A Test Automation Language Framework for Behavioral Models

Nan Li
Research and Development
Medidata Solutions
nli@mdsol.com

Jeff Offutt
Software Engineering
George Mason University
offutt@gmu.edu

***Abstract*—Model-based testers design tests in terms of models, such as paths in graphs. This results in abstract tests, which have to be converted to concrete tests because the abstract tests use names and events that exist in the model, but not the implementation. Model elements often appear in many abstract tests, so testers write the same redundant code many times. However, many existing model-based testing techniques are very complicated to use in practice, especially in agile software development. Thus, testers usually have to transform abstract tests to concrete tests by hand. 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 tests.

Based on the language, we developed a test automation language framework. The paper also presents results from an empirical comparison of testers using the framework with manual mapping on 11 open source and 6 example programs. We found that the automated test generation method took 29.6% of the time the manual method took on average, and the manual tests contained 48 errors in which concrete tests did not match their abstract tests while the automatic tests had zero errors.

I. Introduction and Motivation

In *model-based testing* (*MBT*), testers generate tests from behavioral models that reflect functional aspects of the system. For example, finite state machines (FSM) that represent the behavior of a system are often used to generate tests. Figure 1 shows a general process to derive tests from behavioral models.

Testers can generate test requirements by hand or by using a coverage criterion based on a model. This research focuses on coverage criteria. *Abstract tests*, which are expressed in terms of a model, are generated to satisfy the test requirements. Additional information, including test values and data mappings, may be needed to convert abstract tests to concrete tests. *Concrete tests* are expressed in terms of the implementation of the model, and are ready to be run automatically. Expected outputs can be specified in models or provided manually by testers. Then *test oracles* are used in concrete tests to determine whether the tests pass. This research addresses the *mapping problem*, which is the problem of translating model-level elements in abstract tests to code-level elements in concrete tests (the bold rectangles in Figure 1) [1], [2]. The mapping problem may result in testers writing redundant code because elements in the model could appear multiple times in tests, leading to errors and lost time. Thus, a major goal of this research is to avoid this redundancy.

N. Li is with the research and development division at Medidata solutions, New York, NY, USA. This work was done while the first author was at George Mason University.

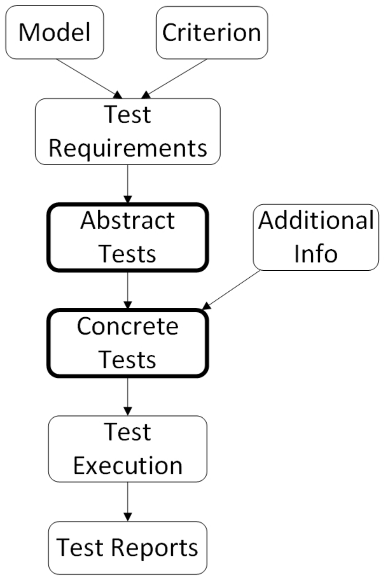

Fig. 1. A Generic Model-based Testing Process

This research strictly focuses on *non-executable* models that are translated to source by hand, as opposed to *executable models*, which can use automatic model-to-source translations [3]. The mapping problem is significantly easier with model-to-source translations, because the mappings are created automatically by the translator.

We have identified four major issues in model-based testing: (1) building test models, (2) using test criteria and algorithms to generate abstract tests from models, (3) transforming abstract tests to concrete tests, and (4) generating test oracles in concrete tests. The second, third, and fourth issues largely depend on which and how many models are used (issue 1). When many types of models are available, the information to construct the mappings may be encoded in the models, allowing the mappings (issue 3) to be largely automated. We assume a development scenario where only behavioral models are available, thus the information needed to construct abstract to concrete test mappings is only available in the developers' heads (or encoded implicitly in the implementation). This is common in industry.

We have developed a complete MBT approach to address issues 2 through 4. This research uses non-executable Unified Modeling Language (UML) behavioral diagrams, as discussed below. For issue 2, we developed a *prefix-graph based solution* to generate abstract tests with minimum cost [4]. This paper addresses issue 3 (the mapping problem) with a structured test automation language to automate the transformation from abstract tests to concrete tests. For issue 4, we developed new cost-effective *test oracle strategies* to check various program states with different frequencies [5].

How we solve the mapping problem depends on the kind of models used. While creating test models, testers can either write *model program*s in a specific language or draw visual diagrams. Model programs use specification languages such as Spec# in Spec Explorer [6] or programming languages such as C# in NModel [7] to describe model behaviors. Then the model programs can be converted to finite state machines or extended finite state machines.

Many testers prefer to use visual diagrams such as UML behavioral diagrams directly as test models [8], [9], [10], [11], [12], [13], [14], [15]. They can reuse and adapt design models or build new test models [16]. UML behavioral models are widely used and easy to understand to many people. Design models usually reflect system behaviors and can be reused to build test models. Thus, this paper applies the same approach.

Non-executable behavioral 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 solution is to use additional supporting UML models (for example, use case diagrams and class diagrams) to provide the missing information [10], [11], [15].

Briand and Labiche [10] used this technique in their TOTEM system, which assumes many artifacts that are not always available. Nebut et al. [11] presented a use case driven approach, which extracted additional information from use case models and required a behavioral model (sequence, state machine, or activity diagram) to specify the sequence ordering of the use cases. They assumed use cases had contracts 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 were assumed to be the same as those in the behavioral models.

With these assumptions, the abstract tests have lots of abstract test elements, including objects, parameters, actions, constraints, and so on. To make tests executable, abstract test elements have to be mapped to concrete objects, parameters, method calls, or other test code. In many cases, one distinct abstract object is mapped to one distinct concrete object and one action is mapped to one method call. We call this "fine-grained mapping" because what to map from abstract tests to implementation has been very well defined. The mapping creation becomes straightforward since the models are highly consistent and abstract elements can be identified in the models.

Creating many formalized diagrams for testing may be practical for organizations that can reuse models from the design phase and have many model-based testing experts. However, it is expensive for organizations that do not have such resources. It is also hard to apply such MBT approaches to agile software development. Agile processes release software products daily or weekly. Frequent changes of requirements can introduce inconsistencies among the diagrams, making the MBT approaches hard to apply in practice.

This research assumes non-executable models and a limited collection of behavioral model diagrams. In this scenario, practical testers must map abstract tests to concrete tests by hand. To support this scenario, we have developed a Structured Test Automation Language (STAL). A previous paper [17] proposed the idea; this paper follows with the language and a detailed solution to the mapping problem.

We assume that the organization has a limited number of non-executable behavioral models. Moreover, testers are more comfortable with programming than with modeling. Without supporting diagrams, abstract tests based on state machine diagrams are limited to transitions (actions) and constraints. We call this "coarse-grained mappings," since abstract tests do not have many kinds of abstract elements. Our Structured Test Automation Language framEwork (STALE) has been designed specifically to support this scenario.

This paper addresses three key issues for developing and building the framework 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.

We evaluate our technique by comparing the use of STALE with manual translation of abstract to concrete tests. Most previous research has relied on case studies [10], [15], [16], [18], [19], [20]. We compared test generation using STALE with the manual approach on 17 programs, finding that using STALE took only 29.6 percent of the time that manual test generation took. Additionally, the manual approach resulted in 48 errors in 240 tests in which the executable code did not match the abstract tests. Testers made no errors when using STALE.

The paper is organized as follows. Section II presents additional related work and background about model-based testing. Section III describes the key issues that had to be addressed when using STAL and building STALE. Section IV presents STALE and the empirical validation. The paper is concluded in section V.

II. Related Work and Background

This section presents related work about the mapping problem, then introduces background in transforming behavioral models to general graphs and applying coverage criteria on the graph. Some related work was already summarized in the introduction and is not repeated here.

A. Related Work

Transforming abstract tests to concrete tests depends on the model and what elements the abstract tests have. Spec Explorer [18] reads model programs and converts them to extended finite state machines. The transitions contain the abstract class objects and actions. Each distinct object in the model is mapped to a different concrete class object. Then its associated action is mapped to one method call according to its concrete objects. Paiva et al. [19] used Spec Explorer to generate tests for a NotePad GUI application but found difficulty in identifying concrete objects that are mapped to model actions. So they developed a GUI mapping tool to bind model actions to method calls. Each action in the model is mapped to only one method call in their mappings. Briand and Labiche's TOTEM system [10] generates abstract tests with abstract test values, including abstract objects and parameters. Because of the high consistency among behavioral and supporting diagrams, the abstract test values can be identified in supporting diagrams and then converted to concrete test values.

The UML Testing Profile (UTP) [15] 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.

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 [21]. *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 [22]. Our tool uses the node, edge, edge-pair and prime path coverage criteria [23].

We started by trying to use the existing model-to-test transformation language *Meta-Object Facility Model To Text Transformation Language (MOFM2T)* [24] to automate the mapping problem. Unfortunately, characteristics of the tool made it not suitable for our research.

MOFM2T [24] is part of OMG's *model-driven architecture (MDA)* [25] and was designed to transform models to code for general use. *Acceleo* [26] is the only Eclipse Foundation project that implements MOFM2T. It reads Eclipse Modeling Framework [27] (*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 could not 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 element names directly, so testers would need to write code to look for the identifiable element from the top level to the bottom level of the model structure.

Because MOFM2T cannot be used directly for testers to provide mappings from models to executable test code, testers have to use the test automation language (STAL) to write mappings. Then there will be one more layer to translate the test automation language to MOFM2T if we adapt MOFM2T. Therefore, using MOFM2T would make the implementation complicated.

B. 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. STALE transforms UML state machine diagrams into generic graphs, and then uses graph coverage criteria to design tests. The following definitions are taken from Ammann and Offutt [23]. Formally, a graph G is a non-empty set of *nodes* N, and a set of *edges* E, where E is a subset of $N \times N$. N contains a non-empty set N_0 of *initial nodes* and a non-empty set N_f of *final nodes*.

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 the number of nodes. 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.

III. The Test Automation Language STAL

This section presents our language for automating the generation of concrete tests from abstract tests. This paper uses UML state machine diagrams as examples to explain how to use STAL. STAL can also be used for other behavioral diagrams. The vending machine example is used to illustrate how testers use STAL to create mappings. The vending machine sells only chocolates; the price for all chocolates is 90 cents; only dimes, quarters, and dollars are accepted; and only 10 chocolates are allowed in stock. Figure 2 is a UML state machine diagram for the vending machine example created with the *EMF*-based tool *Papyrus* [28]. Figure 2 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*. The implementation of class *VendingMachine* has a constructor, method *coin (int)* to insert coins, method *getChoc (StringBuffer)* to get chocolates, and method *addChoc (String)* to add chocolates.

Three key issues have been addressed in this research: (1) creating mappings and generating test values, (2) transforming the graph and generating test paths using coverage criteria, and (3) solving constraints and generating concrete tests.

[1]Constraints can be specified to be guards and post-conditions on transitions or state invariants in states.

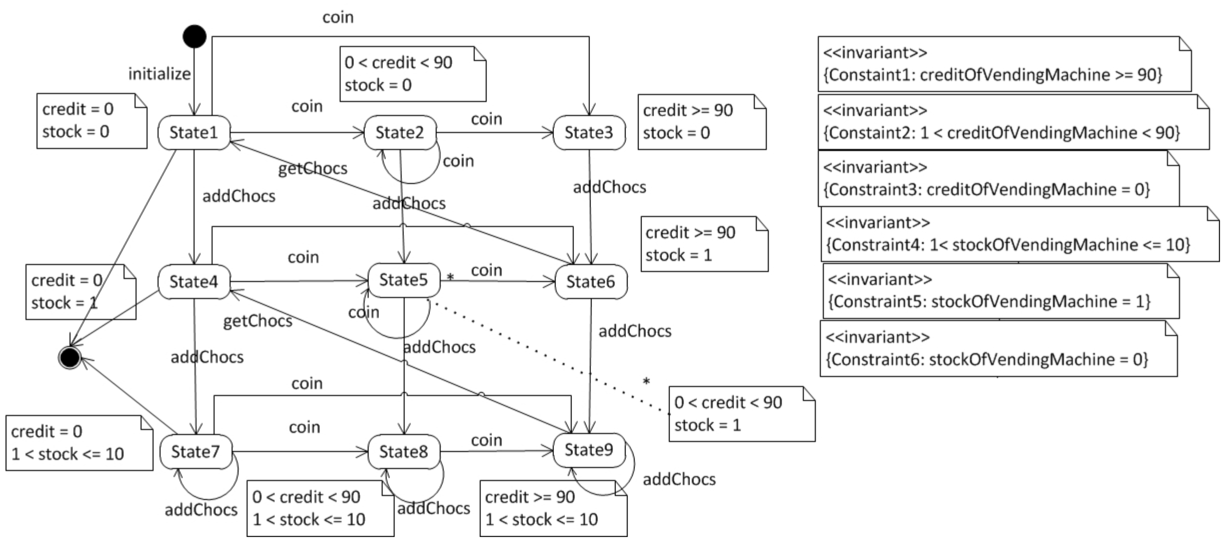

Fig. 2. A UML State Machine Diagram for the Class VendingMachine

A. Creating Mappings and Generating Test Values

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

- 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

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 ...*
*[**TYPEOFCONSTRAINT** nameOfElement ...] ...*
{*testCode*}

Mapping and **Requires** are keywords in element mappings. The mapping name must be unique. **TYPEOFELEMENT** may vary depending on the actual type of the concrete element. **Transition** and **Constraint** are used in this research. 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, element, or type may be required. If an element is a constraint, the mapping needs to give the type of the constraint (state invariant, guard, etc.) and elements (e.g., states or transitions) in which the constraint is held. Thus, **TYPEOFCONSTRAINT** can be **StateInvariants**, or **Guards**. Constraint 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 and post-conditions on transitions at the same time. Test code is required for any mapping and written in curly brackets. An object mapping is formally defined as:

TABLE I. ATTRIBUTES OF ELEMENT AND OBJECT MAPPINGS

Attributes	Element Mapping	Object Mapping
Element Name	X	
Element Type	X	
Mapping Name	X	X
Test Code	X	X
Required Mappings	X	X
Constraints	X	
Object Name		X
Class Name		X

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

An object mapping asks for the class type and name of the object. **Mapping**, **Class**, **Object**, and **Requires** are keywords in object mappings. The initialization of an object may also need other objects. So an object mapping may require extra object mapping as well. Table I indicates which attributes can be used in element and object mappings.

For the state machine of the vending machine program in Figure 2, 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 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:

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

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, the test code "*vm.coin (10);*" can be mapped to the transition *coin* to insert 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 as:

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 values will be selected arbitrarily for the parameter *c*. An *int* parameter could also be given a random number. Testers just write "*anyInt*" to have the testing framework generate a random *int* number. Test values can be generated automatically for all Java primitive types (*boolean, byte, short char, float, double*, and *long*). Testers can also provide predicates such as $\{c > 0, c \leq 100\}$, separating conditions by commas. A constraint solver, Choco [29], will return a value that satisfies all constraints. The constraint solver has a limited language. It accepts numeric variables (*int*, *float*, and *double*), and arithmetic operators. Another constraint solver Xeger [30] is used to generate *String*s to match regular expressions. Neither constraint solver accepts disjuncts or function calls.

A mapping that specifies a constraint to be a state invariant is shown below. In this example, Constraint1 is a state invariant for State3, State6, and State9. The credit of the vending machine has to be equal to or greater than 90 cents in these three states.

Mapping *constraintForCredit* **Constraint** *Constraint1*
  StateInvariants *State3, State6, State9*
  { *vm.getCredit()* $\geq$ *90;* }

B. 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 [31] are used to generate test paths.

C. Creating Mappings and Generating Concrete Tests

After mappings are created, STALE will save the mappings as *XML*. Figure 3 shows example mappings in *XML*.

Seven test paths are generated to satisfy edge coverage using the graph web application [32]. An example is [*initial*, *state1*, *state4*, *state7*, *state7*, *state9*, *state4*, *final*], whose abstract test is: initialize, addChocs, addChocs, addChocs, coin, getChocs.

Testers can use the mapping *addChocolate* in Figure 3 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

```
⟨ mappings ⟩
 ⟨ mapping ⟩
   ⟨ name ⟩ vMachineInit ⟨ /name ⟩
   ⟨ transition-name ⟩ initialize ⟨ /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 ⟩ coinOneDollar ⟨ /name ⟩
   ⟨ transition-name ⟩ coin ⟨ /transition-name ⟩
   ⟨ code ⟩ vm.coin(100); ⟨ /code ⟩
 ⟨ /mapping ⟩
 ⟨ mapping ⟩
   ⟨ name ⟩ coinOneDime ⟨ /name ⟩
   ⟨ transition-name ⟩ coin ⟨ /transition-name ⟩
   ⟨ code ⟩ vm.coin(10); ⟨ /code ⟩
 ⟨ /mapping ⟩
 ⟨ mapping ⟩
   ⟨ name ⟩ coinAnyCredit ⟨ /name ⟩
   ⟨ transition-name ⟩ coin ⟨ /transition-name ⟩
   ⟨ code ⟩ vm.coin(c); ⟨ /code ⟩
 ⟨ /mapping ⟩
 ⟨ mapping ⟩
   ⟨ name ⟩ intCInit ⟨ /name ⟩
   ⟨ object-name ⟩ c ⟨ /object-name ⟩
   ⟨ class-name ⟩ int ⟨ /class-name ⟩
   ⟨ code ⟩ 10, 25, 100 ⟨ /code ⟩
 ⟨ /mapping ⟩
 ⟨ mapping ⟩
   ⟨ name ⟩ getTwoChocolates ⟨ /name ⟩
   ⟨ transition-name ⟩ getChocs ⟨ /transition-name ⟩
   ⟨ code ⟩ vm.getChoc(sb); vm.getChoc(sb); ⟨ /code ⟩
   ⟨ required-mappings ⟩ stringBufferInit ⟨ /required-mappings ⟩
 ⟨ /mapping ⟩
 ⟨ mapping ⟩
   ⟨ name ⟩ stringBufferInit ⟨ /name ⟩
   ⟨ object-name ⟩ sb ⟨ /object-name ⟩
   ⟨ class-name ⟩ StringBuffer ⟨ /class-name ⟩
   ⟨ code ⟩ StringBuffer sb = new StringBuffer(“MM”); ⟨ /code ⟩
 ⟨ /mapping ⟩
 ⟨ mapping ⟩
   ⟨ name ⟩ constraintForCredit ⟨ /name ⟩
   ⟨ constraint-name ⟩ Constraint1 ⟨ /constraint-name ⟩
   ⟨ code ⟩ vm.getCredit() ≥ 90; ⟨ /code ⟩
   ⟨ state-invariant ⟩ State3,State6,State9 ⟨ /state-invariant ⟩
 ⟨ /mapping ⟩
⟨ /mappings ⟩
```

Fig. 3. Mappings

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 analyzes the dependency relationship among all related object mappings. While generating concrete tests, the test code of object mappings that have no dependencies 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 and at least 90 cents. The next step in the abstract test is *getChocs*, which should cause a transition to State4. However, Constraint5 on State4 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 tester can then change the model by modifying the constraint to be $stockOfVendingMachine \geq 1$, add a transition on *getChocs* to State7, or change *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.

IV. Empirical Study

The goal of STAL is to decrease cost and errors made during test automation by reducing the repetitive, mechanical work involved in automating model-based tests. In this experiment, experimental subjects used STALE that implements STAL to generate tests automatically, and then generated tests manually for the same program. We pose three research questions:

1) RQ1: Can STALE be used to create automated tests in a practical setting?
2) RQ2: Can using STALE 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?
3) RQ3: Can STALE be used for different kinds of programs such as web applications and GUIs?

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

A. Implementation of STALE

STALE uses EMF library to read EMF-based UML models. STALE supports UML state machine diagrams, and work on supporting other diagrams is ongoing. The tool transforms the behavioral models to generic graphs, then uses the test generation tool on Ammann and Offutt's book website [32] to generate test paths. Testers develop mappings in STAL, which are saved in XML files. Finally, the tool generates concrete tests by choosing test values that satisfy constraints, reporting unsatisfied constraints to the tester. The detailed installation and user guides for STALE can be found at *http://cs.gmu.edu/∼nli1/stale/*.

B. 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 STALE 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 STALE. Any benefits from using STALE during initial development should also be present when modifying the tests.

Nine testers designed tests for 17 programs. Six participants were assigned one program and three participants were assigned more than one. Our subjects took from three to ten

TABLE II. STEPS IN AUTOMATED AND MANUAL TEST GENERATION PROCESSES

	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 STALE 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)

hours to design and develop model-based test by hand, so each program was assigned to one tester. The tests satisfied 100% edge coverage on the model, a widely used and relatively simple test criterion [23]. The testers developed two sets of tests for each program, one by hand and the other using STALE. This study had two levels of automation. STALE helps testers **automate** the creation of tests, which are encoded in **automated** JUnit scripts. That is, we are automatically creating tests that execute automatically. 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 II shows the steps for each.

Step 1 is the same by hand and with STALE. 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.

For M2, the testers first analyze the test paths to find matched elements from the model, then write the 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, slow, 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 because separating object declarations and creating mappings takes most of the time during A2. Thus, this would introduce a bias in favor of the automated process. If the automatic process is done first, testers would learn how to write code for each transition and state, since they must analyze them thoroughly to create mappings. However, this does not reduce much time for M2 because testers spend most of their time checking if the test code matches the model, checking if elements in the model match test paths, writing code for redundant elements, and correcting errors when mismatched code is found. Additionally, the testers did not see the complete tests from A2, therefore, knowledge gained during step A2 did not simplify M2.

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 A3, 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 M3 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 to avoid introducing bias, then the manual process. The guide that was given to the testers is online at *http://cs.gmu.edu/∼nli1/experiment/*.

C. Experimental Subjects

Seven of the 17 programs are open source projects: Calculator, Snake, TicTacToe, CrossLexic, Jmines, Chess, and DynamicParser. Six are from textbooks: VendingMachine [23], ATM [33], Tree [34], BlackJack [35], Triangle [36], and Poly [37]. The other four were taken from the coverage web application for Ammann and Offutt's book [32]. All programs are in Java. Four types of subjects were selected to evaluate RQ3. Calculator, Snake, CrossLexic, Jmines, Chess, BlackJack, and DynamicParse were GUIs. GraphCoverage, DFCoverage, LogicCoverage, and MinMCCoverage were web applications. TicTacToe was a command-line program. The other five programs are subsystems that do not have user interfaces.

The first author drew UML state machine diagrams using the Eclipse tool Papyrus [28], then used STALE to transform them into generic graphs. 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 allotted 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. Prior knowledge of the subject programs would not affect the experiment. The participants were given the programs before the experiment and asked to run and understand them. The goal was to ensure that part of the learning process did not affect which test generation technique was used first.

D. Experimental Procedure

The testers were given the experimental guide and asked to understand the process and gain a preliminary familiarity with STALE. They had to understand the assigned model and program, and know how to create mappings for model elements. 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

TABLE III. THE QUESTIONNAIRE

	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?

the same computer and the first author measured their times. Each subject generated tests separately at different times. They did not know who the other participants were, and thus had no opportunity to communicate with each other during the experiment. The automated steps were:

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 constraints in the states had to be satisfied. Wall-clock time was measured.
2) Compile the tests and make sure that all tests pass. Wall-clock time was measured.

After completing the experiment, participants were given the questionnaire in Table III anonymously. Most had taken a graduate course in user interface design and development, so could be expected to be fairly knowledgeable and critical with question 3.

E. Experimental Results

The data are shown in Table IV. The subject names and statistics about the sizes of the graphs and programs are given first. The nodes and edges are from the generic graph, not the original model, and the lines of code were counted by CLOC [38].

The next three columns show the number of distinct mappings created from the models, the number of times the mappings appear in all the tests, and the ratio of the Mappings column over the All Mappings column. The *VendingMachine* model needed 13 mappings and the seven tests used 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 and 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.

Our first research question asked if STALE could be used in a practical situation. All nine subjects were able to use STALE with only a short tutorial, so we conclude the answer to RQ1 is yes. Our second research question asked if testers could use STALE to reuse redundant test code and reduce errors. Table IV 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%. We examined the tests for errors by hand and found 48 errors in the manually created tests and none in the automatically created tests. We found two types of errors: (1) unmatched test code for a transition in the model; and (2) redundant test code for same transitions. The participants made more errors with large programs. Thus, we conclude that the answer to RQ2 is also yes. The STALE process included an explicit error correction step (A3 in Table II). Although testers also checked for errors in M3, they often missed those errors, especially when the program needed more tests. Nine subjects were able to generate tests for the programs including GUI applications, web applications, command-line programs, and normal programs. Therefore, the answer to RQ3 is yes as well.

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

F. Experimental Analysis

Figure 4 compares *% Mapping* and *% Time* for the 17 subject programs. *% Mapping* and *% Time* have the same meaning as in Table IV. If *% Mapping* is small, testers do not need to create many distinct mappings by comparison with all mappings that appear in all tests. Therefore, it is likely that the automatic process takes far less time than the manual process, since manual test generation for mappings on repeated transitions requires repeated work.

Boddy and Smith [39] suggest using Pearson’s correlation coefficient if the data have a normal distribution; otherwise, we should use a non-parametric correlation test [40] 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 [41] 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.

G. 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 STALE’s implementation may be imperfect. Cutting questionnaire for space Another internal threat is that the answers of the participants to the questionnaires could be influenced by the fact that some participants knew the experimenter.

TABLE IV. TIME FOR AUTOMATIC AND MANUAL TEST GENERATION

Programs	LOC	Nodes	Edges	Map-pings	All Map-pings	% Map-ping	Tests	Automatic (Seconds)		Manual (Seconds)	% Time
								Mapping Creation	Test Gen-eration	Test Gen-eration	
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	4, 480	20	67	17	253	6.7	19	521	14	4091	13.1
DFCoverage	4, 512	15	56	16	147	10.9	19	401	7	2824	14.4
LogicCoverage	1, 808	15	83	15	196	7.7	38	522	7	4512	11.7
MinMCCoverage	3, 252	14	53	14	150	9.3	22	434	1	3642	11.9
Total	31, 832	232	742	196	2,325		240	8,364	344	39,670	
Average						13.8					29.6

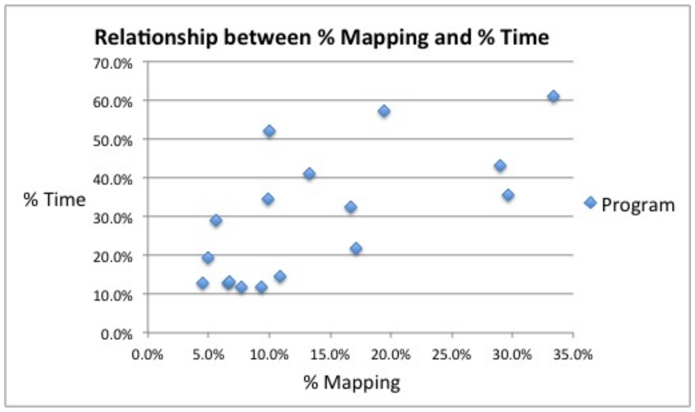

Fig. 4. Ratio of Number of Distinct Mappings over Number of Mappings in All Tests

V. CONCLUSIONS AND FUTURE WORK

This paper presents three results. The first is a general, practical solution to transforming abstract model-based tests to concrete executable tests when testers have only behavioral models. This is done using a test automation language, STAL, as described in section III. 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.

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.

Second, we developed a test automation language framework, STALE, which implements STAL. Testers can use STALE to accept models, programs, and mappings and then automatically create fully executable concrete tests.

This paper also compares test generation using STALE 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, on average. The manual tests also contained 48 errors in which concrete tests do not map abstract tests.

In the future, we would like to extend STALE to accept more diagrams such as UML activity diagrams and use more programming languages such as C++. Moreover, we will be looking for the possibilities to use this framework in the real world.

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.

[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] A. Eriksson, B. Lindström, and J. Offutt, "Transformation rules for platform independent testing: An empirical study," in *Proceedings of IEEE 6th International Conference on Software Testing, Verification and Validation*, ser. ICST'13, Luxembourg, Luxembourg, 2013, pp. 202–211.

[4] N. Li, F. Li, and J. Offutt, "Better algorithms to minimize the cost of test paths," in *Proceedings of IEEE 5th International Conference on Software Testing, Verification and Validation*, ser. ICST'12, Montreal, Quebec, April 2012.

[5] N. Li and J. Offutt, "An empirical analysis of test oracle strategies for model-based testing," in *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, ser. ICST '14, Cleveland, Ohio, USA, 2014.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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, September 2004, pp. 181–189.

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

[17] 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.

[18] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Formal methods and testing," R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Model-based Testing of Object-oriented Reactive Systems with Spec Explorer, pp. 39–76.

[19] A. C. Paiva, J. C. Faria, N. Tillmann, and R. A. Vidal, "A model-to-implementation mapping tool for automated model-based gui testing," in *Formal Methods and Software Engineering*, ser. Lecture Notes in Computer Science, K.-K. Lau and R. Banach, Eds. Springer Berlin Heidelberg, 2005, vol. 3785, pp. 450–464.

[20] F. Bolis, A. Gargantini, M. Guarnieri, E. Magri, and L. Musto, "Model-driven testing for web applications using abstract state machines," in *Proceedings of the 12th International Conference on Current Trends in Web Engineering*, ser. ICWE'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 71–78.

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

[22] 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.

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

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

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

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

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

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

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

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

[31] 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.

[32] 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.

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

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

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

[36] 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.

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

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

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

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

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