

Integration testing of object-oriented components using finite state machines



Leonard Gallagher¹, Jeff Offutt^{2,*},[†] and Anthony Cincotta¹

¹*Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-8970, U.S.A.*

²*Information and Software Engineering Department, George Mason University, Fairfax, VA 22032-4444, U.S.A.*

SUMMARY

In object-oriented terms, one of the goals of *integration testing* is to ensure that messages from objects in one class or component are sent and received in the proper order and have the intended effect on the state of the objects that receive the messages. This research extends an existing single-class testing technique to integration testing of multiple classes. The single-class technique models the behaviour of a single class as a finite state machine, transforms the representation into a data flow graph that explicitly identifies the definitions and uses of each state variable of the class, and then applies conventional data flow testing to produce test case specifications that can be used to test the class. This paper extends those ideas to *inter-class* testing by developing flow graphs, finding paths between pairs of definitions and uses, detecting some infeasible paths and automatically generating tests for an arbitrary number of classes and components. It introduces flexible representations for message sending and receiving among objects and allows concurrency among any or all classes and components. Data flow graphs are stored in a relational database and database queries are used to gather def-use information. This approach is conceptually simple, mathematically precise, quite powerful and general enough to be used for traditional data flow analysis. This testing approach relies on finite state machines, database modelling and processing techniques and algorithms for analysis and traversal of directed graphs. The paper presents empirical results of the approach applied to an automotive system. This work was prepared by U.S. Government employees as part of their official duties and is, therefore, a work of the U.S. Government and not subject to copyright. Published in 2006 by John Wiley & Sons, Ltd.

KEY WORDS: software integration testing; data flow testing; data modelling; finite state machines; object-oriented

*Correspondence to: Jeff Offutt, Information and Software Engineering Department, George Mason University, Fairfax, VA 22032-4444, U.S.A.

[†]E-mail: offutt@ise.gmu.edu

Contract/grant sponsor: U.S. National Science Foundation; contract/grant number: CCR-98-04111



1. INTRODUCTION

Testing of object-oriented software is complicated by the fact that software being tested is often constructed from a combination of previously written, off-the-shelf components with some new components developed to satisfy new requirements. The previously written components are often 'sealed' so that source code is not available, yet objects in the new components will interoperate via messages with objects in the existing components. Software *conformance testing* is the act of determining whether or not a software product conforms to a functional specification, where the *functional specification* is a set of rules that the product must satisfy. One goal of this paper is to provide conformance-testing techniques for the integration of new product components into a complete software system.

Each component is assumed to be object-oriented, that is, it is implemented with objects that have state and behaviour. In this paper, a *class* is the basic unit of semantic abstraction, a *component* is a closely related collection of classes and a *system* is a collection of components designed to solve a problem. Each component is assumed to be a separate executable entity, thereby allowing asynchronous behaviour. An *object* is an instance of a class. Each object has state and behaviour, where state is determined by the values of *variables* defined in the class and behaviour is determined by *methods* (functions or procedures) defined in the class that operate on one or more objects to read and modify their state variables. The *behaviour* of an object when acted upon by a method can be modelled 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*. The results in this paper are independent of programming language and this paper uses a mix of Java and C++ terminology.

If a finite state machine represents the states and transitions of a class, then the behaviour of an object can be captured as a set of transition rules for each method. Thus finite state machines are often used for class specifications in object-oriented analysis and design [1–4]. The behaviour of a component is specified by the behaviour of its constituent classes. The public interface to a component is a list of public classes, which are accessed through the public methods in those classes. A *state transition specification* for a class is the set of state transition rules for each method of the class. The *state* of an object is determined by the values of its instance and class variables, which are collectively called *state variables*. Given a state transition specification for each class in a software system, the goal of this research is to construct *test specifications* that are used to construct an *executable test suite* to determine if an implementation of a software system conforms to its functional specification.

This paper uses definitions from Booch [5] and Rumbaugh *et al.* [6] to characterize an object as something that has state, behaviour and identity. Also, an object's class is characterized in terms of the *states*, *events* and *transitions* of a finite state machine. A graph model of the software is used as a basis for generating test specifications. Hong *et al.* [7] developed a *class-level flow graph* to represent control and data flow within a *single class*. This research uses their ideas as a basis for integration testing of *multiple interacting classes*. The state transition specification is stored in a database, which is then used as a basis for creating a *component flow graph*, which includes control and data flow information. The methodology described here defines test criteria on this graph and generates test specifications to satisfy the criteria.



The paper describes a process that begins with state transition specifications for each class in an object-oriented software system, defines the transitions that are relevant to a specific component of that system and then translates the relevant transitions into a *component flow graph* with nodes and edges labelled for control and variable definitions and uses. Test criteria are defined on this graph and sets of paths are selected that constitute test specifications to satisfy the criteria. Executable tests are then constructed from the test specifications.

The use of a database to store definition/use information makes it convenient to provide additional information to the tester. In traditional data flow testing [8], the tester is provided with pairs of definitions and uses of variables (def-use pairs) and the tester attempts to find tests to cover those DU-pairs by supplying tests through an instrumented program. These tests are sometimes random, arbitrary, automatically generated or generated by humans with well-defined goals. Traditional data flow testing works for individual functions because the number of possible tests is fairly small but is likely to run into trouble during inter-class testing because the number of possible tests is much larger. Thus it is necessary to provide the tester with more information. The database representation helps provide more information; instead of simply identifying *def-use* pairs, the tester is given full paths between the definitions and uses (DU-paths). In traditional code-based data flow testing, storing the complete path predicates for anything more than a tiny (20–50 lines of code) function is impractical and this has been a factor in the lack of widespread adoption of the technique. Using the database allows efficient management of these potentially large predicates.

The attributes and constraints of classes and methods are modelled as attributes and constraints of tables in a relational database. In this manner, mathematical specifications over the class properties can be translated to database operations. Sections 3–6 describe the process of representing state transition specifications in a database, determining relevant transitions in the state machine, generating a component flow graph and determining test specifications. Section 7 presents empirical results from applying this technique to an automotive system that includes the cruise control, engine, brakes, gas, throttle, ignition, transmission, wheels and displays.

2. BACKGROUND

Much of testing has been based on data and control flow through programs [8,9]. In such testing, graphs are defined in which nodes are formed from *basic blocks*, which are maximal sequences of straight-line statements with the property that if the first statement is executed, then all the statements will be executed. In a control flow graph, edges are formed from the branching statements of the program. A *definition (def)* of a memory location 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. An edge is formed from nodes in which a memory location is defined to nodes in which the memory location is used and there is a def-clear control subpath from the def to the use. A *def-clear subpath* for a location x is a control subpath that does not contain a definition of x . A *DU-pair* is a definition and a use of the same location 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.

Data flow testing criteria [8,10,11] require tests that execute from data definitions to data uses under various conditions. Most research papers in data flow analysis have derived graphs directly from the code, referred to as *traditional data flow analysis* here. This paper uses a form of data flow analysis



that is defined on finite state machines that are derived from the behaviour of classes; thus the data flow might not be directly reflected in the implementation.

Harrold and Rothermel [12] describe an approach that applies traditional data-flow analysis to classes. That approach emphasizes three levels of testing: (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. Integration testing attempts to test interactions among different classes; thus this paper introduces the term *inter-class testing*, in which more than one class is tested at the same time. To perform these analyses, Harrold and Rothermel represent a class as a Class Control Flow Graph (CCFG), which contains information that can be used during testing.

Most research in object-oriented testing has been at the intra-class level. This includes work by Hong *et al.* [7], Parrish *et al.* [13], Turner and Robson [4], Doong and Frankl [14] and Chen *et al.* [15]. Intra-class testing strategies focus on one class at a time, so do not look for problems that exist in the interfaces between classes or in inheritance and polymorphism among classes. In their TACCLE methodology [16], Chen *et al.* define class semantics algebraically as axioms and construct test cases as paths through a state-transition diagram with path selection based on *attributely non-equivalent ground terms*. They extend this methodology to multiple classes by defining inter-class semantics in terms of *contracts*. The contract notion increases complexity substantially and is difficult to re-use when other components are added to the system.

Inter-class testing work has been done by Jin and Offutt [17], who defined *coupling-based testing*, which requires tests to be found that cover code-level control and data couplings between methods in different classes. Alexander and Offutt [18,19] have extended these ideas to cover couplings formed from inheritance and polymorphism. Chen and Kao [1] describe an approach to testing object-oriented programs called *Object Flow Testing*, in which testing is guided by data definitions and uses in pairs of methods that are called by the same caller and in which testing should cover all possible type bindings in the presence of polymorphism. Kung *et al.* [20] address object-oriented testing of inheritance, aggregation and association relationships among multiple classes in C++ source code by automatically generating an object-relation diagram and finding a test in order to minimize the effort to construct test stubs.

Some related work has been done on the subject of testing Web software. Kung *et al.* [21,22] and Liu *et al.* [23] have carried out some initial work in this area. They 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.

This paper extends the intra-class data flow work by Hong *et al.* [7] to the inter-class level, thus providing full integration level testing. This paper does not explicitly deal with inheritance and polymorphism.

Following Rumbaugh *et al.* [6], the behaviour of a class is specified as a finite state machine in terms of states and events. When an event is received, a transition occurs and the current state, a guard and the event determine the next state. A *state* is represented by a categorization of values of the state variables, i.e. by a predicate that evaluates to true. Note that state predicates are explicitly allowed to



overlap, that is, two states may share the same predicate. In this case, a target state is determined by all of the properties of a transition, not just the predicate that defines the target state.

A *transition* is composed 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 the 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 event function. An *action* is an operation that 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 code in which all operations are executed if any one is executed.

Pre-conditions and post-conditions of methods in a class 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, which would allow stronger post-conditions. In turn, stronger post-conditions would yield larger graphs and more tests, so this becomes a choice of granularity that results in a cost versus potential benefit tradeoff. Although future experimentation may provide some guidance, it is likely that the wisdom and experience of both system analysts and test engineers will be needed to make the best choice of granularity.

A class state machine (CSM) for a single class is defined in Definition 1. This definition is from Hong *et al.*'s paper [7], with the addition of the parameter set P , which will be needed for multiple classes. The CSM is extended to a CSM for multiple classes in Section 2.2.

Definition 1. (CSM) A class state machine of a class C is a tuple (V, F, P, S, T) , where the following statements apply.

- V is a finite set of instance variables of C .
- F is a finite set of member functions of C .
- P is a finite set of parameters of member functions.
- S is a finite set of states, $S = \{s | s = (pred)\}$ where $pred$ is a predicate on the instance variables in V .
- T is a finite set of transitions, $T = \{t | t = (source, target, fn, guard, action)\}$ where:
 - $source, target \in S$ are the states before and after the transition;
 - $fn \in F$ is a member function that triggers t if the guard predicate evaluates to *true*;
 - $guard$ is a predicate on instance variables in V and parameters of member functions in F ;
 - $action$ is a sequence of computations on instance variables in V and parameters of member functions in F .

2.1. Single-class example—Engine

As a simple example, consider a class Engine, which has states ON and OFF, instance variables *speed* and *keyOn* and methods *Start(sp)* and *Stop()*. Each state is associated with values of the instance variables as follows:

$$\text{OFF: speed} = 0 \wedge \text{keyOn} = \text{false}; \quad \text{ON: } 0 \leq \text{speed} \leq 110 \wedge \text{keyOn} = \text{true}$$

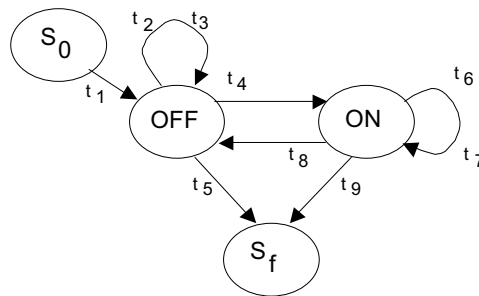


Figure 1. CSM for Engine.

In the Engine example, the transition from OFF to ON is triggered by the class member function `Start()`. The guard for this transition should require the key to be in ($keyOn = true$) and the action should specify that the speed is set ($speed = sp$). The sets of variables, member functions, states and transitions are defined as follows:

```

S = {S0, Sf, ON, OFF}
V = {int speed, boolean keyOn}
F = {Engine (), ~Engine (), setKeyOn (boolean in), Start (int sp), Stop (), setSpeed (int sp),
    int getSpeed ()}
P = {setKeyOn: in, Start: sp, setSpeed: sp }
T = {ti | 1 ≤ i ≤ 9}
t1 = (S0, OFF, Engine(), true, {speed = 0, keyOn = false})
t2 = (OFF, OFF, getSpeed(), true, {return speed})
t3 = (OFF, OFF, setKeyOn(in), true, {keyOn = in})
t4 = (OFF, ON, Start(sp), keyOn==true ∧ 0 ≤ sp ≤ 110, {speed = sp})
t5 = (OFF, Sf, ~Engine(), true, { })
t6 = (ON, ON, getSpeed(), true, {return speed })
t7 = (ON, ON, setSpeed(sp), 0 ≤ sp ≤ 110, {speed = sp})
t8 = (ON, OFF, Stop(), true, {speed = 0})
t9 = (ON, Sf, ~Engine(), true, { })

```

`Engine()` and `~Engine()` are the class constructors and destructors. The method `setKeyOn(in)` allows the key to be inserted into the ignition and `setSpeed(sp)` and `getSpeed()` control the speed of the engine. `Start(sp)` starts the engine running at speed sp and `Stop()` turns the engine off. The state transition diagram for Engine is shown in Figure 1, with each transition represented as a labelled and directed arc between two states.

In the class Engine, the engine is turned on (transition t_4) by method `Start(sp)` and can only be turned on if the key is in the ignition and the initial speed is between 0 and 110 (the guard $keyOn==true \wedge 0 \leq sp \leq 110$). If the guard is true, then the new speed is set to the parameter given to the `Start()` method (the action $speed = sp$). The other transitions are similar to t_4 .



Table I. Classes in CruiseControl components.

| Component | Classes |
|-----------------|--|
| Acceleration | GasUser, Throttle, Transmission, Wheel |
| Brakes | BrakeUser, BrakeControl |
| Automobile | CruiseUser, CruiseUnit |
| Engine | Engine |
| InstrumentPanel | Gauges |
| SystemControl | AutoSystem, Ignition |

2.2. Multi-class example—Automobile

Inter-class integration testing addresses interactions among multiple components, so the following example modifies the Engine class from Section 2.1 and integrates it with other components. Each message received by an object is interpreted as an *event*. Components can function as independent processes, possibly running on different computers and possibly receiving concurrent messages from many sources, so the sending object may not be certain of the recipient object's state when the event is processed.

The Automobile system consists of six core components: Acceleration; Brakes; CruiseControl; Engine; InstrumentPanel; and SystemControl. This example tests how the CruiseControl component integrates with the remainder of the system. The classes that make up the components are shown in Table I.

The Ignition, GasUser, BrakeUser, Transmission and CruiseUser classes have external interfaces that are accessible to a human driver. The Gauges are all read-only for external users but these human observations are not part of the automobile specification. The CruiseUser class has an *On/Off* switch, as well as *Cancel*, *Resume/Accel (RA)* and *Set/Decel (SD)* buttons for CruiseControl. If the user holds the RA or SD button down, the user mode is that button; when the button is released the user mode returns to *Neutral (NT)*. Environmental conditions such as wind and hills are simulated by an externally controlled ExternalDrag variable. Users can use controls in the car to invoke 12 methods:

| | |
|----------------------------------|----------------------------------|
| BrakeUser.IsActive (status) | status \in {true, false} |
| BrakeUser.PedalPressure (press) | $0 \leq \text{press} \leq 99$ |
| CruiseUser.Cancel() | |
| CruiseUser.Mode (mode) | mode \in {NT, SD, RA} |
| CruiseUser.Switch (status) | status \in {On, Off} |
| Engine.ExternalDrag (drag) | $0 \leq \text{drag} \leq 2$ |
| GasUser.PedalPosition (position) | $0 \leq \text{position} \leq 99$ |
| Gauges.OilPressure (press) | press ≥ 0 |
| Gauges.WaterTemp (temp) | temp ≥ 0 |
| Ignition.Key(status) | status \in {On, Off} |
| Ignition.StartEngine() | |
| Transmission.Gear(gear) | gear \in {N, R, 1, 2, 3, 4, 5} |

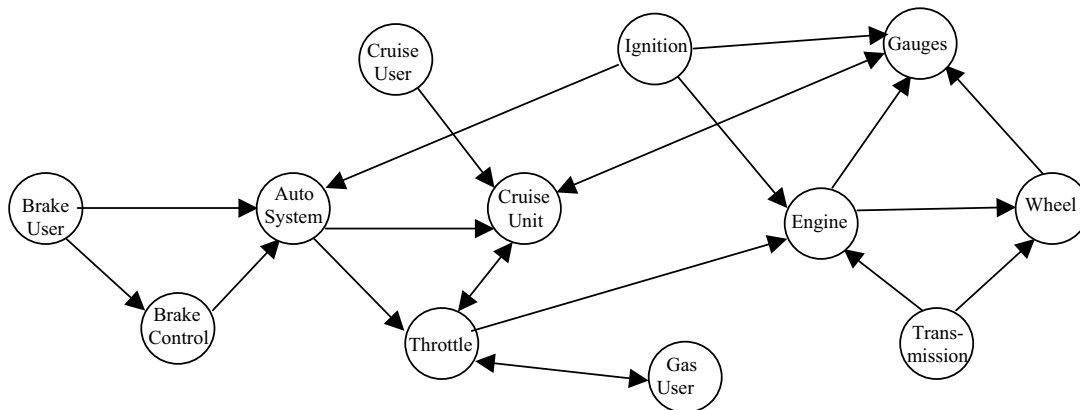


Figure 2. Class-to-class data flow.

All other methods are internal methods that can only be invoked by internal actions. Thus, all test case inputs are sequences of calls to the above 12 methods. The CruiseUser class has a number of non-feasible transitions; for example, the cruise control RA button cannot be pushed at the same time as the SD button because their physical placement prohibits them from being depressed simultaneously.

Definition 1 is extended to define a *combined* CSM for multiple classes by merging the sets V , F , P , S and T and adding a new set C of classes. The resulting tuple is (C, V, F, P, S, T) . For the automotive example, C is a set of 12 classes, V is a set of 58 variables consisting of the union of all state variables from each class, F is a set of 106 methods consisting of the union of all member functions from each class, P is a set of 44 parameters representing inputs of mutator functions, S is a set of 44 states consisting of the union of all states from each class and T is a set of 263 transitions consisting of the union of all transitions from each class. A database schema for representing these sets and the relationships among them is defined in Section 3 and a partial table that lists relevant transitions for the CruiseControl component of a combined CSM is given in Appendix A.

Figure 2 is a directed graph that shows an abstraction of the relevant communication paths among the classes. Since the Gauges class is passive, the double-arrow between CruiseUnit and Gauges indicates that methods in CruiseUnit can read from and write to state variables in Gauges. The Throttle class, however, is active and can change the pedal position in GasUser as well as increase the gas supply to the Engine. In order to simulate road conditions such as hills, the Engine class has an externally controlled drag variable that changes Engine Revolutions per minute (Rpm) and thereby affects the axel speed of the Wheel and ultimately the speedometer setting in Gauges. The Wheel sets the speed in Gauges, so the loop from CruiseUnit through Throttle, Engine, Wheel and Gauges back to CruiseUnit will be important in integration testing.

The automobile example uses some special syntax to distinguish a situation where an object sends an asynchronous message to itself with the intent that the message is put on a queue to be acted upon in a subsequent transition. This is used in several classes in lieu of a system clock to keep processes from terminating. For example, in most of the cruise control transitions, the action of the

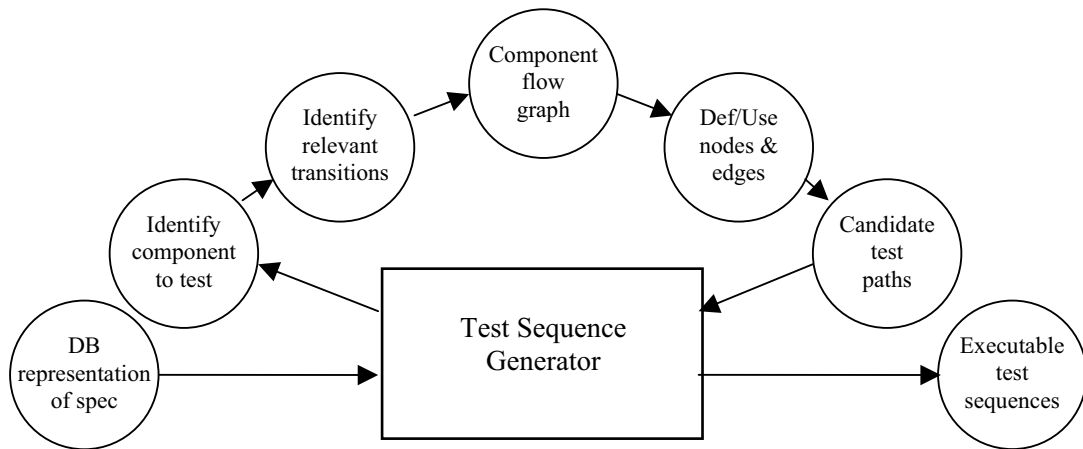


Figure 3. Flow of tool automation.

transition will set parameters for gas flow and throttle but before relinquishing control they will send an asynchronous message back to the underlying object to check all of the gauges to see if further action is required. This message will be put on a queue along with other explicit messages received from other components and will be executed when it moves to the head of the queue. The cruise control component could be in a different state when this message is finally handled. Different priorities for handling these messages are not addressed.

2.3. Overview of methodology

The overall goal is to automate the process of developing integration tests from the behavioural specifications of the various components. This process is illustrated at a high level in Figure 3. To begin, a state/transition specification must exist for each class, with behaviour specified by a CSM as in Definition 1. The CSM could have been produced during design, perhaps as UML diagrams [24], or might be produced by the tester. The CSMs for the classes are combined to form the needed sets according to a database schema (defined in Section 3). Particular attention is paid to associations between the sets such as when a state or guard references a state variable from its own class or calls a *get* function to reference a state variable from some other class. Each action of a transition is also analysed to identify all calls to *actor* or *mutator* functions from other classes and the passing of state variables as parameters of mutator functions. An *actor* function returns a value from a class and a *mutator* function can change a value.

Once the software system is represented in the database schema (*DB representation of spec* in Figure 3), the next step is to identify one or more individual components to test and to determine how they integrate with other components (*Identify component to test*). Then, transitions into and out of the components to test are identified (*Identify relevant transitions*). In the Automobile system, the focus is on the CruiseControl component and its relevant transitions are the interactions with other classes in the



Automobile system. Since CruiseControl activity is cancelled whenever the brake is active or whenever an emergency state is entered, this example safely ignores the complex BrakeControl behaviour dealing with anti-lock brakes and all of the AutoSystem behaviour dealing with items such as air bags. Section 4 of the paper defines *relevant transitions* for a given component.

The next step is to model all potential finite state transitions as a directed graph (*Component flow graph*). Section 5 begins with the relevant transitions and treats those transitions, together with all of the states and guards associated with those transitions, as the nodes of a graph. All data and control flow is modelled as directed edges between these nodes. Following the example of Hong *et al.* [7], the process starts with directed edges from a source state node to the guard node or transition node of each transition, from each guard node to its transition node and from each transition node to its target state node. In addition, each call of an actor function results in directed edges from potential transitions of the called object to states, guards or transitions of the calling object and each call of a mutator function in the action of a transition results in edges from the calling transition to potential source states of the called object. If a mutator function returns a value, then there are edges from potential called transitions back to the calling transition. This process results in a *component flow graph* (formally defined in Section 5).

The final step is to choose a testing criterion and to adapt it to the information stored in the database and the component flow graph (*Def/Use nodes & edges*). The *all-uses* criterion is adapted by defining *defs* and *uses* in terms of references to class variables (formally defined in Section 6). Each *def* takes place at a transition node and each *use* takes place either at a transition node or at a state-to-guard, state-to-transition or guard-to-transition edge. The procedure then looks for *candidate test paths* through the component flow graph for each def-use pair. Much of the remaining effort described in Section 6 is to construct candidate test paths that are *potentially feasible* and *def-free*. The goal is to find paths that result in *executable test cases* for each def-use pair or to prove that such a path cannot exist. If none of the candidate test paths result in an executable test case, then the new information learned from that failure is added to the information base and the methodology is applied again to all untested pairs.

Section 7 describes the overall effect of this methodology on the Automobile example; in particular, Section 7.2 describes a tool that automates most of the processes in Figure 3. For the Automobile example, the tool analyses over 4300 *def-use* pairs, constructs candidate test paths for over 2000 pairs, proves that nearly 1500 pairs are *def-bound* with no possible *def-free* path (infeasible) and constructs an executable test sequence of 145 tests that cover about 50% of the DU-pairs. This research project is actively pursuing the development of efficient executable test case development from candidate test paths, partly relying on algorithms that were previously developed for specification-based testing [9].

3. REPRESENTING COMPONENT SPECIFICATIONS

A specification that defines the states and transitions for each class in a system must be available before test development can begin. This specification will include names of classes, methods and variables. Some of these methods will be invoked from an external interface; they will be the names that are used in the test cases. The eventual test cases will be expressed in terms of these names. These names may or may not be used by the programmers in the eventual implementation of the system but for the context of this work it is assumed that the names are the same. If not, additional work will need to be done to apply the resulting tests to the software; specifically, the test specifications will need to be translated

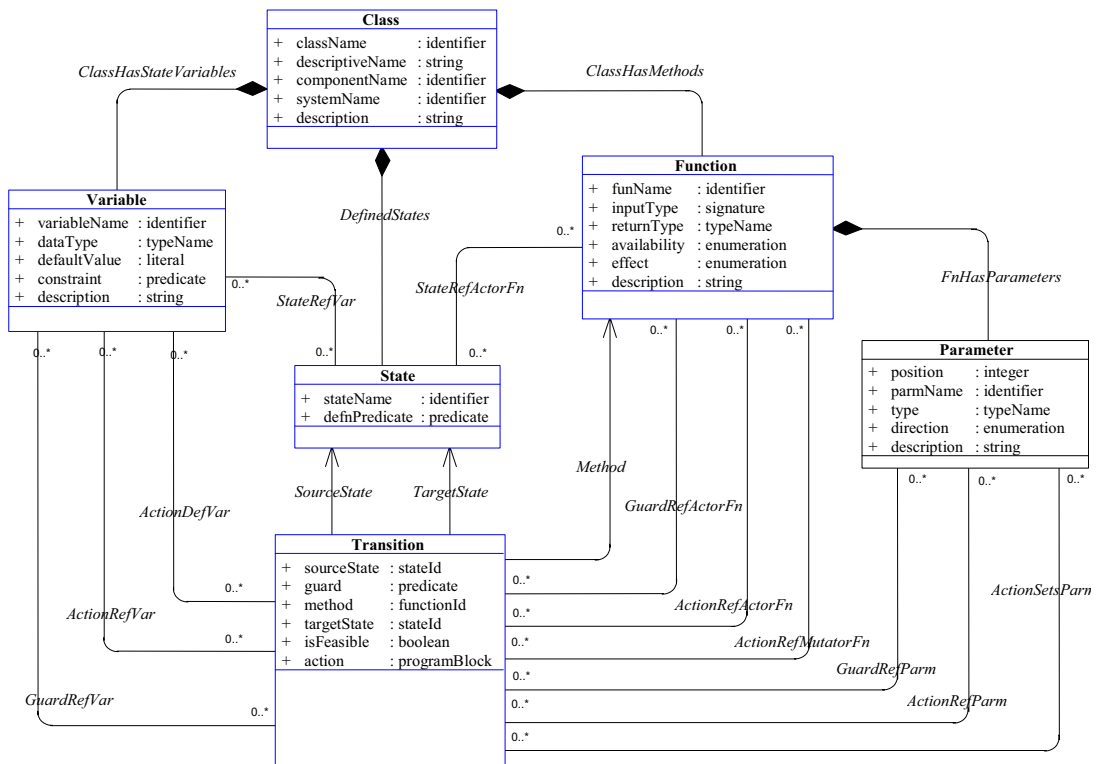


Figure 4. Database schema as a UML class diagram.

to a form that can be used by the implementation. The mapping for this translation will need to be supplied by the designers or programmers of the software.

Each class is used to derive a CSM as defined in Definition 1. Using the relational database model [25–27], classes and sets associated with classes are represented as relational tables.

Figure 4 shows the UML class diagram [24] of a general schema definition for representing combined class state machines. This schema allows representation of class state machines in a way that is convenient to store, access and process the information. Without loss of generality, it is assumed that all methods and procedures can be represented as functions. Each of the six UML classes in Figure 4 represents a table in the model as follows: (1) the *Class* table contains information about the classes that have been defined for the system; (2) the *Variable* table defines instance variables for each class; (3) the *Function* table identifies all of the methods that are associated with each class; (4) the *Parameter* table identifies the input and output parameters for each function; (5) the *State* table contains information about the states in the class state machine; and (6) the *Transition* table describes all transitions among the states.



Since variable, function and state names need be unique only within a class and parameter names need be unique only within a function body, compound identifiers are used for each. For example, (c, v) is a unique identifier for a variable v that is defined in class c . Similarly (c, f) and (c, s) are compound identifiers for functions and states and (c, f, n) is a unique identifier for the n th parameter of a function. In each case, the ordered tuple becomes the primary key of the underlying table. In addition, c serves as a foreign key back to the class definition and fully represents the one-to-many associations identified in the diagram by *ClassHasStateVariables*, *ClassHasMethods*, *FnHasParameters* and *Defined States*. The associations *SourceState* and *TargetState* from Transition to State represent referential integrity constraints on the sourceState and targetState attributes of the Transition table. An additional constraint is that source and target states for a transition are always from the same class. The *Method* association from Transition to Function represents a referential integrity constraint on the method attribute of the Transition table. The remaining associations identify many-to-many relationships among Transitions, States, Variables, Functions and Parameters derived from syntactic analysis of guard and state predicates and transition actions. They are explained further below.

A unique ClassId identifies each class in the Class table from Figure 4 and is the primary key of that table. The className is a surrogate for ClassId and is used to reference the class in state and guard predicates and in the actions of transitions. Similarly, variableName, funName, parmName and stateName are surrogates for hidden identifiers for variables, functions, parameters and states, respectively; each need be unique only within its class. Each class is *owned* by exactly one component, identified by componentName but may be used by many components. In the syntax for predicates, guards and actions, fully qualified names are used to disambiguate the references when necessary.

In the Function table, the availability attribute defines functions to be private (PRI), protected (PRO), public (PUB), or external (EXT). Public functions may be invoked from other classes in the system, whereas external functions are part of the external component interface and can be invoked by other systems or users. External functions typically represent actions that are available to the human user or for black-box testing purposes. The inputType values identify the number of input variables, as well as their data types, so className, funName, inputType and returnType determine the complete *signature* of a function. The effect attribute allows functions to be categorized as Get, Set, Constructor, Actor, Mutator, etc. These are based on standard object-oriented concepts: a Get function is read-only and is said to be an *actor* method on the object; a Set function can update state variables and is said to be a *mutator* method. The following pays particular attention to classifying all methods as *actor*, *mutator*, or *mutator with return*. In the Parameter table, both position and parmName uniquely identify a parameter and one will determine the other. A parameter is used by name but is set by position. Each parameter has a data type and a direction, i.e. In, Out or InOut.

In the State table, the defnPredicate is a Boolean predicate over the state variables. It may reference an in-class variable by name only and may reference a variable in another class by invoking the appropriate actor method, if it is available, to read the value of that variable. Only actor methods can be called from a state's definition predicate. Mutator and constructor methods may only be called from an action that is part of a state transition.

In the Variable table, the dataType attribute identifies the data type of the variable, the defaultValue attribute identifies all automatic value assignments upon creation of a new class instance and the constraint attribute identifies a post-assignment requirement on every variable definition.



For a class c and a transition t , the primary key of the Transition table is the pair (c, t) , which determines all of the other properties of a transition. Some transitions may be well defined in the model but the implementation will not be able to execute them because of a rule or by physical or mechanical impossibility. Such transitions are identified by the `isFeasible` attribute. These types of transitions can be divided into three categories.

- (1) Category one is an error handling transition. Consider an elevator example where a user is at floor five. It is an error to push the button to go to floor five.
- (2) Category two transitions are prevented by hardware; for example, hardware interlocks prevent doors from opening when an elevator is between floors.
- (3) Category three transitions represent logical and physical impossibilities; for example, it is not possible to transition from the 'not pushing button' state to the 'not pushing button' state.

Transitions in category one will be tested as a natural result of the technique presented in this paper. Transitions in category three do not need to be tested. Whether to test transitions in category two depends on the goals of the testers. Since the situation is controlled by hardware, not software, any testing that only involves the software (integration and subsystem testing) may be able to safely ignore these transitions. At the system level, however, these transitions must be carefully tested.

The predicates on guards and states may reference variables and the actions of transitions may reference and assign values to variables. Just as in traditional data flow analysis [8], predicates *reference* a set of objects (*use*) and actions *define* a set of values (*def*). Of course, how to determine the defs and uses depends on the semantics of the language used to express the predicates and transitions of the class state machine. The implementation in this research uses a simple general language to describe state machines, which allows the analysis to proceed in a fairly straightforward manner. Subsequent plans are to expand this part of the prototype to include syntactic analysis of predicates and actions specified in UML [24], Java [28] and other commonly used class definition languages.

Once this syntactic analysis is complete, the results can be captured in the UML diagram of Figure 4 as many-to-many associations among classes. In the database representation, each such association will be a new base table as follows.

- The *StateRefVar* association between State and Variable is a table of tuples (c, s, v) , where (c, s) identifies a state and (c, v) identifies a variable in the same class as the state. The definition predicate of the state *references* the variable. In the Engine example of Figure 1, the OFF state references both *speed* and *keyOn*.
- The *GuardRefVar*, *ActionDefVar* and *ActionRefVar* associations between Transition and Variable are each a table of tuples (c, t, v) , where (c, t) identifies a transition and (c, v) identifies a variable in the same class as that transition. In the first association, the guard of the transition *references* the variable, in the second association the action of the transition *defines* the variable and in the third association the action of the transition *references* the variable. Since each action in a sequence of actions has a sequence number, an occurrence attribute, `SeqNbr`, is assigned to each instance of the second and third associations. In the Engine example, the guard of t_4 references *keyOn*, the action of t_1 defines first *speed* and then *keyOn* and the actions of t_2 and t_6 both reference *speed*.
- The *StateRefActorFn* association between State and Function is a table of tuples (c_s, s, c_f, f) , where (c_s, s) identifies a state and (c_f, f) identifies an actor function. The definition predicate



of the state *references* the actor function. In the Automobile example, all of the states defined for CruiseUnit reference the Cruise variable from the Gauges class of the InstrumentPanel component to see if cruise control is On or Off (not visible in Appendix A).

- The *GuardRefActorFn*, *ActionRefActorFn* and *ActionRefMutatorFn* associations between Transition and Variable are each a table of tuples (c_t, t, c_f, f) , where (c_t, t) identifies a transition and (c_f, f) identifies a function. In the first association the guard of the transition *references* an actor function, in the second association the action of the transition *references* an actor function and in the third association the action of the transition *references* a mutator function. As above, a SeqNbr attribute is assigned to each instance of the second and third associations to identify the position of that reference in the action sequence. The Guard and Action columns of Appendix A show many instances of these types of references for the Automobile example.
- The *GuardRefParm* and *ActionRefParm* associations between Transition and Parameter are each a table of tuples (c, t, n) , where (c, t) identifies a transition whose guard or action references (by name) a parameter of the function associated with that transition and n is the position of that parameter in the signature of the function. In the Automobile guards shown in Appendix A, nearly every guard of a transition derived from a mutator function that has a parameter references that parameter by name. Moreover, the actions of all transitions derived from the Speed method in the Gauges class and the Floor and GasPedal methods in the Throttle class reference the incoming parameter by name. As above, an additional attribute in the *ActionRefParm* association, called SeqNbr, captures the sequence number of that reference in the action sequence of the transition.
- The *ActionSetsParm* association between Transition and Parameter is a table of tuples (c_t, t, c_f, f, n) , where (c_t, t) identifies a transition whose action calls a function, identified by (c_f, f) , from some other class and sets the n th parameter of that function to some non-constant value, possibly the value of a state variable from yet another class c . For the Automobile example, Appendix A shows actions in several transitions of CruiseUnit (e.g. t064, t050) that call the Throttle.Floor() function and set the floor either to the TargetThrottle variable of CruiseUnit or to a value derived from the value of the Position variable from the Throttle class. The *floor* represents a temporary minimum throttle setting, which can be set by AutoSystem or by CruiseUnit. This association also carries an additional attribute to capture the SeqNbr of the set operation in the action sequence of the transition.

Each of the above tables satisfies appropriate referential integrity constraints to the corresponding Transition, Variable, Function, Parameter or State tables.

Every state variable in a class definition is associated with two predefined methods, one to *get* its value and one to *set* its value. An additional association *VarAssocFn* is defined between Variable and Function to maintain the relationship between a state variable and the *get* function that reads its value. This association is not visible in Figure 4 but it is represented by a table of tuples (c, v, f) where (c, v) identifies the state variable and (c, f) identifies the function.

The *ActionSetsParm* association defined above identifies all transitions that: (1) call an external function; and (2) set some parameter of that function to a non-constant value. It is particularly important if the setting of a parameter involves a state variable either from the same class as the calling transition or from some other class. Thus a new three-way association among transitions, state variables and parameters is defined. This is denoted by *ActionSetsParmUsingVar* as a table of tuples



$(c_t, t, c_f, f, n, c_v, v)$ where (c_t, t, c_f, f, n) is a tuple in the *ActionSetsParm* association and (c_v, v) identifies a state variable that is referenced in the setting of that parameter. If the state variable is from the same class as the transition, then $c_t = c_v$, and $c_f = c_v$ if the state variable is from the same class as the called function but in general (c_v, v) could identify a variable in any class that is called by the *get* function on that variable. Appendix A shows examples of the first and second alternatives; for example, several transitions derived from *CheckState()* in *CruiseUnit* call the *Position* variable from *Throttle* and pass it back to *Throttle* by setting *Throttle*'s *floor* variable.

An important consideration in this type of testing has to do with concurrent interactions of the classes. Sometimes the action of a transition will make an *asynchronous* call to a method defined by the same class: it does not wait for a reply before completing the transition and the call does not return a value. Instead, the function call is put on an input queue for that class and considered later. An additional association *ActionRefLocalAsyn* is defined between *Transition* and *Function* to represent such calls. This association is not visible in Figure 4 but it is represented by a table of tuples (c, t, f) where (c, t) identifies the transition and (c, f) represents the asynchronously called function. In the *Automobile* example, many of the *CruiseUnit* and *Wheel* transitions seen in Appendix A have final actions that put *CheckState()* on a queue to be executed by *CruiseUnit* or *Wheel* when they are not busy with other requests.

Although this information is conveniently stored in database tables, it is helpful to consider the tables as sets for most of the development of this work. This is done by a straightforward mapping. Every table can be associated with a mathematical set, where the set is a set of sequences consisting only of the primary key values of the table. In this sense, the sequence (c, f) is an element of the *Function* set if and only if there exists a row in the *Function* table with primary key values (c, f) . If X is such a table-derived set, if w is a non-key column of the corresponding table T and if $x \in X$, then $w(x)$ is defined to be the value in column w of the row of table T identified by x . For example, in the *ActionRefVar* association defined above, $\text{SeqNbr}(c, v, t)$ identifies the value of the *SeqNbr* attribute of that instance. This notational convenience is used freely in the following sections, with *C, F, P, V, S* and *T*, as the sets derived from the tables *Class, Function, Parameter, Variable, State* and *Transition*.

4. CHOOSING RELEVANT STATE MACHINE TRANSITIONS

Given even a moderately large system, the number of transitions available over all class state machines could be quite large. Developing tests over such a large scope would probably be prohibitively expensive and would properly be considered system testing. This paper divides testing into pieces by focusing on one component at a time and generating tests based on the integration interactions that a test component has with other components.

As a testing model, this research assumes there is a *test component M*, whose interactions with the rest of the system are being tested. This is typical when integrating a new component into a complete or partial system. The methodology first determines which transitions from the overall system specification are *relevant* to *M*, that is, into or out of *M*. Relevant transitions fall into two types. *In* transitions represent actions or data that flow into *M*, transitions from other classes in the system that can modify the value of a state variable in any of *M*'s classes. *Out* transitions flow out from *M* to other classes, that is, transitions that can be invoked, directly or indirectly, from actions of transitions in



any of M 's classes. Transitions from classes in M are called *Base* transitions, since they are the starting points for the process that finds the transitive closure of relevant transitions, explained below.

This static analysis process begins by putting all feasible *Base* transitions from any class in M into a set R_0 . The iterative process starts with R_0 . At each step, assume that n steps of the process have been completed, resulting in a set R_n of relevant transitions, each of which is labelled as *In*, *Out* or *Base*. The same transition may appear in R_n as many as three times with different labels. To create the next set of relevant transitions, R_{n+1} , first initialize R_{n+1} to be R_n , and then insert newly labelled transitions as indicated below. A mutator function that returns a usable value to the calling action results in both *In* and *Out* labels for each of its transitions. The following rules control how new transitions are added to the relevant collection.

- Let t be a feasible transition and let f be an *actor* or *mutator with return* function that is the method associated with t . If the SourceState, Guard or Action of any transition in R_n calls f , then t is added to R_{n+1} with an *In* label.
- Let t be a feasible transition and let f be a *mutator* or *constructor* function that is the method associated with t . If the Action of any *Base* or *Out* labelled transition in R_n calls f , then t is added to R_{n+1} with an *Out* label.
- Let t be a feasible transition. Let t' be any transition in R_n labelled either as a *Base* transition or as an *Out* transition. Let f' be an *actor* function that is the method associated with t' . If the SourceState, Guard or Action of t calls f' , then t is added to R_{n+1} with an *Out* label.
- Let t be a feasible transition. Let t' be any *Base* or *In*-labelled transition in R_n and let f' be a *mutator* function that is the method associated with t' . If the Action of t calls f' , then t is added to R_{n+1} with an *In* label.
- Let t be a feasible transition and let f be a function that is the method associated with t . Let t' be a transition in R_n , from the same class as t , labelled either as a *Base* transition or as an *Out* transition. If the Action of t' calls f *asynchronously*, then t is added to R_{n+1} with an *Out* label.
- Let t be a feasible transition whose Action defines a state variable v . Let t' be any transition in R_n from the same class as t , labelled as an *In* transition. If the method associated with t' is the *get* method for the variable v or if the method associated with t' is an actor method and t' has a Guard or Action that references v , then t is added to R_{n+1} with an *In* label.
- Let t be a feasible transition. Let t' be any transition in R_n from the same class as t , labelled as an *Out* transition. If the Action of t' defines a state variable v and if the method associated with t is the *get* method for v or if the method associated with t is an actor method and t has a Guard or Action that references v , then t is added to R_{n+1} with an *Out* label.

Since there are only a finite number of transitions in the system, and since $\{R_n\}$ is a monotonically increasing sequence of sets, the process must terminate at some iteration with no new additions. At that point, the transition labels are discarded and the remaining unlabelled transitions are defined to be the set of transitions in the system that are *relevant* to M . These transitions will determine the *component flow graph* when integrating M with the system. The set of relevant transitions that are determined by this process is the same, no matter in which order the above rules are followed.

Definition 2. (Relevant transitions) Let M be any component of a software system. $R(M)$ is the set of all transitions from the software system that are determined to be relevant to M according to the preceding iterative process.



The initial collection of transitions in the Automobile example includes several transitions in the BrakeControl class that deal with anti-lock brakes and many in the Gauges class that deal with gauges on the instrument panel but that are unrelated to CruiseControl. The above procedure focuses only on transitions relevant to CruiseControl and eliminates these unrelated transitions. Of the original 235 feasible transitions for Automobile, only 160 are relevant to CruiseControl by this definition. Each relevant transition that has a non-trivial action is listed in Appendix A.

5. A DATA FLOW GRAPH MODEL OF STATE TRANSITIONS

The traditional testing literature [4,8,13,20,29] defines a *data flow graph* to be a graph representation of a program's control structure and the flow of data through that structure. A data flow graph is composed of *nodes*, which represent statements or basic blocks, and *edges*, which represent flows of data between basic blocks. If a variable X is given a value, or *defined*, in a node d , and that value can be *used* in another node u , then there is a *data flow dependency* from d to u . The two nodes d and u form a *def-use* pair for the variable X .

This research expands the traditional notion of data flows among statements in a program to be defined among states, guards and transitions in finite state machines. A *component flow graph* represents both the control and data flows for the state transitions of the classes of a component and its relevant transitions from and to other classes in the software system. The definitions in this paper extend those of Hong *et al.* [7] from the single-class case to the multiple-class case.

In a component flow graph, nodes and edges are derived from the relevant transitions of that component. Each such transition has pre-determined associations with the *states*, *guards*, *variables* and *functions* of other *transitions*, as defined in Section 3 and represented in Figure 4.

Definition 3. (Component flow graph) Let M be any component of a software system and let $R(M)$ be the set of all transitions in the system that are relevant to M . Then the *component flow graph* G of M is a directed graph $G = (N, E)$, where N is drawn from elements of the relevant transitions and E represents potential flows of data between nodes in N .

Specifically, the nodes N in G are formed from the union of states, transitions and guards that appear in the relevant transitions of M as follows:

$$N = N_s \cup N_t \cup N_g$$

where:

- N_s is the set of all *states* in the finite state machine that are a source state or target state of a relevant transition;
- N_t is the set of all relevant *transitions*;
- N_g is the set of all *guards* in the finite state machine that are non-trivial guards of a relevant transition.

The edges are derived from potential data flows among states, transitions and guards in the relevant transitions. Some of the edges represent actions that call methods from other classes. Each edge that results from a call to any external function is labelled with the sequence number of that call in the action sequence of the transition. It helps to distinguish these labels as being on outgoing or incoming



edges, so the sequence number label for an edge that represents an outgoing call of a mutator function is defined to be the OutSeq number and the sequence number label for an edge that represents an incoming data flow from an actor function, or from a mutator function that returns a value, is defined to be the InSeq number. All other edges will be left unlabelled. No edge carries more than one such label. In some cases an edge label will disambiguate multiple edge instances. An edge is represented as an ordered pair (n_1, n_2) or as an ordered triple (n_1, L, n_2) , depending on whether a label L is required to distinguish multiple edges from one node to another.

Nine types of edges are defined. Four come from the paper by Hong *et al.* [7] and are termed ‘intra-class’ edges because they are all defined within a single class. These intra-class edges are also *synchronous* in the sense that in all messages that are sent, the caller waits for the callee to complete before proceeding. To handle multiple classes, one new intra-class edge type and four new inter-class edge types are introduced. The inter-class edges are potentially *asynchronous* because each component is assumed to be a separate executable process. The new intra-class edge type that is introduced (E_{cts}) is also *asynchronous*, as explained below. The total set of edges E is defined as

$$E = E_{st} \cup E_{sg} \cup E_{gt} \cup E_{ts} \cup E_{gtg} \cup E_{sts} \cup E_{its} \cup E_{itt} \cup E_{cts}$$

Hong’s four original intra-class edge types are as follows.

- State-Transition (E_{st}) edges represent data flow from *states* to *transitions*. The transition has no non-trivial guard (i.e. guard is *true*).
- State-Guard (E_{sg}) edges represent data flow from *states* to *guards*. The state is the source state of the transition that specifies the non-trivial guard.
- Guard-Transition (E_{gt}) edges represent data flow from *guards* to *transitions*. The guard is non-trivial and is specified by the transition.
- Transition-State (E_{ts}) edges represent synchronous data flow from *transitions* to *states*. The state is the target state of the transition.

There are four types of edges between classes, which are potentially asynchronous. These are more complicated than intra-class edges. They are constructed when guards, states and transitions invoke methods in other classes. The invoking guard (g), state (s) or transition (t) may be the source or the target of the edge, depending on whether the data flow is into or out of that node.

- Guard-Transition-Guard (E_{gtg}) edges represent inter-class data flow, triggered by a *guard*, which flows from a *transition* in another class back to that *guard*. The predicate of the guard invokes an actor function from the other class and data flows from transitions in that class back to the guard. The *GuardRefActorFn* association determines these edges. The Automobile example has 34 instances of this type of edge.
- State-Transition-State (E_{sts}) edges represent inter-class data flow, triggered by a *state*, which flows from a *transition* in another class back to that *state*. The predicate of the state invokes an actor function from the other class and data flows from transitions in that class back to the state. The *StateRefActorFn* association determines these edges; the Automobile example has five instances.
- Inter-class-Transition-State (E_{its}) edges represent *inter-class* data flow from a *transition* to a *state* in a different class. The action of the transition invokes a mutator function from a different class and data flows from the transition to a feasible source state of any transition in that class



that has the mutator function as its method. The target of the flow is the source state rather than the other transition because it may be subject to the constraint of a guard and because it is not known which state the other object might be in when the request is received. These outgoing edges are labelled with an OutSeq number equal to the SeqNbr of the call of the mutator method in the action sequence of the calling transition. These edges are also labelled with the function name of the mutator function. The function label is used later in Section 6 as part of a constraint on certain path segments. The *ActionRefMutatorFn* association determines these labelled edges; the Automobile example has 174 instances.

- Inter-class-Transition-Transition (E_{itt}) edges represent *inter-class* data flow from a *transition* to a *transition* in a different class. The target transition action invokes a method from another class and data flows from all transitions in the class that are derived from the function back to the target transition. These incoming edges are labelled with an InSeq number equal to the SeqNbr of the method call in the action sequence of the calling transition. The *ActionRefMutatorFn* and *ActionRefActorFn* associations determine these labelled edges; the Automobile example has 68 instances.

There is one new intra-class asynchronous edge type.

- Class-Transition-State (E_{cts}) edges represent asynchronous intra-*class* data flow from *transitions* to *states*. The transition calls a mutator function, asynchronously, in its own class. Note that E_{ts} edges are synchronous. Since the call is asynchronous, it is put on a queue and the class may be in some other state when the function is executed. These outgoing edges are labelled with an OutSeq number equal to the SeqNbr of the method call in the action sequence of the transition. These edges are also labelled with the function name of the mutator function. The *ActionRefLocalAsyn* association determines these labelled edges; the Automobile example produces 80 instances.

A more formal specification of edge derivation is in Section 5 of an earlier technical report [30]. Transition nodes whose method has external availability determine the external interface to the system. Input values can only be provided through this interface in black box testing. In the Automobile example, the external methods listed in Section 2.2 produce all such externally invokable transitions. Various combinations of external methods with different inputs will produce different paths through the component flow graph. The goal is to find appropriate paths through the graph to ensure that all aspects of the specification are thoroughly covered and then to choose input values for a sequence of external methods to execute those paths. The paths through the graph are called *test specifications* and the sequences of external methods with appropriate input values are called *executable test cases*.

6. GENERATING TEST REQUIREMENTS

A *testing criterion* is a rule that imposes requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*: a test set achieves 100% coverage if it completely satisfies the criterion. Coverage is measured in terms of the requirements that are imposed; partial coverage is defined to be the percentage of requirements that are satisfied. *Test requirements* are specific things that must be satisfied or covered; for example, the requirements for statement coverage are individual statements that must be executed during testing.



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 [8,29]. This research follows the lead of other researchers and uses the *all-defs* and *all-uses* criteria [10,31–36]. Although some scientists 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.

6.1. Definition-use pairs

The formal definitions of variable *defs* and *uses* for the component flow graph are in a technical report [30] and given informally here. Finding *def-use* pairs for inter-class testing of object-oriented components is somewhat more complicated than for intra-class testing. This complication is caused by several factors, including the information hiding common in object-oriented classes (which makes it difficult to identify *defs* and *uses*), the multiple edge types that are being considered and the concurrent nature of the software. First, the various types of uses are defined. These include direct/indirect uses and predicate/computation uses. These are used to define *def-use* pairs. This paper deviates somewhat from traditional data flow testing papers by separating def-use pairs and DU-pairs. The purpose is to emphasize the fact that this research finds def-clear paths from definitions to uses and then analyses those paths to find input values that will execute the paths. A def-use pair that has a definition-clear path from the definition to the use is called a *DU-pair*.

Defs and uses are defined in terms of the associations defined in the database schema of Figure 4. Using the notation introduced in Section 3, let V be the set of all variables in the software system and let the variables be defined by the Greek nu, $\nu = (c, v) \in V$, where c identifies the class that declares the variable, that is, $c \in C$.

Definition 4. (Definitions and uses) Let M be any component of a software system, let $R(M)$ be the set of transitions that are relevant to M , and let $G = (N, E)$ be the component flow graph of M .

- ν is *defined* at a transition-node $n_t \in N_t$ if the variable and the transition are from the same class and if they satisfy the association $(c, t, \nu) \in ActionDefVar$. Each variable definition carries along the SeqNbr attribute of the *ActionDefVar* association.
- ν is *directly computation-used* at a transition-node $n_t \in N_t$ if the variable and the transition are from the same class and if they satisfy the association $(c, t, \nu) \in ActionRefVar$.
- ν is *indirectly computation-used* at a transition-node $n_t \in N_t$ if the variable is associated with the *get* method f in its class c and if the transition and the function satisfy the association $(c_t, t, c, f) \in ActionRefActorFn$.
- ν is *directly predicate-used* at any state-transition-edge if the state satisfies the association $(c, s, \nu) \in StateRefVar$.
- ν is *indirectly predicate-used* at any state-transition-edge if the variable is associated with the *get* method f in its class c and if the state and that function satisfy the association $(c_s, s, c, f) \in StateRefActorFn$.
- ν is *directly predicate-used* at any state-guard-edge if the state satisfies the association $(c, s, \nu) \in StateRefVar$.
- ν is *indirectly predicate-used* at any state-guard-edge if the variable is associated with the *get* method f in its class c and if the state and the method satisfy the association $(c_s, s, c, f) \in StateRefActorFn$.



- v is *directly predicate-used* at a guard-transition-edge if the transition satisfies $(c_t, t, c, v) \in \text{GuardRefVar}$.
- v is *indirectly predicate-used* at a guard-transition-edge if the variable is associated with the *get* method f in its class c and if the transition and f satisfy the association $(c_t, t, c, f) \in \text{GuardRefActorFn}$.
- v is *parameter computation-used* at a transition-node $n_t \in N_t$ if the action of the transition associated with n_t , called (c_t, t) , references the n th parameter of the function associated with t by name, that is, if $(c_t, t, n) \in \text{ActionRefParm}$ and if the variable is used to set the n th parameter of some function, that is, if there exists a transition t_1 whose action calls a function (c_f, f) such that $(c_{t_1}, t_1, c_f, f, n, c, v) \in \text{ActionSetsParmUsingVar}$ and if that function is the function associated with t , that is, if $c_t = c_f$ and $\text{method}(t) = f$.
- v is *parameter predicate-used* at a guard-transition-edge if the guard of the transition associated with n , called (c_t, t) , references the n th parameter of the function associated with t by name, that is, if $(c_t, t, n) \in \text{GuardRefParm}$ and if the variable is used to set the n th parameter of some function, that is, if there exists a transition t_1 whose action calls a function (c_f, f) such that $(c_{t_1}, t_1, c_f, f, n, c, v) \in \text{ActionSetsParmUsingVar}$ and if that function is the function associated with t , that is, if $c_t = c_f$ and $\text{method}(t) = f$.

Each *computation-use* instance carries along the *SeqNbr* attribute of the association to identify the position of that use in the action sequence of the transition. *SeqNbr* values are used later in rules that constrain the creation of candidate test paths. Since guard and state predicates do not have sequence numbers, *predicate-use* instances do not have such a value. These identifications of *defs* and *uses* in a component flow graph are used to define *def-use pairs* in those graphs. The Automobile example produces instances for each of these *def-use* categories, as listed in Section 7.1.

Definition 5. (Def-use pairs) Let M be any component of a software system, let $R(M)$ be the set of transitions that are relevant to M and let $G = (N, E)$ be the component flow graph of M . The Greek μ (μ) represents an edge or a node that is a use. An ordered pair (n_t, μ) is said to be a *def-use pair* for v if v is *defined at the transition-node* n_t and if μ is either a node or an edge in G where v is *directly* or *indirectly used*[‡].

Some variables have no def-use pairs. For example class constants may be defined when an object is created and never redefined in any relevant transition; others may be defined in a relevant transition as a non-relevant side effect but never used in any other relevant transition. All such variables are ignored in the following sections.

Transition nodes can both define and use a variable and which occurs first can affect later references to the variable. If a variable is used first, then defined, the definition from the node is relevant to definitions or uses of the variable in subsequent nodes. These cases are distinguished as follows.

Definition 6. (Internal def-use pairs) Let v be a variable that is both defined and used at one or more transition nodes $n_t \in N_t$. The set of nodes where v is *defined first and then used* is called $\text{DFTU}(v)$ and

[‡]Remember that a def-use pair is distinct from a DU-pair; the def-use pair may not have a def-clear path from the def to the use.



the set of all nodes where v is *used first and then defined* is called $\text{UFTD}(v)$. Both sets $\text{DFTU}(v)$ and $\text{UFTD}(v)$ can be determined by a syntactic analysis of the action associated with the transition node n_t .

The sets $\text{DFTU}(v)$ and $\text{UFTD}(v)$ are not necessarily mutually exclusive. A transition may have a use for a variable, then a definition, then another use (e.g. ‘ $x := x + 1; y := f(x)$ ’).

6.2. Data flow path coverage

A major goal of this research is to automate the generation of test data as much as possible. Most research in data flow testing focuses on recognising whether a set of tests cover *def-use* pairs as opposed to finding paths in the graphs that will allow *def-use* pairs to be covered. This project attempts to find paths in the following way. The algorithm looks for paths in the component flow graph that lead from the definition of a variable to a use. Consider triples (v, n_t, μ) where v is a variable, n_t is a transition node that defines v and μ is a node or edge where v is used. The nodes n_t and use item μ form a *DU-pair* if a path exists in the component flow graph leading from n_t to μ , if the path is free of loops, if there are no defs to v by another transition node in the path and if the path is potentially feasible for testing. The definitions in this section clarify these criteria as applied to testing of class components and lead to a rigorous definition of *test specifications* derived from a component flow graph.

Definition 7. (Path) Let $G = (N, E)$ be a directed graph with labelled edges. A *path* p in G of length $k \geq 1$ is a sequence of nodes and labels $n_1 L_1 n_2 \dots L_{k-1} n_k$ such that $(n_i, L_i, n_{i+1}) \in E$ for $1 \leq i \leq k - 1$. If p is a path, then the *head* of p , denoted by $H(p)$, is the first element of the sequence, the *tail* of p , denoted by $T(p)$ is the last element of the sequence and the *length* of p , denoted by $\text{Len}(p)$, is the number of nodes in the sequence. If p and q are two paths, and if L is a label such that $(T(p), L, H(q)) \in E$, then the concatenation of the two sequences, $p : q$, is a path with $\text{Len}(p : q) = \text{Len}(p) + \text{Len}(q)$. If p is a path and n is a node in the sequence that determines p , then n is said to be an element of p , denoted by $n \in p$. If p is a path, then $\text{InSeq}(p)$ or $\text{OutSeq}(p)$ denotes the label of its first or last edge. The context makes clear which is intended.

Only feasible paths through a component flow graph can be used, so special attention is paid to path segments in the graph that flow from a transition node n_{t1} to a state node n_s and then from that state node to a guard node n_g or to another transition node n_{t2} . If the edge from n_{t1} to n_s is the result of a call to a mutator function f , then the edge from n_s to n_g , or from n_s to n_{t2} , must satisfy some additional feasibility restrictions. In particular, the edge from n_s to n_g or n_{t2} must be from a transition whose function is identical to f and the guard predicate of any n_g must be compatible with the exit conditions from node n_{t1} or with the values of any parameters passed with f . The rules below address the function constraint. The guard constraint is more difficult to address because of exit conditions and dynamic values of passed parameters. To help address such guard constraints, a new association among these types of nodes is defined. A triple of nodes (n_t, n_s, n_g) is a *mutator Transition-State-Guard (TSG) path segment* if the edge from n_t to n_s has a function label. A mutator TSG path segment is *potentially feasible* if the edge from n_s to n_g is known to be compatible with the call of the mutator function. Let MTSG denote the set of all node triples that are *mutator* TSG path segments and let FTSG be the subset of MTSG consisting of TSG path segments that are *potentially feasible*. The Automobile example produces 317 instances of MTSG , of which 85 are provably feasible and 100 are provably not feasible, leaving 132 where a simple analysis cannot determine feasibility or non-feasibility. Appendix A shows the easy situations where a parameter is set to a literal in an action of a transition and the guards of some



of the transitions associated with the called function test that literal directly. The set FTSG contains all but the provably non-feasible triples (217 instances in the Automobile example).

Definition 8. (DU-path and DU-pair) Let $G = (N, E)$ be a component flow graph in a software system. Let v be any variable, let n_t be a transition node that defines v and let μ be a node or an edge where v is used. A path p in G is said to be a *DU-path* from n_t to μ for v if $p = n_t : q : \mu$, where q is a path in G such that no node of q is a definition node for v and every MTSG path segment in p is potentially feasible. The pair (n_t, μ) is said to be a *DU-pair* for v if such a path p exists. Edge labels for p are implicit.

Definition 9. (Candidate test paths) Let $G = (N, E)$ be a component flow graph in a software system. Let VDU be a set of tuples (v, n_t, μ) where (n_t, μ) is a def-use pair for v and let P be a set of tuples (v, n_t, μ, p) where (n_t, μ) is a *DU-pair* for v and p is a *DU-path* from n_t to μ . The set of all such paths p are the *candidate test paths* in G .

6.3. Finding all-uses candidate test paths

The *all-uses* testing criterion requires tests to execute at least one path from each definition to each reachable use. If there is more than one path from a def to a use, the strict interpretation of all-uses is that *any path* will satisfy the criterion. This research represents one of the few attempts to actively find paths, and it turns out that the details of how the algorithm is constructed can represent different choices in which path to use. An extreme choice would be to attempt *every path*. A choice that may save effort would be to use a *shortest path*. This may result in less testing and suggests the alternative of using a *longest path*. This may not be practical, so relaxing the choice to the *longest findable path* may be reasonable. One of the keys to testing this kind of software is evaluating the relationships among classes, which suggests the notion of finding a path that passes through *another class*, or a slightly more restrictive version, finding a path that passes through another class in the *test component*. If such a path cannot be found, it is necessary to relax to another choice, such as shortest path. These seven ways to choose paths are summarised in a subsumption hierarchy in Figure 5.

It is also possible that some paths could be ‘better’ in some sense than others. For example, it might be possible to incorporate a search procedure that uses some measurement function to choose from among a set of potential paths. One measurement might be to require that all MTSG path segments be known to be feasible instead of just known to be not infeasible, but that is a very difficult measurement to determine or represent. The construction described below looks for a shortest path because it is more convenient, thus saving computation expense.

It is easy to construct the set VDU of Definition 9, but the set P may not have any elements. An iterative procedure is defined to construct the elements of P beginning with definitions for P_1 and P_2 below. It searches for candidate test paths using a breadth-first algorithm for finding paths from one node to another in a directed graph, a modification of Dijkstra’s shortest-path algorithm that starts at both beginning and end nodes and meets in the middle. It works breadth-first from definition nodes and use nodes or edges, simultaneously forming two sets of partial paths. The *def-partial paths* are paths whose head is the definition node for a state variable and whose tail is a candidate node for connecting to a use of that variable. The *use-partial paths* are paths whose tail is a transition node where a variable is computation-used or whose last two tail nodes determine an edge where the variable is predicate-used and whose head is a candidate node for connecting to a definition of that state variable.

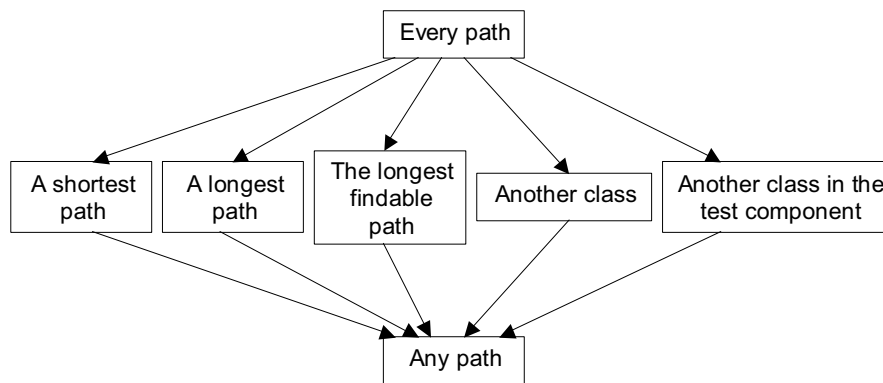


Figure 5. Subsumption hierarchy of choices for finding candidate test paths.

Each step of the algorithm looks for an edge that links the tail of a *def-partial path* for a state variable to the head of a *use-partial path* for that same variable. In addition, the algorithm ensures that all partial paths are *def-free* by requiring that the new candidate node added as the tail of a *def-partial path* or the head of a *use-partial path* does not define the variable. The algorithm enforces a rule that every MTSG path segment be potentially feasible. The algorithm also enforces a rule that *private* functions may only be called by methods within their own class and that *protected* functions may only be called by methods within their own component (if Java is used, this corresponds to a Java package). Also, if the action of a transition calls a private function within its own class, and if the next transition in the candidate path is a transition derived from the private function, the algorithm requires that the target state of the calling transition is the source state of the derived transition. A typical example of an action calling a private function is the asynchronous call of `CheckState()` as the final action of many methods in both `CruiseUser` and `Wheel`. Finally, both sets of partial paths are constructed to ensure that edges entering or leaving a transition node occur in a feasible order for the action sequence of that transition. These partial paths satisfy a set of rules involving `SeqNbr`, `InSeq` and `OutSeq` labels. They must be constructed to help ensure the construction of DU-paths that result in feasible test cases.

The following rules must be followed in the construction of *def-partial paths*:

- the first edge from a definition node of a state variable toward a new candidate intermediate node shall not be labelled with an `OutSeq` number that is less than the `SeqNbr` of that definition;
- if the incoming edge entering the tail of a *def-partial path* is labelled with an `InSeq` number equal to x , then the outgoing edge toward a new candidate intermediate node shall not be labelled with an `OutSeq` number less than x ;
- if the tail of a *def-partial path* is a state node, and if the incoming edge to that tail has a function label, then the outgoing edge to a new candidate intermediate transition or guard node shall be derived from the same function;



- if the tail of a def-partial path is a state node, and if the incoming edge to that tail has a function label, then the TSG path segment derived from the incoming transition node, the state node, and any new candidate intermediate guard node must be a potentially feasible MTSG path segment;
- if the tail of a def-partial path is a state node, and if the incoming edge to that tail does not have a function label, then any new candidate intermediate guard or transition node must not be derived from a *private* function;
- if the tail of a def-partial path is a transition node, and if the edge to any new candidate intermediate state node has a function label, then the class of the new state node must not be equal to the class of the transition node *or* the state of the new state node must be equal to the target state of the transition of the transition node.

The following rules must be satisfied in construction of *use-partial paths*:

- the first edge from a new candidate intermediate node toward a use node (or the lead node of a use edge) for a state variable shall not be labelled with an InSeq number that is greater than the SeqNbr of that use;
- if the outgoing edge from the head of a use-partial path is labelled with an OutSeq number equal to x , then the incoming edge from a new candidate intermediate node shall not be labelled with an InSeq number greater than x ;
- if the head of a use-partial path is a state node (i.e. with an outgoing edge to some adjacent guard or transition node) then any incoming edge from a new candidate intermediate node that has a function label must identify a function that is the same as the function associated with the adjacent guard or transition node;
- if the head of a use-partial path is a state node with an outgoing edge to some adjacent guard node, then any incoming edge from a new candidate intermediate node that has a function label must be from a transition node that forms a potentially feasible MTSG path segment with the state node and its adjacent guard node;
- if the head of a use-partial path is a state node, and if the function label on the outgoing edge from that state to its following guard or transition node identifies a *private* function, then any incoming edge from a new candidate intermediate transition node must have a function label that identifies the same *private* function;
- if the head of a use-partial path is a state node, and if the edge from any new candidate intermediate transition node has a function label, then the class of the new transition node must be not equal to the class of the state node *or* the state of the state node must be equal to the target state of the transition of the new transition node.

In the following definitions, the Q_{2k+1} iterations expand the *def-partial paths* and the Q_{2k+2} iterations expand the *use-partial paths*. The iterative algorithm begins with P_1 and P_2 , where P_1 identifies *DU-paths* for v from n_t to itself if n_t is a definition node and P_2 identifies *DU-paths* for v along a transition-to-transition edge:

$$P_1 = \{(v, n_t, n_t, n_t) \mid (v, n_t, n_t) \in \text{VDU AND } n_t \in \text{DFTU}(v)\}$$
$$P_2 = \{(v, n_t, n, n_t : n) \mid (v, n_t, n) \in \text{VDU AND } (n_t, n) \in E_{\text{itt}}\}$$

All paths in P_1 are of length 1 and all paths in P_2 are of length 2. In general, the sets P_i will identify paths of length i or $i \pm 1$. The definition of P_i for $i \geq 3$ depends upon sets of *partial paths*, Q_k .



and *unresolved def-use pairs*, X_j , both defined iteratively below. Each element of Q_k will be a tuple (v, n_t, μ, h, g) , where $(v, n_t, \mu) \in \text{VDU}$, h is a path from a def node n_t of v to an intermediate node and g is a path from some other intermediate node to a use item μ for v . Each X_j will be a subset of VDU , consisting of variable and def-use pairs that still do not have a connecting path. The algorithm begins with:

$$\begin{aligned} X_1 &= \text{VDU} \\ Q_1 &= \{(v, n_t, \mu, n_t, \mu) \mid (v, n_t, \mu) \in \text{VDU}\} \\ X_2 &= \text{VDU} - \{(v, n_t, n_t) \mid (v, n_t, n_t, n_t) \in P_1 \text{ AND } n_t \notin \text{UFTD}(v)\} \\ Q_2 &= \{(v, n_t, \mu, n_t, \mu) \mid (v, n_t, \mu) \in X_2\} \end{aligned}$$

Given Q_i , ($i \geq 2$) the next step defines:

$$\begin{aligned} P_{i+1} &= \{(v, n_t, \mu, h : g) \mid (v, n_t, \mu, h, g) \in Q_i \\ &\text{AND } e = (T(h), H(g)) \in E, \text{ or } e = (T(h), L, H(g)) \in E \text{ for some edge label } L, \\ &\text{AND InSeq}(h) \leq \text{OutSeq}(e) \text{ AND InSeq}(e) \leq \text{OutSeq}(g) \\ &\text{AND } (\neg \exists \text{Fn}(\text{Pre}(T(h)), T(h))) \text{ OR } \text{Fn}(\text{Pre}(T(h)), T(h)) = \text{method}(H(g))) \\ &\text{AND } ((\text{Pre}(T(h)), T(h), H(g)) \notin \text{MTSG} \text{ OR } (\text{Pre}(T(h)), T(h), H(g)) \in \text{FTSG}) \\ &\text{AND } (\neg \exists \text{Fn}(e) \text{ OR } \text{Fn}(e) = \text{method}(\text{Pre}(H(g)))) \\ &\text{AND } ((T(h), H(g), \text{Pre}(H(g))) \notin \text{MTSG} \text{ OR } (T(h), H(g), \text{Pre}(H(g))) \in \text{FTSG}) \\ &\text{AND } ((T(h) \in N_s \text{ AND } \neg \exists \text{Fn}(\text{Pre}(T(h)), T(h))) \rightarrow \text{availability}(\text{Fn}(e)) \neq \text{PRI}) \\ &\text{AND } ((H(g) \in N_s \text{ AND } \text{availability}(\text{Fn}(H(g), \text{Pre}(H(g)))) = \text{PRI}) \rightarrow \exists \text{Fn}(e)) \\ &\text{AND } ((T(h) \in N_t \text{ AND } H(g) \in N_s \text{ AND } \exists \text{Fn}(e) \rightarrow (\text{class}(T(h)) \neq \text{class}(H(g)) \\ &\text{OR state}(H(g)) = \text{targetState}(T(h))))\} \\ C_{i+1} &= \{(v, n_t, \mu) \mid \exists p[(v, n_t, \mu, p) \in P_{i+1}]\} \\ A_{i+1} &= \{(v, n_t, \mu) \mid \exists h, g[(v, n_t, \mu, h, g) \in Q_i]\} \\ X_{i+1} &= A_{i+1} - C_{i+1} \\ B_{i+1} &= X_i - A_{i+1} \end{aligned}$$

The sets C_{i+1} identify unsolved *def-use pairs* in X_i that have found a *DU-path* in P_{i+1} . The sets A_{i+1} identify unsolved *def-use pairs* in X_i that remain active candidates for resolution in P_{i+1} . The sets B_{i+1} identify *def-use pairs* that drop out of consideration for solution at this step of the iteration; the variable they are associated with is said to be *def-bound*.

Definition 10. (def-bound) A variable v is said to be *def-bound* at a definition node n_t of a *def-use pair* (n_t, μ) if there is no path p from n_t to μ , where $p = (v, n_t, \mu, p) \in P$.



The sets of *partial paths* are defined iteratively as follows. Given Q_{2k} , the *def-partial paths* are extended by defining:

$$\begin{aligned} Q_{2k+1} = \{ & (v, n_t, \mu, h : n, g) \mid (v, n_t, \mu, h, g) \in Q_{2k} \text{ AND } \exists n, L [n \in N \text{ AND } (T(h), L, n) \in E \\ & \text{AND } n \notin D(v) \\ & \text{AND } T(h) : n \notin h \text{ AND } (v, n_t, \mu) \in X_{2k+1} \\ & \text{AND } \text{InSeq}(h) \leq \text{OutSeq}(T(h), n) \\ & \text{AND } (\neg \exists \text{Fn}(\text{Pre}(T(h)), T(h))) \text{ OR } \text{Fn}(\text{Pre}(T(h)), T(h)) = \text{method}(n)) \\ & \text{AND } ((\text{Pre}(T(h)), T(h), n) \notin \text{MTSG} \text{ OR } (\text{Pre}(T(h)), T(h), n) \in \text{FTSG}) \\ & \text{AND } ((T(h) \in N_s \text{ AND } \neg \exists \text{Fn}(\text{Pre}(T(h)), T(h))) \rightarrow \text{availability}(\text{Fn}(T(h), n)) \neq \text{PRI}) \\ & \text{AND } ((T(h) \in N_t \text{ AND } n \in N_s \text{ AND } \exists \text{Fn}(T(h), n)) \rightarrow (\text{class}(n) \neq \text{class}(T(h)) \\ & \text{OR } \text{state}(n) = \text{targetState}(T(h))))\} \end{aligned}$$

Given Q_{2k+1} , the *use-partial paths* are extended by defining:

$$\begin{aligned} Q_{2k+2} = \{ & (v, n_t, \mu, h, n : g) \mid (v, n_t, \mu, h, g) \in Q_{2k+1} \text{ AND } \exists n, L [n \in N \text{ AND } (n, L, H(g)) \in E \\ & \text{AND } n \notin D(v) \\ & \text{AND } n : H(g) \notin g \text{ AND } (v, n_t, \mu) \in X_{2k+2} \\ & \text{AND } \text{InSeq}(n : H(g)) \leq \text{OutSeq}(g) \\ & \text{AND } (\neg \exists \text{Fn}(n, H(g))) \text{ OR } \text{Fn}(n, H(g)) = \text{method}(\text{Pre}(H(g))) \\ & \text{AND } ((n, H(g), \text{Pre}(H(g))) \notin \text{MTSG} \text{ OR } (n, H(g), \text{Pre}(H(g))) \in \text{FTSG}) \\ & \text{AND } ((H(g) \in N_s \text{ AND } \text{availability}(\text{Fn}(H(g), \text{Pre}(H(g)))) = \text{PRI}) \rightarrow \exists \text{Fn}(n, H(g))) \\ & \text{AND } ((n \in N_t \text{ AND } H(g) \in N_s \text{ AND } \exists \text{Fn}(n, H(g))) \rightarrow (\text{class}(n) \neq \text{class}(H(g)) \\ & \text{OR } \text{state}(H(g)) = \text{targetState}(n)))\} \end{aligned}$$

where $D(v)$ is the set of *definition nodes* for v , $\text{Fn}(e)$ is the function label of an edge e , $\text{Pre}(T(h))$ is the preceding node adjacent to $T(h)$ in h , and $\text{Pre}(H(g))$ is the following node adjacent to $H(g)$ in g . InSeq and OutSeq inequalities are satisfied if either label is null. The partial paths with odd subscripts are building partial paths from the def node, whereas the partial paths with even subscripts are building partial paths from the uses.

In the construction of paths, it is possible that a path will exit a node derived from class instance C , leaving C in state S_1 , to enter a node derived from another class instance and then later return to the first node while C is in a new state S_2 . This is possible only if some other action has caused the state of C to change from S_1 to S_2 . To capture this other action as part of the path, a rule is enforced at several points in the above process to ensure that paths may only exit and then re-enter a node derived from a class instance while the state of the class instance is the same. This rule is called the *state compatibility rule*. Experimentation has shown that enforcement of this rule substantially reduces the number of non-feasible candidate test paths at the expense of increasing the number of unsolved def-use pairs.

The iterative process stops when $X_i = \emptyset$. At this point, set $P = \cup P_i$. This must happen for some value of i less than the number of edges in the graph since cycles were avoided by ensuring that no edge appears more than once in any of the partial paths. It is possible for some state nodes and some transition nodes to appear more than once in a partial path. Not all elements $(v, n_t, \mu) \in \text{VDU}$ will



yield a *DU-path*. Some variables may be defined at a node n_t and used at a use item μ but either no path exists from n_t to μ that satisfies the above constraints or every such path contains a new definition of v . If all-defs is being satisfied then another use is found. If all-uses is being satisfied, this def-use pair is infeasible.

The *def-bound* variables surface during the calculation of $B_{i+1} = X_i - A_{i+1}$ in the above iterative process. At that point, $C_{i+1} \subseteq A_{i+1} \subseteq X_i$. It follows that B_{i+1} identifies the def-use pairs that were active during the calculation of X_i , did not find a path to join in P_{i+1} , yet are no longer active for X_{i+1} . They dropped out because in the calculation of the previous Q_i , there was no node n to form a new edge in the partial paths. Thus the sets B_{i+1} identify new *def-bound* variables, if they exist, at each step of the process.

6.4. Executable test cases

If a variable v is both defined and used and is not *def-bound* for a specific *def-use* pair then the path generation of the previous section may produce one or more DU-paths linking a definition node n_t to its corresponding use item μ . These DU-paths are considered to be *abstract test specifications* because no attempt has yet been made to choose explicit parameter values for any of the function calls. There is no guarantee that an abstract test specification will be feasible because it may contain a TSG path segment that is not feasible. However, the process carries along all possible potentially feasible TSG path elements for each *def-use* pair, so there is a good chance that a feasible path element will be in the collection P of candidate test paths constructed by the algorithm of Section 6.3. If at the end of iteration i , all DU-paths for a DU-pair are found to be infeasible then it is necessary to look for DU-paths of length $i + 1$. The DU-pair is re-inserted into the set of active pairs X_i and the algorithm continues by looking for longer DU-paths.

Even at the end of this process, there is no guarantee that a feasible abstract test specification will lead to an executable test case. One must still find externally invocable methods that will trigger each of the function calls in the abstract test specification without violating any of the constraints against redefinition of the state variable. The iterative process of Section 6.3 is modified to help construct executable tests from test specifications. Instead of beginning with *defs* and *uses*, the modified process replaces *defs* with the collection of transition nodes derived from external functions and replaces *uses* with the collection of transition nodes derived from non-external functions. The modified process also eliminates edges that allow a path to enter a terminal target state. The result is a collection of candidate execution paths in the component flow graph from transition nodes with external methods to all other relevant transition nodes. In this manner an appropriate external function can be chosen to invoke a given transition-to-transition or guard-to-transition edge in a candidate test path. With some manual intervention to choose appropriate guard alternatives, it is then possible to select a sequence of external functions that force traversal of many candidate test paths. Some results from using this approach on the Automobile system are given in Section 7.3. Subsequent research will attempt to use this methodology to automate the process of generating executable test cases from abstract test specifications more fully.

The entire process 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 [37–40]. Figure 6 attempts to illustrate the possibilities.

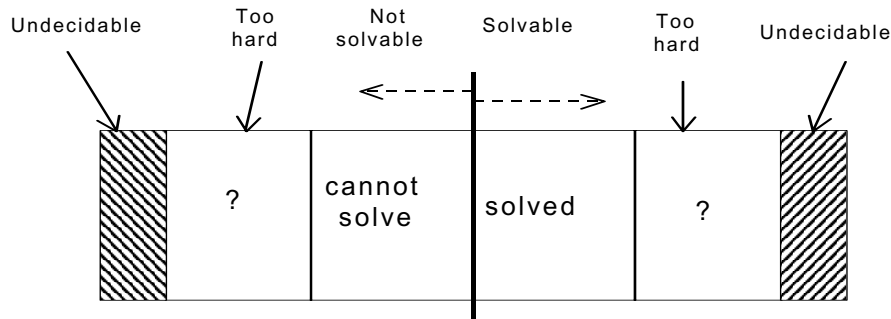


Figure 6. Possibilities when finding candidate test paths and executable test cases.

Some *def-use* pairs have solutions (*solvable*) and some do not (*not solvable*). Some of these can either be solved or recognized as being unsolvable (*solved*, *cannot solve*) by the existing tool. There are other *def-use* pairs that are either too hard to solve or too hard to prove to be unsolvable (*too hard*) by the current tool, and finally there are *DU-paths* for which proving them to be unsolvable or finding solutions is truly undecidable (*undecidable*). This situation is common to all automatic test data generation techniques, which by definition can never be perfect for all situations. The goal of any technique is to increase the number of test requirements that can either be solved or shown to be unsolvable as much as possible, with the explicit recognition that even if the problem is not solved completely, it can still be possible to create good quality tests.

7. EMPIRICAL RESULTS FOR THE AUTOMOBILE SYSTEM

This section presents empirical results from applying the inter-class data flow technique to the Automobile system. Tools have been developed that automate most of the process shown in Figure 3. The database representation of the specification must be created by hand and the final step of generating executable test sequences is only partially automated. As an experimental evaluation, tests were constructed and run on seeded faults and the fault-finding ability of the tests on the seeded faults was evaluated and compared against tests generated by hand. The subjects (full Automobile specifications and tests) are provided in a technical report [41]. This section first illustrates an application of the technique by way of example using the Automobile system, then describes the tool support and finally the case study on fault detection.

7.1. Applying the technique to the Automobile system

Simple versions of Cruise have been used widely in the specification, specification-based testing and modelling literature but the version used in this paper includes significantly more components than other versions, such as in references [42–44]. For example, the version used by Atlee [43]



and Abdurazik *et al.* [42] had one class, four states, seven functions, 184 blocks and 174 decisions. The external interface and the cruise control transitions used in this paper are modelled on the cruise control characteristics of an automobile owned by the first author[§]. Instead of the four states found in the other papers, the system used in this paper contains 12 classes for a total of 44 states. Combined, these states have 43 relevant variables that appear in more than 4300 def-use pairs. For cruise control testing purposes, only external functions such as ignition, brake and gas pedal positions and cruise controls are available to human users and testers. Other functions are encapsulated and hidden.

Each process from Sections 3–6 is illustrated on the CruiseControl below. The contents of the initial tables are as follows.

| | |
|------------------|--|
| Class table | 12 rows—one for each class in the system |
| Variable table | 58 rows—with 9 for CruiseControl |
| Parameter table | 44 rows—with 7 for CruiseControl |
| Function table | 106 rows—with 20 for CruiseControl |
| State table | 44 rows—with 12 for CruiseControl |
| Transition table | 263 rows—with 91 for CruiseControl |

The syntactic analysis from Section 3 on the predicate and action attributes of these tables yields the following association tables.

| | | | |
|-------------------------|---------------|-------------------------------|---------------|
| <i>StateRefVar</i> | 67 instances | <i>ActionRefMutatorFn</i> | 116 instances |
| <i>StateRefActorFn</i> | 5 instances | <i>VarAssocFn</i> | 44 instances |
| <i>ActionDefVar</i> | 252 instances | <i>ActionRefLocalAsyn</i> | 22 instances |
| <i>ActionRefVar</i> | 172 instances | <i>GuardRefParm</i> | 121 instances |
| <i>GuardRefVar</i> | 117 instances | <i>ActionRefParm</i> | 80 instances |
| <i>GuardRefActorFn</i> | 17 instances | <i>ActionSetsParm</i> | 70 instances |
| <i>ActionRefActorFn</i> | 32 instances | <i>ActionSetsParmUsingVar</i> | 70 instances |

Section 4 describes how to derive transitions in each existing class that are *relevant* to CruiseControl. In some cases only a few transitions are relevant; for example, the only state of BrakeControl that is relevant to CruiseControl is whether or not the brakes are engaged. When the brakes are engaged, a message is sent to AutoSystem and AutoSystem sends a Cancel message to CruiseUnit. The relevant transitions are derived iteratively from labelled transitions, as defined in Section 4.

| | |
|----------|---|
| R_{00} | 91 CruiseControl <i>Base</i> transitions |
| R_0 | 68 feasible <i>Base</i> transitions |
| R_1 | 194 labelled transitions at first iteration |
| R_2 | 260 labelled transitions at second iteration |
| R_3 | 292 labelled transitions at third iteration |
| R_4 | 308 labelled transitions at fourth iteration |
| R_5 | 314 labelled relevant transitions at final pass |
| $R(M)$ | 160 final unlabelled relevant transitions |

This process leaves 75 feasible transitions that are not relevant, including oil and water pressure gauges that were intentionally excluded from consideration, anti-lock brake actions that have no

[§]National Institute of Standards and Technology (NIST) policy prohibits publication of the make or model of the car.



relevance after the brakes become active, various transmission actions dealing with neutral and reverse and other actor methods in several classes that could not impact on CruiseControl.

$R(M)$ has 105 transitions that have non-trivial actions, as displayed in Appendix A. The component flow graph defined in Section 5 is summarized as follows.

| Nodes | 293 nodes | Edges | 740 edges |
|-----------------|-----------|-----------------|-----------|
| TransitionNodes | 160 | E_{st} Edges | 49 |
| StateNodes | 44 | E_{sg} Edges | 85 |
| GuardNodes | 89 | E_{gt} Edges | 85 |
| | | E_{ts} Edges | 160 |
| | | E_{gtg} Edges | 34 |
| | | E_{sts} Edges | 5 |
| | | E_{its} Edges | 174 |
| | | E_{itt} Edges | 68 |
| | | E_{cts} Edges | 80 |

The following is the list of defs and uses from the CruiseControl component of the Automobile example in this paper (Sections 6.1 and 6.2).

| | |
|-------------------------|------|
| DefnNodes | 188 |
| DirectCompUses | 131 |
| IndirectCompUses | 30 |
| DirectPredUseByState | 304 |
| DirectPredUseByGuard | 80 |
| IndirectPredUseByState | 32 |
| IndirectPredUseByGuard | 17 |
| ParmUseByGuard | 205 |
| ParmUseByTransition | 216 |
| Total VarDefUse triples | 4319 |
| DFTU nodes | 45 |
| UFTD nodes | 17 |
| DFTU & UFTD | 17 |

From Section 6.3, the following are MTSG path segments.

| | |
|------------------|-----------|
| MTSG triples | 317 paths |
| KnownFeasible | 85 |
| KnownNotFeasible | 100 |
| Indeterminate | 132 |
| FTSG triples | 217 paths |

The above calculations are almost instantaneous, even on a desktop personal computer. However, the next step, to calculate the *DU-paths* leading to DU-pairs becomes much more computationally intensive. The iterative process to construct candidate test paths P_i from Definition 9 proceeds in several steps. Each step is summarised below.

P_1 45 rows (from the DFTU nodes)



Only 45 of the 62 *def-use* pairs with an action that both defines and uses the same variable are DFTU. For example, the CruiseUnit transitions t016 and t064 in Appendix A define and then use TargetThrottle and CurrentSpeed. Likewise, the GasUser transition t002 first defines and then uses PedalPosition and Engine t003 first defines and then uses Rpm. All relevant Throttle transitions define and then use Position. Examples of UFTD transition nodes are CruiseUnit transitions t030 and t031, which both use TargetSpeed and then define it. Throttle transitions t004, t007 and t009 have a parameter-use of Position as a result of calls to Floor() from CruiseUnit and then immediately define Position. Some of the *def-use* pairs are in both DFTU and UFTD and are kept active in the search for new *DU-paths* from those nodes back to themselves. The remaining *def-use* pairs in DFTU have a *use* that can never be reached from outside that node without redefining of the variable, immediately resulting in 175 *def-bound* pairs.

P_2 0 rows (none of the 68 E_{itt} edges give DU-pairs)

There are 68 transitions having actions that return a value from a call to a function in some other class. None of these calls involve mutator functions, so they do not result in any new DU-pairs with a test path of length 2. Thus P_2 is empty. Continuing with initialization of the DU-pair generation process yields the following results.

| | | |
|-------|----------------|--|
| X_1 | 4319 instances | (from VDU triples) |
| Q_1 | 4319 instances | (from VDU—set Head and Tail) |
| B_1 | 175 instances | (from 45 DFTU pairs not in UFTD) |
| X_2 | 4099 instances | (removing 45 found and 175 <i>def-bound</i> pairs) |
| Q_2 | 4099 instances | (removing same 220 instances from Q_1) |

At iteration 3, the algorithm finds 414 candidate test paths of lengths 3 or 4. Approximately one-half of the paths go through the target state of their transitions to an outgoing edge from that target state. Most of these yield either an easy feasible path or an obvious non-feasible path that should have been identified in the initial class definition. The other half of the paths do not go through the target state of the transition; instead, they follow a mutator function to the source state of some other transition in the same class. All of these MTSG path segments are based on a call to CheckState() in the CruiseUnit or Wheel classes. For some of these, it is relatively easy to find an executable test case. Others yield an obvious non-feasible path that should have been identified in the transition-state-guard discussion following Definition 7. At this step, all paths are still completely contained in a single class. All 414 new paths result in new DU-pairs. Seventeen new *def-bound* variables are identified: BrakeActive is defined in AutoSystem transitions but can never reach their predicate uses in some Engine states, IsActive is defined in both BrakeControl and BrakeUser but can never reach its use in state predicates. In all cases a Cancel() message gets sent instead to shut down all further CruiseControl processing.

Iteration 4 creates 341 new paths of lengths 3 and 4, all of which identify new DU-pairs. The paths may be handled similarly to the paths in the previous iteration, although some of the mutator functions force actions in other classes. For each mutator transition in the path, the process described in Section 7.2 is used to find external (user-level) functions that will trigger that transition. The difficulty is to find a function that does not redefine the *def-use* variable. It is often necessary to use a longer indirect external function, such as ExternalDrag(), to cause changes of state in CruiseUnit without redefining the *def-use* variables under test. This iteration also discovers 231 *def-bound* pairs.



Table II. CruiseControl—candidate test paths.

| | New paths P_i | New DU-pairs C_i | Active pairs X_i | New DefnBnd B_i | Partial paths Q_i | Process time (h:mm:ss) |
|--------|--------------------|-----------------------|-----------------------|----------------------|------------------------|---------------------------|
| 1 | 45 | 45 | 4319 | 175 | 4319 | 0:02 |
| 2 | 0 | 0 | 4099 | 0 | 4099 | 0:01 |
| 3 | 414 | 414 | 3668 | 17 | 9332 | 2:57 |
| 4 | 341 | 341 | 3096 | 231 | 30604 | 3:06 |
| 5 | 536 | 502 | 2519 | 75 | 49898 | 4:17 |
| 6 | 194 | 190 | 2304 | 25 | 93116 | 4:14 |
| 7 | 157 | 146 | 2036 | 122 | 58290 | 3:01 |
| 8 | 201 | 132 | 1645 | 259 | 111942 | 5:38 |
| 9 | 124 | 104 | 1414 | 127 | 166281 | 9:05 |
| 10 | 124 | 84 | 1314 | 16 | 378383 | 12:06 |
| 11 | 107 | 72 | 1185 | 57 | 747816 | 20:27 |
| 12 | 18 | 8 | 1177 | 0 | 1083242 | 39:41 |
| 13 | 50 | 12 | 1137 | 28 | 1579445 | 50:36 |
| 14 | 3 | 1 | 1136 | 0 | 3056216 | 2:03:35 |
| 15 | 32 | 8 | 1010 | 118 | 4476452 | 3:32:55 |
| 16 | 10 | 1 | 873 | 136 | 7589298 | 6:07:22 |
| 17 | 0 | 0 | 802 | 71 | 12951518 | 10:16:13 |
| 18 | 16 | 3 | 795 | 4 | 19938236 | 16:32:52 |
| 19 | 0 | 0 | 762 | 33 | ~33114919 | ~31:44:50 |
| 20 | 0 | 0 | 762 | 0 | — | — |
| Totals | 2372 | 2063 | | 1494 | | ~74:00:00 |

The process takes multiple iterations, as shown in Table II. The ‘New paths’ column corresponds to new candidate test paths. Sometimes multiple paths are found for the same def-use pair so the ‘New DU-pairs’ column corresponds to def-use pairs that have found a path and thereby become DU-pairs. The ‘Active pairs’ column shows the number of pairs for which no path has been found up to and including the current iteration. The ‘New DefnBnd’ column identifies the number of *def-use* pairs determined to be *def-bound* at this iteration. The ‘Partial paths’ column identifies the number of partial paths existing at the end of the current iteration. The ‘Process time’ column is for each iteration, and is from the prototype implementation using an Access database on a Pentium 4 class PC at 2 GHz with 1 GB RAM.

The majority of DU-path generation takes place in iterations 3–10 (with total processing time less than 45 minutes) but some new DU-paths continue to be generated through to iteration 18. Time and space costs for obtaining new information prevent the tool from continuing past iteration 20. Out of a total of 4319 def-use pairs, 2372 candidate test paths were found to cover 2064 pairs (47%) and 1494 (35%) pairs were shown to be def-bound, with no def-free candidate test paths. Thus, 82% (3557) of the def-use pairs were resolved (with a test or shown to be infeasible), leaving only 18% (762) unknown. This experimental difficulty in solving 18% of all def-use pairs is consistent with the limitation stemming from undecidability and the theoretical discussion in the last paragraph of Section 6.4. However, as shown in Table III in Section 7.3, the resulting tests were good at finding faults.



Table II shows that iteration 5 finds 536 new DU-paths but only 502 of them identify new DU-pairs. In addition, 75 pairs were found to be *def-bound* (B_i). The number of active pairs (X_i) is thus reduced by 577. Many of the paths are similar to the above, either composed of successive application of feasible transitions within a class, going through the target state of one transition to the source state of the next or involving interactions between classes via calls of mutator functions along MTSG edges.

Multiple DU-paths can occur for the same DU-pair in several different ways. This can occur when a transition gets the values of multiple variables from the same class so each call generates a separate potential path; this occurs in Ignition transition t003, since it calls both ThrottleFloor and ThrottleGovernor from AutoSystem where both variables are defined by transition t001. This also occurs in CruiseUnit transition t025, where the guard predicate calls both Position and Floor from Throttle where both variables are defined by transition t004. Another source of multiple DU-paths for the same DU-pair is when a guard predicate identifies multiple choices in the interior of a path, thereby generating separate subpaths to reach the same def and use; this occurs often for variables defined in CruiseUnit where a path may travel through multiple states of throttle, without the variable being redefined, before returning to a use in CruiseUnit.

More interesting paths begin to occur after iteration 5. For example, the following path involving Throttle (c10) and CruiseUnit (c05) begins with Throttle.Position defined by Throttle.Floor(x) at transition node c10t005 and ends with Position used by the action of CruiseUnit.UserMode(SD/NT) at transition node c05t043:

```
c10t005:TS:c10s01:ST:c10t016:TT:c05t064:TS:c05s03:SG:c05gt030:GT:c05t030:TS:c05s05:SG:c05gt043:GT:c05t043
```

After the definition of Position, the path follows a transition-to-state (TS) edge to throttle state node c10s01 (Idle), where it waits for a subsequent action. The action of CruiseUnit.UserMode(SD/NT) at transition node c05t064 calls Throttle.Position() via transition node c10t016 along a transition-to-transition (TT) edge, coming to rest at state node c05s03 (Cruise) to wait for another subsequent action. The only way this can happen is for the automobile to be in a high gear and going downhill so that the idle speed in the throttle is sufficient to maintain a speed greater than SlowCutoff to satisfy the guard predicate of c05t064, thereby allowing an Override-to-Cruise state transition. The next action of CruiseUnit.UserMode(SD) at transition node c05t030 leads the path through state-to-guard (SG) and guard-to-transition (GT) edges, putting the CruiseUnit into its Decel state at state node c05s05. The user holds down the SD button to achieve this deceleration, then releases it, thereby calling Usermode(NT), to effect the final action and usage of the Position variable in the action at transition node c05t043, causing a Decel-to-Cruise state transition.

Using methods introduced in Section 6.4 and further described in Section 7.2, the above *abstract test specification* can be used to generate the following *executable test sequence*. While in Cruise state at highway speed in a high gear and going downhill, the following sequence of actions are performed:

- (1) CruiseUser.Cancel () Puts CruiseUnit in the Override state and calls Throttle.Floor(0), which defines Throttle.Position = fconst and puts Throttle in the Idle state;
- (2) Pause Waits for the Engine to settle at Idle speed while maintaining Gauges.Speed > CruiseUnit.SlowCutoff;
- (3) CruiseUser.Mode (SD) Sets the CruiseUnit.UserMode variable to SD with no other effect;



-
- (4) CruiseUser.Mode (NT) Causes an Override-to-Cruise transition (t064) when button is released;
 - (5) CruiseUser.Mode (SD) Causes a Cruise-to-Decel transition in CruiseUnit while maintaining Throttle in Idle without redefining Position;
 - (6) CruiseUser.Mode (NT) Causes a Decel-to-Cruise transition (t043) and uses Throttle.Position() to set the TargetThrottle variable.

7.2. Status of automation

At least four different testing activities can benefit from automation. First is the selection and generation of test data, second is the evaluation of the test data (e.g. according to a test criterion), third is the identification and generation of expected results and fourth is the execution of the tests. The first activity is widely considered to be the hardest to automate and is supported by the fewest number of tools, both research and commercial. The goal of this research is to automate the generation of test data as much as possible.

Most previous research in data flow testing has focused on the intra-method problem and is usually based on the source code. At the inter-method level (testing multiple methods together), Harrold and Soffa [10] tried to generate tests that covered def-use pairs between two procedures, focusing on the problems associated with inter-method analysis. Jin and Offutt [17] defined a limited form of data flow testing between pairs of methods that only considered certain definitions and uses (first-uses and last-defs). This work was again based on control flow graphs from the program. Alexander and Offutt [18,19] defined def-use pairs among program classes and identified the various possibilities that could occur in the presence of polymorphism and dynamic binding, based on the *polymorphic call set*. The current research is based on the specifications (finite state machines) rather than the code and goes beyond most of the previous papers. All of these papers identified def-use pairs. Only this paper and the work by Alexander and Offutt identify candidate test paths and this paper goes a step further in identifying test path specifications. Work on generating test values has previously been limited to the inter-method case; research into intra-class and inter-class testing has not tried to automatically generate values but has instead focused on generating sequences of method calls where parameters are assigned values by hand or created randomly.

Consider the testing architecture shown in Figure 7. The Test Sequence Generator of Figure 3 is represented as one tool in a collection of tools designed to automate as much of the testing process as possible. A database implementation of the specification is used to separately create a reference implementation or generation of executable test sequences. For each test sequence, the Test Harness executes the external methods in the prescribed order against the reference implementation and at the end of each execution compares the predicted state of each class in the system with the actual state in the implementation.

This process usually finds errors in the specification; fixes can then be applied to the database representation and the process repeated. At an appropriate time, a real implementation can be substituted for the prototype with the same set of test sequences as in the test suite. An implementation passes the test suite if the actual states of the classes at each step match the predicted states. The Test Harness records each mis-match for subsequent analysis. The Graphical User Interface is an optional component that allows a tester to apply *ad hoc* testing procedures or to visualise the effect of changes to the specification. At this point the visualisation is tied to the Automobile example but could be partially

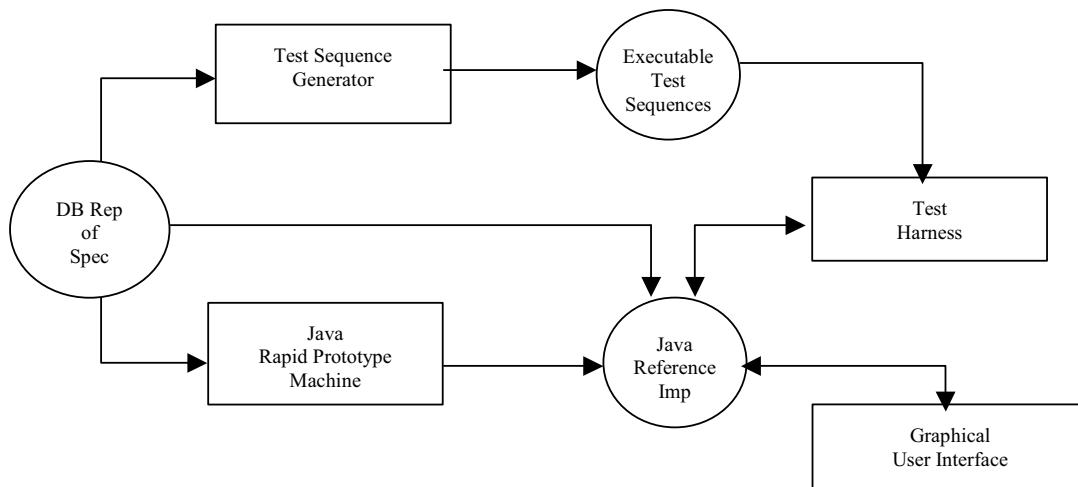


Figure 7. Testing tool architecture.

generalised on a class-by-class basis to provide rapid visualisations for all software systems derived from a base set of classes or components.

The Rapid Prototype Machine (RPM) is a generic test simulator written in Java. The machine consists of a simple kernel 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 class called a *MessageObject*. The *MessageObject* class is a wrapper class that stores data fields traditionally associated with object-oriented programming, such as an object's identity, state and behaviour, applied to that object (a function). The object in question is an instance of a *ClassObject*. Each *ClassObject* instantiated in the RPM is defined by the class, state and variable tables in the database specification. Each *ClassObject* retains a queue of *MessageObject*s, named a *CallQueue*. *CallQueues* are threaded, interruptible, first-in, first-out data structures capable of waiting for, storing and passing *MessageObject*s to an RPM interpreter. The RPM interpreter evaluates a *MessageObject* by inspecting the identity, state and behaviour fields stored in the *MessageObject*. Next, the RPM interpreter queries the database specification's transition table, using a combination of these fields and then proceeds to further evaluate the transition actions stored there. This chain of events may result in: (1) a value returned by the RPM interpreter (accessor function); (2) a *ClassObject* property changed by the RPM interpreter (mutator function); or (3) a combination of the two. The RPM provides an interface for test writers to add visual components for simulation purposes. By implementing the interface, test writers are given access to RPM interpreter return values and *ClassObject* states, can instantiate and evaluate their own *MessageObject*s and can update custom visualisations appropriately.

The Test Harness is designed to support testers who want to run a sequence of test cases under the RPM or against a real implementation inserted into the testing architecture in place of the reference implementation. It is a threaded Java component that sequentially evaluates executable test methods



stored in a database table. The Test Harness requires the test writer to create two application-specific tables; in Figure 7 this is done by the Test Sequence Generator. The first table is a list of test methods to execute under the RPM. The second table is a list of the states to which each ClassObject should undergo a transition after applying a given method. For each method in the list of test methods, the Test Harness instructs the relevant ClassObject to create a MessageObject and place that object on its CallQueue. The MessageObject is evaluated by the RPM interpreter, causing any affected ClassObjects to undergo a transition to an appropriate state as prescribed by the actions codified in the message. After a specified transitional period, the Test Harness interrupts all system CallQueues and samples the state of each RPM ClassObject. Each ClassObject state is compared to the expected ClassObject state in the second table, described above. If the set of all RPM ClassObject states for a given test method is equivalent to the set of all expected states for that method, the system records a passing test, restarts the CallQueues and moves on to the next test method. If, however, the set of all ClassObject RPM states for a given test method is not equivalent to the set of all expected states for that method, the system records a ClassObject state inconsistency, restarts the CallQueues and moves on to the next test method. The preceding process is repeated until all test method items in the table have been executed or until an action causes a fatal system error.

The last step in developing executable test sequences uses the ideas discussed in Section 6.4 to identify methods that can be invoked externally and that lead directly to a transition node in the component flow graph. Given a *system state*, that is, the state of each class instance in the system and a transition node selected to be the next action node in a candidate test path, the partial implementation eliminates all paths from external methods that are not compatible with the current system state and presents the human tester with the remaining external method choices. By examining the guard predicates of each alternative presented, the tester chooses an external method with input parameters to satisfy all guard predicates along some path to the selected action node. Beginning with a system state in which all classes are in their initial states, it is possible to generate an executable sequence of external methods that covers a collection of candidate test paths. As candidate test paths are covered by these actions, they become *test specifications* for their associated *def-use* pairs. The result also records the specific *def-use* pair associated with each new test specification so it is possible to determine the specific pairs solved by each external method of the test sequence.

Using the automobile specification and the results generated above, a tester is able to identify transition nodes derived from external functions (*source transition nodes*) and separate them from other transition nodes (*target transition nodes*). Using these notions of source and target, the following source-to-target pairs and paths are identified. A source is said to *trigger* a target if a path exists in the component flow graph from source to target that does not stop in a wait state. The continuously generated Checkstate() methods in the CruiseUnit and Wheel classes allow many non-intuitive source-to-target paths:

| | | | |
|-------------------------|----------------|---------------------------------------|----------------|
| Source transition nodes | 24 instances | Pairs with one or more generated path | 571 instances |
| Target transition nodes | 160 instances | Generated source-to-target paths | 4598 instances |
| Source-to-target pairs | 3840 instances | | |

This process requires 44 iterations of the algorithm derived from Section 6.3 to test each source-to-target pair. All source to target paths were found by the 27th iteration but it took 17 additional iterations to prove that paths do not exist for the remaining 3029 source-to-target pairs. The CPU processing



time to generate these results, using the same Access database on a Pentium 4 class PC at 2 GHz with 1 GB RAM described in Section 7.1 for candidate test path generation, was 5 hours and 27 minutes.

7.3. Case study

As an experimental evaluation, 108 faults were seeded into the automobile specification; then two sets of tests were run on the seeded faults. The first test set was derived from the Finite State Machine (FSM) data flow methods presented in this paper and the second test set was manually constructed from intuitive use of CruiseControl. To help eliminate bias, all manual steps were independently performed by different researchers. Faults were constructed by modifying the transitions table in the specification database (Appendix A). They were designed by one author (Offutt) by making small syntactic changes, similar to mutants, in every possible location. This process was very mechanical and straightforward. These changes were marked on paper and then implemented by copying the table once for each fault and making one change by another person (Zanon). This resulted in 108 copies of the table.

One author (Gallagher) used the partially automated tool described above to create the executable FSM data flow tests. Another author (Offutt) derived manual test specifications to try to emulate a typical manual testing process. The tests followed a sequence of actions that use every user feature of the CruiseControl in various situations. The fact that the same person designed the faults and also created the manual tests has the potential for introducing a problem with internal validity. However, the faults were designed almost a full year before the manual tests and both activities were very mechanical in nature. These two factors serve to ameliorate the potential for problems with internal validity. The manual test specifications were given to another person (Zanon), who translated them into inputs to the Test Harness, the interface having been independently designed by the third author (Cincotta). The manual test specification sequence is described in user terms as follows:

- (1) start Ignition, change gears and reach highway speed;
- (2) set CruiseControl by turning cruise switch On and pushing SD cruise button;
- (3) use brake to slow speed;
- (4) use RA button to return to previous speed;
- (5) manually increase speed using gas pedal;
- (6) use SD button to set cruise at higher speed;
- (7) use brake to slow speed;
- (8) use SD button to set cruise at new slower speed;
- (9) use ExternalDrag to maintain cruise speed both up and downhill;
- (10) turn cruise switch Off;
- (11) set speed manually with gas pedal to less than SlowCutoff;
- (12) turn cruise switch On;
- (13) test SD button to ensure that nothing happens at slow speed;
- (14) brake to a stop and turn ignition off.

The FSM-derived test cases were created to cover a specific set of candidate test paths (CTPs) and DU-pairs. This information was not determined for the manually created tests. The FSM-derived tests predicted the *system state* for all 12 classes at the end of each execution, whereas intuition and common sense were used to predict the system state for the manually created tests. Both sets of tests were executed by the third author (Cincotta), using the RPM and Test Harness to execute all actions



Table III. Case study results on Automobile example.

| | FSM data flow | Manual |
|------------------|---------------|---------|
| CTPs covered | 1001 | Unknown |
| DU-pairs covered | 954 | Unknown |
| Tests | 145 | 41 |
| Faults | 108 | 108 |
| Faults found | 106 | 24 |
| Percentage found | 98% | 22% |

and to compare the predicted system state with the actual system state after each action. The results are shown in Table III.

Clearly this testing scenario favours the FSM method because it contains more than three times as many tests as the manual method does. In particular, the FSM tests use Accelerate and Decelerate buttons to increase and decrease speed over normal ranges and to exceed the Slow and FastCutoff points, whereas the manual tests do not. However, the manual tests require intensive human effort to decide what new expectations to test, while the FSM approach merely responds to simple answers to questions about which guard predicates to satisfy at each step of the process and cranks out new test actions to cover candidate test paths not previously covered. The actual test sequences and expected system states for both test methods are included in the technical report [41].

A major additional benefit of the testing architecture described above is the ability to generate quickly a reference implementation that responds reasonably to external actions. This capability allowed the tests to find a number of flaws in the original specification that would have been difficult to detect simply by inspecting the guards and actions of the transitions. The granularity of adjustments to the throttle in CruiseUnit transitions t026 and t027 and the granularity of checks in the guard predicates of transitions t023 to t025 have a significant impact on the oscillating effect one gets when using the Resume/Accel button to resume a previously targeted speed or when setting ExternalDrag to sharply different values. Although technically correct according to the specifications, speeds would sometimes wildly oscillate above and below the target speed several times before settling down. Another specification error immediately discovered was forgetting that action queues can hold multiple actions, sometimes resulting in unexpected method invocations in a current state that were put on the queue while the class was in a previous state. Another unexpected result occurs when 'Pauses' are added to the specification, e.g. in CruiseUnit transitions t023, t026, t027, t038, t050 and t065. One sometimes forgets that when one class is pausing, perhaps to wait for delayed effects, other classes may be piling actions on a call queue. Rapid access to a reference implementation helped discover a number of errors of this type in the original specification.

Constructing test cases that predict the expected *system state* at each step greatly improves the ability to find specification errors in transitions that correctly manage the state of their own class, but forget to send proper messages to effect that change in other classes. Early tests found an error in CruiseUnit transition t057 for method UserSwitch(Off) in that it properly changed the state of



CruiseUser, CruiseUnit and Gauges but initially forgot to send a message to the Throttle to go into the Idle state. Very common specification errors are to assign values to variables, declare that the object moves to a new state but forget to check if the state predicate of the new state is satisfied. Test construction found several subtle specification errors of this type. An error found by a test but still existing in the specification is that Transmission transition t005 properly sends a message to the Wheel to change speed when a gear is changed, but forgets that DriveRatio() is handled by the current transition, which returns zero for Neutral, so the Wheel state does not change no matter which gear is set from Neutral. This is an example where a state predicted by a test case (Wheel in DirectDrive) differs from intuition (Wheel in Accel), so a closer look at the specification reveals the error. In all, approximately a dozen errors were found in the specifications. These were found by tests that were created when the test generator was under development and the specifications were corrected. Consequently, the manual tests and the experimental results in Table III were based on the corrected version of the specifications. Of course, this may not be indicative because domain experts did not write the specifications. Still, it is positive anecdotal evidence that the analysis needed to develop tests can help improve specifications.

7.4. Scalability

The collection of tools described in Section 7.2 automates all but two steps in the generation of executable test cases. The database representation of the specification must be created by hand and the final step of generating executable test sequences is only partially automated. A Web-based graphical user interface is currently under development to capture and store specification information into the database. These tool implementations are not typical testing tools that consist of compiled programs. Instead, they consist of the system information represented in a highly structured database schema, together with database queries and other database operations that implement each step in the process. The logical requirements of the algorithm for path generation are implemented as queries and updates to leverage the database system for powerful logical computation and efficient I/O management. Modern commercial databases access the hard disk in the most optimal way, certainly not with every query. They use very effective optimisation strategies that are based on caching, prefetching and other operating system concepts. Optimisation is part of the advantage of using a database—few programmers could manage memory with even a fraction of the efficiency of commercial database products. This allows the methodology to be applied to integration testing in software systems that might otherwise be too large for easy manipulation in the main memory. The literature contains no other methodology that leverage database capabilities in this manner or that can handle data flow testing with graphs this large.

Although it is true that this work thus far has not assured scalability, the authors have experience in both building and using source code-level data flow analysis software and know of *no* other source code-level data flow testing systems, either commercial or experimental, which can handle software specifications that have thousands of DU-pairs. A less obvious advantage was with regards to maintenance. With this technique, it was very easy to modify the algorithm to add additional refinements or to correct small omissions—by modifying a single SQL predicate.

The FSM flow tests found 106 faults, whereas the manual tests only found 24 faults. Not only did the FSM flow tests find more faults, they found more faults per test. This might be a little misleading, however, because the exact number of tests created will vary greatly depending on the decisions made



during test creation. Thus even though not all def-use pairs were covered by the automatic approach, the tests were of a high quality and indicate that in larger systems, not all def-use pairs would need to be covered in order to find many of the critical faults.

The database representation provides a convenient and efficient way to go one step beyond traditional data flow systems and provide definition-clear DU-paths rather than just DU-pairs. Traditional code-level data flow systems provide DU-pairs (as statement numbers) and use *instrumentation* to check whether separately supplied test inputs cause def-clear paths to be executed from the definitions to the uses. 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 [45]. Source code-level data flow analysis has always had problems with the predicates getting too large for memory, which is one reason why data flow testing is seldom, if at all, used in practice. The early papers on data flow discussed data flow paths, but none of the implementations dealt with construction of the paths, which meant that discussions of data flow paths were theoretical.

One reason that traditional code-level data flow programs do not provide complete paths is because the problems of finding feasible paths and determining if the path is def-clear are generally undecidable. When the problems can be solved, the complexity of the control flow, problems with function calls and variable aliasing (where two different names are given to the same memory object) and the size of the data space make the cost of the exponential algorithms prohibitive. This work, however, avoids some of the problems associated with code-level data flow analysis. The 'control flow' on average is much simpler than in code-level control-flow graphs, the data space is much smaller and there is no aliasing. The point of using a database system is that it provides a powerful compute engine for solving predicates, which is one of the most difficult parts of a data flow analyser to implement.

8. CONCLUSIONS AND FUTURE WORK

This paper presents two major and several minor results. The first major result is a method for integration level, inter-class testing for object-oriented programs using data flow techniques. The second is a computational approach to find feasible and infeasible paths. This is a difficult problem where current solutions are not very effective. The approach in this paper could be used in other path-based testing criteria. One minor result is a technique for representing data flow and control flow graphs in a relational database and using the database as a compute engine for deriving DU-pairs and DU-paths to satisfy data flow testing criteria. Software components are modelled as finite state machines and data flows are defined on the finite state machines, yielding DU-paths that are used as a basis for testing. A second minor result is a rapid prototyping machine that allows a quick and convenient way to execute software specifications and a third is an automated test harness to support the test method. A fourth minor result is the introduction of a fairly substantial cruise control example, which can be used by other researchers in their empirical studies. The relevant transitions are in Appendix A, and the full specifications are online in the technical report [41]. These tools and the example were used to obtain empirical results from applying the testing method and comparing it with manual tests.

This paper does not prove the existence of an executable test case for each DU-pair, but by eliminating *def-bound* pairs and by generating a small collection of potentially feasible candidate



test paths for each remaining pair, it substantially increases the likelihood of finding an executable test case. Careful design of the states and guards in a functional specification of transitions and early identification of non-feasible path segments in a component flow graph will also help reduce the number of non-feasible candidate test paths. Future efforts will focus on improving the tool for automatic generation of executable test cases for black-box testing and increasing the automation of generating executable test sequences from candidate test paths.

This paper does not explicitly handle class variables (Java static) or inheritance. However, class variables can be modelled by assuming that they are instance variables in a separate, virtual class, where only one instance of that class is available and where the static methods that access the class variables are methods in the separate class. Inheritance of variables from a superclass is handled by replacing variable references in the subclass with a method invocation of the associated *get* and *set* methods of the superclass. Other aspects of inheritance do not directly impact on this model.

For clarity, the definitions and example in this paper only consider one object per class but extending the method is straightforward. However, aggregation and consideration of multiple class instances are essential for practical application. In static environments with static type hierarchies and static type binding, aggregation and multiple instances are achieved by allowing state variables to be references to some other object. All such reference variables are collected together, creating a new table in the model with a primary key called RefId. Each row of the new table identifies an object whose state and behaviour must be maintained throughout the testing process. Then the associations of Figure 3 are extended to be specified in terms of RefIds instead of just ClassIds. The remainder of the test specification for this situation follows as presented here. The situation is substantially more complex when class hierarchies with dynamic type binding and polymorphism are used. This is an issue for future work.

One interesting question is when to employ the techniques presented in this paper and three possibilities emerge. The most obvious is when software components are integrated. At that time, the FSMs can be generated and relevant transitions can be determined to be those transitions that are included as part of the components in the current integration step. It may also be possible to employ these techniques during maintenance. If a component is to be changed, the *impact* of that change can be estimated in terms of the relevant transitions, and regression testing can proceed on the relevant transitions. This impact could also be limited by applying a *testing firewall* [46], which was found to be very helpful in industrial practice [47] and could readily be incorporated into the FSM model in this paper. Finally, if a new component is to be added to a system, then relevant transitions (and the resulting tests) can be created in terms of the new component. The authors hope to explore these ideas in future work.

With the increasing popularity of object-oriented specification methods, for example, UML [24] and especially state transition specification of classes, for example, UML's state machine package, it becomes possible to align more closely the specification and testing of object-oriented software, with executable test cases generated automatically from the specification. With the addition of database tools, it becomes possible to apply finite state analysis and testing methods to moderate-sized software systems. Follow-on work will focus on further integration of the specification and testing aspects of software development and on the potential application of statistical methods.



APPENDIX A. Relevant feasible mutator transitions for CruiseControl

| Class | TranId | Source | Target | Function | Guard | Action |
|--------------|--------|----------|----------|----------------|---|--|
| AutoSystem | t001 | Initial | Inactive | AutoSystem() | true | ThrottleFloor:=12; ThrottleGovernor:=80; Global BrakeControl:=New BrakeControl(); BrakeActive:=false; Global ClutchUser:=New ClutchUser(); ClutchActive:=false; Global Gauges:=New Gauges(); Danger:=false; Global CruiseUnit:=New CruiseUnit(); BrakeActive:=true; Call CruiseUnit.Cancel(); BrakeActive:=true; Call CruiseUnit.Cancel(); BrakeActive:=false; |
| AutoSystem | t002 | Inactive | Active | BrakeActive(x) | x=true | |
| AutoSystem | t005 | Active | Active | BrakeActive(x) | x=true | |
| AutoSystem | t008 | Active | Inactive | BrakeActive(x) | x=false & ClutchActive=false & Danger=false | |
| BrakeControl | t001 | Initial | Inactive | BrakeControl() | true | Global BrakeUser:=New BrakeUser(); IsActive:=false; PedalPressure:=0; LinePressure:=0; WheelsTurning:=false; IsActive:=true; Call AutoSystem.BrakeActive(true); IsActive:=false; Call AutoSystem.BrakeActive(false); IsActive:=false; Call AutoSystem.BrakeActive(false); IsActive:=false; PedalPressure:=0; pconst:=5; IsActive:=true; Call AutoSystem.BrakeActive(true); Call BrakeControl.IsActive(true); IsActive:=false; Call AutoSystem.BrakeActive(false); Call BrakeControl.IsActive(false); Global CruiseUser:=New CruiseUser(); UserSwitch:=Off; SlowCutoff:=25; FastCutoff:=95; UserMode:=Null; CurrentSpeed:=0; TargetSpeed:=0; TargetThrottle:=0; CurrentSpeed:=Gauges.Speed(); UserSwitch:=On; CurrentSpeed:=Gauges.Speed(); UserMode:=NT; |
| BrakeControl | t002 | Inactive | Braking | IsActive(x) | x=true | |
| BrakeControl | t003 | Braking | Inactive | IsActive(x) | x=false | |
| BrakeControl | t004 | Locked | Inactive | IsActive(x) | x=false | |
| BrakeUser | t001 | Initial | Inactive | BrakeUser() | true | |
| BrakeUser | t002 | Inactive | Braking | IsActive(x) | x=true | |
| BrakeUser | t003 | Braking | Inactive | IsActive(x) | x=false | |
| CruiseUnit | t001 | Initial | Off | CruiseUnit() | true | |
| CruiseUnit | t004 | Off | Off | SetSpeed() | true | |
| CruiseUnit | t009 | Off | Inactive | UserSwitch(x) | x=On | |
| CruiseUnit | t012 | Inactive | Inactive | SetSpeed() | true | |
| CruiseUnit | t014 | Inactive | Inactive | UserMode(x) | x=NT & UserMode=SD & (Gauges.Speed() <= SlowCutoff OR Gauges.Speed() >= FastCutoff) | |



| Class | TranId | Source | Target | Function | Guard | Action |
|------------|--------|----------|----------|---------------|---|---|
| CruiseUnit | t016 | Inactive | Cruise | UserMode(x) | x=NT & UserMode=SD & (SlowCutoff<Gauges.Speed() <FastCutoff) & AutoSystem.BrakeActive()==false & AutoSystem.Danger()==false x<>NT | UserMode:=NT; CurrentSpeed:=Gauges.Speed(); TargetSpeed:=CurrentSpeed; TargetThrottle:=Throttle.Position(); Call Gauges.Cruise(On); Call Throttle.Floor(TargetThrottle); Put CheckState() on Call Queue; UserMode:=x; |
| CruiseUnit | t017 | Inactive | Inactive | UserMode(x) | x<>NT | UserSwitch:=Off; |
| CruiseUnit | t019 | Inactive | Off | UserSwitch(x) | x=Off | Call Gauges.Cruise(Off); Call Throttle.Floor(0); Pause; CurrentSpeed:=Gauges.Speed(); |
| CruiseUnit | t021 | Cruise | Override | Cancel() | true | Put CheckState() on Call Queue; |
| CruiseUnit | t023 | Cruise | Cruise | CheckState() | ABS(TargetSpeed- CurrentSpeed)<0.5 | CurrentSpeed:=Gauges.Speed(); |
| CruiseUnit | t024 | Cruise | Cruise | CheckState() | 0.5<=ABS(TargetSpeed- CurrentSpeed)<1.0 | Put CheckState() on Call Queue; |
| CruiseUnit | t025 | Cruise | Cruise | CheckState() | ABS(TargetSpeed- CurrentSpeed)>=1.0 & Throttle.Position()>Throttle.Floor() | CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue; |
| CruiseUnit | t026 | Cruise | Cruise | CheckState() | CurrentSpeed-TargetSpeed>=1.0 & Throttle.Position()==Throttle.Floor() | Call Throttle.Floor(Throttle.Floor()-0.5); Pause; CurrentSpeed:=Gauges.Speed(); |
| CruiseUnit | t027 | Cruise | Cruise | CheckState() | TargetSpeed-CurrentSpeed>=1.0 & Throttle.Position()==Throttle.Floor() | Put CheckState() on Call Queue; Call Throttle.Floor(Throttle.Floor()+0.5); Pause; CurrentSpeed:=Gauges.Speed(); |
| CruiseUnit | t028 | Cruise | Cruise | SetSpeed() | true | Put CheckState() on Call Queue; |
| CruiseUnit | t030 | Cruise | Decel | UserMode(x) | x=SD | TargetSpeed:=TargetSpeed-1; UserMode:=SD; |
| CruiseUnit | t031 | Cruise | Accel | UserMode(x) | x=RA | Put CheckState() on Call Queue; TargetSpeed:=TargetSpeed+1; UserMode:=RA; |
| CruiseUnit | t034 | Cruise | Off | UserSwitch(x) | x=Off | Put CheckState() on Call Queue; |
| CruiseUnit | t035 | Decel | Override | Cancel() | true | Call Gauges.Cruise(Off); UserSwitch:=Off; UserMode:=Null; Call Throttle.Floor(0); Call Gauges.Cruise(Off); UserMode:=Null; Call Throttle.Floor(0); |
| CruiseUnit | t038 | Decel | Decel | CheckState() | CurrentSpeed>SlowCutoff | Call Throttle.Floor(Throttle.Position()-0.5); Pause; CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue; |



| Class | TranId | Source | Target | Function | Guard | Action |
|------------|--------|----------|----------|---------------|---|---|
| CruiseUnit | t039 | Decel | Override | CheckState() | CurrentSpeed <= SlowCutoff | Call Gauges.Cruise(Off); UserMode:=Null; Call Throttle.Floor(0); CurrentSpeed:=Gauges.Speed(); |
| CruiseUnit | t040 | Decel | Decel | SetSpeed() | true | UserMode:=NT; TargetSpeed:=Gauges.Speed(); |
| CruiseUnit | t043 | Decel | Cruise | UserMode(x) | x=NT | TargetThrottle:=Throttle.Position(); CurrentSpeed:=TargetSpeed; |
| CruiseUnit | t045 | Decel | Off | UserSwitch(x) | x=Off | Put CheckState() on Call Queue; |
| CruiseUnit | t047 | Accel | Override | Cancel() | true | Call Gauges.Cruise(Off); UserSwitch:=Off; UserMode:=Null; Call Throttle.Floor(0); |
| CruiseUnit | t050 | Accel | Accel | CheckState() | CurrentSpeed < FastCutoff | Call Gauges.Cruise(Off); UserMode:=Null; Call Throttle.Floor(0); Call Throttle.Floor(Throttle.Position()+0.5); Pause; |
| CruiseUnit | t051 | Accel | Override | CheckState() | CurrentSpeed >= FastCutoff | CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue; |
| CruiseUnit | t052 | Accel | Accel | SetSpeed() | true | Call Gauges.Cruise(Off); UserMode:=Null; Call Throttle.Floor(0); |
| CruiseUnit | t055 | Accel | Cruise | UserMode(x) | x=NT | CurrentSpeed:=Gauges.Speed(); UserMode:=NT; TargetSpeed:=Gauges.Speed(); TargetThrottle:=Throttle.Position(); |
| CruiseUnit | t057 | Accel | Off | UserSwitch(x) | x=Off | CurrentSpeed:=TargetSpeed; Put CheckState() on Call Queue; |
| CruiseUnit | t061 | Override | Override | SetSpeed() | true | Call Gauges.Cruise(Off); UserSwitch:=Off; |
| CruiseUnit | t063 | Override | Override | UserMode(x) | x < > NT OR Gauges.Speed() <= SlowCutoff OR Gauges.Speed() >= FastCutoff | UserMode:=Null; Call Throttle.Floor(0); CurrentSpeed:=Gauges.Speed(); UserMode:=x; |
| CruiseUnit | t064 | Override | Cruise | UserMode(x) | x=NT & UserMode=SD & (SlowCutoff < Gauges.Speed() < FastCutoff) & AutoSystem.BrakeActive()=false & AutoSystem.Danger()=false | CurrentSpeed:=Gauges.Speed(); TargetSpeed:=CurrentSpeed; TargetThrottle:=Throttle.Position(); Call Gauges.Cruise(On); Call Throttle.Floor(TargetThrottle); UserMode:=NT; Put CheckState() on Call Queue; |



| Class | TranId | Source | Target | Function | Guard | Action |
|------------|--------|----------|----------|------------------|---|--|
| CruiseUnit | t065 | Override | Cruise | UserMode(x) | x=NT & UserMode=RA & (SlowCutoff<Gauges.Speed()< FastCutoff) & AutoSystem.BrakeActive()==false & AutoSystem.Danger()==false | Call Throttle.Floor(TargetThrottle); Call Gauges.Cruise(On); UserMode:=NT; Pause; CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue; |
| CruiseUnit | t066 | Override | Override | UserMode(x) | x=NT & UserMode=Null | UserMode:=NT; |
| CruiseUnit | t069 | Override | Off | UserSwitch(x) | x=Off | UserSwitch:=Off; UserMode:=Null; |
| CruiseUser | t001 | Initial | Off | CruiseUser() | true | Switch:=Off; Mode:=NT; |
| CruiseUser | t002 | Off | Neutral | Switch(x) | x=On | Switch:=On; Call CruiseUnit.UserSwitch(On); |
| CruiseUser | t003 | Neutral | Off | Switch(x) | x=Off | Switch:=Off; Call CruiseUnit.UserSwitch(Off); |
| CruiseUser | t004 | Neutral | Accel | Mode(x) | x=RA | Mode:=RA; Call CruiseUnit.UserMode(RA); |
| CruiseUser | t005 | Accel | Neutral | Mode(x) | x=NT | Mode:=NT; Call CruiseUnit.UserMode(NT); |
| CruiseUser | t006 | Decel | Neutral | Mode(x) | x=NT | Mode:=NT; Call CruiseUnit.UserMode(NT); |
| CruiseUser | t007 | Neutral | Decel | Mode(x) | x=SD | Mode:=SD; Call CruiseUnit.UserMode(SD); |
| CruiseUser | t008 | Accel | Off | Switch(x) | x=Off | Switch:=Off; Mode:=NT; |
| CruiseUser | t009 | Decel | Off | Switch(x) | x=Off | Call CruiseUnit.UserSwitch(Off); |
| CruiseUser | t010 | Neutral | Neutral | Cancel() | true | Switch:=Off; Mode:=NT; |
| CruiseUser | t016 | Accel | Accel | Cancel() | true | Call CruiseUnit.UserSwitch(Off); |
| CruiseUser | t019 | Decel | Decel | Cancel() | true | Call CruiseUnit.Cancel(); |
| Engine | t001 | Initial | Normal | Engine() | true | Call CruiseUnit.Cancel(); |
| Engine | t003 | Normal | Normal | GasFlow(x) | true | Rpm:=0; GasFlow:=0; ExternalDrag:=1; |
| Engine | t005 | Normal | Normal | ExternalDrag(x) | true | WaterTMin:=0; OilPMin:=0; |
| Engine | t001 | Initial | Active | GasUser() | true | GasFlow:=x; Rpm:=(2-ExternalDrag)*GasFlow*630; |
| Engine | t002 | Active | Active | PedalPosition(x) | x > 0 & x < > PedalPosition | Call Gauges.Tach(Rpm); Call Wheel.AxelRpm(Rpm*Transmission.DriveRatio()); ExternalDrag:=x; |
| GasUser | t001 | Initial | Active | GasUser() | true | Rpm:=(2-ExternalDrag)*GasFlow*630; |
| GasUser | t002 | Active | Active | PedalPosition(x) | x > 0 & x < > PedalPosition | Call Gauges.Tach(Rpm); Call Wheel.AxelRpm(Rpm*Transmission.DriveRatio()); PedalPosition:=0; |
| GasUser | t002 | Active | Active | PedalPosition(x) | x > 0 & x < > PedalPosition | PedalPosition:=x; |
| GasUser | t002 | Active | Active | PedalPosition(x) | x > 0 & x < > PedalPosition | Call Throttle.GasPedal(PedalPosition); |



| Class | TranId | Source | Target | Function | Guard | Action |
|----------|--------|-----------|-----------|----------------|---------------------|--|
| Gauges | t000 | Initial | Normal | Gauges() | true | Speed:=0; Cruise:=Off; Tach:=0; OilPressure:=0; OilLight:=Off; Odometer:=Null; TripMeter:=Null; WaterTemp:=0; AbsLight:=Off; Battery:=Off; SeatBelt:=Off; HandBrake:=Null; LowGas:=Off; Tach:=x; |
| Gauges | t006 | Normal | Normal | Tach(x) | true | Speed:=x; |
| Gauges | t008 | Normal | Normal | Speed(x) | x<180 | Speed:=Min(x,250); Call AutoSystem.Danger(true); |
| Gauges | t009 | Normal | Danger | Speed(x) | x>=180 | Cruise:=x; |
| Gauges | t017 | Normal | Normal | Cruise(x) | true | Speed:=Min(x,250); |
| Gauges | t034 | Danger | Danger | Speed(x) | x>=180 | Key:=On; EngineOn:=false; |
| Ignition | t000 | Initial | On | Ignition() | true | Global AutoSystem:=New AutoSystem(); |
| Ignition | t001 | On | Initial | Key(x) | x=Off | Key:=Off; EngineOn:=false; Destroy Throttle; |
| Ignition | t003 | On | On | StartEngine() | EngineOn=false | Destroy Engine; Destroy AutoSystem; Destroy Self; Global Transmission:=New Transmission(); Global Engine:=New Engine(); |
| Throttle | t001 | Initial | Idle | Throttle(x, y) | 0<x & x<y & y<100 | Global GasUser:=New GasUser(); Global Throttle:=New Throttle(AutoSystem.ThrottleFloor()); AutoSystem.ThrottleGovernor(); EngineOn:=true; fconst:=x; gconst:=y; Position:=fconst; |
| Throttle | t002 | Idle | Manual | GasPedal(x) | x>fconst | Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(fconst); |
| Throttle | t003 | Manual | Idle | GasPedal(x) | x<=fconst | GasPedal:=x; Position:=Min(GasPedal, gconst); Call Engine.GasFlow(Convert(Position)); GasPedal:=x; Position:=fconst; |
| Throttle | t004 | Idle | Automatic | Floor(x) | x>fconst | Call Engine.GasFlow(Convert(fconst)); Call GasUser.PedalPosition(fconst); Floor:=Min(x, gconst); Position:=Floor; |
| Throttle | t005 | Automatic | Idle | Floor(x) | x<=fconst | Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(Position); Floor:=fconst; Position:=fconst; |
| Throttle | t006 | Manual | Automatic | GasPedal(x) | x>fconst & x<=Floor | Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(fconst); GasPedal:=x; Position:=Floor; |
| Throttle | t007 | Manual | Automatic | Floor(x) | x>=Position | Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(Floor); Floor:=Min(x, gconst); Position:=Floor; |
| Throttle | t007 | Manual | Automatic | Floor(x) | x>=Position | Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(Floor); |



| Class | TranId | Source | Target | Function | Guard | Action |
|--------------|--------|-------------|-------------|----------------|---|---|
| Throttle | t008 | Automatic | Manual | GasPedal(x) | $x > fconst$ & $x > Floor$ & $x <= gconst$ $x > fconst$ | GasPedal:=x; Position:=x; Call Engine.GasFlow(Convert(Position)); Floor:=Min(x, gconst); Position:=Floor; Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(Position); GasPedal:=x; |
| Throttle | t009 | Automatic | Automatic | Floor(x) | | Floor:=fconst; |
| Throttle | t024 | Idle | Idle | GasPedal(x) | $x <= fconst$ | |
| Throttle | t025 | Idle | Idle | Floor(x) | $x <= fconst$ | |
| Throttle | t026 | Manual | Manual | GasPedal(x) | $x > fconst$ & $x <= gconst$ & $x > Floor$ | GasPedal:=x; Position:=x; Call Engine.GasFlow(Convert(Position)); Floor:=Max(fconst, x); |
| Throttle | t028 | Manual | Manual | Floor(x) | $x < Position$ | |
| Throttle | t030 | Automatic | Automatic | GasPedal(x) | $x > fconst$ & $x <= Floor$ | GasPedal:=x; |
| Transmission | t001 | Initial | Neutral | Transmission() | true | Gear:=N; Ratio_R:=1.846; Ratio_1:=2.563; Ratio_2:=1.552; Ratio_3:=1.022; Ratio_4:=0.653; Ratio_5:=0.471; Ratio_Diff:=4.429; Global Wheel:=New Wheel(); Gear:=x; |
| Transmission | t005 | Neutral | Forward | Gear(x) | $x=1$ OR $x=2$ OR $x=3$ OR $x=4$ OR $x=5$ | Call Wheel.AxelRpm(Gauges.Tach()*DriveRatio()); Gear:=N; Call Wheel.AxelRpm(0); |
| Transmission | t007 | Forward | Neutral | Gear(x) | $x=N$ | |
| Transmission | t008 | Forward | Forward | Gear(x) | $x=1$ OR $x=2$ OR $x=3$ OR $x=4$ OR $x=5$ | Gear:=x; |
| Wheel | t001 | Initial | DirectDrive | Wheel() | true | Call Wheel.AxelRpm(Gauges.Tach()*DriveRatio()); AxelRpm:=0; WheelRpm:=0; |
| Wheel | t002 | DirectDrive | DirectDrive | AxelRpm(x) | $ABS(x - WheelRpm) <= 2$ | WheelDiam:=0.00056; AxelRpm:=x; WheelRpm:=x; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); AxelRpm:=x; WheelRpm:=WheelRpm-1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); |
| Wheel | t003 | DirectDrive | Decel | AxelRpm(x) | $x+2 < WheelRpm$ | Put CheckState() on Call Queue; AxelRpm:=x; WheelRpm:=WheelRpm+1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); |
| Wheel | t004 | DirectDrive | Accel | AxelRpm(x) | $x-2 > WheelRpm$ | Put CheckState() on Call Queue; AxelRpm:=x; WheelRpm:=WheelRpm-1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); |
| Wheel | t005 | Decel | Decel | AxelRpm(x) | $x+2 < WheelRpm$ | Put CheckState() on Call Queue; |
| Wheel | t006 | Decel | Accel | AxelRpm(x) | $x-2 > WheelRpm$ | AxelRpm:=x; WheelRpm:=WheelRpm+1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call Queue; |
| Wheel | t007 | Decel | DirectDrive | AxelRpm(x) | $ABS(x - WheelRpm) <= 2$ | AxelRpm:=x; WheelRpm:=x; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); |



| Class | TranId | Source | Target | Function | Guard | Action |
|-------|--------|--------|-------------|--------------|---------------------------------------|---|
| Wheel | t008 | Accel | Decel | AxelRpm(x) | $x+2 < \text{WheelRpm}$ | AxelRpm:=x; WheelRpm:=WheelRpm-1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call Queue; |
| Wheel | t009 | Accel | DirectDrive | AxelRpm(x) | $\text{ABS}(x-\text{WheelRpm}) <= 2$ | AxelRpm:=x; WheelRpm:=x; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); |
| Wheel | t010 | Accel | Accel | AxelRpm(x) | $x-2 > \text{WheelRpm}$ | AxelRpm:=x; WheelRpm:=WheelRpm+1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call Queue; |
| Wheel | t012 | Decel | Decel | CheckState() | $\text{AxelRpm}+2 < \text{WheelRpm}$ | WheelRpm:=WheelRpm-1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call Queue; |
| Wheel | t013 | Decel | DirectDrive | CheckState() | $\text{AxelRpm}+2 >= \text{WheelRpm}$ | WheelRpm:=AxelRpm; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); |
| Wheel | t014 | Accel | Accel | CheckState() | $\text{AxelRpm}-2 > \text{WheelRpm}$ | WheelRpm:=WheelRpm+1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call Queue; |
| Wheel | t015 | Accel | DirectDrive | CheckState() | $\text{AxelRpm}-2 <= \text{WheelRpm}$ | WheelRpm:=AxelRpm; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); |



ACKNOWLEDGEMENTS

The second author gratefully acknowledges the support by the U.S. National Science Foundation under grant CCR-98-04111 during this research. The second author is also employed as a part-time faculty researcher at NIST. It is a pleasure to acknowledge Julie Zanon for the tool that generates all of the derived associations described in Figure 4 as well as initial tool development for making this approach available on the Internet as a Web application and also Roger Alexander, Paul E. Black and the anonymous reviewers for a number of helpful suggestions.

REFERENCES

1. Chen M-H, Kao M-H. Testing object-oriented programs—an integrated approach. *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99)*, Boca Raton, FL, November 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999; 73–83.
2. Chow TS. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering* 1978; **4**(3):178–187.
3. Linn RJ, Uyar MÜ. *Conformance Testing Methodologies and Architectures for OSI Protocols*. IEEE Computer Society Press: Los Alamitos, CA, 1994.
4. Turner D, Robson DJ. The state-based testing of object-oriented programs. *Proceedings of the International Conference on Software Maintenance*, Montreal, Quebec, Canada, September 1993. IEEE Computer Society Press: Los Alamitos, CA, 1993; 302–310.
5. Booch G. *Object Oriented Design with Applications*. Benjamin Cummings: Menlo Park, CA, 1991.
6. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. *Object Oriented Modeling and Design*. Prentice-Hall: Englewood Cliffs, NJ, 1991.
7. Hong HS, Kwon YR, Cha SD. Testing of object-oriented programs based on finite state machines. *Proceedings of the 2nd Asia-Pacific Software Engineering Conference*, Brisbane, Australia, December 1995. IEEE Computer Society Press: Los Alamitos, CA, 1995; 234–241.
8. Frankl PG, Weyuker EJ. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering* 1988; **14**(10):1483–1498.
9. Offutt J, Liu S, Abdurazik A, Ammann P. Generating test data from state-based specifications. *Software Testing, Verification and Reliability* 2003; **13**(1):25–53.
10. Harrold MJ, Soffa ML. Selecting and using data for integration testing. *IEEE Software* 1991; **8**(2):58–65.
11. Laski J, Korel B. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering* 1983; **9**(3):347–354.
12. Harrold MJ, Rothermel G. Performing data flow testing on classes. *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, New Orleans, LA, December 1994. ACM Press: New York, 1994; 154–163.
13. Parrish S, Borie RB, Cordes DW. Automated flow graph-based testing of object-oriented software modules. *Journal of Systems and Software* 1993; **23**(11):95–109.
14. Doong RK, Frankl PG. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology* 1994; **3**(2):101–130.
15. Chen HY, Tse TH, Chan FT, Chen TY. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology* 1998; **7**(3):250–295.
16. Chen HY, Tse TH, Chen TY. TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology* 2001; **10**(4):56–109.
17. Jin Z, Offutt J. Coupling-based criteria for integration testing. *Software Testing, Verification and Reliability* 1998; **8**(3):133–154.
18. Alexander R, Offutt J. Analysis techniques for testing polymorphic relationships. *Proceedings of the 30th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA'99)*, Santa Barbara, CA, August 1999. IEEE Computer Society Press, 1999; 104–114.
19. Alexander R, Offutt J. Criteria for testing polymorphic relationships. *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, San Jose, CA, October 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000; 15–23.
20. Kung D, Suchak N, Gao J, Hsia P, Toyoshima Y, Chen C. On object state testing. *Proceedings of the 18th Computer Software and Applications Conference (COMPSAC'94)*, Taipei, Taiwan, November 1994. IEEE Computer Society Press: Los Alamitos, CA, 1994; 222–227.



21. Kung D, Liu CH, Hsia P. An object-oriented Web test model for testing Web applications. *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC 2000)*, Taipei, Taiwan, October 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000; 537–542.
22. Kung D, Gao J, Hsia P, Toyoshima Y, Chen C. A test strategy for object-oriented programs. *Proceedings of the 19th Computer Software and Applications Conference (COMPSAC'95)*, Dallas, TX, August 1995. IEEE Computer Society Press: Los Alamitos, CA, 1995; 239–244.
23. Liu CH, Kung D, Hsia P, Hsu CT. Structural testing for Web applications. *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE 2000)*, San Jose, CA, October 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000; 84–96.
24. Unified Modeling Language (UML) Version 1.4.2, International Standard ISO/IEC 15959:2005, published 13 April 2005. Related documentation available at: <http://www.omg.org/technology/documents/formal/uml.htm> [December 2005].
25. Codd EF. A relational model of data for large shared data banks. *Communications of the ACM* 1970; **13**(6):377–387, reprinted in January 1983, **26**(1):64–69.
26. Date J. *An Introduction to Database Systems* (6th edn). Addison-Wesley: Reading, MA, 1995.
27. Melton J, Simon A. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann: San Mateo, CA, 1993.
28. Gosling J, Joy B, Steele G, Bracha G. *The Java Language Specification* (3rd edn). Sun Microsystems and Addison-Wesley Professional, 2005. Related documentation available at: <http://java.sun.com/> [December 2005].
29. Ntafos SC. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering* 1988; **14**(6):868–874.
30. Gallagher L. Conformance testing of object-oriented components specified by state/transition classes. *Technical Report NISTIR 6592*, National Institute of Standards and Technology, May 1999. Available at: <http://www.itl.nist.gov/div897/ctg/conformance/obj-comp-testing.pdf>.
31. Clarke LA, Podgurski A, Richardson DJ, Zeil SJ. A comparison of data flow path selection criteria. *Proceedings of the 8th International Conference on Software Engineering*, London, U.K., August 1985. IEEE Computer Society Press: Los Alamitos, CA, 1985; 244–251.
32. Frankl PG, Weiss SN. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering* 1993; **19**(8):774–787.
33. Frankl PG, Weiss SN, Hu C. All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software* 1997; **38**(3):235–253.
34. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994. IEEE Computer Society Press: Los Alamitos, CA, 1994; 191–200.
35. Mathur P, Wong WE. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability* 1994; **4**(1):9–31.
36. Offutt J, Pan J, Tewary K, Zhang T. An experimental evaluation of data flow and mutation testing. *Software—Practice and Experience* 1996; **26**(2):165–176.
37. Goldberg A, Wang TC, Zimmerman D. Applications of feasible path analysis to program testing. *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA'94)*, Seattle, WA, August 1994. ACM Press: New York, 1994; 80–94.
38. Hedley D, Hennell MA. The causes and effects of infeasible paths in computer programs. *Proceedings of the 8th International Conference on Software Engineering*, London, U.K., August 1985. IEEE Computer Society Press: Los Alamitos, CA, 1985; 259–266.
39. Jasper R, Brennan M, Williamson K, Currier B, Zimmerman D. Test data generation and feasible path analysis. *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA'94)*, Seattle, WA, August 1994. ACM Press: New York, 1994; 95–107.
40. Offutt J, Pan J. Detecting equivalent mutants and the feasible path problem. *Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS'96)*, Gaithersburg, MD, June 1996. IEEE Computer Society Press: Los Alamitos, CA, 1996; 224–236.
41. Gallagher L, Offutt AJ. Integration testing of object-oriented components using FSMS: Theory and experimental details. *Technical Report ISE-TR-04-04*, Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2004. Available at: <http://www.ise.gmu.edu/techrep/>.
42. Abdurazik A, Ammann P, Ding W, Offutt J. Evaluation of three specification-based testing criteria. *Proceedings of the 6th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'00)*, Tokyo, Japan, September 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000; 179–187.
43. Atlee JM. Native model-checking of SCR requirements. *Proceedings of the 4th International SCR Workshop*, November 1994.
44. Gomma H. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley: Reading, MA, 2000.



-
45. Offutt J, Jin Z, Pan J. The dynamic domain reduction approach to test data generation. *Software—Practice and Experience* 1999; **29**(2):167–193.
 46. Leung H, White L. A study of integration testing and software regression at the integration level. *Proceedings of the IEEE International Conference on Software Maintenance*, San Diego, CA, November 1990. IEEE Computer Society Press: Los Alamitos, CA, 1990; 290–301.
 47. White L, Robinson B. Industrial real-time regression testing and analysis using firewalls. *Proceedings of the 20th IEEE International Conference on Software Maintenance*, Chicago, IL, September 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 18–27.