases present to our approach. Section 3 describes the approach in detail. Section 4 presents a detailed evaluation of our approach. Section 5 presents related work.

2 Challenges in Interpreting Manual Tests

Figure 1 presents a sample manual test case for an online bookstore application written by a professional tester.[1] We use this sample as the running example to illustrate the challenges involved in mechanically interpreting manual test cases, and describe our approach.

To facilitate the discussion, we introduce some terminology. A *manual test case* is stated in natural language and consists of a sequence of *test steps*. A *segment* is a phrase in a test step that contains one verb (corresponding to a user-interface action) and one or more nouns (representing the user-interface elements on which the action is performed). For example, the manual test case in Figure 1 consists of six test steps. An example of a segment is `Enter login and password` in test step 4.

The goal of our approach is to infer an automatically interpretable *action-target-data* (*ATD*) tuple, which is a triple (a, t, d) consisting of an action, a target user-interface element, and a data value. Figure 2 illustrates the *ATD* tuples computed by our approach for the sample manual test case. This example illustrates several challenges for automated interpretation.

Identifying valid segments A manual test step can consist of multiple segments combined using coordinating conjunctions, such as "and", which can complicate interpretation. For example, consider step 4. This step consists of two segments: `Enter login and password` and `click login`. However, simply splitting a manual test step based on conjunctions would create the invalid segment `password`, which does not contain any verb. Our approach considers all possible segmentations based on coordinating conjunctions, explores each alternative, and eventually arrives at the correct segmentation, illustrated by the *ATD* tuples 6–8 in Figure 2.

[1] The bookstore application, available at `www.gotocode.com`, is one of the subjects used in our empirical studies (Section 4).

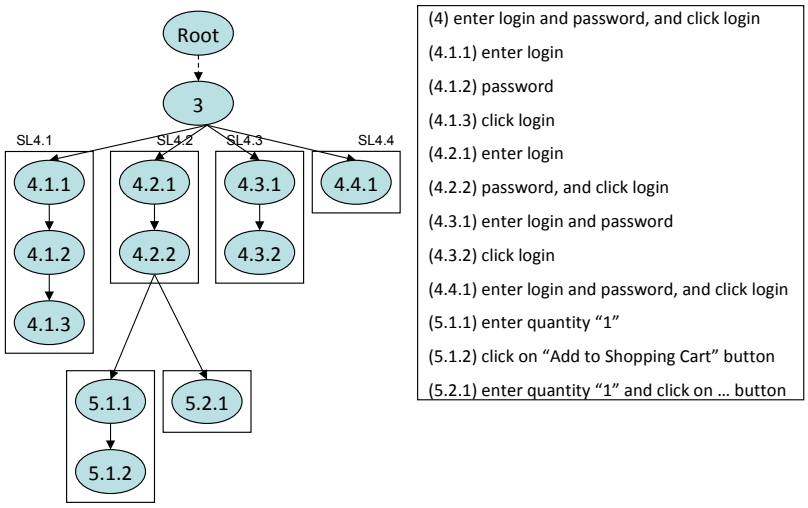

Figure 6: Illustrative example for our backtracking technique.

Identifying irrelevant tuples A test step can contain valid segments that do not correspond to actions on the user interface. Such a segment may specify a verification step or it may be totally irrelevant. Because of the informality of natural-language expressions, irrelevant segments that are meant to be informative, or provide a clarification, are often present in manual tests. For example, consider step 3, which seems to suggest two actions, indicated by the verbs `Select` and `click`. However, the tester needs to perform only one action: select from a list of books by clicking on a link or an image. Our approach explores both segments and, eventually, discards one to compute only one *ATD* tuple—tuple 5 in Figure 2.

Computing tuple order In scenarios where a test step contains multiple actions, the actions may need to be ordered for executing the test step successfully. The test in Figure 1 does not contain such a step. However, consider the following test step, written by a professional tester, for another one of our empirical subjects: `Select priorities from the administration tab`. To execute this step, the administration tab must be selected before `priorities` can be selected. Our approach is able to compute two *ATD* tuples in the correct order

```
<goto,administration tab,>
<select,priorities,>
```

Disambiguating targets In some cases, there can be multiple targets for a given action in a segment. This happens in scenarios where either the tester does not mention the target properly or the application under test includes multiple targets with the same label. Consider again step 3 of the manual test. The application web page on which the step is applicable can contain multiple user-interface elements labeled "Title": a text box, and zero or more HTML links (depending on the results of a book search). Thus, the segment `select a title` poses target ambiguity, which a mechanical interpretation needs to address, as our backtracking-based approach does. Our approach explores all potential targets labeled "Title" and, for each target, it attempts to interpret and execute the next test step (step 4), which is applicable on the login page only. While exploring the text-box target, our tool does not reach the login page and, therefore, is unable to execute step 4. Therefore, it backtracks and explores a link labeled "Title". In this traversal, it is able to complete step 4 successfully.

3 Our Technique

Algorithms 1, 2, and 3 show key algorithms of our backtracking technique. The core idea of our approach is to take an imprecise test step in English and generate various candidate precise *ATD* tuples that can be executed by the runtime. Whenever a candidate *ATD* tuple cannot be generated or executed, our technique automatically backtracks to other possible candidates. We next explain each algorithm in detail using the manual test shown in Figure 1 and describe the scenarios where our technique backtracks to other possible candidates.

Algorithm 1 $HandleTestCase(TCase, index)$

Require: A test case $TCase$, $index$
Ensure: SUCCESS or FAIL

```
1: if index > TCase.length then
2:     return SUCCESS
3: end if
4: TestStep = TCase[index]
5: //Compute possible segments
6: List<SegList> seglists = ComputeSegLists(TestStep)
7: for all SegList sl ∈ seglists do
8:     OrderSegments(sl)
9:     ret = HandleSegList(sl, 0)
10:    if ret == SUCCESS then
11:        return SUCCESS
12:    end if
13:    //Backtrack
14: end for
15: return FAIL
```

3.1 Algorithm 1: HandleTestCase

The *HandleTestCase* algorithm accepts a test case and analyzes each step of the test case recursively. In particular, for each test step, *HandleTestCase* computes all candidate lists of segments, referred to as SegLists, using *ComputeSegLists* (Line 6). Given a test step, *ComputeSegLists* splits the test step into various SegLists based on conjunctions such as `and` and `from`. If the test step includes n conjunctions, *ComputeSegLists* generates 2^n SegLists, representing all possible combinations of segments. For example, consider Step 4 of the Figure 1. This step includes two `and` conjunctions. Therefore, *ComputeSegLists* generates 4 SegLists. Figure 6 shows all four SegLists ($SL4.1$, $SL4.2$, $SL4.3$, and $SL4.4$) for Step 4. Here, SegList SL1 includes three segments 4.1.1, 4.1.2, and 4.1.3. We use the notation "$a.b.c$", where "a", "b", and "c" represent the test step index, SegList index, and segment index, respectively. Among these four segment lists, SegList $SL4.3$ is the correct segment list that helps in successful execution of Step 4. We next describe how our technique successfully executes $SL4.3$ among these four segment lists.

The *HandleTestCase* algorithm next analyzes each SegList. In particular, *HandleTestCase* uses *OrderSegments* (Line 8) to order segments within a SegList based on heuristics . Such an ordering is required when conjunctions such as `from` are used in the test step. For example, consider the test step `T1: select priorities from the administration tab`. We cannot execute the segment `select priorities` unless the `administration tab` is selected. To address this issue during segmentation, we identify special conjunctions such as `from` or `under` that require ordering among segments. We next introduce verb (`goto`) in the succeeding segment and sort those segments. The primary reason to introduce a verb is to satisfy our segment criteria that each segment should contain one verb. For the test step $T1$, our technique generates two ordered segments as follows:

```
(1) goto administration tab
(2) select priorities
```

After ordering segments, *HandleTestCase* invokes *HandleSegList* to analyze and execute a SegList. In case, *HandleSegList* fails, *HandleTestCase* automatically backtracks to other SegLists such as $SL4.2$ (Loop 7-14). We next explain the *HandleSegList* algorithm. Consider that $SL4.1$ is passed as argument to *HandleSegList*.

3.2 Algorithm 2: HandleSegList

The *HandleSegList* algorithm accepts a list of segments and extracts *ATD* tuples from each segment of the list. Before extracting *ATD* tuples, *HandleSegList* uses *DisambiguateNV* (Line 5) to disambiguate nouns and verbs within a segment. The primary reason is that often the same word can be used as both verb and noun. For example, consider the Segment 4.1.3: `click login`. Here, both click and login can be verbs or nouns, posing challenges in identifying actions and targets from segments. To address this issue, we explicitly generate multiple tagged segments, represented as `TSegment`, for each segment. A tagged segment is a segment where some words in the segment are explicitly tagged with a parts-of-speech (POS). For example, for the preceding segment, *DisambiguateNV* generates the following tagged segments.

Algorithm 2 $HandleSegList(TCase, tindex, seglist, segindex)$

Require: TCase $TCase$, index $tindex$,
Require: SegList $seglist$, index $segindex$
Ensure: SUCCESS or FAIL

```
1: if seglist.size() == segindex then
2:     //Proceed to next test step
3:     return HandleTestCase(TCase, tindex + 1)
4: end if
5: Segment seg = seglist[segindex]
6: List<TSegment> dseglist = DisambiguateNV(seg)
7: for all TSegment localseg ∈ dseglist do
8:     List<ATDTuple> adtList = ExtractATD(localseg)
9:     if adtList.size() == 0 then
10:        Continue
11:    end if
12:    ret = ExecuteATDList(seglist, segindex, adtList, 0)
13:    if ret == SUCCESS then
14:        return SUCCESS
15:    end if
16:    //Backtrack
17: end for
18: return FAIL
```

```
(4.1.3.1) click/VB login/VB
(4.1.3.2) click/VB login/NN
(4.1.3.3) click/NN login/VB
(4.1.3.4) click/NN login/NN
```

Here, VB and NN represent verb and noun, respectively. These four tagged segments describe all four possible combinations of the original segment. For each tagged segment, *HandleSegList* uses *ExtractATD* for extracting an *ATD* tuple from a tagged segment. Initially, *ExtractATD* checks whether the segment satisfies the segment criteria whether segment contains only one verb. For each segment that satisfies the preceding criteria, *ExtractATD* generates an *ATD* tuple. (Section 4.1 provides more details on how we extract *ATD* from a segment). There can be multiple *ATD* tuples for each tagged segment. *HandleSegList* next invokes *ExecuteATDList* to execute all *ATD* tuples generated for a tagged segment.

Backtracking We next explain how backtracking happens if *ExtractATD* fails to generate an *ATD* tuple for a segment. Consider that *HandleSegList* is invoked with SegList $SL4.1$. When *HandleSegList* analyzes segment 4.1.2 (`password`), *ExtractATD* identifies that there is no verb in the segment and returns failure to *HandleTestCase*. In this scenario, *HandleTestCase* backtracks and invokes *HandleSegList* with SegList $SL4.2$. Since both segments in $SL4.2$ satisfy the criteria for segmentation, *ExtractATD* generates the following *ATD* tuples for SegList $SL4.2$ and invokes *ExtractATD* with tuples of Segment 4.2.1.

```
(4.2.1) <enter, login, >
(4.2.2) <click, login, >
```

3.3 Algorithm 3: ExecuteATDList

The *ExecuteATDList* algorithm executes a list of *ATD* tuples of a tagged segment using runtime. If is successfully executes all tuples, *ExecuteATDList* automatically invokes *HandleSegList* with the next segment in the list.

Given an *ATD* tuple, *ExecuteATDList* first identifies the target element in the application under analysis using *DisambiguateTarget* (Line 5). Often, we identify that there can be multiple target elements for the target t of an *ATD* tuple. In such scenarios, *DisambiguateTarget* collects all candidate target elements and interprets each element using *InterpretStep*. *InterpretStep* executes the *ATD* tuple using runtime. If the step succeeds, *ExecuteATDList* invokes the next *ATD* tuple, otherwise backtracks to another alternative identified by *DisambiguateTarget* (Loop 6 - 15).

Algorithm 3 $ExecuteATDList(seglist, segindex, sslist, index)$

Require: an ATDTuple $atuple$
Ensure: SUCCESS or FAIL

```
1: if sslist.size() == index then
2:     //Proceed to the next segment
3:     return HandleSegList(seglist, segindex + 1)
4: end if
5: SS scriptstep = sslist[index]
6: List<SS> dislist = DisambiguateTarget(atuple)
7: for all SS localss ∈ dislist do
8:     ret = InterpretStep(localss)
9:     if ret == SUCCESS then
10:        //Proceed to the next ATD tuple
11:        ret = ExecuteATDList(seglist, segindex, sslist, index + 1)
12:        if ret == SUCCESS then
13:            return SUCCESS
14:        end if
15:    end if
16:    //Backtrack
17: end for
18: return FAIL
```

Backtracking We next explain another backtracking scenario, where backtrack happens based on runtime. Consider that *InterpretStep* successfully executed *ATD* tuple of segment 4.2.2. However, logging into application under analysis does not succeed, since the *ATD* tuple for entering the password is missing as shown in *ATD* tuples generated for $SL4.2$ above. If there exists a verification step after Step 4, we can automatically identify whether the verification step is passed or not and either proceed or backtrack based on the verification step. However, in the absence of verification steps, our technique proceeds with test step 5 as shown in Figure 6. Since the login does not succeed, *DisambiguateTarget* cannot identify any target element for `quantity` in segment 5.1.1, thereby *ExecuteATDList* backtracks. This backtrack happens all the way to Loop 7-14 in *HandleTestCase*, where *HandleTestCase* invokes *HandleSegList* with $SL4.3$, thereby successfully executing Step 4.

4 Empirical Evaluation

We implemented our approach for extracting keyword tuples from manual test cases, and conducted two empirical studies to evaluate the effectiveness of the approach. In the first study, we evaluated our approach end-to-end, including executing the generated test actions. We used two open-source web applications, with a corpus of manual tests written by professional testers for the purpose of this this study. In the second study, we used two commercial applications, with a small suite of real manual tests for the applications, to evaluate the effectiveness of segmentation and segment interpretation.

In this section, first, we describe the implementation (Section 4.1). Following that, we present the studies (Sections 4.2 and 4.3). Finally, we discuss the limitations of our technique and the threats to the validity of our observations (Section 4.4).

4.1 Implementation

In our prototype, we used Stanford NLP Parser[2] for parsing segments and for extracting ATD-tuples from segments. In particular, we first perform Parts-of-speech (POS) tagging for each word within the segment using the NLP Parser. We next identify verbs and nouns within the segments. In a few scenarios, we identify that target labels such as "`user name`" in the application under analysis can include multiple words that result in incorrect POS tagging of those words. To address this issue, we maintain a repository of all labels and data values in the application under analysis and explicitly tag those multiple words as a single noun. We next consider verbs as actions and nouns as target elements or data values. The primary reason for the ambiguity between target elements or data values is that our

[2] `http://nlp.stanford.edu/software/lex-parser.shtml`

technique does not impose any restrictions on the format of the test step. We disambiguate a noun as a target element or a data value by using the runtime information available in the later phases. To extract relations between actions and targets, we exploit dependency relations identified by the parser. For example, consider a test step as `enter login guest`. The NLP parser generates the following output:

```
POS tagging: enter/VB login/NN guest/NN
Dependency relations: dobj(enter, guest), nn(guest, login)
```

The preceding output shows that the NLP Parser tagged `Enter` as verb, and `login` and `guest` as nouns. Using relations `dobj` (direct object) and `nn` (noun modifier), our prototype extracts an ATD-tuple $(enter, guest, login)$ of the form (a, t, d). In our prototype, we use 21 dependency relations for extracting ATD-tuples. As shown, our prototype does not have knowledge whether `guest` is a data value or a target element during parsing.

To gather runtime information from the application under analysis and to execute extracted ATD-tuples, our prototype uses HtmlUnit[3]. HtmlUnit is a GUI-less browser for Java programs and provides APIs for various actions such as clicking buttons or filling out forms. Given an ATD-tuple, our technique extracts all Html elements in the current webpage and identifies candidate targets for the ATD-tuple using both target t and data value d of the ATD-tuple (a, t, d). We also identify that testers may not mention the exact name of the Html element. To address this issue, we leverage Levenshtein edit distance to identify the Html elements that are similar to the target described in the test step.

In a few scenarios, due to ambiguities in the test step, there can be multiple candidate Html elements on which the action a of (a, t, d) needs to be performed. Our prototype extracts all such candidates and systematically executes those ATD-tuples via backtracking. Our prototype next uses HtmlUnit for executing the ATD-tuple. Based on the output, our prototype either proceeds with the next segment (if successful) or backtracks to another candidate ATD-tuple (in case of failure).

4.2 Study 1

4.2.1 Goals and Method

The goal of the first study was to demonstrate that our approach can automatically interpret and execute the natural-language test cases. Moreover, this study also shows the importance of identifying valid segments and disambiguating targets. For this study, we used two popular open-source web applications: `BookStore` and `BugTracker`.[4] Specifically, in this study, we investigated the following research questions.

- **RQ1:** How often can our tool automatically execute manual test cases?

To create manual test cases for the two subject applications, we identified a few scenarios and requested two professional testers at IBM Testing services team to create the manual test cases for those scenarios. The testers were unaware of our tool and technique, and of the goal of the study (*i.e.,* the evaluation of a test-automation tool). The only instructions they were given were to be elaborate in writing the tests and mention the test data wherever needed. They were given no instructions on the levels of segmentation and different ambiguities. Thus, there is practically no bias in the creation of the test cases. In total, testers wrote 23 manual test cases (11 for `BookStore` and 12 for `BugTracker`), which consisted of 117 steps.

While analyzing the tests, we discovered a valuable side-benefit of our approach—that it can identify incomplete manual test cases. If after attempting to execute each alternative interpretation, the tool is unable to find an executable flow, it has *likely* identified an incomplete/incorrect test case. In some cases, the tool may fail because of some other factors or because the application itself is incorrect (so it is not the oracle for the test cases). But, it is highly likely that the tests are incomplete/incorrect. In fact, this is exactly what we found for some of the manual tests for `BookStore` and `BugTracker`. For example, some of the tests test cases do not include the step for logging into the web application. This logging step is required for performing the other actions mentioned in the test step. The primary reason for such missing steps is that the testers were not aware that these test cases were requested for the evaluation of our test-automation tool. But, nonetheless, these tests illustrate the power of our approach.

To address this issue of missing steps, we made minor enhancements to the test cases. Table 1 shows the enhancements made to unique steps among all test cases. In particular, we made three categories of enhancements. First, we inserted missing steps such as logging steps. Rows 1 to 4 in Table 1 belong to this category, where we inserted

[3] `http://htmlunit.sourceforge.net/`

[4] `www.gotocode.com`

Table 1: Enhancements made to manual test cases.

S.No.	Original test step	Modified test step
1	-	Launch the application through the link http://.../Login.jsp
2	-	Enter login and password, and click login
3	Enter Login ID = Guest and Password =Guest	Enter Login ID Guest and Password Guest. Click login
4	Login as Guest. (Use Guest for both Login and Password fields).	Enter login guest and password guest, and click login
5	Select the Book of your choice from the list of all Search Results displayed and then click either on the image of the book or on Name of the Book	Select **a title** from the list of all Search Results displayed and then click either on the image of the book or on Name of the Book
6	Enter the Advanced Search criteria's for minimum price = 20 maximum price = 40 Then Click on "Search" Button	Enter the Advanced Search criteria, enter "Price more then" 20 and enter "Price less then" 40, Then Click on "Search" Button
7	Click on the Employee link of the Employee that has been updated.	Click on the EmployeeMaint.jsp of the Employee created in setup.
8	Fill The Registration From With Appropriate Details for mandatory Fields Login* Password* Confirm Password* First Name* Last Name* Email* Then Click "Cancel" Button	Fill The Registration From With Appropriate Details for mandatory Fields. Fill Login, Password, Confirm Password, First Name, Last Name, and email, Then Click "Cancel" Button.
9	Enter the Necessary details /Changes to the Following fields and Click on Cancel Button "Login*" "Password*" "Confirm Password*" "First Name*" "Last Name*" "Email*" "Address" "Phone" "Credit Card Type" "Credit Card Number"	Enter the Necessary details /Changes to the Following fields "Login*", "Password*", "Confirm Password*", "First Name*", "Last Name*","Email*" "Address", "Phone","Credit Card Type","Credit Card Number" and Click on Cancel Button
10	From the Project, Assigned To, Priority and Status fields dropdowns, select the same values as selected in the setup.	Select Project, Assigned To, Priority and Status fields dropdowns.
11	Logout and Login as admin. (Use admin for both Login and Password fields)	Select Logout and Login as admin. (Use admin for both Login and Password fields)

Table 2: Evaluation results of manual test cases.

Subject	Test Cases	Test Steps Before	Test Steps After	Time (sec)	Test Steps with Multiple Segments	# SB	# CB
BookStore	11	60	70	926	36 (51.4%)	1448	698
BugTracker	12	57	76	132	42 (55.2%)	164	72

necessary steps for logging into the subject application. Second, we added hints to the step to identify target. More specifically, in a few steps, we found that there is no information available for identifying the target. For example, consider Row 5: `Select the book of your choice ...`. Our technique does not know which target to identify because none of the nouns in the step exist on the page where the action needs to be taken. To address this issue, we enhanced the test step as `Select a title ...`, which provides information to our technique to identify a title as a candidate target. Rows 5 to 7 belong to this category. Third, we modified the format of the test steps as required by our tool. Rows 8 to 11 belong to this category.

4.2.2 Results and Analysis

Among these 23 test cases, our technique successfully executed 19 test cases. Among the remaining four failing test cases, two test cases fail due to data dependences within the test case. In particular, these test cases include steps that refer to data (not explicitly mentioned) created in the earlier steps of the test case. Executing these test cases requires enhancements beyond the three preceding categories. The remaining two tests fail due to the limitations of our current prototype, which we plan to address in our immediate future work.

Table 2 shows the evaluation results. Columns 3 and 4 show the number of test steps before and after our enhancements, respectively. All the newly added test steps are related to logging. Column 5 shows the amount of time taken in seconds[5]. The time taken for a test case primarily depends on the number of possible segments within the test steps of that test case. For example, our technique took 457 seconds for one test case in `BookStore`. The primary reason is that this test case includes a complex test step that has 512 candidate segment lists. Among these candidates, segment 504 is identified as the correct segment for further processing.

Column 6 shows the number of test steps that include multiple segments. The percentages shown are with respect to the number of test steps after enhancements (Column 4). Our results show that testers often include multiple *ATD* tuples in a single test step, showing the significance of our technique for automatically identifying *ATD* tuples via segmentation. Column 7 shows the number of times our technique backtracked due to incorrect candidate segment lists. Finally, Column 8 shows the number of times our technique backtracked due to ambiguity in identifying target elements. As the data show, often multiple targets are possible for executing an action. Thus, an approach such as ours that resolves the ambiguities dynamically, using the application as an oracle, can be very useful. Overall, our results show that our technique can be highly effective in automatically executing test cases written in stylized natural language.

[5] All experiments were conducted on an Intel Core 2 Duo CPU machine with 2.53 GHz and 3 GB RAM.

Table 3: Accuracy of segmentation.

Subject	Test Cases	Test Steps	Test Steps with Multiple Segments	$\|AS\|$	$\|RS\|$	$\|AS \cap RS\|$	Precision	Recall
App1	121	648	325 (50.2%)	675	682	664	97.3%	98.4 %
App2	46	426	120 (28.2%)	258	362	250	69%	96.9%
Total	167	1074	445 (41.4%)	933	1044	914	87.5%	97.9%

4.3 Study 2

4.3.1 Goals and Method

The goal of the second study was to evaluate our approach on real commercial applications and manual test cases. For this study, we used two enterprise applications that are maintained by the Testing Services practice of IBM. For confidentiality reasons, we refer to the applications as `App1` and `App2`. We used a suite of the real manual tests written for these applications. Because we had access to the manual tests only, and not the application code, we focused the evaluation on the accuracy of segmentation and segment interpretation; we could not execute the interpreted test actions and, therefore, could not evaluate the approach end-to-end.

We define different measures to evaluate the accuracy of segment interpretation. Our measures are intended to compare the interpretation computed by our approach with the interpretation that a human, without knowledge of the application, would perform. Unlike the previous study, where the accuracy of our approach could be measured automatically via program execution, in this study, we evaluated accuracy manually. The interpretation of a segment by our approach is *correct* if it can identify the action and the target. The interpretation is *incorrect* if an action or a target identified by the tool does not match that suggested by a human, as per their analysis of the tester's intent in the test case. The interpretation is *ambiguous* if the step specification is incomplete in a manner that even human interpretation, without knowledge of the application, is impossible.

Specifically, in the study, we investigated the following research questions.

- **RQ2:** How often can test steps contain multiple segments? Can our approach identify the segments accurately?
- **RQ3:** How often does our approach compute correct segment interpretations, incorrect interpretations, and ambiguous interpretations?

4.3.2 Results and Analysis

Table 3 presents data about RQ2, which illustrate the need for segmentation and the accuracy of our segmentation approach. We find that a large fraction (50.2% in `App1` and 28.2 % in `App2`) of manual test steps require to be split into segments before an automation task becomes meaningful. On average, 87% of the 445 steps that needed segmentation are composed of just two segments. But, we do find test steps that contain as many as six segments. In most cases, either the composite test step strings together multiple segments with punctuations and/or conjunctions, or it contains a *from* clause. For example, the test step

```
User selects a provider from the left side-bar menu
and clicks [update provider]
```

resolves to three *ATD* tuples

```
<goto, left side-bar menu>
<select,provider>
<click, [update provider]>
```

One split happens corresponding to the "and"; a "goto" segment is introduced for the "from" clause.

To evaluate the accuracy of our segmentation approach, we measure *precision* and *recall* of retrieving segments from the set of composite manual steps, CS, where segmentation was necessary. Let, AS denote the set of segments that can be potentially derived from CS and RS denote the set of segments retrieved by running our prototype on CS. Then, we can compute precision and recall as follows:

$$Precision = \frac{|AS \cap RS|}{|RS|},\ Recall = \frac{|AS \cap RS|}{|AS|}$$

Table 4: Accuracy of segment interpretation.

Subject	Total Segments	Interpretation			
		Correct	Incorrect	Ambiguous	Missed
App1	998	898 (90%)	54 (5.4%)	21 (2.1%)	25 (2.5%)
App2	564	397 (70.4%)	34 (6%)	81 (14.4%)	52 (9.2%)
Total	1562	1295 (82.9%)	88 (5.6%)	102 (6.6%)	77 (4.9%)

For `App1`, both precision (97.3%) and recall (98.4%) are very high. The cases for `App2` are complex—many sentences are long-winding, with a wide variety of constructions of clauses. As a result, many spurious segments are identified and precision drops to 69%, even as recall remains high at 96.9%. As a strategy, our approach sacrifices precision to retain high recall rates. We realize that spurious segments, which bring down precision, can be easily discarded at runtime (as illustrate by our results for Study 1)—invalid targets/actions would not be resolved successfully at runtime, and the approach would backtrack to the previous state to execute the next (possibly correct) alternative. However, it is important not to miss any segment in the list of segments (*i.e.,* keep the recall high), so that all candidate segments can be evaluated by the runtime interpretation.

Table 4 presents the data about RQ3—it illustrates how effectively we can interpret segments not only to identify the correct verb that indicates action but also to zero-in on the target to be used by the runtime for target disambiguation. As explained earlier, we contrast our tool's output with the interpretation of a human to classify interpretations for segments as: *correct*, *incorrect* and *ambiguous*. Moreover, we list the number of cases where the tool fails to report any interpretation for a segment identifiable by the human.

Across the two applications, there are a total of 1562 segments. On average, we could precisely determine both the action and the target in 82.9% of segments (90% for `App1`,7 0.4% for the more complex `App2`). Only in 10.5% of cases (the total of columns 4 and 6), our tool proves to be inferior to human interpretation. In approximately half of these imprecise cases, the tool is not able compute any interpretation; for the remaining , it reports incorrect results. We consider an interpretation to be incorrect if it cannot point out the most apt verb present in the sentence as the action, or if it misses the most useful phrases that can serve as the context for disambiguating the target. For example, the following segment:

```
Click edit from the right hand action menu
```

is incorrectly resolved to the tuple:
`<edit, menu, right-hand-action >`.
Clearly, "click" should have been correct action verb and not "edit".

Finally, we report 6.6% segments to be *ambiguous*. The level of ambiguity is much greater in `App2` (14.4%) as compared to `App1` (2.1%). We stipulate three main classes of ambiguities. First, we find many manual test steps and segments derived thereof, exist at a higher level of abstraction than what the runtime can handle; this is especially true in `App2`. Ideally, a segment should specify an atomic unit of execution of an action on a target. However, we found examples such as:

```
Complete the workflow to publish the  bulletin
```

where the manual step does not give a clear idea on what action needs to be performed. Second, there are instances where there is insufficient data to perform disambiguation of the target. For example, in a step like:

```
Fill in the details on the page
```

the action "fill in" makes it clear that we are interested in text fields as targets but does not indicate which ones. Third, there may be non-standard verbs used in manual steps that may not be resolved by the runtime. Resolution of an action is particularly hard when both the action and target are ambiguous. In several cases, the action can be disambiguated from the target and vice-versa. For example, consider the segment `<Add, approver, >`. "Add" is a non-standard action verb unlike "click", "enter", etc. Depending upon whether the target context "approver" is resolved to a list or a button, we can decide whether to add an element or to click. However, if we cannot resolve the target context, the segment cannot be executed.

4.4 Limitations of the Current Approach

Our results indicate that the backtracking-based approach is remarkably effective in being able to interpret and execute manual test cases even in the presence of different types of ambiguities. But there are limitations of the technique. Mechanical interpretation of natural-language tests presents many other challenges; we mention some of such difficult cases that we found in our corpus of test cases.

We found cases where a test step made a cross reference to another test case.

```
Refer tc01 bulletin - create and publish
   test case for instructions to create a bulletin.
   Select external distribution
```

Another example of a difficult scenario that we came across involves looping over test steps

```
Repeat the steps above; select only technical reviewer
   and approver.
```

This step instructs the tester to perform the previous steps again with some variations. Whereas for a human, interpreting this step may not be difficult, for a computer, the interpretation can be challenging.

Finally, as we mentioned in Section 4.2, we found examples of dependences between test steps, *e.g.,* , where a subsequent test step referred to a data value used in the preceding step. An example of such a test step is

```
Enter with New Login ID and Password
```

which instructs the tester to enter the ID and password created in a preceding test step.

In future work, we will extent our technique to handle such cases.

5 Related Work

There is a large body of work on synthesizing programs via mechanical interpretation of natural-language phrases, which is inspired by the ideal of bringing programming to end-users and bridging the gap between human-readable natural languages and mechanically interpretable programming languages [7]. A recent symposium on the future of software-engineering research discussed the automated generation of formal software-engineering artifacts, such as models and test cases, from natural-language descriptions as one of the important research directions [20].

The most notable effort in the direction of end-user programming, with a long line of research, is programming by demonstration [1, 10]—which expounds the principle that a program is synthesized automatically by observing the manual actions performed by the user. This is also the underlying philosophy of record-replay features of many commercial and open-source testing tools. Other efforts at bridging the gap between natural languages and programming languages include the development natural-language interfaces for programming languages (*e.g.,* [16]); providing semantics to natural-language interfaces (*e.g.,* [13]); and the development of structured editors [7].

Automated program synthesis has been studied more broadly as the problem of discovering a program that realizes a user intent—the intent could be stated in different forms such as, natural language, input-output examples, logical relations between inputs and outputs, and demonstrations; the users could range from algorithm designers and programmers to students and end-users [4]. Gulwani [4] describes three dimensions of the synthesis problem: the form of user intent, the search space of programs, and the search technique. Our work can be formulated within this framework. We target intent stated as natural-language imperative-form commands. The search space consists of tuples of keywords (from the keyword domain of a specific testing framework) and target elements (from the domain of user-interface elements used in the application under test). The search technique is based on natural-language parsing combined with exploration of multiple flows with backtracking. A key aspect of our synthesis approach is that we use the application under test as an oracle for the test program being synthesized.

Recent approaches in automated program synthesis consider user intent specified in different forms, such as input-output mappings (*e.g.,* [5]), logical relations between inputs and outputs (*e.g.,* [19]), and traces (*e.g.,* [8]). (In the programming-by-demonstration approach, the user demonstration can be regarded as a trace of the intended system.)

Closer to our work, techniques for automatically interpreting user intent stated in stylized natural-language expressions have been developed in the contexts of test scripting [14], synthesizing code from informal keyword expressions [11], automating web-based processes [3, 12], and guiding users in following "how-to" instructions [9].

The CoTester system [14] uses an English-like test-scripting language, called ClearScript, which can be automatically interpreted. CoTester provides a record-replay feature similar to that available in conventional testing tools, with the difference that the recorded scripts are in the stylized English form of ClearScript and, therefore, easily readable even by non-programmers or novice programmers. Although ClearScript test steps are expressed in the English language, each step must correspond precisely to a segment. Thus, the burden of segmentation and ordering is on the user. Our approach removes this restriction and attempts to address segmentation and ordering ambiguities automatically. In essence, programming in ClearScript is at the level of keyword-driven approaches, albeit, with a more user-friendly natural-language syntax; the higher level of informality is more in the syntax, and not in degree of precision in specifying test steps. Moreover, the target-ambiguity problem is not addressed in CoTester.

The test-scripting language of CoTester, ClearScript, was developed in earlier work in the context of automating web-based processes (*e.g.,* procurement and expense reimbursement) in enterprises [3, 12]. The ClearScript specifications require accurately segmented and ordered process steps.

Recently, Lau and colleagues [9] have developed techniques for automated interpretation of "How-to" instructions to provide guidance to end-users in accomplishing web-based tasks, such as making an online purchase or creating a web-based email account. They formalize the interpretation problem as the inference of a tuple (A, V, T) consisting of an action, a data value, and a target element; the inference of such a specification is our goal too. Their approach targets instructions written by end-users, which can have a greater degree of informality and ambiguity than what would typically be present in the manual tests targeted by our approach. However, they assume that the instructions have been properly segmented and, therefore, do not handle segmentation and ordering ambiguities. They developed and experimented with three techniques for tuple inference: keyword-based interpretation, grammar-based interpretation, and a technique based on supervised learning. In their studies, keyword-based interpretation emerged as the most effective technique. Our approach is complementary to their approach in different ways and, in fact, the two approaches could be combined in a synergistic manner. Our technique performs automated segmentation and segment ordering. The ordered segments computed by our tool could be input to their keyword-based interpreter to infer the tuples. Next, the tuples could be executed using our approach, with backtracking to the earlier steps to resolve the three types of ambiguities.

Little and Miller [11] present a technique for automatically translating keywords, *i.e.,* informal plain-text expressions, to the API of a particular system. The goal of their work is to lower the barriers for end-users in leveraging the scripting interfaces provided by applications such as Microsoft Word. The users need only be familiar with the vocabulary of a system—they can specify a task, such as formatting a document, using plain-text keywords, which the system attempts to translate to the application API. In this approach, an expression need not have the imperative command form of ClearScript, but it still corresponds roughly to a segmented action. This approach is similar in principle to API discovery techniques, such as Jungloid mining [15], but requires less formal expressions that are more appropriate for end-users.

In related work, researchers have presented techniques for analyzing natural-language descriptions of use cases to extract models, measure quality, detect defects, and generate test cases (*e.g.,* [2, 17, 18]). In particular, the Reference [18] presents techniques for extracting models of use cases, and performing control-flow and data-flow analyses on the models to generate test cases.

6 Conclusion

We showed a technique using which, manual test cases written in the form of test steps described in English can be almost automatically converted to a form ready for mechanical interpretation. The technique uses backtracking search to resolve ambiguities inherent in the tester's instructions. Yet, it is precisely the tolerance to ambiguities that makes it commonplace to use a natural language as a medium in which to describe test cases.

The importance of this work is that it can drastically reduce the cost of test automation. Test automation is a necessary cost in testing practice, because manual test cases cannot be executed repeatedly in an efficient manner; they have to be converted to executable scripts. Our work automates the conversion to script form. The technique accomplishes this with remarkably high accuracy, failing less than 15% of time.

Our work can also be viewed as an instance of program synthesis from informal descriptions. Here the manual test steps are the informal descriptions, and the output is a program that an interpreter can use to drive an application under test.

References

[1] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch what I do: Programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[2] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Application of linguistic techniques for use case analysis. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 157–164, 2002.

[3] T. Matthews G. Leshed, E. M. Haber and T. Lau. Coscripter: Automating and sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1719–1728, 2009.

[4] S. Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 13–24, 2010.

[5] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 215–224, 2010.

[6] JUnit. URL `http://junit.org`.

[7] C. Kelleher and R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37:83–137, June 2005.

[8] T. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2nd International Conference on Knowledge Capture*, pages 36–43, 2003.

[9] T. Lau, C. Drews, and J. Nichols. Interpreting written how-to instructions. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, pages 1433–1438, 2009.

[10] H. Lieberman, editor. *Your wish is my command: Programming by example*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[11] G. Little and R. C. Miller. Translating keyword commands into executable code. In *Proceedings of the 19th ACM Symposium on User Interface Software and Technology*, pages 135–144, 2006.

[12] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 943–946, 2007.

[13] H. Liu and H. Lieberman. Programmatic semantics for natural language interfaces. In *CHI 2005 Extended Abstracts on Human Factors in Computing Systems*, pages 1597–1600, 2005.

[14] J. Mahmud and T. Lau. Lowering the barriers to website testing with CoTester. In *Proceedings of the 14th International Conference on Intelligent User Interfaces*, pages 169–178, 2010.

[15] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–61, 2005.

[16] D. Price, E. Riloff, J. Zachary, and B. Harvey. NaturalJava: A natural language interface for programming in Java. In *Proceedings of the 5th International Conference on Intelligent User Interfaces*, pages 207–211, 2000.

[17] A. Sinha, A. M. Paradkar, P. Kumanan, and B. Boguraev. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 327–336, 2009.

[18] A. Sinha, S. M. Sutton Jr., and A. Paradkar. Text2Test: Automated inspection of natural language use cases. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 155–164, 2010.

[19] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 313–326, 2010.

[20] W. F. Tichy and S. J. Koerner. Text to software: Developing tools to close the gaps in software engineering. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 379–384, 2010.

[21] Watson. URL `http://www-03.ibm.com/innovation/us/watson/index.html`.