Automated Testing of Software-as-a-Service Configurations using a Variability Language

Sachin Patel
TCS Innovaton Labs.
Tata Consultancy Services
Pune, India
sachin.patel@tcs.com

Vipul Shah
TCS Innovaton Labs.
Tata Consultancy Services
Pune, India
v.shah@tcs.com

ABSTRACT

The benefits offered by cloud technologies have compelled enterprises to adopt the Software-as-a-Service (SaaS) model for their enterprise software needs. A SaaS has to be configured or customized to suit the specific requirements of every enterprise that subscribes to it. IT service providers have to deal with the problem of testing many such configurations created for different enterprises. The software gets upgraded periodically and the configurations need to be tested on an ongoing basis to ensure business continuity. In order to run the testing organization efficiently, it is imperative that the test cycle is automated. Developing automated test scripts for a large number of configurations is a non-trivial task because differences across them may range from a few user interface changes to business process level changes. We propose an approach that combines the benefits of model driven engineering and variability modeling to address this issue. The approach comprises of the Enterprise Software Test Modeling Language to model the test cases. We use the Common Variability Language to model variability in the test cases and apply model transformations on a base model to generate a test model for each configuration. These models are used to generate automated test scripts for all the configurations. We describe the test modelling language and an experiment which shows that the approach can be used to automatically generate variations in automated test scripts.

Keywords

software-as-a-service; test automation; variability specification; model based testing; enterprise software testing

1. INTRODUCTION

The proliferation of cloud technologies has resulted in enterprise applications being offered through the Software-as-a-Service (SaaS) model. SaaS is a software delivery model in which software is centrally hosted and is accessed by users through a browser. Customers who subscribe to it are charged periodically or by amount of usage. This model is gaining popularity because it requires very little upfront investment in software licenses and hardware. One of the important characteristics of a SaaS application is that it should be highly configurable. This is necessary because every enterprise that subscribes to the SaaS would have its own business-specific requirements. These include variations in Graphical User Interface (GUI) structures, labels, navigational flows, master data, branding and business processes.

When enterprises subscribe to a SaaS, they choose an IT service provider (ITSP) to configure it according to their requirements. This results in the creation of as many configurations as the number of enterprises using the SaaS. ITSPs have to deal with the problem of testing many such configurations created for different enterprises. The objective of this testing is to ensure that the setup works for the enterprise specific master data and executes all the business transactions as expected. The software gets upgraded periodically and the configurations need to be tested on an ongoing basis to ensure business continuity. In order to run the testing organization efficiently, it is imperative that the test cycle is automated to the extent possible. Test execution constitutes a large portion of the overall effort and there exist well established techniques to automate it. However, these techniques are not designed to handle variability.

We propose an approach that combines model driven engineering and variability modeling to address this problem. We have developed a Enterprise Software Test Modeling Language (ESTML) to model test cases. The model can be automatically populated by a recorder and edited using an editor. The model is used to generate automated test scripts for web-applications. These scripts can be executed using the open source test automation tool - Selenium. The approach uses the Common Variability Language(CVL)[4] to create a Variability Specification over Test Model. Variability Resolution models are created corresponding to each configuration to be tested. The CVL tool resolves the specifications and generates test models corresponding to each configuration. These test models are then used for test script generation. We performed an experiment with an open source enterprise software application, orangeHRM. Our experiment shows that, ESTML and CVL can be used to model enterprise software test cases and variations.

The paper is structured as follows: Section 2 contains some background information about test automation and Common Variability Language(CVL). Section 3 describes the Enterprise Software Test Modeling Language, identifies variation points and explains how CVL will be used with it.

SPLC 2015, July 20 - 24, 2015, Nashville, TN, USA

DOI: http://dx.doi.org/10.1145/2791060.2791072

Section 4 describes the tool set created to implement the approach. based approach. Section 5 describes an experiment that demonstrates the functioning of the approach. This is followed by related work in Section 6 and Conclusions and Future Work in the last section.

2. BACKGROUND

This section provides an introduction to automated test execution and variability specification languages.

2.1 Test automation of web-based enterprise applications

Regression testing, is one of the last activities in the maintenance cycle. Testers tend to run fewer regression tests when there is shortage of time. Since the regression tests are test cases already identified and executed repeatedly, they are a good candidate for automation. Many commercial and open source tools are available for automation of test execution. Quick Test Pro, Rational Functional Tester and Selenium are examples of such tools. Testers use these tools to develop scripts that automatically perform user actions on the Graphical User Interface (GUI). One of the problems in test automation is that, systems continuously change; hence keeping the automation scripts up to date is a major overhead.

Application specific frameworks and a modular design can make the test scripts easy to maintain. However, this requires higher design time and skilled developers. Model-based generative approaches can help reduce the design and development time, as well as maintenance overhead [14]. A Model Driven Engineering (MDE) approach to generate test scripts from a Test model has been described in Section 4.

2.2 Common Variability Language

The Common Variability Language (CVL) is a generic language for modeling variability in models in any Domain Specific Language (DSL). While there are many variability modeling languages available, we choose CVL because it is a standard supported by the Object Management Group. The CVL approach consists of three models:

1. The base model : a model described in a DSL.
2. The variability model : the model that defines variability on the base model.
3. The resolution model : the model that defines how to resolve the variability model to create a new model in the base DSL.

The CVL language supports multi-stage model configuration by supporting the expression of variability into two conceptually distinctive layers:

- The feature-specification layer : this is the user-centric layer and expresses the variability of at feature level, just like in feature modeling. The concepts of CVL (type, composite variability, constraint and iterator) are sufficient to mimic feature diagrams (mandatory or optional feature, feature dependencies, XOR/OR and cardinality).
- The product-realization layer : the variability of product realization layer defines lower level, fairly fine-grained but necessary operations that are required to complete the transformation from the base model to the new resolved model. The CVL concepts value substitution, reference substitution, fragment substitution and boundary point binding are used to express the variability of the product realization layer.

When using CVL, the user first needs to create a Base Model which is defined using a DSL. The Vspecs (Variability Specifications) are specified on the base model to define variation points and choices available at the variation points. A resolution model will specify the choice at each variation point. CVL executes model transformations using these specifications and generates a DSL model corresponding to each resolution model.

3. VARIABILITY MODELING OF ENTERPRISE SOFTWARE TEST CASES

In this section we describe the modeling languages that we will use to specify enterprise software test cases and their variants.

3.1 The Enterprise Software Test Modeling Language

The Enterprise Software Test Modeling Language (ESTML) is an amalgamation of two Object Management Group (OMG) standards, namely, UML 2.0 Testing Profile (UTP)[12] and Knowledge Discovery Meta-model (KDM)[13].

The high level structure of ESTML resembles UTP. We have implemented concepts from the Test Architecture, Test Behaviour and Test Data sections. We have not implemented Time Concepts because we did not find them relevant for enterprise software test cases. The UTP does not provide means to specify details about the interface of the System under Test (SUT). As deemed necessary for our implementation we have added a Test UI section to model the GUI structure of the system under test. The granular details such as the interactions model in the behaviour section, the structure of the data pool in the test data section, and UI model in the test UI section are adapted from relevant packages of the KDM. The meta model for the 4 subsections of the test model may be seen in Fig. 1. The key concepts in each of the packages has been described below.

3.1.1 Test Architecture Package

Test Architecture section consists of elements that help organize and manage the test information. It comprises of the following elements:

- TestElement: This is an abstract element, which is the base class for TestSuite, TestCase and TestProject. A TestElement is associated with a TestLog that is generated after its execution. A TestElement has optional Setup and TearDown routines associated with it.
- TestProject: A TestProject represents the group of related test activities, typically performed on a particular software application or group of interrelated software applications. A TestProject is associated with one or more TestSuites and with a SystemUnderTest. TestProject is derived from TestElement.
- TestSuite: A TestSuite is a group of related TestCases. Typically TestCases for a particular component, screen

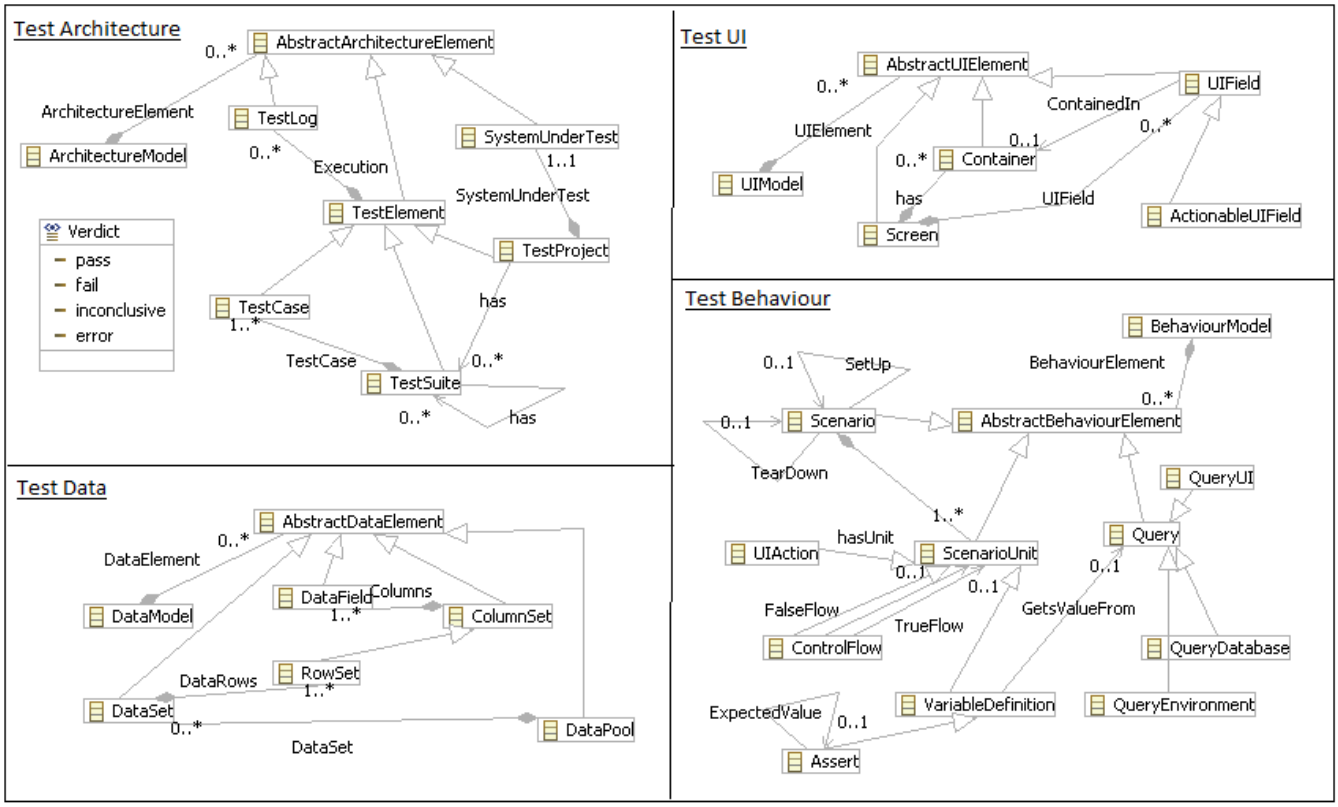

Figure 1: The Test Meta-Model

or transaction may be clubbed together as a TestSuite. Sometimes maintenance teams create TestSuites for particular versions of the software and such a TestSuite may contain TestCases which test the changes made in that version. A TestSuite is derived from TestElement, hence can have a Setup and TearDown as well as a TestLog generated after its execution.

- SystemUnderTest: This is the system being tested in the TestProject with which it is associated.
- TestCase: A TestCase is the central element in the test model. A TestCase is associated with a Scenario which it will execute and a DataPool that will supply the test inputs for the execution. The Scenario will contain the Asserts that serve as the TestOracle. The DataPool will contain the DataFields corresponding to the UIFields being used in the Scenario.
- TestLog: The TestLog contains the execution trace of the TestElement (which may be a TestCase or a TestSuite). It contains the verdict of execution. Verdict is an enumerated type that can take values *pass*, *fail*, *inconclusive* and *error*.

3.1.2 Test UI Package

The Test UI model contains the user interface specifications of the system under test. It comprises of the following elements:

- Screen: A screen is a form in the user interface which allows the user to perform a transaction in the application. A screen has a title (Screen.title), which is typically the name of the transaction. A screen has a one to many association with UIField (Screen->UIField).
- Container : A container is a group of UIFields that are related to each other and frequently occur together on the user interface. Containers have a label that describes the group of fields (UIField.label).
- UIField: UIField is a field on the user interface. Examples of UIFields are text box, checkbox, radio button, button, hyperlink, menu option and so on. Every UIField has a label that describes the information displayed or input in it. UIField has a type which indicates how the field has been implemented in the Screen (such as text box, check box and so on). An UIField is associated with the Container in which it is placed.
- ActionableUIField: An ActionableUIField is a kind of a UIField on which the user can perform an action (other than input data). Such fields typically initiate a transaction with the database. Examples of ActionableUIFields are buttons and hyperlinks.

3.1.3 Test Data Package

Test Data section contains datasets organized as tabular structures of data elements. It comprises of the following elements:

- DataField: A DataField represents the value that is input in a Screen while performing a UIAction. A

DataField has two attributes, which are, Type (number, text, date) and Value.

- Columnset: A ColumnSet is associated with one or more DataField. A DataField is created corresponding to each UIField that is used in the UIAction.
- Rowset: RowSet is derived from ColumnSet. A RowSet is created for every set of inputs that needs to be passed to the TestCase.
- Dataset: A DataSet consists of one or more RowSets, which represents a tabular structure for all the input sets passed to the TestCase.
- DataPool: A DataPool is associated with many DataSets. Multiple DataSets are required when one has to execute the same TestCase with different inputs at different testruns (which may be for different customers or different versions and so on)

3.1.4 Test Behaviour Package

The Test Behaviour section contains test scenarios defined as a sequence of test steps of various types. It comprises of the following elements:

- Scenario: A Scenario is a sequence of actions performed in the application. A Scenario is associated with optional Setup and TearDown procedures, which are of the type Scenario as well. Scenario is associated with one or more ScenarioUnits.
- Setup: Setup is derived from Scenario. It is a sequence of steps that are required to bring the system to the initial state required by the TestCase that will be executed.
- Teardown: TearDown is derived from Scenario. It is the sequence of steps that will be executed after the TestCase has been executed. Typically it is used to reset the database to its original state, or to delete the data created by the TestCases.
- ScenarioUnit: A ScenarioUnit represents a single step in the test sequence (Scenario). ScenarioUnits can be of different kinds, such as, UIAction, ControlFlow, VariableDefinition and Assert.
- UIAction: A UIAction is an action performed on a Screen. Examples of UIAction are filling a form and clicking the Submit button, entering the userid and password followed by clicking the Login button. UIAction is associated with zero or more UIFeilds and one ActionableUIField.
- ControlFlow: A ControlFlow is used to change the sequence of execution of ScenarioUnits based upon a condition. ControlFlow has attributes to define the condition and is associated with two ScenarioUnits, one of the TrueFlow and one for the FalseFlow.
- VariableDefinition: A VariableDefinition is used to hold a certain observation or value through the execution of the TestSuite. Variables can be assigned values in one TestCase and used in another TestCase. VariableDefinition is associated with Query which can be used to get its value.
- Query: Query is used to make an observation. Observations can be made on the user interface or the database or the environment. The observations are used to create the test oracle.
- QueryUI: QueryUI makes and observation on the user interface. It is associated with the UIField which it observes.
- QueryDatabase: QueryDatabase makes and observation on the database. It is has an attribute which contains the database query language specification of the database field to be queried.
- QueryEnvironment: QueryEnvironment makes and observation on the environment. It has an attribute that specifies a procedure which will execute in the environment and return a value.
- Assert: An assert is the test oracle. It uses a query (UI or database or environment) to make an observation and compares it with an ExepectedValue. ExpectedValue may be a value or another Assert.

3.2 Variation Points in ESTML

An analysis of the ESTML reveals the possible variation points. We assume that every concrete model element could have a variation point. Further, we assume that every attribute of a concrete element could have a variation point. This gives us list of elements that could have variation points. We identified 41 such elements shown in the table 1

In this list some concepts such as TestProject, TestLog are higher level concepts in the model hierarchy. Since we are concerned with variations in test cases and its elements, we eliminated these concepts from our list of variation points. Further, certain elements such as Scenario, DataPool, DataSet are used to represent collection of ScenarioUnits and DataFields respectively. We do not consider these are variation points because their variations would be represented by the elements in the collection. These elements have been striked out in table 1. This leaves us with a list of 29 elements that can have variation points.

Authors of [15] have conducted a study on test case reuse in enterprise software implementations. Test case reuse metrics for 6 implementations were analyzed in this study. In each implementation, test cases were reused from a reuse repository and modified to suit the enterprise specific requirements. In the course of these six implementations 8726 test cases were delivered, of which 6434 test cases were newly developed, 1016 test cases were reused from a test case repository and 1276 test cases were changed after reuse. The kinds of changes that were made to the reused test cases were : business process, scenario, form field, form navigation, naming conventions and test data. Variations in business processes and scenarios are manifested in the form of changes in the form fields or asserts (test oracles). Thus we concur that form field elements *UIField, Screen, ActionableUIField*, form navigation elements *Scenario, UIAction, ControlFlow* and assert elements *Assert, QueryUI, QueryDatabase* will be the variation points that will occur most frequently in the test cases for different enterprises. While the study was not on SaaS applications, the kinds of variations are expected to be of a similar nature because in both cases the software is an enterprise application.

Test UI	Test Behaviour	Test Data	Test Architecture
Screen	~~Scenario~~	~~DataPool~~	~~TestProject~~
Screen.Title	Scenario.ScenarioUnit	~~Dataset~~	~~TestSuite~~
Screen.UIField	~~ScenarioUnit~~	~~Rowset~~	SystemUnderTest
Container	Setup	~~Columnset~~	~~TestCase~~
Container.Label	Teardown	DataField	~~TestCase.Scenario~~
UIField	UIAction	DataField.Value	~~TestCase.DataPool~~
UIField.Label	UIAction.UIField	DataField.Type	~~TestLog~~
UIField.Type	UIAction.ActionableUIField		
ActionableUIField	VariableDefinition		
ActionableUIField.Label	QueryUI		
ActionableUIField.Type	QueryDatabase		
	QueryEnvironment		
	Assert		
	ControlFlow		
	ControlFlow.TrueFlow		
	ControlFlow.FalseFlow		

Table 1: Variation Points in ESTML

3.3 Common Variability Language with ESTML

Feature Models (FM) and Variability Language are the most prominent forms used to express variability. In a case study[17], a feature model was used to express variability and features were mapped to test cases. It was found that majority of the test cases span across multiple features which results in a many to many mapping between features and test cases. Such a mapping results in too many test cases being selected while doing feature-based selection and many of them are not necessarily relevant to the search. For this reason we chose to use the Common Variability language for expressing the variability in the ESTML.

Multiple strategies have been proposed to create the base model[3]. The *Additive* strategy the base model contains a minimum set of elements that are common to all variants. A library model is created with the additional elements required by the variants. In the *Subtractive* strategy the base model contains the complete set of elements required by all variants. CVL is used to remove the elements not required by particular variant. A third strategy would be a combination strategy where instead of creating a maximum or minimum model we create a model that is similar to most variants. Authors of [3] suggest that the objective should be to have a CVL specification that is compact and requires minimal substitutions.

The VSpec (Variability Specification) will specify the feature choices that are available in the Feature Specification Layer (FSL). The Iterator and CompositeVariability concepts can be used to specify the features and choices available at the variation point. The Product Realization Layer (PRL) will specify the different kinds of substitutions (value, reference or fragment) required for product realization. The resolution model will specify the choices made at each variation point. More details on modeling using CVL can be found at [5].

4. THE TEST SCRIPT VARIANTS GENERATOR

The Test Script Variants Generator (TSVG) is an approach for model driven test script generation using ESTML. The TSVG comprises two components, the Test Script Generator (TSG) and the Model Variants Generator (MVG). Refer to Fig. 2.

The authors of [14] have published a detailed description and evaluation of the TSG. The central idea behind the TSG is that testers should specify/model tests without worrying about how to automate them. However, specifying the tests should not take significant time/skill. The various components of the toolset have been described below :

4.1 Recorder

The TSG recorder works like a proxy server. The application under test is launched using this proxy. This enables the recorder to capture user actions, UI elements, input data, and so on. The recorder also allows the tester to add asserts during the recording process. Unlike existing tools which have recorders that create test scripts, the TSG recorder creates a model[14]. This allows the tester to create the test model without having to learn the model structure or any scripting language.

4.2 Model Editor

The test editor is an interface which provides a view of the test model for editing. In the current implementation it is an excel sheet that is generated from the model. After editing it can be imported back to update the model. The tester can edit test cases, scenarios, UI structures (Test UI model) and data pools (Test Data model) using this editor.

4.3 Script Generator

The Script Generator generates the test scripts into robot-framework - an open source framework[19] built on top of the open source tool Selenium. The scripts are executed using the Selenium Remote Control execution engine. An HTML log with test verdict, timing, screenshots, and so on is generated at the end of execution. In future, the test results will be sent back into the model to populate the TestLog element.

The script generator structures the output scripts into a

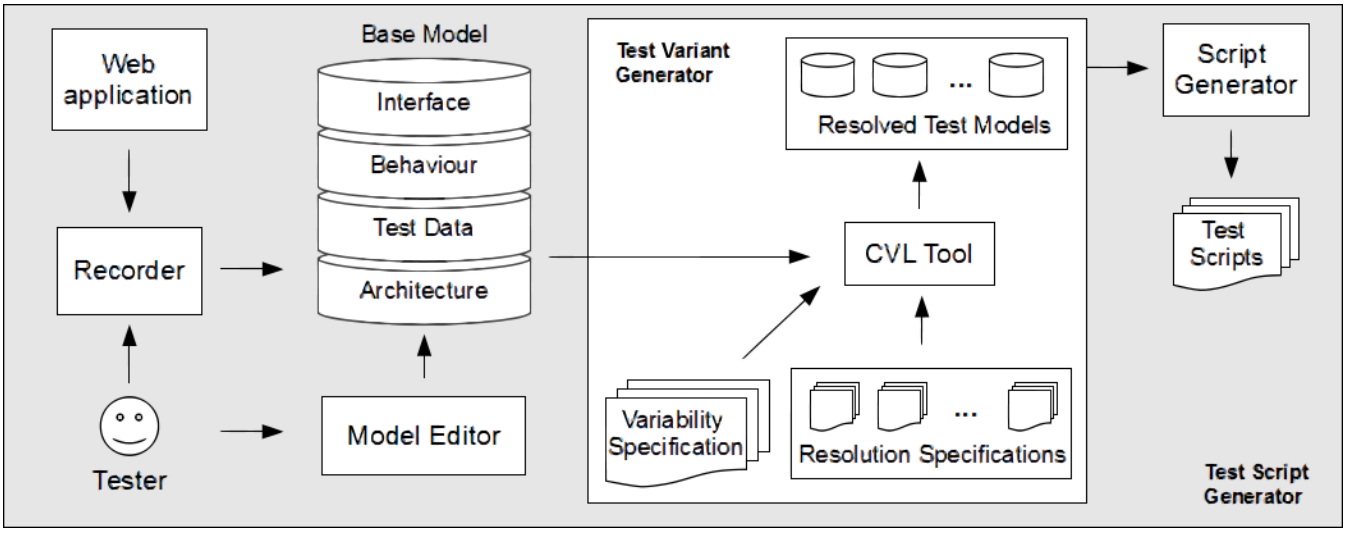

Figure 2: Test Script Variants Generator

modular form for better reuse and ease of maintenance. For example, it automatically identifies all the UI actions and related inputs for each screen and generates a reusable script for that action. Thus is builds an action library that can be used to create tests very quickly. For example, if a login process is recorded, the generator will automatically create a function named *Login()* that takes parameters corresponding to the fields on the screen, that are : *username*, *password* and clicks the actionable field, that is, *Submit*. Any test case which requires to Login would call this function from the Action Library. The generator provides options to configure the generated test script architecture. The appropriate test suite architecture depends on the dynamics of the application. In an application with many and frequently changing scenarios, a keyword-driven architecture may suit, whereas in an application with fewer scenarios but a large variety of input classes, a data-driven architecture may be better. The script generator allows the tester to make such a choice without any new script development effort.

4.4 CVL Tool

The CVL tool [4] is based on Eclipse Modeling Framework (EMF)and thus supports any DSL defined through EMF. The tool has a graphical editor and a tree editor for creating and viewing the CVL model including the variability model and the resolution model. It has a resolution model generator that is integrated with the CVL graphical editor. A resolution model can be generated from the variability model elements that the user wants to include in the resolution model by selecting them in the CVL graphical editor. The tool implements a model-to-model transformation to generate new resolved models in the base DSL.

5. EXPERIMENT WITH ORANGEHRM

In this section we describe the experiment that we conducted with an open source SaaS application - OrangeHRM[18].

5.1 Objective

The objective of this experiment is to validate the modeling approaches described in Section 3. Our research question is : Can we use the approach to express all kinds of test case variations across multiple configurations of a SaaS ?

5.2 Subject

The subject of our experiment is OrangeHRM, an open source Human Resource Management (HRM) system[18]. OrangeHRM is a web application that can be hosted on a cloud and it targets small and medium enterprises. The functionality includes personnel information management, leave management, recruitment management, training, performance evaluation and other HRM modules.

All modules of OrangeHRM have some configuration options. Many of the enterprise specific configurations can be achieved through creating appropriate Master Data. These includes data fields such as : job titles, salary components, grades, work shifts, leave types, organization structure and so on. We have chosen the Personnel Information Management (PIM) module for our experiment. The module is most appropriate for our experiments because it has many configuration options. It has the option to add custom fields to any of the PIM screens, and to select some optional fields such as Social Security Number (SSN) and optional menu items such as US Tax Exemptions. Further, many of the configurable master data fields are used in this module.

The PIM module is used to manage employee information. It allows the HR user to create, delete and update employee information such as name, address, contact details, dependents, reporting structure, tax exemptions, qualifications and salary account details. One can view the job and salary history in this module.

A test engineer who does not have the knowledge of ESTML and CVL was recruited for this experiment. The engineer required 4 weeks of learning to be able to perform the experiment. This includes understanding concepts of test automation and model driven engineering. The engineer had to get familiarized with ESTML, CVL, OrangeHRM and the TSVG toolset. The four week learning was essential to help our recruit to have the same skills as the target population. All the steps mentioned in the next section have been executed by the test engineer.

5.3 Execution

5.3.1 Configuring OrangeHRM

We configured OrangeHRM for 2 enterprises A and B. The details of the configuration have been listed in Table 2. OrangeHRM allows the Admin user to make these configurations through its user interface.

5.3.2 Creating the Base Model

We developed a set of test cases for the PIM module which includes the the major functions of PIM, that are : searching, creating, deleting and editing employee information. The test cases were recorded using the TSG recorder. This recording was done on configuration A. We have chosen to conduct our experiments with the subtractive approach. For this we have to create a base model with elements required by all configurations. The model at this point contains the recorded test cases and all Test UI elements of configuration A. However, the configuration B has an additional field, that is UID Number. We added this field through the Model Editor.

5.3.3 Creating the Variability Specification

We created a new CVL model using the Eclipse CVL tool plug in. In the CVL model, we created a hierarchy similar to the ESTML using *CompositeVariability* objects. We have followed the steps suggested in the CVL User Guide[6] for creating the VSpec.

For the UIModel, we have to create a VSpec which specifies that a certain object reference may or may not exist. This choice is specified using the *Iterator*. An *Iterator* was created for each UIField SSN, UIDNumber and ActionableUIField USTaxExemptions. For the realization, a *ReferenceSubstitution* has to be created for each of these with an Empty *ReplacementObject*. However, the ESTML has an additional reference to the UIField. For every UIField in the screen, the corresponding UIAction has an test data field reference. This specifies the test data field that provides input to the UIField. When we add or remove a UIfield this reference has to be created or removed as well. Hence we created a *FragmentSubstitution* for all the UIFields. The *FragmentSubstitution* replaces the 2 references UIField and UIAction.Input by an Empty *ReplacementFragment*. See Fig. 3.

In the Behaviour Model, we have the scenarios used in the test cases. Table 3 shows a test case in which the testers logs in, selects an employee from the list and edits the information. One of the steps in this test case is to open the "US Tax Exemptions Menu" and edit the tax details. This is an optional step which is performed only if an organization is a US organization. This optionality is specified using an Iterator. While a *ReferenceSubstitution* may be used to realize the variability, in this test case there are 2 UIActions, "Open Tax Exemtions" and "Edit Exemption Details" that have to be substituted. Hence, we use a *FragmentSubstitution* to substitute both the steps with Empty *ReplacementFragment*.

In the TestData section, we have to specify a choice of test data values for the last step "Add Skills". The options in the Skills drop down changes with variants. The test data value "Sales" has to be replaced with "Assembly Line". The choices are specified by using an *Iterator* and substitution is acheived using a *ValueSubstitution*.

5.3.4 Creating the Resolution Models

We created two resolution models corresponding to each configuration A and B. This is achieved by creating *ResolutionElement* corresponding to each Vspec and specifying a *ResolutionValue* as 0 or 1 to specify the choice for the corresponding *Iterator*. The selections for Resolution A and B are made in accordance with the choices specified in Table 2.

5.3.5 Generating the test scripts

The CVL transformation tool was executed to generate 2 ESTML complaint models corresponding to Resolution A and Resolution B. These models were input to the TSG to generate the Selenium test scripts. As was expected, the scripts for configuration A had the SSN field and the US Tax Exemptions Menu, while the scripts for configuration B had the UID number field. The test steps "Open US Tax Exemptions" and "Edit Tax Exemptions" were there only in Resolution A. The test data value for Skills was "Sales" in resolution A, and was "Assembly Line" in resolution B.

5.4 Observations and Analysis

The experiments show that the identified cases of variations in test cases can be modeled using the ESTML and CVL. The TSG approach has been validated before [14]. It has been used to automate the test execution for an online banking system comprising of 250 complex test cases. Hence we focus this discussion on feasibility of CVL for the expression on variability in test cases. We were able to model all the identified variation points identified in the experiment. Following are some observations made in the process.

- Knowledge of the Test Model : The CVL user is required to have thorough knowledge of the ESTML. This is an area of concern for us, because testers may not fully understand the dependencies across test model elements. They create the test scripts through the recorder. The TSG comes with an editor through which all the changes to the test model are made. The recorder and editor are designed to handle dependencies within model elements. Thus users of TSG are not required to understand the test model. However, if they have to use CVL for variability specification, they will have to learn the models.

- Additions to the User Interface : While adding new elements such as *Screens, UIFields, UIActions* to the model sections are populated by the TSG recorder. The recorder captures *UIField* attributes such as XPath, field type, field label automatically. Making such additions using a library model and CVL will be effort intensive.

- Creating placement fragments : We observed that to achieve a simple variation such as removing a *UIField*, we have to create a complex fragment substitution specification. This makes the approach difficult to use.

- Test Data variations : The TSG is capable of generating test scripts in a data-driven architecture. In this architecture the test data specification is separated from the test specification and the tester can create multiple data sets which can be used to run tests with

#	Parameter	Enterprise A	Enterprise B
1	Job Titles	sales person, cashier, store manager, administrative assistant	worker, floor supervisor, plant manager, administrator
2	Job Categories	front office, back office	laborer, professional, helper, manager
3	Pay Grades	FOJunior, FOSenior, BOJunior, BOSenior	WorkerJunior, WorkerSenior, Manager, Assistant
4	Employment Status	contract, full time	contract, full time, part time
5	Organization- Structure	Store, Billing, Admin	Plant, Admin
6	Qualification- Skills	Accounting, Sales, Inventory Management, Administration	Assembly Line, Engineering, Administration, Project Management
7	Custom Fields	None	UID number (Personal Details)
8	Optional Field	SSN	None
9	Optional Menu	US Tax Exemptions	None

Table 2: Configurations for Enterprises A and B

#	Description	Test Steps	Test data	Expected
1	Login and edit employee information	Login	username = Admin, password = admin	Message: Saved Successfully should be displayed at the top
		Open Employee List		
		Select employee	employee number = 001	
		Edit Personal Details	SSN = 100100100, Middle Name = Peter	
		Open Tax Exemptions		
		Edit Exemption details	Federal Income Tax Status = Single, Exemptions = 2	
		Open Qualifications		
		Add Skills	Skill = Sales, Years of Experience = 4	

Table 3: A test case for the orangeHRM-PIM module

large number of inputs. When there is variation in the test data across variations the user has 2 choices. One choice is to create multiple data pools corresponding to each test variant. Second choice is to write a CVL *ValueSubstitution* for every varying DataField. We found that using the CVL method for specifying test data variations was far too effort intensive as compared to using the TSG data-driven architecture.

While there is a need for work arounds at this point, the results of our experiments are motivating. In our organization we have observed a SaaS application that has as many as 150 configurations and more than 100 test scripts to automate. Using the current methods the testers would have to develop a huge number of test scripts along with a reuse strategy, which would be highly effort intensive. We think that the TSVG approach will make it possible to achieve a magnitude of reduction in the test script engineering effort. Amongst the concerns listed above the common issue is the dependencies in the test model. To deal with such issues we can develop a Model Validator tool which can be executed after a CVL model transformation is applied. The Model Validator can add or remove the dependencies in the model and make it complaint to the ESTML.

6. RELATED WORK

A review of the literature in variability management[2] indicates that Feature Models (FM) and UML models have been most prominent forms used to express variability. This strategy is suitable in a situation where the testing team has access to product engineering artifacts. In the IT Services industry, there are independent testing teams who may not have access to the product engineering artifacts. In such cases, there is a need for variability management in test assets[16].

Authors of [11] propose a method to define variability in extended UML activity diagrams and use them as a basis for testing. [10] proposes an approach to develop domain test cases from use cases that contain variability and to derive application test cases from them. [9] proposes a mechanism to model variability in a UTP model. However, the behavior specification in UTP has to be one of the UML structures such as a state-transition diagram, activity diagram or sequence diagram. These are not adequate for the representation of enterprise software test cases[16]. [8] uses a state transition diagram to model the behaviour of the system under test and CVL to specify variability. The models are used by their tool to generate JUnit test cases. [7] proposes a method to support variability of enterprise services in the cloud, which uses principles of aspect-oriented software development to modularize the variability concerns. This method has a similar objective as ours, but is aimed towards the development process.

The differentiator of our approach is the ESTML designed for specification of test cases of enterprise software with CVL

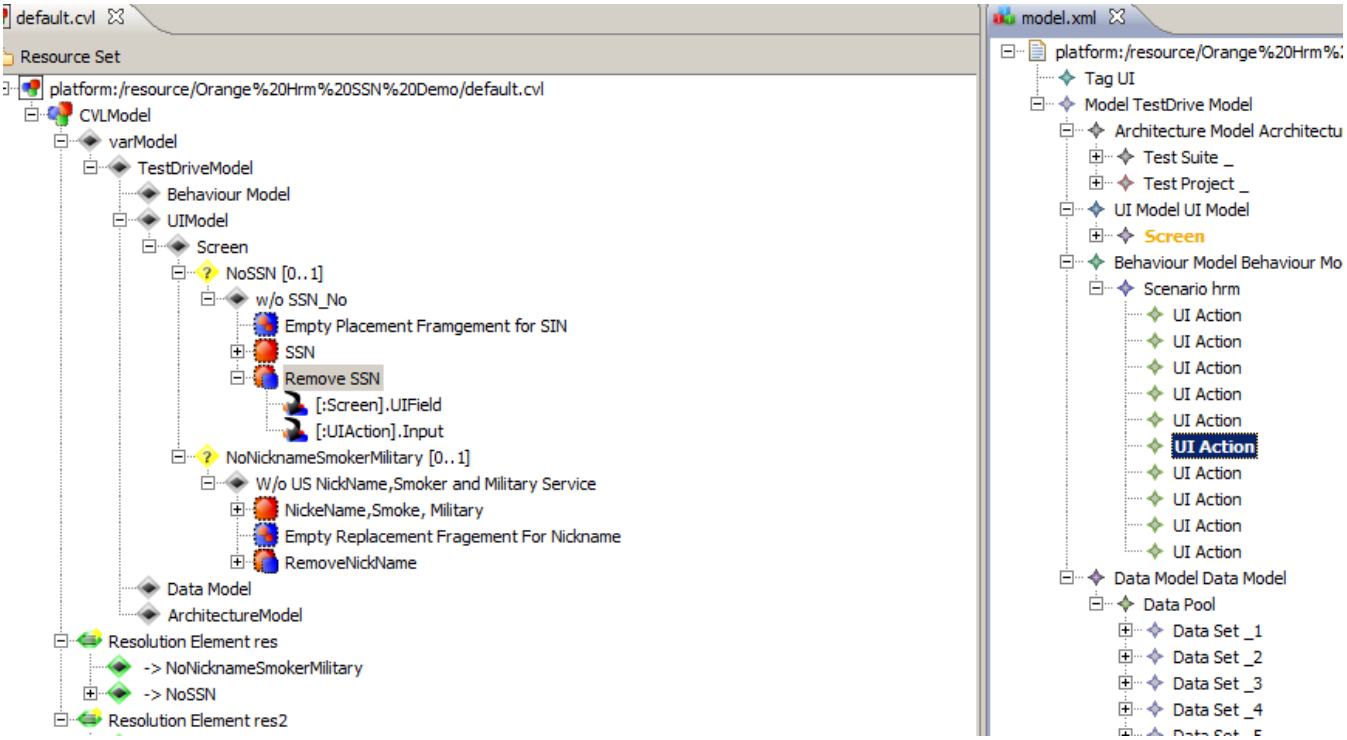

Figure 3: HRM system example - Removing the SSN field from the base model

for variability management. Further it achieves automation of the complete test script engineering life cycle.

7. CONCLUSIONS AND FUTURE WORK

We presented an approach to generate automated test scripts for multiple configurations of a SaaS. The approach uses a Test Model designed represent test cases of enterprise software. The Common Variability Language is used to specify variability in the test model. We described the toolset which implements the approach, comprising of the recorder, script generator, model editor and the CVL tool. We analyzed the test model elements to identify possible variation points and discussed how the variation points could be implemented used a CVL construct. Lastly we describe an experiment that shows that the approach is feasible. We found that while all the kinds of variations can be handled by the approach, it takes a considerable amount of time to model certain variations. These are variation points in which there are references across model elements which have to be created or removed depending on the variation. Listed below is the work we plan to do in the future.

- Validation in practice : While we have validated the expressibility of the approach, it is important to validate its efficacy in real world constraints.
- Automatic generation of CVL specifications : After the SaaS is configured, the configuration parameters and choices made would be stored in the database or a configuration file. We believe this information can be used to automatically extract the variability and resolution specifications.
- Integrated User Interface : The model editor that is part of the TSG has been designed to specify test cases in a format that testers are familiar with. It allows them to model test cases without understanding the test model. If testers have to use CVL to specify variability, they would have to understand the test model. This is a major issues with regards to the usability of the approach. We would like to develop an integrated model editor interface in which test cases as well as CVL specifications can be specified.
- Test Model Reuse across products and industries : Owing to similarity in the domain, there is a lot of commonality across products and product configurations within the same industry vertical. Other than products, there exist variations in the business process and technology platforms. Dealing with these multiple dimensions of variations along with having a compact variability specification is challenging. We would like to develop techniques to handle such multiple causes of variation during test case reuse.

This work is a part of our mission to build methods and tools to engineer tests for multiple variants of software. The test model is the basis for all testing activities, such as, test design and specification, automated script development, variability management, reuse, test optimization, test selection and prioritization.

8. REFERENCES

[1] Oystein Haugen, Birger Moller-Pedersen, Jon Oldevik, Gran K. Olsen and Andreas Svendsen : Adding Standardized Variability to Domain Specific Languages. In : Proceedings of the 2008 12th

International Software Product Line Conference, IEEE Computer Society, Washington, DC, USA, 139-148. (2008)

[2] Lianping Chen, Muhammad Ali Babar and Nour Ali : Variability management in software product lines: a systematic review : In : Proceedings of the 13th International Software Product Line Conference (SPLC '09). Carnegie Mellon University, Pittsburgh, PA, USA, 81-90. (2009)

[3] Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg, Franck Fleurey, Oystein Haugen, Birger Moller-Pedersen, and Goran K. Olsen. : Developing a software product line for train control: a case study of CVL. In : Proceedings of the 14th international conference on Software product lines: going beyond (SPLC'10), Jan Bosch and Jaejoon Lee (Eds.). Springer-Verlag, Berlin, Heidelberg, 106-120. (2010)

[4] CVL Tool http://www.omgwiki.org/variability/doku.php/doku.php?id=cvl_tool_from_sintef

[5] CVL Specifications http://omgwiki.org/variability/lib/exe/fetch.php?media=cvl-revised-submission.pdf

[6] CVL User Guide http://www.omgwiki.org/variability/lib/exe/fetch.php?media=cvl1.2_user_guide.pdf

[7] Jegadeesan, H.; Balasubramaniam, S: A Method to Support Variability of Enterprise Services on the Cloud. In: IEEE International Conference on Cloud Computing, vol., no., pp.117,124, 21-25 Sept. 2009 (2009)

[8] Garcia Gutierrez, Bonifacio and Garcia Carmona, Rodrigo and Navas Baltasar, Alvaro and Parada Gelvez, Hugo Alexer and Cuadrado Latasa, Felix and Duenas Lopez, Juan Carlos: An automated Model-based Testing Approach in Software Product Lines Using a Variability Language. In: Third Workshop on Model-Driven Tool and Process Integration, MDTPI 2010, 15-18 June 2010, Paris, Francia (2010)

[9] Lamancha, B.P., Usaola, M.P., Velthius, M.P. : Towards an automated testing framework to manage variability using the UML Testing Profile. In : Automation of Software Test, 2009. AST '09. ICSE Workshop on , pp.10-17 18-19 May 2009 (2009)

[10] Erik Kamsties, Klaus Pohl, Sacha Reis and Andreas Reuys. : Testing Variabilities in Use Case Models. In : Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Lecture Notes in Computer Science, Vol. 3014 (2003)

[11] Andre Heuer, Vanessa Stricker, Christof J. Budnik, Sascha Konrad, Kim Lauenroth, and Klaus Pohl. : Defining variability in activity diagrams and Petri nets. In : Journal Science of Computer Programming, Volume 78 Issue 12, December, 2013, Pages 2414-2432 (2013)

[12] UML Testing Profile http://www.omg.org/technology/documents/formal/test_profile.htm

[13] Knowledge Discovery Meta model http://www.omg.org/technology/kdm/index.htm

[14] Sachin Patel, Priya Gupta, Prafullakumar Surve: TestDrive - A Cost Effective Way to Create and Maintain Test Scripts for Web Applications, In: 22nd International Conference on Software Engineering and Knowledge Engineering, pp.474-476, 1-3 July, 2010 (2010)

[15] Sachin Patel, Ramesh Kumar Kollana: Test Case Reuse in Enterprise Software Implementation - An Experience Report, In: IEEE Seventh International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA, pp.99-102, (2014)

[16] Sachin Patel: Practical problems with modeling variability in test cases - an industrial perspective, In: The Eighth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis,France, January 22-24, 2014 (2014)

[17] Sachin Patel, Priya Gupta and Vipul Shah: Feature Interaction Testing of Variability Intensive Systems, In: 4th International workshop on Product Line Approaches in Software Engg., ICSE 2013 (2013)

[18] OrangeHRM http://www.orangehrm.com/

[19] Robot Framework http://www.robotframework.org