An Industrial Study on Applications of Combinatorial Testing in Modern Web Development

Murat Ozcan
Siemens Building Technologies CPS Software Hub
Chicago, USA
murat.ozcan@siemens.com

***Abstract*—The purpose of this study is to describe the new paradigms in front-end web application testing and how combinatorial testing (CT) fits in this modern development environment to create cost effective, highly fault-detecting automated tests. The system under test (SUT) is a cloud based application for monitoring and controlling building operations from Siemens Building Technologies, currently under development.**

Three examples for the applications of combinatorial testing in the front-end are analyzed, incorporating the CT model into automation using Protractor test framework's end-to-end (e2e) UI tests in behavioral driven development (BDD) style. The manner in which the CT model translates automation code is studied. A scenario where a sequence of actions are incorporated into a CT model is illustrated; with a focus on verification of these sequences, compositions of the actions and streamlining the expected assertions per the test oracle.

***Index Terms*—Cloud, combinatorial testing, sequence, test automation, Docker, Angular, Protractor**

I. Introduction

In this paper we will showcase how CT fits in the front-end test automation in a continuous integration (CI) environment. Examples will include modeling of the input parameter model (IPM) and how the model translates to test code. One example will include a scenario where a sequence of actions will be incorporated into a CT model, a problem that has been addressed in a variety of ways in previous works.

Due to space constraints, this study will focus on black box test automation of the front-end. The CI environment will be described, however CT applications in the CI pipeline and testing at different levels of the architecture are not the focus of the paper, they are planned for a future effort.

II. Related Work

Existing studies have focused on integrating covering arrays and sequence testing [1], [2], [3]. In comparison, the method of modeling sequenced parameter groups applied in this paper is to solve a simpler problem: to incorporate testing three test actions in their varying sequences, without increasing the number of tests achieved using a covering array which does not factor-in sequences of these three test actions. Constraints are the driving factor behind the logic. Implementation borrows ideas from a combination of the modeling patterns found in the previous work on sequences as well as other common patterns in CT [4], which include:

- *Optional and conditionally-excluded values:* the use of N/A value for parameters that are not a part of the parameter group in the current sequence.
- *Multi-selection & Order and padding:* a variation of these ideas was used for sequence control parameters.
- *Equivalence partitioning & Boundary value analysis:* used in input parameter modeling.

III. The System Under Test

A. Description of the Architecture

The SUT under development is a cloud application utilizing microservices, that enables the users to connect to their buildings though a browser interface, monitor and operate the buildings remotely. The front-end of the SUT is an Angular framework in TypeScript. The back-end is an ExpressJS application on top of the NodeJS platform. Further backend components serve the purpose of exposing Siemens or third-party edge-devices to the cloud.

B. Description of the CI Environment

The team uses GitLab for CI. When any code is submitted, the application is built in a docker container, unit testing and linting are done, the container containing the application is deployed and run in a cloud-hosted container and finally automated UI tests are executed targeting this deployment. If testing passes, the code can get merged to the master branch. This automated testing process ensures that after any code commit, the application is fully regression tested and quality is ensured.

C. Description of the Test Code

1) Test data: The test data is stored in JavaScript Object Notation (JSON) files. This allows parameterization of test inputs, input-driven tests, as well as ease of maintenance as the test configurations or the UI changes. It was found that another perk of using JSON is being able to convert test suites generated with CT tools from csv format to JSON format.

2) Test Specifications: The team uses Protractor test framework, which is the default e2e test framework for Angular applications. The test specifications are stored in Protractor spec files written in TypeScript.

IV. Test Methodology

In the following sections test designs using CT techniques will be described. For all CT suite generation CTWedge [5] was used and for coverage measuring CAMetrics [6] was used. In all IPMs, a partial script is shown due to space constraints. They can be provided upon request. The CT test design and test automation workflow in all subsections is as such:

1) CT modeling of the IPM as in CTWedge script snippet.
2) Generation of test suite in csv, conversion to JSON.
3) Based on the test suite, writing the automated test code.

A. Input driven testing of hardware

The IPM for the hardware test configuration is agnostic to device types and building control points. This is achieved through parameters and values in JSON format. The following code snippet shows how the input parameters and their values are represented in JSON. For example, a binary-output point which may control a digital light or a fan may be referred to as *FanCmd* (Fan Command) or *LDO* (Logical Digital Output). Through parameterization this complexity is abstracted. The test code only uses the parameters and the values of the parameters are easily manipulated depending on UI changes and or test configurations.

```
"SubSystemType": {
 "SystemOne": "DXR-VAV",
 "DesigoClassic": "DesigoClassic-ASG03",
 "ApogeeBACnetFLN": "PTEC-Heat_Pump"
},
"PointType_SystemOne": {
 "AnalogValue": "ECO CLG STPT",
 "BinaryOutput": "FAN 1 SPD 2" ....
}
```

In the CTWedge script snippet, it can be observed that the parameters *subsystemType* and *pointType* map directly from JSON data. On the other hand, parameters such as *commandType* and *commandToPointValue* represent abstracted test actions. All possible test actions are defined in the parameters and later, depending on the type of point, are restricted through constraints. Verbal description of constraints are included as comments in the CT model, easing readability.

```
Model Commanding
Parameters:
 subsytemType : {ApogeeBACnet, ApogeeBACnetFLN, 3
     rdPartyBACnet, Modbus, DesigoClassic,
     SystemOne, SystemOneAX100}
 pointType:  {BO, MO, AO, BV, MV, AV, BI, AI,
     HoldingReg1, HoldingReg2}
 commandType: {slider, buttons_plusMinus, textBox,
     buttonsOnOff, buttons3stages, NA}
 commandToPointValue: { readOnly, cancel, anyValue,
      higherByOne, lowerByOne, lowBoundary,
     highBoundary, lowBoundaryBeyond,
     highBoundaryBeyond}
 Constraints:
 // if pointType is BO or BV, you can only command
     it +1, -1, cancel, command type is fixed to
     OnOff and it's not readOnly
```

subsytemType	pointType	commandType	commandToPointValue
DesigoClassic	AV	slider	highBoundary
DesigoClassic	AV	buttons_plusMinus	higherByOne

Fig. 1. Partial input driven commanding test suite

```
 # (pointType = BO) || (pointType = BV) => ((
     commandToPointValue = higherByOne) || (
     commandToPointValue = lowerByOne) || (
     commandToPointValue = cancel)) && (commandType
      = buttonsOnOff) && (commandToPointValue !=
     readOnly) #
 // if pointType is MO or MV, you can only command
     it +1 (and another), -1 (and another), cancel,
      command type is fixed to buttons3stages and
     it's not readOnly
 # (pointType = MO) || (pointType = MV) => ((
     commandToPointValue = higherByOne) || (
     commandToPointValue = lowerByOne) || (
     commandToPointValue = cancel)) && (commandType
      = buttons3stages) && (commandToPointValue !=
     readOnly) #
```

Binary points and Multi-state points have two or more states, followed by confirm or cancel commands. Certain points such as inputs may be *readOnly* and do not have a *commandType*. Such points have a *N/A* value for *commandType*, and the parameter is *negated* through its use; a CT technique described in [4].

Compared to binary and multistate points, analog points are more complex and have a variety of command options. They can be commanded via slider, text box, incremented or decremented. They are prime candidates for boundary value analysis and equivalence partitioning in CT [4].

It is our opinion that the test suite in Figure 1 and test code snippet will solidify the understanding of the application of constraints for analog points. For ease of readability and confidentiality, test actions and assertions are handled under the functions in the code snippet.

```
describe('Testing commanding', () => {
 it('analog point, slider high bound.', () => {
  pointCommand.slider.highBoundary(); });
 it('analog point, (+) button', () => {
  pointCommand.button.incrementValue(); });
 });
```

B. Filtering Points by Tag Combinations

In the building technologies domain, a point is any object used to control the building; i.e. a light switch can be a digital point, a thermostat can be an analog point. In the SUT, tag filtering functionality is used to filter points in the system, by the devices the points are in. There are 13 possible tags, up to 5 tags can be chosen, tag choices cannot repeat, the final tag has only one tag choice, and there can be less than 5 tags for certain tag combinations if so is the nature of the points in the system. The following CTWedge script samples the IPM of the functionality. The constraint logic is that no tag can repeat, with 4 constraints for each tag.

tag1	tag2	tag3	tag4
air	cmd	sensor	temp
cmd	cooling	heating	writable

Fig. 2. Partial Covering array for tag filtering

```
Model tags
Parameters:
 tag1 : {cmd, writable, sensor, temp, cooling..}
 tag2 : {cmd, writable, sensor, temp, cooling..}
 tag3 : {cmd, writable, sensor, temp, cooling..}
 tag4 : {cmd, writable, sensor, temp, cooling..}
Constraints:
// can't duplicate tags!
# tag1=cmd => tag2!=cmd && tag3!=cmd ..
# tag2=cmd => tag1!=cmd && tag3!=cmd ..
# tag3=cmd => tag2!=cmd && tag1!=cmd ..
# tag4=cmd => tag2!=cmd && tag3!=cmd ..#
```

With 4 parameters -final tag being default-selected due to a single tag choice being left in the tag dropdown menu- and 13 values each, the amount of tests generated could be vast. However there were three factors that reduced the number of tests to the covering array of Figure 2:

1) 13 x 4 lines of constraint logic.
2) the final tag always being a single choice in the tag-dropdown menu. For example, *humidity* tag has 1 more tag choice available in the dropdown menu, but since it is a single choice it does not affect the number of tests.
3) certain points having only so many tags. For example, any point with *fan* or *humidity* having a single tag (+1 tag), and certain combinations of tags - *cmd* & *fan* or *cmd* & *lighting* having at most 3 tags (+1 tag).

With the test suite of Figure 2 generated, only one automation method had to be written and repeated for each test:

- the method takes a single argument: a JSON parameter with a predefined string of delimited tags. For example *tagTest1 = "air,cmd,sensor,temp"*; there are 14 similar varieties of tag tests. These are infinitely flexible, simply a different string can be passed in for a different set of tags.
- in the method, the string of tags get split by the delimiter. Using these tags, the tag selection is done in an asynchronous for-loop: *for await(tag of tags)* {...}.
- as the tags are selected, the panel & point combinations get filtered. The tag details of the points are verified to match the tags chosen in the filter.

C. Sequenced Filter, Search, Sort of Geo-located Sites

The SUT is designed so that the operator can control a number of geo-located sites. The operator has the ability to filter and sort the sites. The following functionality is available:

- Sites can be filtered by their status: *Working Well, In Alarm, Disconnected*: 3 Boolean states.
- Sites can be filtered by any string entered in the text box.
- Sites can be sorted by *Status, Name, Street, Country, Creation Time*
- The above actions can happen in any order and should give the same resulting list of filtered and sorted sites.

The following CTWedge script shows the IPMs for the models of alarm filter and sequence tests. Alarm filters are modeled as 3 Boolean states and the resulting covering array includes 4 configurations. This result is plugged in to the next model, a technique from [7] to reduce the number of tests and still provide cost effective coverage. The parameter *filter-workingWell-inAlarm-disconnected* includes the 4 configurations of these checkboxes as values.

```
Model alarmFilter
 Parameters:
 workingWell : Boolean
 InAlarm:  Boolean
 Disconnected : Boolean
// the result {101, 011, 110, 000} is plugged in
    to the next model
Model Sequences
Parameters:
 filter_WW_IA_DC: {101, 011, 110, 000}
 filterBeforeSearch: Boolean
 search : {searchString, siteStreet, siteCity,
     siteState, siteCountry }
 searchBeforeSort: Boolean
 sortBy:  {status, sName, sStreet, sCountry,
     createdAt}
 filterBeforeSort: Boolean
Constraints: // constraints to reduce address
    redundancies
# filterBeforeSearch=TRUE && searchBeforeSort=TRUE
     => filterBeforeSort=TRUE #
# filterBeforeSearch=FALSE && searchBeforeSort=
    FALSE => filterBeforeSort=FALSE #
```

The *search* parameter is a text-box, which is open ended. For this, possible values were contemplated to be any string as well as an assortment of address fields. The *sortBy* parameter simply replicates the names of the columns. In order to keep the number of tests low, instead of being a parameter in the model, reverse sorting of columns gets covered in each automated test. This is also utilized as a setup & cleanup in tests since sort order of the previous column gets retained. In order to have a deterministic test oracle, an array of sites was generated with varying data parameters - all stored in JSON - so that after filtering by site status and a search string, there would be at least 2 sites to test sorting with.

Finally, the sequence of the user actions needed to be addressed. The two filtering operations and the sorting can happen in any order, and there are 6 possible ways of ordering them. If the sequences of the 3 actions *filter, search, sort* were represented as 3 parameters *filterBeforeSearch, searchBefore-Sort, filterBeforeSearch*, they could be simplified as *AbeforeB, BbeforeC, AbeforeC*. Figure 3 displays this logic in a table. As observed, case 1 and case 6 are impossible; if *AbeforeB* and *BbeforeC* have the same Boolean value, *AbeforeC* has to match that value. This logic of sequence was addressed with 2 constaints:

- *if filterBeforeSearch is TRUE && searchBeforeSort is TRUE then filterBeforeSort is FALSE*
- *if filterBeforeSearch is FALSE && searchBeforeSort is FALSE then filterBeforeSort is TRUE*

ABC example	filterBeforeSearch	searchBeforeSort	filterBeforeSort	
	AbeforeB	BbeforeC	AbeforeC	sequence ABC
case 0	0	0	0	cba
case 1	0	0	1	
case 2	0	1	0	bca
case 3	0	1	1	bac
case 4	1	0	0	cab
case 5	1	0	1	acb
case 6	1	1	0	
case 7	1	1	1	abc

Fig. 3. Sequences of test actions

CAMetrics was used to measure pairwise coverage gain per test. The constraints eliminated 5 redundant test cases and the end result became a test suite of 19 tests. With this, a 20.8% cost saving was achieved while developing automation test code. Introducing the Boolean parameters for sequencing and the constraints did not impact the number of test cases; the final number was still 19.

At a small scale the CT technique used was effective; for 3 parameters, the 6 sequences were possible to be incorporated into the IPM without increasing the number of tests. Also the cost of creating the constraint logic was low since only two constraints had to be written. However, it can be observed that with a higher number of parameters this approach can get less cost effective. The team has not encountered such a problem yet and is looking forward to the challenge in a future study.

To conclude this section, the code snippet displays automated Protractor test code for one of the 19 tests. The test actions are in the sequence: *search* (by text string), *set site status* (dropdown) and *sort* (by country). The 3 functions execute these actions. After so, status pills are verified which verify the site status (In-Alarm, Disconnected, Normal). Following that, the sorting and reverse sorting of the sites are verified. Finally, the test oracle is verified to check the sites that are displayed as a result of the filters.

```
it('should setFilter_110, searchBy_siteCity,
    sortBy_siteStreet', () => {
  filterAndSort.setFilter(1, 1, 0);
  filterAndSort.textfilter(siteCity);
  filterAndSort.sortSitesBy('Street');
  // alarm pill assertions
  expect(statusWorkingWell).toBeTruthy();
  expect(statusInAlarm).toBeTruthy();
  // sorting assertions
  filterAndSort.verifyOrder(expectedOrder);
  // alarm state assertions
  expect(siteInfo.isPresent()).toBeTruthy();
  // reset sorting and clear filters
  filterAndSort.sortSitesBy('Street');
  filterAndSort.xFilters(2);
});
```

V. Results, Conclusion and future work

The tests are executed in CI pipeline environment with every code commit; if the test suite passes, the commit gets checked in and if there is a single failure it does not. Throughout the effort, defects preventing such check-ins have not been tracked rather fixed on the development branch causing the failure. On an average, 27% of the commits run a successful CI pipeline and get merged to the master development branch. At the time of writing, 157 defects have been found outside of CI pipeline executions and 32 have been found while implementing the automated tests. There are under 5 thousand lines of automation in Protractor (using TypeScript), more than 300 tests, executing under 600 seconds. The CI pipeline runs many times a day, at the time of writing there has been 6987 CI pipeline executions in the project. The tests were written in a 6 month period, by a single developer in test. This time included CT modeling, interpreting the model in Protractor code, and all related development and CI activities.

Interpreting the CT model into code is a costly process.To minimize cost in coding time, 2-way strength was utilized and the results achieved in defect prevention were satisfactory for the team.

In this paper, a variety of CT designs driving e2e UI testing in the front-end of a cloud application have been detailed. CT was utilized to drive test designs ensuring not only requirement but also high fault-coverage for the functionality under test. CT assured that the test design process led to writing meaningful and cost-effective automated tests that are more likely to find defects, bolstering confidence in the build quality of the SUT. All development and test code were unified in the CI pipeline, where every code change results in a full exection of the automated test suite. Having automated, near-instant feedback with full regression, hundreds of tests running in minutes with each modular code change accellerated development daily.

Consequent of the space restrictions, this study focused on the front-end of the SUT. In the future we plan to study the applications of CT in lower test layers and the CI environment.

References

[1] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, "Combinatorial methods for event sequence testing," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 601–609.

[2] E. Farchi, I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Combinatorial testing with order requirements," in *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 118–127.

[3] M. Brain, E. Erdem, K. Inoue, J. Oetsch, J. Pührer, H. Tompits, and C. Yilmaz, "Event-sequence testing using answer-set programming," *International Journal on Advances in Software Volume 5, Number 3 & 4, 2012*, 2012.

[4] I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Common patterns in combinatorial models," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 624–629.

[5] A. Gargantini and M. Radavelli, "Migrating combinatorial interaction test modeling and generation to the web," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 308–317.

[6] M. Leithner, K. Kleine, and D. E. Simos, "Cametrics: A tool for advanced combinatorial analysis and measurement of test sets," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 318–327.

[7] M. Ozcan, "Applications of practical combinatorial testing methods at siemens industry inc., building technologies division," in *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on*. IEEE, 2017, pp. 208–215.