Crazy title

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

***Abstract*—The overall purpose of this study is to describe the new paradigms in testing, in lieu of the transition of applications from the desktop, to web and to cloud, and how combinatorial testing (CT) fits in this modern development environment. We study the Siemens Building Technologies Horizon Cloud Program's TODO: insert long version (BOIC) application. TODO: insert short sentence about BOIC In Horizon Cloud, products are enabled by the Horizon ecosystem with a focus on speed, independence and small complexity. 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. Horizon Cloud features its own apps based on building integration framework, along with themes, with an open API for ecosystem and SSP. The program also provides a common data model and API to Building Technologies Building Integration Framework to map southbound data against model.**

We start by describing the architecture of BOIC the components that make up the whole, how they relate, technologies being used in each layer and how they are being tested individually and as a whole.

We continue by describing the continuous delivery process; Dockerization of the continuous integration pipeline, the architectural components being built as Docker images and being run as Docker containers, being deployed and tested in the cloud at different test levels.

We describe the abstraction and the structure of the test code, the page object model, how test specifications relate to the test code architecture and the test data.

Finally, we analyze three applications of combinatorial testing (CT) in this modern development environment, incorporating the CT model into automation using BDD style Protractor end to end UI tests. We go through the CT modeling process using newly released, open source, cloud based CTWedge: Combinatorial Testing Web-based Editor and Generator. We describe how the model translates into functions in the code and its utilization in behavioral driven tests.

Finally, we analyze a scenario where a sequence of actions are incorporated into a CT model; with a focus on verification of these sequences, compositions of the actions and streamlining the expected assertions per the data-manipulated test oracle.

***Index Terms*—Cloud, Combinatorial testing, Sequence, Angular, Protractor, test automation**

TODO: Add comments from Rcik Rick : A number of things I think would be of particular interest. One would be your comments on how CT fits into container/Docker development and some of the other platforms that are used a lot today. Most researchers don't know a lot about modern development environments, so your experience and recommendations on how to use CT for these would be very valuable, e.g., what sort of extra tooling would be helpful, characteristics that are good or poor fits for existing CT tools.

Also of interest is the way you integrate covering arrays and sequence testing. This is something that isn't always handled well, and yet very important for an awful lot of applications. The measurements of coverage at the end of the presentation are also of interest, in particular how to use coverage to determine when it's cost effective to complete testing.

I. Introduction

Modern web development typically falls into a design pattern. In the front-end there can be 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 : JS (NodeJS), GoLang, Python, C#, Java to name a few. This is coupled by a database, examples include MySQL, MongoDB, Elasticsearch.

Previous to cloud technologies, such an application would be hosted on a dedicated web server, or 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.

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. TODO: insert Docker reference In containerization, a layer between OS and applications is introduced to optimize resource usage and eliminate the need for an OS.

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 Combinatorial Testing (CT) fits in the front-end test automation and continuous deployment of cloud computing paradigm. Examples will include modeling of the input parameter model (IPM), how the model translates to fields and methods in page objects TODO: insert reference, and the utilization of these in behavioral driven test specifications. 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.

II. Related Work

A. Introduction to Combinatorial Testing

TODO: work on paraphrasing this. Combinatorial testing (CT) is a highly sophisticated testing methodology, that is capable of producing comparable small test sets (e.g. when compared to exhaustive testing), while at the same time providing guarantees of (certain) input space coverage. To apply CT to a system under test (SUT), it is necessary to have an *input parameter model* (IPM) of the SUT [1]. To devise an IPM for an SUT it is necessary to identify *input parameters* and their respective *values*, such that an input to such a model can be represented by parameter-value assignments. Engineering an IPM can be a tedious and time intensive task itself [2]. 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 [3]. 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. 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 [3]. Note that the general problem of constructing a t-way test set is believed to be NP-hard as it is tightly coupled with hard combinatorial optimization problems (see [4]). A study from the National Institute of Standards and Technology (NIST) [3] shows that in all tested software products all faults rely on the interaction of at most six input parameters. This means that all faults 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

The method of modeling sequenced parameter groups applied in this paper borrows ideas from a combination of the modeling patterns found in [5]. These modeling patterns include:

1) *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.
2) *Ranges and boundaries:* the reduction of parameter values for certain parameters with over 100 possibilities
3) *Multi-selection & Order and padding:* a variation of these ideas was used for control parameters which enable/disable the parameter groups per the sequence.

C. Previous works on page objects

TODO: insert some papers about Page objects

D. Previous works on cloud computing

TODO: insert some papers about Cloud

III. The System Under Test

A. Description of the Architecture

Horizon Cloud program is composed of many teams and microservice architectural applications. The application / system under test in this study will be Building Operator IC (BOIC). BOIC is used for TODO: describe BOIC , also insert what IC means . BOIC is being development by the Chicago team at Siemens Software Hub.

Figure 1 represents the BOIC architecture. The front-end is an Angular framework, in TypeScript. 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.

The back-end is an ExpressJS application on top of 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) - 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 in the globe.

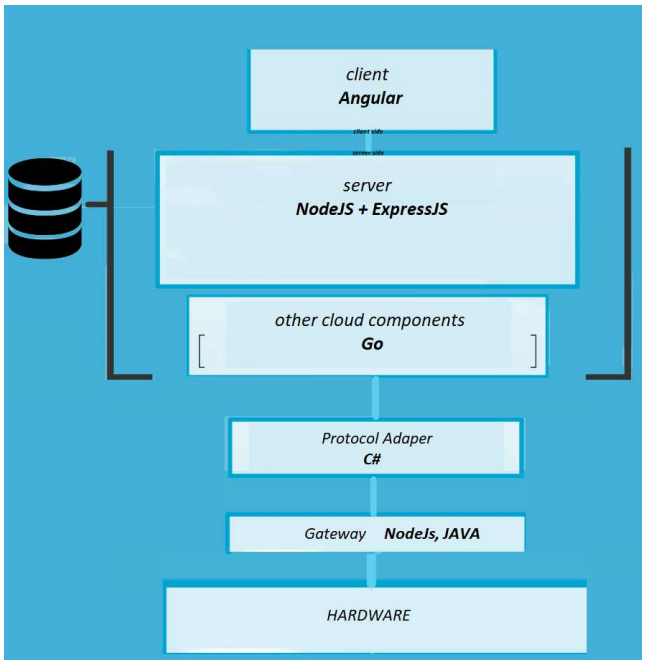

Fig. 1. BOIC architecture

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 at production server(s) may take place on a number of

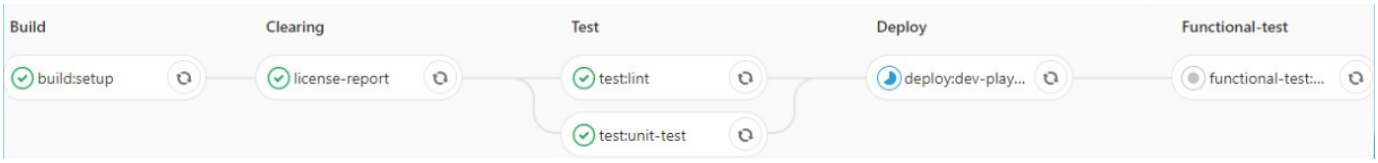

Fig. 2. UI component pipeline

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 (similar to GitHub), providing defect tracking, enabling code reviews and continuous integration support. Another example of a continuous integration tool is Jenkins, which can serve the same purpose thorough is extensions.

The team uses Git for source control. TODO: about git. 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 application is deployed to an cloud-hosted container and automated UI tests are run 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 tested end to end and the 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 end to end (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: Released in 2011, Angular is popular front-end JavaScript framework for designing dynamic web applications. 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. TODO: refer to wiki.

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.

Page object pattern is a popular UI test automation design pattern in the industry. TODO: refer to Fowler and selenium paper. In the spirit of Fowler's proposed terminology, we refer to web pages as 'pages' and Angular components as 'panels'. Each panel is a class. A page can be made up of many panels, which have a class-based object oriented structure.

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 element selectors become the class fields and the test actions become class methods within the component / panel. Enabling this class-based object oriented structure is TypeScript TODO: insert reference TypeScript is a superset of JavaScript. It complies into JavaScript and includes the latest EcmaScript features TODO: reference to EcmaScript . One of the recent advantages in the latest EcmaScript features is how it helps developers write sequential looking asynchronous code TODO: refer to async awaits.

With regards to test automation this means clean code that executes as fast as the environment allows, resistant to flakey tests and stale elements on the page. 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.

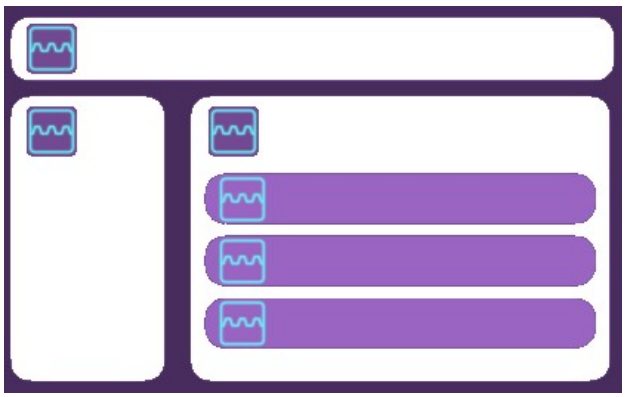

Fig. 3. Angular components as page and panel objects

2) Test data: The test data is stored in JSON files. This allows parameterization of test inputs, input-driven tests, as well as ease of maintenance as the UI changes. Another perk of using JSON is being able to convert test suites generated with CT tools - ex: ACTS, CTWedge TODO: references - from csv format to JSON format.

3) Test Specifications: The team uses Protractor test framework, which is the default end-to-end test framework for Angular applications. TODO: insert reference It runs tests against the application running in a real browser, interacting with

```
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

it as a user would. It combines technologies from Selenium webDriver, NodeJS and allows tests to be described in a 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.

Over time during our development, the test specification came to be described in State-based, Behavior-driven Acceptance Scenarios TODO: reference blog post. 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

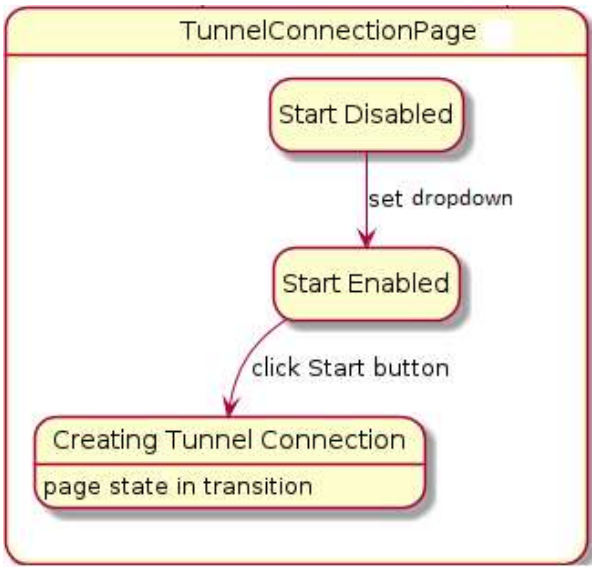

Fig. 5. State-based, Behavior-driven Acceptance Scenario

It was found out that this style of expression is not only descriptive at a meta level, but also maps well to the page object pattern. Figure 4 and Figure 5 show an a Protractor test spec sample with a state-based, behavior-driven acceptance scenario.

As can be 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.

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 was used and for coverage measuring CAMetrics and CCM were used TODO: reference. Traditionally, ACTS 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.
- *Cloud concept* : the entirety of the project is web-based and the web-based CT tools fit this concept.

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

A. Input driven testing of hardware

BOIC provides widespread connectivity to legacy & new controls, 3rd party, IoT and edge devices. The devices have no restriction to region; US, EU, Asia etc. Consequently, the input parameter model (IPM) for the test setup needs to be agnostic to device types and control-points. This is achieved through parameters and values in JSON. Figure 6 describes the possible parameters for the hardware setup.

The following code snippet shows how the input parameters and their values are represented in JSON. For example, a binary-output point (BO) 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 parameterazation 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",
  "Modbus": "Meter-DEM-1",
  "ApogeeBACnetFLN": "PTEC-Heat_Pump",
  "ApogeeBACnet": "PXC24-1",
  "SystemOneAX100": "RDY-Thermostat",
  "ThirdPartyBACnet": "Viconics"
},
"PointType_SystemOne": {
  "AV": "ECO CLG STPT",
  "BO": "FAN 1 SPD 2",
  "BV": "CMF BTN",
  "MV": "RM OP MODE",
},
"PointType_DesigoClassic": {
  "AO": "AS03'PTP'AnaObj'AO001",
  "BO": "AS03'Mis'CycTi'LED",
  "MO": "AS03'PTP'MulObj'MO001",
  "AV": "AS03'Ala'Per'AnaObj'AV001",
  "BV": "AS03'Ala'AsSta.DsavSta",
  "MV": "AS03'Ala'Per'MulObj'MV001",
  "BI": "AS03'PTP'BinObj'BI001",
```

Figure 8 details the CT model scripted 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 test actions are relaxed 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.

A constraint for Binary Outputs (BO) and Binary Values (BV) is scripted in CTWedge and would verbally translate as such: *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*.

A constraint for MultiState Outputs (MO) and MultiState Values (MV) is scripted in CTWedge and would verbally translate as such: *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*

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 inspired by TODO: reference Rachel's paper

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 equivalance partitioning in CT TODO: reference paper. Partial script example for constraints can be viewed in Figure 8. It is our opinion that the test code snippet in Figure 9 will solidify the understanding of the application of constraints for analog points.

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

subsytemType	pointType	commandType	commandToPointValue
ApogeeBACnet	BO	buttonsOnOff	cancel
ApogeeBACnet	BO	buttonsOnOff	higherByOne
ApogeeBACnet	BO	buttonsOnOff	lowerByOne
ApogeeBACnetFLN	AI	NA	readOnly
ApogeeBACnetFLN	AO	buttons_plusMinus	lowerByOne
ApogeeBACnetFLN	AO	slider	lowBoundary
ApogeeBACnetFLN	AO	textBox	anyValue
ApogeeBACnetFLN	AO	textBox	highBoundary
ApogeeBACnetFLN	AO	textBox	highBoundaryBeyond
ApogeeBACnetFLN	AO	textBox	higherByOne
ApogeeBACnetFLN	AO	textBox	lowBoundaryBeyond
ApogeeBACnetFLN	BI	NA	readOnly
ApogeeBACnetFLN	BO	buttonsOnOff	cancel
DesigoClassic	AO	textBox	highBoundaryBeyond
DesigoClassic	AO	textBox	lowBoundary
DesigoClassic	AV	buttons_plusMinus	higherByOne
DesigoClassic	AV	slider	anyValue
DesigoClassic	AV	slider	highBoundary
DesigoClassic	AV	textBox	lowBoundaryBeyond
DesigoClassic	AV	textBox	lowerByOne
DesigoClassic	BI	NA	readOnly
DesigoClassic	BO	buttonsOnOff	higherByOne
DesigoClassic	BV	buttonsOnOff	higherByOne
DesigoClassic	MO	buttons3stages	cancel
DesigoClassic	MO	buttons3stages	higherByOne
DesigoClassic	MO	buttons3stages	lowerByOne
DesigoClassic	MV	buttons3stages	lowerByOne

Fig. 7. Input driven commanding test suite - partial

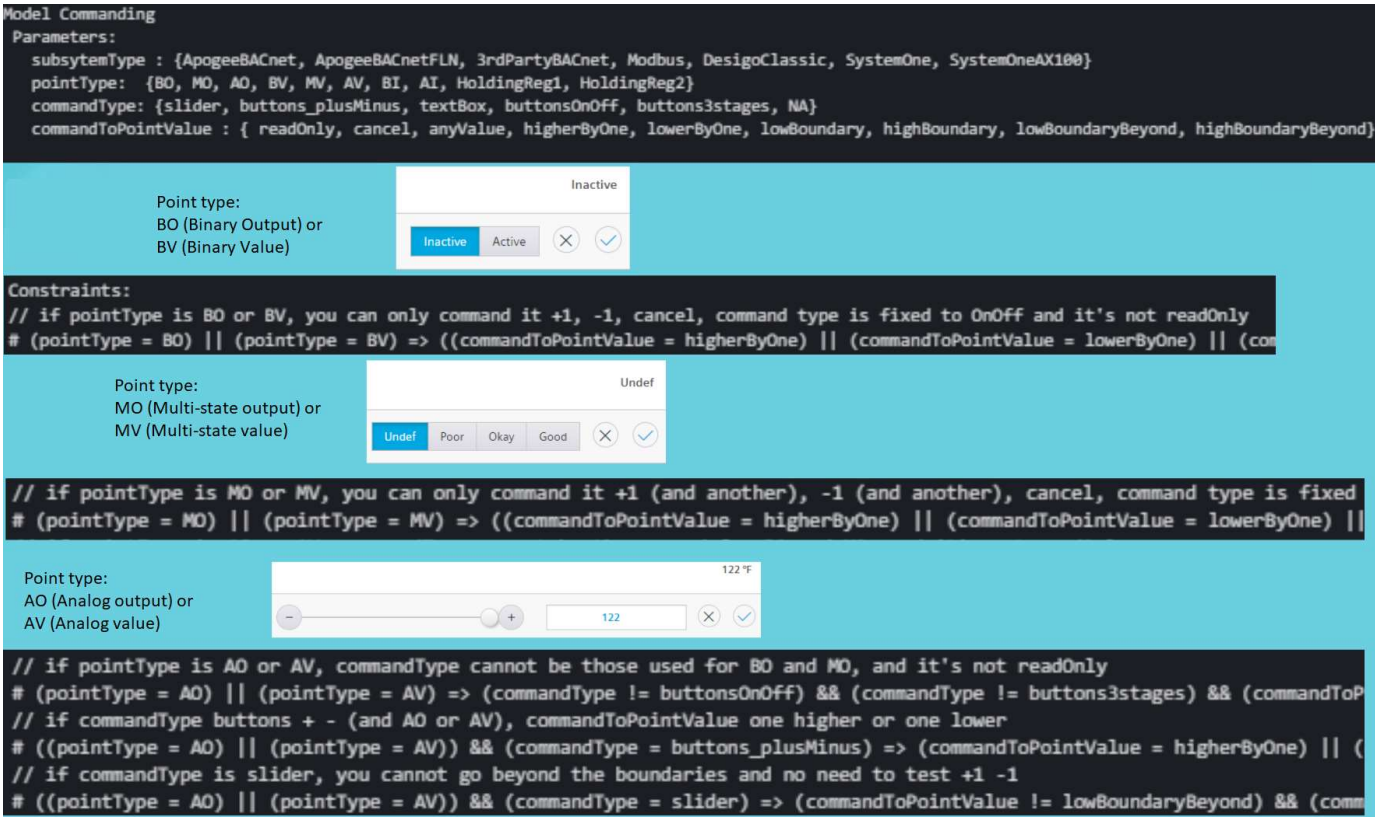

Fig. 8. CT model of panel and point commanding in CTWedge

- CT modeling of the IPM as in Figure 8.
- Generation of test suite in csv as in Figure 7, conversion into JSON as needed.
- Based on the test suite, writing the automated test code as in Figure 9.

```
describe('When: testing pointType BV', () => {
});

describe('When: testing pointType AV', () => {
});

describe('When: testing WRITE properties', () => {
  it('Then: command analog point with TEXT BOX to ANY value', () => {
    pointCommandService.commandTestAnalogPointWithTextBox(boundaryValueAnalysis.
  });
  it('Then: command analog point with (+) / (-) buttons, higher by one ', () =>
    pointCommandService.commandTestAnalogPointWithButtonsPlusOne();
  });
  it('Then: command analog point with (+) / (-) buttons, lower by one ', () => {
    pointCommandService.commandTestAnalogPointWithButtonsMinusOne();
  });
  it('Then: command analog point with SLIDER to the HIGH BOUNDARY', () => {
    pointCommandService.commandTestAnalogPointWithSliderToFarRight();
  });
  it('Then: command analog point with SLIDER to the LOW BOUNDARY', () => {
    pointCommandService.commandTestAnalogPointWithSliderToFarLeft();
  });
  it('Then: command analog point with TEXT BOX to BEYOND the LOW BOUNDARY', () =
    pointCommandService.commandTestAnalogPointWithTextBox(boundaryValueAnalysis.
  });
  it('Then: command analog point with TEXT BOX to BEYOND the HIGH BOUNDARY', ()
    pointCommandService.commandTestAnalogPointWithTextBox(boundaryValueAnalysis.
```

Fig. 9. Protractor *describe* block for Analog Value point commanding in Desigo Classic panel

Note that Figure 9 shows a part of the test spec in Protractor. For ease of readability and confidentiality, test actions and assertions are handled under the functions in the code snippet. For example the function *commandTestAnalogPointWithSliderToFarRight()* does boundary value testing using the slider, and can be applied to any kind of analog point. Similarly the function *commandTestAnalogPointWithTextBox()* continues boundary value analysis by passing in an integer to the text box, with a value greater than the boundary.

B. Filtering Points by Tag Combinations

In BOIC, 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 10 displays an input parameter model of the functionality with a screen capture. Note that due to space restrictions not all tags are displayed and only 2 blocks of 13 blocks of constraints are shown.

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 saved considerable amount of time.

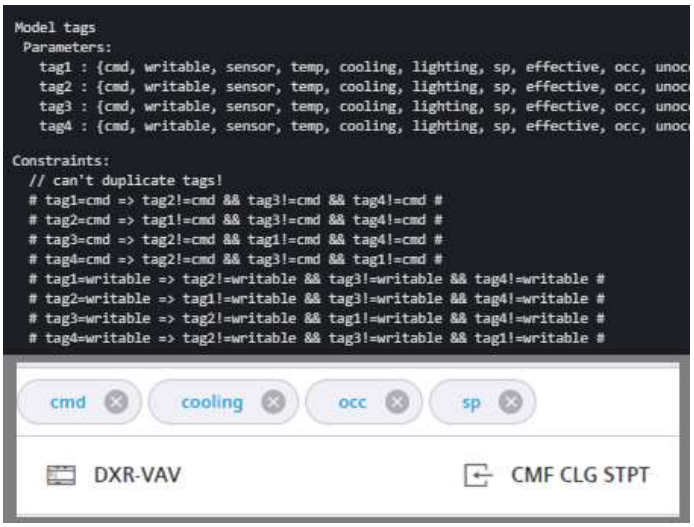

Fig. 10. CT model of tag filtering in CTWedge

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. 11. Covering array for tag filtering

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 11:

- 13 x 4 lines of constraint logic.
- 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 effect the number of tests.
- 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 11 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 delimeted tags. For example *tagTest1 = "air,cmd,sensor,temp"*. 14 varieties, one for each test. These are infinately 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 delimeter. Using these tags, the tag selection is done in an asynchronous for loop.
- 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

BOIC is designed so that the building operater 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 they have administration over:

- 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 17 shows the alarm filtering menu as well as the search text box and sorting buttons. 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 TODO: refer to your own paper to reduce the number of tests and still provide cost effective coverage. The parameter *filter-workingWell-inAlarm-disconnected* includes the the 4 configurations of these checkboxes as values.

The *search* parameter is a text-box, which is open ended. For this, possible values were contemplated to be any string, address fields.

The *sortBy* parameter simply replicated 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 got covered in each automated test. This was also utilized as

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. 12. sequences of test actions

a setup & cleanup in tests since sort order of the previous column gets retained; not utilizing this behaviour would cause tests to be non-modular.

In order to have a deterministic test oracle, an array of sites were 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.

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 TODO: reference to CAMetrics to measure pairwise coverage gain per test: Figure 14 and Figure 15 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 savings was achieved while developing automation test code.

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 TODO: insert section . The two filtering operations and the sorting can happen in any order, and there 6 possible ways of ordering them. If the sequence of the 3 actions *filter, search, sort* were represented as 3 parameters *filterBeforeSearch, searchBeforeSort, filterBeforeSearch*, they could be simplified as *AbeforeB, BbeforeC, AbeforeC*. Figure 12 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 && searchBeoreSort is TRUE then filterBeforeSort is FALSE
if filterBeforeSearch is FALSE && searchBeoreSort is FALSE then filterBeforeSort is TRUE

Introducing the boolean parameters for sequencing and the constraints did not impact the number of test cases; the final number was still 19. Figure 13 shows 2-way and 3-way completeness coverage using CCM TODO: reference to CCM. For the 2-way covering array, completeness coverage of 2-way combinations was found to be as such: 100% of 2 tuples

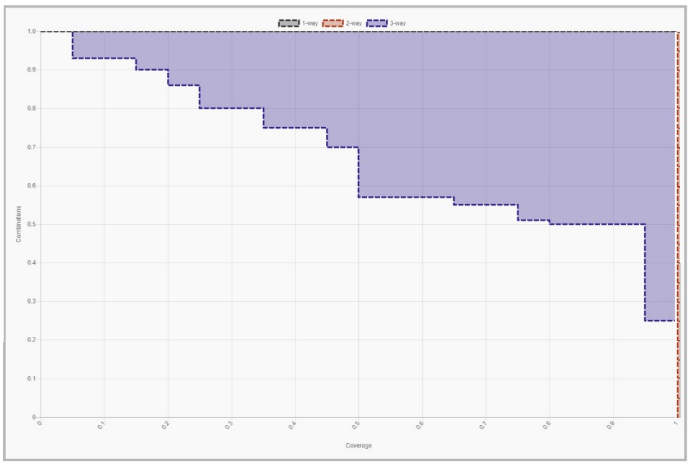

Fig. 13. 2-way and 3-way completeness coverage, measured with CCM

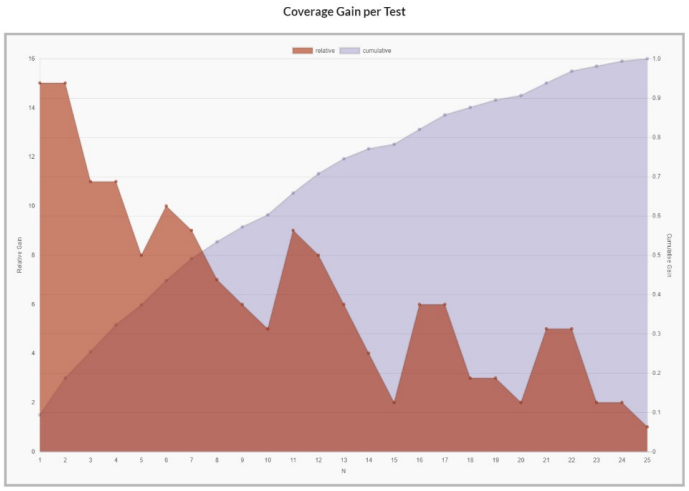

Fig. 14. No constraints : relative gain (red) vs cumulative gain (blue) per test

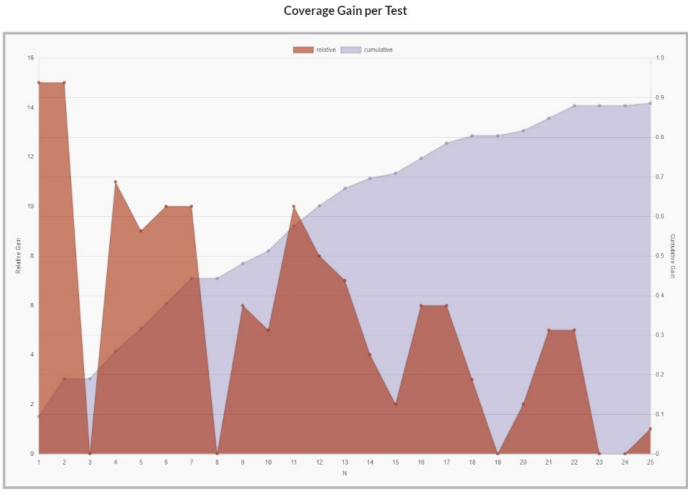

Fig. 15. Constraints applied to Figure 14

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 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 TODO: insert reference Measuring and Specifying Combinatorial Coverage of Test Input Configurations. 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 input parameter model 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('SearchFilterSort_filterBy02_searchBy_searchString_sortBy_sCountry_ ', (
  // test actions
  filterAndSortService.inputInSearch(filterByText.searchString); // 'dupe'
  filterAndSortService.openStatusDropdownAndSetFilter(0, 2); // 0th and 2nd
  filterAndSortService.sortSitesBy('Country');
  // alarm pill assertions
  expect(statusWorkingWell).toBeTruthy(); // expect status pill
  expect(statusDisconnected).toBeTruthy(); // expect status pill
  // sorting assertions
  filterAndSortService.verifyFirstSite(listOfSiteNamesInThisExpectedOrder);
  // test reverse sort
  filterAndSortService.sortSitesBy('Country');
  filterAndSortService.verifyLastSite(listOfSiteNamesInThisExpectedOrder);

  // alarm state assertions
  expect(DupeStreetStateCountrySchweizAG_DiSCO.isPresent()).toBeTruthy();
  expect(DupeStateGeorgiaTech_DiSCO.isPresent()).toBeTruthy();
  expect(DupeStreetCountrySWHouse_DiSCO.isPresent()).toBeTruthy();
  // reset sorting and filters
  filterAndSortService.sortSitesBy('Country');
  filterAndSortService.xToCrossOffFilters(2);
```

Fig. 16. One of the 19 automated Protractor tests in site filtering test suite

To conclude this sections, Figure 16 displays autormated 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. In turn, status pills are verified, verifying site status. Following that, the sorting and reverse sorting of the sites are verified. Finally, the test oracle is verified to verify that the sites that are displayed in the filter.

V. Conclusion and Future Work

The initial challenge in this study was how to best test EEO configurations in the most efficient manner. During the study it became apparent that the methods applied can be used to test any system where the parameter groups for the Combinatorial Input Parameter Model (IPM) are not simultaneously available, and instead may appear sequentially. Future work on this front may incorporate system test ideas where each seed (in the SBA method) or different color (in the IBM method) being a different module of functionality.

Acknowledgments. This research was carried out in the context of Austrian COMET K1 program and publicly funded by the Austrian Research Promotion Agency (FFG) and the Vienna Business Agency (WAW).

References

[1] 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.

[2] R. Bartholomew and R. Collins, "Using combinatorial testing to reduce software rework," *CrossTalk*, vol. 23, pp. 23–26, 2014.

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

[4] C. T. Cheng, "The test suite generation problem: Optimal instances and their implications," *Discrete Applied Mathematics*, vol. 155, no. 15, pp. 1943 – 1957, 2007.

[5] 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.

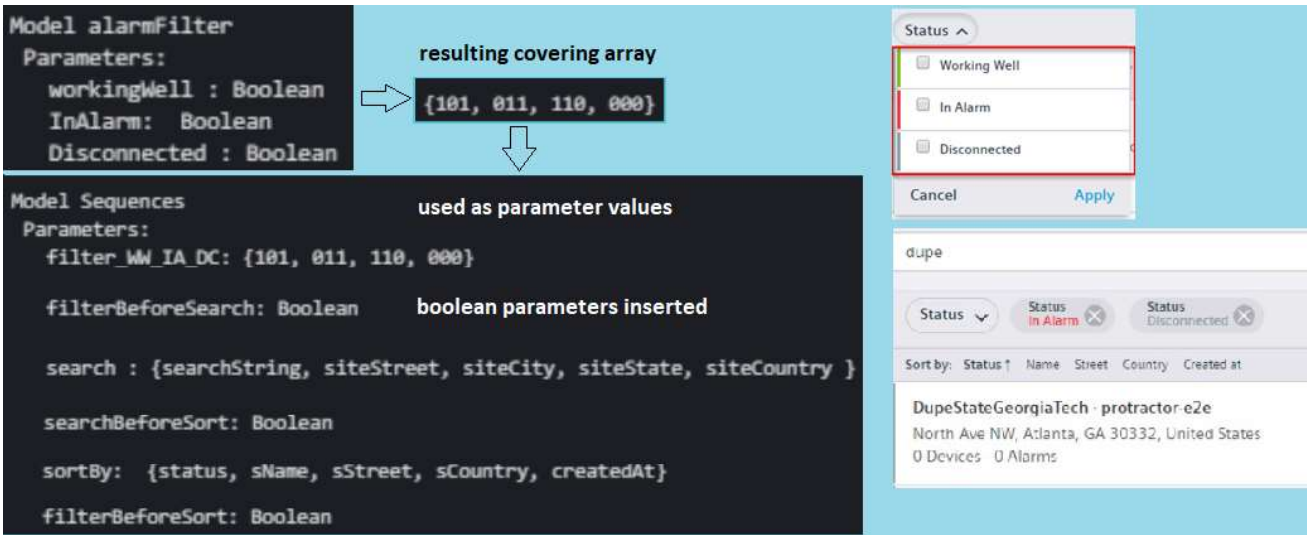

Fig. 17. CT model of site filtering in CTWedge & screen capture