doi:10.1093/comjnl/bxh000

# Test Sequence Generation for Integration Testing of Component Software*

LEONARD GALLAGHER[1], JEFF OFFUTT[2]

[1]*Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA,* [2]*Information and Software Engineering, George Mason University, Fairfax, VA 22030, USA*
*Email: lgallagher@nist.gov, offutt@gmu.edu*

**Ensuring high object interoperability is a goal of integration testing for object-oriented software. When messages are sent, objects that receive them should respond as intended. Ensuring this is especially difficult when software uses components that are developed by different vendors, in different languages, and the implementation sources are not all available. A finite state machines model of inter-operating OO classes was presented in a previous paper. The previous paper presented details of the method and empirical results from an automatic tool. This paper presents additional details about the tool itself, including how test sequences are generated, how several difficult problems were solved, and the introduction of new capabilities to help automate the transformation of test specifications into executable test cases. Although the test method is not 100% automated, it represents a fresh approach to automated testing. It follows accepted theoretical procedures while operating directly on object-oriented software specifications. This yields a data flow graph and executable test cases that adequately cover the graph according to classical graph coverage criteria. The tool supports specification-based testing and helps to bridge the gap between theory and practice.**

*Received 30 August 2006; revised 24 August 2007*

## 1. INTRODUCTION

Many object-oriented (OO) applications are constructed from a combination of previously written, externally obtained components with some new components added for specialization. Source is often not available for the previously written components, yet new objects must interoperate via messages with objects in the existing components. This research is concerned with ensuring that objects inter-operate correctly, particularly when new objects are added to existing components.

In this paper, a *class* is the basic unit of semantic abstraction and each class is assumed to have *state* and *behavior* [1, 2]. A *component* is a collection of one or more classes that collectively implement a coherent function. Components inter-operate to implement the system and each component is assumed to execute independently, thereby allowing asynchronous behavior. For simplicity, we assume the interface to a component is just the union of the interfaces to its classes. An *object* is an instance of a class. Each object has state and behavior, where state is determined by the values of *variables* defined in the class, and behavior is determined by *methods* defined in the class that operate on one or more objects to read and modify their state variables. The *behavior* of an object is modeled as the effect the methods have on the variables of that object (the *state*), together with the messages it sends to other objects. Variables declared by the class that have one instance for each object are called *instance variables*, and variables that are shared among all objects of the class (static in Java) are *class variables.* This research is independent of programming language and the paper uses a mix of Java and C++ terminology.

Our test scenario is that a new or modified component is being integrated into an existing collection of

*This paper is an extended version of *Automatically Testing Interacting Software Components*, published in the *Workshop on Automation of Software Test (AST 2006)*, pages 57-63, May 2006, Shanghai, China.

components. Thus we are testing how the new component interacts with the existing components.

Behavior of objects is captured as a set of transition rules for each method and described as finite state machines that are extended with transition triggering information [3, 4, 5]. A transition is triggered by a call to a method with a particular *signature*, and is comprised of a source state, a target state, an event, a guard, and a sequence of actions. *Events* are represented as calls to member functions of a class. A *guard* is a predicate that must be true for the transition to be taken; guards are expressed in terms of predicates over state variables (possibly from multiple classes) and input parameters to the method. An *action* is performed when the transition occurs; actions are usually expressed as assignments to class member variables, calls sent to other objects, and values that are returned from the event method. A sequence of actions is assumed to be a block of statements in which all operations are executed if any one is executed.

Pre-conditions and post-conditions of methods can be derived directly from the transitions. The pre-condition is a combination of the predicates of the source state and the guard; the post-condition is the predicate of the target state. Note that the post-condition derived from a transition is not the strongest post-condition. If the tester desired, state definitions could be more refined, allowing stronger post-conditions, which would yield larger graphs and more tests. Whether to do so is a choice of granularity that results in a cost versus potential benefit tradeoff.

A *state transition specification* for a class is the set of state transition rules for each method of the class. Given a state transition specification for each class, the goal of this research is to construct *test specifications* that are used to construct an *executable test suite*. Hong et al. [6] developed a class-level flow graph to represent control and data flow within a single class. Our previous paper [7] extended their ideas to integration testing of multiple interacting classes.

The state transition specification is stored in a relational database consisting of six tables, as described in our previous paper [7]. The tables can be populated in a semi-automatic fashion from existing object-oriented finite state specification tools (such as the UML state machine). Some manual effort is currently needed to identify all the guards necessary to determine which transition is followed when a given method is invoked and which actions are performed as side-effects of that transition. All other database tables and operations mentioned in this paper are derived algorithmically from these six.

Transitions that are relevant to the component under test are used to create a *component flow graph*, which includes control and data flow information. Classical data flow test criteria are applied to this graph and converted to test specifications in the form of candidate test paths, and then to executable test cases.

In traditional data flow testing [8], the tester is provided with pairs of definitions and uses of variables (def-use pairs), and then attempts to find tests to cover those def-use pairs. This research stores information about the specification in the database, represents object behavior as branch choices in a directed graph, provides the tester with full def-use paths instead of just def-use pairs, and provides control mechanisms to construct calls of external methods that force traversal of the identified paths.

The rest of this paper describes our tool in more detail. Section 2 describes how the behavior of an OO component is modeled as a directed graph of states and transitions. Section 3 discusses how data flow criteria are applied to this model; in particular, how definitions, uses, and def-use paths are described. Section 4 describes how the test criteria are used to generate sequences of transitions to cover def-use pairs. Section 5 gives numerous details about how test sequences are automatically generated, and a detailed example is shown in Section 6. The architecture of our test environment is described in Section 7, related work is summarized in Section 8, and conclusions are given in Section 9.

## 2. BEHAVIOR IN A DIRECTED GRAPH

A *combined class state machine* represents the variables, methods, parameters, states and transitions of all state transition specifications for all classes in a system of interoperating components. If a specific component is identified as the *test component* in this system, then transitions from the classes in the test component form the basis of transitions that are *relevant* to that component. Relevant transitions that call *mutator* methods in other components of the system represent outward data flow, whereas transitions that call *actor* functions in other components represent inward data flow. The collection of all transitions in other components that have a method called by a relevant transition become relevant transitions to the test component. The transitive closure of this process defines the collection of *relevant transitions* for the test component.

A *component flow graph* represents all data and control flow relevant to a test component. It is a directed graph derived from the relevant transitions as follows:

- Every relevant transition is a *transition node*
- Every state that is either a source state or a target state of a relevant transition is a *state node*
- Every non-trivial guard of a relevant transition is a *guard node*
- There is a directed edge from every guard node to its corresponding transition node
- There is a directed edge from every transition node to its target state

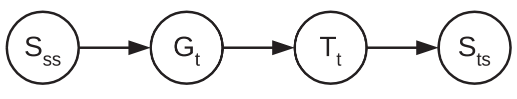

**FIGURE 1.** Path Derived From a Relevant Transition

- There is a directed edge from the source state node of a transition to either the guard node of that transition or to the transition node itself if the guard is trivially true

In general, every relevant transition $t$ produces a path in the component flow graph from its source state node, through the guard and transition nodes, to the target state node. This is shown in Figure 1.

The path in Figure 1 represents control flow from the state that an object is in when a method event is invoked, to the guard node that evaluates to true, to the action of the transition with that guard, to the target state of the transition. Actions on variables are also represented by transitions with associated get and set methods. Transitions with get (or actor) methods typically do not have non-trivial guards, and never change the state of an object, so usually produce paths in the component flow graph only from state nodes where the method is defined through the transition node and back to the same state node.

In multi-class systems, state and guard predicates are allowed to call actor functions from other classes and the action of a transition is allowed to invoke actor or mutator functions in other classes. Control and data flow resulting from these actions is represented by edges that connect nodes from different classes, as follows:

- If a state or guard predicate, or the action of a transition, calls an actor method in another class, there will be an edge from every transition node of the called method back to the node representing the calling state, guard or transition.
- If the action of a transition calls a mutator function in another class, there will be an edge from the transition node that represents that transition to the source state node of every transition of the called method in the other class. In addition, if the mutator method returns a value, there will be an edge from every transition node of the called method back to the transition node of the action that makes the call.

To illustrate how the component flow graph represents data and control flow across communicating objects, consider three objects, $obj1$, $obj2$ and $obj3$, which are instances of classes $A$, $B$ and $C$ (note that to make the example small, these components are single classes, whereas in general, the components can be much larger). Class $A$ defines a transition $t_1$, which represents a mutator function $f()$, and whose action defines a variable $v$. This transition as represented in a component flow graph is presented in Figure 2. The transition node is $T_1$, the guard node is $G_1$, and $S_a$ and $S_b$ are the source and target state nodes. Transition nodes that define a variable are labeled, such as $def\ v$ on node $T_1$. Uses of $v$ are represented by points $u_i$ on edges and inside transition nodes. Both $S_a$ and $S_b$ use $v$ ($u_1$ and $u_3$) in their predicate definitions. Function $f()$ is invoked from some other transition, which is represented by transition node $T_x$. The get function for $v$ is represented by transition node $T_{va}$ if $obj1$ is in state $S_a$ when it is called, and by $T_{vb}$ if $obj1$ is in state $S_b$.

Consider $obj2$, and a transition $t_2$ (transition node $T_2$). The guard predicate for $t_2$ ($G_2$) calls the get function for $v$ in $obj1$. This get is represented by two edges from $obj1$, both labeled by $get\ v$. $G_2$ uses $v$ ($u_4$), and for it to evaluate to true, the mutator function $g()$ must be called by some other action (represented by $T_y$) when $obj2$ is in state $S_c$. The action of $T_2$ uses $v$ ($u_5$) and sends a message to $obj3$ by invoking the mutator function $h()$, passing the value of $v$ as a parameter. $Obj2$ changes to state $S_d$ as the target state of transition $t_2$.

Transition $t_3$ handles the function call of $h()$ when $obj3$ is in state $S_e$, if the guard predicate for $t_3$ evaluates to *true*. The guard of $t_3$ ($G_3$) uses the value of the incoming parameter $p$, which has the value of $v$, represented by use $u_6$. The action of $t_3$ uses the value in a computation ($u_7$). After the action of $t_3$ is complete, $obj3$ changes to the target state $S_f$.

In the figure, edges from transition nodes to state nodes that result from calling mutator functions in another class are labeled with the name of that function ($callf()$, $callg()$, and $callh()$). Function names are also put on edges from transition nodes to guard, state, and transition nodes if the edge is a result of a call to a get function in another class ($get\ v$). Edges from source state nodes to guard and transition nodes are also labeled with the function that is called ($f()$, $g()$, and $h()$). These labels are used to access metadata about the functions when the component flow graph is traversed.

In a general data flow graph, uses of a variable in a state predicate are represented by use labels on all edges leaving the corresponding state node ($u_1$, $u_2$, $u_3$); uses of a variable or parameter in a guard predicate are represented by use labels on all edges leaving the corresponding guard node ($u_4$, $u_6$). These are called *predicate uses*. Uses of a variable in the action of a transition are called *computational uses* ($u_5$, $u_7$). Uses of a variable in a passed parameter are called *parameter uses* ($u_6$, $u_7$).

To summarize, Figure 2 represents the data and control flow resulting from definition of variable $v$ by a call of function $f$ in $obj1$, the predicate and computational use of $v$'s value in $obj2$, the passing of that value as a parameter to function $h(p)$, and the resulting predicate and computational parameter use of $v$ in the guard and action of $t_3$ in $obj3$. This is a flexible

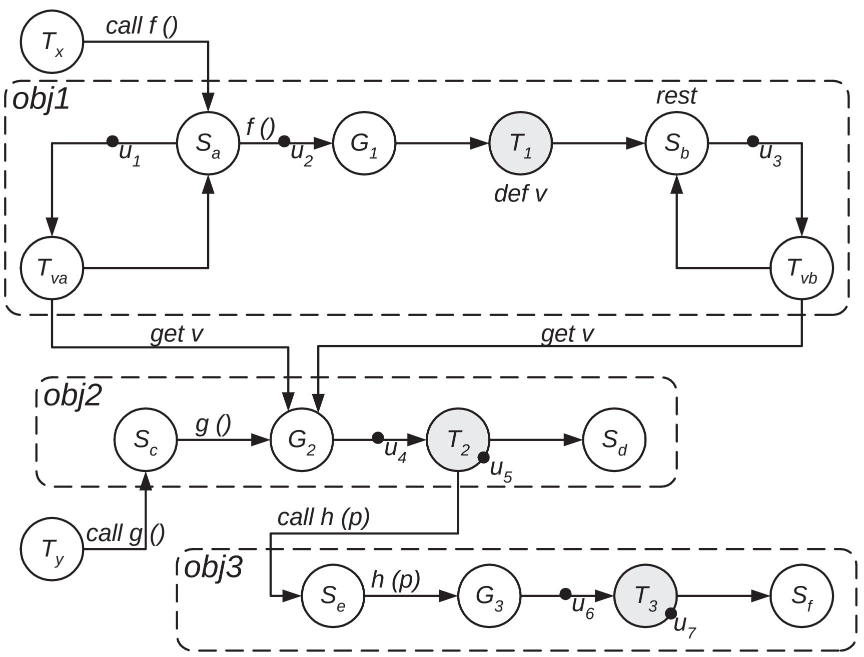

**FIGURE 2.** Portion of Component Flow Graph

model that can and should be adapted by developers. For example, if things such as counters are important to the FSM, they can be captured in the state predicates (we are currently employing this technique for another application). Space requirements and limitations are discussed in the previous paper [7].

How complete our model is will depend on how complete the design model is. We simply need to ensure that all metadata from the finite-state specification is captured. The database representation covers 100% of the state transitions identified in the source specification, including the guard predicates that are pre-conditions for each transition, and the actions (i.e. variable definitions and uses and messages sent to other objects) of each transition. If the initial specification covers all possible behaviors via defined state transitions, then this model captures them all. Thus, if each class in the source specification is defined as a state machine, with defined methods that produce state transitions under specific guard predicates and defined action blocks that describe actions that are triggered as part of the transition, then the database representation is equivalent to the source specification and all possible behaviors can be derived from that specification. The difficulty comes when portions of the source specification are textual sentences that must be represented more specifically as actions that can be processed, but this is common to all non-formal source specifications. One advantage of this approach is that once the methods and possible states are in the database, it is possible to do some rather rigorous testing to ensure that all possible combinations are addressed.

## 3. COVERAGE CRITERIA

Testing has long used data and control flow through programs [8]. Graphs are defined with nodes representing basic blocks, and edges representing branching statements in the program. A *definition* (*def*) of a variable $x$ is a node in which $x$ is given a value, and a *use* is a node in which that value is accessed, either through the same name or a different name via aliasing. Data flow graphs form edges from definition nodes to use nodes, where there is a def-clear control subpath from the def to the use in the control flow graph. A *def-clear subpath* for a variable x is a control subpath that does not have another definition of $x$. A *DU-pair* is a definition and a use of the same variable such that there is a def-clear subpath from the def to the use. A *DU-path* is a def-clear subpath from a specific definition to a use.

A number of different *coverage criteria* can be defined on data flow graphs, including *all-defs*, *all-uses* and *all-paths*. These have been discussed and compared extensively in the literature [8, 9, 10, 11, 12]. The object-oriented testing methodology described here follows the lead of other researchers and focuses on defs and uses of class variables in each object. This allows a tester to focus on testing criteria that require traversal of a def-use path in the component flow graph from a transition node that defines a variable to a node or edge that uses the variable, with no re-definition of the variable at any node along the path.

The all-uses testing criterion applied to a component flow graph requires tests to execute at least one path from each definition of a variable at a transition node to each reachable use at another node or edge. In Figure 2, applying the all-uses criterion to variable $v$ in class $A$ requires tests that will force execution of *def-use* paths from transition node $T_1$ to each of the uses $u_1$ to $u_7$. In the portion of the component flow graph pictured, one sees that uses $u_1$ and $u_2$ are not reachable from $T_1$ and that any path from $T_1$ to $u_7$ will include subpaths from $T_1$ to each of the uses $u_3$, $u_4$, $u_5$ and $u_6$.

In general, it is desirable to find two different kinds of paths in the component flow graph. One is a *def-use* path from the definition of a variable to a use that includes as many other uses as possible. Another is an *ext-int* path from an external transition that can be executed by a tester, which in turn forces execution of other, possibly internal transitions that need to be executed as part of a def-use path. Both types of paths are shown in Figure 2. The path $T_1 : S_b : T_{vb} : G_2 : T_2 : S_e : G_3 : T_3$ is of the def-use type and paths $T_x : S_a : G_1 : T_1$ and $T_y : S_c : G_2 : T_2$ are of ext-int type. In the def-use path, execution of transition $T_1$ defines $v$ and moves $obj1$ to the *rest state* $S_b$. Then execution of transition $T_2$ gets the value of $v$ from $obj1$ and forces traversal of the remainder of the path. If the methods of $T_1$ and $T_2$ are internal functions that cannot be executed directly by a tester, then one must be able to find *external* functions that can be executed by a tester in a sequence that forces traversal of the def-use path.

Construction of def-use paths must satisfy the following properties:

- Paths must respect the order of execution of statements in the action of a transition
- Function labels on adjacent transition-to-state-to-guard edges must be identical
- Guard predicates must be feasible with respect to passed parameters and current state
- Paths that exit and then re-enter a class must satisfy a *state compatibility rule* to ensure that the state after exit is identical to the state of re-entry

The *state compatibility rule* requires that if a path leaves a transition node $T_1$ from class $A$ to go to a node derived from some other class, and if the path later returns to the same or another transition node $T_2$ of class $A$, then the target state of $T_1$ must be equal to the source state of $T_2$. This requirement ensures that the action that causes class $A$ to change state is captured as part of the path.

Construction of ext-int paths must satisfy all of the properties of def-use paths, but in addition must not have any rest states that would require additional user actions to traverse the path. This ensures that execution of the identified external transition triggers successive actions that result in execution of the identified internal transition.

Our previous paper [7] includes an algorithm that allows construction of both types of paths. Given the set of all def-use pairs in a component flow graph it is possible to partition that set as follows:

- Pairs for which a *def-use* path can be constructed
- Pairs in which the *use* is provably *not reachable* from the *def*
- Pairs that remain *unresolved* and for which it is still unknown whether there exists a feasible, *def-free*, path from the *def* to the *use*

The def-use paths generated are test specifications for the all-uses criterion over the component flow graph. Using the ext-int paths, it is then possible to generate executable test cases that result in traversal of the test specifications. *Coverage* is determined by the number of def-use pairs that can be resolved by a def-free traversal of a test specification from the identified def to its use.

## 4. TEST SPECIFICATION COVERAGE

Given test specifications for the all-uses criterion over a component flow graph, it is desirable to construct a sequence of externally executable functions that when executed will cause traversal of as many of the underlying def-use paths as possible. A tester begins with each object of the software system in some initial state. The initial states determine which external transitions may be executed by calling various external functions.

As an illustration of the methodology, consider the three objects that determined the portion of the component flow graph in Figure 2. To develop a test sequence for the def-use path $T_1 : S_b : T_{vb} : G_2 : T_2 : S_e : G_3 : T_3$, a tester begins with objects 1, 2 and 3 in some initial states $S_1$, $S_2$ and $S_3$. The goal is to execute a sequence of external functions $\{Fi|i = 1, 2, ...\}$ that will properly place each object into states that support execution of the ext-int paths $T_x : S_a : G_1 : T_1$ and $T_y : S_c : G_2 : T_2$. The desired sequence of actions and effects can be seen in Figure 3.

Figure 3 is called a "feather graph," because there is a primary path (from the def at $T_1$ to the use at $T_3$), and external paths are needed to put the objects represented by the primary path into the proper states. These external paths "feather" in to the primary path and are essential to ensure controllability [13] of the system under test. First, if $obj3$ is not already in state $S_e$, some external function $F_1$ must be executed to move it from initial state $S_3$ to state $S_e$ through some transition $T_4$. Next, if $obj2$ is not already in the source state $S_y$ for transition $T_y$, some external function $F_2$ must be executed to move it from initial state $S_2$ to state $S_y$ through some transition $T_5$. This must be done while keeping $obj3$ in state $S_e$. Next, if $obj1$ is not already in the source state $S_x$ for transition $T_x$, some external function $F_3$ must be executed to move $obj1$ from its initial state $S_1$ to state $S_x$ through some transition $T_6$. Then external function $F_4$ can be executed while $obj1$ is in state $S_x$ to invoke transition $T_x$, which in turn calls function $f$, which results in transition $T_1$, which defines

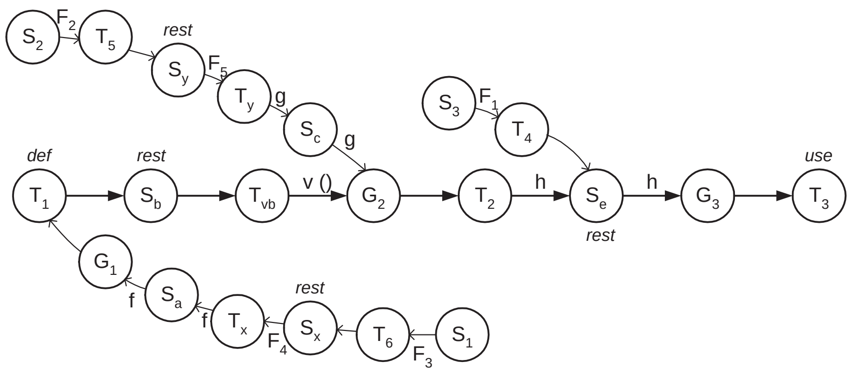

**FIGURE 3.** External Function Calls to Traverse a Def-Use Path

variable $v$ and puts $obj1$ in state $S_b$. State $S_b$ is defined to be a *rest state* for the given def-use path; another external method must be invoked by the tester to force continued traversal of the path. Rest states for this and other paths are labeled in the figure. Next, external function $F_5$ can be executed while $obj2$ is in state $S_y$ to invoke transition $T_y$, which in turn calls function $g$ that results in transition $T_2$. The action of transition $T_2$ gets the value of $v$ from $obj1$ and then sends a message to $obj3$ that calls function $h$ and completes traversal of the desired def-use path from $T_1$ to $T_3$. Note that the def-use path $T_1 : S_b : T_{vb} : G_2 : T_2 : S_e : G_3 : T_3$ has only one rest state $S_b$; the other state node in this path, $S_e$, is a rest state for a different path, i.e. $S_3 : T_4 : S_e$. The recognition of rest states for a given path in the component flow graph is an important part of test case generation.

The generated test sequence is $F_1, F_2, F_3, F_4, F_5$. Care must be taken to choose input parameters carefully to ensure that guard predicates will be satisfied and that the desired transitions will be executed. We have developed tools to automate the construction of test sequences, as described in the following sections.

## 5. TEST SEQUENCE GENERATION

If an object-oriented software system is represented by a *combined class state machine*, and if a specific component of that system has been identified for integration testing, then the processes described in Sections 2 and 3 can be used to construct a *component flow graph*, a set of *def-use* candidate test paths, and a set of *ext-int* transition triggering paths. Section 4 gives an example of constructing a single sequence of external function invocations to force the traversal of a single *def-use* path. This section describes a more general approach for constructing *test sequences* that force coverage of as many *def-use* paths as possible, thereby determining coverage results for integration testing of the identified component with the remainder of the software system.

Let $CTP$ be the set of all def-use paths and $EXT$ the set of all ext-int paths generated from a component flow graph. $CTP$ is the set of all *candidate test paths* that determine the test specifications for integration testing of the selected component. Let $F$ be the set of all external functions defined in the combined class state specification that could be called in black box testing. Functions in $F$ make up the external interface to the software system and are available for execution as part of a test suite. The goal of this research is to identify a sequence of functions $F_i$ from $F$ that when executed in the identified order will cover as many paths in $CTP$ as possible.

Given a software system consisting of multiple objects, its *system state* at a given point in time is defined to be the state of each object at that time. Suppose a program is in some initial system state $SS_0$ and suppose an external function $F_1$ is executed with some choice of values for any of its input parameters. Depending on the object states identified by $SS_0$, and depending on the values chosen for any input parameters, traversal of a subset of paths in $EXT$ will be initiated. Since none of the paths in $EXT$ have any rest states, if the guard predicates are satisfied each path will be traversed to its ending transition node. By construction of the $EXT$ paths, these ending transition nodes may be the head definition nodes of a number of def-use paths in $CTP$, each of which will be initiated and traversed up to its first rest state. If a path in $CTP$ has no rest states, then the entire def-use path will be completed and one can conclude that execution of $F_1$ covers any such path for which all of its guard predicates evaluate to *true*. At the completion of all triggered transitions with compatible guards, the software system will be in a new system state $SS_1$. The test generation module records both the function and the new system state. Any system being tested should be in the same system state; if not, it fails the test at this point in the test sequence.

The tester then executes a second external function $F_2$ to initiate another subset of ext-int paths from $EXT$.

Some of these paths may then, in turn, initiate a set of new def-use paths from $CTP$. In addition, every execution of a transition in the middle of an ext-int path, or in the middle of a leg between rest states of a def-use path, may call functions that trigger the next transition in a previously initiated def-use path that is currently in a rest state. Figure 4 presents the possibilities of choosing successive functions from $F$ to initiate new $EXT$ paths, which in turn initiate new $CTP$ paths or force traversals along existing $CTP$ paths to the next rest state. Rest state nodes in a def-use path are labeled $r$, and transition nodes that force a traversal from a rest state to a new *leg* in a def-use path are labeled $t$. The test generation module identifies and records all of this intermediate action and is able to predict the new system state at the conclusion of all triggered events. The state of each object in a system being tested must not fail the state predicate of any state in the resulting system state.

An external function $F_i$ may initiate multiple new paths in $EXT$, and each $EXT$ path may initiate or extend multiple paths in $CTP$. At each iteration, these new, *potential*, test paths are added to the existing partially traversed paths that remain active from previous iterations. Let $ATP$ be the collection of all such *active test paths* at this stage of test sequence generation. This set and its associated metadata drives subsequent test sequence generation. The test generation module identifies, records and maintains the following metadata for every path $p$ in $ATP$:

- Whether $p$ is a new path, directly generated from execution of the new $F_i$, or an existing path in a rest state from a previous iteration
- For new paths, the point in the path where the $EXT$ path joins up with the $CTP$ path
- For paths in a rest state, the *prior action* that caused the previous leg to be traversed, the location of the rest state node in the path, and the *next action* that if executed would cause traversal of the next leg
- The variable $v$ defined by the $CTP$ portion of $p$, and the def-use pair that would be resolved by successful traversal of $p$.
- Markers to indicate the current position in $p$, and identification of the sequence of transitions that would be executed to reach the next rest state in $p$
- Indicators to determine if $p$ is *pushed* along by execution of the *next action* or *pulled* along by the reading of some exposed data from the object associated with the current rest state

For any path in $ATP$, the path segment from the current position in the path to the next rest state is defined to be the current *active leg*. The test generation module helps the human tester analyze, very carefully, the sequence of all transitions in the *active legs* of all paths in $ATP$. This step currently requires manual intervention from the tester, who can use domain knowledge and analyze the predicates in ways that are currently beyond the ability of the tool. Many of the paths in $ATP$ will be mutually inconsistent. Some will have conflicting guard predicates, some will violate the *state compatibility rule* for paths, and some will redefine a variable that has already been defined by the definition node at the head of the $CTP$ portion of the path.

The process begins with all transitions triggered by execution of an external function $F_i$ and considers, in sequence, the transitions in the current active leg of all paths triggered by $F_i$. Analysis at this step of test sequence generation will remove from further consideration any paths in $ATP$ that would produce inconsistencies in the parallel traversal of all current active legs. Whenever two or more paths identify alternative choices for guard predicates to be satisfied before traversing to the next transition, the test generation module presents these choices to the tester and, based on the then current system state, helps the tester decide which transition choices are feasible at this point in the execution. The human tester merely responds to choices about which guard predicates are to be satisfied at each step of the process; the tool presents only feasible choices and handles all other aspects of the process. The tester decides which transition alternatives to pursue, then the test generation module takes the following actions:

- Delete all paths from $ATP$ that have transitions in the current active leg with guard predicates that are inconsistent with the guard predicates of the chosen transitions
- Identify and delete any paths in $ATP$ that violate the state compatibility rule
- Delete any paths currently in a rest state whose next action is inconsistent with any triggered transition up to this point in the analysis
- Identify and delete any paths in $ATP$ that are currently at a rest state, but for which the triggered actions of the current external function result in redefinition of the def variable
- When a path is fully traversed, mark the $CTP$ segment as *executed*; for optimization of choices, subsequent calls of external functions emphasize initiation of $CTP$ paths that are not yet executed.
- Delete all completed paths from $ATP$
- If a path in $ATP$ is currently in a rest state, but its next action is triggered by the next transition of some existing active leg, then *activate* the next leg of that path

Deleted def-use paths are not lost forever. They may reappear later after execution of a subsequent external function, possibly with different input parameter values, and with selection of new $EXT$ paths that may reinitiate the desired path. This time, different transition choices may cause complete traversal to the use node of this def-use path.

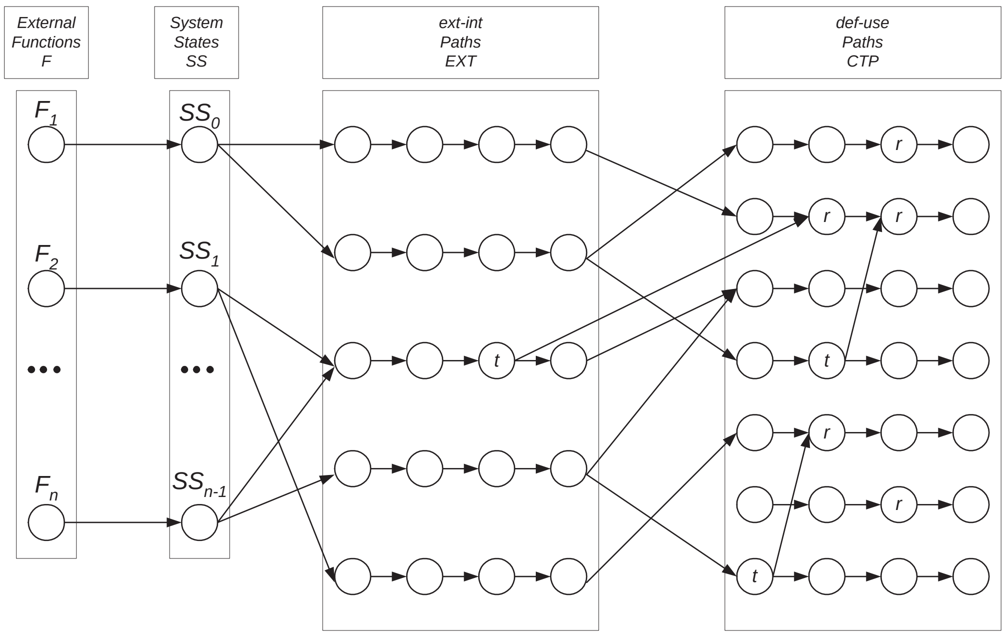

**FIGURE 4.** Generating an External Test Sequence

For each execution of an external function $F_i$, the test sequence generation algorithm iterates through all the active legs in $ATP$ until all remaining paths are at a rest state. At this point the module records the current system state, $SS_i$, as the state that each object in the system should reach after considering all possible actions triggered by $F_i$. A system under test passes test $F_i$ if each object in the system is in a state consistent with the recorded system state. With assistance from the application, the human tester will then choose a new external function $F_{i+1}$ to execute as the next test function in the test sequence, and the process continues.

The algorithm for test sequence generation helps a human tester make the above choices in a way that allows maximal coverage of untraversed def-use paths. It considers every external function that could be executed during the current system state, $SS_i$, and counts the number of feasible, untested paths in $EXT$ x $CTP$ that might be initiated and the number of paths in $ATP$ that might be progressed along the next leg. With knowledge of these counts, a human tester can choose the next external function $F_{i+1}$ that has the best possibility of maximizing these counts toward covering the most remaining untraversed def-use paths. Testers who have more knowledge and experience will naturally choose more effective functions. However, even inexperienced testers who make uninformed choices with respect to choosing from among conflicting guard predicates should still get useful test cases because incompatible paths are automatically eliminated from further consideration. Further research may help automate this existing human role in test sequence generation.

After completing the analysis of the triggered effects of each external function call $F_i$, the test generation module adds $F_i$ to the sequence of test functions being generated and records the following two pieces of information:

- The set of def-use paths covered by $F_i$
- The expected system state $SS_i$

The construction of test sequences continues until as many def-use paths as possible have been completely traversed from def to use. In some cases this may involve construction of multiple test sequences, each starting with a different initial system state. In other cases, a single test sequence may suffice to cover all def-use paths in $CTP$. In either case, the set of executable test sequences generated by this process determines a *test suite* for integration testing of the test component with the remainder of the system. Each test sequence in a test suite is executed against the implementation. After the triggering effects of each $F_i$, the implementation should be in the system state predicted by $SS_i$. If an implementation fails to be in a system state compatible with $SS_i$ it fails the test sequence. An implementation passes the test suite if each test sequence in the test suite is completed with no test sequence failures.

## 6. CRUISE CONTROL EXAMPLE

As an example of how the test generation module helps a human tester construct a test sequence, consider the Cruise Control specification from our previous paper [7]. The Cruise specification consists of multiple objects, the most important of which are: CruiseUser, CruiseUnit, Throttle, Engine, Gauges, and Wheel. The CruiseUser models the human controls available for Cruise and includes an On/Off toggle switch, a Cancel button, and Set/Decel and Resume/Accel buttons. The CruiseUnit specifies cruise control logic; it receives messages from CruiseUser and sends messages to the Throttle, which in turns sends messages to the Gauges and the Engine. The Engine gets Gear and DriveRatio information from the Transmission and sends messages to the Wheel. The Wheel simulates a Differential, which allows it to respond gradually to increases or decreases in axelRpm as set by the Engine. The Wheel then sends Speed information to the Gauges. The CruiseUnit reads Speed from the Gauges and takes any appropriate actions. The Engine object has an ExternalDrag variable to simulate wind and hills; this variable can be controlled by the tester to simulate those conditions.

When modeled as interacting finite state machines, cruise control is one component of a larger automobile system consisting of six components and twelve classes: Acceleration (GasUser, Throttle, Transmission, Wheel), Brakes (BrakeUser, BrakeControl), CruiseControl (CruiseUser, CruiseUnit), Engine (Engine), InstrumentPanel (Gauges), SystemControl (AutoSystem, Ignition). The combined finite state specification has 44 states, 106 functions, 58 state variables, 44 parameters, and 263 transitions, of which only 160 proved to be relevant for integration testing of the CruiseControl component with the other five components. Using the definitions of Section 2, this specification yields a *component flow graph* with 293 nodes and 740 edges. Then, applying the methods of Section 3, we found 188 locations where the action of a transition defines a state variable and 1015 locations where a variable is directly or indirectly used, yielding a total of 4319 distinct *def-use pairs* involving the 58 state variables. Using the all-uses testing criterion and the algorithms from our previous paper [7], we were able to generate 2372 candidate test paths covering 2063 def-use pairs while proving that 1494 def-use pairs had no feasible path from def to use, leaving only 762 def-use pairs for which it could not be determined if a feasible test path exists. Externally accessible components in the example system have 12 external methods that produce 24 distinct external transitions when considering different possible source states and guard predicates. The collection of candidate test paths has 160 distinct head nodes that define a variable, so it is necessary to again use the methods of Section 3 to construct ext-int paths for the potential 3840 external transition to internal CTP head node pairs. We found 4598 ext-int paths for 811 such pairs and were able to prove that the remaining 3029 pairs had no feasible ext-int path. Many of the generated ext-int paths were superfluous and only about one hundred were actually used in test sequence construction.

This methodology has a resource issue in that generation of the candidate test paths and the ext-int paths requires a significant amount of off-line storage of intermediate results. Generation of these paths was suspended when total database size exceeded 2 gB, even though not all def-use pairs or source to target nodes had been resolved; this explains the 762 unresolved def-use pairs mentioned above.

### 6.1. Building a Test Sequence

The goal at this point is to use the techniques described in Sections 4 and 5 to generate a sequence of external function calls that will link the ext-int paths to the candidate test paths, covering as many of them as possible.

Returning to the cruise control example, suppose the initial system state of the automobile system is that the automobile is parked and turned off. At that time there is only one feasible external test function; i.e. use the ignition to start the car. Inserting the key into the ignition causes AutoSystem to be created, which in turn creates several other components. This simple action covers 25 CTPs. All of the activated components should be in the predicted states, else the implementation fails the test.

Turning the key starts the car and activates the remainder of the components, covering 26 additional CTPs. At this point all of the automobile system components are active, there are 431 active candidate test paths, and the tester has several choices: use GasUser to increase the gas flow, adjust the controls on CruiseUser, put the Transmission into gear, and adjust the engine.ExternalDrag function to increase or decrease engine rpm. Any of these actions will result in coverage of additional CTPs, but counting how many of the active test paths will be advanced by these actions leads the tester to choose to increase the gas, covering an additional 25 CTPs and leaving 466 active test paths.

Again the tester has multiple options for external actions, but chooses to put the car in a forward gear, thereby covering 50 additional CTPs and leaving 507 active test paths. Continuing in this manner, the tester uses the metadata that is available at the end of each external call to make an informed choice of the most productive next action. At the end of each action, the components must be in the predicted states or the implementation fails the test. The tester always has the option to turn on cruise control, but that action by itself only covers one CTP; however, it adds a number of new active test paths and activates a number of sleeping paths that were in rest states. At each subsequent step the tester now has options to turn off the engine, turn off cruise control, push any of the buttons on CruiseUser, or accelerate to highway speed. The tester chooses to click the Cancel, Set/Decel and Resume/Accel buttons on CruiseUser to ensure that they do nothing while the cruising speed is still below the highway threshold. These actions cover a small number of additional CTPs. Finally the tester uses GasUser.position to increase the gas flow and bring the automobile to highway speed. At this point the tester has successfully covered 215 CTPs and there are 550 active test paths at various rest states.

The metadata on active test paths now shows that it is a good time to put the system into cruise control. First the tester pushes the Set/Decel button down, and then, as a second action, releases it to let it spring back to its neutral position. The first action only covers 4 CTPs, but adds 9 new active test paths. The second action, to release the button, adds 272 new active test paths and covers 57 CTPs, leaving the automobile cruising at highway speed with the Throttle in an automatic state and under the control of CruiseUnit, and with 619 active test paths at various stages of coverage.

When turned On, the CruiseUnit can be in one of four states: Cruise, Accel, Decel, and Override. While in the Cruise state, a Cancel() message from CruiseUser or from AutoSystem (after sensing Brake or Clutch actions or a Danger such as excessive speed or overheating) can cause a transition to the Override state. A Set/Decel or Resume/Accel message from CruiseUser will cause CruiseUnit to transition to the Decel or Accel states; acceleration or deceleration will continue until the user releases the button, causing a return to the Cruise state at a new cruise speed, or until a threshold speed is reached or some Danger is detected by another object, causing a transition from Accel or Decel to Override. While in the Cruise state, the CruiseUnit keeps reading the speedometer and sending Checkstate() messages to itself to determine if any adjustments are needed to maintain the cruise targetSpeed.

When the Ignition is On, the Throttle can be in any of the states Idle, Automatic or Manual. The Automatic state is achieved when the throttle Floor and Position variables are equal and above idle position, probably meaning that the throttle is under the control of CruiseUnit. The AutoSystem object can also set the throttle Floor variable, which it does when the car is warming up. The GasPedal sets the throttle Position variable and Throttle is in the Manual state when Position > Floor. Throttle transitions into the Idle state whenever Position is less than or equal to an idle threshold position.

Since the Wheel also simulates a Differential, it can be in Accel, Decel, or DirectDrive states. A message from the Engine will set a new axelRpm, which causes the Wheel to go into Accel or Decel states. While in either of these transition states, the Wheel incrementally changes its wheelRpm and repeatedly sends Checkstate() messages to itself until wheelRpm reaches the target axelRpm. This then causes a transition back into the DirectDrive state and the Wheel stops sending Checkstate() messages to itself.

The issue at this moment in test sequence generation is to determine the *best* external function to execute as the next test in the test sequence in order to cover the maximal number of untested paths in $CTP$. The tester can request that the test generation module show different views and counts of the active test paths in $ATP$. One view of $ATP$ may now show a list of partially traversed paths whose next action will be the final action of that path. Examples of final actions may be new Danger circumstances that will cause Cancel() messages to be sent to CruiseUnit, or transitions that do not normally happen such as using two hands to simultaneously push multiple buttons on CruiseUser, or turning the cruise switch Off while CruiseUnit is in the Accel or Decel states.

Figure 5 shows a series of possible functions and transitions. Suppose the view of $ATP$ shows that a maximal number of paths will be completed, and a number of other paths extended to a new rest state, if CruiseUnit transition c05t026 is executed. This transition occurs if CruiseUnit is in the Cruise state and if the function Checkstate() is executed while the guard predicate defined by CurrentSpeed > TargetSpeed and Throttle.Position()=Throttle.Floor() is satisfied. The metadata for $ATP$ also shows that the $CTP$ segments for all of these paths begin with definition of the Position variable in Throttle (by various different functions) and end with a use of Position in the Guard predicate of transition c05t026. Since Checkstate() is constantly calling itself while CruiseUnit is in the Cruise state, the tester only needs to find a way to increase the CurrentSpeed without redefining the Position variable.

The tester can request another view of $ATP$, which does a cross product with $EXT$, to determine all of the externally executable functions that will trigger transition c05t026 when executed in the current system state. The metadata for this result shows that ExternalDrag(x) in the Engine object should cause an increase in Gauges.Speed that will satisfy the guard predicate of c05t006 and complete all of these paths without redefining Position in Throttle. Based on this evidence, the tester wishes to add ExternalDrag(x) to the test sequence. The test generation module then leads the tester through the eight iterations displayed in Figure 5. At each iteration, the module presents all choices that may be feasible at that point in the execution of transitions in the active legs of paths in $ATP$. With knowledge of the current state, the tester is able to choose feasible transitions that will successfully complete the identified paths, and possibly pick up several additional paths along the way. The test generation module will record all of the state transitions along the way to determine the resulting system state after execution of this test case.

The tester knows that a decrease in Engine drag will cause an increase in speed. So in the initial iteration, the tester chooses input parameter x for ExternalDrag(x) to be less than the existing drag to increase the speed just enough to be recognized by the CruiseUnit, but not so much as to trigger any excessive speed or other Danger conditions. As shown in Figure 5, the Engine reads information from the Transmission and sends messages to both Gauges and Wheel.

| Id | Class | Function | Transition | Guard | Action | Chose |
|---|---|---|---|---|---|---|
| **0** | Engine | ExtDrag(-) | c07t005 | true | Sets drag variable; sends new engine rpm to Gauges; gets driveRatio from Transmission; sends new axelRpm to Wheel to increase axelRpm | yes |
| **1** | CruiseUnit | Checkstate() | c05t023 | Speed very close to target speed. | Sets currentSpeed = Gauges.Speed(); puts Checkstate() on the call queue | yes |
|  |  | Checkstate() | 4 others | Other conditions | Varies | no |
|  | Gauges | Tach(x) | c09t006 | true | Sets Gauges.tach in rpm. | yes |
|  | Wheel | AxelRpm(x) | c13t004 | x > current axelRpm | Sets target axelRpm; increases Wheel rotation incrementally; sends higher speed to Gauges; calls Wheel.Checkstate() to see when target reached; Wheel enters Accel state | yes |
|  |  | AxelRpm(x) | c13t003 | x < current axelRpm | Sets target axelRpm; decreases Wheel rotation incrementally; sends lower speed to Gauges; calls Wheel.Checkstate() to see when target reached; Wheel enters Decel state | no |
| **2** | CruiseUnit | Checkstate() | c05t023 | Speed very close to target speed. | Sets currentSpeed = Gauges.Speed(); puts Checkstate() on the call queue | yes |
|  |  | Checkstate() | 4 others | Other conditions | Varies | no |
|  | Gauges | Speed(x) | c09t008 | x < dangerSpd | Sets Gauges.speed in km/hr | yes |
|  |  | Speed(x) | c09t009 | x >= dangerSpd | Set speed to maxSpd; send Danger warning; enter Danger state | no |
|  | Wheel | Checkstate() | c13t014 | Wheel rotation increasing and not yet to target | Increases Wheel rotation incrementally; sends higher speed to Gauges; calls Wheel.Checkstate() | yes |
|  |  | Checkstate() | c13t015 | Wheel rotation reaches target rotation | Sends new higher speed to Gauges. Wheel goes into DirectDrive state. | no |
| **3** | CruiseUnit | Checkstate() | c05t026 | CurrentSpeed > TargetSpeed and Throttle in Automatic state. | Calls Throttle to decrease Floor and Position by incremental amount; Pause; call Gauges.Speed() to re-set currentSpeed; puts Checkstate() on the call queue | yes |
|  |  | Checkstate() | 4 others | Other conditions | Varies | no |
|  | Gauges | Speed(x) | c09t008 | x < dangerSpd | Sets Gauges.speed in km/hr | yes |
|  |  | Speed(x) | c09t009 | x >= dangerSpd | Set speed to maxSpd; send Danger warning; enter Danger state | no |
|  | Wheel | Checkstate() | c13t014 | Wheel rotation increasing and not yet to target | Increases Wheel rotation incrementally; sends higher speed to Gauges; calls Wheel.Checkstate() | no |
|  |  | Checkstate() | c13t015 | Wheel rotation reaches target rotation | Sends new higher speed to Gauges. Wheel goes into DirectDrive state. | yes |
| **4** | CruiseUnit | Checkstate() | Pausing from c05t026 |  | While Pausing, other calls could be put on call queue. For example, a user action from CruiseUser class. Finally, c05t026 puts another Checkstate() call on the call queue | yes |
|  | Gauges | Speed(x) | c09t008 | x < dangerSpd | Sets Gauges.speed in km/hr | yes |
|  |  | Speed(x) | c09t009 | x >= dangerSpd | Set speed to maxSpd; send Danger warning; enter Danger state | no |
|  | Throttle | Floor(x) | c10t005 | x <= minimum value<br>Tester decides can't happen because current cruise speed not close to this threshold. | Floor and Position set to minimum values; call Engine.GasFlow(minimum); Resets GasUser.PedalPosition(); Throttle goes into Idle state | no |
|  |  | Floor(x) | c10t009 | x > minimum value | Floor and Position set to new values; call Engine.GasFlow(Min(x,max)); Resets GasUser.PedalPosition; Throttle stays in Automatic state | yes |
| **5** | CruiseUnit | Checkstate() | Pausing or c05t026 | As above | As above | yes |
|  | Engine | GasFlow(x) | c07t003 | true | Re-sets GasFlow; calculates new Rpm and sends to Gauges; gets driveRatio from Transmission and sends new AxelRpm to Wheel | yes |
|  | GasUser | PPositin(x) | c08t004 | true | PedalPosition gets re-set by the Throttle when Throttle is in Automatic state | yes |
| **6** | CruiseUnit | Checkstate() | Pausing or c05t026 | As above | As above | yes |
|  | Gauges | Tach(x) | c09t006 | true | Sets Gauges.Tach in rpm | yes |
|  | Wheel | AxelRpm(x) | c13t004 | x > current axelRpm | As above. Wheel goes into Accel state. | no |
|  |  | AxelRpm(x) | c13t003 | x < current axelRpm | As above. Wheel goes into Decel state. | yes |
| **7** | CruiseUnit | Checkstate() | c05t026 or c05t023 or c05t027 | As above. Adjustment in Speed from #6 may be enough to regain targetSpeed; if c05t026 executed multiple times, may need to readjust with c05t027 to increase speed. | As Above.<br>What happens here is sensitive to the length of the Pause in c05t026. The current Speed may or may not oscillate around the TargetSpeed. | yes |
|  | Gauges | Speed(x) | c09t008 | x < dangerSpd | Sets Gauges.speed in km/hr | yes |
|  |  | Speed(x) | c09t009 | x >= dangerSpd | Set speed to maxSpd; send Danger warning; enter Danger state | no |
|  | Wheel | Checkstate() | c13t012 | Wheel rotation decreasing and not yet to target | Decreases Wheel rotation incrementally; sends lower speed to Gauges; calls Wheel.Checkstate() | no |
|  |  | Checkstate() | c13t013 | Wheel rotation reaches target rotation | Sends new lower speed to Gauges. | yes |
| **8** | CruiseUnit | Checkstate() | c05t023 | Speed very close to target speed. | Sets currentSpeed = Gauges.Speed(); puts Checkstate() on the call queue | yes |

**FIGURE 5. Active Leg Iterations for ExternalDrag Test Case**

In iteration 1, the tester is asked to choose from among all of the possible Checkstate() transitions in CruiseUnit. While the car is in the Cruise state, this function is continually called. Since the car is currently cruising normally, and there are no external influences (other than external drag), a transition is chosen that simply resets the currentSpeed and calls another Checkstate(). All other Checkstate() transitions are rejected. The Engine sets a new axelRpm for the Wheel that the tester knows will increase speed, so the tester chooses Accel instead of Decel. The tachometer is set in Gauges with no guard choices necessary.

In iteration 2 the speedometer speed has not yet changed, so the same Checkstate() alternative is chosen. The Wheel sends a new speed to the Gauges, forcing the tester to choose whether or not this new speed exceeds some danger speed. The tester knows it does not because the system state was not close to this threshold. If, instead, the system was close to this danger threshold, the tester could have chosen the c09t009 alternative resulting in a totally different analysis of paths in $ATP$. The tester does not know how quickly the Wheel will reach its new target rotation speed, but knows from previous experience that different paths taken by the Wheel at this point will have no effect on the final system state or on the eventual effects seen by CruiseUnit.

In iteration 3, the CruiseUnit finally picks up the new speedometer speed from the Gauges. The tester chooses the Checkstate() alternative that recognizes this increase in speed as sufficiently above the target speed to require an adjustment. The Gauges choices remain the same while the Wheel reaches its target rotation speed. The desired transition (c05t026) has been executed; the test generation module recognizes this and marks all of the identified $CTP$ paths as covered by this test case. The only thing left is to determine if any other paths are completed by these actions and what the resulting system state will be.

In iteration 4 the CruiseUnit has sent a correction message to the Throttle, which re-defines the Position variable. All paths in $ATP$ that are not already completed and that have Position as their definition variable are then deleted. Fortunately, the $CTP$ paths of interest were completed in the previous iteration and have already been marked as covered. The usage of Position takes place in the guard node that precedes the action of the transition node that redefines Position. The tester rejects the Throttle.Floor(x) alternative that would put the Throttle into Idle because the throttle floor and position were increased, not decreased.

Iteration 5 shows the effects of the Throttle sending a new gas flow message to the Engine and a new gas pedal position to the GasPedal. The tester cannot make any choices here. Depending on how long the CruiseUnit pauses in the action of c05t026 before rechecking speed, it may send multiple correction messages to the Throttle, but that has no effect on any of the candidate test paths. It may, however, cause excessive adjustments to the speed, resulting in further potential corrective actions by the CruiseUnit.

In iteration 6, the Wheel finally sees the correction initiated by the CruiseUnit. The tester knows that this will be a decrease in axelRpm, so chooses the Decel option over Accel. The tester faces the same CruiseUnit Checkstate() options as in previous iterations and chooses whether or not the CruiseUnit is still pausing from the original transition c05t026. If the CruiseUnit has stopped pausing, it will send another correction to decrease speed because the Wheel has not yet sent a new decreased speed to Gauges.

In iteration 7, the wheel has finally sent a new decreased speed to Gauges, which may be enough to cause the CruiseUnit to be satisfied, but may not be enough or may be too much. These corrective actions may continue while the speedometer speed oscillates above and below the targetSpeed maintained by CruiseUnit. The specification being tested does not prohibit such oscillation; it only requires that each object respond to method calls in the prescribed manner. Additional specification constraints could be added to require that the speed reach stability within a reasonable amount of time (adding a real-time dimension to the specification).

Iteration 8 shows that the effects of the change in engine drag have finally been fully achieved. The test generation module will have recorded that the CruiseUnit never left the Cruise state and that the Throttle never left the Automatic state. However, the Wheel moved from DirectDrive to Accel to Decel and then back to DirectDrive. The system under test passes this test case if after an appropriate period of time all of these objects are in the final predicted states. The system under test fails this test case if speed continues to oscillate or if the effects of these actions cause some other object in the system to change state.

Using the above methods, we were able to generate one test sequence consisting of 145 test cases that covered 1001 candidate test paths and 954 def-use pairs.

## 6.2. Fault Detection Effectiveness

To study the effectiveness of this approach for detecting faults, we created 108 faults for the initial automobile specification and then constructed two sets of tests to run against 108 implementations, each containing one seeded fault. The first test set was the single test sequence consisting of the 145 test cases generated using the finite-state methodology explained above. The second test set was a single test sequence manually constructed from intuitive use of cruise control and consisting of normal actions to start the car, go to highway speed, activate cruise control, brake to slow speed, resume to recover cruise speed, use ExternalDrag to maintain cruise speed up and down hills, turn cruise off, slow to slow speed, turn cruise on, test to make

**TABLE 1.** Basic Information

| Atttribute | Value |
|---|---|
| Components | 6 |
| Classes | 12 |
| States | 44 |
| State variables | 58 |
| Transitions | 263 |
| Relevant transitions | 160 |
| Component flow graph nodes | 293 |
| Component flow graph edges | 740 |
| Variable defined locations | 188 |
| Variable used locations | 1015 |
| Def-use pairs | 4319 |
| Path generated | 2063 |
| No path possible | 1494 |
| Indeterminant | 762 |
| Candidate Test Paths (CTP) | 2372 (covering 2063 pairs) |
| Ext-int pairs | 3840 |
| Path generated | 811 |
| No path possible | 3029 |
| Ext-int Paths (EXT) | 4598 (covering 811 pairs) |

sure cruise controls inactive at slow speeds, brake to stop, and turn ignition off. The intuitively generated test sequence consisted of 41 test cases.

Summarizing the analysis presented in our previous paper [7], we found that the test sequence generated using the finite-state methodology found 106 faults for 98% effectiveness, while the intuitively generated test sequence discovered only 24 of the faults for 22% effectiveness. Clearly this testing scenario favors the finite-state methodology because it generated more than three times as many tests as did the intuitive method. However, the manually generated tests require intensive human effort to decide what new expectations to test, while the finite-state approach initiates a dialog with the tester to determine potentially good actions to take at each step of the process, provides metadata to help the user make good choices, and creates new test actions to cover candidate test paths not previously covered.

### 6.3. Numeric Summary of Data

Tables 1 and 2 summarize the numbers in Section 6. Table 1 summarizes the numbers of the various attributes of the software studied, starting with the number of components and classes, on through the number of candidate test paths and ext-int paths.

Table 2 summarizes the results of running tests on the faulty versions of the example system. The first column shows data from the automatically generated tests and the second column shows data from the manual tests. The automated method resulted in many more tests, but because of the automation, they are cheaper to generate.

**TABLE 2.** Detection Effectiveness

| | Automated Method | Manual Method |
|---|---|---|
| CTPs covered | 1001 of 2372 | Unknown |
| Def-use pairs covered | 954 of 2063 | Unknown |
| Tests | 145 | 41 |
| Faults | 108 | 108 |
| Faults found | 106 | 24 |
| Percentage found | 98% | 22% |

## 7. TEST TOOL ARCHITECTURE

The test tool architecture is presented in Figure 6. The combined class state machine of the software system specification is represented in a database of six relational tables. The tables represent the **Classes**, **Variables**, **Functions**, **Parameters**, **States**, and **Transitions** of the combined class state machine. Upon choosing a component of the system for integration testing, the Test Sequence Generator constructs the component flow graph and follows the processes described in Sections 2 through 5 to generate one or more sequences $\{(F_i, SS_i)|i = 1, ..., n\}$ of executable external functions and predicted system states, both represented by Executable Test Sequences in the figure. The Test Harness imports an executable test sequence with predicted system states, executes each test against a claimed implementation of the specification, and determines if the implementation passes or fails the test sequence by checking the actual system states against the predicted states.

The Java Rapid Prototype Machine reads a database representation of a finite-state specification, and produces a generic test simulator written in Java. The machine consists of a simple kernel that is able to wait for, queue, and process input tasks from either a user or from the Test Harness. An input task is codified as an instance of a Java wrapper class that stores data fields traditionally associated with object-oriented programming, such as an objects identity, state, and behavior, applied to that object (a function). The object in question is an instance of a class defined by the tables of the database representation. An interpreter evaluates an input task and queries the database representation to simulate the actions of each defined transition. The resulting Java *reference implementation* provides an optional Graphical User Interface for test writers to add visual components for simulation purposes. The Test Harness is designed to support testers who want to run a sequence of test cases under the reference implementation, or against a real implementation inserted into the testing architecture in place of the reference implementation.

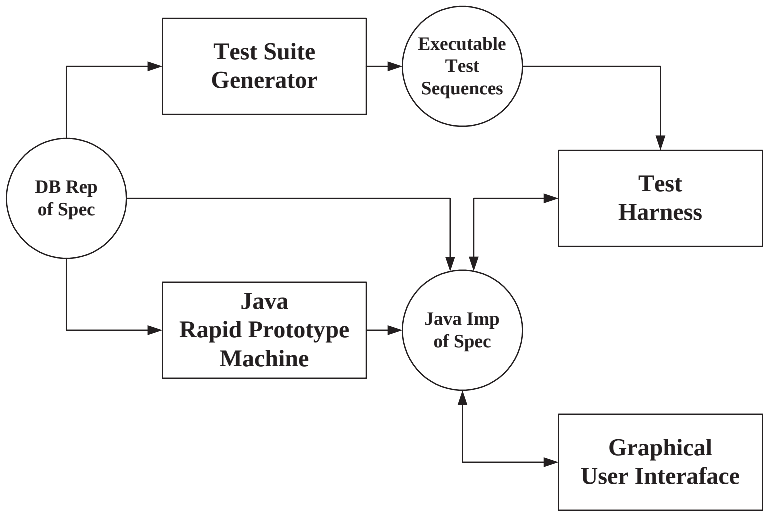
Test Suite Generator
Executable Test Sequences
DB Rep of Spec
Test Harness
Java Rapid Prototype Machine
Java Imp of Spec
Graphical User Interaface

**FIGURE 6.** Test Tool Architecture

## 8. RELATED WORK

*Intra-class* testing [14] is when tests are constructed for a single class, usually as sequences of calls to methods within the class. Integration testing attempts to test interactions among different classes; thus *inter-class* testing [7] refers to when more than one class is tested together.

This research project is specifically developing methods to carry out **inter-class** integration testing of object-oriented software. Although many papers have been published in **intra-class** testing, the authors have not found many papers on inter-class testing. This research also uses ideas from the rich literatures on data flow testing, intra-class object-oriented testing, object-oriented data flow testing, integration testing, and automatic test data generation. Some of the highlights of these areas are discussed below.

Testing has long used data and control flow through programs [8]. Several data flow coverage criteria have been defined, including all-defs, all-uses and all-paths. These have been described extensively in the literature [8, 15]. This research uses the all-defs and all-uses criteria, as have other researchers [16, 17, 18, 19, 20, 21, 22]. Although some may question the value of all-defs, others believe it to be useful and if all-uses is satisfied, all-defs comes for free anyway.

Data flow testing criteria [8, 19, 23] force tests to execute from data definitions to data uses in different ways. Most research papers have derived graphs directly from the code; which can be called *traditional* or *code-based* data flow testing. This paper defines data flow on finite state machines that are derived from the behavior of classes.

Most object-oriented testing research has focused on intra-class testing. This includes work by Hong et al. [6], Parrish et al. [24], Turner and Robson [5], Doong and Frankl [25] and Chen et al. [26]. Intra-class testing strategies focus on one class at a time, and do not address problems in the interfaces between classes, or in inheritance and polymorphism. In their TACCLE methodology [27], Chen et al. used algebraic axioms to define class semantics and construct test cases as paths through a state-transition diagram with path selection based on ground terms that are non-equivalent. They tried to extend this methodology to multiple classes by defining inter-class semantics in terms of contracts, however, this increases the complexity substantially and is difficult to re-use when other components are added to the system.

Harrold and Rothermel applied traditional data-flow analysis to classes [14]. Their approach emphasized (1) intra-method testing, in which tests are constructed for individual methods; (2) inter-method testing, in which multiple methods within a class are tested in concert; and (3) intra-class testing in which tests are constructed for a single class, usually as sequences of calls to methods within the class. Harrold and Rothermel represent a class as a Class Control Flow Graph (CCFG), which contains information that can be used during testing.

Hong et al. [6] developed a class-level graph to represent control and data flow for individual classes. Our research extends their ideas to perform integration testing on multiple interacting classes.

Some related work has been done on the subject of testing web software. Kung et al. [28, 29, 30] have developed a model to represent web sites as a graph, and provide preliminary definitions for developing tests based on the graph in terms of web page traversals. They define *intra-object* testing, where test paths are selected for the variables that have def-use chains within an object, *inter-object* testing, where test paths are selected for variables that have def-use chains across objects, and *inter-client* testing, where tests are derived from a reachability graph related to the data interactions among clients.

Inter-class testing, which this paper addresses, has seen much less attention. Most previous papers have used the program source. Yoon and Choi addressed the problem of integrating an externally procured class, for which the source is **not** available, into a pre-existing component, for which the source **is** available. Jin and Offutt [31] developed coupling-based testing, which requires tests to cover code-level control and data couplings between methods in different classes. Chen and Kao [3] describe object flow testing, in which testing is guided by data definitions and uses in pairs of methods that are called by the same caller. Testing should cover all possible type bindings in the presence of polymorphism. Kung et al. [32] address object-oriented testing of inheritance, aggregation and association relationships among multiple classes by automatically generating an object-relation diagram and finding a test in order to minimize the effort to construct test stubs.

Alexander and Offutt [33, 34, 35, 36] extended coupling to cover inheritance and polymorphism couplings. Alexander's research led to a categorization of inter-class OO software faults [37], which was then used to develop a collection of mutation operators for inter-class problems [38]. Ma, Offutt and Kwon developed a mutation testing tool, muJava, to carry out inter-class testing [39].

A related problem is that of deciding which order to test a group of classes, the class integration and test order (CITO) problem [40]. This problem is orthogonal to the problem of actually developing tests.

One of the hardest problems in testing has long been automatic test data generation (ATDG) [41, 42, 43, 44, 45, 46]. To automate the generation of test data according to test criteria requires solutions to very difficult analysis problems. This is often a hit-or-miss process, with the tester throwing test inputs at the software, hoping that the data flow system eventually reports that the DU-pairs were covered. It is sometimes very difficult for a tester to find a test case that will cover a particular DU-pair, and attempts have been made to generate tests by generating and solving predicates [47].

The general problem is undecidable. In the terms of this paper, ATDG will not always succeed because some def-use pairs cannot be satisfied by any candidate test path and the problem of finding executable test cases is generally undecidable. This has been called the feasible path problem in previous research [48, 49, 50, 51].

## 9. CONCLUSIONS

This paper presents technical details about an automated tool to support integration testing of object-oriented software. The assumed test scenario is that a new or modified collection of classes (a component) is being integrated into an existing collection of classes (such as a component or system). The classes are represented as interacting finite state machines, which model the information that is normally included in design models such as UML statecharts, and augmented with details about definitions and uses of class variables. This augmentation is currently done by hand, but this information could be obtained by a detailed analysis of the implementation.

The combined class state machine and component flow graph are new to this research, and this paper describes in detail how the behavior of OO software is represented in them and details for how they are constructed. The test criteria used are based on traditional data flow criteria, but applying them to these types of models is new and introduces numerous complexities. This paper describes how this model works with specific examples.

The use of a database to store definition/use information simplifies the construction of full def-use paths from definitions to uses. Storing and manipulating complete path predicates for traditional code-based data flow is impractical due to size. The database allows these potentially large predicates to be managed more efficiently.

As with any automated test systems, undecidable problems prohibit complete solutions. In this research, some candidate test paths cannot be resolved into executable test cases. This sometimes happens because resolving the test paths is too complicated, and sometimes because they represent truly undecidable portions of the problem space. This problem is common to all automatic test data generation techniques.

One advantage of this work is that potential concurrent actions among objects is fully captured in the component flow graph and in the generation of candidate test paths through that graph. The model assumes that receipt of asynchronous messages is handled by queuing and that any one of the received messages could be the next to execute. This ensures that all possible executions are considered by the candidate test paths. If certain concurrent executions are not feasible, that information will be detected during testing and the infeasible path segments can be removed in subsequent test path generation. More details of the concurrency and asynchronous aspects of this work are in the previous paper [7].

### 9.1. Future Work

Future research will focus on additional automation of existing manual steps in this methodology. One current direction is to provide automation for translating software specifications into the database representation. If the specifications are in a state-machine representation (such as UML statecharts), then much of the meta-data can be captured automatically. However, UML statecharts do not specify the details of the transition actions. Indeed, this information is seldom included in design models, so this may still need to be supplied by the human. Another possibility is to extract this information from the implementation. Although a challenging problem, it is feasible to analyze the source to determine what state variables are modified in each method, and use that to describe the transition actions.

Another target of more automation is the identification of infeasible path segments in the component flow graph, which should be avoided in def-use and ext-int path construction.

Section 5 emphasizes the goal of trying to "force coverage of as many def-use paths as possible." Without additional information about the specification it is not possible to know whether some def-use paths are more important to test than others. One approach might be to incorporate usage statistics into the model to determine "high-priority" edges in the component flow graph and to prioritize def-use paths depending on the number of high-priority edges they cover. This could result in more effective test sequence generation.

The cruise control system is modeled with 12 classes, 44 states, and 43 relevant variables that appear in 4319 def-use pairs, so although it is not large it is certainly non-trivial. We are currently assessing the utility of this approach on a larger and significantly different example, a healthcare standardization organization (HL7) that is defining hundreds of application roles that send and receive messages among themselves.

The information accumulated as part of this approach might also be useful during regression testing. At the end of test sequence development the database contains the original six tables that represent the specification, together with the derived component flow graph (293 nodes and 740 edges in the Cruise Control example), the candidate test paths (2372 in Cruise Control), and the sequences of test cases (145 tests with predicted system states in Cruise Control). Subsequent changes to the specification could result in changes to the component flow graph and the candidate test paths, and thus allow the test cases that cover those paths to be redefined. In addition, small additions or small modifications to the specification leave many of these parts unchanged, so it should be possible to automatically determine which tests need to be altered and re-run, and which tests do not.

Although the testing approach in this paper was intended for use during software development, a natural question is whether it could also be applied to test legacy software. The first requirement is that the software would have to use an OO design. The most significant effort involved in using this approach to test legacy software would be the effort required to represent (i.e. re-specify) the legacy system as a collection of communicating components, with each component specified in terms of states and transitions. Some systems may not lend themselves to the finite state model; however, for those that do, the effort should be worthwhile.

## ACKNOWLEDGMENTS

We would like to thank Tony Cincotta for implementing the rapid prototype machine and the test harness, and Julie Zanon, Hankim Ngo and Boris Etho-Assoumou for different features of a web application that implements the algorithms of the test suite generator.

## REFERENCES

[1] Booch, G. (1991) *Object-Oriented Design With Applications.* Benjamin-Cummings Publishing Co. Inc., Reading, MA.

[2] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991) *Object Oriented Modeling and Design.* Prentice Hall.

[3] Chen, M.-H. and Kao, M.-H. (1999) Testing object-oriented programs - an integrated approach. *Proceedings of the 10th International Symposium on Software Reliability Engineering*, Boca Raton, FL, November, pp. 73–83. IEEE Computer Society Press.

[4] Chow, T. (1978) Testing software designs modeled by finite-state machines. *IEEE Transactions on Software Engineering*, **SE-4**, 178–187.

[5] Turner, C. D. and Robson, D. J. (1993) The state-based testing of object-oriented programs. *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM-93)*, Montreal, Quebec, Canada, September, pp. 302–310. IEEE Computer Society Press.

[6] Hong, H. W., Kwon, Y. R., and Cha, S. D. (1995) Testing of object-oriented programs based on finite state machines. *Proceedings of the Asia-Pacific Software Engineering Conference*, Brisbane, Australia, December, pp. 234–241. IEEE Computer Society Press.

[7] Gallagher, L., Offutt, J., and Cincotta, T. (2007) Integration testing of object-oriented components using finite state machines. *Software Testing, Verification, and Reliability*, **17**, 215–266.

[8] Frankl, P. G. and Weyuker, E. J. (1988) An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, **14**, 1483–1498.

[9] Herman, P. (1976) A data flow analysis approach to program testing. *Australian Computer Journal*, **8**, 92–96.

[10] Laski, J. (1982) On data flow guided program testing. *Sigplan Notices*, **17**, 62–71.

[11] Rapps, S. and Weyuker, E. J. (1982) Data flow analysis techniques for test data selection. *6th International Conference on Software Engineering*, Tokyo, Japan, May, pp. 272–278. IEEE Computer Society Press.

[12] Zhu, H., Hall, P. A. V., and May, J. H. R. (1997) Software unit test coverage and adequacy. *ACM Computing Surveys*, **29**, 366–427.

[13] Binder, R. (2000) *Testing Object-oriented Systems.* Addison-Wesley Publishing Company Inc., New York, New York.

[14] Harrold, M. J. and Rothermel, G. (1994) Performing data flow testing on classes. *Symposium on*

*Foundations of Software Engineering*, New Orleans, LA, December, pp. 154–163. ACM SIGSOFT.

[15] Ntafos, S. C. (1984) On required element testing. *IEEE Transactions on Software Engineering*, **10**, 795–803.

[16] Clarke, L. A., Podgurski, A., Richardson, D. J., and Zeil, S. J. (1989) A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, **15**, 1318–1332.

[17] Frankl, P. G., Weiss, S. N., and Hu, C. (1997) All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software*, **38**, 235–253.

[18] Frankl, P. G. and Weiss, S. N. (1993) An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, **19**, 774–787.

[19] Harrold, M. J. and Soffa, M. L. (1991) Selecting and using data for integration testing. *IEEE Software*, **8**, 58–65.

[20] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. (1994) Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of the Sixteenth International Conference on Software Engineering*, Sorrento, Italy, May, pp. 191–200. IEEE Computer Society Press.

[21] Mathur, A. P. and Wong, W. E. (1994) An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification, and Reliability*, **4**, 9–31.

[22] Offutt, J., Pan, J., Tewary, K., and Zhang, T. (1996) An experimental evaluation of data flow and mutation testing. *Software–Practice and Experience*, **26**, 165–176.

[23] Laski, J. and Korel, B. (1983) A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, **SE-9**, 347–354.

[24] Parrish, A. and Zweben, S. H. (1991) Analysis and refinement of software test data adequacy properties. *IEEE Transactions on Software Engineering*, **17**, 565–581.

[25] Doong, R. K. and Frankl, P. G. (1991) Case studies on testing object-oriented programs. *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, Victoria, British Columbia, Canada, October, pp. 165–177. IEEE Computer Society Press.

[26] Chen, H. Y., Tse, T. H., Chan, F. T., and Chen, T. Y. (1998) In black and white: An integrated approach to class-level testing. *ACM Transactions on Software Engineering Methodology*, **7**, 250–295.

[27] Chen, H. Y., Tse, T. H., and Chen, T. Y. (2001) TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering Methodology*, **10**, 56–109.

[28] Kung, D., Liu, C. H., and Hsia, P. (2000) An object-oriented Web test model for testing Web applications. *24th Annual International Computer Software and Applications Conference (COMPSAC2000)*, Taipei, Taiwan, October, pp. 537–542. IEEE Computer Society Press.

[29] Kung, D., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. (1995) A test strategy for object-oriented programs. *19th Computer Software and Applications Conference (COMPSAC 95)*, Dallas, TX, August, pp. 239 –244. IEEE Computer Society Press.

[30] Liu, C. H., Kung, D., Hsia, P., and Hsu, C. T. (2000) Structural testing of Web applications. *Proceedings of the 11th International Symposium on Software Reliability Engineering*, San Jose CA, October, pp. 84–96. IEEE Computer Society Press.

[31] Jin, Z. and Offutt, J. (1998) Coupling-based criteria for integration testing. *Software Testing, Verification, and Reliability*, **8**, 133–154.

[32] Kung, D., Suchak, N., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. (1994) On object state testing. *Eighteenth Annual International Computer Software & Applications Conference*, Los Alamitos, CA, November, pp. 222–227. IEEE Computer Society Press.

[33] Alexander, R. T. and Offutt, J. (1999) Analysis techniques for testing polymorphic relationships. *Proceedings of the Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '99)*, Santa Barbara CA, August, pp. 104–114. IEEE Computer Society Press.

[34] Alexander, R. T. and Offutt, J. (2000) Criteria for testing polymorphic relationships. *Proceedings of the 11th International Symposium on Software Reliability Engineering*, San Jose CA, October, pp. 15–23. IEEE Computer Society Press.

[35] Alexander, R. T., Offutt, J., and Bieman, J. M. (2002) Fault detection capabilities of coupling-based oo testing. *Proceedings of the 13th International Symposium on Software Reliability Engineering*, Annapolis MD, November, pp. 207–218. IEEE Computer Society Press.

[36] Alexander, R. T. and Offutt, J. (2004) Coupling-based testing of O-O programs. *Journal of Universal Computer Science*, **10**, 391–427. http://www.jucs.org/jucs_10_4/coupling_based_testing_of.

[37] Offutt, J., Alexander, R., Wu, Y., Xiao, Q., and Hutchinson, C. (2001) A fault model for subtype inheritance and polymorphism. *Proceedings of the 12th International Symposium on Software Reliability Engineering*, Hong Kong China, November, pp. 84–93. IEEE Computer Society Press.

[38] Ma, Y.-S., Kwon, Y.-R., and Offutt, J. (2002) Inter-class mutation operators for Java. *Proceedings of the 13th International Symposium on Software Reliability Engineering*, Annapolis MD, November, pp. 352–363. IEEE Computer Society Press.

[39] Ma, Y.-S., Offutt, J., and Kwon, Y.-R. (2005) Mujava : An automated class mutation system. *Software Testing, Verification, and Reliability*, **15**, 97–133.

[40] Tai, K.-C. and Daniels, F. J. (1997) Test order for inter-class integration testing of object-oriented software. *The Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97)*, Santa Barbara CA, August, pp. 602–607. IEEE Computer Society.

[41] Hanford, K. V. (1970) Automatic generation of test cases. *IBM Systems Journal*, **4**, 242–257.

[42] Ramamoorthy, C. V., Ho, S. F., and Chen, W. T. (1976) On the automated generation of program test data. *IEEE Transactions on Software Engineering*, **2**, 293–300.

[43] Clarke, L. A. (1976) A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, **2**, 215–222.

[44] Lundstrom, S. F. (1978) Adaptive random data generation for computer software testing. *Proceedings of the National Computer Conference*, ???, ???, pp. 505–511.

[45] DeMillo, R. A. and Offutt, J. (1991) Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, **17**, 900–910.

[46] Korel, B. (1992) Dynamic method for software test data generation. *Software Testing, Verification, and Reliability*, **2**, 203–213.

[47] Offutt, J., Jin, Z., and Pan, J. (1999) The dynamic domain reduction approach to test data generation. *Software–Practice and Experience*, **29**, 167–193.

[48] Goldberg, A., Wang, T. C., and Zimmerman, D. (1994) Applications of feasible path analysis to program testing. *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, Seattle WA, August, pp. 80–94. ACM Press.

[49] Hedley, D. and Hennell, M. A. (1985) The causes and effects of infeasible paths in computer programs. *Proceedings of the Eighth International Conference on Software Engineering*, London UK, August, pp. 259–266. IEEE Computer Society Press.

[50] Jasper, R., Brennan, M., Williamson, K., Currier, B., and Zimmerman, D. (1994) Test data generation and feasible path analysis. *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, Seattle WA, August, pp. 95–107. ACM Press.

[51] Offutt, J. and Pan, J. (1997) Detecting equivalent mutants and the feasible path problem. *Software Testing, Verification, and Reliability*, **7**, 165–192.