AbsCon

A Test Concretizer for Model-based Testing

Vanhecke, Jeremy; Devroey, Xavier; Perrouin, Gilles

DOI
10.1109/ICSTW.2019.00027

Publication date
2019

Published in
Proceedings of the 2019 IEEE 12th International Conference on Software Testing, Verification and Validation Workshops

Citation (APA)
Vanhecke, J., Devroey, X., & Perrouin, G. (2019). AbsCon: A Test Concretizer for Model-based Testing. In *Proceedings of the 2019 IEEE 12th International Conference on Software Testing, Verification and Validation Workshops: 15th Workshop on Advances in Model Based Testing (A-MOST '19)* (pp. 15-22). [8728920] IEEE. https://doi.org/10.1109/ICSTW.2019.00027

AbsCon: A Test Concretizer for Model-based Testing

Jeremy Vanhecke
IBA
Louvain-La-Neuve, Belgium
jeremy.vanhecke@iba-group.com

Xavier Devroey
Delft University of Technology
Delft, The Netherlands
x.d.m.devroey@tudelft.nl

Gilles Perrouin
PReCISE / NaDI,
Faculty of Computer Science,
University of Namur
Namur, Belgium
gilles.perrouin@unamur.be

***Abstract*—Test definition and execution is an essential but time-consuming task during system development. To speed up the process, model-based testing and other related approaches propose to generate/write abstract test cases and to concretize them using either transformations, an adapter, or a mixture of the two. QTaste is an industrial data-driven test case definition and execution environment used to perform black-box testing on various kinds of systems. QTaste's test cases are manually written in Python and use an adapter, called test API, to execute operations on the System Under Test (SUT) interfaces. In this paper, we describe AbsCon (*Abstract test case Concretizer*), a plugin used to generate test cases executable in QTaste based on their definition: *i.e.*, sequences of abstract actions and assertions. AbsCon uses programmer friendly mappings (written in Python) for the SUT's interfaces, actions, and assertions, to generate standard test cases in QTaste format. Rather than having a complete model-based testing transformation chain, the plugin is bridging the gap between existing model-based test case generation tools and an industrial test case execution system using a mix of transformation and adaptation.**

***Index Terms*—Test case concretization; software testing tool; QTaste**

I. Introduction

During software development, a lot of effort is put on testing the produced system. Test cases are executed on this System Under Test (SUT) to detect bugs as soon as possible. In order to ease the required effort, researchers and industries created different techniques to automate the test cases execution using dedicated test management tools on the one hand, and to automatically derive relevant test cases in order to detect bugs on the other hand. The latter is a well studied problem in Model-Based Testing (MBT) [1], where the overall goal is to define a suitable set of test cases based on a model of the SUT and some selection criteria. To obtain a set of executable test cases, MBT tools usually proceed in two steps: first a set of abstract test cases is generated from the model based on the selection criteria. Those test cases contain abstract actions and assertions over the SUT but cannot be executed as-is (except if a manual testing process is envisioned). In the second step, MBT tools concretizes those abstract test cases into executable test cases, based on mapping information provided by the test engineer. This mapping may take one of the following forms [1]: (i) an adapter which interprets the actions and assertions of the abstract test case and execute them on the SUT; (ii) a transformation from the abstract test cases to code executable directly on the SUT; or (iii) a mixture of the above two. In this last case, an abstract test case is transformed into executable code which uses an intermediate adapter (like an intermediate library for instance) to bridge the gap between the test case and the SUT. Other development/testing approaches, like Behavioural Driven Development (BDD), heavily rely on test case conretization. In a BDD process, requirements are defined as expected behaviours of the system. Those requirements are expressed using abstract test cases, which are conretized to form a set of acceptance tests. The complete process is supported by tools like Cucumber [2].

However, such approaches barely consider that abstract test cases can be generated from other techniques, yielding a *mapping* problem. This requirement stems from our research on testing variability intensive systems, which generate abstract test cases from variability-aware transition systems [3]–[5].

In this paper, we describe *AbsCon* (Abstract test case Concretizer). AbsCon is defined as a QTaste[1] plugin, an open-source industrial data-driven test case definition and execution environment, used to perform black-box testing on various kinds of systems. QTaste abstracts the SUT's interface by using an adapter called *test API*, test cases are written in Python where the operations on and the readings from the SUT's interface are encapsulated into calls to the test API dedicated to the kind of the SUT. For instance, to test a Web-application, QTaste encapsulates the access to the elements of the Web page in a Web test API which is responsible to perform the effective Selenium (a popular Web browser automation tool [6]) calls. After considering different options, we chose to define AbsCon as a QTaste plugin for the following reasons: (i) QTaste is an open source industrial tool, used to test various kinds of systems, from Web-applications to mobile applications and even cyber-physical systems [7], thanks to its test API adaptation mechanism; (ii) plugin development is already included in QTaste and this architecture was suggested

Gilles Perrouin is a research associate at the FNRS. This research was partially funded by the EU Project STAMP ICT-16-10 No.731529, the NIRICT 3TU.BSR (Big Software on the Run) project as well as by the European Regional Development Fund (ERDF) "Ideas for the future Internet" (IDEES) project.

[1] http://www.qtaste.org

by a QTaste developer the first author contacted; (iii) the inital goal of AbsCon was to concretize abstract test cases generated by our Variability Intensive System Behavioural teSting framework (VIBeS) [8] using additional mapping information. To this end, abstract test cases are defined in AbsCon using an XML file, where each test case is a sequence of actions and assertions on the SUT. But this definition is not specific to VIBeS, it also allows QTaste test engineers to define test cases in a more abstract and systematic fashion (rather than directly Python scripts), as long as they follow the same pattern (*i.e.*, sequences of assertions and actions).

The remainder of this paper is as follows: Section II gives a general description of the QTaste environment, Section III describes AbsCon's concretization process as well as the required mapping information, Section IV presents AbsCon's implementation, advantages and limitations are discussed in Section V, Section VI discusses related work. Finally, Section VII wraps up with conclusions and future work.

II. Test automation using QTaste

The *QSpin Tailored Automated System Test Environment* (QTaste) [9] is an open source functional and non-functional black-box test environment developed in Java and Python. It has been originally developed by Qspin Experts[2] in order to automate testing process of medical cyber-physical systems developed by IBA[3] and used for proton therapy. Since its inception, QTaste has been extended to support different kinds of SUTs, like Web-applications, mobile applications, or more classical desktop applications [7]. It is released as an open source project on GitHub under GNU GPL 3.0 license [9].

A. Overview of the QTaste environment

QTaste follows the data-driven testing philosophy [10]: data used by the tests are externalized in order to allow test cases parametrization. Each test case is written in Python and describes a sequence of steps, *i.e.*, *operations* executed by the SUT or *verifications* of the outputs produced by this SUT, using the given data as input. For instance, when testing a form which values are recorded in a database, one test case fills the form with the given data and check that the values are effectively recorded in the database. This test case is repeated with different values (*e.g.*, positive, null, and negative values for numeric fields) specified in a separate CSV file and automatically executed by QTaste on the SUT.

QTaste provides *test APIs* which communicate with the SUT and manage the operations executions and/or SUT's outputs reading. Each test API consists in a Java interface, defining the operations and information accessible by the test cases, and a Java implementation of this interface which manages communication with the SUT. This mechanism allows QTaste to test a large variety of systems: Web-applications using a Selenium-based test API, hardware components with dedicated API, or any other kind of system for which a test API may be developed. The test API, together with the configuration of the SUT instance is called a *Testbed*: this mechanism allows to write test cases independently from the execution environment, using only test (and standard Python) API(s). Once all the test cases have been executed, QTaste generates a summary report, with the number of success and fails, the Testbed used, for each test case, the CSV lines used, *etc.*.

[2]http://www.qspin.be

[3]https://iba-worldwide.com

Listing 1. Google search test case

```
1  from qtaste import *
2
3  api = testAPI.getSelenium(INSTANCE_ID='Google')
4
5  def init():
6    api.openBrowser(testData.getValue("BROWSER"))
7    api.windowMaximize()
8    api.open("https://www.google.be/")
9    api.waitForPageToLoad("15000")
10   if api.getTitle() != "Google":
11     testAPI.stopTest(Status.FAIL)
12
13 def searchAndClick():
14   api.type("id=lst-ib", testData.getValue("
         SEARCHVALUE"))
15   api.clickAt("name=btnK", "0.0" )
16   api.waitForPageToLoad("15000")
17   api.click("link=" + testData.getValue("
         LINKTOCLICK"))
18   if api.getTitle() != testData.getValue("
         LINKTITLE"):
19     testAPI.stopTest(Status.FAIL)
20
21 def exit():
22   api.stop()
23
24 doStep(init)
25 doStep(searchAndClick)
26 doStep(exit)
```

Listing 1 presents a (simplified) test case for the Google search engine that is executed for each line of the external CSV file. It launches a Web browser and connects to the Google search website, fills the search field with a string, and click on a specified link. Line 3 creates a Selenium instance test API, which manages the connection to the browser; lines 5, 13 and 21 declare the steps of this test case, called at lines 24, 25, and 26; explanations about each step is given as a comment in Python format (not shown here) and is used during the generation of the test reports. At each step, the test API instance is used to manipulate the browser user interface (lines 6 to 10, 14 to 18, and 22) according to the data provided in the external CSV file (identified by column names at lines 6, 14, 17, and 18). Finally, each step may check assertions on the outputs of the browser to validate the execution (lines 10 and 18).

B. Advantages and limitations

The main advantage of QTaste is the test API mechanism, allowing test cases to manipulate a large variety of SUTs using a general purpose programming language: Python. Expressing test cases using a general purpose and popular programming language like Python benefits from the large number of available Python libraries. This can be very handful when

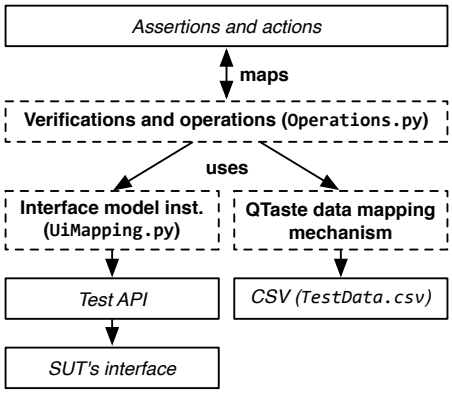


Fig. 1. Mappings in AbsCon

writing test cases in order to perform more complex operations or access external resources. The environment provides extensibility mechanisms to the test engineers in order to write dedicated adapters between QTaste and SUTs, and describes the usage of those adapters in a test API. Coupled to the externalisation of data and SUT's configuration, it improves test cases *reusability* and *automation* of the test process [1]. As it works as a black-box test environment, QTaste access the SUT through its interface, manipulated by the test cases trough the test API. This means that whenever the interface and/or the test API evolve, all the test cases using this interface and/or modified test API are impacted, increasing *maintenance cost* [1]. AbsCon provides an additional abstraction layer separating the different concerns thus reducing maintenance costs when combined with abstract test cases as presented in the next section.

III. Concrete test cases generation using AbsCon

AbsCon (stands for *Abstract test cases Concretizer*) was originally developed to support abstract test case concretization [11]. In our case, an *abstract test case* is a sequence of abstract assertions and actions, usually automatically derived by a model-based testing tool [1] (VIBeS in this case [8]). The concretization process translates the abstract test case into a (concrete) test case executable by QTaste: (resp.) assertions and actions are *mapped* to (resp.) verifications and sequences of operations manipulating the SUT through the test API. The most common way to perform this task is to give, for each assertion and each action, the corresponding Python code. It allows to improve the reusability and automation, while decreasing the maintenance costs (each assertion or action is defined only once in the mapping).

However, access to the SUT's interface elements remains hard coded in the different test cases (*e.g.*, lines 10 or 15 in Listing 1). This potentially raises some issues: (i) element of the SUT's interface are accessed using test API methods, requiring to know and provide at each method call the *access method* (*e.g.*, using the element's `id` or `name` or at lines 14 and 15 in Listing 1) and the *access value* (*e.g.*, `lst-ib` at line 14 and `btnK` at line 15 in Listing 1); (ii) besides being

Listing 2. Google instant search test cases in AbsCon XML format

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <realisation id="Google testing">
3    <uimodel>web</uimodel>
4    <uimapping>UiMappings.py</uimapping>
5    <operations>Operations.py</operations>
6    <datas>TestData.csv</datas>
7    <tests>
8      <test>
9        <action>start</action>
10       <action>goHomePage</action>
11       <assert>onHomePage</assert>
12       <action>inputSearchString</action>
13       <assert>searchResultsPrinted</assert>
14       <action>clickLink</action>
15       <assert>pageLoaded</assert>
16       <action>exit</action>
17     </test>
18   ...
19   </tests>
20 </realisation>
```

cumbersome when writing test cases, requiring access method and value in each method calls may also raise problems, as not all elements of test API may be called on all elements of the SUT's interface (e.g., for a Web-application, it is only possible to type text in text fields or in text areas), which will only be checked when running the test case; (iii) as previously, when an interface or test API element is updated, all the actions and/or assertions using this element are impacted, requiring to update the mapping in different places and thus increasing the maintenance cost (with a more limited magnitude).

To mitigate those issues, we divide the mapping in AbsCon in 3 (as illustrated by the dashed boxes in Figure 1): a SUT's interface elements mapping trough a *model instance* of this interface; a *data mapping*; and an *assertions and actions mapping*, giving for each (resp.) assertion/action the (resp.) verifications/operations to perform on the SUT. The verifications and operations on the SUT are defined as Python functions that will use the interface model instance, using the methods of the different elements, and the external data. The external data are recorded in a CSV file and managed using QTaste's dedicated mechanism. The interface model, *i.e.*, a set of Python classes, uses one or more test APIs in order to execute the operations and retrieve information on/from the SUT.

The model of the interface and the assertion/actions mapping is detailed in the following sections. To illustrate those different mappings, we will use Google instant search as SUT and consider the following test cases:

(1) open Google search website, enter a keyword, see that search results are printed, click on a link, and check that the website is loaded;
(2) open Google search website, enter a keyword, see that search results are printed, deactivate the instant search in the parameters and validate, go back to the main page, and check that search results are not printed when a keyword is entered;
(3) open Google search website, enter a keyword, see that

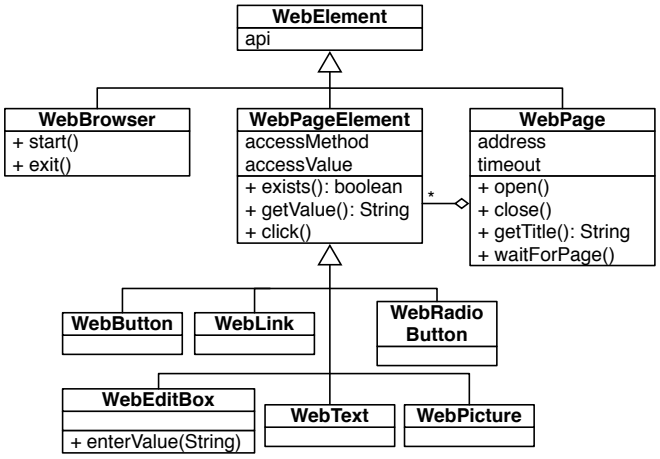


Fig. 2. Web-applications SUT's interface class diagram (`web`)

search results are printed, deactivate the instant search in the parameters and cancel, go back to the main page, and check that search results are printed when a keyword is entered.

Test case 1 checks the common usage of Google instant search, test case 2 checks that, when the instant search is deactivated in the parameters options, the instant search is not performed, and test case 3 checks that when instant search is deactivated but the change is cancelled in the parameters options, instant search is still active. Test cases are defined in XML format as a sequence of assertions and actions: lines 8 to 17 in Listing 2 gives test case 1 definition (other test cases are omitted). Additional information are the SUT's interface model to use (line 3), the path to the Python file defining the instance of this model for the Google instant search interface (line 4), the path to the Python file defining the operations mapping (line 5), and the path to the external CSV data file (line 6).

A. SUT's interface model

A SUT's interface model describes for a particular family of SUTs the different elements accessible when performing black-box testing. For instance, for Web-applications, the `web` model at line 3 in Listing 2 is defined (here using a class diagram notation to ease the reading) in Figure 2. It describes the different elements we can found on a Web page: each class has to access the QTaste Selenium test API (using inherited attribute `api`) and will extend `WebElement`; `WebBrowser` objects will `start` and `exit` the Web browser specified for the current test case execution using the data from the external CSV file (as on line 6 in Listing 1); a `WebPage` is available at a given URL `address`, may be opened (and expected to load before a given `timeout`), closed, and has a title; `WebElements` will appear on this page, each one is accessible using an `accessMethod` (*e.g.*, XPath) and an access value (*e.g.*, an XPath query to this element), may or may not exist on the page, has a value, and may be clicked; the different elements we identified (relevant for the examples of this paper) are `WebButton`, `WebLink`, `WebRadioButton`, `WebText`, `WebPicture`, and `WebEditBox` which may be filled using textual values.

Listing 3. Google instant search interface model instance (`UiMapping.py`)

```
1  from uimodel_web import *
2
3  #mapping definitions
4  googlePage = WebPage("https://www.google.be/",
       5000)
5  searchBar = WebEditBox("id", "lst-ib")
6  searchButton = WebButton("name", "btnK")
7  disableInstSearch = WebRadioButton("xpath", "//
       div[@id='instant-radio']/div[3]/span")
8  enableInstSearch = WebRadioButton("xpath", "//
       div[@id='instant-radio']/div[2]/span")
9  ...
```

In AbsCon, each interface model is defined in Python. It is instantiated to represent the SUT's interface. For instance, for Google instant search, the `web` model instance is defined in `UiMapping.py` (Listing 3), as specified at line 4 in Listing 2. Each object is built using the dedicated constructor, which will require in most cases an access method and an access value: *e.g.*, search bar is accessed using its `id` in the page, which is `lst-ib` (line 5), or using an XPath expression (lines 7 and 8). As for test APIs, interface models may be reused across different SUTs, as long as they share the same kind of interface (Web pages in this case).

Another option to the modelling offer described here would be to use User Interface Description Languages (UIDLs) [12] such as USIXML [13]. These languages provide generic constructs (organized in one or more metamodels that represent both platform independent and platform specific views, according to Model-Driven Architecture [14] principles) allowing to model any kind of user-interface (including non-conventional interfaces such as voice-enabled ones). However, the use of such proposals in ours raises the following problem: the number of concepts they are offering being quite large, modelling a simple user interface can be cumbersome and complex, unless we tailor the language to specific needs. In our context, we do not try to model the whole user interface but the subset concerned by the tests. We therefore adopted a lightweight approach that has the complementary advantage of not requiring any new modelling language to learn, by exploiting Python's object-orientation facilities. Furthermore, as initially mentioned, QTaste's spectrum is larger than testing graphical user interfaces.

B. Assertions and actions mapping

AbsCon extracts assertions and actions from the abstract test cases (Listing 2). Each assertion is mapped to a verification (*i.e.*, a function returning true or false) over the SUT's interface; and each action is mapped to a sequence of operations over elements of the SUT's interface (*i.e*, again, a function). Verifications as well as operations are defined using the interface model instance defined in `UiMapping.py` (and will manipulate the different elements using the methods defined for those elements) and the QTaste data mapping mechanism

Listing 4. Verifications and operations mapping (`Operations.py`)

```
1  from qtaste import *
2  from UiMappings import *
3
4  #Actions definition
5  def goHomePage():
6    googlePage.open()
7
8  def inputSearchString():
9    searchBar.enterValue(testData.getValue("
         SEARCHVALUE"))
10 ...
11 #Asserts definition
12 def searchResultsPrinted():
13   googlePage.waitForPage()
14   if (not(navPicture.exists())):
15     time.sleep(3) # wait for loading and retry
16   return navPicture.exists()
17 ...
```

Listing 5. Generation result for test case 1

```
1  from qtaste import *
2  from Operations import *
3
4  #Assert
5  def doAssert(method, message):
6    res = method()
7    if res == 0:
8      raise QTasteTestFailException(message)
9
10 #Steps
11 doStep(start)
12 doStep(goHomePage)
13 doAssert(onHomePage, "assertion onHomePage has
       failed")
14 doStep(inputSearchString)
15 doAssert(searchResultsPrinted, "assertion
       searchResultsPrinted has failed")
16 ...
```

in order to retrieve data from the external `TestData.csv` file. The mapping between the assertions and actions from the abstract test case is done by using the same name for the verifications and operations functions.

For instance, Listing 4 presents the verifications and operations mapping (`Operations.py`) for the Google instant search test cases from Listing 2. Function `inputSearchString` (line 8) corresponds to the action with the same name in the test cases and inputs a search string, coming from the `SEARCHVALUE` column of the external CSV file, in the Google search bar defined in `UiMappings`. Function `searchResultsPrinted` (line 12) corresponds to the assertion with the same name in the test case, and returns true if the navigation picture from the result page is loaded.

C. Test cases generation and execution

Once the mappings are defined, AbsCon generates concrete (*i.e.*, executable) test cases for QTaste: for each test case, it creates a Python script which imports the mappings and executes a sequence of `doStep` and `doAssert` calls using the verifications and operations functions. Those Python files, with the `TestData.csv` file, are used as input for QTaste to execute the test cases on the SUT and generate a summary test report. Listing 5 presents the result of the generation for test case 1: each `doStep` call (part of the standard QTaste API) corresponds to one action in the test case and will execute the given function. The `doAssert` function, defined by AbsCon, calls the given function (corresponding to an assertion in the test case) and prints the given error message if the call returns false.

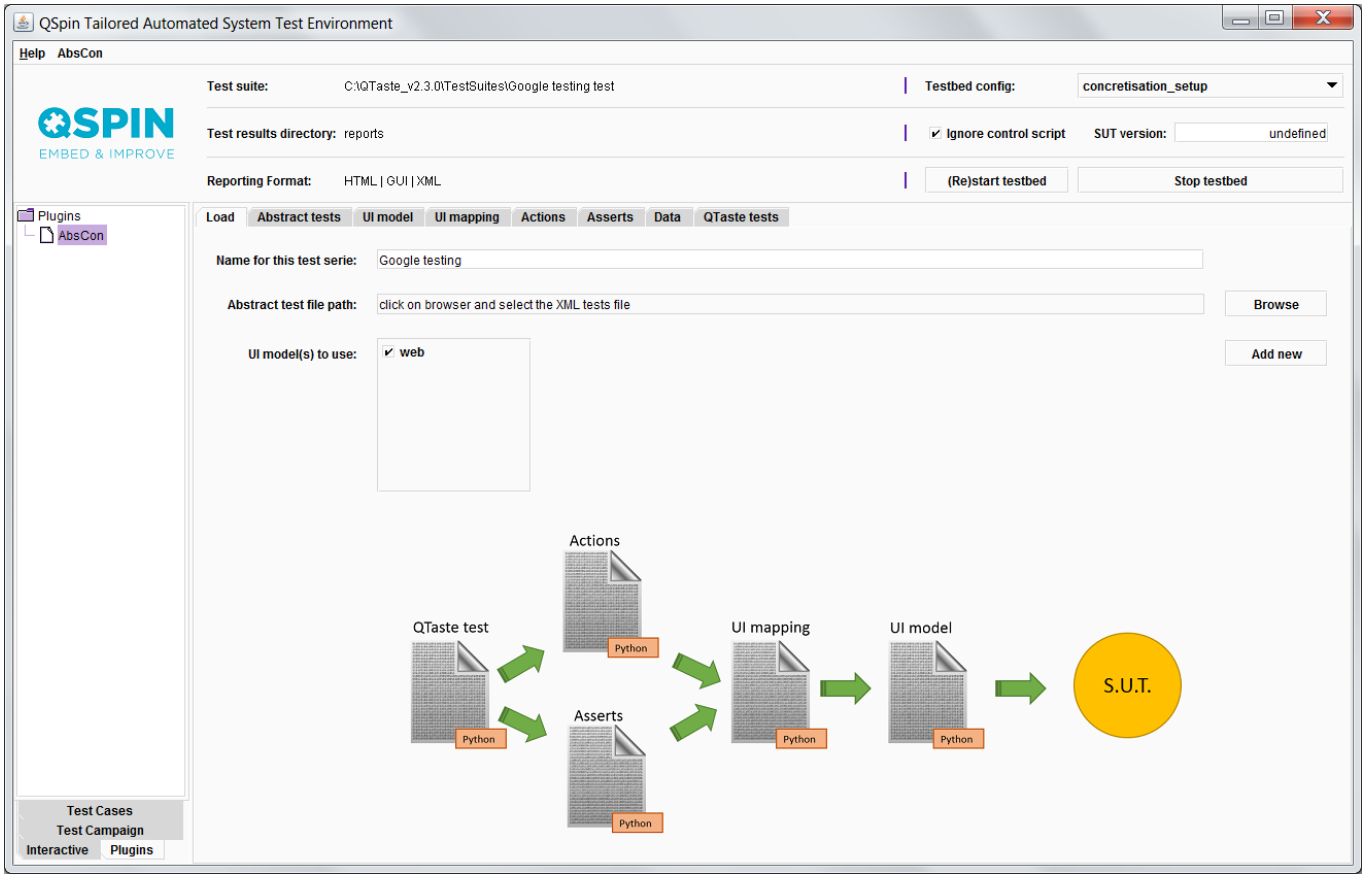
IV. Implementation

AbsCon provides a graphical user interface integrated to QTaste (as shown in Figure 3a) with different tabs, one for each mapping phase. When executing the plugin, the first step is to load the abstract test cases from an external XML file, AbsCon extracts the different actions and assertions for which a mapping has to be provided and presents them under the `Abstract tests` tab. The second step is to define or load a SUT's interface model (under tab `UI model`) and to instantiate this model under the `UI mapping` tab presented in Figure 3b: for each element of the interface, one has to instantiate a class of the interface model (selected using a drop-down list) by providing the required parameters for the constructor, and add it to the mapping using the `Declare` button (or load an existing mapping using the `Load` button). Actions and assertions mappings are given using the two next tabs: the user select the action/assertion using a drop-down list and provides the Python code for this action/assertion as shown in Figure 3c (the assertion mapping tab in this case). In the `Data` tab, the user provides the data in an editable table, and finally generates the QTaste executable test cases in the `QTaste tests` tab.

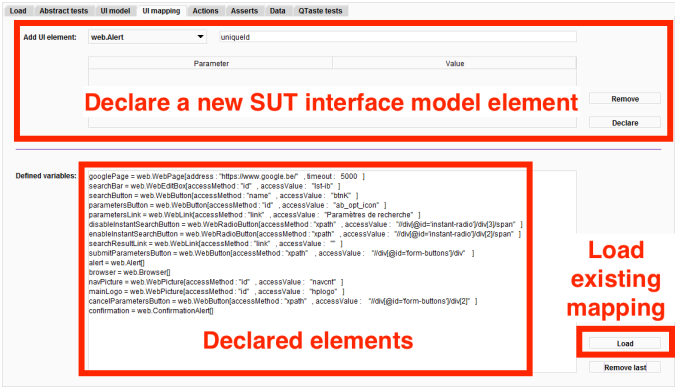
V. Discussion

AbsCon heavily relies on *abstraction*, of the SUT's interface on the one hand, and on the assertion and actions on the other hand. This abstraction layer allows to define each mapping independently from the higher or lower levels: abstract test cases are defined using assertions and actions with meaningful names for the user/test engineer; each assertion and each action is mapped to a Python function representing a verification or an operation, and is defined as a manipulation of the SUT's interface meaningful from a user point of view (depending on the nature of the SUT, the user may be a human or another system), thanks to the SUT's interface model; finally the SUT's interface model encapsulates the test API calls in charge of the effective communication with the SUT.

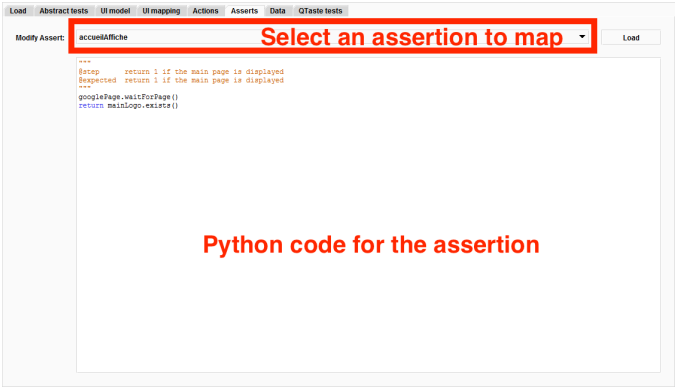
The goal of our abstraction layer is to reduce the overall complexity of the test cases and to decrease the *maintenance costs*. Indeed, when the SUT's interface evolves, only the mapping to this interface has to be (potentially) changed, AbsCon can then re-generate concrete test cases for QTaste that will serve for non-regression. This process is much lighter than the update of QTaste test cases as it will (potentially) require to update all the test cases containing code that manipulate the SUT's interface (using directly the test API



(a) Main tab



(b) SUT interface mapping tab



(c) Assertions mapping tab

Fig. 3. AbsCon plugin printscreens

in this case). In the same way, when functionalities are added to the SUT, only the new interface elements, and verifications and operations mappings have to be added. New abstract test cases may then be written for those elements and AbsCon can re-generate a complete set of test cases for the whole SUT.

The definition of the different mappings may represent an additional effort during test activities. However, different aspects have to be taken into account. First, the SUT's interface model depends solely on the nature of the SUT, *e.g.*, Web-application for the `web` model from Figure 2. Once defined, this model may be reused across different projects. As this model abstracts the test API by defining methods from the interface point of view (*e.g.*, `click`, `open`, `getTitle`, *etc.* in Figure 2), we believe that it will also soften QTaste's learning curve. Second, the definition of the mappings enables integrating existing model-based testing techniques (e.g., [1], [8]) rather than defining a new complete test development process.

In our opinion, the most time consuming task will be to *identify* and *map* the different SUT's *interface elements*. This

cost may be reduced in some cases using existing tools: for instance, *Inspect* [15] or *SwingInspector* [16] are tools used to identify and access graphical user interface elements in classical desktop applications. In our `UiMapping.py` example in Listing 3, we used Firefox's inspection tool to identify the different elements on a Web page. Depending on the nature of the SUT, this mapping may also be partially or totally automated (this will be part of our future works), like for Web-applications for which each element on a Web page describes itself using HTML tags.

VI. Related work

Test case conretization techniques are classified by Utting *et al.* [1] in 3 categories: *adaptation* approaches abstracts the SUT by using a wrapper (also called an adapter), test cases call this wrapper in order to execute operations on the SUT; *transformation* approaches transform abstract test cases into test cases directly executable by the SUT, possibly using additional information; and *mixed* approaches also transform abstract test cases in executable test cases, but using an adaptation layer in order to abstract the SUT.

Using this classification, QTaste uses adaptation to abstract the SUT using its test APIs and requires to write test cases which will use those test APIs. There exists other adapters, like *Selenium* and *Sahi* [17] to test Web-applications, or *AutoHotKey* [18] to test Windows applications. Tools like *Sikuli* [19] and *Squish* [20] provide adaptation mechanisms to perform graphical user interface testing using techniques like image recognition, or recording and playback. None of these tools natively support abstract test case concretization.

Other transformation and mixed tools like *TOTEM* [21], *SpecExplorer* [22], *MaTeLo* [23], *Smartesting* solutions [24], or *STALE* [25] implement full model-based testing approaches, including abstract test case generation and concretization from different modelling languages (e.g., UML Testing Profile [10], etc).

Rather than having a complete transformation chain (from models to executable test cases), we developed AbsCon in order to plug it on an existing approach (VIBeS [8] in this case), concretize abstract test cases, no mater their origin as long as they are described as sequences of actions and assertions, and get executable test cases on a generic and industrial test environment like QTaste.

As for VIBeS, other model-based testing approaches produce abstract test cases that are concretized using existing tools, this is the case for *Skyfire* [26] which uses a transformation approach to produce *Cucumber* [2] abstract test cases from UML diagrams. Cucumber is a popular behaviour-driven development tool that aims at producing typical examples of the behaviour of a system under development, described using a semi-structured language: Gherkin. Those examples are used as acceptance tests and concretized using a Java annotations based mechanism, mapping semi-structured sentences to Java methods using a defined string pattern. The executable test cases are run in standard JUnit environment. Cucumber could have been another test execution environment target, but, to the best of our knowledge, it does not provide any SUT's interface abstraction mechanism (like QTaste's test APIs) and would have required more effort to define a programmer friendly abstraction mechanism of this interface. FitNesse, a wiki-based integration testing approach, would also require a similar effort as it works "just below the user interface" [27].

QTaste relies on a data-driven approach: each test cases is parametrized to be executed with different data values [10]. Used in isolation, the concretization process of AbsCon also allows one to raises that level of abstraction by using a keyword-driven approach [28], [29]: keywords in the test cases (actions and assertions) are mapped to executable test code.

VII. Conclusion and future work

In this paper, we presented AbsCon, a QTaste plugin developed to concretize abstract test cases represented as sequences of actions and assertions. The adaptation mechanism provided by QTaste's test API is enhanced by a programmer friendly way to encapsulate the calls to this API using a common model specific to the kind of the SUT's interface. This model, *reusable* for different SUTs as long as their interface are of the same kind, defines the possible interactions with the SUTs. An instance of this model, specific to a SUT, is used in operations and verifications corresponding to actions and assertions defined in the abstract test cases. Using the different mappings, AbsCon is able to generate test cases executable in QTaste.

Originally developed to bridge the gap between VIBeS and concrete test cases, AbsCon offers multiple advantages, even in a non model-based testing context. We chose to implement it over an existing industrial test case management and execution tool, which will, we believe, eases its broader adoption. As a standalone tool (i.e., not used in an model-based testing chain), AbsCon enhances QTaste's *genericity* by *raising the abstraction level* of different elements: the SUT's interface and test APIs, thanks to the SUT's interface model mechanism; and the test cases themselves by allowing to provide definitions using abstract actions and assertions (which is to the user) instead of Python scripts.

So far, the plugin has only been used on small examples, a more complete validation is part of our future works. We will also explore automated SUT's interface mapping possibilities using existing inspection tools. Finally, another potentially interesting research direction is the definition of the test cases using a structured natural language (like Gherkin [2]) as an input to AbsCon instead of XML files. This could be used to automatically define, not only the actions and assertions, but also the data to use during the test cases execution. Ideally, the definition of the test cases in a structured language would be processed by AbsCon to populate both the list of assertions and actions to map, the elements of the SUT's interface to use (based on the text describing the test cases steps), and the CSV file used by QTaste.

References

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

[2] "Cucumber," https://cucumber.io.

[3] X. Devroey, G. Perrouin, P. Y. Schobbens, and P. Heymans, "Poster: Vibes, transition system mutation made easy," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 817–818.

[4] X. Devroey, G. Perrouin, and P.-Y. Schobbens, "Abstract test case generation for behavioural testing of software product lines," in *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, ser. SPLC '14. New York, NY, USA: ACM, 2014, pp. 86–93. [Online]. Available: http://doi.acm.org/10.1145/2647908.2655971

[5] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, and P. Heymans, "Featured model-based mutation analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 655–666. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884821

[6] "SeleniumHQ Web Browser Automation," http://www.seleniumhq.org.

[7] B. Doucet, "Iba: quand la qualité logicielle devient vitale. au sens strict du terme," *Regional-IT*, 2014, last access: 07/11/2016. [Online]. Available: http://www.regional-it.be/2014/05/07/iba-qspin-qualite-logiciel/

[8] X. Devroey and G. Perrouin, "Variability Intensive system Behavioural teSting framework (VIBeS)," Namur, Belgium, 2016. [Online]. Available: https://projects.info.unamur.be/vibes/

[9] "QTaste," https://github.com/qspin/qtaste, version 2.3.0.

[10] P. Baker, Z. Ru Dai, J. Grabowski, Ø. Haugen, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[11] J. Vanhecke, "Concrétisation de tests abstraits avec AbsCon, un AddOn QTaste," Master thesis, PReCISE, University of Namur, Belgium, 2016. [Online]. Available: https://github.com/modji-be/AbsCon

[12] J. Guerrero-Garcia, J. M. Gonzalez-Calleros, J. Vanderdonckt, and J. Munoz-Arteaga, "A theoretical survey of user interface description languages: Preliminary results," in *Web Congress, 2009. LA-WEB'09. Latin American*. IEEE, 2009, pp. 36–43.

[13] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero, "Usixml: a language supporting multi-path development of user interfaces," in *International Workshop on Design, Specification, and Verification of Interactive Systems*. Springer, 2004, pp. 200–220.

[14] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[15] "Inspect," https://msdn.microsoft.com/en-us/library/windows/desktop/dd318521(v=vs.85).aspx.

[16] "SwingInspector," http://www.swinginspector.com/index_en.htm.

[17] "Sahi: The tester's web automation tool," http://sahipro.com.

[18] "AutoHotKey," http://ahkscript.org.

[19] "Sikuli Script," http://www.sikuli.org.

[20] "Squish GUI Tester," https://www.froglogic.com/squish/.

[21] L. C. Briand and Y. Labiche, "A UML-Based Approach to System Testing," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. Springer-Verlag, 2001, pp. 194–208.

[22] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Formal Methods and Testing," R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Model-based Testing of Object-oriented Reactive Systems with Spec Explorer, pp. 39–76.

[23] W. Dulz, "MaTeLo - statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3," *Third International Conference on Quality Software, 2003. Proceedings.*, pp. 336–342, 2003.

[24] "Smartesting," http://www.smartesting.com.

[25] N. Li and J. Offutt, "A test automation language framework for behavioral models," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, no. 1. IEEE, apr 2015, pp. 1–10.

[26] N. Li, A. Escalona, and T. Kamal, "Skyfire: Model-Based Testing with Cucumber," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2016, pp. 393–400.

[27] "FitNesse: The fully integrated standalone wiki and acceptance testing framework," http://www.fitnesse.org.

[28] R. Mugridge and W. Cunningham, *Fit for developing software: framework for integrated tests*. Pearson Education, 2005.

[29] D. J. Mosley and B. A. Posey, *Just enough software test automation*. Prentice Hall Professional, 2002.