Research Article

A Proposal for Automatic Testing of GUIs Based on Annotated Use Cases

Pedro Luis Mateo Navarro,^{1,2} Diego Sevilla Ruiz,^{1,2} and Gregorio Martínez Pérez^{1,2}

¹ Departamento de Ingeniería de la Información y las Comunicaciones, University of Murcia, 30071 Murcia, Spain

² Departamento de Ingeniería y Tecnología de Computadores, University of Murcia, 30071 Murcia, Spain

Correspondence should be addressed to Pedro Luis Mateo Navarro, pedromateo@um.es

Received 16 June 2009; Accepted 14 August 2009

Academic Editor: Phillip Laplante

Copyright © 2010 Pedro Luis Mateo Navarro et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This paper presents a new approach to automatically generate GUI test cases and validation points from a set of annotated use cases. This technique helps to reduce the effort required in GUI modeling and test coverage analysis during the software testing process. The test case generation process described in this paper is initially guided by use cases describing the GUI behavior, recorded as a set of interactions with the GUI elements (e.g., widgets being clicked, data input, etc.). These use cases (modeled as a set of initial test cases) are annotated by the tester to indicate interesting variations in widget values (ranges, valid or invalid values) and validation rules with expected results. Once the use cases are annotated, this approach uses the new defined values and validation rules to automatically generate new test cases and validation points, easily expanding the test coverage. Also, the process allows narrowing the GUI model testing to precisely identify the set of GUI elements, interactions, and values the tester is interested in.

1. Introduction

It is well known that testing the correctness of a *Graphical User Interfaces* (GUI) is difficult for several reasons [1]. One of those reasons is that the space of possible interactions with a GUI is enormous, which leads to a large number of GUI states that have to be properly tested (a related problem is to determine the coverage of a set of test cases); the large number of possible GUI states results in a large number of input permutations that have to be considered. Another one is that validating the GUI state is not straightforward, since it is difficult to define which objects (and what properties of these objects) have to be verified.

This paper describes a new approach between Model-less and Model-Based Testing approaches. This new approach describes a GUI test case autogeneration process based on a set of use cases (which are used to describe the GUI behavior) and the annotation (definition of values, validation rules, etc.) of the relevant GUI elements. The process generates automatically all the possible test cases depending on the values defined during the annotation process and incorporates

new validation points, where the validation rules have been defined. Then, in a later execution and validation process, the test cases are automatically executed and all the validation rules are verified in order to check if they are met or not.

The rest of the paper is structured as follows. Related work is presented in Section 2. In Section 3 we describe the new testing approach. Annotation, autogeneration, and execution/validation processes are described in Sections 4, 5 and 6, respectively. Finally, Section 8 provides conclusions and lines of future work.

This paper is an extended version of the submitted contribution to the "Informatik 2009: Workshop MoTes09" [2].

2. Related Work

Model-Based GUI Testing approaches can be classified depending on the amount of GUI details that are included in the model. By GUI details we mean the elements which are chosen by the *Coverage Criteria* to faithfully represent the tested GUI (e.g., window properties, widget information and properties, GUI metadata, etc.).

Many approaches usually choose all window and widget properties in order to build a highly descriptive model of the GUI. For example, in [1] (Xie and Memon) and in [3, 4] (Memon et al.) it is described a process based on GUI Ripping, a method which traverses all the windows of the GUI and analyses all the events and elements that may appear to automatically build a model. That model is composed of a set of graphs which represent all the GUI elements (a tree called GUI Forest) all the GUI events and their interaction (Event-Flow Graphs (EFG), and Event Interaction Graphs (EIG)). At the end of the model building process, it has to be verified, fixed, and completed manually by the developers.

Once the model is built, the process explores automatically all the possible test cases. Of those, the developers select the set of test cases identified as meaningful, and the *Oracle Generator* creates the expected output (a *Test Oracle* [5] is a mechanism which generates outputs that a product should have for determining, after a comparison process, whether the product has passed or failed a test (e.g., a previous stored state that has to be met in future test executions). Test Oracles also may be based on a set of rules (related to the product) that have to be validated during test execution). Finally, test cases are automatically executed and their output compared with the Oracle expected results.

As said in [6], the primary problem with these approaches is that as the number of GUI elements increases, the number of event sequences grows exponentially. Another problem is that the model has to be verified, fixed, and completed manually by the testers, with this being a tedious and error-prone process itself. These problems lead to other problems, such a scalability and modifications tolerance. In these techniques, adding a new GUI element (e.g., a new widget or event) has two worrying side effects. First, it may cause the set of generated test cases to grow exponentially (all paths are explored); second, it forces a GUI Model update (and a manual verification and completion) and the regeneration of all affected test cases.

Other approaches use a more restrictive coverage criteria in order to focus the test case autogeneration efforts on only a section of the GUI which usually includes all the relevant elements to be tested. In [7] Vieira et al. describe a method in which enriched UML Diagrams (UML Use Cases and Activity Diagrams) are used to describe which functionalities should be tested and how to test them. The diagrams are enriched in two ways. First, the UML Activity Diagrams are refined to improve the accuracy; second, these diagrams are annotated by using custom UML Stereotypes representing additional test requirements. Once the model is built, an automated process generates test cases from these enriched UML diagrams. In [8] Paiva et al. also describe a UML Diagrams-based model. In this case, however, the model is translated to a formal specification.

The scalability of this approach is better than the previously mentioned because it focuses its efforts only on a section of the model. The diagram refinement also helps to reduce the number of generated test cases. On the other hand, some important limitations make this approach not so suitable for certain scenarios. The building, refining, and annotation processes require a considerable effort since they

have to be performed manually, which does not suit some methodologies such as, for instance, Extreme Programming; these techniques also have a low tolerance to modifications; finally, testers need to have a knowledge of the design of the tested application (or have the UML model), which makes impossible to test binary applications or applications with an unknown design.

3. Overview of the Annotated Use Case Guided Approach

In this paper we introduced a new GUI Testing approach between Mode-less and Model-Based testing. The new approach is based on a Test Case Autogeneration process that does not build a complete model of the GUI. Instead, it models two main elements that are the basis of the *test case autogeneration process*.

- (i) A Set of Use Cases. These use cases are used to describe the behavior of the GUI to be tested. The use cases are used as the base of the future test cases that are going to be generated automatically.
- (ii) A Set of Annotated Elements. This set includes the GUI elements whose values may vary and those with interesting properties to validate. The values define new variation points for the base use cases; the validation rules define new validation points for the widget properties.

With these elements, the approach addresses the needs of GUI verification, since, as stated in [7], the testing of a scenario can usually be accomplished in three steps: launching the GUI, performing several use cases in sequence, and exiting. The approach combines the benefits from both “Smoke Testing” [4, 9] and “Sanity Testing” [10], as it is able to assure that the system under test will not catastrophically fail and test the main functionality (in the first steps of the development process) and fine-tune checking and properties validation (in the final steps of the development process) by an automated script-based process.

The *test case generation process* described in this paper takes as its starting point the set of use cases (a use case is a sequence of events performed on the GUI; in other words, a use case is a test case) that describe the GUI behavior. From this set, it creates a new set of autogenerated test cases, taking into account the variation points (according to possible different values of widgets) and the validation rules included in the annotations. The resulting set includes all the new autogenerated test cases.

The *test case autogeneration process* can be seen, in a test case level, as the construction of a tree (which initially represents a test case composed of a sequence of test items) to which a new branch is added for each new value defined in the annotations. The validation rules are incorporated later as validation points.

Therefore, in our approach, modeling the GUI and the application behavior does not involve building a model including all the GUI elements and generating a potentially large amount of test cases exploring all the possible event

FIGURE 1: Schematic representation of the Widget Annotation Process.

sequences. In the contrary, it works by defining a set of test cases and annotating the most important GUI elements to include both interesting values (range of valid values, out-of-range values) and a set of validation rules (expected results and validation functions) in order to guide the *test case generation process*. It is also not necessary to manually verify, fix, or complete any model in this approach, which removes this tedious and error-prone process from the GUI Testing process and eases the work of the *testers*. These characteristics help to improve the scalability and the modifications tolerance of the approach.

Once the new set of test cases is generated, and the validation rules are incorporated, the process ends with the *test case execution process* (that includes the *validation process*). The result of the execution is a report including any relevant information to the tester (e.g., number of test performed, errors during the execution, values that caused these errors, etc). In the future, the generated test case set can be re-executed in order to perform a regression testing process that checks if the functionality that was previously working correctly is still working.

4. Annotation Process

The *annotation process* is the process by which the tester indicates what GUI elements are important in terms of the following: First, which values can a GUI element hold (i.e., a new set of values or a range), and thus should be tested; second, what constraints should be met by a GUI element at a given time (i.e., validation rules), and thus should be validated. The result of this process is a set of annotated GUI elements which will be helpful during the *test case autogeneration process* in order to identify the elements that represent a variation point, and the constraints that have to be met for a particular element or set of elements. From now on, this set will be called *Annotation Test Case*.

This process could be implemented, for example, using a capture and replay (C&R) tool(a *Capture and*

Replay Tool captures events from the tested application and use them to generate test cases that replay the actions performed by the user. Authors of this paper have worked on the design and implementation of such tool as part of a previous research work, accessible on-line at <http://sourceforge.net/projects/openhmitester/> and at <http://www.um.es/catedraSAES/> . These tools provide the developers with access to the widgets information (and also with the ability to store it), so they could use this information along with the new values and the validation rules (provided by the tester in the *annotation process*) to build the *Annotation Test Case*.

As we can see in Figure 1, the *annotation process*, which starts with the tested application launched and its GUI ready for use, can be performed as follows:

- (1) For each widget the tester interacts with (e.g., to perform a click action on a widget or enter some data by using the keyboard), he or she can choose between two options: annotate the widget (go to the next step) or continue as usual (go to step 3).
- (2) A widget can be annotated in two ways, depending on the chosen Test Oracle method. It might be an “Assert Oracle” (checks a set of validation rules related to the widget state), or a “State Oracle” (checks if the state of the widget during the *execution process* matches the state stored during the *annotation process*).
- (3) The annotations (if the tester has decided to annotate the widget) are recorded by the C&R tool as part of the *Annotation Test Case*. The GUI performs the actions triggered by the user interaction as usual.
- (4) The GUI is now ready to continue. The tester can continue interacting with the widgets to annotate them or just finish the process.

The annotated widgets should be chosen carefully as too many annotated widgets in a test case may result in an explosion of test cases. Choosing an accurate value set also

helps to get a reasonable test suite size, since during the *test case autogeneration process*, all the possible combinations of annotated widgets and defined values are explored in order to generate a complete test suite which explores all the paths that can be tested. So, these are two important aspects to consider, since the scalability of the generated test suite depends directly on the amount of annotated widgets and the values set defined for them.

Regarding to the definition of the validation rules that are going to be considered in a future *validation process*, the tester has to select the type of the test oracle depending on his or her needs.

For the *annotation process* of this approach we consider two different test oracles.

- (i) *Assert Oracles*. These oracles are useful in two ways. First, if the tester defines a new set of values or a range, new test cases will be generated to test these values in the *test case autogeneration process*; second, if the tester also defines a set of validation rules, these rules will be validated during the *execution and validation process*.
- (ii) *State Oracles*. These oracles are useful when the tester has to check if a certain widget property or value remains constant during the *execution and validation process* (e.g., a widget that can not be disabled).

In order to define the new values set and the validation rules, it is necessary to incorporate to the process a specification language which allows the tester to indicate which are going to be the new values to be tested and what constraints have to be met. This specification language might be a constraint language as, for instance, the Object Constraint Language (OCL) [11], or a script language as, for instance, Ruby [12]. This kind of languages can be used to allow the tester to identify the annotated object and specify new values and validation rules for it. It is also necessary to establish a mapping between widgets and constructs of the specification language; both languages have mechanisms to implement this feature.

Validation rules also can be set to specify if the tester wants the rules to be validated before (precondition) or after (postcondition) an action is performed on the annotated widget. For example, if the tester is annotating a button (during the *annotation process*), it might be interesting to check some values before the button is pressed, as that button operates with those values; it also might be interesting to check, after that button is pressed, if the obtained result met some constraints. The possibility to decide if the validation rules are going to be checked before of after an action is performed (these are the well-known preconditions and postconditions) allows the tester to perform a more powerful *validation process*. This process could be completed with the definition of an invariant, for example, together with the state oracles, since the invariant is composed of a set of constraints that have to be met through the process (an invariant in this domain would be a condition that is always met in the context of the current dialog.).

FIGURE 2: Schematic representation of the Test-Case Auto Generation Process.

5. Test Case AutoGeneration Process

The *test case autogeneration process* is the process that automatically generates a new set of test cases from two elements:

- (i) a test suite composed of an initial set of test cases (those corresponding to the use cases that represent the behavior of the GUI);
- (ii) an special test case called *Annotation Test Case* which contains all the annotations corresponding to the widgets of a GUI.

As can be seen in Figure 2, the process follows these steps:

- (1) As said above, the process is based on an initial test suite and an *Annotation Test Case*. Both together make up the initial *Annotated Test Suite*.
- (2) The *test case autogeneration process* explores all the base use cases. For each use case, it generates all the possible variations depending on the values previously defined in the annotations. It also adds validators for ensuring that the defined rules are met. (This process is properly explained at the end of this section).
- (3) The result is a new *Annotated Test Suite* which includes all the auto-generated test cases (one for each possible combination of values) and the *Annotation Test Case* used to generate them.

The set of auto-generated test cases can be updated, for example, if the tester has to add or remove new use cases due to a critical modification in the GUI, or if new values or validation rules have to be added or removed. The tester will then update the initial test case set, the *Annotation Test Case*, or both, and will rerun the generation process.

The algorithm corresponding to the *test case autogeneration process* is shown in Algorithm 1.

The process will take as its starting point the *Annotation Test Case* and the initial set of test cases, from which it will generate new test cases taking into account the variation points (the new values) and the validation rules included in the annotations.

For each test case in the initial set, the process inspects every test item (a test case is composed of a set of steps called test items) in order to detect if the widget referred by this test item is included in the annotated widget list. If so, the process generates all the possible variations of the test case (one for each different value, if exist), adding also a validation point if some validation rules have been defined. Once the process has generated all the variations of a test case, it adds them to the result set. Finally, the process returns a set of test cases which includes all the variations of the initial test cases.

Figure 3 is a graphical representation of how the algorithm works. The figure shows an initial test case which includes two annotated test items (an *Annotated Test Item* is a test item that includes a reference to an annotated widget). The annotation for the first widget specifies only two different values (15 and 25); the annotation for the second one specifies two new values (1 and 2) and introduces two validation rules (one related to the *colour* property of the widget and another related to the *text* property). The result of the *test case autogeneration process* will be four new test cases, one for each possible path (15-1, 15-2, 25-1, and 25-2), and a validation point in the second annotated test item which will check if the validation rules mentioned before are met or not.

6. Execution and Validation Process

The *execution and validation process* is the process by which the test cases (auto-generated in the last step) are executed over the target GUI and the validation rules are asserted to check whether the constraints are met. The *test case execution process* executes all the test cases in order. It is very important that for each test case is going to be executed, the GUI must be reset to its initial state in order to ensure that all the test cases are launched and executed under the same conditions.

This feature allows the tester to implement different test configurations, ranging from a set of a few test cases (e.g., to test a component, a single panel, a use case, etc.), to an extensive battery of tests (e.g., for a nightly or regression testing process [4]).

As for the *validation process*, in this paper we describe a Test Oracle based validation process, which uses test oracles [1, 5] to perform widget-level validations (since the validation rules refer to the widget properties) (A *Test Oracle* is a mechanism that generates the expected output that a product should have for determining, after a comparison process, whether the product has passed or failed a test). The features of the *validation process* vary depending on the oracle method selected during the *annotation process* as we can read below.

- (i) *Assert Oracles*. These oracles check if a set of validation rules related to a widget are met or not. Therefore, the tester needs to somehow define a set of validation rules. As said in Section 4 corresponding to the *annotation process*, defining these rules is not straightforward. Expressive and flexible (e.g., constraint or script) languages are needed to allow the tester to define assert rules for the properties of the annotated widget, and, possibly, to other widgets. Another important pitfall is that if the GUI encounters an error, it may reach an unexpected or inconsistent state. Further executing the test case is useless; therefore it is necessary to some mechanism to detect these “bad states” and stop the test case execution (e.g., a special statement which indicates that the *execution and validation process* have to finish if an error is detected).
- (ii) *State Oracles*. These oracles check if the state of the widget during the *execution process* matches the state stored during the *annotation process*. To implement this functionality, the system needs to know how to extract the state from the widgets, represent it somehow, and be able to check it for validity. In our approach, it could be implemented using widget adapters which, for example, could represent the state of a widget as a string; so, the validation would be as simple as a string comparison.

The *validation process* may be additionally completed with *Crash Oracles*, which perform an application-level validation (as opposed to widget-level) as they can detect crashes during test case execution. These oracles are used to signal and identify serious problems in the software; they are very useful in the first steps of the development process.

Finally, it is important to remember that there are two important limitations when using test oracles in GUI testing [5]. First, GUI events have to be deterministic in order to be able to predict their outcome (e.g., it would not make sense if the process is validating a property which depends on a random value); second, since the software back-end is not modeled (e.g., data in a data base), the GUI may return a nonexpected state which would be detected as an error (e.g., if the process is validating the output in a database query application, and the content of this database changes during the process).

7. Example

In order to show this process working on a real example, we have chosen a *fixed-term deposit calculator* application. This example application has a GUI (see Figure 4) composed of a set of widgets: a menu bar, three number boxes (two integer and one double), two buttons (one to validate the values and another to operate with them), and a label to output the obtained result. Obviously, there are other widgets in that GUI (i.e., a background panel, text labels, a main window, etc.), but these elements are not of interest for the example.

FIGURE 3: Test case branching.

```

initial_test_set ← ... // initial test case set
auto_gen_test_set ← {} // auto-generated test case set (empty)
annotated_elements ← ... // user-provided annotations

for all TestCase tc ∈ initial_test_set do
  new_test_cases ← add_test_case (new_test_cases, tc)
  for all TestItem ti ∈ tc do
    if ti.widget ∈ annotated_elements then
      annotations ← annotations_for_widget(ti.widget)
      new_test_cases ← create_new_test_cases (new_test_cases, annotations.values)
      new_test_cases ← add_validation_rules (new_test_cases, annotations.rules)
    end if
  end for
  auto_gen_test_set ← auto_gen_test_set ∪ new_test_cases
end for
return auto_gen_test_set

```

ALGORITHM 1: Test case autogeneration algorithm.

FIGURE 4: Example dialog.

A common use case for this application is the following:

- (1) start the application (the GUI is ready),
- (2) insert the values in the three number boxes,
- (3) if so, click the “Calc Interest” button and see the result,
- (4) exit by clicking the “Exit” option in the “File” menu.

The valid values for the number boxes are the following.

- (i) *Interest Rate*. Assume that the interest rate imposed by the bank is between 2 and 3 percent (both included).
- (ii) *Deposit Amount*. Assume that the initial deposit amount has to be greater or equal to 1000, and no more than 10 000.
- (iii) *Duration*. Assume that the duration in months has to be greater or equal to 3, and less than or equal to 12 months.

The behavior of the buttons is the following. If a number box is out of range, the “Calc Interest” button changes its background colour to red (otherwise, it has to stay white); once it is pressed, it calculates the result using the values, and writes it in the corresponding label. If the values are out of range, the label must read “Data error.” In other case, the actual interest amount must be shown.

FIGURE 5: Auto-generated test case representation for the example dialog.

Therefore, the annotations for widgets are as follows.

- (i) “Interest rate” *spinbox*: a set of values from 2 to 3 with a 0.1 increase.
- (ii) “Deposit amount” *spinbox*: a set of values composed of the three values 500, 1000, and 8000. (Note that the value of 500 will introduce a validation error in the test cases.)
- (iii) “Duration” *spinbox*: a set of three values, 6, 12, and 24. Again, the last value will not validate.
- (iv) “Calc Interest” *button*: depending on the values of the three mentioned text boxes, check the following.
 - (1) If the values are within the appropriate ranges, the background color of this button must be white, and as a postcondition, the value of the label must hold the calculated interest value (a formula may be supplied to actually verify the value).
 - (2) Else, if the values are out of range, the background color of the button must be red, and as a postcondition, the value of the label must be “Data error.”

Once the initial use case is recorded and the widgets are properly annotated (as said, both processes might be performed with a capture/replay tool), they are used to compose the initial *Annotated Test Suite*, which will be the basis for the *test case autogeneration process*.

We can see the *test case autogeneration process* result in Figure 5. The new *Annotated Test Suite* generated by the process is composed of 99 test cases (11 values for the “Interest rate,” 3 different “Deposit amounts,” and 3 different “Durations”) and a validation point located at the “Calc Interest” button clicking (to check if the values are valid and the background colour accordingly).

The process automatically generates one test case for each possible path by taking into account all the values defined in the annotation process; it also adds validation points where the validation rules have been defined. The new set of auto-generated test cases allows the *tester* to test all the possible variations of the application use cases.

Finally, the *execution and validation process* will execute all the test cases included in the generated *Annotated Test Suite* and will return a report including all the information related to the *execution and validation process*, showing the number of test cases executed, the time spent, and the values not equal to those expected.

8. Conclusions and Future Work

Automated GUI test case generation is an extremely resource intensive process as it is usually guided by a complex and fairly difficult to build GUI model. In this context, this paper presents a new approach for automatically generating GUI test cases based on both GUI use cases (required functionality), and annotations of possible and interesting variations of graphical elements (which generate families of test cases), as well as validation rules for their possible values. This reduces the effort required in test coverage and GUI modeling processes. Thus, this method would help reducing the time needed to develop a software product since the testing and validation processes spend less efforts.

As a statement of direction, we are currently working on an architecture and the details of an open-source implementation which allow us to implement these ideas and future challenges as, for example, to extend the GUI testing process towards the application logic, or to execute a battery of tests in parallel in a distributed environment.

Acknowledgments

This paper has been partially funded by the “Cátedra SAES of the University of Murcia” initiative, a joint effort between Sociedad Anónima de Electrónica Submarina (SAES), <http://www.electronica-submarina.com/> and the University of Murcia to work on open-source software, and real-time and critical information systems.

References

- [1] Q. Xie and A. M. Memon, “Model-based testing of community-driven open-source GUI applications,” in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pp. 203–212, Los Alamitos, Calif, USA, 2006.
- [2] P. Mateo, D. Sevilla, and G. Martínez, “Automated GUI testing validation guided by annotated use cases,” in *Proceedings of the 4th Workshop on Model-Based Testing (MoTes '09) in Conjunction with the Annual National Conference of German Association for Informatics (GI '09)*, Lübeck, Germany, September 2009.
- [3] A. Memon, I. Banerjee, and A. Nagarajan, “GUI ripping: reverse engineering of graphical user interfaces for testing,” in *Proceedings of the 10th IEEE Working Conference on Reverse Engineering (WCRE '03)*, pp. 260–269, Victoria, Canada, November 2003.
- [4] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan, “Dart: a framework for regression testing “nightly/daily builds” of GUI applications,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '03)*, pp. 410–419, 2003.

- [5] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI based software applications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, p. 4, 2007.
- [6] X. Yuan and A. M. Memon, "Using GUI run-time state as feedback to generate test cases," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, Minneapolis, Minn, USA, May 2007.
- [7] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier, "Automation of GUI testing using a model-driven approach," in *Proceedings of the International Workshop on Automation of Software Test*, pp. 9–14, Shanghai, China, 2006.
- [8] A. Paiva, J. Faria, and R. Vidal, "Towards the integration of visual and formal models for GUI testing," *Electronic Notes in Theoretical Computer Science*, vol. 190, pp. 99–111, 2007.
- [9] A. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884–896, 2005.
- [10] R. S. Zybin, V. V. Kuliainin, A. V. Ponomarenko, V. V. Rubanov, and E. S. Chernov, "Automation of broad sanity test generation," *Programming and Computer Software*, vol. 34, no. 6, pp. 351–363, 2008.
- [11] Object Management Group, "Object constraint language (OCL)," version 2.0, OMG document formal/2006-05-01, 2006, <http://www.omg.org/spec/OCL/2.0/>.
- [12] Y. Matsumoto, "Ruby Scripting Language," 2009, <http://www.ruby-lang.org/en/>.