Towards Domain-Specific Testing Languages for Software-as-a-Service

Dionny Santiago, Adam Cando, Cody Mack, Gabriel Nunez,
Troy Thomas, and Tariq M. King

Ultimate Software Group, Inc.
2000 Ultimate Way, Weston, Florida 33326, USA
{dionny_santiago,adam_cando,cody_mack,gabriel_nunez,
troy_thomas,tariq_king}@ultimatesoftware.com
http://www.ultimatesoftware.com

Abstract. There continues to be a trend towards using the power of cloud computing to tackle inherently large and complicated problem domains. Validating domain-intensive cloud applications presents a significant challenge because of the complexity of both the problem domain and the underlying cloud platform. In this paper, we describe an approach that leverages model-driven engineering to improve testing domain-intensive cloud applications. Our approach combines a set of abstract test commands with various domain and configuration models to define a domain-specific testing language. We have developed a prototype of our approach that provides language editing and platform configuration tools to aid test specification, execution and debugging.

Keywords: Testing, Model-Driven Engineering, Domain-Specific Languages, Cloud Computing, Human Capital Management Software

1 Introduction

Validating software-as-a-service applications is difficult due to the large size of the problem domain, coupled with the complexity of the underlying cloud platform. Adequate functional testing is heavily dependent on domain expertise from each product area, and typically requires extensive data setup. Since the software is delivered as a service over the Internet, functional UI testing must be performed using different browsers to ensure a good user experience. In addition, engineers need to be able to set up tests to run on specific configurations of the underlying cloud infrastructure.

Model-driven engineering (MDE) seeks to simplify software development by raising the level of abstraction through domain modeling, while promoting communication between groups working on the same system [1]. Researchers and practitioners have developed a number of MDE tools and techniques, most of which have focused on exploiting domain models for automatic code generation. However, there has also been research on MDE approaches to enhance software

testing [2, 3]. A more recent interest that has arisen is the use of MDE to support emerging paradigms such as adaptive and cloud computing [4].

In this paper, we describe an approach that leverages MDE to improve the specification, execution, and debugging of functional tests for domain-intensive cloud applications. Our approach is the result of investigating new and innovative ways to test UltiPro, a comprehensive cloud-based human capital management (HCM) solution [5]. Domain models and abstractions for common UI interactions, data setup, environment and platform configurations are combined with highly extensible testing frameworks [6–8]. The result is a powerful domain-specific language (DSL) for creating automated functional tests. Our test authoring DSL is supported by an editor that provides syntax checking and highlighting, intelli-sense, tooltips, and debugging features.

The major contributions of this research paper are as follows: (1) describes a novel approach that integrates various domain and configuration models into a test case specification language for cloud applications; (2) presents the design of a prototype used to demonstrate the feasibility of the approach; and (3) discusses our experience developing the prototype, focusing on the lessons learned. The rest of this paper is organized as follows: the next section motivates the research problem. Section 3 describes our domain-specific test case specification approach for cloud applications. Section 4 presents a prototype that implements the proposed approach. Section 5 discusses the lessons learned from building the prototype. Section 6 is the related work and Section 7 concludes the paper.

2 Motivation

Our research has been motivated by the challenges faced when testing UltiPro [5]. Delivered on-demand as software-as-a-service in the cloud, UltiPro provides HCM functionality including recruitment, onboarding, payroll, payment services, benefits, compensation management, performance management and reviews, succession planning, and more. Data is available across all areas of HCM, and can be accessed by department, division, or country. Several reporting and analysis features are also available through UltiPro's web-based portal.

The large size and complexity of the problem domain makes testing the functionality of UltiPro challenging. Individual product areas (e.g., recruitment, payroll) encompass so many features that each area could be considered as a product itself. Adequately testing UltiPro requires each product area to be validated, which is impossible without domain expertise. Although each product area is large, UltiPro has been designed and developed as a unified solution which seamlessly integrates all aspects of HCM. Validation of the overall product therefore relies heavily on the collaboration of domain experts across all product areas. This ensures that changes to one product area does not have an adverse effect on other product areas.

Testing UltiPro is further complicated because of its development and delivery as a cloud application service. Cloud application services are hosted on complex, distributed infrastructures with multiple servers and architectural lay-

ers that extend from the underlying network up to the web-based user interface. To ensure a good user experience, functional UI testing must be performed using different web browsers. Other non-functional factors such as high performance and security requirements also make it difficult to test cloud applications. However, this paper limits the scope of the testing problem for cloud applications to functional UI and platform compatibility testing.

3 Approach

Our approach defines a test specification language that can be used to develop automated tests for a particular application domain. As shown in Figure 1, we leverage abstract test commands, domain and platform models, and test automation frameworks for the purpose of creating a domain-specific language (DSL) for testing cloud applications. The DSL allows us to provide test case editing, execution, and debugging tools tailored for domain experts, test engineers, and end users. Abstract tests defined using the DSL are translated into executable scripts that run on the underlying testing tools and frameworks. For the remainder of this section, we describe the various test abstractions and models used in our approach. Transformation of abstract tests into tool-specific testing scripts is discussed as part of the prototype design in Section 4.

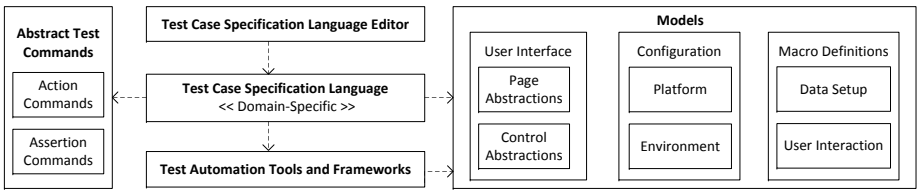

Fig. 1. A Domain-Specific Testing Language for Cloud Applications

3.1 Abstract Test Commands

At the core of the language is a set of abstract test commands. There are two types of commands: `Action Commands` and `Assertion Commands` (left of Figure 1). Action commands apply inputs that exercise the system under test (SUT). This includes stimulating UI controls such as textboxes, dropdowns, and buttons, as well as database-related actions. On the other hand, assertion commands perform UI and database checks to verify the behavior of the SUT. Table 1 describes some of the key test commands defined under our approach.

3.2 Application Domain Models

Domain concepts are introduced into our testing language through two types of models: *User Interface* and *Macro Definitions* (right of Figure 1).

	Command	Description
Actions	Blur	Loses focus of an element
	Clear	Empties the value of an element
	Click	Presses and releases mouse button while hovering over an element
	Mouse Over	Hovers the mouse pointer over an element
	Mouse Out Of	Move the mouse pointer from within the hover area of an element
	Set	Assigns a value to an element
Assertions	Is	Checks if the value of an element equals a given value
	Is Like	Checks if the value of an element contains a given value as a substring
	Is Visible	Checks if an element is visible
	Is Enabled	Checks whether an element used for input is enabled
	Exists	Checks if an element is present
	Has Options	Checks if a dropdown provides a specified list of values
	Has Number Of Options	Checks if the size of a dropdown list is equal to a specified value

Table 1. Web UI Abstract Test Commands

User Interface This model is a generalization of the user interface of the SUT. For example, in the case of UltiPro the UI model consists of abstractions representing its web pages and controls. These page and control objects encapsulate the CSS selectors used to identify web elements in the HTML source document tree. Automated tests then reference these abstractions instead of the implementation details, which makes tests easier to maintain as the application changes [8, 9]. Furthermore, page and control objects are named using domain-specific concepts. For example, a grid control used for entering pay data would be named `PayDataEntryGrid`. Using such terms allows domain experts and end users to easily identify and specify various aspects of the SUT.

Macros A macro in computer science is a pattern that specifies how a sequence of inputs is mapped to a replacement input sequence. Macros are often used to make programming tasks less repetitive and less error-prone. Our approach leverages the benefits of macros to improve test specification. Testers can define frequently used test setup, input, or assertion command sequences, and store them in a central location. These macros are then named using domain-specific terms, and integrated into our testing language. Similar to our abstract test commands, test macros can target user or database interactions.

3.3 Configuration Models

Several abstractions for configuring the underlying platform and environment of the SUT are integrated into our language. These include abstractions for: *Server Environment Configuration* – Application servers, web servers, database servers, reporting servers that make up the test environment; *OS/Platform Configuration* – Operating systems on which to test the desktop and mobile versions of the cloud application; *Web Browser Configuration* – Clients to use during cross-browser compatibility testing, e.g., Chrome, Firefox, Internet Explorer or a combination thereof; and *Test Harness Configuration* – Modes and settings that allow users to tweak aspects of test execution including timing characteristics, logging, among others.

```
1  Summary
2      Purpose   Validate Run Payroll Feature
3      Authors   Dionny Santiago, Adam Cando
4      Configs   .NET, Payroll-14
5
6  Declarations
7      UltiPro UltiPro
8
9  Setup
10     Given I Setup a Payroll (...)
11     And launch UltiPro
12     And login as a Payroll Administrator
13       By setting the UserNameTextBox to "admin",
14                     PasswordTextBox to "ulti"
15       And clicking on the LoginButton
16
17 Tests
18     Scenario: Payroll Should Run to Completion
19     Given I Navigate to the Payroll Overview
20     When I click the StartPayrollButton
21     Then the ProcessStatusLabel is "Completed."
22     And the CreateBatchesButton is enabled
```

Fig. 2. Example Test Case Specification

3.4 Illustrative Example

Figure 2 presents an example test case specification defined using our approach. The example test consists of four main blocks: Summary, Declarations, Setup, and Tests. The summary block (Lines 1-4) holds meta information about the test, which includes a purpose, authors, and various configurations. In Line 4, `.NET` is a configuration that runs tests against the UltiPro desktop web application using three browsers for compatiblity testing. On the same line, `Payroll-14` sets up the test to run on a server environment configured for payroll processing.

Applications and data that will be used throughout the test must appear in the declarations block (Lines 6-7). In this block, automatic word completion popups (i.e., intelli-sense) is filtered to a list of available application models and database types. Line 7 of Figure 2 specifies that the UltiPro application model will be referenced whenever the name `UltiPro` appears in the test.

The setup block (Lines 9-15) contains a set of preconditions for the test, written using a behavioral-driven development (BDD) style syntax. Line 10 illustrates the use of a database macro to `Setup a Payroll`. Note that elipses are used to mask the actual parameters passed into the macro. Inline, users can declare high-level test steps (Line 12), and define how those steps are implemented as actions or assertions on the application model (Lines 13-15).

Test cases appear within the block named `Tests` (Lines 17-22). Line 18 provides a name for the single test in the example, while Line 19 demonstrates the usage of a UI navigation macro. The `click` action command is illustrated in Line 20, while Lines 21 and 22 show the `Is` and `Is Enabled` assertion commands.

4 Prototype

This section presents the setup, design and implementation of the Legend prototype, which was developed to demonstrate the feasibility of the proposed approach. Legend primarily consists of tools for authoring, executing and debugging tests written to validate UltiPro [5].

Legend has been developed in C# as a Visual Studio (VS) extension. The VS 2010 SDK provides components for extending the VS Editor with a custom language service. The Legend language service supports many of the VS SDK features including syntax coloring, error highlighting, intelli-sense, outlining, tooltips, and debugging. Integration of application and configuration models using domain concepts is a novel feature of Legend that separates it from other test specification languages and tools.

The underlying testing framework used to run Legend tests is an in-house tool called Echo [8]. Echo was developed based on Selenium, a cross-browser web UI automated testing framework [7]. Page and control abstractions for the application domain model are defined using Echo, in accordance with the page object pattern [9]. Frameworks such as MbUnit [6] are used to provide capabilities such as test fixture setup and teardown, data-driven testing, and reporting. The custom tooling developed for the prototype is divided into two categories: *Editing Tools* and *Configuration Tools*.

Editing Tools Figure 3 provides a UML package diagram showing the design of Legend DSL Editor. As shown in Figure 3, the editor is comprised of three major packages: `EditorExtension`, `ApplicationModelIntegration` and `CodeGeneration`. Key classes from each package, along with their interdependency relationships are also shown in the diagram.

The `EditorExtension` subsystem (top of Figure 3) contains the components that implement token colorization, syntax checking, block outlining, and intellisense. This subsystem is the main point of interaction between the Visual Studio editor and the Legend code extensions. The classes with the stereotypes `Providers`, `Controllers`, `Taggers`, and `Sources` are derived from the Visual Studio SDK, and directly interact with the .NET Managed Extensibility Framework (MEF) [10]. Classes stereotyped `Augmentors` and `Services` represent our custom extensions. The `LanguageService` class coordinates several of the interactions between the editor and the application models.

Integration with the application model is achieved via the `ApplicationModelIntegration` subsystem (bottom-left of Figure 3). It contains two types of classes: `ModelProviders` and `Models`. The `ModelProvider` classes use reflection to read the page objects, control objects, macros and elements that make up the `Models`. It is also responsible for filtering intelli-sense on the model, given a specific test context. For example, at the point where a test declares access to a particular web page, the `ModelProvider` scopes the word completion picklist for elements to include only elements that appear on that page.

Lastly, the `CodeGeneration` subsystem (bottom-right of Figure 3) provides logic for translating the abstract tests written in Legend into code executable

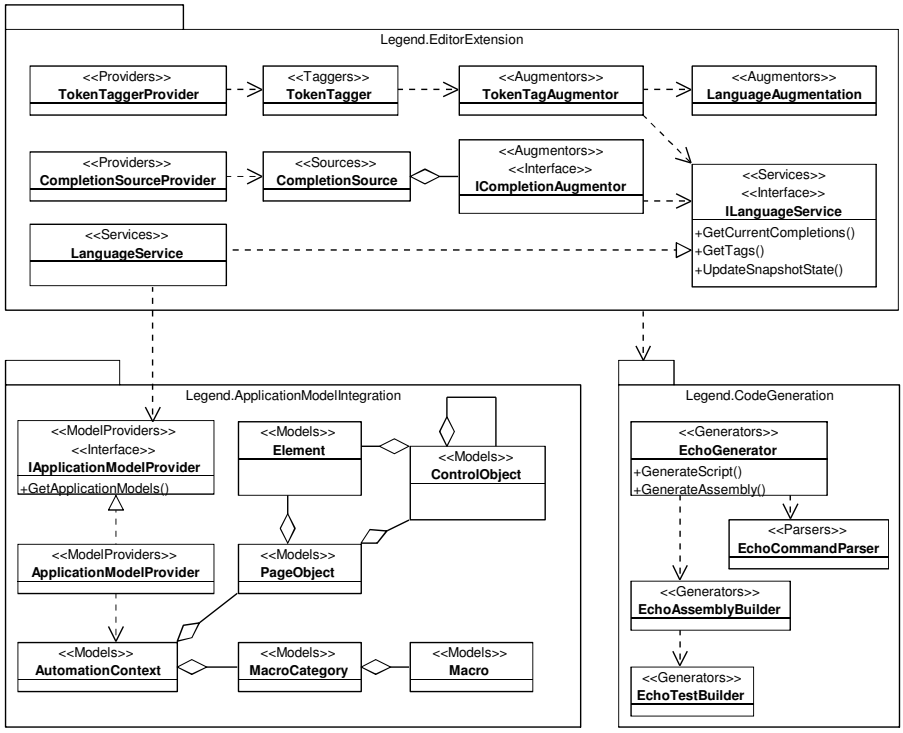

Fig. 3. Diagram showing the package structure and class design of the prototype

by the Echo framework. In order to support debugging at multiple levels of abstraction, the `EchoGenerator` has two modes of generation: *Script* – generates test scripts written in the Echo syntax; and *Assembly* – generates a Common Intermediate Language (CIL) representation of the Echo test. Our prototype maps the CIL to the domain-specific test steps. A test can therefore be executed and debugged at the level of the domain-specific test language, or the script language of the underlying Echo testing framework.

Configuration Tools Abstractions for the test environment and platform configurations are implemented in two distinct XML files called `Environments.xml` and `Parameters.xml`. The environments XML contains a list of all the test environments that are connected to a tool that can automatically request the latest UltiPro build. The file is auto-generated and populated with the unique identifiers used by each team to refer to their test environments. Information related to the specific network and database servers, along with any credentials for authentication are also stored in the `Environments.xml` file.

The `Parameters.xml` allows users to specify a range of configurations ranging from desktop web settings such as browsers and languages, to mobile web settings such as device screen size and orientation. Although each team can create their own configurations, there are a set of fixed configurations that are available for

use across all teams during regression testing. The data in both the environments and parameters XML files are passed directly to the Echo testing framework prior to test execution.

5 Lessons Learned

A key factor that contributed to the successful development of the prototype was having robust, highly extensible and configurable underlying testing frameworks. Echo and Selenium provided the implementations to support many of the abstractions described in the approach. Building the application models required a collaborative effort among developers, testers and domain experts. Developers would create the page objects and control objects, while testers and domain experts made sure they were named and exposed appropriately for testing. Macro creation was primarily done by testers and domain experts, with occasional assistance from the developers if necessary.

One of the more complex aspects of the prototype implementation was the module for keeping track of the test context in order to filter intelli-sense. This required the creation of a state-based rules engine to allow us to perform different editor actions based on previously specified lines in the test. Although challenging to implement, this feature was necessary to provide meaningful intelli-sense that guides testers during test creation. In other words, testers are only presented with commands, model elements, and other keyword suggestions if they are applicable in the current context.

Based on initial responses to prototype demonstrations, a major benefit of Legend is the ease with which test cases can be specified and reviewed by non-technical users. The tool can therefore be leveraged by domain experts and end users during acceptance testing. It also allows these stakeholders to assist in debugging issues using a language they understand, and without being concerned with the low-level implementation details of the test automation. Since tests are specified in an english-like syntax, using Legend could reduce or eliminate the need to maintain a separate inventory of test documentation. However, further evidence through a case study or empirical evaluation is needed to be able to fully validate these claims.

A limitation of the current prototype is the lack of an externalized point of extension for the test commands and their syntax. Since domain experts, testers and developers from several teams will be defining new model elements as the application evolves, we need to provide a mechanism that allows new page or control-specific commands to be easily added to the language. Web UI elements with dynamically generated identifiers are also not supported by the prototype, or the underlying Echo testing framework, but are planned for future releases.

6 Related Work

Although the use of domain modeling to support software engineering is not new, only a few researchers have leveraged MDE and DSLs to support software

testing [2, 3, 11]. Kanstren and Puolitaival [3] present the OSMOTester approach and tool that is very related to our work. OSOMOTester automatically generates tests with domain-specific concepts. A domain expert is used to construct a test model of the system, which is combined with a domain-specific modeling language that constrains and guides test case generation.

Hernandez et al. [2] describe a model-driven technique for designing platform independent tests for validating web-based applications. These platform independent tests are then combined with a model of the web technologies used to implement the application, and generate platform-specific tests [2]. Kuhn and Gotzhein [11] present an approach that uses configurable simulations to do platform-specific testing. They extend the UML testing profile to include platform models for real deployments, and describe how to use these models to test embedded systems by simulating various hardware configurations.

Some researchers have proposed model-driven approaches that aid the development of high-performance and cloud computing systems [12]. Palyart and Lugato define a high-performance computing modeling language (HPCML). HPCML provides constructs for specifying different concerns in high-performance scientific computing such as mathematics, parallelism, and validation. Nagel et al. [13] introduce a meta-model for specifying bindings between business processes and cloud services considering service-level agreements. They further extend that meta-model to support dynamic adaptation of cloud-based services.

There are several general purpose behavioral-driven development (BDD) testing tools that help to tie acceptance tests to business requirements [14–16]. Similar to Legend, such tools aim to bridge the gap between domain experts, developers, and testers [16]. These tools typically work by creating and linking two sets of files – specifications and step definitions [15]. Legend combines these two activities into a single, domain-specific test authoring experience. Our research on Legend extends previous work on the Echo Web UI Test Automation Framework [8]. Echo provides a thin layer of abstraction on top of Selenium [7], and adds several features including command timeouts, wait throttling between commands, database interaction, and environment and test configuration.

7 Conclusion

The work in this paper presented an approach that applied model-driven engineering to the development of a domain-specific test case specification language. Our approach is general in the sense that it can be applied to any domain, but in terms of technologies we focus on web-based applications which are deployed on cloud computing platforms. We have implemented a prototype of the proposed approach for a cloud-based human capital management solution. Developing the prototype gave us first-hand experience on some of the benefits and challenges associated with creating a DSL for testing UltiPro. Feedback from interactive prototype demonstrations has been positive. Our next steps are to develop a full implementation of Legend, and perform a case study using data from UltiPro.

Acknowledgments. The authors would like to thank Jorge Martinez, Michael Mattera, and members of the Virtual Team at Ultimate Software for their contributions to this work. We also give thanks to the judges and participants of the Summer 2012 Ultimate Software 48 Hours Project for their valuable feedback. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the Ultimate Software Group, Inc.

References

1. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2006)
2. Hernandez, Y., King, T.M., Pava, J., Clarke, P.J.: A meta-model to support regression testing of web applications. In: SEKE. (2008) 500–505
3. Kanstrén, T., Puolitaival, O.: Using built-in domain-specific modeling support to guide model- based test generation. In: MBT. (2012) 58–72
4. Brunelière, H., Cabot, J., Jouault, F.: Combining Model-Driven Engineering and Cloud Computing. In: Modeling, Design, and Analysis for the Service Cloud - MDA4ServiceCloud'10, Paris, France (June 2010)
5. Ultimate Software: Human Capital Management Solutions: Ultipro Enterprise (July 2013) `www.ultimatesoftware.com/solution`.
6. Jeff Brown: MbUnit Test Framework `http://mbunit.com/` (July 2013).
7. Stewart, S., Huggins, J.: Selenium - Web Browser Automation `http://docs.seleniumhq.org/` (July 2013).
8. Virtual Team: Echo Web UI Test Automation Framework. Technical report, Ultimate Software Group, Inc. (October 2010)
9. Wilk, J.: Page Object Pattern (March 2012) `http://blog.josephwilk.net/cucumber/page-object-pattern.html` (July 2013).
10. Microsoft: MSDN - Visual Studio: Extending the Editor (July 2013) `http://msdn.microsoft.com/en-us/library/dd885242.aspx`.
11. Kuhn, T., Gotzhein, R.: Model-driven platform-specific testing through configurable simulations. In: Model Driven Architecture - Foundations and Applications. Volume 5095 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 278–293
12. Palyart, M., Ober, I., Lugato, D., Bruel, J.M.: HPCML: a modeling language dedicated to high-performance scientific computing. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and Cloud computing. MDHPCL '12, New York, NY, USA, ACM (2012) 6:1–6:6
13. Nagel, B., Gerth, C., Yigitbas, E., Christ, F., Engels, G.: Model-driven specification of adaptive cloud-based systems. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and Cloud computing. MDHPCL '12, New York, NY, USA, ACM (2012) 4:1–4:6
14. Chelimsky, D., Myron Marston, M., Lindeman, A., Rowe, J.: RSpec - BDD framework for the Ruby Programming Language (December 2010) `http://rspec.info` (July 2013).
15. Hellesoy, A., Wynne, M.: The Cucumber Book: Behaviour-Driven Development for Testers and Developers. Pragmatic Programmers. Pragmatic Bookshelf (2012)
16. Nagy, G., Bandi, J., Hassa, C.: SpecFlow: Pragmatic BDD for .NET (November 2009) `http://www.specflow.org/specflownew/` (July 2013).