Conformance Testing of Object-Oriented Components Specified by State/Transition Classes

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

Abstract

In object-oriented software development, 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 consists of state and behavior, where state is determined by the values of *state variables* identified in the class definition, and behavior is determined by *methods* (i.e. functions or procedures), defined in the class, that operate on one or more object instances to read or modify their state variables. Objects communicate by sending messages to one another, where a *message* is a request to invoke one of the recipient object's methods. Conformance testing of object-oriented software is often done in three stages: *unit testing*, to ensure that the individual methods of a class are properly defined over all possible object instances; *component testing*, to ensure that methods restricted to the component under test operate as specified by rules of how the component should behave and that messages to external objects are properly sent; and *system testing*, to ensure that the entire system behaves as specified by some overall system functional specification. Both component testing and system testing rely heavily on *integration testing*, which ensures 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 new work focuses on integration testing. It is strongly influenced by the work of Hong, Kwon, and Cha, who model a single class as a finite state machine, transform that representation into a data flow graph that explicitly identifies the definitions and uses of each state variable of the class, and then apply conventional data flow coverage criteria and testing techniques to produce a collection of abstract test cases that can be used to test conformance of the given class to its functional specification. This paper extends those ideas to an arbitrary number of classes and components. It introduces flexible representations for message sending and receiving among objects and allows parallel processing among any, or all, classes and components. Our approach relies on finite state machines, database modeling and processing techniques, and algorithms for analysis and traversal of directed graphs. A prototype implementation of the approach demonstrates its effectiveness on non-trivial, real-world problems.

Keywords: (Conformance testing; data flow graph; data modeling; finite state machine; object-oriented; software testing; statistical methods)

1 Introduction

Conformance testing of object-oriented software is a difficult problem because the 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 new components to be integrated into an existing software system.

We assume that each component is object-oriented, that is, it consists of a collection of object classes. In object-oriented software development, 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 consists of state and behavior, where state is determined by the values of *state variables* identified in the class definition, and behavior is determined by *methods* (i.e. functions or procedures), defined in the class, that operate on one or more object instances to read or modify their state variables. The behavior of an object when acted upon by a method is the effect the method has on the state variables of the object.

If the states of an object are represented by a finite state machine, then the effect of the method can be captured as a set of transition rules. Thus finite state machines are often used for class specification in object-oriented analysis and design [3,4,5,14,19]. Objects within each class may pre-exist, as is often the case with off-the-shelf components, or the component interface may provide access to the *constructor* methods of a class for new object creation. The behavior of a component is specified by the behavior of its constituent classes. The public interface to a component is a list of publicly accessible methods from the classes within the component. A *state/transition specification* for a class is the set of state transition rules for each method that is visible in the class's public interface. Given a state/transition specification for each component in a software system, our goal is to construct an abstract set of tests, i.e. an *abstract test suite*, that can be used to construct an *executable test suite* for determining if an implementation of a software system conforms to its functional specification.

We follow the lead of Hong, *et al.* [10] and use definitions from Brooch [2] and Rumbaugh, *et al.* [18] 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. In a finite state machine, a transition is a change of state caused by an event. When an event is received, the next state depends on the current state as well as the event. For a class, a state is a predicate on the state variables of the class, an event is an invocation of one of the class methods on an object instance of the class, and a transition is any change of state caused by the method invocation. For an individual object instance, a *transition* is composed of a source state, a target state, a method applied to that instance, a *guard*, i.e. a condition that must be satisfied before the transition can occur, and an *action*, i.e. operations that reference or manipulate the state variables of the given object or send messages to itself or other objects in the system.

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 [1] could be applied. The paper describes a process that begins with state/transition specifications for each class in an object-oriented software system, defines the relevant transitions for a specific component of that system, translates the relevant transitions into a data flow graph with nodes and edges labeled for variable definition and usage, and concludes with selection of a set of paths that constitute an abstract test suite. Using statistical methods, one can then choose an executable test suite from the abstract test cases for determining, within a given confidence interval, whether a software product conforms to its specification.

In anticipation of the application of this process to moderately large software systems, we define database representations for the structures at each step. 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 translate to database operations. Section 2 presents a non-trivial automobile system as an example, Section 3 gives the initial database representations for state/transition specification of the classes, Section 4 defines the transitions relevant to a given component of the software system, Section 5 describes the associated data flow graph, Section 6 describes variable definition, variable usage, and their representation as tables, Section 7 defines data flow path coverage, and Section 8 describes the selection of abstract test cases. The Conclusions discuss some related work and follow-on directions.

2 Automobile example

Consider an automobile control system. The software consists of the following existing components: Acceleration, Brakes, Clutch, Engine, InstrumentPanel, and SystemControl. Suppose a new component, CruiseControl,

is to be added to the system with the prerequisite that none of the existing components, except SystemControl, can be modified. The only modifications allowed for System Control are the addition of simple methods to notify CruiseControl whenever a Brake or Clutch object becomes active. With the addition of the new CruiseControl component, the Automobile system will consist of the following components and classes.

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

Assume the existence of a state/transition specification for each class. An on-line reference to the classes, variables, methods, states, and transitions of the system is given in the Conclusions section of this paper. Our initial goal is to determine the transitions in each existing class *relevant* to the new component under test, i.e. CruiseControl. In some cases only a few transitions are relevant; for example, the only state of BrakeControl relevant to CruiseControl is whether or not the brakes are engaged. When the brakes are engaged, a message is sent to AutoSystem, and AutoSystem, in turn, sends a message to CruiseUnit.

Figure 1 presents a directed graph that shows 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, e.g. hills, the Engine class has an externally controlled drag variable that enters into the speed calculation.

3 Database representation

From [10], each class C produces a Class State Machine defined as a tuple (V, F, S, T) where V is the set of state variables defined by C, F is the set of methods (i.e. functions) defined by C, S is a set of states associated with C, and T is a set of transitions associated with S and C. Using the relational database model [6,7,15], we choose to represent classes, and sets associated with classes, as tables.

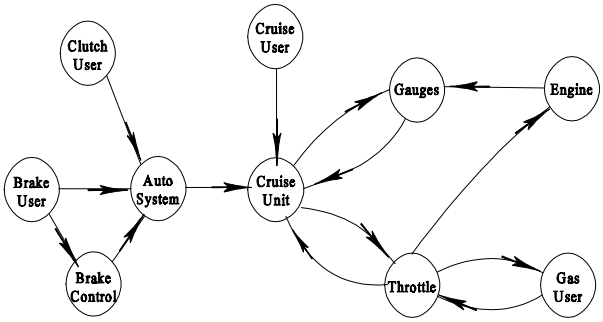

Figure 1 — Class-to-Class Data Flow

The resulting schema definition is shown in Figure 2. The underlined items are *primary keys* for each table. The arrows are *referential integrity* constraints that show class dependencies, e.g. in the Transition table, both the SourceId and the TargetId of a transition must reference a State from the same class as the transition.

Each class is identified by a unique ClassId, which then determines the component and system for that class. Each class is owned by exactly one component even though it may be used by many components. The ClassAlias is a surrogate for ClassId and is used to reference the class in state and guard predicates, and in the Action of a Transition.

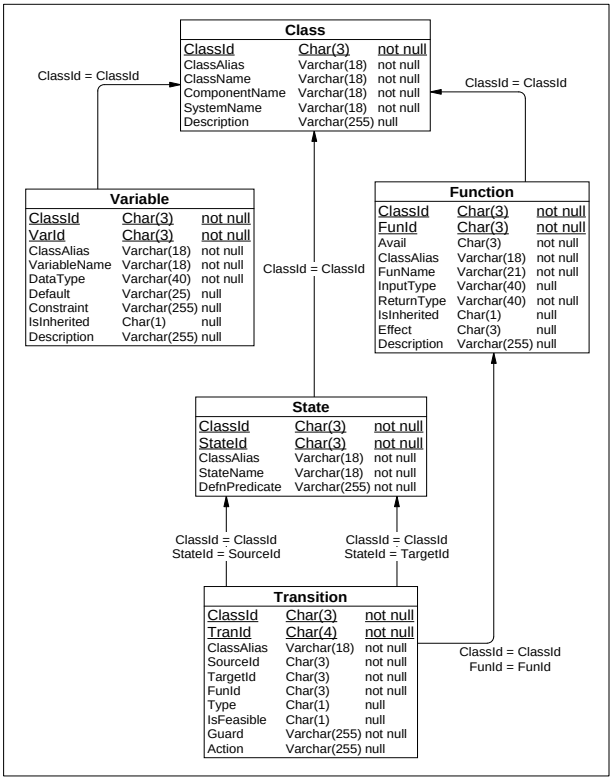

Figure 2 — Relational Database Schema

Similarly, VariableName, FunName, and StateName are surrogates for VarId, FunId, and StateId, respectively; each need be unique only within its class. In the syntax for predicates, guards, and actions, full qualified names are used as needed for disambiguation.

In the Function table, the Avail attribute identifies each function as being private (PRI), protected (PRO), public (PUB), or external (EXT). Public functions may be invoked by any other class in the system, whereas external functions are part of the external component interface and can be invoked by other systems. In our example, only external functions, e.g. clutch and gas pedal positions, are available to the human user. FunName values include parentheses to identify the number of input variables, e.g. PedalPosition(x), so ClassAlias, FunName, InputType, and ReturnType determine the *signature* of a function. The Effect allows categorization of functions as Get, Set, New, etc. 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, and a New function can create new object instances and so is said to be a *constructor* method. If variables or functions are inherited from superclasses, then the IsInherited attribute is set to *yes*.

In the State table, the DefnPredicate is a Boolean predicate over the state variables. It may reference an in-class variable by name only, and may reference a variable in another class by invoking the appropriate actor method, if it is available, to read the value of that external variable. It is not legal to call anything other than an actor method from a state's definition predicate. Mutator and constructor methods may only be called from an Action that is part of a transition. In the automobile example, state predicates for the CruiseUnit class read the value of the cruise control indicator in the Gauges class.

In the Transition table, the TranId attribute uniquely determines a transition within a class. Thus the pair (ClassId,TranId) determines all of the other properties of a transition. In particular, the source and target states are identified by SourceId and TargetId, respectively. The Type attribute is a way to identify transitions that exist but may not be very important, typically they may not even show up in a state/transition diagram because they have no Action, do not affect any state changes, and may never get called by any other methods in the components under analysis. For completeness, they are included, but because they are derived automatically as part of a completeness check, they are labeled as of Derived (D) type. Other transitions may be well-defined, but are blocked from execution by a rule or by physical impossibility. For example, in the automobile system, the cruise control Accel button cannot be pushed at the same time as the Decel button because their physical placement prohibits them from being depressed simultaneously. Such transitions, though well-defined, are labeled as not feasible and the IsFeasible attribute is set to *no*; such transitions may not participate in any data flow analysis.

Later sections of this paper depend upon notions like "the set of variables *referenced* by a predicate" or "the set of variables *assigned a value* by an action". These are semantic notions that depend upon an analysis of whatever syntax is used to represent state predicates, transition guards, and transition actions in the state transition model. In the automobile example, we use a simple syntax that allows these notions to be made concrete without extensive analysis. In subsequent work, we hope to expand this part of our prototype to include syntactic analysis of predicates and actions specified in UML [20], Java [11], or other commonly used class definition languages.

Once this syntactic analysis is complete, the results can be captured in the database representation as the following new tables. For each table, the items in parentheses following the table name are the primary key columns.

Action_DEFINE_Variable
(ClassId, TranId, VarId)
Action_REF_ActorExtFunction
(ActionClassId, TranId, FunClassId, FunId)
Action_REF_LocalFunctionCallQueue
(ClassId, TranId, FunId)
Action_REF_LocalFunctionImmediate
(ClassId, TranId, FunId)
Action_REF_MutateExtFunction
(ActionClassId, TranId, FunClassId, FunId)
Action_REF_Variable
(ClassId, TranId, VarId)
Guard_REF_ActorExtFunction
(GuardClassId, TranId, FunClassId, FunId)
Guard_REF_Variable
(ClassId, TranId, VarId)
State_REF_ActorExtFunction
(StateClassId, StateId, FunClassId, FunId)
State_REF_Variable
(ClassId, StateId, VarId)
Variable_ASSOC_ActorFunction
(ClassId, VarId, FunId)

Each of the above tables satisfies appropriate referential integrity constraints to the corresponding Transition, Variable, Function, or State tables. For example, in the Guard_REF_ActorExtFunction table, the pair (GuardClassId, TranId) identifies the transition that contains the guard, and the pair (FunClassId, FunId) identifies the Get function that is referenced by the guard.

The automobile example uses some special syntax, i.e. Put CheckState() on Call Queue, to distinguish a situation where an object sends a 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. A local function call without this special syntax will be executed immediately as part of the Action. The two Action_REF_ LocalFunction tables above distinguish these two situations.

Every state variable in a class definition will be associated with two pre-defined methods, one to Get its value and one to Set its value. For example, PedalPosition() reads the value of the PedalPosition state variable, whereas PedalPosition(x) sets its value to x. The above table, Variable_ASSOC_ActorFunction, maintains the relationship between a state variable and the Get function that reads its value.

Every table can be associated with a mathematical set, where the set is a set of sequences consisting only of the primary key elements 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 Function.ClassId = c and Function.FunId = f. If X is such a table-derived set, if c is a non-key column of the corresponding table T(X), and if $x \in X$, then we define c(x) to be the value in column c of the row of table T(X) identified by x. We will use this notational convenience freely in the following sections, and will denote by C, F, V, S, and T, respectively, the sets derived from the tables Class, Function, Variable, State, and Transition.

4 Relevant transitions

If M is a component to be added to an existing system, and if our testing goal is to determine if M is properly integrated with existing components of that system, then we must first determine the transitions from the overall system specification that are relevant to M. This will include all transitions from any class in the system that can influence a state variable of any class of M, as well as all transitions that can be invoked, directly or indirectly, from actions derived from any transition of any class of M. The first type of transition is labeled as an *In* transition, since data flows into M, and the second type is labeled an *Out* transition, since actions flow outward from M to other components in the system. Transitions from classes in M are labeled as *Base* transitions. The following definition identifies all *In* and *Out* transitions associated with M, generalizes the notion of *In* and *Out* to transitions that affect any transition already so labeled, and then defines the transitions that are relevant to M.

Definition 4.1 Let Ψ denote an object-oriented software system and let M denote a component of Ψ. Assume that every class of every component of Ψ has a state/transition specification, and that the totality of all classes, functions, variables, states, and transitions of that system are represented by the sets C, F, V, S, and T, defined as above. Denote by R(M) the set of *transitions in* Ψ *that are relevant to* M, defined iteratively as follows:

$$R_0(M) = \{(c,t,\textit{Base}) \mid c \in C \;\&\; \text{ComponentName}(c)=M \;\&\; (c,t) \in T \;\&\; \text{IsFeasible}(c,t)=\textit{yes}\}$$

Given $R_n(M)$, define

$$R_{n+1}(M) = R_n(M) \cup \acute{R}, \text{ where}$$

$$\acute{R} = \acute{R}_1 \cup \acute{R}_2 \cup \acute{R}_3 \cup \acute{R}_4 \cup \acute{R}_5 \cup \acute{R}_6 \cup \acute{R}_7 \cup \acute{R}_8 \cup \acute{R}_9$$

and

$$\acute{R}_1 = \{(c,t,\textit{In}) \mid c \in C \;\&\; (c,t) \in T \;\&\; \text{IsFeasible}(c,t)=\textit{yes} \;\&\; \exists f[(c,f) \in F \;\&\; \text{FunId}(c,t)=f] \;\&\; \exists \acute{c},\acute{t},d[(\acute{c},\acute{t}) \in T \;\&\; (\acute{c},\acute{t},d) \in R_n(M) \;\&\; (\acute{c},\acute{t},c,f) \in X]\}$$

where X represents table Guard_REF_ActorExtFunction,

$$\acute{R}_2 = \{(c,t,\textit{In}) \mid c \in C \;\&\; (c,t) \in T \;\&\; \text{IsFeasible}(c,t)=\textit{yes} \;\&\; \exists f[(c,f) \in F \;\&\; \text{FunId}(c,t)=f] \;\&\; \exists \acute{c},\acute{t},d[(\acute{c},\acute{t}) \in T \;\&\; (\acute{c},\acute{t},d) \in R_n(M) \;\&\; (\acute{c},\acute{t},c,f) \in X]\}$$

where X represents table State_REF_ActorExtFunction,

$$\acute{R}_3 = \{(c,t,\textit{In}) \mid c \in C \;\&\; (c,t) \in T \;\&\; \text{IsFeasible}(c,t)=\textit{yes} \;\&\; \exists f[(c,f) \in F \;\&\; \text{FunId}(c,t)=f] \;\&\; \exists \acute{c},\acute{t},d[(\acute{c},\acute{t}) \in T \;\&\; (\acute{c},\acute{t},d) \in R_n(M) \;\&\; (\acute{c},\acute{t},c,f) \in X]\}$$

where X represents table Action_REF_ActorExtFunction,

$$\acute{R}_4 = \{(c,t,\textit{Out}) \mid c \in C \;\&\; (c,t) \in T \;\&\; \text{IsFeasible}(c,t)=\textit{yes} \;\&\; \exists f[(c,f) \in F \;\&\; \text{FunId}(c,t)=f] \;\&\; \exists \acute{c},\acute{t},d[(\acute{c},\acute{t}) \in T \;\&\; (\acute{c},\acute{t},d) \in R_n(M) \;\&\; d \neq \textit{In} \;\&\; (\acute{c},\acute{t},c,f) \in X]\}$$

where X represents table Action_REF_MutateExtFunction,

$$\acute{R}_5 = \{(c,t,\textit{Out}) \mid c \in C \;\&\; (c,t) \in T \;\&\; \text{IsFeasible}(c,t)=\textit{yes} \;\&\; \exists \acute{c},\acute{t},f,d[(\acute{c},\acute{t}) \in T \;\&\; (\acute{c},\acute{t},d) \in R_n(M) \;\&\; d \neq \textit{In} \;\&\; (\acute{c},f) \in F \;\&\; \text{FunId}(\acute{c},\acute{t})=f \;\&\; (c,t,\acute{c},f) \in X]\}$$

where X represents table Action_REF_ActorExtFunction,

$$\acute{R}_6 = \{(c,t,\textit{In}) \mid c \in C \;\&\; (c,t) \in T \;\&\; \text{IsFeasible}(c,t)=\textit{yes} \;\&\; \exists \acute{c},\acute{t},f,d[(\acute{c},\acute{t}) \in T \;\&\; (\acute{c},\acute{t},d) \in R_n(M) \;\&\; (\acute{c},f) \in F \;\&\; \text{FunId}(\acute{c},\acute{t})=f \;\&\; (c,t,\acute{c},f) \in X]\}$$

where X represents table Action_REF_MutateExtFunction,

$$\acute{R}_7 = \{(c,t,\textit{Out}) \mid c \in C \;\&\; (c,t) \in T \;\&\; \text{IsFeasible}(c,t)=\textit{yes} \;\&\; \exists f[(c,f) \in F \;\&\; \text{FunId}(c,t)=f] \;\&\; \exists \acute{t},d[(c,\acute{t}) \in T \;\&\; (c,\acute{t},d) \in R_n(M)$$

& d≠*In* & (c,t',f)∈X]}

where X represents Action_REF_LocalFunctionCallQueue,

$\acute{R}_8$ = {(c,t,*In*) | c∈C & (c,t)∈T & IsFeasible(c,t)=*yes*
& ∃t',f,d,v[(c,t')∈T & (c,t',d)∈R_n(M) & d=*In*
& (c,f)∈F & FunId(c,t')=f & (c,v)∈V & (c,v,f)∈X
& (c,t,v)∈Y]}

where X represents table Variable_ASSOC_ActorFunction and Y represents the table Action_DEFINE_Variable,

$\acute{R}_9$ = {(c,t,*Out*) | c∈C & (c,t)∈T & IsFeasible(c,t)=*yes*
& ∃f[(c,f)∈F & FunId(c,t)=f]
& ∃t',d,v[(c,t')∈T & (c,t',d)∈R_n(M) & d=*Out*
& (c,v)∈V & (c,v,f)∈X & (c,t',v)∈Y]}

where X represents table Variable_ASSOC_ActorFunction and Y represents the table Action_DEFINE_Variable.

The iterative process continues until $\acute{R} = \varnothing$ for some value of n. Let ń be that value.

R(M) is the projection of $R_{ń}$(M) on its first two terms, i.e.

R(M) = {(c,t) | ∃d[(c,t,d)∈$R_{ń}$(M)}

Clearly the iterative process in Definition 4.1 must stop because each R_n(M) is a finite set, increasing in size as n increases, that is bounded above by the finite set T x {*Base*, *In*, *Out*}. In the automobile example this process stops after the third iteration, yielding 106 relevant transitions.

5 Constructing a Data Flow Graph

In classical testing [5,8,9,12,13,16,17,19,21], a data flow graph is a graphical representation of a program's control structure and the flow of data through that structure. We will construct a data flow graph to represent both the control and data flows of the relevant state/transitions of a component. Our definitions are extensions of those found in [10] and [18].

Definition 5.1 Let M be any component of a software system Ψ, and let R(M) be the set of all transitions in Ψ that are relevant to M. Then the *data flow graph of* M *in* Ψ is a directed graph G=(N,E) with nodes and edges satisfying

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

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

where

N_s = {c⊕s | c∈C & ∃t[(c,t)∈R(M) & (SourceId(c,t)=s OR TargetId(c,t)=s)] }

N_t = {c⊕t | (c,t)∈R(M)}

N_g = {c⊕g⊕t | c∈C & (c,t)∈R(M) & Guard(c,t)≠*true*}

N_u = {c⊕E⊕f | c∈C & ∃t[(c,t)∈R(M) & FunId(c,t)=f
& (c,f)∈F & Avail(c,f)=EXT]}

where ⊕ is the concatenation operator for strings, and

E_{st} = {(n_s,n_t) | n_s∈N_s & n_t∈N_t & ∃c,s,t[n_s=c⊕s & n_t=c⊕t
& SourceId(c,t)=s & Guard(c,t)=*true*] }

E_{sg} = {(n_s,n_g) | n_s∈N_s & n_g∈N_g & ∃c,s,t[n_s=c⊕s & n_g=c⊕g⊕t
& SourceId(c,t)=s & Guard(c,t)≠*true*] }

E_{gt} = {(n_g,n_t) | n_g∈N_g & n_t∈N_t & ∃c,t[n_g=c⊕g⊕t & n_t=c⊕t
& Guard(c,t)≠*true*] }

E_{ts} = {(n_t,n_s) | n_t∈N_t & n_s∈N_s & ∃c,s,t[n_t=c⊕t & n_s=c⊕s
& TargetId(c,t)=s] }

E_{ut} = {(n_u,n_t) | n_u∈N_u & n_t∈N_t & ∃c,f,t[n_u=c⊕E⊕f & n_t=c⊕t
& FunId(c,t)=f] }

E_{gtg} = {(n_t,n_g) | n_t∈N_t & n_g∈N_g
& ∃c_t,t_t,c_g,t_g,f[n_t= c_t⊕t_t & n_g=c_g⊕g⊕t_g
& FunId(c_t,t_t)=f & (c_g,t_g,c_t,f)∈X] }

where X represents table Guard_REF_ActorExtFunction,

E_{sts} = {(n_t,n_s) | n_t∈N_t & n_s∈N_s
& ∃c_t,t_t,f,c_s,s[n_t= c_t⊕t_t & n_s=c_s⊕s
& FunId(c_t,t_t)=f & (c_s,s,c_t,f)∈X] }

where X represents table State_REF_ActorExtFunction,

E_{xts} = {(n_t,n_s) | n_t∈N_t & n_s∈N_s
& ∃c_t,t_t,c_s,t_s,f,s[n_t= c_t⊕t_t & n_s=c_s⊕s
& FunId (c_s,t_s)=f & SourceId(c_s,t_s)=s
& (c_t,t_t,c_s,f)∈X] }

where X represents table Action_REF_MutateExtFunction,

E_{xtt} = {(n_t,$n_{t'}$) | n_t∈N_t & $n_{t'}$∈N_t
& ∃c_t,t_t,$c_{t'}$,$t_{t'}$,f[n_t= c_t⊕t_t & $n_{t'}$=$c_{t'}$⊕$t_{t'}$
& FunId(c_t,t_t)=f & ($c_{t'}$,$t_{t'}$,c_t,f)∈X] }

where X represents table Action_REF_ActorExtFunction,

E_{cts} = {(n_t,n_s) | n_t∈N_t & n_s∈N_s
& ∃c,t_t,t_s,f,s[n_t= c⊕t_t & n_s=c⊕s
& FunId(c,t_s)=f & SourceId(c,t_s)=s

& $(c,t_t,f)\in X]$ }

where X represents Action_REF_LocalFunctionCallQueue.

The external user nodes N_u determine the external interface to the system. In black-box conformance testing, it is only through this interface that one is allowed to provide input values for test cases to determine if component M conforms to its specification. Various combinations of these inputs will produce different paths through the data flow graph (N,E). Our goal is to find appropriate paths through the graph, i.e. *abstract test cases*, to ensure that all aspects of the specification are thoroughly covered, and then to choose input values, i.e. *executable test cases*, to execute those paths.

In the automobile example, the set N_u represents the following user actions:

BrakeUser.IsActive(x)	$x \in$ {*true, false*}
BrakeUser.PedalPressure(x)	$0 \le x \le 99$
ClutchUser.PedalPosition(x)	$0 \le x \le 99$
CruiseUser.Cancel()	
CruiseUser.Mode(x)	$x \in$ { *None, Set/Decel, Resume/Accel*}
CruiseUser.Switch(x)	$x \in$ {*On, Off*}
Engine.ExternalDrag(x)	$-9 \le x \le 9$
GasUser.PedalPosition(x)	$0 \le x \le 99$

The external interface of the automobile example, and the transitions of its CruiseControl component, are modeled on the cruise control characteristics of a 1995 Acura Legend. The data flow graph for CruiseControl, with all of its relevant transitions, consists of 215 nodes and 565 edges. Our subsequent analysis will identify 2716 abstract test cases as paths through this graph. Various combinations of the above input values will produce over 1300 executable test cases, each traversing one or more of the identified paths.

6 Variable definition and usage

Given a data flow graph, there are a number of different criteria, e.g. all_definition, all_uses, all_paths, etc., for determining *coverage* of the graph for testing purposes. These have been discussed and compared extensively in the literature [e.g. 8,16, etc.]. Many researchers have concluded that paths linking the definition of a variable to its first subsequent use provide adequate coverage for most testing purposes. The following definitions pursue this definition-usage criterion for determining coverage of a data flow graph.

Definition 6.1 Let M be any component of a software system Ψ, let R(M) be the set of transitions in Ψ that are relevant to M, and let G=(N,E) be the data flow graph of M in Ψ. Let $\acute{u}=(c,v) \in V$, where V is the set of all variables in Ψ, then:

a) $\acute{u}$ is *defined* at a transition-node $n_t\in N_t$ if n_t is an element of $D(\acute{u})$, where

$$D(\acute{u}) = \{ n_t \mid n_t\in N_t \ \&\ \exists t[n_t=c\oplus t \ \&\ (c,t,v)\in X] \}$$

and X represents the table Action_DEFINE_Variable,

b) $\acute{u}$ is *directly computation-used* at a transition-node $n_t\in N_t$ if n_t is an element of $DCU(\acute{u})$, where

$$DCU(\acute{u}) = \{ n_t \mid n_t\in N_t \ \&\ \exists t[n_t=c\oplus t \ \&\ (c,t,v)\in X] \}$$

and X represents the table Action_REF_Variable,

c) $\acute{u}$ is *indirectly computation-used* at a transition-node $n_t\in N_t$ if n_t is an element of $ICU(\acute{u})$, where

$$ICU(\acute{u}) = \{ n_t \mid n_t\in N_t \ \&\ \exists c_t,t,f[n_t=c_t\oplus t \ \&\ (c_t,t,c,f)\in X \ \&\ (c,v,f)\in Y] \}$$

and X represents the table Action_REF_ActorExtFunction and Y represents table Variable_ASSOC_ActorFunction,

d) $\acute{u}$ is *directly predicate-used* at a state-transition-edge $(n_s,n_t)\in E_{st}$ if (n_s,n_t) is an element of $DPU_{st}(\acute{u})$, where

$$DPU_{st}(\acute{u}) = \{ (n_s,n_t) \mid (n_s,n_t)\in E_{st} \ \&\ \exists s,t[n_s=c\oplus s \ \&\ n_t=c\oplus t \ \&\ (c,s,v)\in X] \}$$

and X represents the table State_REF_Variable,

e) $\acute{u}$ is *indirectly predicate-used* at a state-transition-edge $(n_s,n_t)\in E_{st}$ if (n_s,n_t) is an element of $IPU_{st}(\acute{u})$, where

$$IPU_{st}(\acute{u}) = \{ (n_s,n_t) \mid (n_s,n_t)\in E_{st} \ \&\ \exists c_t,s,t,f[n_s=c_t\oplus s \ \&\ n_t=c_t\oplus t \ \&\ (c,f)\in F \ \&\ (c_t,s,c,f)\in X \ \&\ (c,v,f)\in Y] \}$$

and X represents the table State_REF_ActorExtFunction and Y represents table Variable_ASSOC_ActorFunction,

f) $\acute{u}$ is *directly predicate-used* at a state-guard-edge $(n_s,n_g)\in E_{sg}$ if (n_s,n_g) is an element of $DPU_{sg}(\acute{u})$, where

$$DPU_{sg}(\acute{u}) = \{ (n_s,n_g) \mid (n_s,n_g)\in E_{sg} \ \&\ \exists s,t[n_s=c\oplus s \ \&\ n_g=c\oplus g\oplus t \ \&\ (c,s,v)\in X] \}$$

and X represents the table State_REF_Variable,

g) $\acute{u}$ is *indirectly predicate-used* at a state-guard-edge $(n_s,n_g)\in E_{sg}$ if (n_s,n_g) is an element of $IPU_{sg}(\acute{u})$, where

$$IPU_{sg}(\acute{u}) = \{ (n_s,n_g) \mid (n_s,n_g)\in E_{sg} \ \&\ \exists c_t,s,t,f[n_s=c_t\oplus s \ \&\ n_g=c_t\oplus g\oplus t \ \&\ (c,f)\in F$$

& $(c_t,s,c,f)\in X$ & $(c,v,f)\in Y$] }

and X represents the table State_REF_ActorExtFunction and Y represents table Variable_ASSOC_ActorFunction,

h) ύ is *directly predicate-used* at a guard-transition-edge $(n_g,n_t)\in E_{gt}$ if (n_g,n_t) is an element of $DPU_{gt}(ύ)$, where

$$DPU_{gt}(ύ) = \{ (n_g,n_t) \mid (n_g,n_t)\in E_{gt} \ \& \ \exists t[n_g=c\oplus g\oplus t \ \& \ n_t=c\oplus t \ \& \ (c,t,v)\in X] \}$$

and X represents the table Guard_REF_Variable,

i) ύ is *indirectly predicate-used* at a guard-transition-edge $(n_g,n_t)\in E_{gt}$ if (n_g,n_t) is an element of $IPU_{gt}(ύ)$, where

$$IPU_{gt}(ύ) = \{ (n_g,n_t) \mid (n_g,n_t)\in E_{gt} \ \& \ \exists c_t,t,f[n_t=c_t\oplus t \ \& \ n_g=c_t\oplus g\oplus t \ \& \ (c,f)\in F \ \& \ (c_t,t,c,f)\in X \ \& \ (c,v,f)\in Y] \}$$

and X represents the table Guard_REF_ActorExtFunction and Y represents table Variable_ASSOC_ActorFunction.

Definition 6.2 Let M be any component of a software system Ψ, let R(M) be the set of transitions in Ψ that are relevant to M, and let G=(N,E) be the data flow graph of M in Ψ. Let ύ=(c,v) ∈ V, where V is the set of all variables in Ψ, then the set of all *definition-usage pairs associated with* ύ is denoted by DU(ύ), where

$$DU(ύ) = \{(n_t,ū) \mid n_t \in D(ύ) \ \& \ ū \in CU(ύ) \cup PU(ύ) \}$$

and $CU(ύ) = DCU(ύ) \cup ICU(ύ)$

and $PU(ύ) = DPU_{st}(ύ) \cup IPU_{st}(ύ) \cup DPU_{sg}(ύ) \cup IPU_{sg}(ύ) \cup DPU_{gt}(ύ) \cup IPU_{gt}(ύ)$

Not every variable produces a non-empty set of definition-usage pairs. Some variables, e.g. 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.

We pay special attention 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 last after it is used, then that node may continue to be relevant to other definitions or usages of the variable. We distinguish these cases as follows:

Definition 6.3 Let ύ be a variable that is both defined and used at one or more transition nodes. Denote by DFTU(ύ) the set of all transition nodes in $D(ύ) \cap CU(ύ)$ where ύ is *defined and then used*, and denote by UFDL(ύ) the set of all transition nodes in $D(ύ) \cap CU(ύ)$ where ύ is *used first* before it is defined *or defined last* 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.

The sets DFTU(ύ) and UFDL(ύ) need not be mutually exclusive. A transition involving variable x with an action consisting of the sequence x:=x+1; y:=f(x) would be an element of both sets.

In our database representation, we define new tables to hold values for nodes, edges, variable-defn-nodes, variable-c-usage, variable-p-usage, and variable-to-defn-usage pairs. The set-to-table relationship is:

Set	Table Name	Primary Key
N	Nodes	NodeId
E	Edges	SourceNode TargetNode
D(ύ)	VarDefn	ClassId VarId DefnNode
CU(ύ)	Var_C_Usage	ClassId VarId UseNode
PU(ύ)	Var_P_Usage	ClassId VarId SourceNode TargetNode
DU(ύ)	VarDefnUsage	ClassId VarId DefnNode UsageItem

The relational database schema definition for these new tables is given in Figure 3. In the Nodes table, the NodeId is derived from the three other non-comment attributes as follows: If NodeType is *State* or *Transition*, then NodeId = ClassId ⊕ TypeId where TypeId is the corresponding StateId or TranId and ⊕ is the string concatenation operator. If NodeType is *Guard*, then NodeId = ClassId ⊕ 'g' ⊕ TypeId where TypeId is the TranId of the transition that contains the guard. If NodeType is *ExternalUser* then NodeId = ClassId ⊕ 'E' ⊕ TypeId where TypeId is the FunId of a function having EXT availability.

The VarDefn table associates variables with a definition node, i.e. a node that assigns a value to the variable. The Var_C_Usage table associates variables with transition nodes having an action that reads the variable, either directly in its own class or indirectly via an external

function call; these are called *computation nodes* (c-nodes) because the variable is used in a computation. The variable may also be defined by the same action, but that association is recorded in the VarDefn table.

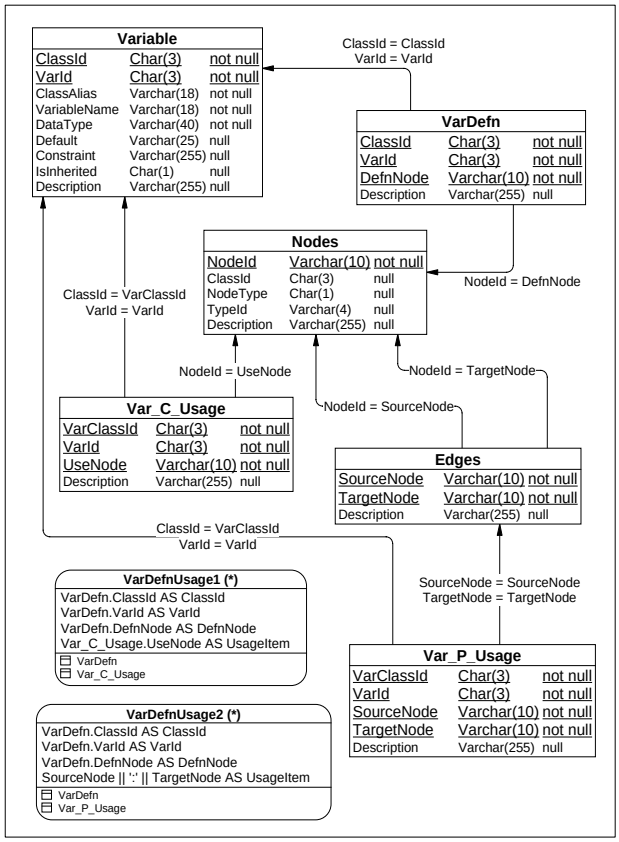

Figure 3 — Schema for Nodes, Edges and Usage

The Var_P_Usage table associates variables with edges where the source node of the edge is a state or a guard that reads the variable; these are called *predicate edges* (p-edges) because the variable is used in a predicate. We use edges to record this usage, rather than just a state or guard with the actual predicate, because our primary goal is to associate data members of a class (i.e. the variables) with their behavior (i.e. the functions of a transition).

Finally, VarDefnUsage1 is a natural join of VarDefn and Var_C_Usage, and VarDefnUsage2 is a natural join of VarDefn and Var_P_Usage; Their mathematical union is the table VarDefnUsage that represents the sets DU(ύ) of all potential definition/usage pairs for any variable. Note that when the UsageItem is a node, it is equal to the NodeId, but when it is an edge it is equal to a concatenation of the edge's SourceNode, a separator ":", and the edge's TargetNode. This makes a p-edge usage item a path of length 2 in the data flow graph (cf. Defn 7.1).

The above definitions do not explicitly consider cases where multiple object instances of a class exist simultaneously. In such situations, each object instance will be represented by a copy of the state variables from the class definition and the state of each object will be separately considered. In static applications, e.g. four wheels in the automobile example, the object instances are known before program execution, so each instance can be treated as if it were from a separate class. However, in dynamic applications, with class variables that are themselves objects or object references, and with methods for *ad hoc* creation and destruction of object instances, some of the preceding definitions will be modified to deal with object identifiers rather than class identifiers. This dynamic case will receive further attention in follow-on work.

7 Data flow path coverage

To complete the definition-usage approach to abstract test case creation, we look for paths in the data flow graph leading from the definition of a variable to its first usage. Consider triples $(ύ,n_t,ū)$ where ύ is a variable, n_t is a transition node that defines ύ, and ū is a usage item for ύ, i.e. ū is either a c-node or a p-edge. Does there exist a path in the directed graph leading from n_t to ū ; and if a path exists, is it free of loops, and does it avoid any modification of the variable by some other transition? The definitions of this section clarify these criteria as applied to conformance testing of object components, and lead to a rigorous definition of *abstract test cases*.

Definition 7.1 Let G=(N,E) be any directed graph. A *path*, p, in G *of length* $k \geq 1$ is any element of N^k satisfying $(n_i,n_{i+1}) \in E$ 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 length of 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, then n is said to be an element of p, denoted by $n \in p$, if n is a member of the sequence that determines p.

Definition 7.2 Let M be any component of a software system Ψ, let G=(N,E) be the data flow graph of M in Ψ, and let VDU be the set of triples $(ύ,n_t,ū)$ that represent the table VarDefnUsage. Let P = $\{(ύ,n_t,ū,p)\}$ denote a set of tuples with $(ύ,n_t,ū) \in$ VDU and with p a path from n_t to ū. The set P is defined iteratively as follows:

$$P_1 = \{(ύ,n_t,n_t,n_t) \mid (ύ,n_t,n_t) \in \text{VDU} \ \& \ n_t \in \text{DFTU}(ύ)\}$$

$$P_2 = \{(ύ,n_t,n_{t'},n_t:n_{t'}) \mid (ύ,n_t,n_{t'}) \in \text{VDU} \ \& \ (n_t,n_{t'}) \in E_{xtt}\}$$

Each P_i will be a set of paths of length i or i-1. The definition of P_i for $i \geq 3$ depends upon sets of *partial paths*, Q_k, and *unresolved defn-usage pairs*, X_j, both defined iteratively below. Each Q_k will be a tuple $(ύ,n_t,ū,h,t)$ where

$(ύ,n_t,ū)\in$VDU, h is a path from a defn node n_t of ύ to an intermediate node, and t is a path from some other intermediate node to a usage item ū for ύ. Each X_j will be a subset of VDU, consisting of variable and defn-usage pairs that are still in search of a connecting path. Begin with

$Q_1 = \{(ύ,n_t,ū,n_t,ū) \mid (ύ,n_t,ū)\in VDU \}$

$X_1 = VDU$

$X_2 = VDU - \{(ύ,n_t,n_t) \mid (ύ,n_t,n_t,n_t)\in P_1 \ \& \ n_t \notin UFDL(ύ) \}$

and given Q_i define

$P_{i+2} = \{(ύ,n_t,ū,h:t) \mid (ύ,n_t,ū,h,t)\in Q_i \ \& \ (T(h),H(t))\in E \}$

$A_{i+2} = \{(ύ,n_t,ū) \mid \exists h,t[(ύ,n_t,ū,h,t)\in Q_i] \}$

$C_{i+2} = \{(ύ,n_t,ū) \mid \exists p[(ύ,n_t,ū,p)\in P_{i+2}] \}$

$X_{i+2} = A_{i+2} - C_{i+2}$

$B_{i+2} = X_{i+1} - A_{i+2}$

and given Q_{2k-1} define

$Q_{2k} = \{(ύ,n_t,ū,h:n,t) \mid (ύ,n_t,ū,h,t)\in Q_{2k-1}$
$\& \ \exists n[n\in N \ \& \ (T(h),n)\in E \ \& \ n\notin D(ύ)$
$\& \ n\notin h \ \& \ n\notin t \ \& \ (ύ,n_t,ū)\in X_{2k+1}] \}$

and given Q_{2k} define

$Q_{2k+1} = \{(ύ,n_t,ū,h,n:t) \mid (ύ,n_t,ū,h,t)\in Q_{2k}$
$\& \ \exists n[n \in N \ \& \ (n,H(t))\in E \ \& \ n\notin D(ύ)$
$\& \ n\notin h \ \& \ n\notin t \ \& \ (ύ,n_t,ū)\in X_{2k+2}] \}.$

The iterative process stops when $X_{i+2} = \varnothing$. At this point set $P = P_{i+2}$. This must happen for some value of i less than the number of nodes in the graph since the generated paths in P_{i+2} , each of length greater than i, have no cycles.

It follows from Definition 7.2 that all generated paths (of length > 1) for some triple $(ύ,n_t,ū)$ will be of the same length. This is because if paths are found in step P_{i+2}, then by the definition of C_{i+2} and X_{i+2} a triple $(ύ,n_t,ū)$ associated with any of those paths is removed from further consideration. Because of the special nature of P_1, some triples $(ύ,n_t,ū)$ will have a path of length 1 in addition to the other paths.

Not all elements $(ύ,n_t,ū)\in$VDU will yield a path in P. Some variables may be defined at a node n_t and used at a usage item ū, but either no path exists from n_t to ū, or every such path contains a re-definition of ύ.

Definition 7.3 A variable ύ is said to be *definition bound* at a definition node n_t of a defn-usage pair $(n_t,ū)\in DU(ύ)$ if there does not exist a path, p, with $(ύ,n_t,ū,p)\in P$.

The definition bound variables surface during the calculation of $B_{i+2} = X_{i+1} - A_{i+2}$ in the iterative process of definition 7.2. At that point we have $C_{i+2} \subseteq A_{i+2} \subseteq X_{i+1}$. It follows that B_{i+2} identifies the defn-usage pairs that were active during the calculation of X_{i+1}, did not find a path to join in P_{i+2}, yet are no longer active for X_{i+2}. 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+2} identify new definition bound items, if they exist, at each step of the process.

In the automobile example, of the 3167 triples $(ύ,n_t,ū)$ satisfying $(n_t,ū)\in DU(ύ)$, 454 are definition bound, and the remaining 2713 have a path in P, and 3 items have a path of length 1 in addition to their longer paths. Most paths are of length 9 or less and can be generated in 15-20 minutes on a PC. Only 62 pairs have paths longer than 9, but it takes an additional 90 minutes of processing time on a 300mHz PC to find 544 paths of length 11 to link 22 pairs, and 2560 paths of length 14 to link the remaining 40 pairs. It took consideration of partial-paths up to length 16 to prove that 92 pairs were definition bound and partial-paths of length 19 to resolve the last 10 remaining pairs as also being definition bound.

A follow-on goal of this research is to exploit the unique structure of a data flow graph for object components, e.g. categorization of nodes by component, or by class, to discover processing shortcuts.

8 Abstract and Executable Test Suites

If a variable ύ is both defined and used, and is not definition bound, then the path generation of the previous section produces one or more abstract test cases linking each definition node n_t to its corresponding usage item ū. But the data flow, definition-usage testing criterion only requires one path per definition-usage pair. For test paths of length > 1, the generation process ensures that all generated paths in P from n_t to ū will have the same length as the shortest path from n_t to ū. We simply need to select one such path in an arbitrary fashion. This is done by using the Group By operator in the relational table representing P, grouping by ύ, n_t, ū, and L, where L is the common length of the generated paths in P, and then selecting an arbitrary element from each group.

Definition 8.1 Let M be any component of a software system Ψ, let $(ύ,n_t,ū)$ be any variable-defn-usage triple generated from the data flow graph of M in Ψ, and let P be generated as in Defn 7.2. An *abstract test suite* for M, denoted by ATS(M) is the set defined by

$$ATS(M) = \{ (\acute{v},n_t,\bar{u},L(p),p) \mid (\acute{v},n_t,\bar{u},p) \in P \ \&\ L(p)=1 \text{ OR } p=SelectOne(\{p \mid (\acute{v},n_t,\bar{u},p) \in P \ \&\ L(p)>1\})$$

Each abstract test in ATS(M) is equally important, because it tests an independent aspect of the state/transition specification for M and its other related components. Some of these paths are *subsumed* by other 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, but they should still be counted as separate tests! In any statistical analysis of test case development, we will assume that these test cases are the sample space from which all executable test cases are drawn. We will pursue the creation of an executable test suite, using statistical methods, as a follow-on activity.

In our database representation we create new tables to represent the set P constructed in Defn. 7.2 and the set of abstract tests in the abstract test suite ATS(M). A relational schema representation is given in Figure 4. Because it is important to retain the identity of each abstract test case in any subsequent executable test case development, we create a special table, AbstractTests, that maintains the relationship between a variable-defn-usage triple and one or more test case Id's. In the AbstractTestSuite table, the UsageType column identifies the usage item associated with the path to be either a c-node (N) or a p-edge (E). In the automobile example, this table identifies 2716 abstract test cases, of which 207 represent paths terminating at a c-node and 2509 represent paths terminating at a p-edge.

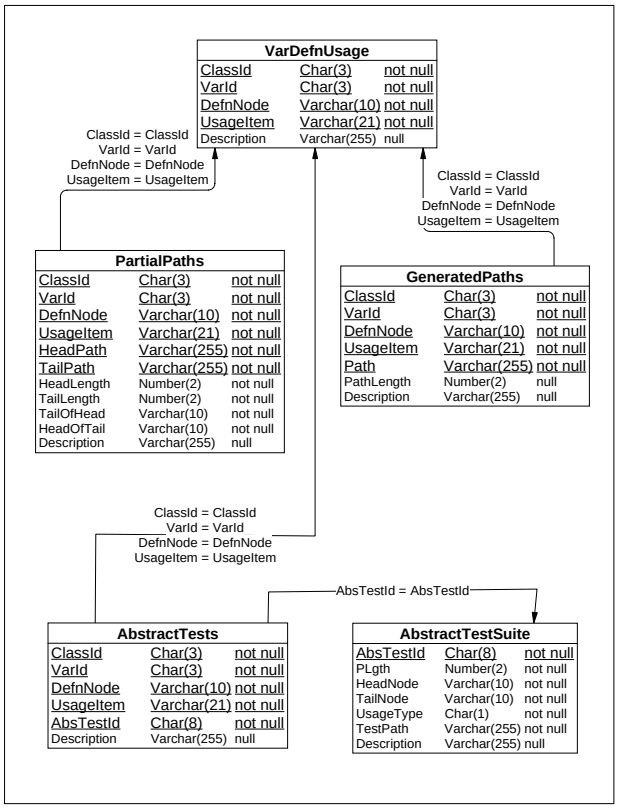

Figure 4 — Schema for Paths and Testing

Conclusions

This paper presents a process for deriving an abstract test suite from a state/transition specification of components in an object-oriented software system. The abstract tests are then suitable for conformance testing of an individual component of the complete system. The abstract test suite also provides a suitable sample space for application of statistical methods to select an executable test suite for actual conformance testing and to determine, within a given confidence interval, whether a software product conforms to its specification.

With the increasing popularity of object-oriented specification methods, e.g. UML [20], 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 the addition of database tools and statistical methods, it becomes possible to apply finite state analysis and testing methods to moderate-sized software systems. Our follow-on work will focus on further integration of the specification and testing aspects of software development and on the application of statistical methods.

The automobile example described in section 2 and referenced throughout this paper is specified via classes, functions, variables, states, and transitions contained in a Microsoft Access database file available via FTP protocols at URL ftp://sdct-sunsrv1.ncsl.nist.gov/stsm/auto/autoslim.mdb. The intermediate tables, and all of the SQL database statements to generate the relevant transitions, the data flow graph, the defn-usage pairs, the partial paths, and the abstract test suite are in a larger database file at URL ftp://sdct-sunsrv1.ncsl.nist.gov/stsm/auto/autosys.mdb.

References

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

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

[3] B. Bosik and M.Ü. Uyar; "Finite State Machine Based Formal Methods in Protocol Conformance Testing," Computer Networks and ISDN Systems, 22, 1991, pp. 7-33.

[4] D. Champeaux, D. Lea, and P. Faure; *Object Oriented System Development*, Addison Wesley, 1993.

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

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

[7] C.J. Date; *An Introduction to Database Systems*, 5th edition, Addison-Wesley, 1990.

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

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

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

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

[12] X. Jia; "Model-Based Formal Specification Directed Testing of Abstract Data Types," in *Proceedings of Computer Software and Applications Conference*, 1993, pp. 360-366.

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

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

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

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

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

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

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

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

[21] S. Zweben, W. Heym, and J. Kimich; "Systematic Testing of Data Abstractions Based on Software Specifications," *Journal of Software Testing, Verification, and Reliability*, 1, 1992, pp. 39-55.

Leonard Gallagher is a computer scientist in the Standards and Conformance Testing Group of the Software Diagnostics and Conformance Testing Division at the U.S. National Institute of Standards and Technology. He has been responsible for data models and integration of database technology with new approaches to information management, including knowledge-based systems, object-oriented software, and multimedia. He was an early leader in the development and testing of Database Language SQL. Dr. Gallagher received the BA degree in mathematics from St. John's University of Minnesota in 1965 and the PhD in mathematics from the University of Colorado in 1972. He taught mathematics at the Catholic University of America for several years and has been involved in database and software research at NIST for the past 20-plus years.