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

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 testing, in lieu of the transition of applications from desktop, to web, to cloud, and how combinatorial testing (CT) fits in this modern development environment. We study Siemens Building Technologies Horizon Cloud Program's Building Operator (BO) application. Horizon Cloud provides widespread connectivity to legacy & new controls, 3rd party, IoT, edge devices and a step-wise covering of on-site, real-time requirements. BO is a cloud based application for monitoring and controlling building operations via a proprietary internet gateway.**

First, the architecture of BO is described along with the technologies being used in each layer. This is followed by a depiction of the continuous delivery process; Dockerizing [1] the continuous integration pipeline, the architectural components being run in containers, being deployed and tested in the cloud. The architecture of the test system is detailed, how test specifications relate to the test code structure and the test data is outlined.

Next, three examples for the applications of combinatorial testing in this modern development environment 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. We go through the CT modeling process using newly released, open source, cloud based CTWedge: Combinatorial Testing Web-based Editor and Generator [2]. The manner in which the model translates into functions in the code and their utilization in behavioral driven tests are studied.

Finally, 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

Technology stack choices for modern web development are diverse. Typically, in the front-end there is a JavaScript (JS) framework, some popular ones being Angular, React and Vue. The back-end preferences are varied, there is a plethora of choices in language: JavaScript (NodeJS), GoLang, Python, C# and Java to name a few. This is mostly coupled by at least one database, examples include MySQL, MongoDB, Elasticsearch.

Previous to cloud technologies, such an application would be hosted on a dedicated web server, on occasion by a hosting company. In contrast, cloud computing adopts a concept called "virtualization," where hardware resources can be further optimized through software functionality. As a result, not only application performance is optimized but also hosting the application is more cost effective and scalable on demand. Currently there are also serverless architecture solutions, a design pattern where applications are hosted by a third-party service, eliminating the need for server software and hardware management.

A challenge for cloud computing is the resource intensive operating system (OS) usage, where the size of the OS image can be in gigabytes while the application is much smaller. Consequently, virtual machines, since they have to host an OS, do not solve this problem. Containerization is one proposed solution, and Docker is one example of a program that performs operating-system-level virtualization. In containerization, a layer between the OS and applications is introduced to optimize resource usage and eliminate the need for an OS [1]. This is highly valuable for application development because it enables the application to be hosted in a minimal, resource and cost effective "container" which allows the application to be built, deployed and tested faster.

In this paper, we will study how CT fits in with the front-end test automation and the continuous deployment of cloud computing paradigm. Examples will include modeling of the input parameter model (IPM), how the model translates to test specifications, and the utilization of these in behavioral driven tests. 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.

Note that due to space constraints, this study will focus on black box test automation of the front-end. Testing at different levels of the architecture is planned to be studied in a future paper.

II. Related Work

A. Introduction to Combinatorial Testing

Combinatorial testing (CT) is a test design technique that provides cost effective test suite generation while guaranteeing a certain input space coverage. CT methodology starts by modeling the system under test (SUT) in an *input parameter model* (IPM) [3]. IPM is composed of *parameters of input* and their respective *values*, such that an input to this model can be represented by parameter-value assignments.

The underlying mathematical primitives of CT are covering arrays (CAs), which are discrete structures appearing in *combinatorial design theory*, and can be represented as matrices with specific coverage properties [4]. To further apply CT, the parameters of the IPM are matched with the columns of an appropriate CA, such that a row of the CA can be interpreted as an assignment of values to the parameters of the IPM - refer to Figure 8 for an example. Translating all rows of a CA in such a way, the mathematical properties of CAs guarantee that the generated test set is a *t-way test set*, i.e. a test set which ensures that all t-way combinations of parameter-value assignments are tested, once all tests have been executed [4].

A study from the National Institute of Standards and Technology (NIST) [4] shows that in all tested software products all faults found rely on the interaction of at most six input parameters. This means that all faults found in the tested software products can be triggered using a 6-way interaction test set, which is generally much smaller than an exhaustive test set, but yet achieving the same testing quality.

B. Previous works on sequences in CT

Existing studies have focused on integrating covering arrays and sequence testing [5], [6], [7], [8], [9]. 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 [10], 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

Horizon Cloud program is composed of many teams and applications. The system under test in this study will be Building Operator (BO). BO is a cloud application that enables the users to connect to their buildings in a secure manner and allows them to monitor & operate their building equipment remotely. BO is being developed by the Chicago team at Siemens Software Hub.

Figure 1 represents the BO architecture. The front-end is an Angular framework in TypeScript. Released in 2011, Angular is a popular front-end JavaScript framework for designing dynamic web applications [11]. It is maintained by Google with the help of the open source community. It recommends the use of Microsoft's TypeScript language, which introduces class-based object-oriented programming, static typing and generics [12].

The back-end is an ExpressJS application on top of the NodeJS platform. At the time of this study, there is an effort to move some of this functionality to other microservices - implemented in GoLang (Go) - in order to reduce operating costs.

The Protocol Adapter component (in C#) and Gateway (in NodeJS and Java) serve the purpose of exposing Siemens or third-party edge-devices to the cloud. This enables the hardware and the gateway at a customer site to be controlled from a web browser, anywhere around the globe.

B. Description of the Continuous Deployment Environment

In a traditional configuration management scenario, there may be a plethora of servers needed for web development activities. Staging, committing, packaging the developed code, testing at each stage, source controlling and finally, publishing to production server(s) may take place on a number of machines. This can end up in high costs of operation and maintenance in order to enable development activities.

In cloud computing, utilizing containerization, each one of these development activities can be operated in a Docker container. Figure 2 shows the GitLab pipeline for the UI component developed in Angular. GitLab is a continuous integration tool that supports the teams by hosting code repositories, providing defect tracking, enabling code reviews and continuous integration support [13]. Another example of a popular continuous integration tool used in the industry is Jenkins, which can serve the same purpose through its extensions.

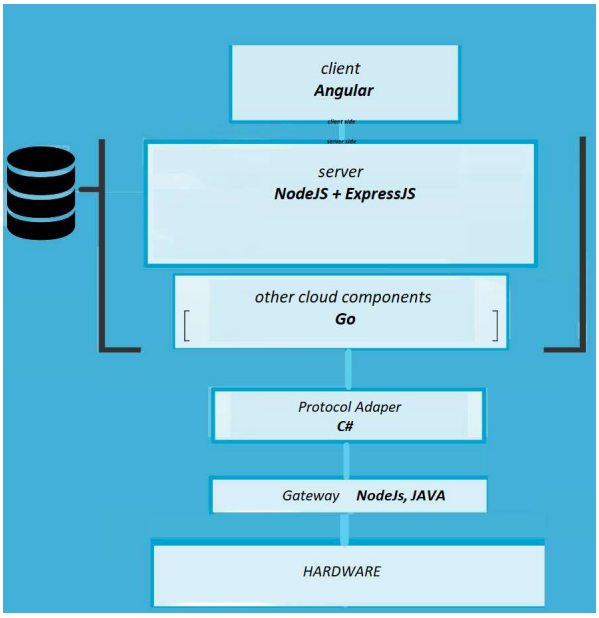

Fig. 1. BO architecture

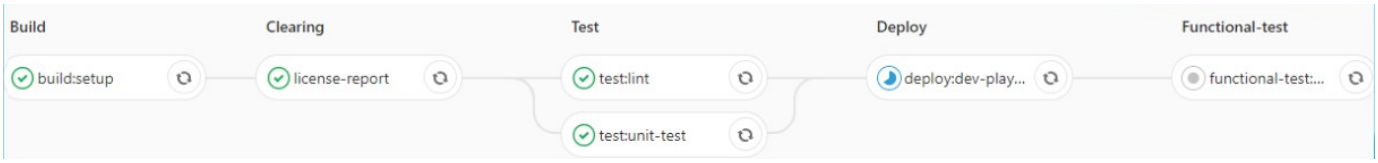

Fig. 2. UI component pipeline

The team uses Git for source control, a version-control system for tracking changes in files and coordinating work on those files among multiple people [14]. When any code is submitted, the application is built in a docker container, OSS Clearing for licensing is executed, 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.

To give a sense of scale, at the time of writing, the front-end is over 15k lines of TypeScript code in Angular, over 1k lines of unit tests in Karma, over 4k lines of e2e UI tests in Protractor. There are over 200 unit tests that execute in seconds, over 300 e2e UI tests that execute under 10 minutes.

C. Description of the Test Code Structure

1) Page Object Pattern: Components are the building blocks of Angular applications and they easily nest inside each other. Each component may have its own class, HTML and CSS file. This structure provides a way to design dynamic web applications while keeping the front-end code clean [11].

Page object pattern is a popular and cost-effective UI test automation design pattern in the software industry. It increases the maintainability of the UI tests by abstracting the application's web pages in order to reduce the coupling between test cases and application under test [15]. In the spirit of Fowler's proposed terminology [16], we refer to web pages as 'pages' and Angular components as 'panels'. Each panel is abstracted as a class. A page can be made up of many panels, which have a class-based object-oriented structure.

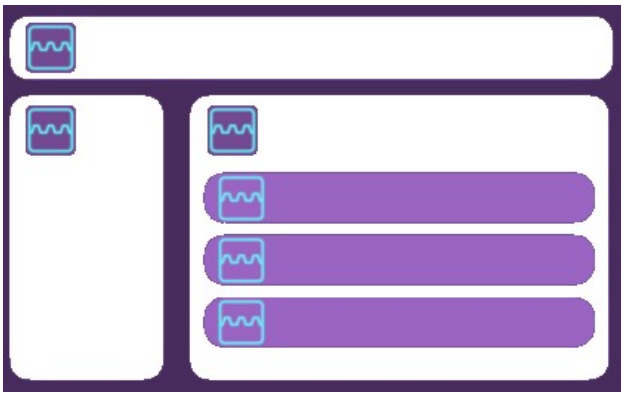

Fig. 3. Angular components as page and panel objects

Designing the test automation architecture, it became clear that it would not only be straightforward but also efficient to replicate the structure of Angular components in page objects as pictured in Figure 3. One advantage of this is easier maintenance of the automation code as the development code changes. The other advantage is its benefits with asynchronous test execution.

In a panel object, the UI element selectors become the class fields and the test actions become class methods within the component / panel [17]. Enabling this class-based object-oriented structure is Typescript. TypeScript is a super-set of JavaScript. It compiles into JavaScript and includes the latest EcmaScript features [12]. One of the recent advantages in the latest EcmaScript features is how it helps developers write sequential looking asynchronous code using *Async Await* style *Promises* [18].

With regards to test automation this means clean code that executes as fast as the environment allows, resistant resistant to flakey tests and flake in tests [19] - tests that give inconsistent results in subsequent executions. This is because the page components load with panel-object-classes simultaneously, while the page element selectors get instantiated. We plan to study this design pattern in a future paper. For the purpose of the current study, it is sufficient to understand that all test relevant methods are housed in classes that represent panel objects / Angular components.

2) 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. More on parameterization in section IV-A.

It was found that another perk of using JSON is being able to convert test suites generated with CT tools ([2], [20]) from csv format to JSON format.

3) Test Specifications: The team uses Protractor test framework, which is the default e2e test framework for Angular applications [21]. It runs tests against the application running in a real browser, interacting with it as a user would. It combines technologies from Selenium webDriver and NodeJS and allows tests to be described in a behavioral driven development (BDD) format -*Given, When, Then*-, describing the overall behavior of a system at a low level.

The test specifications are stored in Protractor spec files written in TypeScript. The aforementioned benefits of the technology stack allow developers in test to write clean, simple, synchronous looking asynchronous code, without hard-waits or sleeps, resistant to stale elements and flakey tests.

```
beforeAll(async () =>{
  await tunnelPage.setDropdown();
});
describe(`Workflow: progress with tunnel connection until it is active\n
  Given: I have arrived at state Creating Tunnel Connection with start button enabled`, () => {
  it(`When I trigger an event:  Start button is clicked,
    Then: I expect the application to move to Creating Tunnel Connection state\n`, async () => {
    await tunnelPage.startButton.click();
    expect(tunnelPage.tunnelConnectionStateText.getText()).toBe('Creating your Tunnel connection...');
  });
});
```

Fig. 4. Protractor test code sample in State-based, Behavior-driven Acceptance format

Over time during the development, some of the newer test specification came to be described in State-based, Behavior-driven Acceptance Scenarios [22]. Generally, these are in the format:

Given I have arrived in some state
When I trigger a particular event
Then the application conducts an action
And the application moves to another state

It showed that this style of expression is not only descriptive at a meta level, but also maps well to the page object pattern. Figure 4 shows a Protractor test spec sample, designed by using a state-based, behavior-driven acceptance scenario from Figure 5 . As observed in the sample code from Figure 4, *tunnelPage* class / page-object is utilized for:

- setup, using the *setDropDown()* class method.
- test action, using the *startButton* class field.
- test assertion, verifying that *tunnelConnectionStateText* class field / selector shows the correct text.

The code also samples the Protractor BDD style where a test workflow scenario is covered. The *it* block is one test in the workflow, the *describe* block houses more tests / *it* blocks that lead to the completion of the workflow - not shown in this example for the sake of simplicity.

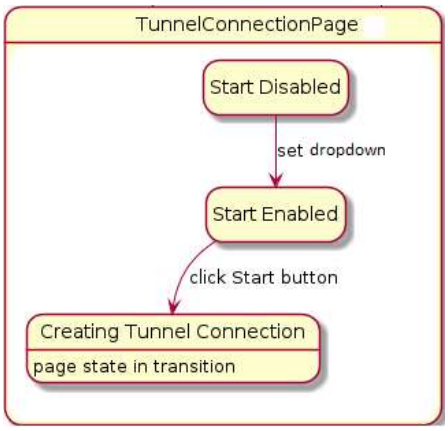

Fig. 5. State-based, Behavior-driven Acceptance Scenario

IV. Test Methodology

In the following sections test designs using combinatorial testing techniques will be described. Note that this study will focus on black box test automation of the front-end. Testing at different levels of the architecture is planned to be studied in a future paper.

For all combinatorial test suite generation CTWedge [2] was used and for coverage measuring CAMetrics [23] and Combinatorial Test Coverage Measurement tool (CCM) [24] were used. At the time of writing CAMetrics was under development, and CCM was used for its completeness coverage measurements. Traditionally, ACTS [20] has been used in the organization during previous work. CTWedge and CAmetrics were preferred over their desktop contenders for a few reasons:

- *Portability and ease of use*: it is easier to copy a few lines of script and paste it at a url to generate a test suite in the case of CTWedge. It is also easier to upload a csv file to CAMetrics url and get a report quickly.
- *Ease of model modification*: it takes a few hours to master CTWedge scripting, but once done so, scripting proves to be a more robust interface to address changing test conditions; parameters, values or complex constraints. Comments in the script is an added benefit to readability.

A. Input driven testing of hardware

BO provides widespread connectivity to legacy & new controls, 3rd party, IoT and edge devices. There is a variety of devices being used in different markets; The Americas, EU and Asia. Consequently, the IPM for the test setup needs to be agnostic to device types and building control points. This is achieved through parameters and values in JSON format. Figure 6 describes the possible parameters for the hardware setup and 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 -depending on context- 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	writable points	readable points
SystemOne	AV, BO, BV, MV	AI
DesigoClassic	AO, BO, MO, AV, BV, MV	BI
Modbus		HoldingRegister1, HoldingRegister2
ApogeeBACnetFLN	AO, BO	AI, BI,
ApogeeBACnet	BO	
SystemOneAX100	AV, BO, MV	BV
3rdPartyBACnet	AO	

Fig. 6. Input Parameter Model for hardware setup

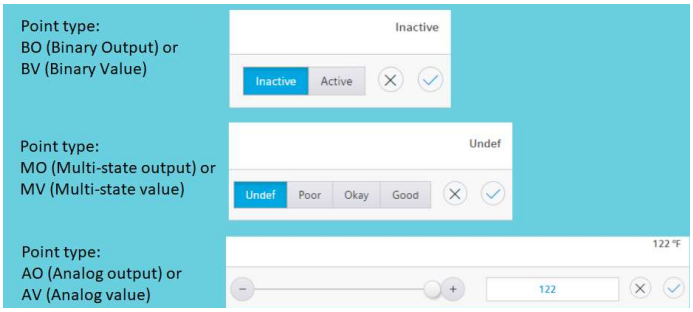

Fig. 7. Commanding UI for various point types

```
"SubSystemType": {
 "SystemOne": "DXR-VAV",
 "DesigoClassic": "DesigoClassic-ASG03",
 "Modbus": "Meter-DEM-1",
 "ApogeeBACnetFLN": "PTEC-Heat_Pump",
 "ApogeeBACnet": "PXC24-1",
 "SystemOneAX100": "RDY-Thermostat",
 "ThirdPartyBACnet": "Viconics"
},
"PointType_SystemOne": {
 "AnalogValue": "ECO CLG STPT",
 "BinaryOutput": "FAN 1 SPD 2",
 "BinaryValue": "CMF BTN",
 "MultiStateValue": "RM OP MODE",
},
"PointType_DesigoClassic": {
 "AnalogOutput": "AS03'PTP'AnaObj'AO001",
 "BinaryOutput": "AS03'Mis'CycTi'LED",
 "MultiStateOutput": "AS03'PTP'MulObj'MO001",
 "AnalogValue": "AS03'Ala'Per'AnaObj'AV001",
 "BinaryValue": "AS03'Ala'AsSta.DsavSta",
 "MultiStateValue": "AS03'Ala'Per'MulObj'MV001",
 "BinaryInput": "AS03'PTP'BinObj'BI001",
```

Figure 7 samples the UI for commanding various point types and the following script details the CT model in CTWedge. 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. Note that due to space restrictions, the model is shown partially and the complete version can be provided upon request.

```
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
# (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) #
// additional constraints
```

A constraint for binary-output or binary-value is scripted in CTWedge and would verbally translate as such: *if pointType is binary-output or binary-value, you can only command it +1, -1, cancel, command type is fixed to OnOff and it's not readOnly*.

A constraint for multistate-output and multistate-value is scripted in CTWedge and would verbally translate as such: *if pointType is multistate-output or multistate-value, you can only command it +1 (and another), -1 (and another), cancel, command type is fixed to buttons3stages and it's not 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 [10].

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 [10].

The CT test design and test automation workflow is as such:

1) CT modeling of the IPM as in CTWedge script snippet.
2) Generation of test suite / covering array in csv as in Figure 8, conversion into JSON as needed.

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

Fig. 8. Input driven commanding test suite - partial

3) Based on the test suite, writing the automated test code, sampled below.

```
describe('Testing Write property', () => {
 it('should command analog point with slider to
     high boundary', () => {
  pointCommand.slider.highBoundary(); });
 it('should command analog point with (+) button',
     () => {
  pointCommand.button.incrementValue(); });
});
```

It is our opinion that the test suite in Figure 8 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. For example, functions *command...ButtonMinusOne()* and *...PlusOne()* test incrementing and decrementing the point with (+) and (-) buttons. Functions *command...SliderToFarRight()* and *...ToFarLeft()* do boundary value testing using the slider, and can be applied to any kind of analog point. Function *command...WithTextBox()* can be used to pass any integer to the text box, helping with equivalence partitions and boundaries.

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. Figure 9 displays a screen capture of tag filtering UI and the following CTWedge script samples IPM of the functionality. Note that due to space restrictions only 2 blocks of 13 blocks of constraints are shown.

```
Model tags
Parameters:
 tag1 : {cmd, writable, sensor, temp, cooling,
     lighting, sp, effective, occ, unocc, fan, air
     , heating}
```

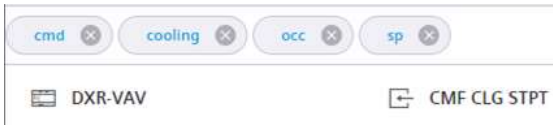

Fig. 9. Tag filtering UI

```
 tag2 : {cmd, writable, sensor, temp, cooling,
     lighting, sp, effective, occ, unocc, fan, air
     , heating}
 tag3 : {cmd, writable, sensor, temp, cooling,
     lighting, sp, effective, occ, unocc, fan, air
     , heating}
 tag4 : {cmd, writable, sensor, temp, cooling,
     lighting, sp, effective, occ, unocc, fan, air
     , heating}


Constraints:
// can't duplicate tags!
# tag1=cmd => tag2!=cmd && tag3!=cmd && tag4!=cmd#
# tag2=cmd => tag1!=cmd && tag3!=cmd && tag4!=cmd#
# tag3=cmd => tag2!=cmd && tag1!=cmd && tag4!=cmd#
# tag4=cmd => tag2!=cmd && tag3!=cmd && tag1!=cmd#
# tag1=writable => tag2!=writable && tag3!=
    writable && tag4!=writable #
# tag2=writable => tag1!=writable && tag3!=
    writable && tag4!=writable #
# tag3=writable => tag2!=writable && tag1!=
    writable && tag4!=writable #
# tag4=writable => tag2!=writable && tag3!=
    writable && tag1!=writable #
\\ additional blocks for each tag
```

Since the 5th tag is always a single choice, there is no need to represent it in the model and it is automated to be selected in the tests. The constraint logic is that no tag can repeat, with 4 constraints for each tag. For example, one block of constraints is: *if tag1 is 'cooling' then tag2 through tag 4 cannot be 'cooling', if tag2 is 'cooling' then tag 1, 3, 4 cannot be 'cooling'* and so on. CTWedge scripting provided cost savings while implementing the constraint logic. Being able to copy the 4 lines of constraint logic, to replace the tag name, paste and repeat for each tag while the web application applies IntelliSense for the syntax, saved considerable amount of time in implementation of constraints.

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 10:

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 10 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 Filtering, Searching, Sorting of Geo-located Sites

BO application is designed so that the building operator can control a number of geo-located sites. Each site has a unique address. The building operator has the ability to filter and sort the sites he/she has administration over. 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.

Figure 11 shows the UI for alarm filtering menu as well as the search text box and sorting buttons for site list. The following CTWedge script shows the IPMs for the models of alarm filter and sequence tests - partial script shown due to space constraints. 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 [25] 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 is {101, 011, 110, 000} and it 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:
# filterBeforeSearch=TRUE && searchBeforeSort=TRUE
     => filterBeforeSort=TRUE #
# filterBeforeSearch=FALSE && searchBeforeSort=
    FALSE => filterBeforeSort=FALSE #
// constraints to reduce address redundancies
```

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; not utilizing this behavior would cause tests to be non-modular.

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.

tag1	tag2	tag3	tag4
air	cmd	sensor	temp
cmd	cooling	heating	writable
cmd	cooling	occ	sp
cmd	heating	occ	sp
cmd	heating	sp	unocc
cmd	fan	writable	
cmd	lighting	writable	
cmd	occ	unocc	writable
cmd	effective	sp	writable
cmd	effective	temp	writable
cmd	cooling	sp	unocc
cmd	effective	sp	writable
fan			
humidity			

Fig. 10. Covering array for tag filtering

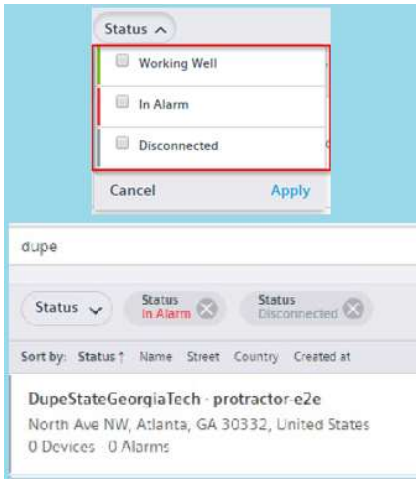

Fig. 11. UI for site filtering

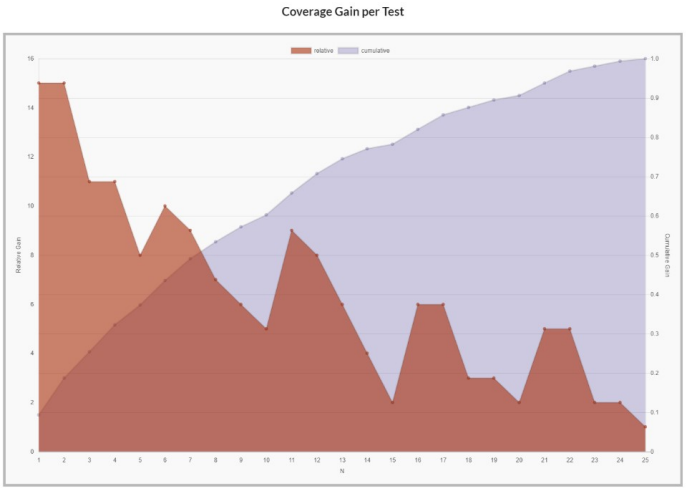

Fig. 12. No constraints : relative gain (red) vs cumulative gain (blue) per test

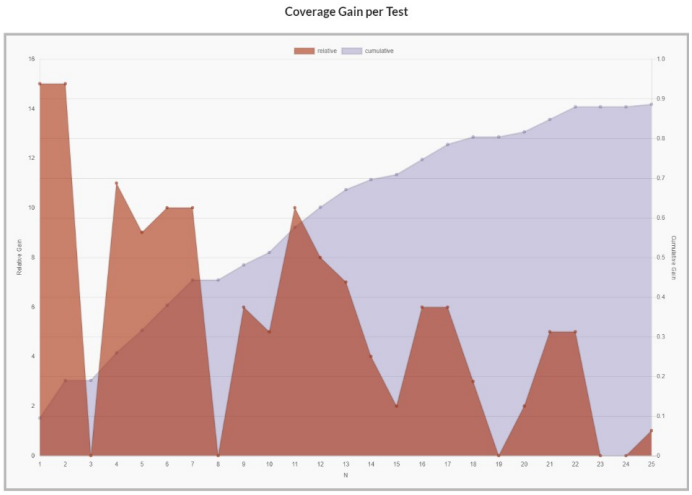

Fig. 13. Constraints applied to Figure 12

Generating the tests, it was promptly realized that if sorting sites by column - possible values being *Status, Name, Street, Country, Creation Time* - matches the search string -possible values being fields of an address - sorting became redundant and had no effect. These redundancies were addressed by adding constraints, for instance *if sorting sites by Street, search string cannot be street name*. CAMetrics was used to measure pairwise coverage gain per test: Figure 12 and Figure 13 show coverage before and after constraints being applied. 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.

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. 14. sequences of test actions

Finally, the sequence of the user actions needed to be addressed. Previous research on this topic has been vast and is referred to in section II-B . 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, searchBeforeSort, filterBeforeSearch*, they could be simplified as *AbeforeB, BbeforeC, AbeforeC*. Figure 14 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*

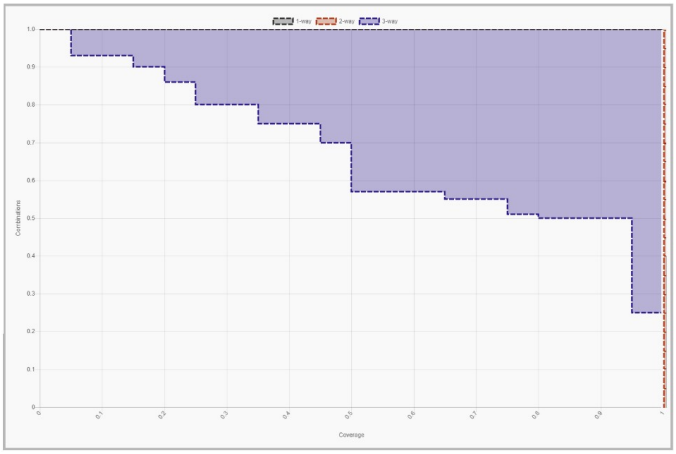

Fig. 15. 2-way and 3-way completeness coverage, measured with CCM

Introducing the Boolean parameters for sequencing and the constraints did not impact the number of test cases; the final number was still 19. Figure 15 shows 2-way and 3-way completeness coverage using CCM. For the 2-way covering array, completeness coverage of 2-way combinations was found to be as such: 100% of 2-way tuples covered to 100%. To elaborate, of the 158 2-way tuples, all are included in the covering array for this test suite to have full pairwise / 2-way coverage. For reference, 3-way completeness coverage is also shown: 25% of 3-way combinations are covered to 100%, 50% of 3-way combinations were covered to 95%. To elaborate, of the 629 3-way tuples, 25% of them provide 3-way coverage at 100%, 50% of them provide 95% 3-way coverage and so on.

The significance and the impact of minimum t-way coverage on branch coverage conditions in the code is discussed in the paper [26]. Label 'M' as the minimum 2-way coverage; i.e., the lowest proportion of settings covered for all t-way combinations of variables. For example, 2-way combinations of binary variables have four possible settings: 00, 01, 10, 11. Some variable pairs may have all four settings covered, but others may have less. M is the smallest proportion of coverage among all of the t-way variable combinations. M is also viewable as the rightmost line in the coverage strength meters.

At a small scale this technique 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.

```
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);
});
```

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.

V. Results

The tests are executed in CI pipeline environment with every code commit; if the automation 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, 20% of the commits run a full 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 was satisfactory for the team.

VI. Conclusion and future work

In this paper, Siemens Building Technologies Horizon Cloud Program's Building Operator application was studied as the SUT. The system architecture, continuous deployment and test system architecture were described in the paradigm of cloud computing. In this modern development environment, we explored a variety of combinatorial test design techniques driving e2e UI testing of the front-end.

Traditionally in software development, requirement specifications are defined and testing -automated or manual- follows these requirements. CT is a test design methodology that goes beyond the requirement specifications of the application and by defining an IPM it covers most possibilities of user interactions with the application, depending on model quality. While testing interactions of at most six input parameters are needed to find nearly all faults in a given functionality of a software product, *2-way* testing still ensures cost-effective coverage of most possible faults [4]. In this study CT was used 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.

After the test design, state-based, behavior-driven acceptance scenarios were used to create descriptive e2e test scenarios with a workflow. This method produced acceptance and workflow coverage by bridging verbose requirements to test code. The significance of this was the simultaneous verification and validation of the SUT build and SUT definition, respectively, strong coupling between the requirement and the test code, boosting certainty that the right system is being built and being tested accurately.

These test scenarios utilized page and panel objects, abstracting the relationship between the tests and the applicaton while reducing the coupling between them. Using latest features in ECMAscript, the overall test designs were interpreted to sequential looking asynchronous test code that executes as fast as the environment allows, resistant to flakey tests and stale page elements. In day to day automated test creation, the majority of the team's effort is invested in unriddling asynchronous logic and producing deterministic tests. Adopting cutting edge tech eased these challenges, resulting in daily performance increase and const savings.

Ending this process, all development and test code were unified in the continious integration pipeline in the cloud, utilizing containerization, 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 UI of the SUT. In the future we plan to study Building Operator's architectural components beyond the UI, design patterns in automation being applied at test layers and possibilities of CT being applied at different test levels.

REFERENCES

[1] Solomon Hykes, "Wikipedia, the free encyclopedia," 2013. [Online]. Available: https://en.wikipedia.org/wiki/Docker_(software)

[2] 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.

[3] M. Grindal and J. Offutt, "Input parameter modeling for combination strategies," in *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering*, ser. SE'07. Anaheim, CA, USA: ACTA Press, 2007, pp. 255–260.

[4] D. Kuhn, R. Kacker, and Y. Lei, "Practical combinatorial testing," NIST Special Publication 800-142, 2010.

[5] 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.

[6] 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.

[7] 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.

[8] C. P. Yang, G. Dhadyalla, J. Marco, and P. Jennings, "The effect of time-between-events for sequence interaction testing of a real-time system," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 332–340.

[9] E. Erdem, K. Inoue, J. Oetsch, J. Pührer, H. Tompits, and C. Yılmaz, "Answer-set programming as a new approach to event-sequence testing," 2011.

[10] 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.

[11] Google, "Wikipedia, the free encyclopedia," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Angular_(application_platform))

[12] Microsoft, "Wikipedia, the free encyclopedia," 2012. [Online]. Available: https://en.wikipedia.org/wiki/TypeScript

[13] GitLab, "Wikipedia, the free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/wiki/GitLab

[14] Linus Torvalds, "Wikipedia, the free encyclopedia," 2005. [Online]. Available: https://en.wikipedia.org/wiki/Git

[15] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Improving test suites maintainability with the page object pattern: An industrial case study," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 108–113.

[16] Martin Fowler, "Pageobject," 2013. [Online]. Available: https://martinfowler.com/bliki/PageObject.html

[17] Carmen Popoviciu, "Protractor style guide," 2014. [Online]. Available: https://github.com/CarmenPopoviciu/protractor-styleguide

[18] Ecma International, "Wikipedia, the free encyclopedia," 2015. [Online]. Available: https://en.wikipedia.org/wiki/ECMAScript

[19] Martin Fowler, "Eradicating non-determinism in tests," 2011. [Online]. Available: https://martinfowler.com/articles/nonDeterminism.html

[20] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "ACTS: A combinatorial test generation tool," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 370–375.

[21] Protractor, "Protractor an end to end testing tool for angular," 2014. [Online]. Available: http://www.protractortest.org/#/

[22] Arie van Deursen, "Beyond page objects: Testing web applications with state objects," 2015. [Online]. Available: https://queue.acm.org/detail.cfm?id=2793039

[23] 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.

[24] D. R. Kuhn, I. D. Mendoza, R. N. Kacker, and Y. Lei, "Combinatorial coverage measurement concepts and applications," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 352–361.

[25] 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.

[26] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Measuring and specifying combinatorial coverage of test input configurations," *Innovations in systems and software engineering*, vol. 12, no. 4, pp. 249–261, 2016.