Integration Testing of Object-oriented Components Using FSMS: Theory and Experimental Details

GMU Technical Report ISE-TR-04-04, July 2004

Leonard Gallagher
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg MD 20899-8970 USA
lgallagher@nist.gov

Jeff Offutt†
Information and Software Engineering
George Mason University
Fairfax VA 22032-4400 USA
ofut@ise.gmu.edu

Abstract

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 external objects that receive the messages. This research extends an existing single-class testing technique to integration testing. The previous method models the behavior of a single class as a finite state machine, transforms that 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 and 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. A second major result is the introduction of a novel approach to performing data flow analysis. 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 modeling and processing techniques, and algorithms for analysis and traversal of directed graphs. A proof-of-concept implementation is used to illustrate how the approach works on an extended example.

Keywords: Software integration testing, conformance testing, data flow testing, data modeling, finite state machines, object-oriented

† Partially supported by the U.S. National Science Foundation under grant 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. The goal of this paper is to provide conformance-testing techniques for the integration of individual components within a complete software system.

Each component is assumed to be object-oriented, that is, it is implemented with objects that have state and behavior. 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. An *object* is an instance of a class. Each object has state and behavior, where state is determined by the values of *variables* of the class, and behavior is determined by *methods* (i.e. functions or procedures) defined in the class that operate on one or more objects to read and modify their state variables. The *behavior* of an object when acted upon by a method can be modeled as the effect the method has on the variables of that object 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 programming language-independent, but 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 behavior of an object can be captured as a set of transition rules for each method. Thus finite state machines are often used for class specification in object-oriented analysis and design [9, 11, 29, 39]. The behavior of a component is specified by the behavior 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 can be 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 [6] and Rumbaugh et al. [38] to characterize an object as something that has state, behavior, and identity, and to characterize an object's class 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. [22] developed a *class-level flow graph* to represent control and data flow within a **single class**. Since testing a single instance of a single class greatly limits the usefulness of the approach, 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. Test criteria are defined on this graph, and test specifications are generated to satisfy the criteria.

This research began as an attempt to determine a *sample space* for data flow analysis in object-oriented software so that software testing by statistical methods [5] could be applied. This paper provides a process that results in a set of test specifications that could be used as a statistical sample space, but specific statistical methods have not been considered. 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 labeled 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. An executable test suite to determine whether a software product conforms to its specification may then be constructed from the test specifications.

This paper introduces a novel approach to storing and computing data flow analysis information. Instead of the traditional storage within program data structures, all information is stored in a relational database. Instead of complicated algorithms, straightforward queries are used to record and process data flow information. This technique enhances scalability, because a lot of information can be stored in the database in an efficient manner, and it makes the computation of data flow information relatively simple. The database schemas and SQL queries are based on rather complex mathematical expressions, but the mathematics is not necessary to understand or use the representation technique.

Moreover, this technique allows additional information to be provided to the tester. In traditional data flow testing [15], the tester is provided with pairs of definitions and uses of variables (DU-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 allows more information to be provided; 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 to 50 LOC) function is impractical, and this has been a major factor in the lack of widespread adoption of the technique. Using the database allows these potentially large predicates to be stored off-line, and all the I/O is handled invisibly by the database.

The attributes and constraints of classes and methods are modeled 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 through 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 an extended example of this technique applied to an extended version of the common automobile cruise control system that includes the engine, brakes, gas, throttle, displays and clutch.

2 Background

Much of testing has been based on data and control flow through programs [15, 35]. In such testing, graphs are defined in which nodes are formed from *basic blocks*, which are 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. In a data flow graph, edges are formed from *definitions* (*def*s) and *uses* of the same memory locations. These memory are usually referenced by one variable, but can also be referenced by multiple variable names through aliasing. A def of a location *x* is a node in which *x* is given a value, and a use is a node in which the value is accessed. An edge is formed from nodes in which a location is defined to nodes in which the location is used **and** there is a def-clear control path 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 [15, 20] 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; called *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 behavior of classes, thus there may be no direct relationship to the implementation. This makes the technique more suitable for conformance testing.

Harrold and Rothermel describe an approach that applies traditional data-flow analysis to classes [21]. 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 we introduce 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. [22], Parrish et al. [37], Turner and Robson [39], Doong and Frankl [14], and Chen et al. [7]. Intra-class testing strategies focus on one class at a time, so does not find problems that exist in the interfaces between classes, or in inheritance and polymorphism among classes. In their TACCLE methodology [8] 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 [25], who defined *coupling-based testing*, which requires tests to be found that cover control and data couplings between methods in different classes. Alexander and Offutt [2, 3] have extended these ideas to cover couplings formed from inheritance and polymorphism. Chen and Kao [9] 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 testing should cover all possible type bindings in the presence of polymorphism. Kung et al. [27] 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 by finding a test order to minimize the effort to construct test stubs. It is difficult to apply this technique to conformance testing since there is no functional specification of class semantics.

Some related work has been done on the subject of testing web software. Kung et al. [27, 28, 30] 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. to the inter-class level, thus providing full integration level testing. This paper does not explicitly deal with inheritance and polymorphism, which are left to future research.

Following Rumbaugh et al. [38], the behavior of classes is specified as finite state machines 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 have 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 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 source state and the guard; the post-condition is the predicate of the target state. Note that the post-condition derived from the transitions is not the strongest post-condition. The post-condition of a transition is the state predicate of the target state. 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 single-class state machine (CSM) is defined in Definition 2.1. This definition is exactly the same as Hong's [22], except for the addition of the parameter set P, which is needed for multiple classes. The CSM is extended to a *combined* CSM in Section 2.2.

Definition 2.1 (CSM): A *class state machine* of a class *C* is a tuple $M = (V, F, P, S, T)$, where

- *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 mutator member functions.
- *S* is a finite set of states, $S = \{s \mid s = (pred)\}$ where *pred* is a predicate on the instance variables in *V*.
- *T* is a finite set of transitions, $T = \{t \mid t = (source, target, fn, guard, action)\}$ 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*(S) and *Stop*(). Each state is associated with values of the instance variables as follows:

OFF: speed = 0 ∧ KeyOn = false ON: 0 ≤ speed ≤ 110 ∧ KeyOn = true

In the **Engine** example, the transition from OFF to ON is triggered by the 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 = S*). The sets of variables, member functions, states, and transitions are defined as follows:

S = {S_0, S_f, ON, OFF}
V = {**int** speed, **boolean** KeyOn}
F = {Engine (), ~Engine (), setKeyOn **(boolean** in), Start (**int** S), Stop (), setSpeed (**int** S), int getSpeed ()}
P = {setKeyOn:in, Start:S, setSpeed:S }
T = {$t_i \mid 1 \le i \le 9$}
t_1 = (S_0, OFF, Engine(), true, {speed = 0, KeyOn = false})
t_2 = (OFF, OFF, getSpeed(), true, {return speed})
t_3 = (OFF, OFF, setKeyOn(in), true, {KeyOn = in})
t_4 = (OFF, ON, Start(S), KeyOn==true ∧ 0 ≤ S ≤ 110, {speed = S })
t_5 = (OFF, S_f, ~Engine(), true, { })
t_6 = (ON, ON, getSpeed(), true, {return speed })
t_7 = (ON, ON, setSpeed(S), 0 ≤ S ≤ 110, {speed = S})
t_8 = (ON, OFF, Stop(), true, {speed = 0})
t_9 = (ON, S_f, ~Engine(), true, { })

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

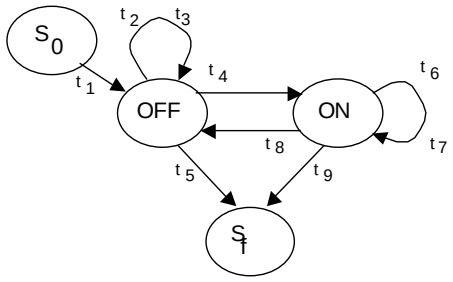

Figure 1: Class State Transition Machine for Engine

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

2.2 Multi-class example - Automobile

Inter-class integration testing addresses interactions among multiple components, so this example modifies the Engine class from Section 2.1 and integrates it with other components. Each received message is an *event* on the recipient object. Components can function as independent processes, possibly running at remote locations 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 seven core components: Acceleration, Brakes, Clutch, 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 1.

Component	Classes
Acceleration	GasUser, Throttle
Brakes	BrakeUser, BrakeControl
Clutch	ClutchUser
CruiseControl	CruiseUser, CruiseUnit
Engine	Engine
InstrumentPanel	Gauges
SystemControl	AutoSystem

Table 1: Classes in Cruise Control Components

The GasUser, BrakeUser, ClutchUser, 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 Cruise Control. If

the user holds the RA or SD button down, the user mode is that button, and 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 Drag variable. The externally invokable methods are:

BrakeUser.IsActive (x)	$x \in$ {*true, false*}
BrakeUser.PedalPressure (x)	$0 \leq x \leq 99$
ClutchUser.PedalPosition (x)	$0 \leq x \leq 99$
CruiseUser.Cancel ()	
CruiseUser.Mode (x)	$x \in$ {NT, SD, RA}
CruiseUser.Switch (x)	$x \in$ {*On, Off*}
Engine.ExternalDrag (x)	$-9 \leq x \leq 9$
GasUser.PedalPosition (x)	$0 \leq x \leq 99$

All other methods are internal methods that can only be invoked by internal actions. 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. Alternatively, the second button could just be ignored when the first is engaged.

Definition 2.1 is extended to define a *combined* Class State Machine for multiple classes by adding a set of classes and parameters that are inputs to mutator functions. The Automobile example is represented as a tuple (C, V, F, P, S, T) where C is a set of 10 classes, V is a set of 46 variables consisting of the union of all state variables from each class, F is a set of 97 rows consisting of the union of all member functions from each class, P is a set of 41 parameters representing inputs of mutator functions, S is a set of 76 states consisting of the union of all states from each class, and T is a set of 143 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 an expanded Class State Machine is in Appendix I.

Figure 2 presents a directed graph that shows an abstraction of the relevant communication paths among the classes. Since the Gauges class is passive, the arrows between CruiseUnit and Gauges indicate 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 is used in the speed calculation.

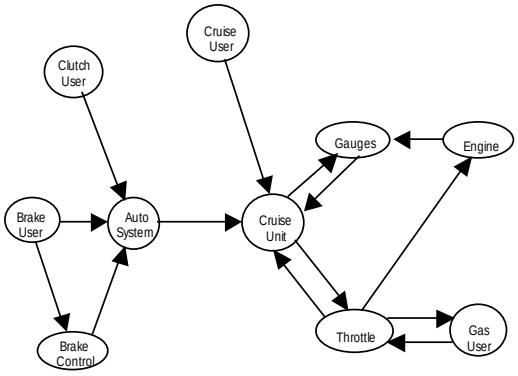

Figure 2: Class-to-Class Data Flow

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 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 behavioral specifications of the various components. To begin, a state/transition specification must exist for each class, with behavior specified by a Class State Machine as in Definition 2.1. The CSM could have been produced during design, perhaps as UML diagrams, or may be produced by the tester. The CSMs for each class are combined and represent the resulting sets according to the 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 analyzed to identify all calls of actor or mutator functions from other classes and the passing of state variables as parameters of mutator functions.

Once the software system is represented in the DB schema, the next step is to focus on individual components and how they integrate with other components. In the Automobile example, the focus is on the CruiseControl component and its relevant interactions with other classes in the Automobile system. Since CruiseControl activity is canceled whenever a brake or clutch is active, or whenever an emergency state is entered, this example safely ignores the complex BrakeControl behavior dealing with anti-lock brakes and all of the AutoSystem behavior dealing with such items as air bags. Section 4 defines *relevant transitions* for a given component, thereby focusing only on the transitions in the entire software system that are both feasible and relevant to the component being tested.

The next step is to model all potential finite state transitions as a directed 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 modeled as directed edges between these nodes. Following the example of Hong et al. [22], the process starts with directed edges from a source state node to the guard node or transition node of each transition, from all guard nodes to their corresponding transition node, and from all transition nodes to their target state nodes. 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 results in a *component flow graph* (formally defined in Section 5).

The next step is to choose a testing criterion and to adapt it to the information stored in the DB schema and the component flow graph. 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 or guard-to-transition edge. The procedure looks for *candidate test paths* through the *component flow graph* for each def-use pair. Much of the remaining effort 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. It is just as valuable to prove that a feasible path cannot exist as it is to find one. A prototype implementation has been developed that constructs a small collection of *candidate test paths* for each *def-use* pair or proves that the pair is *def-bound* so that no such path can exist. Much of the effort in Section 6 is to ensure that the collection of candidate tests paths for each pair is as small as possible. 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.

This implementation is not a typical testing tool that consists of compiled programs. Instead, it consists primarily 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 in order to leverage the database system for powerful logical computation and I/O management. This allows the methodology to be applied to integration testing in software systems that might otherwise be too large for easy manipulation in main memory. We know of no other methodology that can leverage database capabilities in this manner or that can handle data flow testing with graphs this large.

Section 7 demonstrates this methodology on the CruiseControl component of the Automobile example to analyze 3433 *def-use* pairs, constructing candidate test paths for 1933 pairs and proving that the remaining 1500 pairs are *def-bound* with no possible *def-free* path. There is no guarantee that the candidate test paths will yield test cases, but they serve to substantially reduce the search space, making it much more likely to find a test case. The processing time for this moderate example is reasonable, even though up to 200 MB of storage is required for some intermediate results. At the conclusion of the process, many of the shorter test paths are subsumed by longer paths, and many of the paths are connectable end-to-end to produce *executable test cases* that test multiple *def-use* pairs. We intend to pursue the development of efficient executable test case development from candidate test paths in subsequent research efforts, probably adapting algorithms that were previously developed for specification-based testing [35].

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, then additional work will need to be done to apply the resulting tests to the software; specifically, the test specifications will need to be translated 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 C is used to derive a Class State Machine as defined in Definition 2.1. Using the relational database model [12, 13, 32], classes and sets associated with classes are represented as relational tables.

Figure 3 shows the UML class diagram [40] of a general schema definition for representing 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 represents a table in the model and each row of the table identifies an instance of that class: (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 Class 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, which is the primary key of the Class 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. 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 object by invoking the appropriate actor method, if it is available, to read the value of that external 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 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.

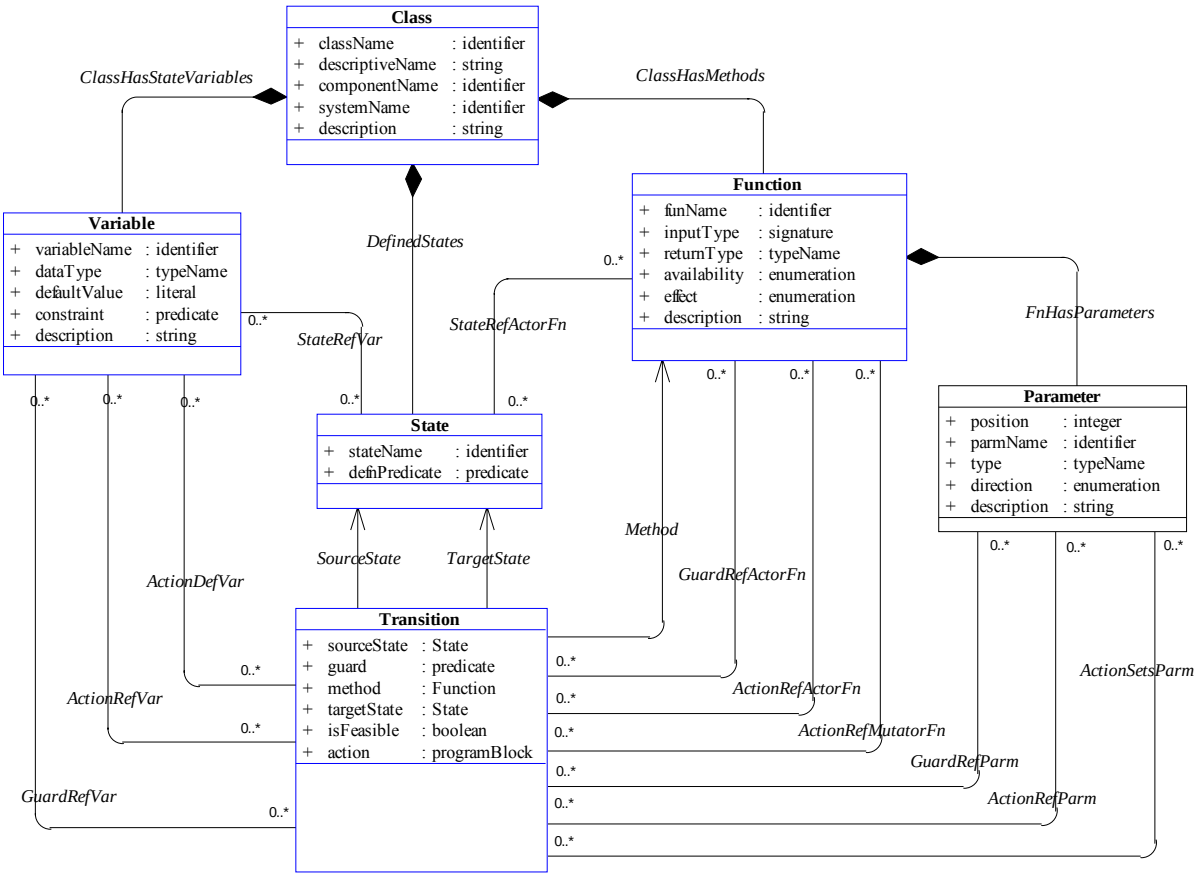

Figure 3: DataBase Schema as a UML Class Diagram

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 5. It is an error to push the button to go to floor 5.
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 transitions may reference variables, and the actions of predicates may reference and assign values to variables. Just as in traditional data flow analysis [15], 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 prototype implementation uses a general simple language to describe state machines, which allows the analysis to proceed in a fairly straightforward manner. In subsequent work, we hope to expand this part of the prototype to include syntactic analysis of predicates and actions specified in UML [40], Java [24], and other commonly used class definition languages.

Once this syntactic analysis is complete, the results can be captured in the UML diagram of Figure 3 as many-to-many associations among classes. In the database representation, each such association will be a new table. Each of the new 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 pre-defined 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 3 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 3-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 I shows examples of the first and second alternatives, e.g. 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.

It is sometimes necessary to consider the case where the action of a transition makes 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 2 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 transitions seen in Appendix I have final actions that put CheckState() on a queue to be executed by CruiseUnit when it's 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. A straightforward mapping does this. 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, 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 high. Developing tests over such a large scope would probably be prohibitively expensive, and would properly be considered system testing as well. Testing is divided into pieces by focusing on one component at a time, and generating tests based on that component's integration interactions with other components.

The *test component M* is the component whose interactions are being tested. The procedure first determines which transitions from the overall system specification are *relevant* to M. Relevant transitions fall into two types. *In* transitions represent actions or data that flow into M, that is, transitions from any class 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 classes in other components, that is, transitions that can be invoked, directly or indirectly from actions on transitions in any of M's classes. Transitions from classes in M are called *Base* transitions, since they are the starting points for a recursive process that finds the transitive closure of relevant transitions.

This process begins by putting all feasible *Base* transitions from any class in M into the 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 labeled as *In*, *Out*, or *Base*. A 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 labeled 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 and when transitions are handled. In some cases, decisions were made to try to balance performance with effectiveness. Further experimentation may cause some decisions to be refined.

- 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 State, 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* labeled 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 labeled 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 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 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, labeled 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, labeled as an *In* transition. If the method associated with t' is the *get* method for the variable 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, labeled 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, 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 unlabeled transitions are defined to be the set of transitions in the system that are *relevant* to M. These are the transitions that will determine the *component flow graph* when integrating M with the system.

Definition 4.1 (relevant transitions): Let M be any component of a software system ***S***. R(M) is the set of all transitions from ***S*** 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 cruise control. The above procedure focuses only on transitions relevant to CruiseControl and eliminates these unrelated transitions. Each relevant transition that has a non-trivial action is listed in Appendix I.

5 A Data-flow Graph Model of State Transitions

The traditional testing literature [15, 26, 33, 37, 39] defines a *data flow graph* to be a graphical 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* is defined to represent both the control and data flows for the state transitions of the classes of a component and its relevant transitions from other classes in the software system. The definitions in this paper extend those of Hong et al. [22] 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 3.

Definition 5.1 (component flow graph): Let M be any component of a software system ***S***, and let R (M) be the set of all transitions in ***S*** that are relevant to M. Then the *component flow graph G of M in* ***S*** 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 source states or target states 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 in the action sequence of a transition that call methods from other classes. Each edge that results from a call to any external function is labeled with the sequence number of that call in the action sequence of the transition. However, it helps to distinguish these labels as being on out-going edges or on in-coming edges, so the sequence number label for an edge that represents an out-going call of a mutator function is defined to be the OutSeq number and the sequence number label for an edge that represents an in-coming 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 unlabeled. No edge carries more than one such label.

Nine types of edges are defined. Four of these types come from Hong et al.'s paper [22] 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, four new inter-class edge types and one new intra-class edge type 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 *asynchronous*, as explained below. The total set of edges E is defined as:

$$E = E_{st} \cup E_{sg} \cup E_{gt} \cup E_{ts} \cup E_{gtg} \cup E_{sts} \cup E_{xts} \cup E_{xtt} \cup E_{cts}$$

Hong's four original intra-class edge types are:

- E_{st} edges represent data flow from states to transitions. The the transition has no non-trivial guard (guard is *true*).
- 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.
- E_{gt} edges represent data flow from guards to transitions. The guard is non-trivial and is specified by the transition.
- E_{ts} edges represent data flow from transitions to states. The state is the target state of the transition.

There are four inter-class, potentially asynchronous types of edges. 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 in or out of that node.

- E_{gtg} edges represent data flow triggered by a guard that flows from an external transition back to that guard. The predicate of the guard invokes an actor function from an external class and data flows from transitions in that class back to the guard. The *GuardRefActorFn* association determines these edges. The Automobile example has three instances of this type of edge.
- E_{sts} edges represent data flow triggered by a state that flows from an external transition back to that state. The predicate of the state invokes an actor function from an external class and data flows from transitions in that class back to the state. The *StateRefActorFn* association determines these edges and the Automobile example has 10 instances.
- E_{xts} edges represent data flow triggered by an external 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 the 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 the state the other object might be in when the request is received cannot be known. These out-going edges are labeled 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 labeled with the function name of the mutator function. Section 6 defines additional conditions on path segments from the transition node, to a source state node, to a guard node of a transition derived from the called mutator function. The *ActionRefMutatorFn* association determines these edges and the Automobile example has 161 instances.
- E_{xtt} edges represent data flow from an external transition to a transition in a different class. The action of the target transition invokes a method from an external class and data flows from any transition in that class derived from that function back to the target transition. These in-coming edges are labeled 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 edges and the Automobile example has 58 instances.

There is one new intra-class asynchronous edge type:

- E_{cts} edges represent intra-class data flow from transitions to states. The transition calls a mutator function, asynchronously, in its own class. 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 out-going edges are labeled with an OutSeq number equal to the SeqNbr of the method call in the action sequence of the transition. These edges are also labeled with the function name of the mutator function. The *ActionRefLocalAsyn* association determines these edges and the Automobile example produces 38 instances.

Section 5 of an earlier technical report [18] provides a more formal specification of how these edges are derived from the referenced associations.

Transition nodes whose method has External (EXT) availability determine the external interface to the system. Input values can only be provided through this interface in black box testing. Such transitions are marked with a virtual edge from a virtual EXT User node. In the Automobile example, the 8 EXT methods listed in Section 2.2 produce 24 such virtual edges. Various combinations of these 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 these EXT methods to execute those paths. The paths through the graph are called *test specifications* and the input values are called *executable test cases*.

6 Generating Test Requirements

A *testing criterion* is a rule or collection of rules 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 percent 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 reached.

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 [15, 33]. Many researchers have concluded that the *all-defs* and *all-uses* criteria provide adequate coverage at acceptable cost for most testing purposes [10, 16, 17, 20, 23, 31, 34].

The formal definitions for variable definitions and variable uses to the component flow graphs defined in the preceding section are in a previous technical report [18] and are presented informally here. First, the various types of uses (direct/indirect, predicate/computation) are defined, and then used to define *def-use* pairs and then DU-pairs.

Defs and uses are defined in terms of the associations defined in the DB schema of Figure 3. 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 contains the variable, that is $c \in C$.

Definition 6.1 (definitions and uses): Let M be any component of a software system ***S***, let R(M) be the set of transitions in ***S*** that are relevant to M, and let G = (N, E) be the component flow graph of M in ***S***.

- 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, v) ∈ *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, v) ∈ *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) ∈ *ActionRefActorFn*.
- ν is *directly predicate-used* at any state-transition-edge $(n_s, n_t) \in E_{st}$ if the state satisfies the association (c, s, v) ∈ *StateRefVar*.
- ν is *indirectly predicate-used* at any state-transition-edge $(n_s, n_t) \in E_{st}$ 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) ∈ *StateRefActorFn*.
- ν is *directly predicate-used* at any state-guard-edge $(n_s, n_g) \in E_{sg}$ if the state satisfies the association (c, s, v) ∈ *StateRefVar*.
- ν is *indirectly predicate-used* at any state-guard-edge $(n_s, n_g) \in E_{sg}$ 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) ∈ *StateRefActorFn*.
- ν is *directly predicate-used* at a guard-transition-edge $(n_g, n_t) \in E_{gt}$ if the transition satisfies the association (c_t, t, c, v) ∈ *GuardRefVar*.
- ν is *indirectly predicate-used* at a guard-transition-edge $(n_g, n_t) \in E_{gt}$ 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) ∈ *GuardRefActorFn*.
- ν 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) ∈ *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_{t1}, t_1, c_f, f, n, c, v) ∈ *ActionSetsParmUsingVar*, and if that function is the function associated with t, that is if $c_t = c_f$ and method(t) = f.
- ν is *parameter predicate-used* at a guard-transition-edge $(n_g, n_t) \in E_{gt}$ 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) ∈ *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_{t1}, t_1, c_f, f, n, c, v) ∈ *ActionSetsParmUsingVar*, and if that function is the function associated with t, that is if $c_t = c_f$ and method(t) = f.

Each *computation-used* instance carries along the SeqNbr attribute of the association to identify the position of that use in the action sequence of the transition. Since guard and state predicates do not have sequence numbers, *predicate-used* 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.

Definition 6.2 (def-use pairs): Let M be any component of a software system ***S***, let R (M) be the set of transitions in ***S*** that are relevant to M, and let G = (N, E) be the component flow graph of M in ***S***. The Greek mu (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 ν if ν *is defined at the transition-node* n_t and if μ is either a node or an edge in G where ν is *directly* or *indirectly used*.[1]

Not every variable produces a non-empty set of def-use pairs. Some variables, 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.

Special attention is paid to transition nodes where a variable is both defined and used. Here the order of execution is important, since a variable may be defined and then used in the same action. If a variable is used first in an action before it is defined, or if it is defined later after it is used, then that node may continue to be relevant to other definitions or uses of the variable. These cases are distinguished as follows:

Definition 6.3 (internal def-use pairs): Let ν be a variable that is both defined and used at one or more transition nodes $n_t \in N_t$. Denote by DFTU(ν) the set of such nodes where ν is *defined first and then used*, and denote by UFDL(ν) the set of all such nodes where ν is *used first* before it is defined or *defined later* after it is used. In each case, the content of the set is determined by a syntactic analysis of the action associated with the transition node n_t.

The sets DFTU(ν) and UFDL(ν) are not necessarily mutually exclusive. A transition involving variable x with an action that consists of the sequence "x := x+1; y := f(x)" would be in both sets.

6.1 Data flow path coverage

To complete the *def-use* approach to test specification creation, the algorithm looks for paths in the component flow graph that lead from the definition of a variable to a use. Consider triples (ν, n_t, μ) where ν is a variable, n_t is a transition node that defines ν, and μ is a node or edge where ν is used. n_t and μ form a *DU-pair* if there exists a path in the component flow graph leading from n_t to μ, if the path is free of loops, if there are no defs to ν 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 object components, and lead to a rigorous definition of *test specifications* derived from a component flow graph.

Definition 6.4 (path): Let G = (N, E) be an directed graph. A *path* p in G of *length* k≥1 is a sequence of nodes n_1 .. n_k such that $(n_i, n_{i+1}) \in E$ for $1 \le i \le k-1$. 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 L(p), is the number of nodes in the sequence. If p and q are two paths such that $(T(p), H(q)) \in E$, then the concatenation of the two sequences, denoted by p:q, is a path with L(p:q) = L(p) + L(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,

[1] Note that a def-use pair is distinct from a DU-pair in that the def-use pair does not require that there be a def-clear path from the def to the use.

denoted by $n \in p$. If p is a path then InSeq(p) or OutSeq(p) denotes the label of its first or last edge. The context makes clear which is intended.

Feasible paths through a component flow graph must be found, 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 another transition node n_{t2}. If the edge from n_{t1} to n_s is the result of a call of a mutator function f, that is if the edge has a function label that identifies 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 not be incompatible with the exit conditions from node 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 (n_t, n_s) has a function label. A mutator TSG path segment is *potentially feasible* if the edge (n_s, n_g) is known not to be incompatible 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 283 instances of MSTG, of which 169 are provably feasible and 53 are provably not feasible, leaving 61 where a simple analysis cannot determine feasibility or non-feasibility. Appendix I 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 (230 instances in the **Automobile** example).

Definition 6.5 (DU-path and DU-pair): Let G = (N, E) be a component flow graph in a software system ***S***. Let ν be any variable in ***S***, let n_t be a transition node that defines ν, and let μ be a node or an edge where ν is used. A path p in G is said to be a *DU-path from* n_t *to* μ *for* ν if $p = n_t{:}q{:}\mu$, where q is a path in G such that no node of q is a definition node for ν and every mutator TSG path segment in p is potentially feasible . The pair (n_t, μ) is said to be a *DU-pair* for ν if such a path p exists.

Definition 6.6 (candidate test paths): Let G = (N, E) be a component flow graph in a software system. Let VDU be a set of tuples (ν, n_t, μ) where (n_t, μ) is a def-use pair for ν and let P be a set of tuples (ν, n_t, μ, p) where (n_t, μ) is a *DU-pair* for ν 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.

The *all-uses* testing criterion is satisfied by any path from a def to a use. The construction below looks for the shortest path because it is more convenient, thus saving computation expense. It is, however, possible that other paths could be "better" in some sense. A reasonable alternative would be to incorporate a searching procedure that uses some measurement function to choose from among a set of potential paths. One measurement might be to require that all mutator TSG path segments be known feasible instead of just known not infeasible, but that is a very difficult measurement to determine or represent.

It is easy to construct the set VDU of Definition 6.6, but the set P may not have any elements. An iterative procedure is defined to construct the elements of P. 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. 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 mutator TSG 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 (that is, 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 CruiseUser. Finally, in order to help ensure the construction of DU-paths that result in feasible test cases the construction of both sets of partial paths is required to satisfy a set of rules involving SeqNbr, InSeq, and OutSeq labels to ensure that edges entering or leaving a transition node occur in a feasible order for the action sequence of that transition.

The iterative process stops when the 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 $(\nu, n_t, \mu) \in VDU$ 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 re-definition of ν.

Definition 6.7 (def-bound): A variable ν 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 from n_t to μ $(p = (\nu, n_t, \mu, p) \in P)$.

The *def-bound* variables surface during the calculation of $B_{i+1} = X_i - A_{i+1}$ in the iterative process of Definition 6.6. 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 did not exist a 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.

6.2 Executable test cases

If a variable ν is both defined and used, and is not *def-bound* for a specific *def-use* pair, then the path generation of the previous section produces 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 one will be in the collection P of candidate test paths constructed by the algorithm of Definition 6.6. If at the end of iteration i, all DU-paths for a DU-pair are discovered to be not feasible, then the def-use pair is re-inserted into the set X_i of active pairs and the iterative algorithm continues.

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 invokable methods that will trigger each of the function calls in the abstract test specification without violating any of the constraints against re-definition of the state variable. The authors believe that the methodology presented in this paper can be used to help find such externally invokable methods. In particular, the algorithm of Definition 6.6 can be used to find potentially feasible paths from the set of externally invokable methods to each of the function calls in an abstract test specification that is not the result of an internal call. Subsequent research will attempt to use this methodology to help generate executable test cases automatically from abstract test specifications.

Each DU-pair is equally important because it tests a distinct *def* and *use* of some variable. Even if two different DU-pairs share essentially the same DU-path, an executable test case that follows that path is an effective test case for each DU-pair. Some paths are included as a subpath within other paths, or shorter paths may be connected end-to-end to produce longer paths, so a traversal of a longer path by an executable test case may test multiple abstract aspects of the state/transition specification at the same time. From a theoretical perspective, they should still be counted as separate tests. In any statistical analysis of test case development, it may safely be assumed that the set of all DU-pairs is the sample space from which all executable test cases are drawn. Such statistical analysis is left as future work.

7 Empirical Results on the Automobile System

This section presents results from testing the Automobile example introduced in Section 2.2 and its CruiseControl component. Cruise has been used widely in the specification, specification-based testing, and modeling literature [1, 4, 19], but the version used in this paper includes significantly more components than other versions. The version used by Atlee and Abdurazik et al. [1, 4] had seven functions, 184 blocks, and 174 decisions. The external interface and the cruise control transitions used in this paper are modeled on the cruise control characteristics of a 1995 Acura Legend. Instead of the four states found in the other papers, the system used in this paper contains 10 classes, each of which has a number of states. Combined, these states have 21 relevant variables that appear in more than 3433 def-use pairs. For cruise control testing purposes, only external functions such as clutch and gas pedal positions and the cruise controls are available to human users. Other functions are encapsulated and hidden.

Each process in Sections 3 through 6 are followed, using 16 iterations and resulting in the data shown in Table 2. The Process Time column is from the prototype implementation using an Access database on a Pentium 4 class PC at 1.5 Ghz and 256 MB RAM. Other columns are explained below.

	New Paths P_i	**New DU-pair** C_i	**Active Pairs** X_i	**New DefnBnd** B_i	**Partial Paths** Q_i	**Process Time (m:ss)**

1	18	18	3433	99		0:01
2	0	0	3316	0	3316	0:01
3	363	363	2948	5	4174	0:03
4	69	69	2879	0	15,077	0:27
5	291	287	2564	28	49,664	1:02
6	526	355	2209	0	71,697	1:25
7	209	85	2080	44	25,851	0:33
8	330	153	1117	810	19,122	0:22
9	130	109	938	70	18,752	0:17
10	263	263	665	10	14,401	0:25
11	231	214	445	6	46,509	0:59
12	26	17	428	0	50,822	0:27
13	0	0	428	0	10,206	0:23
14	0	0	428	0	12,130	0:18
15	0	0	420	8	0	0:05
16	0	0	0	420	0	0:01
Totals	2456	1933		1500		6:47

Table 2: Cruise Control – Candidate Test Paths

Table 2 shows that iteration 5 finds 291 new DU-paths, but only 287 of them identify new DU-pairs. In addition, 28 pairs were found to be *def-bound* (B_i). The number of active pairs (X_i) is thus reduced by 287 and 28. 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. However, some of the paths introduce the first transition-to-transition edges. For example, Rpm of Engine is defined in transition t003, but can reach its parameter computation-use in Gauges either by going through the target state of t003 to Gauges via a call to Engine to read the value of ExternalDrag or by being passed as a parameter via a call of Gauges.Speed(x) to set the Speed variable in Gauges. A tester could choose either path to test the def-use of Rpm, but might be biased toward the transition-to-transition path because it does not contain any potentially infeasible MTSG path segments. Similarly, the Speed variable of Gauges is defined in t017 then called by many CruiseUnit transitions for indirect computation-use. This iteration also discovers 28 new *def-bound* pairs, primarily because ThrottlePosition is defined in all of the relevant Throttle transitions but its predicate use in many edges coming out of the Danger state can never be reached.

All DU-path generation takes place in iterations 3 through 12. Iterations 13 through 16 follow potentially feasible paths until it is no longer possible to extend either the def-partial or use-partial paths without violating one of the path constraints (no new paths are added). At iteration 16, all 3433 def-use pairs are resolved, finding candidate test paths for 1933 pairs and proving that the remaining 1500 pairs are *def-bound* with no possible *def-free* test path.

7.1 Experimentation with test cases

Once the candidate test paths are found, executable test cases are constructed by finding appropriate external calls to execute the methods on the candidate test paths and appropriate parameter values. Tools for automating this step are under construction. As an experimental evaluation, we have constructed the tests (145), seeded faults into the program (106), and evaluated the fault-finding ability of the tests on the seeded faults. The subjects (full specification Engine specifications and tests) are shown in appendixes of this report; the results will appear in a forthcoming paper. Faults were constructed by modifying the transitions table in the specification database (Appendix VI). Each fault was created by copying the table and making one change, resulting in 106 copies of the table. These tables will be provided on request.

8 Conclusions and Future Work

This technical report presents theoretical concepts for constructing tests for component-based testing. This is a method for integration level, inter-class testing for object-oriented programs using data flow techniques. The data flow and control flow graphs are stored in a relational database, which is used as a compute engine for deriving DU-pairs and DU-paths to satisfy data flow testing criteria. Software components are modeled 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.

The database representation provides a convenient 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 statements), 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 [36]. 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.

Traditional code-level data systems do not provide complete paths for data flow testing, partially because the problems of finding a feasible path and determining whether the path is def-clear are generally undecidable. In cases where the problem can be solved, the complexity of the control flow, problems with aliasing and function calls, 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 the database is that it provides a powerful compute engine for solving predicates, which is one of the most difficult parts of a data flow analyzer to implement.

Although it is true that this work thus far has not assured scalability, the authors have experience both building and using source code-level data flow analysis software. We know of **no** source code-level data flow testing systems, either commercial or experimental, which can handle software specifications that have thousands of DU-pairs, as we have done for the Engine system.

This paper does not explicitly handle class variables (Java static) or inheritance. However, class variables can be modeled 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 this model.

For clarity, the definitions and example in this paper only consider one object per class. 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 behavior 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, 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. 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. We hope to explore this idea in future work.

With the increasing popularity of object-oriented specification methods, e.g. UML [40], and especially state transition specification of classes, e.g. UML's state machine package, it becomes possible to more closely align 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.

Acknowledgements

It is a pleasure to acknowledge Roger Alexander, Paul Black, and the reviewers for a number of helpful suggestions.

References

[1] A. Abdurazik, P. Ammann, W. Ding, and J. Offutt, "Evaluation of Three Specification-based Testing Criteria", in P*roceedings of the Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, pp. 179-187, Tokyo, Japan, September, 2000.

[2] Roger Alexander and Jeff Offutt, "Analysis Techniques for Testing Polymorphic Relationships," *Proceedings of the Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '99),* August 1999, Santa Barbara, CA, 104-114.

[3] Roger Alexander and Jeff Offutt, "Criteria for Testing Polymorphic Relationships," *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE '00),* October, 2000, San Jose, CA, 15-23.

[4] J. M. Atlee, "Native Model-checking of SCR Requirements", in *Proceedings of the Fourth International SCR Workshop*, November 1994.

[5] D. Banks, W. Dashiell, L. Gallagher, C. Hagwood, R. Kacker, L. Rosenthal, *Software Testing by Statistical Methods: Preliminary Success Estimates for Approaches Based on Binomial Models, Coverage Designs, Mutation Testing and Usage Models*, NISTIR 6129, U.S. National Institute of Standards and Technology, March 1998. http://www.nist.gov/stsm.html

[6] G. Booch, *Object Oriented Design with Applications*, Benjamin Cummings, 1991.

[7] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen, "In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs." *ACM Transactions on Software Engineering Methodology*, 7(3):250-295, 1998.

[8] H. Y. Chen, T. H. Tse, and T. Y. Chen, "TACCLE: A Methodology for Object-Oriented Software Testing at the Class and Cluster Levels," *ACM Transactions on Software Engineering Methodology*, 10(4):56-109, 2001.

[9] Mei-Hwa Chen and Ming-Hung Kao, "Testing Object-Oriented Programs -- An Integrated Approach," *Proceedings of the 10th International Symposium on Software Reliability Engineering*, IEEE Computer Society, November 1999, Boca Raton, FL, 73-83.

[10] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A Comparison of Data Flow Path Selection Criteria," Proceedings of the Eighth International Conference on Software Engineering, IEEE Computer Society Press, London UK, August 1985, pp. 244-251.

[11] T. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 3, May 1978, pp. 178-187.

[12] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," in *Communications of the ACM*, Vol. 13, No. 6, June 1970, pp. 377-387, reprinted in Vol. 26, No. 1, Jan. 1983.

[13] C. J. Date, *An Introduction to Database Systems*, 6th edition, Addison-Wesley, 1995.

[14] R. K. Doong and P. G. Frankl, "The ASTOOT Approach to Testing Object-Oriented Programs," ACM *Transactions on Software Engineering and Methodology*, 3(2):101-130, April 1994.

[15] P. G. Frankl and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria" *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, Oct. 1988, pp. 1483-1498.

[16] P. G. Frankl and S. N. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," Transactions on Software Engineering, 19(8):774-787, August 1993.

[17] P. G. Frankl, S. N. Weiss, and C. Hu, "All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness," The Journal of Systems and Software, Elsevier North Holland Inc, 1997, 38(3):235-253.

[18] L. J. Gallagher, Conformance Testing of Object-oriented Components Specified by State/Transition Classes, National Institute of Standards and Technology Technical Report NISTIR 6592, May 1999.
ftp://xsun.sdct.itl.nist.gov/stsm/NISTIR6592.pdf.

[19] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison-Wesley, 2000.

[20] M. J. Harrold and M. L. Soffa, "Selecting and Using Data for Integration Testing," IEEE Software, 8(2): 58—65, March 1991.

[21] M. J. Harrold and G. Rothermel, "Performing Data Flow Testing on Classes," in *Proceedings of 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, Dec. 1994, pp. 154-163.

[22] H. S. Hong, Y. R. Kwon, and S. D. Cha, "Testing of Object-Oriented Programs Based on Finite State Machines," in Proceedings of Asia-Pacific Software Engineering Conference, pp. 234-241, 1995.

[23] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow-and Controlflow-Based Test Adequacy Criteria," Proceedings of the Sixteenth International Conference on Software Engineering, IEEE Computer Society Press, May 1994, Sorrento, Italy, pp. 191-200.

[24] Java Development Kit, version 1.2, Sun Microsystems, Inc., Copyright . 1995, *http://java.sun.com/products/jdk/1.2.*

[25] Zhenyi Jin and Jeff Offutt, "Coupling-based Criteria for Integration Testing," *The Journal of Software Testing, Verification, and Reliability*, 8(3):133-154, September 1998.

[26] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On Object State Testing," in *Proceedings of Computer Software and Applications Conference*, 1994, pp. 222-227.

[27] D. Kung, C. H. Liu and P. Hsia, "An Object-oriented Web Test Model for Testing Web Applications," in *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC 2000)*, IEEE Computer Society, October 2000, Taipei Taiwan, 73-83.

[28] D. Kung, J. Gao, Pei Hsia, Y. Toyoshima, and C. Chen, "A Test Strategy for Object-oriented Programs," 19th Computer Software and Applications Conference (COMPSAC 95), August 1995, Dallas, TX, pp. 239-244.

[29] R. J. Linn and M. Ü. Uyar, *Conformance Testing Methodologies and Architectures for OSI Protocols*, IEEE Computer Society Press, 1994.

[30] C. H. Liu, D. Kung, P. Hsia and C. T. Hsu, "Structural Testing for Web Applications," *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE 2000*, IEEE Computer Society, October 2000, San Jose CA, 84-96.

[31] A. P. Mathur and W. E. Wong, "An Empirical Comparison of Data Flow and Mutation-based Test Adequacy Criteria," Journal of Software Testing, Verification and Reliability, Wiley, 4(1):9-31, March 1994.

[32] J. Melton and A. Simon, *Understanding the New SQL: A Complete Guide*, Morgan Kauffman, 1993.

[33] S. C. Ntafos, "A Comparison of Some Structural Testing Strategies," *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, June 1988, pp. 868-874.

[34] A. J. Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang, "An Experimental Evaluation of Data Flow and Mutation Testing," Software--Practice and Experience, 26(2):165-176, February 1996.

[35] J. Offutt, Generating Test Data From Requirements/Specifications: Phase II Final Report, Technical Report ISE-TR-99-01, Department of Information and Software Engineering, George Mason University, Fairfax VA, January 1999, http://www.ise.gmu.edu/techrep/.

[36] J. Offutt , Z. Jin and J. Pan, "The Dynamic Domain Reduction Approach to Test Data Generation", Software--Practice and Experience, January 1999, 29(2), pp. 167-193.

[37] A. S. Parrish, R. B. Borie, and D. W. Cordes, “Automated Flow Graph-Based Testing of Object-Oriented Software Modules,” *Journal of Systems and Software*, 23, 1993, pp. 95-109.

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

[39] C. D. Turner and D. J. Robson, “The State-based Testing of Object-Oriented Programs,” in *Proceedings of the Conference on Software Maintenance*, 1993, pp. 302-310.

[40] Unified Modeling Language, Object Constraint Language Specification and UML semantics, version 1.1, Sept. 1997, Rational Software, *http://www.rational.com/uml/.*

[41] Hong Zhu, Patrick A. V. Hall, and John H. R. May, "Software Unit Test Coverage and Adequacy," ACM Computing Surveys, 29(4):366-427, December 1997.

Appendices

The appendices to this report include the full specifications for the Engine system and executable test cases. The specifications are in several tables. Appendix I lists the thirteen classes of the system. Appendix II lists the external functions of all the classes. Appendix III lists the parameters for each of the external function. Appendix IV lists the states for all objects in the Engine system. Appendix V lists the state variables for the classes in Engine. Appendix VI lists the mutator transitions for all the classes in the Engine. Appendix VII lists the executable test cases generated by our technique, in the form of sequences of calls to external functions.

Appendix I: Classes for CruiseControl

ClassId	ClassAlias	ClassName	ComponentName	SystemName
c01	AutoSystem	AutoSystem_ADT	SystemControl	Automobile
c02	BrakeControl	BrakeControl_ADT	Brakes	Automobile
c03	BrakeUser	BrakeUser_ADT	Brakes	Automobile
c04	ClutchUser	ClutchUser_ADT	Clutch	Automobile
c05	CruiseUnit	CruiseUnit_ADT	CruiseControl	Automobile
c06	CruiseUser	CruiseUser_ADT	CruiseControl	Automobile
c07	Engine	Engine_ADT	Engine	Automobile
c08	GasUser	GasPedalUser_ADT	Acceleration	Automobile
c09	Gauges	Gauges_ADT	InstrumentPanel	Automobile
c10	Throttle	ThrottleUnit_ADT	Acceleration	Automobile
c11	Ignition	Ignition_ADT	IgnitionControl	Automobile
c12	Transmission	Transmission_ADT	TransmissionBox	Automobile
c13	Wheel	Wheel_ADT	WheelHousing	Automobile

Appendix II: External Functions for CruiseControl

ClassAlias	Avail	FunName	InputType	ReturnType	Description
AutoSystem	PRO	AutoSystem()		AutoSystem	Creates a new instance of the AutoSystem ADT.
AutoSystem	PUB	BrakeActive()		Boolean	Read value.
AutoSystem	PUB	BrakeActive(x)	Boolean	Boolean	Set value, returns true if successful.
AutoSystem	PUB	ClutchActive()		Boolean	Read value.
AutoSystem	PUB	ClutchActive(x)	Boolean	Boolean	Set value, returns true if successful.
AutoSystem	PUB	Danger()		Boolean	Read value.
AutoSystem	PUB	Danger(x)	Boolean	Boolean	Set value, returns true if successful.
AutoSystem	EXT	ThrottleFloor()		Boolean	Read value of CONSTANT.
AutoSystem	EXT	ThrottleGovernor()		Boolean	ReadValue or CONSTANT.
BrakeControl	PRO	BrakeControl()		BrakeControl	Creates a new instance of the BrakeControl ADT.
BrakeControl	PRO	IsActive()		Boolean	Read value.
BrakeControl	PRO	IsActive(x)	Boolean	Boolean	Set value, returns true if successful.
BrakeControl	PRO	WheelsTurning()		Number(2)	Read value.
BrakeControl	PRI	WheelsTurning(x)	Number(2)	Boolean	Set value, returns true if successful.
BrakeControl	PRO	LinePressure()		Number(2)	Read value.
BrakeControl	PRI	LinePressure(x)	Number(2)	Boolean	Set value, returns true if successful.
BrakeControl	PRO	PedalPressure()		Number(2)	Read value.
BrakeControl	PRO	PedalPressure(x)	Number(2)	Boolean	Set value, returns true if successful.
BrakeUser	PRO	BrakeUser()		BrakeUser	Creates a new instance of the BrakeUser ADT.
BrakeUser	EXT	IsActive()		Boolean	Read value.
BrakeUser	EXT	IsActive(x)	Boolean	Boolean	Set value, returns true if successful.
BrakeUser	EXT	PedalPressure()		Number(2)	Read value.
BrakeUser	EXT	PedalPressure(x)	Number(2)	Boolean	Set value, returns true if successful.
ClutchUser	PRO	ClutchUser()		ClutchUser	Creates a new instance of the ClutchUser ADT.
ClutchUser	EXT	PedalPosition()		Number(2)	Read value.
ClutchUser	EXT	PedalPosition(x)	Number(2)	Boolean	Set value, returns true if successful.
CruiseUnit	PRO	CruiseUnit()		CruiseUnit	Creates a new instance of the CruiseUnit ADT.
CruiseUnit	PUB	UserSwitch()		Enum(On,Off)	Reads the Switch state.
CruiseUnit	PUB	UserSwitch(x)	Enum(On,Off)	Boolean	Sets the Switch state, returns true if successful.
CruiseUnit	PUB	UserMode()		Enum(Null,NT,RA,SD)	Reads the Mode state.
CruiseUnit	PUB	UserMode(x)	Enum(Null,NT,RA,SD)	Boolean	Sets the Mode state, returns true if successful.
CruiseUnit	PRI	CurrentSpeed()		Number(4,1)	Reads CurrentSpeed variable.
CruiseUnit	PRI	CurrentSpeed(x)	Number(4,1)	Boolean	Sets CruiseSpeed variable, returns true if successful
CruiseUnit	PRI	TargetSpeed()		Number(4,1)	Reads TargetSpeed variable.
CruiseUnit	PRI	TargetSpeed(x)	Number(4,1)	Boolean	Sets TargetSpeed variable, returns true if successful.
CruiseUnit	PRI	TargetThrottle()		Number(2)	Reads TargetThrottle variable.
CruiseUnit	PRI	TargetThrottle(x)	Number(2)	Boolean	Sets TargetThrottle variable, returns true if successful.

ClassAlias	Avail	FunName	InputType	ReturnType	Description
CruiseUnit	PUB	Cancel()		Boolean	An interrupt - halts any active state, puts system in Override state, returns true if successful.
CruiseUnit	PRI	SetSpeed()		Boolean	Sets CurrentSpeed from Gauges.
CruiseUnit	PRI	CheckState()		Boolean	Reads all gauges, checks all state variables, decides next action.
CruiseUser	PRO	CruiseUser()		CruiseUser	Creates a new instance of the CruiseUser ADT.
CruiseUser	EXT	Switch()		Enum(On,Off)	Reads the Switch state.
CruiseUser	EXT	Switch(x)	Enum(On,Off)	Boolean	Sets the Switch state, returns true if successful.
CruiseUser	EXT	Mode()		Enum(NT,RA,SD)	Reads the Mode state.
CruiseUser	EXT	Mode(x)	Enum(NT,RA,SD)	Boolean	Sets the Mode state, returns true if successful.
CruiseUser	EXT	Cancel()		Boolean	Sends Cancel message to the CruiseUnit ADT, returns true if message successfully sent.
Engine	PRO	Engine()		Engine	Creates a new instance of the Engine ADT.
Engine	PRI	Rpm()		Number(4)	Read value.
Engine	PRI	Rpm(x)	Number(4)	Boolean	Set value, returns true if successful. Continuous Update by private process.
Engine	PUB	GasFlow()		Real	Read value.
Engine	PUB	GasFlow(x)	Real	Boolean	Set value, returns true if successful. Controlled by external calls.
Engine	PRI	Check()		Boolean	Check all state variables to see if move to new state.
Engine	PUB	ExternalDrag()		Real	Used to simulate hills and wind resistance
Engine	EXT	ExternalDrag(x)	Real	Boolean	Used to simulate hills and wind resistance (0,2) 1 is neutral.
GasUser	PRO	GasUser()		GasUser	Creates a new instance of the GasUser ADT.
GasUser	EXT	PedalPosition()		Number(2)	Read value.
GasUser	EXT	PedalPosition(x)	Number(2)	Boolean	Set value, returns true if successful.
Gauges	PRO	Gauges()		Gauges	Creates a new instance of the Gauges ADT.
Gauges	EXT	Odometer()		Number(7,1)	Read value.
Gauges	PUB	Odometer(x)	Number(7,1)	Boolean	Set value, returns true if successful.
Gauges	EXT	TripMeter()		Number(5,1)	Read value.
Gauges	PUB	TripMeter(x)	Number(5,1)	Boolean	Set value, returns true if successful.
Gauges	EXT	Tach()		Number(4)	Read value.
Gauges	PUB	Tach(x)	Number(4)	Boolean	Set value, returns true if successful.
Gauges	EXT	Speed()		Number(3)	Read value.
Gauges	PUB	Speed(x)	Number(3)	Boolean	Set value, returns true if successful.
Gauges	PUB	OilPressure()		Number	Read value.
Gauges	EXT	OilPressure(x)	Number	Boolean	Set value, returns true if successful.
Gauges	PUB	WaterTemp()		Number	Read value.
Gauges	EXT	WaterTemp(x)	Number	Boolean	Set value, returns true if successful.
Gauges	EXT	Cruise()		Enum(On,Off)	Read value.

ClassAlias	Avail	FunName	InputType	ReturnType	Description
Gauges	PUB	Cruise(x)	Enum(On,Off)	Boolean	Set value, returns true if successful.
Gauges	EXT	AbsLight()		Enum(On,Off)	Read value.
Gauges	PUB	AbsLight(x)	Enum(On,Off)	Boolean	Set value, returns true if successful.
Gauges	EXT	Battery()		Enum(On,Off)	Read value.
Gauges	PUB	Battery(x)	Enum(On,Off)	Boolean	Set value, returns true if successful.
Gauges	EXT	OilLight()		Enum(On,Off)	Read value.
Gauges	PRI	OilLight(x)	Enum(On,Off)	Boolean	Set value, returns true if successful.
Gauges	EXT	SeatBelt()		Enum(On,Off)	Read value.
Gauges	EXT	SeatBelt(x)	Enum(On,Off)	Boolean	Set value, returns true if successful.
Gauges	EXT	HandBrake()		Enum(On,Off)	Read value.
Gauges	EXT	HandBrake(x)	Enum(On,Off)	Boolean	Set value, returns true if successful.
Gauges	EXT	LowGas()		Enum(On,Off)	Read value.
Gauges	EXT	LowGas(x)	Enum(On,Off)	Boolean	Set value, returns true if successful.
Throttle	PRO	Throttle(x,y)	(Number(2),Number(2))	Throttle	Creates a new instance of the Throttle ADT with two constant values.
Throttle	PUB	Position()		Number(2)	Read value.
Throttle	PRI	Position(x)	Number(2)	Boolean	Set value, returns true if successful.
Throttle	PRI	GasPedal()		Number(2)	Read value.
Throttle	PUB	GasPedal(x)	Number(2)	Boolean	Set value, returns true if successful.
Throttle	PUB	Floor()		Number(2)	Read value.
Throttle	PUB	Floor(x)	Number(2)	Boolean	Set value, returns true if successful.
Throttle	PRI	Convert(x)	Number(2)	Number(3,2)	Converts Position to GasFlow.
Ignition	EXT	Ignition()		Ignition	Creates a new instance of the Ignition ADT
Ignition	EXT	Key()		Enum(On,Off)	Always reutrns On when object is active.
Ignition	EXT	Key(x)	Enum(On,Off)	Boolean	Can only turn Ignition Off - On creates the object
Ignition	EXT	EngineOn()		Boolean	Reports if engine has been started.
Ignition	PRI	EngineOn(x)	Boolean	Boolean	Sets value privately
Ignition	EXT	StartEngine()		Boolean	Returns true if successful.
Transmission	PRO	Transmission()		Transmission	Creates a new Transmission instance with several constants
Transmission	EXT	Gear()		Enum(N,R,1,2,3,4,5)	
Transmission	EXT	Gear(x)	Enum(N,R,1,2,3,4,5)	Boolean	Returns true if successful
Transmission	PUB	DriveRatio()		Number	Returns multiplier for Engine RPM to Wheel RPM
Wheel	PRO	Wheel()		Wheel	Creates a new instance of Wheel ADT
Wheel	PRI	AxelRpm()		Number(4)	Reads value
Wheel	PUB	AxelRpm(x)	Number(4)	Boolean	Return true if successful
Wheel	PRI	WheelRpm()		Number(4)	Reads value.
Wheel	PRI	WheelRpm(x)	Number(4)	Boolean	Private function never called externally.
Wheel	PRI	WheelDiam()	Number	Boolean	constant function never called externally.

ClassAlias	Avail	FunName	InputType	ReturnType	Description
Wheel	PRI	CheckState()		Boolean	Checks all state variables, decides next action.

Appendix III: Parameters for CruiseControl Functions

ClassAlias	FunName	Name	Type	Constraint	Direction
Throttle	Throttle(x,y)	x	Number(2)	(x>=0 & x<=99)	IN
Throttle	Throttle(x,y)	y	Number(2)	(y>=0 & y<=99)	IN
Gauges	Odometer(x)	x	Number(7,1)	(x>=0 & x<=99)	IN
Engine	Rpm(x)	x	Number(4)	(x>=0 & x<=99)	IN
AutoSystem	BrakeActive(x)	x	Boolean	(x=true OR x=false)	IN
CruiseUser	Switch(x)	x	Enum(On,Off)		IN
CruiseUnit	UserSwitch(x)	x	Enum(On,Off)		IN
BrakeUser	IsActive(x)	x	Boolean	(x=true OR x=false)	IN
BrakeControl	IsActive(x)	x	Boolean	(x=true OR x=false)	IN
ClutchUser	PedalPosition(x)	x	Number(2)	(x>=0 & x<=99)	IN
GasUser	PedalPosition(x)	x	Number(2)	(x>=0 & x<=99)	IN
Throttle	Position(x)	x	Number(2)	(x>=0 & x<=99)	IN
Ignition	Key(x)	x	Enum(On,Off)		IN
Transmission	Gear(x)	x	Enum(N,R,1,2,3,4,5)		IN
Wheel	AxelRpm(x)	x	Number(4)	(x>=0 & x<=9999)	IN
Gauges	TripMeter(x)	x	Number(5,1)	(x>=0 & x<=9999.9)	IN
AutoSystem	ClutchActive(x)	x	Boolean	(x=true OR x=false)	IN
CruiseUser	Mode(x)	x	Enum(NT,RA,SD)		IN
CruiseUnit	UserMode(x)	x	Enum(NT,RA,SD)		IN
BrakeUser	PedalPressure(x)	x	Number(2)	(x>=0 & x<=99)	IN
BrakeControl	WheelsTurning(x)	x	Number(2)	(x>=0 & x<=99)	IN
Ignition	EngineOn(x)	x	Boolean	(x=true OR x=false)	IN
Wheel	WheelRpm(x)	x	Number(4)	(x>=0 & x<=9999)	IN
Gauges	Tach(x)	x	Number(4)	(x>=0 & x<=9999)	IN
CruiseUnit	CurrentSpeed(x)	x	Number(4,1)	(x>=0 & x<=999.9)	IN
BrakeControl	LinePressure(x)	x	Number(2)	(x>=0 & x<=99)	IN
Throttle	GasPedal(x)	x	Number(2)	(x>=0 & x<=99)	IN
AutoSystem	Danger(x)	x	Boolean	(x=true OR x=false)	IN
Gauges	Speed(x)	x	Number(3)	(x>=0 & x<=999)	IN
CruiseUnit	TargetSpeed(x)	x	Number(4,1)	(x>=0 & x<=999.9)	IN
BrakeControl	PedalPressure(x)	x	Number(2)	(x>=0 & x<=99)	IN
Throttle	Floor(x)	x	Number(2)	(x>=0 & x<=99)	IN
Throttle	Convert(x)	x	Number(2)	(x>=0 & x<=99)	IN
Gauges	OilPressure(x)	x	Number		IN
CruiseUnit	TargetThrottle(x)	x	Number(2)	(x>=0 & x<=99)	IN
Gauges	WaterTemp(x)	x	Number		IN

ClassAlias	FunName	Name	Type	Constraint	Direction
Engine	GasFlow(x)	x	Number(3,2)	(x>=0 & x<=9.99)	IN
Gauges	Cruise(x)	x	Enum(On,Off)		IN
Engine	ExternalDrag(x)	x	Number(1)	(x>=-9 & x<=9)	IN
Gauges	AbsLight(x)	x	Enum(On,Off)		IN
Gauges	Battery(x)	x	Enum(On,Off)		IN
Gauges	OilLight(x)	x	Enum(On,Off)		IN
Gauges	SeatBelt(x)	x	Enum(On,Off)		IN
Gauges	HandBrake(x)	x	Enum(On,Off)		IN
Gauges	LowGas(x)	x	Enum(On,Off)		IN

Appendix IV: Object States for CruiseControl

ClassAlias	StateId	StateName	DefnPredicate
AutoSystem	s00	Initial	Undefined
AutoSystem	s01	Inactive	BrakeActive=false & ClutchActive=false & Danger=false
AutoSystem	s02	Active	BrakeActive=true OR ClutchActive=true OR Danger=true
BrakeControl	s00	Initial	Undefined
BrakeControl	s01	Inactive	IsActive=false
BrakeControl	s02	Braking	IsActive=true & WheelsTurning=true
BrakeControl	s03	Locked	IsActive=true & WheelsTurning=false
BrakeUser	s00	Initial	Undefined
BrakeUser	s01	Inactive	IsActive=false
BrakeUser	s02	Braking	IsActive=true
ClutchUser	s00	Initial	Undefined
ClutchUser	s01	Inactive	PedalPosition=0
ClutchUser	s02	Transition	PedalPosition>0 & PedalPosition<pconst
ClutchUser	s03	Engaged	PedalPosition>=pconst
CruiseUnit	s00	Initial	Undefined
CruiseUnit	s01	Off	UserSwitch=Off
CruiseUnit	s02	Inactive	UserSwitch=On & Gauges.Cruise()=Off & TargetSpeed=0 & UserMode=Null
CruiseUnit	s03	Cruise	UserSwitch=On & UserMode=NT & Gauges.Cruise()=On & SlowCutoff<TargetSpeed<FastCutoff
CruiseUnit	s04	Accel	UserSwitch=On & UserMode=RA & Gauges.Cruise()=On
CruiseUnit	s05	Decel	UserSwitch=On & UserMode=SD & Gauges.Cruise()=On
CruiseUnit	s06	Override	UserSwitch=On & Gauges.Cruise()=Off & SlowCutoff<TargetSpeed<FastCutoff
CruiseUser	s00	Initial	Undefined
CruiseUser	s01	Off	Switch=Off
CruiseUser	s02	Neutral	Switch=On & Mode=NT
CruiseUser	s03	Accel	Switch=On & Mode=RA
CruiseUser	s04	Decel	Switch=On & Mode=SD
Engine	s00	Initial	Undefined
Engine	s02	Normal	true
GasUser	s00	Initial	Undefined
GasUser	s01	Active	PedalPosition>=0
Gauges	s00	Initial	Undefined
Gauges	s01	Normal	Speed<180 & OilLight=Off & WaterTemp<100
Gauges	s02	Danger	Speed>=180 OR OilLight=On OR WaterTemp>=100
Throttle	s00	Initial	Undefined
Throttle	s01	Idle	Position=fconst
Throttle	s02	Manual	fconst<Position<=gconst & Position>Floor
Throttle	s03	Automatic	fconst<Position<=gconst & Position=Floor
Throttle	s04	Danger	GasPedal>gconst

ClassAlias	StateId	StateName	DefnPredicate
Ignition	s00	Initial	Undefined
Ignition	s01	On	Key=On
Transmission	s00	Initial	Undefined
Transmission	s01	Neutral	Gear=N
Transmission	s02	Reverse	Gear=R
Transmission	s03	Forward	Gear=1 OR Gear=2 OR Gear=3 OR Gear=4 OR Gear=5
Wheel	s00	Initial	Undefined
Wheel	s01	DirectDrive	AxelRpm=WheelRpm
Wheel	s02	Decel	AxelRpm<WheelRpm
Wheel	s03	Accel	WheelRpm<AxelRpm

Appendix V: State Variables for CruiseControl

ClassAlias	VariableName	DataType	Default	Constraint	Description
AutoSystem	BrakeActive	Boolean	null		
AutoSystem	ClutchActive	Boolean	null		
AutoSystem	Danger	Boolean	null		
AutoSystem	ThrottleFloor	Number(2)	12	CONSTANT	Will determine fconst when Throttle object is created.
AutoSystem	ThrottleGovernor	Number(2)	80	CONSTANT	Will determine gconst when Throttle objects is created.
BrakeControl	IsActive	Boolean	null		
BrakeControl	WheelsTurning	Boolean	null		
BrakeControl	LinePressure	Number(2)	0		
BrakeControl	PedalPressure	Number(2)	0		
BrakeUser	IsActive	Boolean	null		
BrakeUser	PedalPressure	Number(2)	0	0<=PedalPressure<100	
BrakeUser	pconst	Number(2)	5	CONSTANT	
ClutchUser	PedalPosition	Number(2)	0	0<=PedalPosition<100	
ClutchUser	pconst	Number(2)	5	CONSTANT	
CruiseUnit	UserSwitch	Enum(On,Off)	Off		
CruiseUnit	UserMode	Enum(Null,NT,RA,SD)	NT		
CruiseUnit	CurrentSpeed	Number(4,1)	0	0<=CurrentSpeed<200	
CruiseUnit	TargetSpeed	Number(4,1)	0		
CruiseUnit	TargetThrottle	Number(2)	0	0<=TargetThrottle<99	
CruiseUnit	SlowCutoff	Number(4,1)	25	CONSTANT	
CruiseUnit	FastCutoff	Number(4,1)	95	CONSTANT	
CruiseUser	Switch	Enum(On,Off)	Off		
CruiseUser	Mode	Enum(NT,RA,SD)	NT		
Engine	Rpm	Number(4)	0	0<=Rpm<=8000	
Engine	GasFlow	Real	0	0<=GasFlow<10	
Engine	ExternalDrag	Real	1	0<ExternalDrag<2	Used to simulate hills and wind resistance
Engine	WaterTMin	Number(3)	15	CONSTANT	
Engine	OilPMin	Number(2)	8	CONSTANT	
GasUser	PedalPosition	Number(2)	0	0<=PedalPosition<100	
Gauges	Odometer	Number(7,1)	previous value	Odometer>=0	
Gauges	TripMeter	Number(5,1)	previous value	TripMeter>=0	
Gauges	Tach	Number(4)	0	0<=Tach<=8000	
Gauges	Speed	Number(3)	0	0<=Speed<=220	Measured in km/hr

ClassAlias	VariableName	DataType	Default	Constraint	Description
Gauges	OilPressure	Number	null		
Gauges	WaterTemp	Number	null		
Gauges	Cruise	Enum(On,Off)	Off		
Gauges	AbsLight	Enum(On,Off)	Off		
Gauges	Battery	Enum(On,Off)	Off		
Gauges	OilLight	Enum(On,Off)	Off		
Gauges	SeatBelt	Enum(On,Off)	Off		
Gauges	HandBrake	Enum(On,Off)	null		
Gauges	LowGas	Enum(On,Off)	Off		
Throttle	Position	Number(2)	fconst	0<=Position<100	
Throttle	Floor	Number(2)	fconst	0<=Floor<100	
Throttle	GasPedal	Number(2)	fconst	0<=GasPedal<100	
Throttle	fconst	Number(2)	null	CONSTANT	Default = AutoSystem.ThrottleFloor()
Throttle	gconst	Number(2)	null	CONSTANT	Default = AutoSystem.ThrottleGovernor()
Ignition	Key	Enum(On,Off)	On		
Ignition	EngineOn	Boolean	false		
Transmission	Gear	Enum(N,R,1,2,3,4,5)	N		
Transmission	Ratio_R	Number	1.846	CONSTANT	
Transmission	Ratio_1	Number	2.563	CONSTANT	
Transmission	Ratio_2	Number	1.552	CONSTANT	
Transmission	Ratio_3	Number	1.022	CONSTANT	
Transmission	Ratio_4	Number	0.653	CONSTANT	
Transmission	Ratio_5	Number	0.471	CONSTANT	
Transmission	Ratio_Diff	Number	4.429	CONSTANT	
Wheel	AxelRpm	Number	0	0<=Rpm<=8000	
Wheel	WheelRpm	Number	0	0<=Rpm<=8000	
Wheel	WheelDiam	Number	0.00056	CONSTANT	Measured in Kilometers (56cm)

Appendix VI: Mutator Transitions for CruiseControl

ClassAlias	Source State	Target State	Function Name	Guard	Action
AutoSystem	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();
AutoSystem	Inactive	Active	BrakeActive(x)	x=true	BrakeActive:=true; Call CruiseUnit.Cancel();
AutoSystem	Inactive	Active	Danger(x)	x=true	Danger:=true; Call CruiseUnit.Cancel();
AutoSystem	Inactive	Active	ClutchActive(x)	x=true	ClutchActive:=true; Call CruiseUnit.Cancel();
AutoSystem	Active	Active	BrakeActive(x)	x=true	BrakeActive:=true; Call CruiseUnit.Cancel();
AutoSystem	Active	Active	ClutchActive(x)	x=true	ClutchActive:=true; Call CruiseUnit.Cancel();
AutoSystem	Active	Active	Danger(x)	x=true	Danger:=true; Call CruiseUnit.Cancel();
AutoSystem	Active	Inactive	BrakeActive(x)	x=false & ClutchActive=false & Danger=false	BrakeActive:=false;
AutoSystem	Active	Inactive	ClutchActive(x)	x=false & BrakeActive=false & Danger=false	ClutchActive:=false;
AutoSystem	Active	Inactive	Danger(x)	x=false & BrakeActive=false	Danger:=false;
AutoSystem	Inactive	Inactive	ThrottleFloor()	true	Return ThrottleFloor;
AutoSystem	Inactive	Inactive	ThrottleGovernor()	true	Return ThrottleGovernor;
AutoSystem	Active	Active	ThrottleFloor()	true	Return ThrottleFloor;
AutoSystem	Active	Active	ThrottleGovernor()	true	Return ThrottleGovernor;
AutoSystem	Inactive	Inactive	Danger()	true	Return Danger;
AutoSystem	Active	Active	Danger()	true	Return Danger;
AutoSystem	Inactive	Inactive	BrakeActive()	true	Return BrakeActive;
AutoSystem	Active	Active	BrakeActive()	true	Return BrakeActive;
AutoSystem	Inactive	Inactive	ClutchActive()	true	Return ClutchActive;
AutoSystem	Active	Active	ClutchActive()	true	Return ClutchActive;
BrakeControl	Initial	Inactive	BrakeControl()	true	Global BrakeUser:=New BrakeUser(); IsActive:=false; PedalPressure:=0; LinePressure:=0; WheelsTurning:=false;
BrakeControl	Inactive	Braking	IsActive(x)	x=true	IsActive:=true; Call AutoSystem.BrakeActive(true);
BrakeControl	Braking	Inactive	IsActive(x)	x=false	IsActive:=false; Call AutoSystem.BrakeActive(false);
BrakeControl	Locked	Inactive	IsActive(x)	x=false	IsActive:=false; Call AutoSystem.BrakeActive(false);
BrakeControl	Braking	Braking	PedalPressure(x)	x>PedalPressure & WheelsTurning=true	PedalPressure:=x; LinePressure:=(1.1)*LinePressure; WheelsTurning:=Sensor.Turning();
BrakeControl	Braking	Braking	PedalPressure(x)	x<PedalPressure & WheelsTurning=true	PedalPressure:=x; LinePressure:=(0.9)*LinePressure; WheelsTurning:=Sensor.Turning();

ClassAlias	Source State	Target State	Function Name	Guard	Action
BrakeControl	Braking	Locked	PedalPressure(x)	WheelsTurning=false	PedalPressure:=x; LinePressure:=(0.9)*LinePressure; WheelsTurning:=Sensor.Turning();
BrakeControl	Locked	Locked	PedalPressure(x)	WheelsTurning=false OR x>PedalPressure	LinePressure:=(0.9)*LinePressure; WheelsTurning:=Sensor.Turning();
BrakeControl	Locked	Braking	PedalPressure(x)	WheelsTurning=true& x<PedalPressure	LinePressure:=(0.9)*LinePressure; WheelsTurning:=Sensor.Turning();
BrakeUser	Initial	Inactive	BrakeUser()	true	IsActive:=false; PedalPressure:=0; pconst:=5;
BrakeUser	Inactive	Braking	IsActive(x)	x=true	IsActive:=true; Call AutoSystem.BrakeActive(true); Call BrakeControl.IsActive(true);
BrakeUser	Braking	Inactive	IsActive(x)	x=false	IsActive:=false; Call AutoSystem.BrakeActive(false); Call BrakeControl.IsActive(false);
BrakeUser	Braking	Braking	PedalPressure(x)	x<pconst	PedalPressure:=x;
BrakeUser	Braking	Braking	PedalPressure(x)	x>=pconst & x<>PedalPressure	PedalPressure:=x; Call BrakeControl.PedalPressure(PedalPressure);
ClutchUser	Initial	Inactive	ClutchUser()	true	pconst:=5; PedalPosition:=pconst;
ClutchUser	Inactive	Transition	PedalPosition(x)	x>0	PedalPosition:=x; Call AutoSystem.ClutchActive(true);
ClutchUser	Transition	Inactive	PedalPosition(x)	x=0	PedalPosition:=x; Call AutoSystem.ClutchActive(false);
ClutchUser	Transition	Engaged	PedalPosition(x)	x>pconst	PedalPosition:=x; Call ClutchUnit.PedalDown(true);
ClutchUser	Engaged	Transition	PedalPosition(x)	x<=pconst	PedalPosition:=x; CallClutchUnit.PedalDown(false);
CruiseUnit	Initial	Off	CruiseUnit()	true	Global CruiseUser:=New CruiseUser(); UserSwitch:=Off; SlowCutoff:=25; FastCutoff:=95; UserMode:=Null; CurrentSpeed:=0; TargetSpeed:=0; TargetThrottle:=0;
CruiseUnit	Off	Off	Cancel()	true	
CruiseUnit	Off	Off	CheckState()	true	
CruiseUnit	Off	Off	SetSpeed()	true	CurrentSpeed:=Gauges.Speed();
CruiseUnit	Off	Off	UserMode()	true	Return UserMode;
CruiseUnit	Off	Off	UserMode(x)	true	
CruiseUnit	Off	Off	UserSwitch()	true	Return UserSwitch;
CruiseUnit	Off	Off	UserSwitch(x)	x=Off	
CruiseUnit	Off	Inactive	UserSwitch(x)	x=On	UserSwitch:=On;
CruiseUnit	Inactive	Inactive	Cancel()	true	
CruiseUnit	Inactive	Inactive	CheckState()	true	
CruiseUnit	Inactive	Inactive	SetSpeed()	true	CurrentSpeed:=Gauges.Speed();

ClassAlias	Source State	Target State	Function Name	Guard	Action
CruiseUnit	Inactive	Inactive	UserMode()	true	Return UserMode;
CruiseUnit	Inactive	Inactive	UserMode(x)	x=NT & UserMode=SD & (Gauges.Speed()<=SlowCutoff OR Gauges.Speed()>=FastCutoff)	UserMode:=NT;
CruiseUnit	Inactive	Inactive	UserMode(x)	x=NT & UserMode<>SD	
CruiseUnit	Inactive	Cruise	UserMode(x)	x=NT & UserMode=SD & (SlowCutoff<Gauges.Speed()<FastCutoff) & AutoSystem.BrakeActive()=false & AutoSystem.ClutchActive()=false	UserMode:=NT; CurrentSpeed:=Gauges.Speed(); TargetSpeed:=CurrentSpeed; TargetThrottle:=Throttle.Position(); Call Gauges.Cruise(On); Call Throttle.Floor(TargetThrottle); Put CheckState() on Call Queue;
CruiseUnit	Inactive	Inactive	UserMode(x)	x<>NT	UserMode:=x;
CruiseUnit	Inactive	Inactive	UserSwitch()	true	Return UserSwitch;
CruiseUnit	Inactive	Off	UserSwitch(x)	x=Off	UserSwitch:=Off;
CruiseUnit	Inactive	Inactive	UserSwitch(x)	x=On	UserSwitch:=On;
CruiseUnit	Cruise	Override	Cancel()	true	Call Gauges.Cruise(Off); Call Throttle.Floor(0);
CruiseUnit	Cruise	Cruise	CheckState()	UserMode<>NT	
CruiseUnit	Cruise	Cruise	CheckState()	ABS(TargetSpeed-CurrentSpeed)<0.5	Pause; CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;
CruiseUnit	Cruise	Cruise	CheckState()	0.5<=ABS(TargetSpeed-CurrentSpeed)<1.0	CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;
CruiseUnit	Cruise	Cruise	CheckState()	ABS(TargetSpeed-CurrentSpeed)>=1.0 & Throttle.Position()>Throttle.Floor()	CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;
CruiseUnit	Cruise	Cruise	CheckState()	CurrentSpeed-TargetSpeed>=1.0 & Throttle.Position()=Throttle.Floor()	Call Throttle.Floor(Throttle.Floor()-0.5); Pause; CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;
CruiseUnit	Cruise	Cruise	CheckState()	TargetSpeed-CurrentSpeed>=1.0 & Throttle.Position()=Throttle.Floor()	Call Throttle.Floor(Throttle.Floor()+0.5); Pause; CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;
CruiseUnit	Cruise	Cruise	SetSpeed()	true	CurrentSpeed:=Gauges.Speed();
CruiseUnit	Cruise	Cruise	UserMode()	true	Return UserMode;

ClassAlias	Source State	Target State	Function Name	Guard	Action
CruiseUnit	Cruise	Decel	UserMode(x)	x=SD	TargetSpeed:=TargetSpeed-1; UserMode:=SD; Put CheckState() on Call Queue;
CruiseUnit	Cruise	Accel	UserMode(x)	x=RA	TargetSpeed:=TargetSpeed+1; UserMode:=RA; Put CheckState() on Call Queue;
CruiseUnit	Cruise	Cruise	UserSwitch()	true	Return UserSwitch;
CruiseUnit	Cruise	Cruise	UserSwitch(x)	x=On	
CruiseUnit	Cruise	Off	UserSwitch(x)	x=Off	Call Gauges.Cruise(Off); UserSwitch:=Off; UserMode:=Null; Call Throttle.Floor(0);
CruiseUnit	Decel	Override	Cancel()	true	Call Gauges.Cruise(Off); UserMode:=Null; Call Throttle.Floor(0);
CruiseUnit	Decel	Override	Cancel()	UserMode<>SD	
CruiseUnit	Decel	Decel	CheckState()	UserMode<>SD	
CruiseUnit	Decel	Decel	CheckState()	CurrentSpeed>SlowCutoff	Call Throttle.Floor(Throttle.Position()-0.5); Pause; CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;
CruiseUnit	Decel	Override	CheckState()	CurrentSpeed<=SlowCutoff	Call Gauges.Cruise(Off); UserMode:=Null; Call Throttle.Floor(0);
CruiseUnit	Decel	Decel	SetSpeed()	true	CurrentSpeed:=Gauges.Speed();
CruiseUnit	Decel	Decel	UserMode()	true	Return UserMode;
CruiseUnit	Decel	Override	UserMode(x)	x=RA	Call Gauges.Cruise(Off); UserMode:=Null; Call Throttle.Floor(0);
CruiseUnit	Decel	Cruise	UserMode(x)	x=NT	UserMode:=NT; TargetSpeed:=Gauges.Speed(); TargetThrottle:=Throttle.Position(); CurrentSpeed:=TargetSpeed; Put CheckState() on Call Queue;
CruiseUnit	Decel	Decel	UserSwitch()	true	Return UserSwitch;
CruiseUnit	Decel	Off	UserSwitch(x)	x=Off	Call Gauges.Cruise(Off); UserSwitch:=Off; UserMode:=Null; Call Throttle.Floor(0);
CruiseUnit	Decel	Decel	UserSwitch(x)	x=On	
CruiseUnit	Accel	Override	Cancel()	true	Call Gauges.Cruise(Off); UserMode:=Null; Call Throttle.Floor(0);
CruiseUnit	Accel	Override	Cancel()	UserMode<>RA	
CruiseUnit	Accel	Accel	CheckState()	UserMode<>RA	
CruiseUnit	Accel	Accel	CheckState()	CurrentSpeed<FastCutoff	Call Throttle.Floor(Throttle.Position()+0.5); Pause; CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;
CruiseUnit	Accel	Override	CheckState()	CurrentSpeed>=FastCutoff	Call Gauges.Cruise(Off); UserMode:=Null; Call Throttle.Floor(0);
CruiseUnit	Accel	Accel	SetSpeed()	true	CurrentSpeed:=Gauges.Speed();
CruiseUnit	Accel	Accel	UserMode()	true	Return UserMode;
CruiseUnit	Accel	Override	UserMode(x)	x=SD	Call Gauges.Cruise(Off); UserMode:=Null; Call Throttle.Floor(0);
CruiseUnit	Accel	Cruise	UserMode(x)	x=NT	UserMode:=NT; TargetSpeed:=Gauges.Speed(); TargetThrottle:=Throttle.Position(); CurrentSpeed:=TargetSpeed; Put CheckState() on Call Queue;
CruiseUnit	Accel	Accel	UserSwitch()	true	Return UserSwitch;

ClassAlias	Source State	Target State	Function Name	Guard	Action
CruiseUnit	Accel	Off	UserSwitch(x)	x=Off	Call Gauges.Cruise(Off); UserSwitch:=Off; UserMode:=Null; Call Throttle.Floor(0);
CruiseUnit	Accel	Accel	UserSwitch(x)	x=On	
CruiseUnit	Override	Override	Cancel()	true	
CruiseUnit	Override	Override	CheckState()	true	
CruiseUnit	Override	Override	SetSpeed()	true	CurrentSpeed:=Gauges.Speed();
CruiseUnit	Override	Override	UserMode()	true	Return UserMode;
CruiseUnit	Override	Override	UserMode(x)	x<>NT OR Gauges.Speed()<=SlowCutoff OR Gauges.Speed()>=FastCutoff	UserMode:=x;
CruiseUnit	Override	Cruise	UserMode(x)	x=NT & UserMode=SD & (SlowCutoff<Gauges.Speed()<FastCutoff) & AutoSystem.BrakeActive()=false & AutoSystem.ClutchActive()=false	CurrentSpeed:=Gauges.Speed(); TargetSpeed:=CurrentSpeed; TargetThrottle:=Throttle.Position(); Call Gauges.Cruise(On); Call Throttle.Floor(TargetThrottle); UserMode:=NT; Put CheckState() on Call Queue;
CruiseUnit	Override	Cruise	UserMode(x)	x=NT & UserMode=RA & (SlowCutoff<Gauges.Speed()<FastCutoff) & AutoSystem.BrakeActive()=false & AutoSystem.ClutchActive()=false	Call Throttle.Floor(TargetThrottle); Call Gauges.Cruise(On); UserMode:=NT; Pause; CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;
CruiseUnit	Override	Override	UserMode(x)	x=NT & UserMode=Null	UserMode:=NT;
CruiseUnit	Override	Override	UserMode(x)	x=NT & UserMode=NT	UserMode:=NT;
CruiseUnit	Override	Override	UserSwitch()	true	Return UserSwitch;
CruiseUnit	Override	Off	UserSwitch(x)	x=Off	UserSwitch:=Off; UserMode:=Null;
CruiseUnit	Override	Override	UserSwitch(x)	x=On	
CruiseUser	Initial	Off	CruiseUser()	true	Switch:=Off; Mode:=NT;
CruiseUser	Off	Neutral	Switch(x)	x=On	Switch:=On; Call CruiseUnit.UserSwitch(On);
CruiseUser	Neutral	Off	Switch(x)	x=Off	Switch:=Off; Call CruiseUnit.UserSwitch(Off);
CruiseUser	Neutral	Accel	Mode(x)	x=RA	Mode:=RA; Call CruiseUnit.UserMode(RA);

ClassAlias	Source State	Target State	Function Name	Guard	Action
CruiseUser	Accel	Neutral	Mode(x)	x=NT	Mode:=NT; Call CruiseUnit.UserMode(NT);
CruiseUser	Decel	Neutral	Mode(x)	x=NT	Mode:=NT; Call CruiseUnit.UserMode(NT);
CruiseUser	Neutral	Decel	Mode(x)	x=SD	Mode:=SD; Call CruiseUnit.UserMode(SD);
CruiseUser	Accel	Off	Switch(x)	x=Off	Switch:=Off; Mode:=NT; Call CruiseUnit.UserSwitch(Off);
CruiseUser	Decel	Off	Switch(x)	x=Off	Switch:=Off; Mode:=NT; Call CruiseUnit.UserSwitch(Off);
CruiseUser	Neutral	Neutral	Cancel()	true	Call CruiseUnit.Cancel();
CruiseUser	Off	Off	Cancel()	true	
CruiseUser	Off	Off	Switch(x)	x=Off	
CruiseUser	Off	Off	Mode(x)	true	
CruiseUser	Neutral	Neutral	Mode(x)	x=NT	Call CruiseUnit.UserMode(NT);
CruiseUser	Neutral	Neutral	Switch(x)	x=On	
CruiseUser	Accel	Accel	Cancel()	true	Call CruiseUnit.Cancel();
CruiseUser	Accel	Accel	Switch(x)	x=On	
CruiseUser	Accel	Accel	Mode(x)	x<>NT	
CruiseUser	Decel	Decel	Cancel()	true	Call CruiseUnit.Cancel();
CruiseUser	Decel	Decel	Switch(x)	x=On	
CruiseUser	Decel	Decel	Mode(x)	x<>NT	
Engine	Initial	Normal	Engine()	true	Rpm:=0; GasFlow:=0; ExternalDrag:=1; WaterTMin:=0; OilPMin:=0;
Engine	Normal	Normal	GasFlow()	true	Return GasFlow;
Engine	Normal	Normal	GasFlow(x)	true	GasFlow:=x; Rpm:=(2-ExternalDrag)*GasFlow*630; Call Gauges.Tach(Rpm); Call Wheel.AxelRpm(Rpm*Transmission.DriveRatio());
Engine	Normal	Normal	ExternalDrag()	true	Return ExternalDrag;
Engine	Normal	Normal	ExternalDrag(x)	true	ExternalDrag:=x; Rpm:=(2-ExternalDrag)*GasFlow*630; Call Gauges.Tach(Rpm); Call Wheel.AxelRpm(Rpm*Transmission.DriveRatio());
GasUser	Initial	Active	GasUser()	true	PedalPosition:=0;
GasUser	Active	Active	PedalPosition(x)	x>0 & x<>PedalPosition	PedalPosition:=x; Call Throttle.GasPedal(PedalPosition);
GasUser	Active	Active	PedalPosition()	true	Return PedalPosition;
Gauges	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;

ClassAlias	Source State	Targe tState	Function Name	Guard	Action
Gauges	Normal	Normal	Odometer()	true	Return Odometer;
Gauges	Normal	Normal	Odometer(x)	true	Odometer:=x;
Gauges	Normal	Normal	TripMeter()	true	Return TripMeter;
Gauges	Normal	Normal	TripMeter(x)	true	TripMeter:=x;
Gauges	Normal	Normal	Tach()	true	Return Tach;
Gauges	Normal	Normal	Tach(x)	true	Tach:=x;
Gauges	Normal	Normal	Speed()	true	Return Speed;
Gauges	Normal	Normal	Speed(x)	x<180	Speed:=x;
Gauges	Normal	Danger	Speed(x)	x>=180	Speed:=Min(x,250); Call AutoSystem.Danger(true);
Gauges	Normal	Normal	OilPressure()	true	Return OilPressure;
Gauges	Normal	Danger	OilPressure(x)	x>=57;	OilPressure:=x; OilLight:=On; Call AutoSystem.Danger(true);
Gauges	Normal	Normal	OilPressure(x)	x<57	OilPressure:=x;
Gauges	Normal	Normal	WaterTemp()	true	Return WaterTemp;
Gauges	Normal	Danger	WaterTemp(x)	x>=100	WaterTemp:=x; Call AutoSystem.Danger(true);
Gauges	Normal	Normal	WaterTemp(x)	x<100	WaterTemp:=x;
Gauges	Normal	Normal	Cruise()	true	Return Cruise;
Gauges	Normal	Normal	Cruise(x)	true	Cruise:=x;
Gauges	Normal	Normal	AbsLight()	true	Return AbsLight;
Gauges	Normal	Normal	AbsLight(x)	true	AbsLight:=x;
Gauges	Normal	Normal	Battery()	true	Return Battery;
Gauges	Normal	Normal	Battery(x)	true	Battery:=x;
Gauges	Normal	Normal	OilLight()	true	Return OilLight;
Gauges	Normal	Normal	OilLight(x)	true	Private method!
Gauges	Normal	Normal	SeatBelt()	true	Return SeatBelt;
Gauges	Normal	Normal	SeatBelt(x)	true	SeatBelt:=x;
Gauges	Normal	Normal	HandBrake()	true	Return HandBrake;
Gauges	Normal	Normal	HandBrake(x)	true	HandBrake:=x;
Gauges	Normal	Normal	LowGas()	true	Return LowGas;
Gauges	Normal	Normal	LowGas(x)	true	LowGas:=x;
Gauges	Danger	Danger	Odometer(x)	true	Odometer:=x;
Gauges	Danger	Danger	TripMeter(x)	true	TripMeter:=x;
Gauges	Danger	Danger	Tach(x)	true	Tach:=x;
Gauges	Danger	Danger	Speed(x)	x>=180	Speed:=Min(x,250);
Gauges	Danger	Normal	Speed(x)	x<180 & OilLight=Off & WaterTemp<100	Speed:=x; Call AutoSystem.Danger(false);
Gauges	Danger	Danger	OilPressure(x)	x>=57	OilPressure:=x;
Gauges	Danger	Normal	OilPressure(x)	x<57 & Speed<180 & WaterTemp<100	OilPressure:=x; OilLight:=Off; Call AutoSystem.Danger(false);
Gauges	Danger	Danger	WaterTemp(x)	x>=100	WaterTemp:=x;
Gauges	Danger	Danger	WaterTemp(x)	x<100 & OilLight=Off	WaterTemp:=x; Call AutoSystem.Danger(false);

ClassAlias	Source State	Target State	Function Name	Guard	Action
				& Speed<180	
Gauges	Danger	Danger	Cruise(x)	true	Cruise:=x;
Gauges	Danger	Danger	AbsLight(x)	true	AbsLight:=x;
Gauges	Danger	Danger	Battery(x)	true	Battery:=x;
Gauges	Danger	Danger	SeatBelt(x)	true	SeatBelt:=x;
Gauges	Danger	Danger	HandBrake(x)	true	HandBrake:=x;
Gauges	Danger	Danger	LowGas(x)	true	LowGas:=x;
Gauges	Danger	Danger	Cruise()	true	Return Cruise;
Gauges	Danger	Danger	AbsLight()	true	Return AbsLight;
Gauges	Danger	Danger	Tach()	true	Return Tach;
Gauges	Danger	Danger	Speed()	true	Return Speed;
Gauges	Danger	Danger	WaterTemp()	true	Return WaterTemp;
Gauges	Danger	Danger	OilPressure()	true	Return OilPressure;
Throttle	Initial	Idle	Throttle(x,y)	x=0 OR x>=y OR y=100	Configuration Error! Do not consider.
Throttle	Initial	Idle	Throttle(x,y)	0<x & x<y & y<100	fconst:=x; gconst:=y; Position:=fconst; Call Engine.GasFlow(Convert(Position)); Floor:=fconst; Call GasUser.PedalPosition(fconst);
Throttle	Idle	Manual	GasPedal(x)	x>fconst	GasPedal:=x; Position:=Min(GasPedal,gconst); Call Engine.GasFlow(Convert(Position));
Throttle	Manual	Idle	GasPedal(x)	x<=fconst	GasPedal:=x; Position:=fconst; Call Engine.GasFlow(Convert(fconst)); Call GasUser.PedalPosition(fconst);
Throttle	Idle	Automatic	Floor(x)	x>fconst	Floor:=Min(x,gconst); Position:=Floor; Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(Position);
Throttle	Automatic	Idle	Floor(x)	x<=fconst	Floor:=fconst; Position:=fconst; Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(fconst);
Throttle	Manual	Automatic	GasPedal(x)	x>fconst & x<=Floor	GasPedal:=x; Position:=Floor; Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(Floor);
Throttle	Manual	Automatic	Floor(x)	x>=Position	Floor:=Min(x,gconst); Position:=Floor; Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(Floor);
Throttle	Automatic	Manual	GasPedal(x)	x>fconst & x>Floor & x<=gconst	GasPedal:=x; Position:=x; Call Engine.GasFlow(Convert(Position));
Throttle	Automatic	Automatic	Floor(x)	x>fconst	Floor:=Min(x,gconst); Position:=Floor; Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(Position);
Throttle	Manual	Danger	Position(x)	true	Position is Private!!
Throttle	Automatic	Danger	Position(x)	true	Position is Private!!
Throttle	Idle	Idle	Convert(x)	0<=x<100	Return x/10;
Throttle	Initial	Initial	Convert(x)	0<=x<100	Return x/10;
Throttle	Manual	Manual	Convert(x)	0<=x<100	Return x/10;
Throttle	Automatic	Automatic	Convert(x)	0<=x<100	Return x/10;
Throttle	Idle	Idle	Position()	true	Return Position;
Throttle	Manual	Manual	Position()	true	Return Position;

ClassAlias	Source State	Target State	Function Name	Guard	Action
Throttle	Automatic	Automatic	Position()	true	Return Position;
Throttle	Danger	Danger	Position()	true	Return Position;
Throttle	Idle	Idle	Floor()	true	Return Floor;
Throttle	Manual	Manual	Floor()	true	Return Floor;
Throttle	Automatic	Automatic	Floor()	true	Return Floor;
Throttle	Danger	Danger	Floor()	true	Return Floor;
Throttle	Idle	Idle	GasPedal(x)	x<=fconst	GasPedal:=x;
Throttle	Idle	Idle	Floor(x)	x<=fconst	Floor:=fconst;
Throttle	Manual	Manual	GasPedal(x)	x>fconst & x<=gconst & x>Floor	GasPedal:=x; Position:=x; Call Engine.GasFlow(Convert(Position));
Throttle	Manual	Danger	GasPedal(x)	x>gconst	GasPedal:=x; Position:=gconst; Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(gconst);
Throttle	Manual	Manual	Floor(x)	x<Position	Floor:=Max(fconst,x);
Throttle	Automatic	Danger	GasPedal(x)	x>gconst	GasPedal:=x; Position:=gconst; Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(gconst);
Throttle	Automatic	Automatic	GasPedal(x)	x>fconst & x<=Floor	GasPedal:=x;
Throttle	Danger	Automatic	GasPedal(x)	x<Floor	GasPedal:=x; Position:=Floor; Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(Position);
Throttle	Danger	Danger	GasPedal(x)	x>gconst	GasPedal:=x;
Throttle	Danger	Manual	GasPedal(x)	x>=Floor & x<=gconst	GasPedal:=x; Position:=x; Call Engine.GasFlow(Convert(Position)); Call GasUser.PedalPosition(Position);
Throttle	Danger	Danger	Floor(x)	true	Floor:=Max(fconst,Min(x,gconst));
Throttle	Automatic	Manual	Floor(x)	x>fconst & x<Floor	Left over from some earlier analysis? NOT exclusive with c10t009!
Throttle	Idle	Danger	GasPedal(x)	x>gconst	Not Feasible because GasPedal cannot "jump" this far!
Throttle	Danger	Idle	Position(x)	true	Position is Private!!
Ignition	Initial	On	Ignition()	true	Key:=On; EngineOn:=false; Global AutoSystem:=New AutoSystem();
Ignition	On	Initial	Key(x)	x=Off	Key:=Off; EngineOn:=false; Destroy Throttle; Destroy Engine; Destroy AutoSystem; Destroy Self;
Ignition	On	??	Key(x)	x=On	Can not turn Key On when already On!
Ignition	On	On	StartEngine()	EngineOn=false	Global Transmission:=New Transmission(); Global Engine:=New Engine(); Global GasUser:=New GasUser(); Global Throttle:=New Throttle(AutoSystem.ThrottleFloor(),AutoSystem.ThrottleGovernor()); EngineOn:=true;
Ignition	On	On	StartEngine()	EngineOn=true	GrindingNoise;
Ignition	On	On	EngineOn(x)	true	Private method -- variable can only be set internally to object.
Ignition	On	On	EngineOn()	true	Private method -- variable can only be read internally by object.
Ignition	On	On	Key()	true	Return Key;
Transmission	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();

ClassAlias	Source State	Target State	Function Name	Guard	Action
Transmission	Neutral	Neutral	Gear()	true	Return N;
Transmission	Neutral	Reverse	Gear(x)	x=R	Gear:=R;
Transmission	Neutral	Neutral	Gear(x)	x=N	
Transmission	Neutral	Forward	Gear(x)	x=1 OR x=2 OR x=3 OR x=4 OR x=5	Gear:=x; Call Wheel.AxelRpm(Gauges.Tach()*DriveRatio());
Transmission	Reverse	Neutral	Gear(x)	x=N	Gear:=N;
Transmission	Forward	Neutral	Gear(x)	x=N	Gear:=N; Call Wheel.AxelRpm(0);
Transmission	Forward	Forward	Gear(x)	x=1 OR x=2 OR x=3 OR x=4 OR x=5	Gear:=x; Call Wheel.AxelRpm(Gauges.Tach()*DriveRatio());
Transmission	Neutral	Neutral	DriveRatio()	true	Return 0;
Transmission	Reverse	Reverse	DriveRatio()	true	Return -1/(Ratio_R * Ratio_Diff);
Transmission	Forward	Forward	DriveRatio()	Gear=2	Return 1/(Ratio_2 * Ratio_Diff);
Transmission	Forward	Forward	DriveRatio()	Gear=3	Return 1/(Ratio_3 * Ratio_Diff);
Transmission	Forward	Forward	DriveRatio()	Gear=5	Return 1/(Ratio_5 * Ratio_Diff);
Transmission	Forward	Forward	DriveRatio()	Gear=4	Return 1/(Ratio_4 * Ratio_Diff);
Transmission	Forward	Forward	DriveRatio()	Gear=1	Return 1/(Ratio_1 * Ratio_Diff);
Transmission	Reverse	Reverse	Gear()	true	Return R;
Transmission	Forward	Forward	Gear()	true	Return Gear;
Wheel	Initial	DirectDrive	Wheel()	true	AxelRpm:=0; WheelRpm:=0; WheelDiam:=0.00056;
Wheel	DirectDrive	DirectDrive	AxelRpm(x)	ABS(x-WheelRpm)<=2	AxelRpm:=x; WheelRpm:=x; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam);
Wheel	DirectDrive	Decel	AxelRpm(x)	x+2<WheelRpm	AxelRpm:=x; WheelRpm:=WheelRpm-1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call queue;
Wheel	DirectDrive	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	Decel	Decel	AxelRpm(x)	x+2<WheelRpm	AxelRpm:=x; WheelRpm:=WheelRpm-1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call queue;
Wheel	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	Decel	DirectDrive	AxelRpm(x)	ABS(x-WheelRpm)<=2	AxelRpm:=x; WheelRpm:=x; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam);
Wheel	Accel	Decel	AxelRpm(x)	x+2<WheelRpm	AxelRpm:=x; WheelRpm:=WheelRpm-1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call queue;
Wheel	Accel	DirectDrive	AxelRpm(x)	ABS(x-WheelRpm)<=2	AxelRpm:=x; WheelRpm:=x; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam);
Wheel	Accel	Accel	AxelRpm(x)	x-2>WheelRpm	AxelRpm:=x; WheelRpm:=WheelRpm+1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call queue;

ClassAlias	Source State	Target State	Function Name	Guard	Action
Wheel	DirectDrive	DirectDrive	CheckState()	true	
Wheel	Decel	Decel	CheckState()	AxelRpm+2<WheelRpm	WheelRpm:=WheelRpm-1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call queue;
Wheel	Decel	DirectDrive	CheckState()	AxelRpm+2>=WheelRpm	WheelRpm:=AxelRpm; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam);
Wheel	Accel	Accel	CheckState()	AxelRpm-2>WheelRpm	WheelRpm:=WheelRpm+1; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam); Put CheckState() on Call queue;
Wheel	Accel	DirectDrive	CheckState()	AxelRpm-2<=WheelRpm	WheelRpm:=AxelRpm; Call Gauges.Speed(WheelRpm*(3.14159)*60*WheelDiam);

Appendix VII: Executable Test Cases

SeqNbr	VDUs	ExecutableCall	WaitFunction	Comments
1	0	Ignition.Ignition()	Pause(5)	Starts application and initializes several classes
2	4	AutoSystem.ThrottleFloor()	Pause(5)	read only - no effect
3	4	AutoSystem.ThrottleGovernor()	Pause(5)	read only - no effect
4	4	Gauges.Tach()	Pause(5)	read only - no effect
5	4	Gauges.Speed()	Pause(5)	read only - no effect
6	4	Gauges.Cruise()	Pause(5)	read only - no effect
7	7	Ignition.StartEngine()	Pause(8)	Initializes several other classes
8	1	Transmission.Gear()	Pause(5)	read only - no effect
9	40	Transmission.Gear(1)	Pause(5)	Puts car in gear at Idle speed
10	1	Gauges.Tach()	Pause(5)	read only - no effect
11	1	Gauges.Speed()	Pause(5)	read only - no effect
12	1	Gauges.Cruise()	Pause(5)	read only - no effect
13	45	GasUser.PedalPosition(15)	Pause(10)	Higher manual speed in first gear - approx 8
14	1	Gauges.Tach()	Pause(5)	read only - no effect
15	2	Transmission.Gear(2)	Pause(5)	Higher gear - higher speed < 25 - approx 14
16	2	CruiseUser.Switch(On)	Pause(5)	Prepare Cruise for action - no other effect
17	3	CruiseUser.Mode(RA)	Pause(5)	No effect because speed < 25
18	4	CruiseUser.Mode(NT)	Pause(5)	No effect because speed < 25
19	0	CruiseUser.Mode(SD)	Pause(5)	No effect because speed < 25
20	8	CruiseUser.Mode(NT)	Pause(5)	No effect because speed < 25
21	4	CruiseUser.Cancel()	Pause(5)	No effect because speed < 25
22	4	GasUser.PedalPosition(20)	Pause(10)	increases speed - approx 19
23	0	Transmission.Gear(3)	Pause(10)	Higher gear - higher speed - approx 25
24	0	GasUser.PedalPosition(25)	Pause(10)	increases speed - approx 35
25	0	Transmission.Gear(4)	Pause(15)	Higher gear - higher speed - approx 55
26	0	GasUser.PedalPosition(30)	Pause(10)	Car at Hwy speed - 4th gear - approx 66
27	0	CruiseUser.Cancel()	Pause(5)	No effect on Cruise in this state
28	1	CruiseUser.Mode(RA)	Pause(5)	No effect on Cruise in this state
29	2	CruiseUser.Mode(NT)	Pause(5)	No effect on Cruise in this state
30	0	CruiseUser.Mode(SD)	Pause(5)	No effect on Cruise in this state - but prepares for "Set"
31	67	CruiseUser.Mode(NT)	Pause(5)	Sets Cruise at Hwy speed - approx 66
32	1	Gauges.Cruise()	Pause(5)	read only - no effect
33	2	Engine.ExternalDrag(0.9)	Pause(10)	speed increases - downhill or tailwind - Cruise maintains @ 66
34	72	Engine.ExternalDrag(1.1)	Pause(10)	speed decreases - uphill or headwind - Cruise maintains @ 66
35	37	GasUser.PedalPosition(50)	Pause(40)	Manual throttle to pass a car or something - Max speed 105 > FastCutoff
36	24	Engine.ExternalDrag(0.8)	Pause(30)	speed increases - Throttle still manual - approx 162
37	2	Engine.ExternalDrag(1.6)	Pause(80)	speed decreases until reaches Targetspeed - May fail here and get oscillation!! Intermitant!

SeqNbr	VDUs	ExecutableCall	WaitFunction	Comments
38	9	GasUser.PedalPosition(20)	Pause(45)	To ensure that Cruise and Throttle return to Automatic state - speed approx 66
39	3	Engine.ExternalDrag(1.0)	Pause(30)	speed increases - downhill or tailwind - Cruise maintains @ 66
40	0	Engine.ExternalDrag(1.1)	Pause(5)	speed decreases - uphill or headwind - Cruise maintains @ 66
66	21	CruiseUser.Mode(SD)	Pause(6)	
67	14	CruiseUser.Mode(NT)	Pause(10)	Car in new slower cruise speed - approx 57
68	17	CruiseUser.Mode(RA)	Pause(6)	
69	13	CruiseUser.Mode(NT)	Pause(10)	Car in new faster cruise speed - approx 66
70	8	CruiseUser.Cancel()	Pause(5)	Cruise in Override state - speed starts to fall
71	4	CruiseUser.Mode(RA)	Pause(2)	no effect - prepare to Resume
72	45	CruiseUser.Mode(NT)	Pause(40)	Return to Cruise state - speed increases to that of #69 - approx 66
73	8	Engine.ExternalDrag(1.04)	Pause(10)	drag decreases - speed increases - Cruise maintains @ 66
74	5	Engine.ExternalDrag(1.3)	Pause(30)	drag increases - speed decreases - Cruise maintains @ 66
75	4	GasUser.PedalPosition(50)	Pause(15)	Manual override to pass car or something - speed approx 80
76	0	Engine.ExternalDrag(0.8)	Pause(60)	drag decreases - speed increases even more - approx 137
77	0	Engine.ExternalDrag(1.0)	Pause(20)	drag increases - speed decreases - approx 117
78	0	GasUser.PedalPosition(20)	Pause(15)	To ensure that Cruise and Throttle return to Automatic state @ ??
81	0	Engine.ExternalDrag(1.0)	Pause(10)	
82	0	Engine.ExternalDrag(0.9)	Pause(10)	
83	0	Engine.ExternalDrag(0.8)	Pause(10)	
84	0	Engine.ExternalDrag(0.7)	Pause(10)	The VDUs identified in tests 33 - 37 above are really
85	0	Engine.ExternalDrag(0.6)	Pause(10)	spread out over these External Drag actions and those
86	0	Engine.ExternalDrag(0.5)	Pause(10)	identified in 121 - 135 below.
87	0	Engine.ExternalDrag(0.67)	Pause(10)	
88	0	Engine.ExternalDrag(0.77)	Pause(10)	
89	0	Engine.ExternalDrag(0.87)	Pause(10)	Testing for gradual changes in external drag
90	0	Engine.ExternalDrag(0.97)	Pause(10)	at smooth increments down-up-down
91	0	Engine.ExternalDrag(1.07)	Pause(10)	
92	0	Engine.ExternalDrag(1.17)	Pause(10)	
93	0	Engine.ExternalDrag(1.27)	Pause(10)	
94	0	Engine.ExternalDrag(1.37)	Pause(10)	
95	0	Engine.ExternalDrag(1.47)	Pause(10)	
96	0	Engine.ExternalDrag(1.57)	Pause(10)	
97	0	Engine.ExternalDrag(1.67)	Pause(10)	
98	0	Engine.ExternalDrag(1.53)	Pause(10)	
99	0	Engine.ExternalDrag(1.43)	Pause(10)	
100	0	Engine.ExternalDrag(1.33)	Pause(10)	
101	0	Engine.ExternalDrag(1.23)	Pause(10)	
102	0	Engine.ExternalDrag(1.13)	Pause(10)	

SeqNbr	VDUs	ExecutableCall	WaitFunction	Comments
103	0	Engine.ExternalDrag(1.03)	Pause(10)	
104	0	Engine.ExternalDrag(1.0)	Pause(10)	
106	7	CruiseUser.Mode(RA)	Pause(5)	
107	0	CruiseUser.Mode(NT)	Pause(20)	Car in new faster cruise speed - approx 77
108	7	CruiseUser.Mode(SD)	Pause(5)	
109	0	CruiseUser.Mode(NT)	Pause(20)	Car in new slower cruise speed - approx 71
110	1	CruiseUser.Cancel()	Pause(5)	Speed falls - catch before < 30
111	0	CruiseUser.Mode(SD)	Pause(0)	
112	22	CruiseUser.Mode(NT)	Pause(30)	Car in new slower cruise speed - approx 55
113	36	CruiseUser.Mode(RA)	Pause(3)	Speed increases - stop before 70
114	4	CruiseUser.Mode(NT)	Pause(15)	Car in new faster cruise speed - approx 64
115	0	CruiseUser.Mode(RA)	Pause(3)	Speed increases - stop before 80
116	0	CruiseUser.Mode(NT)	Pause(15)	Car in new faster cruise speed - approx 71
117	4	CruiseUser.Mode(SD)	Pause(6)	Speed decreases - stop before 65
118	1	CruiseUser.Mode(NT)	Pause(15)	Car in new slower cruise speed - approx 62
119	0	CruiseUser.Mode(SD)	Pause(3)	Speed decreases - stop before 55
120	0	CruiseUser.Mode(NT)	Pause(15)	Car in new slower cruise speed - approx 55
121	3	Engine.ExternalDrag(1.2)	Pause(1)	
122	0	Engine.ExternalDrag(0.82)	Pause(1)	
123	0	Engine.ExternalDrag(0.62)	Pause(1)	
124	0	Engine.ExternalDrag(0.52)	Pause(1)	
125	0	Engine.ExternalDrag(0.56)	Pause(1)	
126	0	Engine.ExternalDrag(0.78)	Pause(1)	
127	0	Engine.ExternalDrag(0.98)	Pause(1)	Testing for rapid changes in external drag
128	0	Engine.ExternalDrag(1.18)	Pause(1)	at both smaller and larger increments
129	0	Engine.ExternalDrag(1.20)	Pause(1)	encompassing down-up-down
130	0	Engine.ExternalDrag(1.48)	Pause(1)	
131	0	Engine.ExternalDrag(1.58)	Pause(1)	
132	0	Engine.ExternalDrag(1.54)	Pause(1)	
133	0	Engine.ExternalDrag(1.34)	Pause(1)	
134	0	Engine.ExternalDrag(1.14)	Pause(1)	
135	0	Engine.ExternalDrag(1.0)	Pause(1)	
148	0	CruiseUser.Mode(SD)	Pause(2)	Speed decreasing - stop before 45
149	27	CruiseUser.Cancel()	Pause(1)	Hit cancel while holding SD down - Special override (Usermode null)
150	5	CruiseUser.Mode(NT)	Pause(2)	No effect because Usermode is null. Speed falling - keep > 35
151	0	CruiseUser.Mode(SD)	Pause(0)	
152	27	CruiseUser.Mode(NT)	Pause(5)	New Cruise speed set > 35
153	7	CruiseUser.Cancel()	Pause(1)	Speed falls again - keep > 25

SeqNbr	VDUs	ExecutableCall	WaitFunction	Comments
154	0	CruiseUser.Mode(SD)	Pause(0)	
155	0	CruiseUser.Mode(NT)	Pause(5)	New Cruise speed set > 25 - approx 29
156	0	CruiseUser.Mode(RA)	Pause(6)	Speed starts increasing - keep < 70
157	2	CruiseUser.Cancel()	Pause(2)	Hit Cancel while holding RA down - Special override (Usemode null)
158	1	CruiseUser.Mode(NT)	Pause(3)	No effect because Usermode is null. Speed falling
159	0	CruiseUser.Mode(RA)	Pause(0)	To resume previous speed set at #155 - approx 30
160	14	CruiseUser.Mode(NT)	Pause(5)	Speed identical to #155 speed - approx 30
161	2	CruiseUser.Mode(SD)	Pause(4)	Hold SD down until speed < SlowCutoff
162	0	CruiseUser.Mode(NT)	Pause(5)	No effect - Cruise in Override state
163	48	GasUser.PedalPosition(26)	Pause(5)	Throttle in Manual state - speed approx 60
164	0	CruiseUser.Mode(SD)	Pause(1)	
165	9	CruiseUser.Mode(NT)	Pause(5)	New Cruise Hwy speed - approx 60
166	65	CruiseUser.Cancel()	Pause(4)	speed falls - keep > 25
167	0	CruiseUser.Mode(RA)	Pause(0)	
168	28	CruiseUser.Mode(NT)	Pause(5)	Resume cruise speed of #165 - approx 60
169	37	GasUser.PedalPosition(30)	Pause(5)	Pass a car with Throttle in Manual state
170	7	GasUser.PedalPosition(20)	Pause(5)	Cruise keeps speed approx 60 and resets Pedal to approx 26
171	16	CruiseUser.Mode(RA)	Pause(3)	Speed increases - keep < 70
172	4	CruiseUser.Mode(NT)	Pause(5)	New higher Cruise speed - approx 70
173	2	CruiseUser.Mode(RA)	Pause(6)	Hold until speed > HighCutoff Speed begins to fall
174	0	CruiseUser.Mode(NT)	Pause(3)	No effect - Cruise in special Override state - speed falling
175	0	CruiseUser.Mode(RA)	Pause(3)	Hold until speed approx 45
176	4	CruiseUser.Mode(NT)	Pause(20)	Resume cruise speed of #172 - approx 70
177	11	CruiseUser.Switch(Off)	Pause(5)	Speed begins to fall
178	1	CruiseUser.Cancel()	Pause(5)	No effect - speed continues to fall - Test may be INFEASIBLE since CruiseUser is OFF
179	1	CruiseUser.Mode(SD)	Pause(5)	No effect - speed continues to fall - Test may be INFEASIBLE since CruiseUser is OFF
180	0	GasUser.PedalPosition(25)	Pause(5)	Puts speed at slow hwy speed like #24 - approx 35-40
181	57	CruiseUser.Switch(On)	Pause(5)	Prepare for new Cruise actions - no other effect
182	0	CruiseUser.Mode(SD)	Pause(1)	
183	15	CruiseUser.Mode(NT)	Pause(5)	Sets Cruise at slow hwy speed equal to #180 - approx 35-40
184	26	CruiseUser.Cancel()	Pause(5)	Cruise in Override state - speed begins to fall
185	2	CruiseUser.Switch(Off)	Pause(5)	Cruise in Off state - speed continues to fall - ends at Idle state