Automatically Testing Interacting Software Components*

Leonard Gallagher
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
lgallagher@nist.gov

Jeff Offutt
Information and Software Engineering
George Mason University
Fairfax, VA 22030, USA
offutt@ise.gmu.edu

ABSTRACT

One goal of integration testing for object-oriented software is to ensure high object interoperability. Sent messages should have the intended effects on the states and subsequent actions of the receiving objects. This is especially difficult when software is comprised of components developed by different vendors, with different languages, and the implementation sources are not all available. A previous paper presented a model of inter-operating OO classes based on finite state machines. It addresses methods for identifying the relevant actions of a test component to be integrated into the system, transforms the finite state specification into a control and data flow graph, labels the graph with all $defs$ and *uses* of class variables, and presents an algorithm to generate test specifications as specific paths through the directed graph. It also presents empirical results from an automatic tool that was built to support this test method. This paper presents additional details about the tool itself, including how several difficult problems were solved, and adds new capabilities to help automate the transformation of test specifications into executable test cases. The result is a fresh approach to automated testing. It follows accepted theoretical procedures while operating directly on an object-oriented software specification. 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.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

*The second author is sponsored in part by National Institute of Standards and Technology (NIST), Software Diagnostics and Conformance Testing Division (SDCT).

General Terms

Verification

1. INTRODUCTION

Many object-oriented 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* [2, 11]. A *component* is a closely related collection of classes (possibly even a single class), and components inter-operate to provide needed functionality. Each component is assumed to be a separate executable, thereby allowing asynchronous behavior. 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 method has 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.

Behavior of objects is captured as a set of transition rules for each method and described as finite state machines [3, 4, 12]. 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. [8] developed a class-level flow graph to represent control and data flow within a single class. Our previous paper [6] extended their ideas to integration testing of multiple interacting classes. The state transition specification is stored in a relational database. Transitions that are relevant to the class 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 [5], 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.

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 so called by a relevant transition become themselves 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
- 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

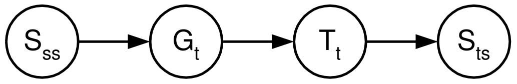

$$\begin{tikzcd} S_{ss} \arrow[r] & G_t \arrow[r] & T_t \arrow[r] & S_{ts} \end{tikzcd}$$

Figure 1: Path Derived From a Relevant Transition

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's called, and by T_{vb} if $obj1$ is in state S_b.

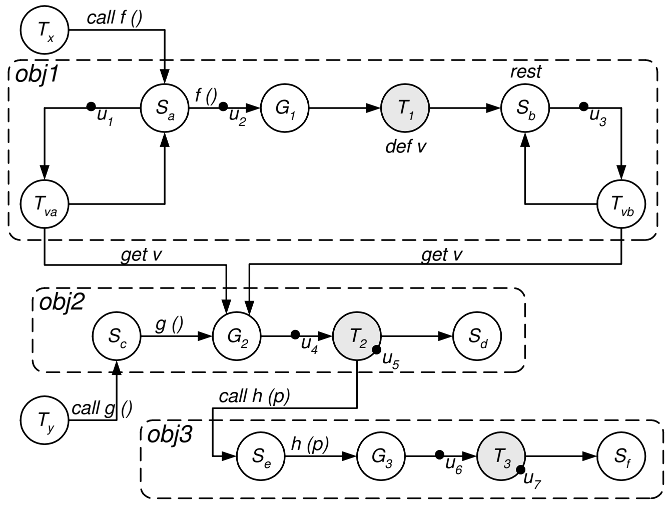


Figure 2: Portion of Component Flow Graph

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 model that can and should be adapted by the developer. 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). How complete our model is will depend on how complete the design model is. If the initial specification covers all possible behaviors, then they will be captured in the database. An advantage of this approach is that once the methods and possible states are in the database, we can do some rather rigorous testing to ensure that all possible combinations are addressed. Space requirements and limitations are discussed in the previous paper [6].

3. COVERAGE CRITERIA

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 [5, 7, 9, 10, 13]. 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 transitions that need to be executed as part of a def-use path. Both types of paths can be seen in Figure 2. The path $T_1 : S_b : T_{wb} : G_2 : T_2 : S_e : G_3 : T_3$ 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.

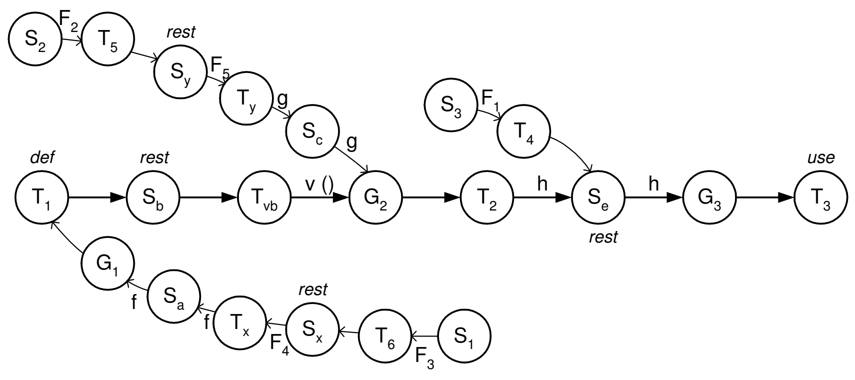


Figure 3: External Function Calls to Traverse a Def-Use Path

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 [6] 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 generated def-use paths 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 [1] of the system under test. First, if *obj*3 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 *obj*2 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 *obj*3 in state S_e. Next, if *obj*1 is not already in the source state S_x for transition T_x, some external function F_3 must be executed to move *obj*1 from its initial state S_1 to state S_x through some transition T_6. Then external function F_4 can be executed while *obj*1 is in state S_x to invoke transition T_x, which in turn calls function f, which results in transition T_1, which defines variable v and puts *obj*1 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 *obj*2 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 development.

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. These tools are described further 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. 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, each path will be traversed to its ending transition node. But, by construction of such paths, this node may be the head definition node 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 all such paths. At the completion of all triggered transitions identified by these paths, the software system will be in a new system state SS_1. The tester could then execute 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 traversal 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 traversal from a rest state to a new *leg* in a def-use path are labeled t.

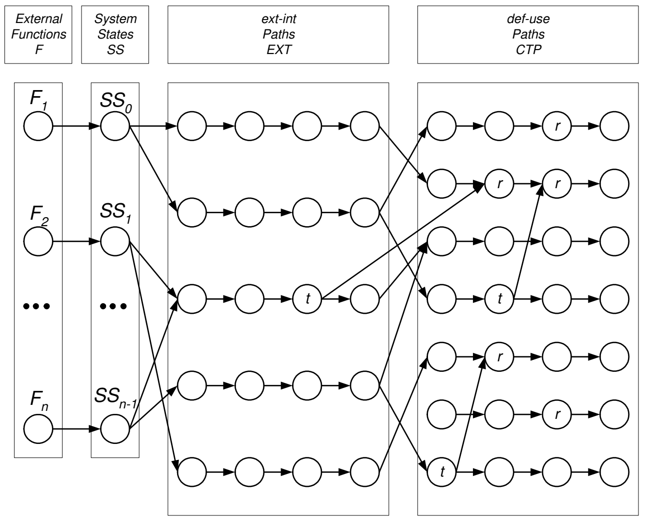


Figure 4: Generating an External Test Sequence

An external function F_i may initiate multiple paths in EXT and each EXT path may initiate or extend multiple paths in CTP. Many of these paths will be mutually inconsistent. Some will have contradictory guard predicates, some will violate the *state compatibility rule* for paths, and some will re-define a variable that has already been defined by a definition node at the head of a def-use path. It will be necessary to analyze, very carefully, all initiated paths one transition at a time. This analysis will remove from further consideration at this step in test sequence generation paths that produce inconsistencies in the parallel traversal of all initiated paths. At each step in this process, the following actions must be taken:

- Identify one guard predicate to be satisfied when a choice of guard nodes appear in simultaneous paths
- Delete all paths with guard nodes for the same function that differ from the selected guard
- Identify and delete any combined ext-int-def-use paths 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 combined ext-int-def-use paths that are currently at a rest state, but for which the triggered actions of the current external function result in re-definition of the def variable

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 re-initiate the desired path. This time, subsequent actions may cause complete traversal to the use node of this def-use path. We have developed a tool that helps a human tester make the above choices in a way that allows maximal coverage of untraversed def-use paths. The human tester merely responds to questions 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.

Consider the set of ext-int-def-use paths that have been initiated by execution of previous functions in a test sequence and remain undeleted and uncompleted after the triggered effects of function F_i. Call this set ATP for *active test paths.* The algorithm for test sequence generation 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 EXT x 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. Further research may help to automate this existing human role in test sequence generation.

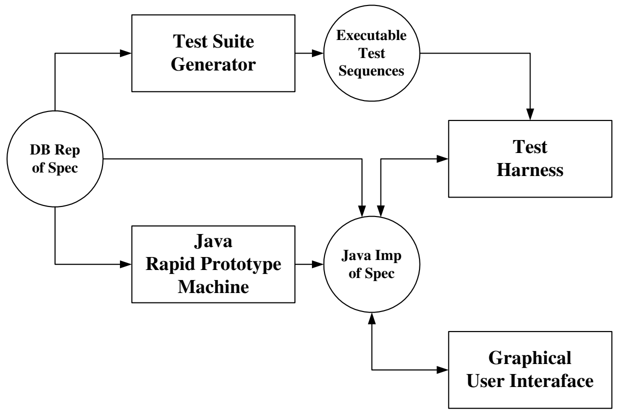
After completion of the analysis of the triggered effects of each external function call F_i, the tool records two pieces of information:

- The set of def-use paths covered by F_i
- The expected system state SS_i, after stepping through all intermediate transitions in the identified paths.

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. When an implementation claiming to implement the specification is tested, each test sequence in a test suite is executed against the implementation. After the triggered effects of each F_i, the implementation should be in the system state predicted by SS_i. If an implementation fails to be in the predicted system state after each F_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. TEST ARCHITECTURE

Consider the testing architecture presented in Figure 5. 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.



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

Figure 5: Test Tool Architecture

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.

7. CONCLUSIONS

This paper has presented 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 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 [6].

Future research will focus on additional automation of existing manual steps in this methodology. One current direction is to provide automation for translation of software specifications into the database representation. If the specifications are in a state-machine representation (such as UML statecharts), then much of the metadata 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, if one exists. 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. Additional tester support for choosing the *best* external method to call in test sequence construction is also desirable.

8. 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.

9. REFERENCES

[1] B. Binder. *Testing Object-oriented Systems.* Addison-Wesley Publishing Company Inc., New York, New York, 2000.

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

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

[4] T. Chow. Testing software designs modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[5] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[6] L. Gallagher and A. J. Offutt. Integration testing of object-oriented components using finite state machines. *The Journal of Software Testing, Verification, and Reliability.* In press - published online January 2006.

[7] P. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96, November 1976.

[8] H. W. Hong, Y. R. Kwon, and S. D. Cha. Testing of object-oriented programs based on finite state machines. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 234–241, Brisbane, Australia, 1995.

[9] J. Laski. On data flow guided program testing. *Sigplan Notices*, 17(9), September 1982.

[10] S. Rapps and E. J. Weyuker. Data flow analysis techniques for test data selection. In *Software Engineering 6th International Conference.* IEEE Computer Society Press, 1982.

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

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

[13] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.