Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

A Test Automation Language for Behavioral Models

Nan Li nli1@gmu.edu
Jeff Offutt offutt@gmu.edu

Technical Report GMU-CS-TR-2013-7

Abstract

Model-based testers design tests in terms of models, such as paths in graphs. Abstract tests cannot be run directly because they use names and events that exist in the model, but not the implementation. Testers usually solve this mapping problem by hand. Model elements often appear in many abstract tests, so testers write the same redundant code many times. This is time-consuming, labor-intensive, and error-prone.

This paper presents a language to automate the creation of mappings from abstract tests to concrete tests. Three issues are addressed: (1) creating mappings and generating test values, (2) transforming graphs and using coverage criteria to generate test paths, and (3) solving constraints and generating concrete test.

The paper also presents results from an empirical comparison of testers using the mapping language with manual mapping on 17 open source and example programs. We found that the automated test generation method took a fraction of the time the manual method took, and the manual tests contained numerous errors in which concrete tests did not match their abstract tests.

1 Introduction

In *model-based testing* (*MBT*), a model is a partial abstract description of a program that usually reflects functional aspects of the system. For example, a finite state machine (FSM) represents the behavior of a system. This research focuses on dynamic models such as the Unified Modeling Language (UML) behavioral diagrams. Tests expressed in terms of a model are called *abstract tests*. An abstract test is defined using elements and objects from the model, thus cannot be executed on an implementation. For example, if the model is a finite state machine, an abstract test might be a path through that machine. *Concrete tests* are expressed in terms of the implementation of the model, and are ready to be run automatically. Thus, a *JUnit* test is concrete. Model-based abstract tests must be transformed to concrete tests since abstract tests cannot be applied directly to the actual program. The *mapping problem* refers to the problem of converting abstract tests to concrete tests [1, 2].

Testers currently map abstract tests to concrete tests by hand. If an abstract test consists of several *events*, or changes in the state of the model, testers have to write the code for each event by hand. If one basic event is used multiple times in different abstract tests, testers write redundant code each time the same event is used. This process is time-consuming, labor-intensive, and error-prone. For example, to write a concrete test for the abstract test event *authentication for an account*, testers may have to set up the test environment, including making a database connection and creating an account, and then write test sequences and an oracle. If this event is used multiple times in abstract tests, testers have to repeat the same process for all concrete tests.

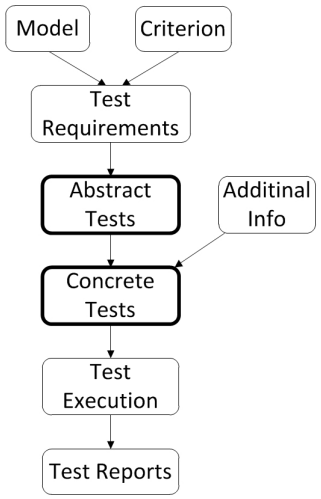

Figure 1: A Generic Model-based Testing Process

Figure 1 shows a general process to derive tests from models. Testers choose a coverage criterion to generate test requirements based on a model. Then abstract tests are generated to satisfy the test requirements. Additional information, including abstract test values and data mappings, is needed to convert abstract tests to concrete tests. Expected results (the *oracles*) that are usually specified in models are used to compare with actual results.

The steps in bold rectangles are being addressed in this research to automate the mapping problem. The mapping problem results in testers writing redundant code, which leads to errors and lost time. Thus, a major goal of this research is to develop techniques to avoid this redundancy. Our approach is that for each basic identifiable element in a model, if we write the executable code once, the code can be inserted automatically the next time the same element appears in another abstract test. An identifiable element in a model can be used multiple times in abstract tests and can be mapped to one or more lines of code in concrete tests, for example, a transition in an FSM.

Our previous paper [3] proposed using a test automation language to solve the mapping problem. This paper follows that research with a language and detailed solution to the mapping problem.

This paper presents a test automation language, the Structured Test Automation Language (STAL), to automate the generation of concrete tests from abstract tests. Testers use STAL to create mappings from each basic identifiable element of the model to test code. Once the mappings are generated, testers do not need to write concrete tests line by line for each abstract test. Instead, a test automation framework, the Structured Test Automation Language framEwork (STALE) [4], generates concrete tests automatically from the abstract tests. STALE can improve the efficiency of generating concrete tests from abstract tests and reduce the number of errors.

STALE is built based on the *Eclipse Modeling Framework (EMF)* [5], which can read *EMF*-based UML diagrams. STALE supports UML state machine diagrams, and work on supporting other diagrams is ongoing.

Below is a summary of our MBT solution from a perspective of a taxonomy of model-based testing [6]. UML state machine diagrams used in our approach are input-output models because constraints in UML state machine diagrams are used as test oracles to verify the correctness of behaviors. Nondeterministic UML state machine diagrams are considered and the diagrams do not involve timing issues. UML state machine diagrams use transition-based notations. Structural model coverage criteria such as edge, edge-pair, and prime path coverage [7] are used to generate tests. The tester supplies a collection of possible values and constraint solving is used to choose values to execute each test path. Initial values are chosen for each transition, then constraints (for example, state invariants) are evaluated. If the constraints are not satisfied, then other values are chosen. If a constraint cannot be satisfied with the values supplied, the tester is informed and given the opportunity to modify the model, the program, or the mapping by adding additional values.

This paper addresses three key issues for STALE: (1) creating mappings and generating test values, (2) graph transformation and test path generation using coverage criteria, and (3) solving constraints and concrete test generation.

This paper presents the Structured Test Automation Language (STAL), which is the foundation for the Structured Test Automation Language framEwork (STALE). STAL is defined and illustrated through a running example. Most companies currently solve the mapping problem by hand, so we present a comparison of the use of STAL with the manual approach by having programmers and testers apply both on 17 UML state machine diagrams. The results show that the automatic test generation uses only 29.6 percent of the time for the manual test generation. Additionally, the manual approach resulted in 48 errors in 240 tests in which the executable code did not match the abstract tests.

The paper is organized as follows. Section 2 gives a motivating example to illustrate the mapping problem and why test automation is needed. Section 3 introduces background and related work about model-based testing. Section 4 presents the key issues that had to be addressed when building STALE. Section 5 presents the tool, experimental design, subjects, procedure, results, analysis and threats to validity. The paper is summarized in section 6, and section 7 describes future work.

2 A Motivating Example

A simple example is shown in figures 2, 3, and 4 to clarify the mapping problem. The program simulates the behavior of a vending machine for chocolate. Some assumptions are made to simplify the program: only chocolates are available for sale; the price for all chocolates is 90 cents; only dimes, quarters, and dollars are accepted; and the vending machine can contain an infinite number of chocolates. Figure 2 shows the class specifications.

```
public class vendingMachine
{
    private int credit; // Current credit in the machine.
    private LinkedList stock; // Used to store all chocolates.

    // Constructor: vending machine starts empty.
    public vendingMachine() {}

    // A coin is given to the vendingMachine.
    // Must be a dime, quarter or dollar.
    public void coin (int coin) {}

    // User asks for a chocolate. Returns change
    // and sets the parameter StringBuffer variable Choc.
    public int getChoc (StringBuffer choc) {}

    // Add one new piece of chocolate to the machine.
    public void addChoc (String choc) {}

    // Get the current credit value.
    public int getCredit () {}

    // Get all chocolates in stock.
    public LinkedList getStock () {}
}
```

Figure 2: Class of vendingMachine

Figure 3 shows a simplified finite state machine (FSM) for *vendingMachine*. It has nine transitions, four states, and three events, *AddChoc, Coin*, and *GetChoc*. State 1 is the initial

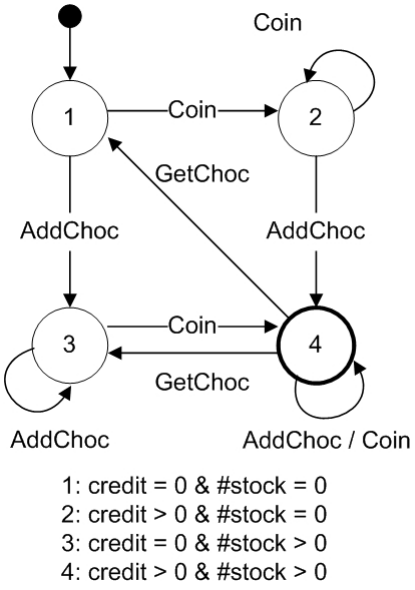

Figure 3: Simplified FSM for Vending Machine

state, where the credit is 0 and the number of chocolates is 0. If a customer adds coins, the FSM transitions to state 2, where the credit is greater than 0 but the number of chocolates is still 0. If a service person adds chocolates, the FSM transitions to state 4, where the credit is greater than 0 and the number of chocolates is greater than 0. State 4 returns to itself if coins or chocolates are added to the vending machine, and state 4 can transition to state 1 or 3 if chocolates are taken from the machine. Similarly, when adding chocolates, state 1 can transition to state 3, where the credit is 0 and the number of the chocolates is greater than 0; state 3 returns to itself when adding chocolates; and state 3 transitions to state 4 when customers insert coins.

Testers may generate tests from the FSM by applying coverage criteria. If a tester uses the prime path coverage criterion [7], nine test paths are generated to cover 14 prime paths by a graph coverage web application [8]:

1. [1, 3, 4, 1, 2, 4]
2. [1, 2, 4, 1, 2, 4]
3. [1, 2, 4, 3, 4]
4. [1, 2, 4, 1, 3, 4]
5. [1, 3, 4, 1, 3, 4]
6. [1, 3, 4, 3, 4]
7. [1, 2, 2, 4]
8. [1, 3, 3, 4]
9. [1, 2, 4, 4]

Since each transition has only one event, we use the event names to represent the transitions. The transitions for path 1 are *AddChoc*, *Coin*, *GetChoc*, *Coin*, and *AddChoc*, which defines an abstract test. The tester can translate this abstract test to the *JUnit* concrete test in figure 4.

In figure 4, the code in testFirstTestPath() is written to map the abstract test [*AddChoc*, *Coin, GetChoc, Coin*, *AddChoc*]. To be specific, *AddChoc* maps to line 1; *Coin* maps to line

```
public class vendingMachineTest
{
    private vendingMachine vm;
    @Before
    public void setUp() throws Exception {
       vm= new vendingMachine();
    }
    @After
    public void tearDown() throws Exception {
       vm= null;
    }
    @Test
    public void testFirstTestPath() {
     1: vm.addChoc ("MM");
     2: assertEquals (1, vm.getStock().size());
     3: vm.coin (100);
     4: assertEquals (95, vm.getCredit());
     5: StringBuffer choc = new StringBuffer().append ("MM");
     6: vm.getChoc (choc);
     7: assertEquals (0, vm.getStock().size());
     8: vm.coin (10);
     9: vm.coin (25);
    10: vm.coin (25);
    11: vm.coin (25);
    12: vm.coin (25);
    13: vm.addChoc ("MM");
    14: vm.addChoc ("MM");
    15: assertEquals (100, vm.getCredit());
    16: assertEquals (2, vm.getStock().size());
    }
}
```

Figure 4: A JUnit test for class vendingMachine

3; *GetChoc* maps to lines 5-6; the second *Coin* maps to lines 8-12; and the second *AddChoc* maps to lines 13-14. (The tool includes comments in the JUnit tests to document which abstract test is implemented, but these are omitted in this example to save space.)

For the second test path [1, 2, 4, 1, 2, 4], the transitions are *Coin, AddChoc, GetChoc, Coin*, and *AddChoc*. The concrete test code that corresponds to *Coin* and *AddChoc* should be very similar to the code from lines 8-14 in testFirstTestPath() above, and the concrete test code that corresponds to *GetChoc* may be the same as the code from lines 5-6. In total, the nine abstract tests have 15 *AddChoc*s, 16 *Coin*s, and 6 *getChoc*s. Each transition corresponds to at least one method call. So for transitions *AddChoc, Coin*, and *GetChoc*, the same or similar code will be written 15 times, 16 times, and 6 times. If the mappings from transitions to concrete code are automated, testers can avoid writing redundant code.

3 Background and Related Work

We view model-based testing as considering three major issues: (1) how to build test models, (2) how to use test criteria and algorithms to generate abstract tests from models, and (3) how to transform abstract tests to concrete tests. The second and third issues largely depend on what models are chosen.

Model abstractions need to be specified because they determine what artifacts will be used to generate tests. Prenninger and Pretschner [9] said that a system can be abstracted for six purposes: (1) getting enriched knowledge of the system and its environment, (2) obtaining the specification of the system, (3) accessing parts of a system, (4) communicating between developers, (5) generating code, and (6) testing systems.

While creating test models, testers can either write *model program*s in a specific language or draw visualization diagrams. Model programs use specification languages such as Spec# or programming languages such as Java or C# to describe model behaviors. ModelJUnit [10], Spec Explorer [11], NModel [12], and Conformiq [13] use Java, Spec#, C#, and Conformiq Modeling Language (QML) [13] to write model programs that can be converted to finite state machines or extended finite state machines. So model programs are created only for the purpose (6) above. Model programs cannot reuse diagrams from the design phase directly.

Many testers prefer to use visual diagrams such as finite state machines, extended finite state machines, and UML behavioral diagrams directly as test models. They can reuse and adapt design models or build new test models [14]. Researchers have come up with different ways to transform models created for (1) and (2) to test models for (6) [15, 16, 17, 18, 19, 20, 21, 22]. Many [15, 16, 17, 18, 20, 21, 22] use UML models to generate tests. This paper applies the same approach.

Models only specify key aspects of software's behavior, so cannot provide enough information for generating tests. When deriving test cases from UML behavioral models, additional information needs to be specified, such as test values and test oracles. One common solution is to use other UML models (for example, use case diagrams and class diagrams) to provide the missing information.

Briand and Labiche [17] developed the TOTEM system, which uses many artifacts: use case diagrams, use case descriptions, sequence or collaboration diagrams for each use case, class diagrams, and a data dictionary specifying the details of classes. Nebut et al. [18] applied a use case driven approach. This approach extracts additional information from use case models and requires a behavioral model (sequence, state machine, or activity diagram) to specify the sequence ordering of the use cases. Moreover, use cases must have contracts (pre- and post-conditions) to help infer the partial ordering of functionalities. Furthermore, the behavioral models have to be consistent with the use cases. That is, the parameters in use cases have to be the same as those in the behavioral models. This use case driven approach was validated on one embedded system.

The UML Testing Profile (UTP) [23] reuses some concepts of the UML but adds components for testing such as test context, test case, test component, and verdicts. When creating concrete tests, information in abstract tests have to match attributes of other diagrams such as class and object diagrams.

When creating tests from UML state machine diagrams, existing approaches use additional diagrams and structures to provide abstract test values and transform them to concrete test values. The abstract test values have to be consistent with those defined in diagrams such as class diagrams or object diagrams. Thus, abstract tests not only have test sequences, but also lots of abstract test values, which complicates the transformation from abstract to concrete tests. Creating many formalized diagrams for testing may not be difficult for companies and organizations that can reuse models from the design phase and have lots of experts in model-based testing. However, it is very expensive for organizations that do not have such resources.

This paper presents the test automation language STAL to generate tests from UML behavior models, specifically UML state machine diagrams. Programmers and testers use STAL to provide missing information by creating mappings from identifiable elements of the model to concrete test code. Previous papers included few empirical studies in comparing concrete MBT solutions with manual approaches. This paper includes an empirical comparison of transforming abstract tests to concrete tests by using STAL with the manual method.

STALE can read diagrams from the *Eclipse Modeling Framework (EMF)*. EMF is a modeling framework based on the Eclipse platform. The core of EMF provides tools and APIs to view and edit the models that are described in the *XML Metadata Interchange (XMI)* framework [24]. *EMF* also supports other *EMF*-based applications. Different kinds of coverage criteria such as structural modeling, data flow, random, and stochastic coverage have been used to generate tests [6]. Our tool uses the node, edge, edge-pair and prime path coverage criteria [7].

We started by trying to use the existing model-to-test transformation language *Meta-Object Facility Model To Text Transformation Language (MOFM2T)* [25]. Unfortunately, this language has several characteristics that made it impossible to use for our research.

MOFM2T [25] is part of OMG's *Model-driven architecture (MDA)* [26] and was designed to transform models to code for general use. *Acceleo* [27] is the only Eclipse Foundation project that implements MOFM2T. It reads *EMF*-based models and transforms them into programs in several languages.

Adapting *MOFM2T* and *Acceleo* to define mappings from abstract to concrete tests posed two problems. First, STALE cannot reuse much of the syntax of *MOFM2T*. For example, the *for* loop structure used in *MOFM2T* goes through each component of the same type (e.g. states) in a model and translates them to similar code. However, the *for* loop cannot be used for the mappings because each identifiable element in a model is mapped to different test code.

Second, testers cannot write test code to create mappings with *MOFM2T*. Ideally, testers first choose an identifiable element, write down its name, write the code for it, then create the mapping. However, *MOFM2T* and *Acceleo* cannot recognize the element names. Testers would need to write code to look for the identifiable element from the top level to the bottom level of the model structure.

3.1 Background in Graph Coverage Criteria

STALE is generic enough to be used with any test design strategy, whether criteria-based or human-based. This paper describes STAL and uses examples based on graph coverage criteria. UML state machine diagrams are transformed into generic graphs, and then graph coverage criteria are used.

The following definitions are taken from Ammann and Offutt [7]. A graph G defined formally as

- a set N of *nodes*, where $N \neq \emptyset$
- a set N_0 of *initial nodes*, where $N_0 \subseteq N$ and $N_0 \neq \emptyset$
- a set N_f of *final nodes*, where $N_f \subseteq N$ and $N_f \neq \emptyset$
- a set E of *edges*, where E is a subset of $N \times N$

A graph **must have** at least one initial and one final node, but **allows** more. For graphs, coverage criteria define the set of test requirements in terms of properties of test paths in a graph G. Test requirements are *satisfied* by *visiting* specific nodes or edges or by *touring* specific paths or subpaths.

A *path* is a sequence $[n_1, n_2, ..., n_M]$ of nodes, where each pair of adjacent nodes, (n_i, n_{i+1}), $1 \leq i < M$, is in the set E of edges. The *length* of a path is defined as the number of nodes it contains. A *subpath* of a path p is a subsequence of p (including p itself). A *test path* represents the execution of a test case on a graph. Test paths must start at an initial node and end at a final node. A test path p *tours* a subpath q if q is a subpath of p. **Edge** coverage requires that each edge is covered by test paths. That is, each transition on a UML state machine diagram should be toured.

4 The Test Automation Language STAL

This section presents our language for automating the generation of concrete tests from abstract tests. The vending machine example is used to illustrate how testers use STAL to create mappings. This vending machine example differs slightly from the example in section 2 by allowing only 10 chocolates. Figure 5 is a UML state machine diagram for this vending machine example created with the *EMF*-based tool *Papyrus* [28]. Figure 5 has one initial state, one final state, nine normal states, and 26 transitions. It also includes six constraints that are used as state invariants for states 1-9[1]. Some states have internal transitions. For example, state 2 has an internal transition on *coin*.

Three key issues have been addressed in this research: (1) creating mappings and generating test values, (2) graph transformation and test path generation using coverage criteria, and (3) solving constraints and concrete test generation. How to include test oracles, and which part of the program state to evaluate, will be addressed in the future.

[1]Constraints can be specified to be guards or state invariants.

	Element
Need mappings	Entry Point, Exit Point, and Do Activity of a State, Transition, Constraint
Do not need mappings	State Machine, Region, Initial PseudoState, Final State, Fork, Join, Junction, Choice, Simple State, Composite State, Submachine State
Not used in test models	Shallow History PseudoState, Deep History PseudoState

Table 1: Which elements need mappings?

4.1 Creating Mappings and Generating Test Values

This subsection describes how to create mappings from identifiable elements to executable *Java* code. Some elements of UML state machine diagrams need test code to map them, some do not need mappings, and others should not be used in test models. Table 1 summarizes which elements of a UML state machine diagram should be mapped.

STAL defines two kinds of mappings: *element mappings* and *object mappings*. Element mappings directly connect an identifiable element in a UML state machine diagram to test code. For instance, a transition *coin* may be mapped to the test code "*vm.coin (c);*". However, objects and parameters used in this element mapping, such as *vm* (an object of class VendingMachine) and *c* (an int parameter of method *coin (int)*), also need to be initialized in object mappings, which will be marked as required mappings of this element mapping.

An element mapping is formally defined as:

Mapping *mappingName* ***TypeOfElement*** *nameOfElement*
Requires *objectMappingName ...*
*[**StateInvariants** nameOfState ...] [**Guards** nameOfTransition ...]*
{*testCode*}

Mapping, **TypeOfElement**, **Requires**, **StateInvariants**, **Guards** are keywords. The mapping name must be unique. **TypeOfElement** may vary depending on the actual type of one concrete element such as a transition or constraint. If a mapping uses an object defined in another mapping, the names of additional mappings have to be included in the **Requires** field. The notation "..." means that more than one mapping may be required. If an element is a constraint, the mapping needs to point out the type of constraint (state invariant, guard, etc.) and elements (e.g. states or transitions) in which the constraint is held. StateInvariants and Guards fields are optional and marked by "[ ]" since they are used only for constraints. A constraint may be used as a state invariant in states and guards at the same time. Test code is required for any mapping and written in curly brackets.

An object mapping is formally defined as:

Mapping *mappingName* ***Class*** *nameOfClass*
Object *nameOfObject* ***Requires*** *objectMappingName ...*
{*testCode*}

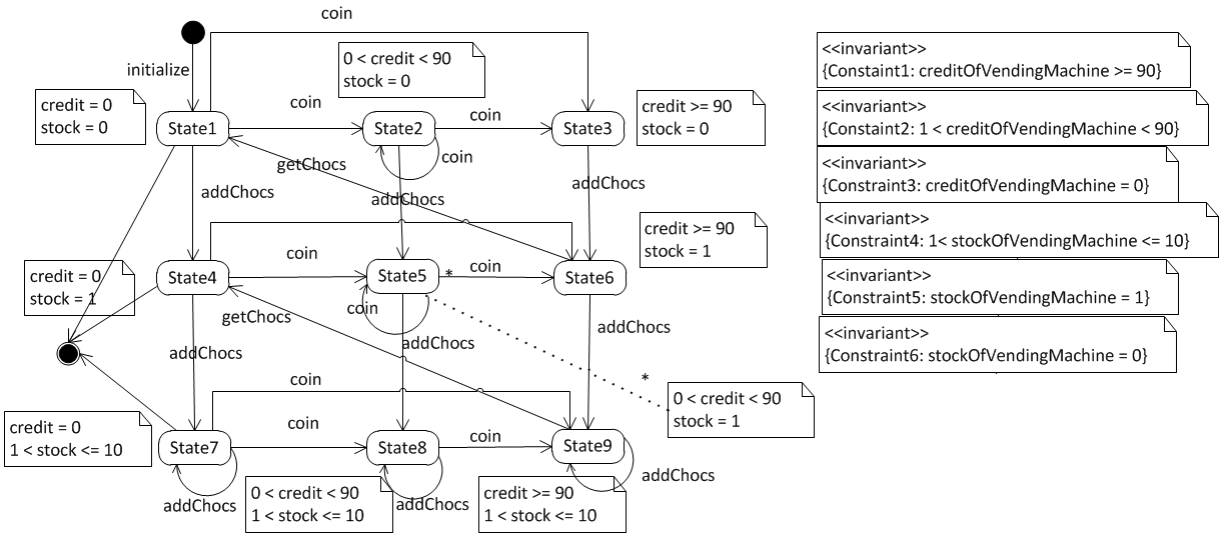

Figure 5: A UML state machine diagram for the class vendingMachine

Attributes	Element Mapping	Object Mapping
Element Name	X	
Element Type	X	
Mapping Name	X	X
Test Code	X	X
Required Mappings	X	X
State Invariants & Guards	X	
Object Name		X
Class Name		X

Table 2: Attributes of element and object mappings

An object mapping asks for the class type and name of the object. The initialization of an object may also need other objects. So an object mapping may require extra object mapping as well. Table 2 indicates which attributes can be used in element and object mappings.

For the state machine of the vending machine program in figure 5, we need to create mappings for four transitions: *initialize*, *addChocs*, *getChocs*, and *coin*; and define six constraint mappings. The first mapping is for the transition *initialize*:

Mapping *vMachineInit* ***Transition*** *initialize*
{ *vendingMachine vm = new vendingMachine();* }

vMachineInit is the mapping name. The keyword **Transition** specifies that the mapping *vMachineInit* is created for a state transition.

Next is a mapping for the transition *getChocs*. The method *getChoc (StringBuffer)* is used to get chocolates from the vending machine. The *StringBuffer* object represents a chocolate.

Mapping *getChocolate* ***Transition*** *getChocs*
{
StringBuffer sb = new StringBuffer ("MM");
vm.addChoc (sb);
}

The mapping *getChocolate* gets only one chocolate from the vending machine. More chocolates can be taken from the vending machine if the method *getChoc (StringBuffer)* is called multiple times. Two objects *vm* and *sb* in the test code need to be initialized. Because the transition *initialize* appears in every test path, the object *vm* is initialized before any other test code, thus, it does not need an object mapping to initialize itself again. A *StringBuffer* variable *sb* is initialized directly in the test code of this mapping. Alternatively, the initialization of *sb* can be defined in an object mapping and reused in other mappings. The next example shows another mapping that gets two chocolates. It requires an object mapping.

Mapping *getTwoChocolates* ***Transition*** *getChocs*
Requires *stringBufferInit*
{
vm.getChoc (sb);
vm.getChoc (sb);
}

The object mapping for *stringBufferInit* is shown below:

Mapping *stringBufferInit* ***Class*** *StringBuffer* ***Object*** *sb*
{ *StringBuffer sb = new StringBuffer ("MM");* }

Please note that the initialization of an object should be either embedded in the test code of an element mapping or defined as an object mapping separately but not both. Otherwise the object will be defined twice.

Testers can provide multiple test values for primitive types and values will be chosen arbitrarily. For instance, test code "*vm.coin (10);*" can be mapped to the transition *coin*, and inserts a dime to the vending machine. Instead of assigning

a concrete *int* value 10, we can use the test code "*vm.coin (c);*" to map the transition *coin* and provide test values for the parameter *c* in an object mapping. An object mapping can be written below:

Mapping *cForCoin* ***Class*** *int* ***Object c***
{ *10, 25, 100* }

The vending machine only accepts dimes (10), quarters (25), and dollars (100). One of the three *int* values will be selected arbitrarily for the parameter *c*. Testers can also provide predicates such as $\{c > 0, c \leq 100\}$, separating conditions by commas. A constraint solver used in STALE will return a value that satisfies all constraints. The constraint solver has a limited language. It accepts numeric variables (*int*, *float*, and *double*), arithmetic operators, and regular expressions for *String*s. It does not accept disjuncts or function calls.

A mapping that specifies a constraint to be a state invariant is shown below.

Mapping *constraintForCredit* ***Constraint*** *constraint1*
StateInvariants *State3, State6, State9*
{ *vm.getCredit()* $\geq$ *90;* }

4.2 Graph Transformation and Test Path Generation

Each UML state machine diagram is transformed to a generic graph with initial nodes and final nodes. For a UML state machine diagram, an initial state is mapped to an initial node in the graph, a final state to a final node, and other states to unique nodes. Each transition becomes an edge and an internal transition from one state to itself becomes a self-loop on the corresponding node. Elements including composite state, choice, fork, junction, and join need special treatment.

Each sub-state of a composite state becomes a unique node and the composite state itself will not be transformed to a node. If a composite state has an initial state, an edge will be created for an incoming transition to the initial state of the composite state. If a composite state has *n* regular sub-states but no initial states, *n* edges will be created for an incoming transition, one for each node. Likewise, if a composite state has a final state, an edge will be created for an outgoing transition from the final state of the composite state. If a composite state has *n* sub-states but no final states, *n* edges will be created for an outgoing transition.

An edge will be created for each outgoing transition of a choice or fork. An edge will be created for each incoming transition of a join or junction.

Once the transformation from a UML state machine diagram to a generic graph is done, test paths can be generated based on a graph coverage criterion. The algorithms from our previous paper [29] are used to generate test paths.

4.3 Creating Mappings and Concrete Test Generation

After mappings are created, the test automation tool will save the mappings as *XML*. Figure 6 shows the saved mappings *vMachineInit*, *addChocolate*, *coinAnyCredit*, and *intCInit* in *XML*. Space does not allow all mappings to be shown.

⟨ *mappings* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***vMachineInit*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***i****nitialize* ⟨ */transition-name* ⟩
⟨ *code* ⟩
vendingMachine vm = new vendingMachine();
⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***addChocolate*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***addChocs*** ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***vm.addChoc ("MM");*** ⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***coinAnyCredit*** ⟨ */name* ⟩
⟨ *transition-name* ⟩ ***coin*** ⟨ */transition-name* ⟩
⟨ *code* ⟩ ***vm.coin(c);*** ⟨ */code* ⟩
⟨ */mapping* ⟩
⟨ */mappings* ⟩
⟨ *mapping* ⟩
⟨ *name* ⟩ ***intCInit*** ⟨ */name* ⟩
⟨ *object-name* ⟩ ***c*** ⟨ */object-name* ⟩
⟨ *class-name* ⟩ ***int*** ⟨ */class-name* ⟩
⟨ *code* ⟩ ***10, 25, 100*** ⟨ */code* ⟩
⟨ */mapping* ⟩

Figure 6: Mappings

Seven test paths are generated to satisfy edge coverage using the graph web application [8]. An example is [*initial*, *state1*, *state4*, *state7*, *state7*, *state9*, *state4*, *final*], whose abstract test is shown below:

initialize;
addChocs;
addChocs;
addChocs;
coin;
getChocs;

Testers can use the mapping *addChocolate* in figure 6 for the transition *addChocs* since only one mapping is created for the transition. If a transition has more than one mapping, the tool has to choose which to use. If the destination state of a transition has a constraint, the constraint has to be satisfied by the selected mapping. If the constraint is not satisfied, another mapping will be selected. If none of the mappings can satisfy the constraint, the tester is informed. This usually results in a correction to the model, the program, or the mappings.

An object or element mapping may require more than one object mapping. STALE is able to analyze the dependency relationship among all related object mappings. While generating concrete tests, the test code of object mappings that have

no dependency will be written first, followed by other object mappings that use variables from prior mappings.

When executing the example test path above, the vending machine will reach State9 with three chocolates in stock and one coin. The next step in the abstract test is *getChocs*, which should cause a transition to State4. However, State4 has a constraint (Constraint5), which says that the vending machine should only have one chocolate in stock. There is no way to satisfy that constraint, so an error will be reported to the tester. The error can be corrected in one of several ways. The model can be changed by modifying the constraint to be $stockOfVendingMachine \geq 1$, adding a transition on *getChocs* to State7, or by changing *getChocs* to allow more than one chocolate to be dispensed. Finding errors in the model when generating tests is a major benefit of this approach.

5 Experiment

The goal of this research is to decrease cost and errors made during test automation by reducing the amount of repetitive, mechanical work required to automate model-based tests. We pose two research questions:

1. RQ1: Can STAL be used to create automated tests in a practical setting?
2. RQ2: Can using STAL help testers reuse redundant test code and reduce errors when converting abstract tests to concrete tests as compared with doing the same procedure by hand?

This section presents STALE, the experimental design, subjects, procedure, results, threats to validity, and then an analysis of the results.

5.1 Tool Implementation

STALE uses the Eclipse Modeling Framework (EMF) library [5] to read EMF-based UML models. The tool transforms the behavioral models to generic graphs, then uses the test generation library on Ammann and Offutt's book website [8] to generate test paths. Testers develop the mappings in STAL, which are saved in XML files. The tool uses the numeric constraint solver Choco [30] and the *String* generator Xeger [31] to generate test values. Finally, the tool generates concrete tests by choosing test values that satisfy constraints, reporting unsatisfied constraints to the tester.

5.2 Experimental Design

Test engineers automate tests to reduce the cost of running the same test many times, to reduce the errors inherent in running tests by hand, and to make it easier to modify the test suite when the model, software, or test criterion changes. Evaluating STAL for all of these scenarios would require extensive human resources, so we evaluate the initial development of tests. The scenario is, given a program and its model, testers generate automated tests to satisfy a coverage criterion. Our testers did this by hand and with STAL. Any benefits from using STAL during initial development would also be present when modifying the tests.

	Automated (A)	**Manual (M)**
1	Find the test code for each element from a model. (A1)	The same as A1. (M1)
2	Extract object declarations and initializations from the element mappings. Enter mappings into the test automation tool and provide enough mappings to satisfy constraints. (A2)	Write executable tests to map test paths. (M2)
3	Generate concrete tests and correct errors. (A3)	Correct errors. (M3)

Table 3: Steps in automated and manual test generation processes

Nine testers designed tests for 17 programs. It can take many hours to design and develop model-based tests by hand, so each program was assigned to one tester. The tests were designed to satisfy edge coverage [7], a widely used and relatively simple test criterion. The testers developed two sets of tests for each program, one by hand and the other using STAL. There are two levels of automation in this study. The STALE tool helps testers **automate** the creation of tests, which are encoded in **automated** JUnit scripts. That is, we are automatically creating automated tests. We try to clarify which one we refer to in the following text.

An important decision was which process to use first, manual or automated. Table 3 shows the steps for each.

Step 1 is the same both by hand and with STAL. The testers need to understand the software, analyze its controllability and observability, and decide how to implement events from the model in a test. Step 2 is quite different for each process.

For A2, the testers identify the declarations and initializations of objects used in the test code for the elements of the model. Because an element may appear more than once in a test path, the corresponding test code will appear multiple times. Object declarations in the test code can result in duplicated object declarations. To avoid errors from this duplication, testers can either put all object declarations and necessary initializations in the mapping for the first transition if all test paths share the same transition, or create object mappings to be required mappings for elements. Testers then enter test code in STALE, satisfying the model constraints. This is the most time consuming part of the automated process.

In M2, the testers first look at the test paths, find matched elements from the model, and write the corresponding test code for the elements. This is the most time consuming part of manual test generation. Switching among the test paths, the model, and the test code is difficult, takes time, and can result in errors where the test code does not match the test paths.

When testers generate tests manually, they learn how to separate object declarations and how to create enough mappings to satisfy all constraints while writing the concrete tests.

If done first, A2 will become much easier and shorter, thus introducing a bias in favor of the automated process. However, if the automatic process is used first, the testers do not see the complete tests, thus the knowledge gained during step A2 does not make M2 easier.

Testers may get compilation errors in the automated process if they did not include all the classes or JAR files needed or if the test code in the mappings contain syntax errors. Also, if some constraints are not satisfied, the tester may need to add additional mappings or values. Testers correct these errors in step A3. If M3 is done before A2, the testers will be less likely to make mistakes, so A3 will become easier, again introducing a bias in favor of the automated process. However, doing A3 before M2 does not affect errors in M3 because they are arbitrary syntax errors.

Given these considerations, we concluded that the testers needed to first generate tests using the automated method, then manual. The guide that was given to the testers is online at *http://cs.gmu.edu/∼nli1/experiment/*.

5.3 Experimental Subjects

Seven of the 17 programs used are open source projects: Calculator[2], Snake[3], TicTacToe[4], CrossLexic[5], Jmines[6], Chess[7], and DynamicParser[8]. Six are from textbooks: VendingMachine [7], ATM [32], Tree [33], BlackJack [34], Triangle [35], and Poly [36]. The other four were taken from the coverage web application for Ammann and Offutt's book [8]. All programs are in Java.

The first author drew UML state machine diagrams using the Eclipse tool Papyrus [28], then the diagrams were transformed into generic graphs by STALE. For a few of the more complicated programs, parts of the programs' functionalities were omitted from the diagrams to ensure the testers could complete the program in the alloted two hour time frame. Nine testers (not including the authors) participated in the experiment. They were part-time and full-time graduate students at George Mason University, all of whom have taken Mason's graduate testing class.

5.4 Experimental Procedure

The testers were given the experimental guide and asked to understand the process and gain a preliminary familiarity with STALE. This took about two hours apiece. Next the testers entered our lab and generated tests automatically with STALE, then by hand, in a controlled environment. All subjects used the same computer and the first author measured their times. The automated steps were:

[2]http://jcalcadvance.sourceforge.net/
[3]http://sourceforge.net/projects/javasnakebattle/
[4]http://sourceforge.net/projects/tttnsd/
[5]http://crosslexic.sourceforge.net/
[6]http://jmines.sourceforge.net/
[7]http://twoplayerchess.sourceforge.net/
[8]http://dynamic-parser.sourceforge.net/

	Questions
1	Are you working (enter "programmer," "manager," "tester," etc.)? If not, enter "student."
2	If you have to generate tests from models, would you consider using this automatic test generation / tool?
3	Please rate the ease of use of the test automation tool on a scale of 1 to 5 (1 being impossible and 5 being trivially easy).
4	Do you have any suggestions for improving this automatic test generation / tool and other comments?

Table 4: The Questionnaire

1. Create a new project and add the model and program under test.
2. Create the abstract to concrete mappings. Wall-clock time was measured.
3. Create concrete tests using the tool to satisfy edge coverage. The tool measured the time for this step.

The manual steps were:

1. Write concrete tests by hand. The concrete tests have to be written in the same order as the test paths to make test comparison easy. The testers did not have to write test oracles, but the constraints in the states had to be satisfied.
2. Compile the tests and make sure that all tests pass. Wall-clock time was measured.

After completing the experiment, each participant was given an anonymous questionnaire, shown in table 4. Most of the participants had taken a graduate course in user interface design and development, so could be expected to be fairly knowledgeable and critical with question 3.

5.5 Experimental Results

Table 5 gives the experimental data. The first four columns give the subject names and statistics about the sizes of the graphs and programs. The nodes and edges are from the generic graph, not the original model, and the lines of code were calculated by CLOC [37]. Note that the last four subjects belong to one program and LOC is for the entire program.

The next three columns show the number of distinct mappings created from the models, the number of times the distinct mappings appear in all the tests, and the ratio of the Mappings column over the All Mappings column. For example, the *vendingMachine* model needed 13 mappings and the seven tests used a total of 132 mappings, 9.8% of which are distinct.

The next two columns present the number of seconds used to create mappings and generate tests in the automated process, followed by the time used by the manual process. Last is the ratio of time for the automated process over the time for the manual process. Thus, the automated process for class *vendingMachine* took 34.5% of the time of the manual process.

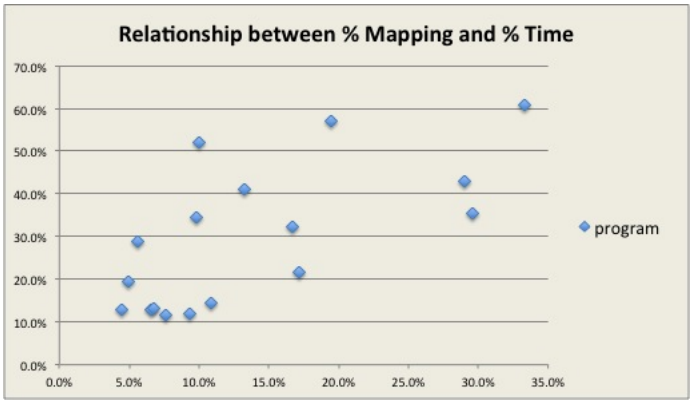

Figure 7: Ratio of number of distinct mappings over number of mappings in all tests

Our first research question asked if STAL could be used in a practical situation. All nine subjects were able to use STAL with only a short tutorial, so we conclude the answer to RQ1 is yes. Our second research question asked if testers could use STAL to reuse redundant test code and reduce errors. Table 5 shows that the automated process ranged from 11.7% to 60.8% of the time the manual process took, with an unweighted average of 29.6%. The manually created tests have 48 errors compared with 0 for the automatically created tests. Thus, we conclude that the answer to RQ2 is also yes.

On the questionnaires, all subjects answered "Yes" to the second question, and the average usability rating (third question) was 4.4.

5.6 Experimental Analysis

Figure 7 compares the *% Mapping* (on the horizontal axis) and *% Time* (on the vertical axis) for the 17 subject programs. We analyze these data to look for a linear correlation relationship.

Boddy and Smith [38] suggest using Pearson's correlation coefficient if the data have a normal distribution; otherwise, we should use a non-parametric correlation test [39] such as Spearman's rank correlation coefficient. Qqplots (not shown due to space) show that the *% Mapping* and *% Time* data deviate from the straight line. Thus, we use Spearman's correlation coefficient.

The *correlation coefficient* (ρ) of Spearman's correlation test is 0.72. Cohen [40] suggests that a value of .5 or greater can be considered to be a large correlation. The statistical significance *p-value* is 0.0017, which is normally considered to be highly significant. Therefore, we conclude that the savings from using the automated process increases as the percentage of distinct mappings in all mappings decreases.

5.7 Threats to Validity

As usual with most software engineering studies, there is no way to show that the selected subjects are representative. This is true both for the programs and the human testers. Another threat to external validity is that the first author created the UML models from the source code. An internal threat is that the test automation tool's implementation may be imperfect.

6 Conclusions

This paper has two results. The first is a general, practical solution to the mapping problem for transforming abstract model-based tests to concrete executable tests. This is done using a test automation language, STAL, as described in section 4. Testers use STAL to define mappings between elements in the abstract tests to specific sequences of code that will be part of the concrete executable tests. STAL is embedded in a test framework, STALE, which accepts model-based abstract tests and automatically creates fully executable concrete tests.

The test automation language can be used whenever abstract tests include the same elements many times, resulting in duplicate components of concrete tests. This paper explains STAL in the context of using graph-based test criteria defined on graphs that were derived from state machine diagrams, but it can also be used with other models and other techniques for designing model-based tests.

This paper also compares test generation using STAL with manual test generation. The results, based on 17 programs, show that automatic test generation uses 29.6% of the time for manual test generation. The manual tests also contained 48 errors in which concrete tests do not map abstract tests.

7 Future Work

Another aspect of this research project is determining tradeoffs among choices of test oracle. Generally speaking, automated tests run a program or a program component, then compare part of the program state with pre-determined values to see if the test passed (the states match) or the test failed (the states differ). Crucial questions are **how much** of the states should be checked, and **when** should they be compared. Comparing the entire program state can be expensive, possibly prohibitively so. On the other hand, comparing an insufficient amount of the program state can lead to type II errors, where a failure is not detected. Research into this question is ongoing.

Acknowledgment

We thank Dr. Paul Ammann for helping us to set up the experiment and all the participants for generating tests.

References

[1] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Software Testing, Verification, and Reliability, Wiley*, vol. 13, no. 1, pp. 25–53, March 2003.

Programs	LOC	Nodes	Edges	Mappings	All Mappings	% Mapping	Tests	Automatic (Seconds)		Manual (Seconds)	% Time
								Mapping Creation	Test Generation	Test Generation	
VendingMachine	52	11	26	13	132	9.8	7	630	4	1,836	34.5
ATM	463	8	12	8	27	29.6	5	449	1	1,267	35.5
Calculator	2,919	17	76	12	182	6.6	14	477	15	3,794	13.0
Triangle	124	7	31	12	72	16.7	6	440	2	1,371	32.2
Snake	1,382	18	116	13	132	9.8	7	503	46	1053	52.1
TicTacToe	665	7	12	9	31	29.0	5	640	2	1,494	43.0
CrossLexic	654	13	51	17	305	5.6	26	609	123	2,539	28.8
JMines	9,486	18	75	10	202	5.0	26	445	62	2,625	19.3
Chess	2,048	9	17	7	36	19.4	6	510	6	904	57.1
BlackJack	403	13	20	12	36	33.3	8	300	4	500	60.8
Tree	234	14	24	11	83	13.3	6	685	2	1671	41.1
Poly	129	8	21	11	64	17.2	5	330	3	1537	21.7
DynamicParser	1,269	22	73	12	269	4.5	21	468	45	4010	12.8
GraphCoverage	14,155	20	67	17	253	6.7	19	521	14	4091	13.1
DFCoverage	14,155	15	56	16	147	10.9	19	401	7	2824	14.4
LogicCoverage	14,155	15	83	15	196	7.7	38	522	7	4512	11.7
MinMCCoverage	14,155	14	53	14	150	9.3	22	434	1	3642	11.9
Total	232	742	33,983	196	2,325		240	8,364	344	39,670	
Average						13.8					29.6

Table 5: Time for automatic and manual test generation

[2] J. Offutt and A. Abdurazik, "Generating tests from UML specifications," in *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*. Fort Collins, CO: Springer-Verlag Lecture Notes in Computer Science Volume 1723, October 1999, pp. 416–429.

[3] N. Li, "A smart structured test automation language (SSTAL)," in *The Ph.D. Symposium of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST),*, ser. ICST '12, Montreal, Quebec, Canada, April 2012, pp. 471–474.

[4] ——, "The structured test automation language framework," Online, 2013, http://cs.gmu.edu/ nli1/stale/, last access May 2013.

[5] E. Foundation, "Eclipse modeling framework," Online, 2008, http://www.eclipse.org/modeling/emf/, last access Sept 2012.

[6] Mark, Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, August 2012.

[7] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, UK: Cambridge University Press, 2008, iSBN 0-52188-038-1.

[8] P. Ammann, J. Offutt, W. Xu, and N. Li, "Graph coverage web applications," Online, 2008, http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage, last access May 2013.

[9] W. Prenninger and A. Pretschner, "Abstractions for model-based testing," *Electronic Notes in Theoretical Computer Science*, vol. 116, pp. 59–71, January 2005.

[10] M. Utting, G. Perrone, J. Winchester, S. Thompson, R. Yang, and P. Douangsavanh, "The ModelJUnit model-based testing tool," Online, 2007, http://www.cs.waikato.ac.nz/ marku/mbt/modeljunit/, last access April 2013.

[11] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, and N. Tillmann, "Microsoft SpecExplorer," Online, 2002, http://research.microsoft.com/en-us/projects/specexplorer/, last access April 2013.

[12] J. Jacky and M. Veanes, "NModel," Online, 2006, http://nmodel.codeplex.com/, last access April 2013.

[13] C. Inc., "CONFORMIQ Automated Test Design," Online, 2011, http://www.conformiq.com/, last access April 2013.

[14] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[15] P. Fröhlich and J. Link, "Automated test case generation from dynamic models," in *Proceedings of the 14th European Conference on Object-Oriented Programming*, ser. ECOOP '00. London, UK: Springer-Verlag, 2000, pp. 472–492.

[16] J. Ryser and M. Glinz, "A scenario-based approach to validating and testing software systems using statecharts," in *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications*, ser. ICSSEA '99, Paris, France, 1999.

[17] L. Briand and Y. Labiche, "A UML-based approach to system testing," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, ser. UML '99. London, UK: Springer-Verlag, 2001, pp. 194–208.

[18] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel, "Automatic test generation: a use case driven approach," *IEEE Transaction on Software Engineering*, vol. 32, no. 3, pp. 140–155, March 2006.

[19] S. Liu and S. Nakajima, "A framework for automatic functional testing based on formal specifications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. Waikiki, Honolulu, HI, USA: ACM, May 2011, pp. 107–108.

[20] A. Ulrich, E.-H. Alikacem, H. H. Hallal, and S. Boroday, "From scenarios to test implementations via Promela," in *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ser. ICTSS '10. Natal, Brazil: Springer-Verlag, 2010, pp. 236–249.

[21] D. Lugato, C. Bigot, and Y. Valot, "Validation and automatic test generation on UML models: The AGATHA approach," *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 2, pp. 33–49, 2002.

[22] Y. Kim, H. Hong, D. Bae, and S. Cha, "Test cases generation from UML state diagrams," *IEE Proceedings. Software*, vol. 146, no. 4, pp. 187–192, August 1999.

[23] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, E. Samuelsson, I. Schieferdecker, and C. E. Williams, "The UML 2.0 testing profile," in *Proceedings of the '8th Conference on Quality Engineering in Software Technology 2004*, ser. CONQUEST 2004. Nuremberg, Germany: ASQF e.V., Erlangen, September 2004, pp. 181–189.

[24] O. M. Group, "OMG MOF 2 XMI mapping specification," Online, 2011, http://www.omg.org/spec/XMI/2.4.1/, last access Sept 2012.

[25] ——, "MOF model to text transformation language," Online, 2008, http://www.omg.org/spec/MOFM2T/1.0/, last access Sept 2012.

[26] ——, "OMG model driven architecture," Online, 2003, http://www.omg.org/mda/, last access Sept 2012.

[27] E. Foundation, "Acceleo - transforming models into code," Online, 2009, http://www.eclipse.org/acceleo/, last access Sept 2012.

[28] ——, "Papyrus," Online, 2008, www.eclipse.org/papyrus/, last access Sept 2012.

[29] N. Li, F. Li, and J. Offutt, "Better algorithms to minimize the cost of test paths," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Montreal, Quebec, Canada: IEEE Computer Society, April 2012, pp. 280–289.

[30] T. C. D. Team, "Choco constraint solver," Online, 2004, http://www.emn.fr/z-info/choco-solver/, last access May 2013.

[31] X. Team, "Xeger string generator," Online, 2009, https://code.google.com/p/xeger/, last access May 2013.

[32] H. Deitel and P. Deitel, *Java: How to program*, 6th ed. Pearson Education, Inc., 2005.

[33] Anonymous, "Class of tree," Online, 2008, http://homepage.cs.uiowa.edu/∼sriram/21/fall08/code/tree.java, last access May 2013.

[34] Lewis, Chase, and Coleman, "Class of blackjack," Online, 2004, http://faculty.washington.edu/moishe/javademos/blackjack/, last access May 2013.

[35] M. Rusma, "Class of triangle," Online, 2004, http://www.cs.du.edu/∼snarayan/sada/teaching/COMP3705/FilesFromCD/Exercises/Lab4_WhiteBox/Triangle.java, last access May 2013.

[36] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, 1st ed. Addison-Wesley Professional, 2000.

[37] A. Danial, "CLOC," Online, 2006, http://cloc.sourceforge.net, last access Sept 2012.

[38] R. Boddy and G. Smith, *Effective Experimentation: For Scientists and Technologists*. Wiley, 2010.

[39] J. Miles and M. Shevlin, *Applying Regression and Correlation: A Guide for Students and Researchers, Sage Publications*, 1st ed. SAGE Publications Ltd, 2000.

[40] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed. Hillsdale, New Jersey, USA: Lawrence Erlbaum Associates, Inc., 1988.