Notes on Test Code Generation in BT-TCG

Sentot Kromodimoeljo

June 23, 2016

1 Introduction

Sining Cai and Long Nguyen have investigated the generation of code to test applications from test cases produced by BT-TCG. Sining looked into generating Selenium WebDriver Python code, while Long looked into generating Appium Java code (Appium also uses Selenium WebDriver). Both proposed generating code snippets that sit between some setup code and some tear-down code. The code snippets that they proposed to generate are very similar in form. It is reasonable, therefore, to want an add-on to BT-TCG for code generation that is generic and able to generate testing code for different test platforms (as long as they are web-based). This document discusses some of the issues that are involved in the design of such an add-on. The choice of issues and how they should be tackled reflect my own personal bias.

Currently, BT-TCG generates human-testing instructions based on BT test paths, translating BT nodes designated as actions and observations into corresponding human actions and observations. The translations are specified using the `actions` and `observables` tabs in BT-TCG. Rather than having additional tabs, it would make sense to use the same tabs to specify translations for generating testing code. Without adding any functionality to BT-TCG, code snippets can already be manually entered as translation entries in the tabs. However, this process can be laborious and highly repetitive, thus we want some degree of automation.

The rest of this document starts with notes on frequently used patterns in test code snippets. Variations on patterns are then discussed, as well as patterns for "modifiers." The final section proposes an approach for integrating code generation into BT-TCG.

2 Frequently used patterns in test code snippets

One of the frequently used patterns in code snippets produced by both Sining and Long is for checking the text value of a web-page element. The pattern for checking the text value of an element essentially looks as follows (WebDriver with Python syntax):

```
if driver.find_element_by_id("component").get_text() == "state" then:
  code for success
else:
  code for failure.
```

In Java, the pattern looks as follows:

```
if (driver.findElementById("component").getText().equals("state")) {
  code for success
} else {
  code for failure
}.
```

Typically, the code snippet is for an observation and corresponds to a state realisation node in the BT model. The element identifier corresponds to the component name in the BT node, while the text value corresponds to the state. However, the correspondences might not be verbatim. For example, the element identifier might be `"vectorstatus"` but the corresponding BT component name might be `"Vector"`. Thus in general a mapping is required between BT component—state pairs and the element identifier—value pairs. Note that a mapping from BT component—state pairs, can be viewed as a mapping from state realisations. The mapping can be more general than having an element's `id` be determined by the corresponding BT component and the element text value be determined by the BT state.

Sometimes we want to check an element's attribute value instead of its text value. The pattern becomes:

```
if driver.find_element_by_id("comp").get_attribute("attr") == "val" then:
  code for success
else:
  code for failure.
```

This corresponds naturally to an attribute assignment (a state realisation for an attribute of a component) in a BT model. For attribute checking, the required mapping is to element identifier—attribute—value triplets.

For actions, a frequently used simple pattern is for clicking on an element:

```
driver.find_element_by_id("component").click().
```

Only the element's `id` is needed to instantiate the pattern. Equally simple but less frequently used than clicking are patterns for clearing an element and submitting an element (e.g., a form):

```
driver.find_element_by_id("component").clear(),
driver.find_element_by_id("component").submit().
```

A less simple action pattern is one for entering text into an element:

```
driver.find_element_by_id("component").send_keys("text input").
```

The required mapping would be rather straightforward if the text is automatically generated.

3 Pattern variations, "modifier patterns"

An obvious variation on the patterns of Section 2 is in the way elements are found. Instead of using `id` to find an element, there may be cases where the element may need to be searched by other attributes such as `name`, or using `xpath`.

Another variation is to use the test platform's functionality to handle actions and observations. For example, with WebDriver, the `assert_equals` (or `assertEquals`) method can be used instead of the *if-then-else* pattern for checking the text value described in Section 2, although it would be less general (it restricts the *then* and *else* parts of the *if-then-else*).

An action or observation may need to take into account special circumstances. For example, the element of interest may be in a different web page (for example if `iframes` are used), in which case the context may need to be temporarily changed. The temporary context switch may be viewed as a modifier to the original pattern:

```
driver.switch_to().frame(... find frame ...)
... original pattern ...
driver.switch_to().default_content().
```

An observation may need to wait for some event to be completed (e.g., wait until an element is visible or until a page is loaded) or delay the observation for some period of time. A delay modifier might be as follows:

```
wait(1000)
... original pattern ...
```

More sophisticated waits — e.g., waiting for an event but with timeout — are provided by test platforms such as WebDriver. It may be worthwhile to look into how this issue is handled by other test platforms such as SOAPUI and Roboelectric.

4 Integration into BT-TCG

I have identified the following as the main concepts that are needed to support BT-TCG test code generation:

- Name mapping between BT model and the test code. This can be generalised to a mapping between what are called *node profiles* in BT-TCG to tuples of names/strings in the test code.
- Templates for the patterns discussed in the previous sections.
- Setup and tear-down code.
- Language/platform binding.

BT-TCG already has the `actions` and `observables` tabs where the translation from BT nodes to a target language (currently for human testing, but can be used for test code, since the target is in free text form). It would be sensible to specify the name mappings in these tabs. Templates would then be invoked by right clicking in the free text area of the tabs. Language/platform binding can be early, in which case the result of applying a template produces a language/platform specific snippet in the free text area, or late, in which case applying a template might produce pseudo-code in the free text area which will be transformed to code for a specific language/platform in the code generation phase. Early binding would likely be easier to implement.

A new tab for code generation would be desirable. This tab would provide facility for managing setup and tear-down code, selecting test cases or the full test plan, and specifying parameters for the test code.