RJ10497 (A1201-012) January 12, 2012
Computer Science

IBM Research Report

Towards Automatic Functional Test Execution

Pablo Pedemonte
IBM Argentina
Ing. Butty 275 - C1001AFA
Buenos Aires, Argentina

Jalal Mahmud, Tessa Lau
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Towards Automatic Functional Test Execution

Pablo Pedemonte
IBM Argentina
Ing. Butty 275 – C1001AFA
Buenos Aires, Argentina
ppedemon@ar.ibm.com

Jalal Mahmud
IBM Research – Almaden
650 Harry Rd.
San Jose, CA 95120
jumahmud@us.ibm.com

Tessa Lau
IBM Research – Almaden
650 Harry Rd.
San Jose, CA 95120
tessalau@us.ibm.com

ABSTRACT

As applications are developed, functional tests ensure they continue to function as expected. Nowadays, functional testing is mostly done manually, with human testers verifying a system's functionality themselves, following hand-written instructions. While there exist tools supporting functional test automation, in practice they are hard to use, require programming skills, and do not provide good support for test maintenance. In this paper, we take an alternative approach: we semi-automatically convert hand-written instructions into automated tests. Our approach consists of two stages: first, employing machine learning and natural language processing to compute an intermediate representation from test steps; and second, interactively disambiguating that representation to create a fully automated test. These two stages comprise a complete system for converting hand-written functional tests into automated tests. We also present a quantitative study analyzing the effectiveness of our approach. Our results show that 70% of manual test steps can be automatically converted to automated test steps with no user intervention.

Author Keywords

Manual Test Automation; Natural Language Processing; Supervised Learning

ACM Classification Keywords

H.5.2 Information Interfaces and Presentation: User Interfaces—*Natural Language*;

INTRODUCTION

Testing can be an expensive task [9] whose effectiveness is crucial to a project's success. One way to achieve an effective testing process is by means of test automation. By automating some or all of the work necessary to execute a test, testers can focus on verifying complex cases and key features instead of dedicating time to repetitive, tedious execution tasks. Automated execution excels on repetitive tasks like unit testing [6]. But there are other testing disciplines where automated execution is hard to achieve with existing tools.

Functional testing poses a case where test automation is difficult to achieve. Functional tests verify that a system complies with its specifications and requirements, ignoring internal details such as code, design or architecture. According to our experience in large organizations, functional testing is done mostly manually: system functionality is verified by human testers, following hand-written instructions detailing how to feed a system with some input, and the expected outputs. This situation is particularly problematic for large, long-lived projects, which might carry hundreds of legacy functional tests. Periodically executing such tests by manual means can take a considerable amount of time and resources. Our work aims to automate functional test execution, thus helping to alleviate this problem.

In practice, the approach to functional test automation consists of deriving a program from a test. One can derive such a program either by coding it (which puts the automation burden on the developers) or by means of a recording tool [8]. At first glance, recording looks like a good solution for automating functional tests. At the price of manually following the test instructions during a recording session, one obtains a script that can be executed later without incurring further automation overhead. But there are limitations to recording. First, testers still need to manually execute the test during recording. Second, software artifacts resulting from recording consist of generated code, which is not easy to maintain. If a test needs to be changed one has to modify the resulting program. This requires programming skills, and might not be a simple task. Alternatively one could re-record, which implies manually executing the test again. While end-user programming systems such as CoScripter [13] could reduce the programming requirement, the recording barriers remain. In contrast, we propose a semi-automated approach that eliminates the need to do any recording at all.

We address functional test automation with a two stage approach:

1. Converting a manual functional test into an intermediate representation amenable for automated execution.

2. Interpreting the intermediate representation with user guidance.

Our approach processes a test step by step, converting and interpreting each step in turn. The key problem with understanding natural language is that it is imprecise, so the meaning of a hand-written instruction might not be unambiguously determined. We handle such cases by asking for user guidance: if while executing a test we find an instruction whose meaning can not be precisely determined, we ask the user how to proceed. We use this feedback to create a modified manual test based on the original, which can be automatically executed using our system. Our approach is especially suited for periodic functional test execution, which is the norm in test organizations, as functional tests are used during development to ensure a system continues to work as specified. After a first run of a test requiring user intervention to handle ambiguous instructions, our approach enables fully automatic execution by a returning a new test defining the same actions as the original, but requiring no user assistance to execute.

This approach frees programmers from the tedious task of writing code for functional tests, and testers from the burden of manual step execution (provided the interpreter does not require user intervention at every step). But why is it better than recording? Like recording, our approach requires effort from the user only for the first execution of a test, and future runs will not need user intervention. But unlike a recording session, our first execution requires user intervention sporadically, only when the interpreter finds an ambiguous instruction. Furthermore, since our approach generates written manual tests instead of code, it reduces the cost of maintaining these tests over time.

Specifically, we make the following contributions:

- A machine learning-based algorithm for converting a test into an intermediate representation suitable for automated execution.
- A semi-automated algorithm for interpreting the intermediate representation derived from a test, requesting feedback from the user if the intermediate representation can not be unambiguously interpreted, and using this feedback to create an equivalent test with no ambiguous steps.
- An implemented system for automatic manual test execution, based on our conversion and interpreter algorithms.
- A quantitative study evaluating the performance of our algorithms. Our results show that we can execute more than 70% of the instructions in a test without user intervention.

We start with a formulation of our approach to functional test automation, then we describe the stages of our approach. After that, we present the quantitative study analyzing the effectiveness of our approach. Finally, we discuss key aspects of our approach and future research directions, and conclude with an overview of related work.

PROBLEM DEFINITION

A manual test is an ordered sequence $\{s_i\}$, $i \geq 1$, of test steps (or instructions). A *value-free* manual test is a pair (M, vs), where M is a manual test whose literal values (such as "10" in `type quantity 10`) have been replaced by variables, and vs is an ordered set of values. The idea is that a value-free test is a parametrized test that can be executed many times with different values, by providing distinct sets. We transform a manual test into its value-free version by a manual *value replacement* stage which we will describe later.

Given a value-free manual test (M, vs), $M = \{s_i\}$, our system executes the steps s_i in order of occurrence and returns a new test $M' = \{s'_i\}$ that defines the same actions as M, but *does not* require user intervention when executed. In this way we materialize the benefits of our approach: executing M might require user intervention, but after that we will have a manual test M' equivalent to M, requiring no effort from the user to execute. Fig. 1 (a) depicts the complete process.

Executing a step involves two stages: converting it to an intermediate representation, and interpreting that representation on the application under test. Fig. 1 (b) shows how these two stages interact in order to execute a single step.

Converting Test Steps

Based on existing work about interpreting hand-written instructions [13, 12], we view test steps as entities denoting *actions* to perform on *target* GUI elements of the application. Hence, in order to execute a step, we need to identify:

- The actions denoted by the step.
- The target for each action in the step.
- Any data required by the actions. For example, the "enter" action requires a value to be entered into a text field.

Fig. 2 shows a sample real-world test about reinstating lapsed insurance policies. The actions in step 1 are "enter" and "press", and their respective targets are two GUI elements labeled "policy number" and "search". Moreover, the "enter" action has a policy number template as associated data. Targets can be specified either by a label alone (e.g., "search" in step 1) or by a label and a target type (like "save button" in step 4). Targets specified only by a label are *implicit*.

Therefore, we define the intermediate representation of a test step to be a tuple combining the action intended by the step, the target of the action, and any associated data if present. We will refer to these as (A, T, D) tuples.

We define the current application state st to be the combined state of all the GUI elements of the application at the present time. Performing an action on the application (e.g., entering some text) might change the GUI, and hence, the current state. Our conversion algorithm is then defined as a function taking a pair (st, s) representing the current application state and a test step to convert, and returning an ordered list of one or more tuples $\{(A_i, T_i, D_i)\}$, $i \geq 1$, one for each operation denoted by a step. From now on, we will refer to the conversion algorithm as *tuple extraction*.

Fig. 3 shows the (A, T, D) tuples obtained from the example test (a "•" in the data component means that the tuple carries no data). Note that a test step can be converted into multiple tuples. Note also that there is no translation for step 5 in Fig. 2. That step does not denote an action to apply on the application's GUI, but rather a verification step we know nothing about (even a tester attempting to execute this step might have trouble trying to find out what are the "operational and referral rules" that the system should follow). Processing such instructions is an area for future work.

Interpreting Test Steps

We define the interpreter algorithm to be a function taking the current application state st, the textual representation s of

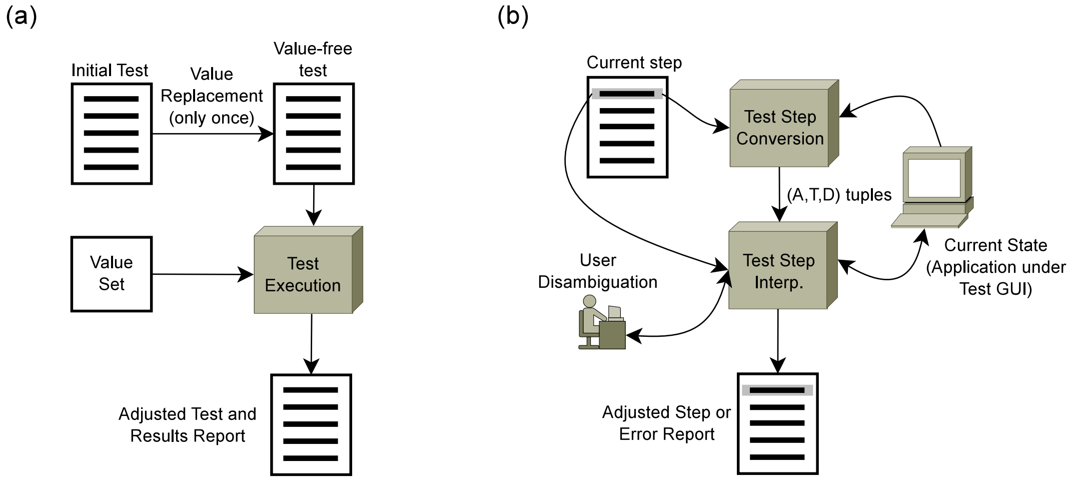


Figure 1. (a) Complete manual test automation process – (b) Execution of a single step

the current step, and a tuple (A, T, D) to interpret, obtained from converting s. The result will be a pair (st', s'), where st' is the new application state resulting from interpreting the tuple, and s' an unambiguous version of the step s, based on the user's feedback if disambiguation took place. If user intervention was not needed, s' will be equal to s. Therefore, the result of running the interpreter on each step of a test M is a new test, M', whose steps have been fully disambiguated.

Interpreting an (A, T, D) tuple requires applying the action A to the target denoted by T. The crucial step is resolving the target, i.e., finding a GUI element matching T. There can be multiple elements matching T. For example, consider a page having a search link and a search icon. Both elements will match the target in the tuple (*click*, *search*, •). It is likely that clicking either element will have the same effect, but we can not assume that in general. When the target of a tuple can not be uniquely determined, we say that the tuple's target is *ambiguous*. In some cases, this ambiguity can be overcome by inspecting the tuple to interpret. But in general, there may not be enough information to select the intended target.

Let $T.\,l$ be the label in the target specification, and $T.\,t$ the type (which will be the undefined value "•" if T is implicit). Let $label$ be a function that given the application state and an GUI element computes its label, and $type$ a function that returns an element's type. Then, the set of elements e matching the target specification T is:

$$\{e \in st \mid T.\,l = label(st, e) \wedge \\ (T.\,t = \bullet \ \vee \ T.\,t = type(e))\}$$

Note that in order to find the candidate targets matching T we need the application state to be up to date, i.e., reflecting the effects of the steps interpreted previously. This means that when we try to interpret the tuples obtained from a step s_i, the application under test must be in a state reflecting the effect of the tuples for the steps $s_1 \ldots s_{i-1}$. One way to achieve this is to "thread" the state through the interpreter, passing the current state to the function and returning a new state resulting from interpretation of the tuple.

```
1. enter the policy number as xxx-xxx-xxxx
     then press search
2. press reinstate button
3. enter the following
     effective date as mm-dd-yyyy
     paid indicator as "y"
     comments as "payment done"
4. press save button
5. validate system follows operational
     and referral rules
```

Figure 2. A real-world manual test

1 (***enter***, *policy number*, *xxx-xxx-xxxx*)
(***press***, *search*, •)
2 (***press***, *reinstate button*, •)
3 (***enter***, *effective date*, *mm-dd-yyyy*)
(***enter***, *paid indicator*, *y*)
(***enter***, *comments*, *payment done*)
4 (***press***, *save button*, •)

Figure 3. Tuples for manual test in Fig. 2

From now on, we will refer to the algorithm interpreting the intermediate representation of a test step (i.e., a sequence of (A, T, D) tuples) as the *tuple interpreter*.

VALUE REPLACEMENT

In practice, functional tests are executed many times with different values. This might have an impact on the way that tests are written. For example, step 1 in Fig. 2 tells us to enter a policy number following the pattern "xxx-xxx-xxxx", but it says nothing about the valid values we could pass to the system in order to move on to step 2. This is because the test is expected to be executed with different policy numbers. For example, one could verify that the system correctly processes different classes of policies, e.g., car accident and credit card insurance.

This complicates repeated test execution: every time we execute a test, we might need to modify it in order to provide new values. We overcome this complication by introducing a mechanism to abstract out values from a functional

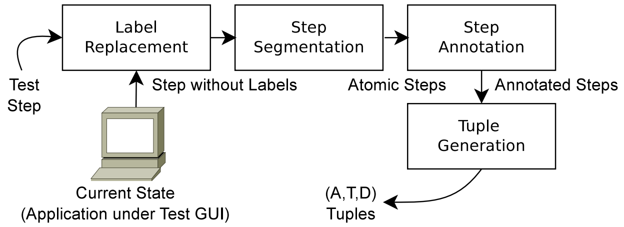


Figure 4. Tuple extraction process

test, namely *value variables*. Value variables have the form *VALUE*(i), referring to the i^{th} value in an ordered set of values available at text execution. This way, step 1 in Fig. 2 would change to:

```
enter policy number as VALUE(0)
  then press Search
```

where the test must be run with an ordered set of values holding a policy number at position 0. Value variables are useful in other cases where values are often missing, such as login instructions. A step like `enter user name and password` can be converted into `enter user name VALUE(0) and password VALUE(1)`, thus making the values explicit.

In the value replacement stage users replace values in a test with variables and define one or more value sets. Like user disambiguation, value replacement will be necessary only for the first execution of a test. Although this represents a slight overhead for the tester, it facilitates repeated test execution. If we need to execute a test several times with different values, we only need to re-execute with a different value set. The test remains unchanged. Automatic value replacement is an area for future work.

TUPLE EXTRACTION

We extract a sequence of (A, T, D) tuples from a step by a succession of fully automatic transformations (depicted in Fig. 4). We start with an example, then we describe all the transformations in detail. For the example we will use the first step of the test in Fig. 2. After value replacement, we have:

```
enter policy number VALUE(0) then press search
```

We start by replacing target labels with label variables. This is similar to introducing value variables, but since we can query the current state of the application for the available labels in the GUI, it is possible to automatically enumerate all the labels and remove their occurrences in the step. This *label replacement* stage leaves us with a simplified, less verbose test step. Our example test step becomes:

```
enter LABEL(policy number) VALUE(0)
  then press LABEL(search)
```

Note that this step denotes two actions. Our next stage, *step segmentation*, decomposes such a *composite* step into steps containing *exactly one action verb*, i.e., a verb that can act as the A component of a tuple. We call such steps *atomic*. Segmentation gives us two atomic steps: `enter LABEL(policy number) VALUE(0)` and `press LABEL(search)`.

The next stage, *step annotation*, identifies the tokens in each atomic step referring to the action, target and data of the tuples we intend to obtain. In our first atomic step we identify "enter" as the action, the label as the (implicit) target, and the value as the associated data. In the second atomic step, "press" is the action and the label the (again implicit) target. Based on this information, in the final *tuple generation* step we compute the two tuples as shown in Fig. 3: (***enter***, *policy number*, *VALUE*(0)) and (***press***, *search*, ●).

Label Replacement

There are cases where different instructions have the same underlying structure. Consider the steps:

```
click Add item to shopping cart icon
click Search button
```

What if we replace target labels with variables, like we did with values? If we write label variables as *LABEL*(l), where l is the replaced target label, the example becomes:

```
click LABEL(Add item to shopping cart) icon
click LABEL(Search) button
```

Replacing labels with variables has has the desirable effect of inducing a notion of a test step normal form that allows us to focus on the *structure* of the step. After label replacement, seemingly different steps end up having the same structure. This simplifies the remaining stages of the extraction process, by reducing the number of forms that test steps can take. Besides, label replacement results in shorter instructions where it is simpler to identify the components of the (A, T, D) tuples to extract. For example, in its original form the first step has 7 tokens; after label replacement it has been reduced to 3 tokens ("click", a *LABEL* variable, and "icon") which directly correspond to the step's action, target label, and target type. Later, we will see that label replacement also allows us to avoid tricky labels that might prove problematic for later stages of our tuple extraction algorithm.

Label replacement starts by tokenizing and POS tagging [3] the current step. Then, we identify target labels with the aid of a table holding all currently available labels. We compute the table by collecting the labels of all the elements in the application's GUI. Once we have this table, we calculate all n-grams in the step using suffix arrays [14]. Then, for each n-gram, we query the table to check if it is a label. We have implemented a few strategies to ensure that we don't have false positives, e.g.: discard candidate unigrams POS-tagged as verbs if they belong to a list of typical actions (e.g., click, select, type, enter, etc.), and ignore n-grams that are included in another candidate n-gram. Finally, we replace each matching n-gram l with a variable *LABEL*(l) POS-tagged with the *NN* tag (i.e., a noun).

Step Segmentation

The latter stages of the tuple extraction algorithm work on atomic test steps, i.e., steps including exactly one action verb. The step segmentation phase does the work of computing the atomic constituents of a possibly composite test step. If the input step is already atomic, then this stage reduces to the identity function.

A possible approach to segmentation is to define a set of delimiter words (e.g., "and" or "then"), using them to obtain the atomic instructions in a step. For example:

```
enter user admin and password admin01
  and click login
```

This naïve approach would result in the steps `enter user admin`, `password admin0`, and `click login`. But note the second step: it does not include an action verb, so it is not atomic. We could determine that the action is "enter" by inspecting the previous step, but this introduces dependencies on previous actions in the test.

We decided to avoid such backward dependencies by an alternative criteria based on action verbs instead of delimiter words. The action verbs in the example step are "enter" (applied to two input fields) and "click" (applied to a button). These two action verbs map to the atomic instructions `enter user admin and password admin0`, and `click login`. Note that these steps are atomic, and that atomic steps might denote more than one action (although they include only one action verb). This new criteria complicates the naïve segmentation strategy: separator words like "and" will not always be delimiters. Simply looking for delimiter tokens is no longer a reliable strategy for segmentation.

Sentence chunking [1] (also known as shallow parsing) is a technique for assigning labels to tokens in a text, typically used to divide text into syntactically related non-overlapping groups of words. Sentence chunking is a well known problem in the natural language processing community (see the CoNLL-2000 shared task [22]), and there are several solutions to it. If we use special labels to delimit atomic instructions in composite steps, then we can reduce the decomposition problem to sentence chunking with these labels. A convenient form for such delimiter labels is the BIO tagging scheme [15]. Under this scheme, a token either starts a step, is inside, or outside an atomic step. So we only need three tags: ***B-CMD***, ***I-CMD*** and *O*, respectively. The example instruction should be chunked like this:

```
enter            B-CMD
LABEL(user)      I-CMD
admin            I-CMD
and              I-CMD
LABEL(password)  I-CMD
admin01          I-CMD
and              O
click            B-CMD
login            I-CMD
```

Notice the labels for the "and" tokens: the first one is *inside* an atomic action, since `password admin01` is not an atomic step. But the second one is a delimiter for the two atomic instructions in the composite step.

Step Annotation

The previous stage transformed a possibly composite test step into a sequence of atomic instructions. In order to derive (A, T, D) tuples from an atomic step, we must identify the regions corresponding to the action, targets and additional data in the step. For that purpose, we will annotate atomic steps using again sentence chunking. But this time, we will have a richer set of labels. From analysis of a comprehensive set of manual steps written by professional testers, we decided to use the following labels:

- *ACT*: action part of the step
- *TGT*: target part of the step
- *VAL*: data part of the step, if any
- *LOC*: location information, explains how to reach a target
- *DSC*: description of step's intention or effect, usually not useful for computing tuples.

The following instruction exemplifies how we annotate test steps:

```
In the LABEL(Sender Data) tab,
enter anonymous in the LABEL(name) textbox
for anonymous feedback
```

Note that this step is atomic, as it features a single action (enter some text). This is the labeling we expect to obtain:

```
In                  B-LOC
the                 I-LOC
LABEL(Sender Data)  I-LOC
tab                 I-LOC
,                   O
enter               B-ACT
anonymous           B-VAL
in                  O
the                 B-TGT
LABEL(name)         I-TGT
textbox             I-TGT
for                 B-DSC
anonymous           I-DSC
feedback            I-DSC
```

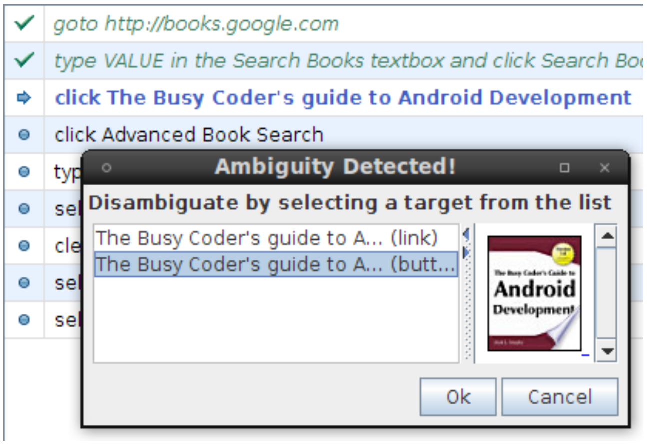
We have the `enter` token marked as the action, and `the name textbox` as the target. We also have a value to enter in the text box, `anonymous`. Finally, we identify location information (the target is in the `Sender Data tab`), and a verbose explanation of the instruction's intent.

Tuple Generation

Having a sequence of annotated atomic steps, we have to derive a set of tuples from this data. Based on the output obtained from the step annotation stage, we discovered that the following rules suffice to generate tuples from an annotated step:

1. *Single target*: generate a single tuple.
2. *Multiple targets, no values*: distribute the action across the targets, generating as many tuples as targets in the step.
3. *Multiple (target,value) pairs*: generate a tuple for every target; using the corresponding value as the data component of the tuple.

The first option is the trivial case. With a single target, we only need to group it with the action and any data (i.e., a value chunk) present in the step. For example, for the step `click ok` the resulting tuple will be (***click***, *ok*, •).



✓ goto http://books.google.com

✓ type VALUE in the Search Books textbox and click Search Bo

➪ click The Busy Coder's guide to Android Development

● click Advanced Book Search

● typ

● sel

● cle

● sel

● sel

Ambiguity Detected!

Disambiguate by selecting a target from the list

The Busy Coder's guide to A... (link)

The Busy Coder's guide to A... (butt...

Figure 5. Disambiguation dialog

Rule 2 refers to steps with multiple targets and no data, such as `click remember me and login`. In this case we distribute the action across all the available targets, to get (***click***, *remember me*, ●) and (***click***, *login*, ●).

When we find a sequence of targets, some or all of them paired with a following data region, it means that we are processing a multiple data entry step. Instructions to fill in a form follow usually follow this pattern. For example:

```
enter
  first name as John,
  last name as Doe
```

According to rule 3, we generate one tuple for each (target, value) pair in the instructions. That gives us the tuples (***enter***, *first name*, *John*) and (***enter***, *last name*, *Doe*).

These three rules define how to map a single action plus multiple targets and values to a sequence of tuples. What about location regions? They specify spatial relations that help to locate targets. The two relations that we identify are containment (for example, `click the ok button inside the checkout tab`) and relative placement (for example, `click the search button next to the settings link`).

Containment relations result in an extra tuple to select the target's container before performing an action on the target. For the containment example above, we will obtain the tuples (***click***, *checkout tab*, ●) and (***click***, *ok button*, ●). The first tuple ensures the container for the "ok" button has focus, while the second one performs the intended "click" action on the "ok" button.

Relative placement relations can be useful for avoiding ambiguity. If we had two buttons with label "b", one next to a panel "p" and another besides a panel "q", we could say `click the b button next to panel p` to unambiguously refer to the first one. At the expense of requiring user intervention due to ambiguities that could have been avoided otherwise, we are still not using relative placement relations in our tuple extraction algorithm. Processing relative placement information so we can reduce target ambiguity is a topic for future work.

TUPLE INTERPRETATION

After converting a step into its intermediate representation, we must interpret the resulting tuples against the application under test. Our tuple interpretation algorithm is defined by the following pseudo-code:

```
procedure interpret $(st, s, (A, T, D))$ :
  $es \leftarrow \{e \in st \mid label(st, e) = T.l \wedge$
                 $(T.t = \bullet \ \vee \ T.t = type(e))\}$
  if $|es| = \varnothing$
    error ("Invalid tuple")
  if $|es| > 1$
    $es \leftarrow$ heuristicDisamb $(es, (A, T, D))$
  if $|es| > 1$
    $(e, s') \leftarrow$ userDisamb $(es, s)$
  else
    $e \leftarrow$ the only element in $es$
    $s' \leftarrow s$
  $st' \leftarrow$ apply $((A, T, D), e)$
  return $(st', s')$
```

Here, st is the current application state, s is the textual representation of the current step, and (A, T, D) is the tuple to interpret. We start by resolving the target denoted by T. For that, we collect the set es of all the available GUI elements from the application state for which the tuple applies. If es is empty, the tuple is invalid. If there are multiple candidate elements, we try to narrow down the set by a heuristic disambiguation procedure. If we still have multiple elements we ask for user disambiguation, using the user's choice to compute a new textual step s'. If user feedback was not required, s' will be equals to s. Once we compute the GUI element e denoted by T, we apply the tuple (A, T, D) to it, resulting in state st'. We return st' and the step s'.

The heuristic disambiguation procedure uses the (A, T, D) tuple to narrow down the candidate set to a single element. For example, if we have a tuple (***enter***, *Search*, *books*) and the candidates are an input field and a button, we must be referring to the former, since entering some text applies to input fields, not buttons. Hence, we can discard the button, ending up with a single candidate element.

If we still have multiple candidates, we ask the user to disambiguate. Our interpreter presents a dialog to the user with the list of candidate elements. Fig. 5 depicts the execution of a test doing a book search on a web application. The user is being asked to pick between the title or an image of the cover of the book for the instruction `click the Busy Coder's Guide to Android Development`. Once the user picks an alternative, the interpreter can proceed to apply the action to the user's selection. But equally important, the user's answer is used to compute a new test step s' reflecting the answer. In our example, if the user picks the link option, the step will be adjusted to `click the Busy Coder's Guide to Android Development` **`link`**.

SVM			CRF		
Prec.	Rec.	F_1	Prec.	Rec.	F_1
97.32	96.29	96.80	97.83	95.76	96.78

Table 1. Step segmentation results

Applying an (A, T, D) tuple to a GUI element e requires programmatically emulating the action A on e. For example, if $A =$ "click" we will generate a click event on e, and if $A =$ "type", we will set e's value to D. The set of available actions is the same as in CoScripter [13], plus synonyms for them (e.g., "press" for "click", or "type" and "fill in" for "enter").

EVALUATION

We conduct a series of experiments to analyze the effectiveness of the segmentation and annotation stages of our tuple extraction algorithm. We also determine how much label replacement contributes to step annotation, by repeating our experiments with the label replacement stage turned off. In addition, we measure how much user intervention our approach needs to create a fully automatable test.

Experiment 1: Step Segmentation

For this experiment, we used a corpus of 27 real world manual tests, written by professional testers. These tests comprise a total of 154 test steps. We manually annotated the test steps with the corresponding *CMD* and *O* labels, and trained two chunk annotators on this data: Yamcha[1] (based on Support Vector Machines [5], or SVM for short), and CRF++[2] (based on Conditional Random Fields [11], or CRF for short). We evaluated the resulting models using a 10 fold cross-validation experiment on the training data, calculating precision, recall, and the F_1 measure for each run.

We calculate precision P, recall R, and F_1 measure for a tag t as follows: let N be the number of occurrences of t in a test corpus, O the number of occurrences of t reported by an annotator, and C the number of true positives (i.e., the number of correct occurrences reported by the annotator). Then, $P = C/O$, $R = C/N$, and $F_1 = 2PR/(P + R)$.

In Table 1 we present the results for the 10 runs in the experiment. The training set includes 189 *CMD* regions. The SVM annotator found 187 regions, with 182 true positives. The CRF annotator found 185 regions, with 181 true positives. Both performances are practically the same, as the similar F_1 scores show.

Experiment 2: Step Annotation

We conducted two 10 fold cross-validations. One was on a training corpus consisting of the 182 atomic instructions correctly identified in the step segmentation experiment by the SVM annotator. The other was on a corpus of 200 atomic instructions, collected from the documentation of an industrial application server giving step by step instructions for several administrative tasks. We manually labeled both training corpora with *ACT*, *TGT*, *VAL*, *LOC* and *DSC* labels. As in step segmentation, we experimented with SVM and CRF-based chunking. We present precision, recall and F_1 for both chunkers in Table 2 and 3.

	SVM			CRF		
	Prec.	Rec.	F_1	Prec.	Rec.	F_1
ACT	99.45	99.45	99.45	99.44	98.35	98.90
DSC	86.05	77.08	81.32	88.64	81.25	84.78
LOC	93.33	66.67	77.78	88.89	76.19	82.05
TGT	88.21	93.94	90.99	88.89	88.89	88.89
VAL	87.01	84.81	85.90	90.53	85.15	87.76
Total	91.65	91.98	91.81	92.65	90.16	91.39

Table 2. Step annotation (corpus #1)

	SVM			CRF		
	Prec.	Rec.	F_1	Prec.	Rec.	F_1
ACT	99.00	99.50	99.25	98.51	99.50	99.00
DSC	70.97	57.89	63.77	65.62	55.26	60.00
LOC	73.08	82.61	77.55	67.86	73.08	70.37
TGT	93.03	92.57	92.80	92.96	91.58	92.27
VAL	93.43	94.52	93.88	93.15	93.15	93.15
Total	93.06	92.02	92.54	92.13	91.28	91.71

Table 3. Step annotation (corpus #2)

On the first corpus, our SVM annotator correctly chunked 164 of the 182 atomic commands (90.11%). The CRF annotator correctly chunked 158 of 182 (86.81%). Overall results are similar in both cases. Individually, results are generally good for each region except location, which shows a low recall (66.67% with SVM chunking, and 76.19% with CRF). This means that we are missing approximately one third of the locations in the test corpus. Of all these missed location tokens, approximately 60% were mislabeled as targets. This happens because in some cases targets "take precedence" over locations. For token sequences of the form $\{$*preposition*, "the", *LABEL*, *type*$\}$ (e.g., `in the save dialog`), the preposition is discarded (labeled as *O*), so the chunker sees a sequence denoting a target: $\{$"the", *LABEL*, *type*$\}$. We believe that the reason is that there are much more targets than location regions in our test corpus (the locator/target ratio is 0.092%), so our model lacks enough training to properly detect all location regions. We plan to continue our experiments with training data featuring a higher locator occurrence ratio.

Results are similar on the second corpus. The SVM annotator chunked correctly 183 out of 200 steps (91.5%); the CRF annotator 181 out of 200 (90.5%). The ratio of correctly annotated steps is above 91%; and precision, recall and F_1 measure are high for the *ACT*, *TGT*, and *VAL* labels. We are missing locations, which is reflected in low precision and recall for the *LOC* regions. Again, we are annotating some locations as targets: approximately 70% of the missed locations were misclassified as targets. Descriptions show a particularly low performance. Given that this corpus was extracted from a book, descriptions explaining instructions are very verbose in some cases, complicating their recognition.

[1] `http://chasen.org/~taku/software/yamcha`

[2] `http://crfpp.sourceforge.net/`

	SVM			CRF		
	Prec.	Rec.	F_1	Prec.	Rec.	F_1
ACT	99.45	99.45	99.45	99.45	99.45	99.45
DSC	85.37	72.92	78.65	87.18	70.83	78.16
LOC	86.67	65.00	74.29	63.64	33.33	43.75
TGT	81.40	83.47	82.43	80.33	81.70	81.01
VAL	72.31	60.26	65.73	88.06	75.64	81.38
Total	86.79	83.87	85.30	87.92	83.87	85.84

Table 4. Step annotation, no label replacement (corpus #1)

	SVM			CRF		
	Prec.	Rec.	F_1	Prec.	Rec.	F_1
ACT	94.87	95.85	95.36	94.90	96.37	95.63
DSC	43.75	18.92	26.42	84.62	29.73	44.00
LOC	72.73	76.19	74.42	71.43	47.62	57.14
TGT	82.41	83.67	83.04	78.95	84.18	81.48
VAL	82.19	83.33	82.76	83.10	81.94	82.52
Total	85.54	82.76	84.13	83.10	81.94	82.52

Table 5. Step annotation, no label replacement (corpus #2)

Experiment 3: Contribution of Label Replacement

To get a measure of how much label replacement contributes to good performance in step annotation, we repeat the step annotation experiments, skipping the label replacement stage. We give the results from this experiment in Tables 4 and 5.

On the first corpus, the SVM annotator correctly chunked 141 of the 182 instructions (77.47%). The CRF annotator correctly chunked 144 of the 182 instructions (79.12%). As expected, the performance in target recognition dropped noticeably. We move from a F_1 around 90 with label replacement, to approximately 80 in this experiment.

We get similar results for the second corpus. The SVM annotator chunked correctly 153 of the total 200 instructions (79.27%); the CRF annotator chunked properly 155 of the total (77.50%). One noticeable result is the very low recall and precision we obtain for descriptions from both annotators. For the same reason as in the previous experiment, recognizing descriptions is challenging for book instructions, and even more in face of target labels, which can also be long chunks of tokens. Therefore, we conclude that label replacement significantly increases the performance of our tuple extraction algorithm.

Experiment 4: Interpreting Tuples

Our manual test automation procedure should keep user intervention low. Otherwise, it degrades to step by step execution or recording approaches. Our last experiment aims to find out how much user intervention is required in practice to execute a test. For this, we execute with our tuple interpreter the 164 (out of 182) correctly annotated steps obtained from our step annotation experiment on the first corpus.

Total test steps	182		
Properly labeled	164	(90.11%)	
	#	% properly labeled	% total
Executed	129	78.66%	70.88%

Table 6. Tuple execution without user intervention

We measure how many steps could be executed without user intervention, relative to the number of correctly annotated steps (i.e., 164) and also relative to the number of total atomic steps (i.e., 182). Table 6 summarizes the results. Out of the 164 properly annotated test steps, our system executed 129 (i.e., 78.66% of them) without any kind of user intervention. The remaining 35 steps referred to ambiguous targets that required user intervention in order to be executed. Relative to the total number of atomic steps, we could execute 70.88% of them without aid from the user.

Given the total 182 atomic instructions, our procedure correctly processed and executed 70.88% (more than two thirds) without any user intervention. And furthermore, user intervention will not be needed in future executions. Recording, on the other hand, would require executing all the steps in a test manually, one by one, before having a script that could be executed without user intervention. Our approach leads to the same results with one third of the work, and the resulting artifacts are more easily maintainable than the programs or scripts obtained with traditional recording.

DISCUSSION AND FUTURE WORK

Our approach requires users to transform a manual test into a value-free test by manually replacing values with variables. Although value replacement needs to be done only once, it still poses an overhead to the test automation process. But how much? Our corpus of 154 instructions required 49 replacements, roughly 1/3 of the instructions. We believe this overhead is acceptable for a do-only-once task. However, we would like to automate value replacement as much as possible.

Label replacement looks like an arbitrary stage. But besides normalizing seemingly different steps, replacing labels can help us to avoid tuple extraction mistakes. Consider the step `click Click Here and win an IPod`. This step just instructs the user to press a (suspicious) “Click Here and win an IPod” button. But this label will fool the segmentation stage, resulting in two tuples: (***click***, *Click Here*, ●) and (***win***, *IPod*, ●). By replacing this step’s label we get `click LABEL(Click Here and win an IPod)` , from which we can extract the expected (***click***, *Click Here and win an IPod*, ●) tuple.

Tuple extraction might deliver the wrong results occasionally. Our tuple extraction algorithm does not attempt to do error detection, instead we simply assume that extraction errors will surface at execution time. Our experiments show that this is the case. All the errors we found manifested as non-existent or ill-formed targets. Our interpreter is prepared to report inexistent targets as well as unrecognized actions (which besides an error, might signal that the action set needs to be extended). Although possible, we have not encountered steps wrongly converted to tuples denoting another valid or ambiguous instruction.

Our approach has a number of limitations that we plan to address in future work. The most immediate are:

- Processing arbitrary position relations (as in `click the activate checkbox in the 10th row`), and using such relations to adjust ambiguous test steps.
- Interpreting instructions not referring to actions on the GUI, such as verification steps.
- Handle high-level steps representing multiple actions without stating them explicitly (e.g., `add item to shopping cart`). There is work in this area that we plan to use as a starting point for this task [2].

In addition, we would like to add more variety to our training data, since writing styles can vary wildly among testers. The more and diverse the training data, the better our chunk annotators will handle such variance. We also plan to work on a more robust algorithm for label replacement, based on named entity recognition – despite being unlikely in practice, there are pathological cases that challenge label replacement, e.g., a step like `click the "click the"`. A similar entity recognition-based strategy could help to automate the value replacement stage as well.

RELATED WORK

Thummalapenta et al. [20] describe an approach for functional test automation based on brute force search. They implemented a heuristic procedure that tries all the possible interpretations of an instruction until one can be executed, backtracking in case of failure. But using heuristics can be problematic for tests including "unseen" instructions, since they will need to modify their heuristics in order to adapt to new cases. Our machine learning-based tuple extraction algorithm is easier to extend. We simply have to enrich our training set with new examples and retrain, without any need to modify our algorithm. We believe that the two approaches can be combined to get a more robust and easily extensible test automation system.

Lau et al. [12] experiment with 3 approaches (keyword, grammar and machine learning-based) to compute (A, T, D) tuples from a corpus of written how-to instructions. The corpus was obtained by crowdsourcing the task of writing how-to instructions on Amazon's Mechanical Turk. Our tuple extraction algorithm aims to solve the same problem, but with some important differences. Our algorithm utilizes domain knowledge about the application to identify labels in an instruction. In addition, we do step segmentation, while their work assume that how-to instructions are already segmented. Finally, we bridge the gap between tuple extraction and execution by providing an interactive tuple interpreter algorithm.

Branavan et al. [2] present a system to execute hand-written instructions. The salient feature of their work is the distinction between low-level and high-level instructions (i.e., instructions specifying a goal to achieve without explicitly stating all the required steps). The authors process low-level instructions by means of reinforcement learning, using a log-linear policy function to map instructions to actions. For high-level instructions, the authors map them into a sequence of low-level instructions by means of an extension to the original reinforcement learning model. Adopting these ideas for processing high-level instructions would be enormously beneficial to our approach. For example, we could process a high-level instruction like `login as Admin` by mapping it into the low-level steps `enter username Admin`, `enter password Admin`, and `click Login`.

Other authors also explore the use of machine learning techniques to process natural language navigation instructions. Chen and Mooney [4] discuss a system that transforms natural language directions into an executable plan. They use a corpus of (instruction, actions, world state) tuples to learn a parser translating instructions into navigation plans. Kollar et al. [10] present a system that follows natural language directions. They use conditional random fields [11] to derive a linguistic structure (spatial direction clauses) from a set of directions. Their system uses these spatial direction clauses to infer an optimal path in a map, according to a model based on the meaning of spatial prepositions. Shimizu [16] uses machine learning to chunk a sequence of instructions in natural language, inferring the action taken by each step from the chunk labels. This scheme is later improved by Shimizu and Haas [17], introducing the notion of "template" labels which can be parametrized with so-called slot fillers. In these two systems, the goal is to guide a robot to some destination. The domain of discourse is limited to moving straight to the end of a hallway, turning right or left, or entering some door. We rely on similar techniques for extracting tuples from a test step, but on a larger scale: we handle several kinds of actions and a broader variety of targets.

There is a large body of work proposing techniques for generating software artifacts from textual descriptions. Tichy et al. [21] argue that progress in natural language processing will leverage the construction of tools for deriving test cases, UML models and source code from natural language input. Fantechi at al. [7] utilizes natural language analysis to perform quality evaluation of use cases. Sinha et al. [19] present an engine for use case translation and analysis. This linguistic engine is used to implement Text2Test [18], a platform for authoring and verifying test cases.

The idea of interpreting manual tests described using natural language stems from CoScripter [13], a system for recording and automating actions taken on a web application. Our machinery for identifying labels in test steps and resolving targets at runtime is based on CoScripter's label finding algorithms and heuristics.

CONCLUSIONS

We presented an approach to functional test automation, consisting of an algorithm for extracting the actions denoted by a test step and an interpreter for executing such actions with user guidance. We performed a series of experiments on our prototype in order to quantify the effectiveness of our approach, as well as to analyze how much user intervention our approach requires in practice.

With an F_1 measure above 90% in the step segmentation and annotation stages, we believe that our tuple extraction algorithm can be useful in practice. Our last experiment shows that approximately 70% of the 182 test steps of our test corpus can be executed without user intervention. We think that these are good indicators of the feasibility and potential usefulness of our approach.

For many organizations manual functional testing is a nuisance they are used to, mainly because of the shortcomings in current tools for test automation. We believe that the contributions in this paper constitute a step towards improving current functional test automation technology.

ACKNOWLEDGMENTS

We thank S. Tummalapenta for his help and the anonymous reviewers for their insightful comments.

REFERENCES

1. Abney, S. P. Parsing by chunks. In *Principle-Based Parsing: Computation and Psycholinguistics* (1991), 257–278.
2. Branavan, S. R. K., Zettlemoyer, L. S., and Barzilay, R. Reading between the lines: learning to map high-level instructions to commands. In *Proc. of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10 (2010), 1268–1277.
3. Brill, E. A simple rule-based part of speech tagger. In *Proc. of the third conference on Applied natural language processing*, ANLC '92 (1992), 152–155.
4. Chen, D. L., and Mooney, R. J. Learning to interpret natural language navigation instructions from observations. In *Proc. of the Twenty-Fifth AAAI Conference on Artificial Intelligence* (2011).
5. Cortes, C., and Vapnik, V. Support-vector networks. *Mach. Learn. 20* (1995), 273–297.
6. Dustin, E., Rashka, J., and Paul, J. *Automated software testing: introduction, management, and performance*. Addison-Wesley Longman Publishing Co., Inc., 1999.
7. Fantechi, A., Gnesi, S., Lami, G., and Maccari, A. Application of linguistic techniques for use case analysis. In *Proc. of the 10th Anniversary IEEE Joint intl. conf. on Requirements engineering*, RE '02 (2002), 157–164.
8. Gouveia, D., Davis, C., Saracevic, F., Bocarsly, J., Chirillo, D., and Quesada, L. *Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource*. IBM Press, 2009.
9. Kit, E., and Finzi, S. *Software testing in the real world: improving the process*. ACM Press/Addison-Wesley Publishing Co., 1995.
10. Kollar, T., Tellex, S., Roy, D., and Roy, N. Toward understanding natural language directions. In *Proc. of the 5th ACM/IEEE intl. conf. on Human-robot interaction*, HRI '10 (2010), 259–266.
11. Lafferty, J. D., McCallum, A., and Pereira, F. C. N. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. of the 18th intl. conf. on Machine learning*, ICML '01 (2001), 282–289.
12. Lau, T., Drews, C., and Nichols, J. Interpreting written how-to instructions. In *Proc. of the 21st intl. joint conf. on Artificial intelligence* (2009), 1433–1438.
13. Leshed, G., Haber, E. M., Matthews, T., and Lau, T. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proc. of the 25th annual SIGCHI conf. on Human factors in computing systems*, CHI '08 (2008), 1719–1728.
14. Manber, U., and Myers, G. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput. 22* (1993), 935–948.
15. Ramshaw, L. A., and Marcus, M. P. Text chunking using transformation-based learning. In *Proc. of the Third Annual Workshop on Very Large Corpora* (1995), 82–94.
16. Shimizu, N. Semantic discourse segmentation and labeling for route instructions. In *Proc. of the 21st intl. conf. on Computational linguistics*, COLING ACL '06 (2006), 31–36.
17. Shimizu, N., and Haas, A. Learning to follow navigational route instructions. In *Proc. of the 21st intl. joint conf. on Artificial intelligence* (2009), 1488–1493.
18. Sinha, A., Jr., S. M. S., and Paradkar, A. Text2Test: Automated inspection of natural language use cases. In *Proc. of the 2010 3rd intl. conf. on Software Testing, Verification and Validation*, ICST '10 (2010), 155–164.
19. Sinha, A., Paradkar, A. M., Kumanan, P., and Boguraev, B. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *DSN* (2009), 327–336.
20. Thummalapenta, S., Sinha, S., Mukherjee, D., and Chandra, S. Automating test automation. Tech. Rep. RI11015, IBM Research, 2011.
21. Tichy, W. F., and Koerner, S. J. Text to software: developing tools to close the gaps in software engineering. In *Proc. of the FSE/SDP workshop on Future of software engineering research*, FoSER '10 (2010), 379–384.
22. Tjong Kim Sang, E. F., and Buchholz, S. Introduction to the CoNLL-2000 shared task: chunking. In *Proc. of the 2nd workshop on Learning language in logic and the 4th conf. on Computational natural language learning - Volume 7*, ConLL '00 (2000), 127–132.