Integration Testing of Object-oriented Components
Using Finite State Machines

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.

This work was prepared by United States Government employees as part of their official duties and is, therefore, a work of the U.S. Government and not subject to copyright.

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 (defs) 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, \( S = \{ s | s = (\text{pred}) \} \) where \( \text{pred} \) is a predicate on the instance variables in \( V \).
- \( T \) is a finite set of transitions, \( T = \{ t | t = (\text{source}, \text{target}, \text{fn}, \text{guard}, \text{action}) \} \) where:
  - \( \text{source}, \text{target} \in S \) are the states before and after the transition.
  - \( \text{fn} \in F \) is a member function that triggers \( t \) if the guard predicate evaluates to \( \text{true} \).
  - \( \text{guard} \) is a predicate on instance variables in \( V \) and parameters of member functions in \( F \).
  - \( \text{action} \) 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: \( \text{speed} = 0 \land \text{KeyOn} = \text{false} \)  
ON: \( 0 \leq \text{speed} \leq 110 \land \text{KeyOn} = \text{true} \)

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 \( (\text{KeyOn} = \text{true}) \), and the action should specify that the speed is set \( (\text{speed} = S) \).

The sets of variables, member functions, states, and transitions are defined as follows:

- \( S = \{ S_0, S_t, \text{ON}, \text{OFF} \} \)
- \( V = \{ \text{int} \text{ speed}, \text{boolean} \text{ KeyOn} \} \)
- \( F = \{ \text{Engine()}, \neg\text{Engine()}, \text{setKeyOn( boolean in)}, \text{Start (int S)}, \text{Stop ()}, \text{setSpeed (int S)}, \text{int getSpeed ()} \} \)
- \( P = \{ \text{setKeyOn:in}, \text{Start:S}, \text{setSpeed:S} \} \)
- \( T = \{ t_i | 1 \leq i \leq 9 \} \)
  - \( t_1 = (S_0, \text{OFF}, \text{Engine()}, \text{true}, \{ \text{speed} = 0, \text{KeyOn} = \text{false} \}) \)
  - \( t_2 = (\text{OFF}, \text{OFF}, \text{getSpeed()}, \text{true}, \{ \text{return speed} \}) \)
  - \( t_3 = (\text{OFF}, \text{OFF}, \text{setKeyOn(in)}, \text{true}, \{ \text{KeyOn = in} \}) \)
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 ∧ 0 ≤ S ≤ 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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

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.
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 as follows:

- The StateRefVar association between State and Variable is a table of tuples \((c, s, v)\) where \((c, s)\) identifies a state and \((c, v)\) identifies a variable in the same class as the state. The definition predicate of the state references the variable. In the Engine example, the OFF state references both speed and KeyOn.

- The GuardRefVar, ActionDefVar and ActionRefVar associations between Transition and Variable are each a table of tuples \((c, t, v)\) where \((c, t)\) identifies a transition and \((c, v)\) identifies a variable in the same class as that transition. In the first association, the guard of the transition references the variable, in the second association the action of the transition defines the variable, and in the third association the action of the transition references the variable. Since each action in a sequence of actions has a sequence number, a non-key attribute, SeqNbr, is assigned to each instance of the second and third associations. In the Engine example, the guard of \(t_4\) references KeyOn, the action of \(t_1\) defines first speed and then KeyOn, and the actions of \(t_2\) and \(t_6\) both reference speed.

- The StateRefActorFn association between State and Function is a table of tuples \((c_s, s, c_f, f)\) where \((c_s, s)\) identifies a state and \((c_f, f)\) identifies an actor function. The definition predicate of the state references the actor function. In the Automobile example, all of the states defined for CruiseUnit reference the Cruise variable from the Gauges class of the InstrumentPanel component to see if cruise control is On or Off (not visible in Appendix I).

- The GuardRefActorFn, ActionRefActorFn, and ActionRefMutatorFn associations between Transition and Variable are each a table of tuples \((c_t, t, c_f, f)\) where \((c_t, t)\) identifies a transition and \((c_f, f)\) identifies a function. In the first association the guard of the transition references an actor function, in the second association the action of the transition references an actor function, and in the third association the action of the transition references a mutator function. As above, a SeqNbr attribute is assigned to each instance of the second and third associations to identify the position of that reference in the action sequence. The Guard and Action columns of Appendix I show many instances of these types of references for the Automobile example.

- The GuardRefParm and ActionRefParm associations between Transition and Parameter are each a table of tuples \((c_t, t, n)\), where \((c_t, t)\) identifies a transition whose guard or action references (by name) a parameter of the function
associated with that transition and \( n \) is the position of that parameter in the signature of the function. In the Automobile guards shown in Appendix I, nearly every guard of a transition derived from a mutator function that has a parameter references that parameter by name, and the actions of all transitions derived from the Speed method in the Gauges class and the Floor and GasPedal methods in the Throttle class all reference the incoming parameter by name. As above, an additional non-key attribute in the \( \text{ActionRefParm} \) association, called \( \text{SeqNbr} \), captures the sequence number of that reference in the action sequence of the transition.

- The \( \text{ActionSetsParm} \) association between Transition and Parameter is a table of tuples \((c_t, t, c_f, f, n)\) where \((c_t, t)\) identifies a transition whose action calls a function, identified by \((c_f, f)\), from some other class and sets the \( n \)-th parameter of that function with some non-constant value, possibly the value of a state variable from some class \( c \).

For the Automobile example, Appendix I shows actions in several transitions of CruiseUnit that call the Throttle.Floor\((x)\) function and set \( x \) either to the TargetThrottle variable of CruiseUnit, or to a value derived from the value of the Position variable from the Throttle class. This association also carries an additional non-key attribute to capture the \( \text{SeqNbr} \) of the set operation in the action sequence of the transition.

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

Every state variable in a class definition is associated with two pre-defined methods, one to \textbf{get} its value and one to \textbf{set} its value. An additional association \( \text{VarAssocFn} \) is defined between Variable and Function to maintain the relationship between a state variable and the \textbf{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 \( \text{ActionSetsParm} \) 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 \( \text{ActionSetsParmUsingVar} \) 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 \( \text{ActionSetsParm} \) 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 \textbf{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 \textit{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 \( \text{ActionRefLocalAsyn} \) 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. This is done by a straightforward mapping. Every table can be associated with a mathematical set, where the set is a set of sequences consisting only of the primary key values of the table. In this sense, the sequence (c, f) is an element of the Function set if and only if there exists a row in the Function table with primary key values (c, f). If X is such a table-derived set, if w is a non-key column of the corresponding table T, and if x ∈ 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 R₀. The iterative process starts with R₀. At each step, assume that n steps of the process have been completed, resulting in a set Rₙ of relevant transitions, each of which is labeled as In, Out, or Base. A transition may appear in Rₙ as many as three times with different labels. To create the next set of relevant transitions, Rₙ₊₁, first initialize Rₙ₊₁ to be Rₙ, 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 \( t' \). If \( t' \) is 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 (Ects) is asynchronous, as explained below. The total set of edges E is defined as:

\[ E = E_{st} \cup E_{sg} \cup E_{gr} \cup E_{ts} \cup E_{gt} \cup E_{xt} \cup E_{xt} \cup E_{cts} \]

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

- **E_{st}** edges represent data flow from states to transitions. 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_{ggt}** 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 \( \text{GuardRefActorFn} \) 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 \( \text{StateRefActorFn} \) association determines these edges and the Automobile example has 10 instances.
- **E_{txs}** 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 \( \text{ActionRefMutatorFn} \) association determines these edges and the Automobile example has 161 instances.
The 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 incoming 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:

- \( E_{\text{cts}} \) 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 Testing 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 the 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 \( S \).

- \( v \) is **defined** at a transition-node \( n_t \in N \), if the variable and the transition are from the same class and if they satisfy the association \( (c, t, v) \in ActionDefVar \). Each variable definition carries along the SeqNbr attribute of the ActionDefVar association.
- \( v \) is **directly computation-used** at a transition-node \( n_t \in N \), if the variable and the transition are from the same class and if they satisfy the association \( (c, t, v) \in ActionRefVar \).
- \( v \) is **indirectly computation-used** at a transition-node \( n_t \in N \), if the variable is associated with the \texttt{get} method \( f \) in its class \( c \) and if the transition and the function satisfy the association \( (ct, t, c, f) \in ActionRefActorFn \).
- \( v \) is **directly predicate-used** at any state-transition-edge \( (ns, nt) \in Est \) if the state satisfies the association \( (c, s, v) \in StateRefVar \).
- \( v \) is **indirectly predicate-used** at any state-transition-edge \( (ns, nt) \in Est \) if the variable is associated with the \texttt{get} method \( f \) in its class \( c \) and if the state and that function satisfy the association \( (cs, s, c, f) \in StateRefActorFn \).
- \( v \) is **directly predicate-used** at any state-guard-edge \( (ns, ng) \in Esg \) if the state satisfies the association \( (c, s, v) \in StateRefVar \).
- \( v \) is **indirectly predicate-used** at any state-guard-edge \( (ns, ng) \in Esg \) if the variable is associated with the \texttt{get} method \( f \) in its class \( c \) and if the state and the method satisfy the association \( (cs, s, c, f) \in StateRefActorFn \).
- \( v \) is **directly predicate-used** at a guard-transition-edge \( (ng, nt) \in Egt \) if the transition satisfies the association \( (ct, t, c, v) \in GuardRefVar \).
- \( v \) is **indirectly predicate-used** at a guard-transition-edge \( (ng, nt) \in Egt \) if the variable is associated with the \texttt{get} method \( f \) in its class \( c \) and if the transition and \( f \) satisfy the association \( (ct, t, c, f) \in GuardRefActorFn \).
- \( v \) is **parameter computation-used** at a transition-node \( n_t \in N \), if the action of the transition associated with \( n_t \), called \( (c_t, t) \), references the \( n \)-th parameter of the function associated with \( t \) by name, that is if \( (c_t, t, n) \in ActionRefParm \), and if the variable is used to set the \( n \)-th parameter of some function, that is if there exists a transition \( t_l \) whose action
calls a function \((c_1, f)\) such that \((c_{t1}, t_1, c_1, f, n, c, v) \in \text{ActionSetsParmUsingVar}\), and if that function is the function associated with \(t\), that is if \(c_t = c_{t1}\) and \(\text{method}(t) = f\).

- \(v\) is parameter predicate-used at a guard-transition-edge \((n_g, n_t) \in \mathcal{E}_{gt}\) 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 \((c_t, t, n) \in \text{GuardRefParm}\), and if the variable is used to set the \(n\)-th parameter of some function, that is if there exists a transition \(t_1\) whose action calls a function \((c_{t_1}, f)\) such that \((c_{t_1}, t_1, c_{t_1}, f, n, c, v) \in \text{ActionSetsParmUsingVar}\), and if that function is the function associated with \(t\), that is if \(c_t = c_{t1}\) and \(\text{method}(t) = f\).

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 (\(\mu\)) represents an edge or a node that is a use. An ordered pair \((n, \mu)\) is said to be a def-use pair for \(v\) if \(v\) is defined at the transition-node \(n\), and if \(\mu\) is either a node or an edge in \(G\) where \(v\) 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 \(v\) be a variable that is both defined and used at one or more transition nodes \(n \in \mathcal{N}_t\). Denote by DFTU\((v)\) the set of such nodes where \(v\) is defined first and then used, and denote by UFDL\((v)\) the set of all such nodes where \(v\) 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 \(n\).

---

\(^1\) 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.
The sets DFTU(\(\nu\)) and UFDL(\(\nu\)) are not necessarily mutually exclusive. A transition involving variable \(x\) with an action that consists of the sequence \(\text{``}x := x+1; y := f(x)\text{''}\) 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 \((\nu, n_t, \mu)\) where \(\nu\) is a variable, \(n_t\) is a transition node that defines \(\nu\), and \(\mu\) is a node or edge where \(\nu\) is used. \(n_t\) and \(\mu\) form a DU-pair if there exists a path in the component flow graph leading from \(n_t\) to \(\mu\), if the path is free of loops, if there are no defs to \(\nu\) 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 \(k \geq 1\) is a sequence of nodes \(n_1 \ldots n_k\) such that \((n_i, n_{i+1}) \in E\) for \(1 \leq i \leq k-1\). If \(p\) is a path, then the head of \(p\), denoted by \(H(p)\), is the first element of the sequence, the tail of \(p\), denoted by \(T(p)\) is the last element of the sequence, and the length of \(p\), denoted by \(L(p)\), 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 \(p\), 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 \((n_i, n, n_j)\) is a mutator Transition-State-Guard (TSG) path segment if the edge \((n_i, n)\) has a function label. A mutator TSG path segment is potentially feasible if the edge \((n_i, n)\) 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 $v$ be any variable in $S$, let $n_t$ be a transition node that defines $v$, and let $\mu$ be a node or an edge where $v$ is used. A path $p$ in $G$ is said to be a DU-path from $n_t$ to $\mu$ for $v$ if $p = n_t; q; \mu$, where $q$ is a path in $G$ such that no node of $q$ is a definition node for $v$ and every mutator TSG path segment in $p$ is potentially feasible. The pair $(n_t, \mu)$ is said to be a DU-pair for $v$ 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 $(v, n_t, \mu)$ where $(n_t, \mu)$ is a def-use pair for $v$ and let $P$ be a set of tuples $(v, n_t, \mu, p)$ where $(n_t, \mu)$ is a DU-pair for $v$ and $p$ is a DU-path from $n_t$ to $\mu$. 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 $P$ beginning with definitions for $P_1$ and $P_2$ below. It searches for candidate test paths using a breadth-first algorithm for finding paths from one node to another in a directed graph, a modification of Dijkstra’s shortest-path algorithm that starts at both beginning and end nodes, and meets in the middle. It works breadth-first from definition nodes and use nodes or edges, simultaneously forming two sets of partial paths. The def-partial paths are paths whose head is the definition node for a state variable and whose tail is a candidate node for connecting to a use of that variable. The use-partial paths are paths whose tail is a transition node where a variable is computation-used, or whose last two tail nodes determine an edge where the variable is predicate-used, and whose head is a candidate node for connecting to a definition of that state variable. 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 following rules must be followed in the construction of def-partial paths:

- The first edge from a definition node of a state variable toward a new candidate intermediate node shall not be labeled with an OutSeq number that is less than the SeqNbr of that definition.
- If the in-coming edge entering the tail of a def-partial path is labeled with an InSeq number equal to x, then the outgoing edge toward a new candidate intermediate node shall not be labeled with an OutSeq number less than x.
- If tail of a def-partial path is a state node, and if the in-coming edge to that tail has a function label, then the outgoing edge to a new candidate intermediate transition or guard node shall be derived from the same function.
- If the tail of a def-partial path is a state node, and if the in-coming edge to that tail has a function label, then the TSG path segment derived from the incoming transition node, the state node, and any new candidate intermediate guard node must be a potentially feasible mutator TSG path segment.
- If the tail of a def-partial path is a state node, and if the in-coming edge to that tail does not have a function label, then any new candidate intermediate guard or transition node must not be derived from a private function.
- If the tail of a def-partial path is a transition node, and if the edge to any new candidate intermediate state node has a function label, then the class of the new state node must be not equal to the class of the transition node OR the state of the new state node must be equal to the target state of the transition of the transition node.

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

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

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

\[
P_1 = \{(\nu, n, n_t, n) : (\nu, n, n_t) \in VDU \land n_t \in DFTU(\nu)\}
\]

\[
P_2 = \{(\nu, n, n_t, n) : (\nu, n, n_t, n) \in VDU \land (n_t, n) \in E_{xt}\}
\]

All paths in \( P_1 \) are of length 1 and all paths in \( P_2 \) are of length 2. In general, the sets \( P_i \) will identify paths of length \( i \) or \( i-1 \). The definition of \( P_i \) for \( i \geq 3 \) depends upon sets of partial paths, \( Q_i \), and unresolved def-use pairs, \( X_i \), both defined iteratively below. Each \( Q_i \) will be a tuple \((\nu, n_t, \mu, h, t) \) where \((\nu, n_t, \mu) \in VDU \) \( h \) is a path from a def node \( n_t \) of \( \nu \) to an intermediate node, and \( t \) is a path from some other intermediate node to a use item \( \mu \) for \( \nu \). Each \( X_i \) will be a subset of VDU, consisting of variable and def-use pairs that still do not have a connecting path. The algorithm begins with:

\[
X_1 = VDU
\]

\[
Q_1 = \{(\nu, n_t, \mu, n) : (\nu, n_t, \mu) \in VDU\}
\]

\[
X_2 = VDU - \{(\nu, n, n_t, n) : (\nu, n, n_t, n) \in P_1 \land n_t \notin UFDL(\nu)\}
\]

\[
Q_2 = \{(\nu, n, \mu, n_t, n) : (\nu, n, \mu) \in X_2\}
\]

and given \( Q_i \) ( \( i \geq 2 \) ) the next step defines:

\[
P_{i+1} = \{(\nu, n, \mu, h, t) : (\nu, n, \mu, h, t) \in Q_i \land e = (T(h), H(t)) \in E \land InSeq(h) \leq OutSeq(e) \land InSeq(e) \leq OutSeq(t) \land (\neg \exists FN(Pre(T(h)), T(h))) \lor FN(Pre(T(h)), T(h)) = \text{method}(H(t)) \land ((Pre(T(h)), T(h), H(t)) \notin MTSG \lor (Pre(T(h)), T(h), H(t)) \in FTSG) \land (\neg \exists FN(e) \lor FN(e) = \text{method}(Pre(H(t))) \land ((T(h), H(t), Pre(H(t))) \notin MTSG \lor (T(h), H(t), Pre(H(t))) \in FTSG) \land ((T(h) \in N_t \land \neg \exists FN(Pre(T(h)), T(h))) \Rightarrow availability(FN(e)) \neq PRI) \land ((H(t) \in N_t \land availability(FN(H(t), Pre(H(t)))) = PRI) \Rightarrow \exists FN(e)) \land ((T(h) \in N_t \land H(t) \in N_t \land \exists FN(e)) \Rightarrow (\text{class}(T(h)) \neq \text{class}(H(t)) \lor \text{state}(H(t)) = \text{targetState}(T(h)))\}
\]

\[
C_{i+1} = \{(\nu, n, \mu) : \exists p \{(\nu, n, \mu, p) \in P_{i+1}\}\}
\]

\[
A_{i+1} = \{(\nu, n, \mu) : \exists h, t \{(\nu, n, \mu, h, t) \in Q_i\}\}
\]

\[
X_{i+1} = X_i - A_{i+1}
\]

\[
B_{i+1} = X_i - A_{i+1}
\]

The sets \( C_{i+1} \) identify unresolved def-use pairs in \( X_i \) that have found a DU-path in \( P_{i+1} \). The sets \( A_{i+1} \) identify unresolved def-use pairs in \( X_i \) that remained active candidates for resolution in \( P_{i+1} \). The sets \( B_{i+1} \) identify def-use pairs that drop out of consideration for resolution at this step of the iteration; the variable they are associated with is said to be def-bound (formally defined in Definition 6.7).
The sets of partial paths are defined iteratively as follows. Given $Q_{2k}$, the def-partial paths are extended by defining:

$$Q_{2k+1} = \{ (v, n, \mu, h, n, t) \mid (v, n, \mu, h, t) \in Q_{2k} \text{ AND } \exists n \in N \text{ AND } (T(h), n) \in E \text{ AND } n \notin D(v) \text{ AND } T(h) \cdot n \notin h \text{ AND } (v, n, \mu) \in X_{2k+1}$$

AND InSeq(h) $\leq$ OutSeq(T(h), n)
AND $\neg (\exists \text{Fn}((\text{Pre}(T(h)), T(h))) \text{ OR } \text{Fn}((\text{Pre}(T(h)), T(h))) = \text{method}(n))$
AND $((\text{Pre}(T(h)), T(h), n) \notin \text{MTSG} \text{ OR } (\text{Pre}(T(h)), T(h), n) \in \text{FTSG})$
AND $((T(h) \in N_s \text{ AND } \neg \exists \text{Fn}((\text{Pre}(T(h)), T(h))) \rightarrow \text{availability}(\text{Fn}(T(h), n)) \neq PRI)$
AND $((T(h) \in N_i \text{ AND } n \in N_s \text{ AND } \exists \text{Fn}(T(h), n)) \rightarrow (\text{class}(n) \neq \text{class}(T(h)) \text{ OR state}(n) = \text{targetState}(T(h))) ) \}$

and given $Q_{2k+1}$, the use-partial paths are extended by defining:

$$Q_{2k+2} = \{ (v, n, \mu, h, n, t) \mid (v, n, \mu, h, t) \in Q_{2k+1} \text{ AND } \exists n \in N \text{ AND } (n, H(t)) \in E \text{ AND } n \notin D(v) \text{ AND } n:H(t) \notin t \text{ AND } (v, n, \mu) \in X_{2k+2}$$

AND InSeq(n:H(t)) $\leq$ OutSeq(t)
AND $\neg (\exists \text{Fn}(n, H(t))) \text{ OR } \text{Fn}(n, H(t)) = \text{method}(\text{Pre}(H(t)))$
AND $((n, H(t), \text{Pre}(H(t))) \notin \text{MTSG} \text{ OR } (n, H(t), \text{Pre}(H(t))) \in \text{FTSG})$
AND $((H(t) \in N_s \text{ AND } \text{availability}(\text{Fn}(H(t), \text{Pre}(H(t)))) = PRI) \rightarrow \exists \text{Fn}(n, H(t)))$
AND $((n \in N_i \text{ AND } H(t) \in N_s \text{ AND } \exists \text{Fn}(n, H(t))) \rightarrow (\text{class}(n) \neq \text{class}(H(t)) \text{ OR state}(H(t)) = \text{targetState}(H(t))) ) \}$

where $D(v)$ is the set of definition nodes for $v$, $\text{Fn}(e)$ is the function label of an edge $e$, $\text{Pre}(T(h))$ is the preceding node adjacent to $T(h)$ in $h$, and $\text{Pre}(H(t))$ is the following node adjacent to $H(t)$ in $t$. InSeq and OutSeq inequalities are satisfied if either label is null. The $Q$’s with odd subscripts are building partial paths from the def node, whereas the $Q$s with even subscripts are building paths from the uses.

The iterative process stops when $X_i = \emptyset$. At this point, set $P = \cup P_i$. This must happen for some value of $i$ less than the number of edges in the graph since cycles were avoided by ensuring that no edge appears more than once in any of the partial paths. It is possible for some state nodes and some transition nodes to appear more than once in a partial path. Not all elements $(v, n, \mu) \in \text{VDU}$ will yield a DU-path. Some variables may be defined at a node $n_i$ and used at a use item $\mu$ but either no path exists from $n_i$ to $\mu$ that satisfies the above constraints, or every such path contains a re-definition of $v$.

**Definition 6.7 (def-bound):** A variable $v$ is said to be def-bound at a definition node $n_i$ of a def-use pair $(n_i, \mu)$ if there is no path from $n_i$ to $\mu$ ($p = (v, n, \mu, p) \in P$).

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 \( \nu \) 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 \( \mu \). 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 Automobile Example

This section presents the results of the methodology applied to 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 illustrated on CruiseControl below. The initial tables contain the following:

- Class table: 10 rows - one for each class in the system
- Variable table: 46 rows - with 9 for CruiseControl
- Function table: 97 rows - with 20 for CruiseControl
- State table: 76 rows - with 16 for CruiseControl
- Transition table: 143 rows - with 80 for CruiseControl

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

- StateRefVar: 78 instances
- StateRefActorFn: 5 instances
- ActionDefVar: 135 instances
- ActionRefVar: 41 instances
- GuardRefVar: 95 instances
- GuardRefActorFn: 3 instances
- ActionRefActorFn: 25 instances
- ActionRefMutatorFn: 61 instances
- VarAssocFn: 40 instances
- ActionRefLocalAsyn: 24 instances
- GuardRefParm: 77 instances
- ActionRefParm: 23 instances
- ActionSetsParm: 27 instances
- VarAssocFn: 40 instances
- ActionSetsParmUsingVar: 27 instances

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

- \( R_{\infty} \): 80 CruiseControl Base transitions
- \( R_0 \): 65 feasible Base transitions
- \( R_1 \): 172 tagged transitions at first iteration
- \( R_2 \): 187 tagged transitions at second iteration
- \( R_3 \): 197 tagged relevant transitions at final pass
- \( R(M) \): 106 final untagged relevant transitions

\( R(M) \) has 80 transitions that have non-trivial actions, as displayed in Appendix I. The component flow graph defined in Section 5 is summarized as:

- Nodes: 208 nodes
  - TransitionNodes: 106
  - SourceStateNodes: 33 (27 are also target states)
  - TargetStateNodes: 27
  - Guard Nodes: 69

- Edges: 551 edges
There are also 8 virtual EXT Action Nodes and 24 virtual EXT Action Edges of the type discussed at the end of Section 5. The EXT nodes and edges help identify the methods that can be used to construct executable test cases; they are not part of the theoretical calculation of abstract test specifications.

The following is the list of defs and uses from the CruiseControl component (Section 6):

<table>
<thead>
<tr>
<th>Definition</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefnNodes</td>
<td>107</td>
</tr>
<tr>
<td>DirectCompUses</td>
<td>35</td>
</tr>
<tr>
<td>IndirectCompUses</td>
<td>25</td>
</tr>
<tr>
<td>DirectPredUseByState</td>
<td>329</td>
</tr>
<tr>
<td>DirectPredUseByGuard</td>
<td>74</td>
</tr>
<tr>
<td>IndirectPredUseByState</td>
<td>40</td>
</tr>
<tr>
<td>IndirectPredUseByGuard</td>
<td>3</td>
</tr>
<tr>
<td>ParmUseByGuard</td>
<td>43</td>
</tr>
<tr>
<td>ParmUseByTransition</td>
<td>46</td>
</tr>
<tr>
<td>Total VarDefUse triples</td>
<td>3433</td>
</tr>
<tr>
<td>DFTU nodes</td>
<td>18</td>
</tr>
<tr>
<td>UFDL nodes</td>
<td>7</td>
</tr>
<tr>
<td>DFTU &amp; UFDL</td>
<td>5</td>
</tr>
</tbody>
</table>

From Section 6.1, the following are mutator TSG path segments.

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTSG triples</td>
<td>283</td>
</tr>
<tr>
<td>KnownFeasible</td>
<td>169</td>
</tr>
<tr>
<td>KnownNotFeasible</td>
<td>53</td>
</tr>
<tr>
<td>Indeterminate</td>
<td>61</td>
</tr>
<tr>
<td>FTSG triples</td>
<td>230</td>
</tr>
</tbody>
</table>

The above calculations are almost instantaneous, even on a desktop personal computer. However, the next step, to calculate the **DU-paths** leading to DU-pairs becomes more computationally intensive. The iterative process to construct candidate test paths \( P_i \) from Definition 6.6 proceeds in several steps. Each step is summarized below:

\[ P_1 \quad 18 \text{ rows (from the DFTU nodes)} \]

There are 20 transitions derived from 25 def-use pairs with an action that both defines and uses the same variable, but only 18 of them were DFTU. The CruiseUnit transitions t004 and t032 in Appendix I define and then use TargetThrottle, and t029 and t031 define and then use CurrentSpeed. The GasUser transition t002 first defines and then uses PedalPosition, and Engine t003 first defines and then uses Rpm. All relevant Throttle transitions define and then use Position, with t004 using Position twice, and Throttle t003 first defines and then uses Floor. The seven UFDL transition nodes are CruiseUnit transitions t005 and t007 with TargetSpeed, which both use TargetSpeed first and then define it,
and Engine t003 with Rpm, which uses-defines-uses Rpm, and Engine t003 with GasFlow, which uses-then-defines
GasFlow. Throttle transitions t004, t007, and t009 have a parameter-use of Position as a result of calls to Floor(x) from
CruiseUnit, and then immediately define Position. Four of the Throttle def-use pairs are in both DFTU and UFDL, as
well as Engine t003 with variable Rpm. These five def-use pairs are kept active in the search for new DU-paths from
those nodes back to themselves. The remaining def-use pairs in DFTU have a use that can never be reached from outside
that node without re-definition of the variable, resulting immediately in 99 def-bound pairs. After step two, the pairs have:

\[ P_2 \quad 0 \text{ rows (none of the 58 E_{xol} edges give DU-pairs)} \]

There are 58 transitions whose actions return a value from a call to a function in some other class, but none of those calls
involved mutator functions so they do not result in any new DU-pairs with a test path of length 2. Thus \( P_2 \) is empty.
Continuing with initialization of the DU-pair generation process yields:

\[
\begin{align*}
X_1 & \quad 3433 \text{ instances (from VDU triples)} \\
Q_1 & \quad 3433 \text{ instances (from VDU – set Head and Tail)} \\
B_1 & \quad 99 \text{ instances (from 13 DFTU pairs not in UFDL)} \\
X_2 & \quad 3316 \text{ instances (removing 18 found and 99 def-bound pairs)} \\
Q_2 & \quad 3316 \text{ instances (removing same 117 instances from } Q_1) \\
\end{align*}
\]

At iteration 3, 363 candidate test paths of length 3 were found. Approximately one-half of the paths go through the target
state of their transitions to an out-going edge from that target state. Each of these yields a feasible path that is one
feasible transition followed by a second feasible transition. It is relatively straightforward to find an executable test case
for each of these paths. The other half of the paths do not go through the target state of the transition; instead, they follow
a mutator function to the source state of some other transition in the same class. All of these MTSG path segments are
based on a call to CheckState() in CruiseUnit and are known to be feasible. Again, it is relatively easy to find an
executable test case for each of these paths. At this step, all paths are still completely contained in a single class. All 363
new paths result in new DU-pairs that lead to executable test cases. Five new def-bound variables are identified:
BrakeActive and ClutchActive are defined in AutoSystem transitions but can never reach their predicate uses in some
Engine states, IsActive is defined in both BrakeControl and BrakeUser but can never reach its use in state predicates, and
PedalPosition is defined in ClutchUser but cannot reach its use a state predicate. In all cases a Cancel() message gets sent
instead to shut down all further CruiseControl processing.

Iteration 4 creates 12 new paths of length 3 and 57 new paths of length 4, all of which identify new DU-pairs. Two of the
length 3 paths are internal feasible transitions of Engine and the other 10 are definitions of the Position variable in
Throttle, passed as a parameter through a call of Engine. GasFlow(x), to parameter computation-use in transition t003 of
Engine. All of these abstract test specifications of length 3 paths are feasible. Of the length 4 paths, all but 16 remain in
the same class and result in known feasible paths consisting of successive calls of two feasible transitions. Of the 16 not in the same class, 8 define TargetSpeed in CruiseUnit and pass it via Floor(x) in t004 or t032 to a parameter predicate-use in Throttle, 4 define Rpm in Engine and pass it via Speed(x) in t003 to a parameter predicate-use in Gauges, and 4 define PedalPosition in GasUser and pass it via GasPedal(x) in t002 to a parameter predicate-use in Throttle. This iteration does not discover any new def-bound pairs.

The process takes 16 iterations as 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.

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

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.

8 Conclusions and Future Work

This paper presents two major results. The first is a method for integration level, inter-class testing for object-oriented programs using data flow techniques. The second is a technique for representing data flow and control flow graphs in a relational database and using the database as a compute engine for deriving DU-pairs and DU-paths to satisfy data flow testing criteria. Software components are 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.

This paper does not prove the existence of an executable test case for each DU-pair, but by eliminating def-bound pairs and by generating a small collection of potentially feasible candidate test paths for each remaining pair it substantially increases the likelihood of finding an executable test case. Careful design of the states and guards in a functional specification of transitions and early identification of non-feasible path segments in a component flow graph will also help reduce the number of non-feasible candidate test paths. Our future efforts will focus on the automatic generation of
executable test cases for black-box conformance testing using only the set of externally invokable methods to initiate data flow through a candidate test path.

Using the database has two advantages: (1) I/O is handled as disk storage, and (2) logical predicates are easier to handle and resolve, both in terms of space and complexity of the algorithms. Traditional code-level data systems do not provide complete paths 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, that can handle software specifications that have thousands of DU-pairs.

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 paper, and three possibilities emerge. The most obvious is when software components are integrated. At that time, the FSMs can be generated and relevant transitions can be determined to be those transitions that are included as part of the components in the current integration
step. It may also be possible to employ these techniques during maintenance. If a component is to be changed, the impact of that change can be estimated in terms of the relevant transitions, and regression testing can proceed on the relevant transitions. 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


### Appendix I: Relevant Feasible Mutator Transitions for CruiseControl

<table>
<thead>
<tr>
<th>Class</th>
<th>Trand</th>
<th>Source</th>
<th>Target</th>
<th>Function</th>
<th>Guard</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoSystem</td>
<td>t002</td>
<td>Inactive</td>
<td>Active</td>
<td>BrakeActive(x)</td>
<td>x=true</td>
<td>BrakeActive:=true; Call CruiseUnit.Cancel();</td>
</tr>
<tr>
<td>AutoSystem</td>
<td>t003</td>
<td>Inactive</td>
<td>Active</td>
<td>ClutchActive(x)</td>
<td>x=true</td>
<td>ClutchActive:=true; Call CruiseUnit.Cancel();</td>
</tr>
<tr>
<td>AutoSystem</td>
<td>t004</td>
<td>Active</td>
<td>Active</td>
<td>BrakeActive(x)</td>
<td>x=true</td>
<td>ClutchActive:=true; Call CruiseUnit.Cancel();</td>
</tr>
<tr>
<td>AutoSystem</td>
<td>t005</td>
<td>Active</td>
<td>Active</td>
<td>ClutchActive(x)</td>
<td>x=true</td>
<td>ClutchActive:=true; Call CruiseUnit.Cancel();</td>
</tr>
<tr>
<td>BrakeControl</td>
<td>t002</td>
<td>Inactive</td>
<td>Braking</td>
<td>IsActive(x)</td>
<td>x=true</td>
<td>IsActive:=true; Call AutoSystem.BrakeActive(true);</td>
</tr>
<tr>
<td>BrakeControl</td>
<td>t003</td>
<td>Braking</td>
<td>Inactive</td>
<td>IsActive(x)</td>
<td>x=false</td>
<td>IsActive:=false; Call</td>
</tr>
<tr>
<td>BrakeControl</td>
<td>t004</td>
<td>Locked</td>
<td>Inactive</td>
<td>IsActive(x)</td>
<td>x=false</td>
<td>IsActive:=false; Call</td>
</tr>
<tr>
<td>BrakeUser</td>
<td>t002</td>
<td>Inactive</td>
<td>Braking</td>
<td>IsActive(x)</td>
<td>x=true</td>
<td>IsActive:=true; Call AutoSystem.BrakeActive(true);</td>
</tr>
<tr>
<td>BrakeUser</td>
<td>t003</td>
<td>Braking</td>
<td>Inactive</td>
<td>IsActive(x)</td>
<td>x=false</td>
<td>IsActive:=false; Call</td>
</tr>
<tr>
<td>ClutchUser</td>
<td>t003</td>
<td>Inactive</td>
<td>Transition</td>
<td>PedalPosition(x)</td>
<td>x&gt;0</td>
<td>PedalPosition:=x; Call AutoSystem.Clutch Active(false); Call ClutchUnit.IsActive(true);</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t001</td>
<td>Initial</td>
<td>Off</td>
<td>CruiseUnit()</td>
<td>true</td>
<td>UserSwitch:=Off;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t002</td>
<td>Off</td>
<td>Inactive</td>
<td>UserSwitch(x)</td>
<td>x=On</td>
<td>UserSwitch:=On;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t003</td>
<td>Off</td>
<td>Inactive</td>
<td>UserSwitch(x)</td>
<td>x=Off</td>
<td>UserSwitch:=Off;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t004</td>
<td>Inactive</td>
<td>Cruise</td>
<td>UserMode(x)</td>
<td>x=SD &amp; (SlowCutoff&lt;CurrentSpeed&lt;FastCutoff)</td>
<td>TargetSpeed:=CurrentSpeed; TargetThrottle:=Throttle.Position(); Call Gauges.Cruise(On); Call Throttle.Floor(TargetThrottle); UserMode:=SD; Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t005</td>
<td>Cruise</td>
<td>Accel</td>
<td>UserMode(x)</td>
<td>x=RA</td>
<td>TargetSpeed:=TargetSpeed+1; UserMode:=RA; Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t006</td>
<td>Accel</td>
<td>Cruise</td>
<td>UserMode(x)</td>
<td>x=NT &amp; (TargetSpeed&lt;CurrentSpeed&lt;0.5)</td>
<td>UserMode:=NT; Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t007</td>
<td>Cruise</td>
<td>Decel</td>
<td>UserMode(x)</td>
<td>x=SD</td>
<td>TargetSpeed:=TargetSpeed-1; UserMode:=SD; Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t008</td>
<td>Decel</td>
<td>Cruise</td>
<td>UserMode(x)</td>
<td>x=NT &amp; (CurrentSpeed&lt;TargetSpeed&lt;0.5)</td>
<td>true; Call Throttle.Floor(0); Call Gauges.Cruise(Off); UserMode:=NT;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t009</td>
<td>Cruise</td>
<td>Override</td>
<td>Cancel()</td>
<td></td>
<td>Call Throttle.Floor(TargetThrottle); Pause; UserMode:=RA; CurrentSpeed:=Gauges.Speed; Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t010</td>
<td>Override</td>
<td>Accel</td>
<td>UserMode(x)</td>
<td>x=RA &amp; (SlowCutoff&lt;CurrentSpeed&lt;FastCutoff)</td>
<td>Call Throttle.Floor(TargetThrottle); Pause; UserMode:=RA; CurrentSpeed:=Gauges.Speed; Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t011</td>
<td>Accel</td>
<td>Override</td>
<td>Cancel()</td>
<td>true</td>
<td>Call Throttle.Floor(0); Call Gauges.Cruise(Off);</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t012</td>
<td>Decel</td>
<td>Override</td>
<td>Cancel()</td>
<td>true</td>
<td>Call Throttle.Floor(0); Call Gauges.Cruise(Off); UserMode:=NT;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t013</td>
<td>Decel</td>
<td>Override</td>
<td>UserMode(x)</td>
<td>CurrentSpeed&lt;SlowCutoff OR CurrentSpeed&lt;FastCutoff</td>
<td>Call Throttle.Floor(0); Call Gauges.Cruise(Off); UserMode:=NT;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t014</td>
<td>Accel</td>
<td>Inactive</td>
<td>CheckState()</td>
<td>UserMode:=RA &amp; TargetSpeed&lt;FastCutoff</td>
<td>Call Gauges.Cruise(Off); TargetSpeed:=0; UserMode:=NT;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t015</td>
<td>Decel</td>
<td>Inactive</td>
<td>CheckState()</td>
<td>UserMode:=SD &amp; TargetSpeed&lt;FastCutoff</td>
<td>Call Gauges.Cruise(Off); TargetSpeed:=0; UserMode:=NT;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t016</td>
<td>Decel</td>
<td>Off</td>
<td>UserSwitch(x)</td>
<td>x=Off</td>
<td>Call Gauges.Cruise(Off); UserSwitch:=Off;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t017</td>
<td>Decel</td>
<td>Off</td>
<td>UserSwitch(x)</td>
<td>x=Off</td>
<td>Call Gauges.Cruise(Off); UserSwitch:=Off;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t018</td>
<td>Override</td>
<td>Off</td>
<td>UserSwitch(x)</td>
<td>x=Off</td>
<td>Call Gauges.Cruise(Off); UserSwitch:=Off;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t019</td>
<td>Cruise</td>
<td>Off</td>
<td>UserSwitch(x)</td>
<td>x=Off</td>
<td>Call Gauges.Cruise(Off); UserSwitch:=Off;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t020</td>
<td>Inactive</td>
<td>Inactive</td>
<td>SetSpeed()</td>
<td>true</td>
<td>CurrentSpeed:=Gauges.Speed(); UserMode:=NT;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t021</td>
<td>Inactive</td>
<td>Inactive</td>
<td>UserMode(x)</td>
<td>x=SD OR CurrentSpeed&lt;SlowCutoff OR CurrentSpeed&lt;FastCutoff</td>
<td>Call Gauges.Cruise(Off); TargetSpeed:=0; UserMode:=NT;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t022</td>
<td>Cruise</td>
<td>Accel</td>
<td>CheckState()</td>
<td>UserMode:=NT &amp; TargetSpeed&lt;CurrentSpeed&lt;1.0</td>
<td>Call Gauges.Cruise(Off); TargetSpeed:=0; UserMode:=NT;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t023</td>
<td>Accel</td>
<td>Cruise</td>
<td>CheckState()</td>
<td>UserMode:=NT &amp; TargetSpeed&lt;CurrentSpeed&lt;0.5</td>
<td>CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t024</td>
<td>Decel</td>
<td>Cruise</td>
<td>CheckState()</td>
<td>UserMode:=NT &amp; Current Speed&lt;TargetSpeed&lt;1.0</td>
<td>CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t025</td>
<td>Decel</td>
<td>Cruise</td>
<td>CheckState()</td>
<td>UserMode:=NT &amp; Current Speed&lt;TargetSpeed&lt;0.5</td>
<td>CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t026</td>
<td>Cruise</td>
<td>Cruise</td>
<td>CheckState()</td>
<td>UserMode:=NT &amp; ABS(Target Speed&lt;CurrentSpeed)&lt;0.5</td>
<td>Pause; CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t027</td>
<td>Cruise</td>
<td>Cruise</td>
<td>CheckState()</td>
<td>UserMode:=NT &amp; 0.5sABS(Target Speed&lt;CurrentSpeed)&lt;1.0</td>
<td>CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t028</td>
<td>Decel</td>
<td>Decel</td>
<td>CheckState()</td>
<td>UserMode:=NT &amp; Current Speed&lt;TargetSpeed&lt;0.5</td>
<td>Call Throttle.Floor(TargetThrottle.Position()+1); CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t029</td>
<td>Decel</td>
<td>Decel</td>
<td>CheckState()</td>
<td>UserMode:=SD &amp; TargetSpeed&lt;SlowCutoff</td>
<td>Call Throttle.Floor(TargetThrottle.Position()+1); CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;</td>
</tr>
<tr>
<td>CruiseUnit</td>
<td>t030</td>
<td>Accel</td>
<td>Accel</td>
<td>CheckState()</td>
<td>UserMode:=NT &amp; TargetSpeed&lt;CurrentSpeed&lt;0.5</td>
<td>Call Throttle.Floor(TargetThrottle.Position()+1); CurrentSpeed:=Gauges.Speed(); Put CheckState() on Call Queue;</td>
</tr>
</tbody>
</table>
CruiseUnit t031 Accel Accel CheckState() UserMode=RA
Call Throttle.Floor(Throttle.Position()+1); CurrentSpeed=Gauges.Speed(); TargetSpeed=CurrentSpeed; TargetThrottle:=Throttle.Position(); Put CheckState() on Call Queue;

CruiseUnit t032 Override Decel UserMode(x) x=SD & (SlowCutoff<CurrentSpeed< FastCutoff)
TargetSpeed=CurrentSpeed; TargetThrottle:=Throttle.Position(); Call Gauges.Speed(); Call Throttle.Floor (TargetThrottle); UserMode=SD; Put CheckState() on Call Queue;

CruiseUnit t033 Cruise Cruise SetSpeed() true CurrentSpeed=Gauges.Speed();
CruiseUnit t034 Accel Accel SetSpeed() true CurrentSpeed=Gauges.Speed();
CruiseUnit t035 Decel Decel SetSpeed() true CurrentSpeed=Gauges.Speed();
CruiseUnit t036 Override Override SetSpeed() true CurrentSpeed=Gauges.Speed();

CruiseUser t001 Initial Off CruiseUser() true Switch:=Off; Mode:=NT;
CruiseUser t002 Off Neutral Switch(x) x=On Switch:=On; Call CruiseUnit.UserSwitch(On);
CruiseUser t003 Neutral Off Switch(x) x=Off Switch:=Off; Mode:=NT; Call CruiseUnit.UserSwitch(Off);

CruiseUser t004 Neutral Accel Mode(x) x=RA Mode:=RA; Call CruiseUnit.SetSpeed();
CruiseUnit t005 Accel Neutral Mode(x) x=NT Call CruiseUnit.UserMode(NT);
CruiseUser t006 Decel Neutral Mode(x) x=NT Mode:=NT; Call CruiseUnit.UserMode(NT);
CruiseUser t007 Neutral Decel Mode(x) x=SD Mode:=SD; Call CruiseUnit.SetSpeed();
CruiseUser t008 Accel Off Switch(x) x=Off Switch:=Off; Mode:=NT; Call CruiseUnit.Cancel();
CruiseUser t009 Decel Off Switch(x) x=Off Switch:=Off; Mode:=NT; Call CruiseUnit.Cancel();
CruiseUser t010 Neutral Neutral Cancel() true Call CruiseUnit.Cancel();
CruiseUser t016 Accel Accel Cancel() true IgnoreCancel vs CancelPrevail
CruiseUser t019 Decel Decel Cancel() true IgnoreCancel vs CancelPrevail

Engine t002 Warmup Normal Check(x) OilPressure=OilPressure & OilPressure
<OilPMax & WaterTemp<WaterTMin & WaterTemp<WaterTMax & BlockTemp=BlockMin & BlockTemp=BlockMax
Put Check() on Call Queue;

Engine t003 Normal Normal GasFlow(x) true
Rpm:=Rpm+((x-GasFlow(10)*8000; GasFlow=x; Call Gauges.Speed( Rpm/8000)*220);
Put Check() on Call Queue;

GasUser t002 Active Active PedalPosition(x) x>0 & x#PedalPosition
Call Throttle.GasPedal(PedalPosition);

Gauges t015 SpeedCity SpeedCity Speed(x) 20<x*Engine.ExternalDrag()<50 Speed:=x*Engine.ExternalDrag();
Gauges t016 SpeedHwy SpeedHwy Speed(x) 40<x*Engine.ExternalDrag()<130 Speed:=x*Engine.ExternalDrag();
Gauges t017 SpeedCity SpeedCity Speed(x) 50x*Engine.ExternalDrag()<130 Speed:=x*Engine.ExternalDrag();
Gauges t018 SpeedHwy SpeedHwy Speed(x) 100x*Engine.ExternalDrag()<40 Speed:=x*Engine.ExternalDrag();
Gauges t023 CruiseOn CruiseOn Cruise(x) x=On Cruise:=On;
Gauges t024 CruiseOff CruiseOn Cruise(x) x=On Cruise:=Off;

Throttle t001 Initial Idle Throttle() fconst5gconst
Call Engine.GasFlow(Convert(Position)); Floor:=fconst;
Call GasUser.PedalPosition(fconst);
GasPedal:=x; Position:=Min(GasPedal,gconst);
Call Engine.GasFlow(Convert(Position));

Throttle t002 Idle Manual GasPedal(x) x=fconst
Call Engine.GasFlow(Convert(Position)); Floor:=fconst;
Call GasUser.PedalPosition(fconst);

Throttle t003 Manual Idle GasPedal(x) x=fconst & Floor=fconst
Call Engine.GasFlow(Convert(Position));
Call GasUser.PedalPosition(fconst);

Throttle t004 Idle Automatic Floor(x) x=fconst
Floor:=x; Position:=Min(Floor.gconst);
Call Engine.GasFlow(Convert(Position));
Call GasUser.PedalPosition(fconst);

Throttle t005 Automatic Idle Floor(x) x=fconst & GasPedals fconst
Floor:=fconst; Position:=Min(Floor.gconst);
Call Engine.GasFlow(Convert(Position));
Call GasUser.PedalPosition(fconst);

Throttle t006 Manual Automatic GasPedal(x) x=fconst & x<Floor
GasPedal:=x; Position:=Floor;
Call Engine.GasFlow(Convert(Position));
Call GasUser.PedalPosition(Floor);

Throttle t007 Manual Automatic Floor(x) x=fconst & x=GasPedal
Floor:=x; Position:=Min(Floor.gconst);
Call Engine.GasFlow(Convert(Position));
Call GasUser.PedalPosition(Floor);

Throttle t008 Automatic Manual GasPedal(x) x=fconst & x<Floor
GasPedal:=x; Position:=Min(GasPedal.gconst);
Call Engine.GasFlow(Convert(Position));

Throttle t009 Automatic Manual Floor(x) x=fconst & x=GasPedal
Floor:=x; Position:=GasPedal;
Call Engine.GasFlow(Convert(Position));
<table>
<thead>
<tr>
<th>Throttle</th>
<th>t012</th>
<th>Danger</th>
<th>Idle</th>
<th>Position(x)</th>
<th>x=const</th>
<th>Position:=const;</th>
<th>Call Engine.GasFlow(Convert(Position));</th>
<th>Floor:=const;</th>
<th>Call GasUser.PedalPosition(const);</th>
</tr>
</thead>
</table>