Test cases generation from UML state diagrams

Y.G.Kim, H.S.Hong, D.H.Bae and S.D.Cha

Abstract: The paper discusses the application of state diagrams in UML to class testing. A set of coverage criteria is proposed based on control and data flow in UML state diagrams and it is shown how to generate test cases satisfying these criteria from UML state diagrams. First, control flow is identified by transforming UML state diagrams into extended finite state machines (EFSMs). The hierarchical and concurrent structure of states is flattened and broadcast communications are eliminated in the resulting EFSMs. Second, data flow is identified by transforming EFSMs into flow graphs to which conventional data flow analysis techniques can be applied.

1 Introduction

Classes encapsulate both data (attributes) and procedures (member functions, methods) and are the basic building blocks in object-oriented software development. In addition, a class is often considered to be a basic unit of testing in the object-oriented testing literature. Several research efforts have been made at the systematic testing of classes and many of them are specification based using algebraic specifications or model-based specifications. They normally involve the generation of test cases as sequences of messages from the specifications [1–5]. Recently, several researchers have proposed using finite state machines (FSMs) in class testing [6–9].

This paper discusses the application of state diagrams in Unified Modeling Language (UML) [10] to class testing. UML combines the three popular approaches of Booch [11], Rumbaugh [12], and Jacobson [13] and has been accepted by the OMG as an industry standard for object-oriented analysis and design notation. It comprises a number of diagrams used to describe different aspects of a system including static, dynamic, and use-case views. Among them, this paper focuses on test cases generation from state diagrams in UML. UML state diagrams are widely used for specifying the dynamic behaviour of classes and are substantially based on Statecharts [14] which have been successfully applied to reactive systems. UML state diagrams provide several concepts that distinguish themselves from conventional FSMs. These include the hierarchical and concurrent structure of states, the communication mechanism through events broadcasting, and the actions associated with states and transitions.

An integral part of class testing is the construction of test cases as sequences of messages from a given specification. Because there exists an infinite number of possible sequences of messages, exhaustive testing is impossible to achieve and we need to have systematic coverage criteria which select a reasonable number of message sequences satisfying certain conditions. In this paper, a method is presented that involves the application of conventional control and data flow analysis techniques to the generation of test cases from UML state diagrams.

After describing work related to class testing, we briefly review the syntax and semantics of UML state diagrams. As our main result, we show how UML state diagrams can be transformed into a form to which conventional flow analysis techniques can be applied. A method is given that generates test cases based on control flow in UML state diagrams. We transform UML state diagrams into extended FSMs (EFSMs) to flatten the hierarchical and concurrent structure of states and eliminate broadcast communications in UML state diagrams. Control flow in UML state diagrams is identified in terms of the paths in the resulting EFSMs. We then transform EFSMs into flow graphs. All the associations between definitions and uses employed in UML state diagrams can be identified in the resulting flow graphs. The transformation enables us to apply conventional data flow analysis techniques to the generation of test cases based on data flow in UML state diagrams.

2 Related work

Flow analysis has been extensively used in conventional program testing and analysis. Flow graphs are often used as a graphical representation of a program's structure. The nodes of a flow graph are a block of statements and the edges indicate possible flow of control between nodes. Based on flow graphs, control flow analysis encodes pertinent and possible program flow of control and data flow analysis ascertains and collects information about the possible modification, preservation, and use of variables in a computer program [15]. When applied to specifications rather than programs, they can also provide useful information for the specifications. In protocol conformance testing, several researchers have proposed the use of flow analysis for the systematic generation of test cases from FSM-based specifications [16]. Recently, the idea of specification slicing, which is a generalisation of program slicing, was introduced in [17] and refined and extended in [18, 19]. In specification slicing, flow analysis is done on specifications to identify dependencies among entities in

© IEE, 1999

IEE Proceedings online no. 19990602

DOI: 10.1049/ip-sen:19990602

Paper first received 20th November 1998 and in revised form 7th June 1999

The authors are with the Department of Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 373-1, Kusong-dong, Yusong-gu 305-701, Taejon, Korea
E-mail: ygkim@salmosa.kaist.ac.kr

the specifications. All these applications show that flow information in specifications can be effectively used in validating and debugging specifications and generating test cases from specifications.

In the last two decades, a number of techniques have been proposed for class testing and many of them model classes using abstract data types using algebraic specifications or model-based specifications. Algebraic specifications consist of signatures defining the syntactic properties and axioms describing the properties of member functions. Model-based specifications describe the precondition and postcondition of each member function using well defined mathematical models such as functions, sets, and sequences. In [1–3], test cases are generated as sequences of member functions based on the axioms in algebraic specifications. Since member functions are treated as a mathematical mapping without side effects in algebraic specifications, interaction between attributes and member functions cannot be explicitly tested in these approaches. Zweben *et al.* [5] applied conventional flow graph based testing techniques to class testing. In their approach, a flow graph is associated with each class using model-based specifications. A node in the flow graph represents a member function and an edge between node A and B means that it is permissible to invoke A followed by B. Determining whether an edge exists is based on the precondition and postcondition of each member function. Then test cases are generated based on control and data flow in the resulting flow graphs. Parrish *et al.* [4] extended Zweben *et al.*'s work so that a flow graph can be obtained with or without specifications. When, in the absence of specifications, there exists an edge for every pair of nodes then all sequences of messages are assumed to be feasible.

In general, FSM models of software abound in the testing literature [20]. Since Chow's proposal [21], various FSM-based testing techniques have been proposed especially in protocol conformance testing [16, 22]. Formal description techniques such as SDL, Estelle, and Lotos have been extensively used for specifying communication protocols and a great deal of attention has been given to the generation of test cases based on control and data flow in these specifications. However, the application of FSMs to class testing is a relatively new concept [6–9, 23]. The state-based approaches to class testing concentrate on the interaction occurring between the attributes and the member functions of classes. They show that FSMs can be effectively used to test this interaction by representing the values of attributes as the states of FSMs and the member functions as the transitions of FSMs. In [6, 8, 9, 23], they considered state machines in which transitions are associated with an enabling predicate and an action and generated test cases based on control flow [8, 9, 23] and data flow [6]. This paper extends this work by applying UML state diagrams, which have several distinguishing features from FSMs (in addition to the FSM features), to class testing. Kung *et al.* [7] proposed a class testing technique using a variation of Statecharts, called object state model (OSD). They extract OSD directly from source codes and generate test cases by constructing a spanning tree from OSD. In contrast, we regard UML state diagrams as class specifications and propose a hierarchy of coverage criteria for UML state diagrams based on control and data flow information.

3 UML state diagrams

State diagrams in UML are used to show the states an object can have during its life, and the events that cause the state to change along with its responses. The notation and semantics of UML state diagrams are substantially based on Statecharts modified to include object-oriented features. Fig. 1 shows a UML state diagram for a simple coffee vending machine where m is a synonym for *money* and l for *light*.

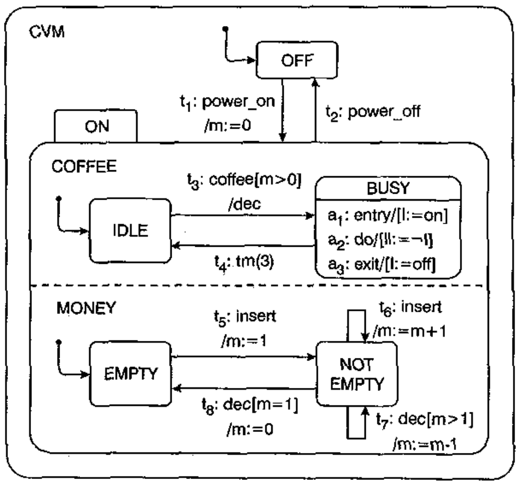


Fig. 1 *Example of a UML state diagram*

The set of states in UML state diagrams represents both basic states and composite states which contain other states as substates. A composite state is classified as either an *or*-state or an *and*-state. An *or*-state has substates that are related to each other by an exclusive-or-relation. In Fig. 1, the state CVM consists of OFF and ON with OFF as the default state. Being in CVM implies being in OFF or in ON, but not in both. An *and*-state has substates, called orthogonal components. Being in the *and*-state ON implies being in COFFEE and MONEY simultaneously. States can have actions associated with them. Actions are performed in response to events received while the object is in the state, without changing the state. Three reserved events are used in the action compartment: *entry*, *exit*, and *do*. The entry event (resp. exit event) is used to specify actions performed at the entry of a state (resp. on exit from a state). The do event is used to specify an action performed while in a given state. For example, the state BUSY has a_1, a_2, and a_3 as entry, do, and exit actions, respectively.

Transitions in UML state diagrams are represented by arrows between states and are labelled by *event* [*guard*]/ *action send*. We classify events into external events that are generated by the environment of an object, i.e. other objects, and internal events that are generated by the object itself. In Fig. 1, the event *dec* is assumed to be internal and all the other events are external. A special event, *tm*(*interval*), is used to represent the passage of a period of time. *Guard* is a Boolean expression that must be satisfied for the transition to occur. *Action* is a list of operations executed as a result of the transition being taken and is assumed to be atomic. *Send* is a list of events generated when the transition is fired.

The semantics of UML state diagrams is based on the notion of steps. External events generated by the environment of an object are accepted by an events queue. The semantics assumes that the events in the queue are processed in sequence one at a time. Once an event is dispatched, one or multiple transitions may be enabled. The state machine selects and fires a maximal set of enabled transitions that are mutually nonconflicting. This basic transformation is called a step. Actions that result from taking a transition may cause events to be generated for this and other objects. Events for the object are broadcast within the present state diagram. After a previous step has completed, the next external event in the queue is dispatched to the state diagram.

We will use the following notation, largely based on [24]. Let *States* and *Trans* be the finite set of states and transitions in UML state diagrams, respectively. Let ρ:*States* $\rightarrow 2^{States}$ be the hierarchy function which gives, for each state, the set of its substates. For a state $s \in States$, $\rho^*(s)$ denotes the least set $S \subseteq States$ such that $s \in S$ and $\rho(s') \in S$ for all $s' \in S$, and $\rho^+(s)$ denotes the set $\rho^*(s) - \{s\}$. If $s_1 \in \rho^*(s_2)$, then we say that s_1 is a *descendant* of s_2, and s_2 is called an *ancestor* of s_1, and that s_1 and s_2 are *ancestrally related.* If, in addition, $s_1 \neq s_2$, then s_1 is a *strict descendant* of s_2 and s_2 is a *strict ancestor* of s_1. There is a unique state $r \in States$, such that $\rho^*(r) = States$, called the *root* state. For a state $s \in States$, we use *entry*(s), *do*(s), and *exit*(s) to denote the entry, do, and exit action associated with s, respectively.

A *configuration* is the maximal set of states which a system can be in simultaneously. Precisely, $C \subseteq States$ is called a configuration if (i) C contains the root state; (ii) for every *and*-state, s, either s and all substates of s are in C, or they are all not in C; (iii) for every *or*-state s, either s and exactly one substate of s are in C, or s and all substates of s are not in C. The *default completion* (if it exists) of a set of states $S \in States$, denoted by *complete*(S), is defined as the configuration C containing S such that, for every *or*-state $s \in C$ such that s is not a strict ancestor of S, the default substate of s is also in C.

For a transition $t \in Trans$, we use *source*(t) and *target*(t) to denote the set of source and target states of t, respectively. We use *event*(t), *guard*(t), *action*(t), and *send*(t) to denote the components of the transition's label. The *scope* of a transition t, denoted by *scope*(t), is defined as an *or*-state such that *scope*(t) is a strict ancestor of all the states in *source*(t) $\cup$ *target*(t), and every such *or*-state is an ancestor of *scope*(t). We say that two transitions *conflict* if their scopes are ancestrally related.

4 Generating test cases based on control flow

To flatten the hierarchical and concurrent structure of states and eliminate broadcast communications in UML state diagrams, EFSMs are used as intermediate forms in the transformation. An EFSM is a tuple $\langle GStates, C_0, GTrans\rangle$ where *GStates* is a set of *global states*, $C_0 \in GStates$ is the *initial global state*, and *GTrans* is a set of *global transitions*. From a given UML state diagram, we identify an EFSM as follows. First, the set of global states *GStates* corresponds to the set of configurations in UML state diagrams. For example, there are five configurations in Fig. 1 and these constitute the global states of the EFSM in Fig. 2

For the formal description of global transitions, we adopt the following notation.

- The *greatest departing state* of a transition t, denoted by *gds*(t), is a state s such that $source(t) \subseteq \rho^*(s)$, $target(t) \not\subseteq \rho^*(s)$, and every such state is a descendant of s. The *departing states* of t, denoted by *DS*(t), is defined as a set of states $\rho^*(gds(t))$.
- The *greatest arriving state* of a transition t denoted by *gas*(t), is a state s such that $source(t) \not\subseteq \rho^*(s)$, $target(t) \subseteq \rho^*(s)$, and every such state is a descendant of

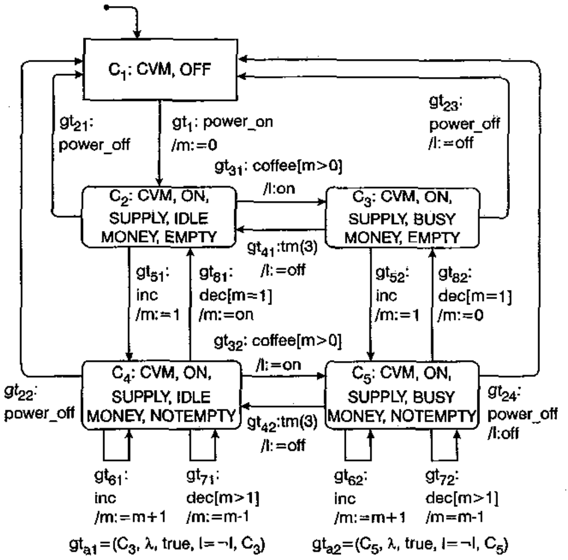


Fig. 2 *Example of an EFSM*

s. The *arriving states* of t, denoted by *AS*(t), is defined as a set of states $\rho^*(gas(t)) \cap complete(target(t))$.

For example, $gds(t_1) = \text{OFF}$, $DS(t_1) = \{\text{OFF}\}$, $gas(t_1) = \text{ON}$, and $AS(t_1) = \{\text{ON, COFFEE, IDLE, MONEY, EMPTY}\}$.

- Let C be a configuration and e be an event. We say that a transition t is *enabled* in C with the event e, if $source(t) \subseteq C$ and $event(t) = e$. We say that a set of transitions T is enabled in C with the event e if every transition in T is enabled and no two transitions in T conflict. In addition, we say that T is *maximal* if every transition not in T but enabled in C conflicts with some transition in T.

A global transition gt in an EFSM is a tuple (C, e, g, a, C') such that $C, C' \in GStates$ and there exists a set of transitions $T \subseteq Trans$ in UML state diagrams satisfying:

- T is enabled in C with the event e and is maximal.
- $C' = (C - \bigcup_{t \in T} DS(t)) \cup \bigcup_{t \in T} AS(t)$.
- $g = \bigcup_{t \in T} guard(t)$.
- $a = \bigcup_{t \in T} A_1(t) \cup A_2(t) \cup A_3(t)$, where $A_1(t) = \bigcup_{s \in DS(t) s \in C} exit(s)$, $A_2(t) = action(t)$, and $A_3(t) = \bigcup_{s \in AS(t) s \in C'} entry(s)$. Note that $A_1(t)$ is the set of all exit actions executed by the occurrence of transition t. Similarly, $A_3(t)$ is the set of all entry actions executed by the occurrence of transition t.

We say that a global transition represents a set of transitions T in UML state diagrams. Intuitively, a global transition is a set of transitions that are executed by the occurrence of one event. For example, consider the event *power-on* and the configuration $C_1 = \{\text{CVM, OFF}\}$ in Fig. 1. We have a global transition $gt_1 = (C_1, \textit{power-on}, \textit{true}, \textit{money} = 0, C_2)$ in Fig. 2. For another example, consider the event *power-off* and the configuration C_3. We have a global transition $gt_{23} = (C_3, \textit{power-off}, \textit{true}, \textit{light} = \textit{off}, C_1)$ because t_2 exists from the state BUSY whose exit action is *light* = *off*.

The entry and exit actions associated in UML state diagrams are included in the actions of global transitions in EFSMs. Now we introduce a special type of global transitions that represents the behaviour of do actions in UML state diagrams. For each global state C, we have a global transition $gt_a = (C_1, e, g, a, C_2)$ such that

- $C_1 = C_2 = C$.
- $e = \lambda$, where λ is the null event.
- $g = true$.
- $a = \bigcup_{s \in C} do(s)$, i.e. where a is the set of all do actions in C.

The global transition gt_a means that the set of do actions in a global state C is performed repeatedly as long as the class is in the global state. For example, we have two global transitions gt_{a1} and gt_{a2} for the global states C_3 and C_5, because C_3 and C_5 include the state BUSY whose do action is $light = light$

Remark 1. (*Enabledness*) When simulating or generating reachability graphs using UML state diagrams, the enabledness should be determined by the current values of the variables as well as the current configuration. However, we do not consider the values of the variables because our work centres on identifying possible control and data flow.

Remark 2. (*Ignoring broadcasting*) When defining global transitions, we eliminate the broadcasting through internal events by treating external and internal events in the same way. This elimination is conservative in the sense that the transformed EFSMs include all the possible execution sequences in UML state diagrams.

Control flow in UML state diagrams is identified in terms of the paths in EFSMs. Let $< GStates, C_0, GTrans >$ be an EFSM. Let $gs_i \in GStates$ and $gt_i = (C_i, e_i, g_i, a_i, C'_i) \in GTrans$ for $0 \leq i \leq n$. A sequence $(gs_0, gt_0), (gs_1, gt_1), \dots, (gs_n, gt_n)$ is called a *path* if $gs_0 = C_0, gs_n = C_n$ and for $0 \leq i \leq n-1$, $gs_i = C_i$ and $gs_{i+1} = C'_i$. In general, there are infinitely many paths in EFSMs and hence it is impossible to cover all these paths. We explore the following coverage criteria out of a potentially infinite family of criteria. Let P be a set of paths.

- P satisfies *path coverage* if P contains all possible paths through EFSMs. This is the strongest criterion and generally impossible to achieve.
- P satisfies *state coverage* (resp. *global state coverage*) if P includes every $s \in States$ (resp. $gs \in GStates$).
- P satisfies *transition coverage* (resp. *global transition coverage*) if P includes every $t \in Trans$ (resp. $gt \in GStates$).

Test cases satisfying these criteria can be constructed in terms of simple breadth or depth first searches over EFSMs. The following shows a test cases generation method for global state coverage by traversing EFSMs in the breadth first order. We can similarly define the methods for generating test cases satisfying other coverage criteria. Let $\langle GStates, C_0, GTrans\rangle$ be an EFSM.

```
make_state_coverage_testing_tree (Node)
begin
        if all $gs \in GStates$ are visited then return;
        for each $gt = (C_i, e_i, g_i, a_i, C'_i) \in GTrans$ do begin
                if $C_i = Node$ and $C'_i$ is not visited then
                        make $C'_i$ as a child of Node;
                end
                for each child cNode of Node do
                        make_state_coverage_testing_tree(cNode)
end
```

For example, Fig. 3 shows the testing tree constructed by the above algorithm in which the set of paths $\{p_1, p_2\}$ such that $p_1 = (C_1, gt_1), (C_2, gt_{31})$ and $p_2 = (C_1, gt_1), (C_2, gt_{51}), (C_4, gt_{32})$ satisfies global state coverage. The path p_1 in the EFSM corresponds to the message sequence of (*power-on, coffee*) in the UML state diagram of Fig. 1 and the path p_2 corresponds to the message sequence (*power-on, inc, coffee*). By

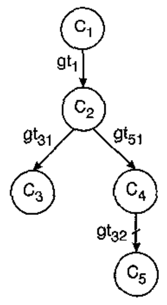


Fig. 3 *Example of testing trees*

sending these message sequences to the class (more precisely, an object instantiated from the class), we can traverse all the global states of the coffee vending machine.

In general, certain paths in EFSMs may be unexecutable or infeasible. For example, the path $p_1 = (C_1, gt_1), (C_2, gt_{31})$ is infeasible because the value of *money* is set as 0 by gt_1 and thus the enabling condition '$money > 0$' of gt_{31} cannot be satisfied. Of course, the selection of feasible paths is undecidable, thus making the application of these criteria undecidable.

5 Generating test cases based on data flow

Data flow in conventional programs is based on the notions of definitions and uses of variables in the statements. In UML state diagrams, variables can be defined and used in the actions of states and in transitions.

- A variable x is *defined* (resp. *used*) in an action a of a state if a assigns a value to x (resp. references x).
- A variable x is *defined* (resp. *used*) in a transition t if *action*(t) assigns a value to x (resp. *guard*(t) or *action*(t) references x).

For example, Table 1 shows the definitions and uses of the variables *money* and *light* in Fig. 1.

Let $\langle GStates, C_0, GTrans\rangle$ be an EFSM. Let $gt = (C, e, g, a, C') \in GTrans$ such that gt represents a set of transitions T in UML state diagrams. Recall that g is the conjunction of the guards of the transitions in T and a is the union of the actions of the transitions in T and the entry and exit actions that are executed by T.

- We say that a variable x is defined in $gt \in GTrans$ if x is defined in at least one of the actions in a.
- We say that a variable x is used in $gt \in GTrans$ if x is used in at least one of the guards in g or the actions in a.

Table 1: Definitions and uses of the variables in Fig. 1

	Definitions	Uses
a_1	*light*	ϕ
a_2	*light*	*light*
a_3	*light*	ϕ
t_1	*money*	ϕ
t_2	ϕ	ϕ
t_3	ϕ	*money*
t_4	ϕ	ϕ
t_5	*money*	ϕ
t_6	*money*	*money*
t_7	*money*	*money*
t_8	*money*	*money*

Table 2: Definitions and uses of the states in Fig. 1

Transitions	Definitions	Uses
t_1	All the states in Fig. 1	OFF, CVM
t_2	All the states in Fig. 1	ON, CVM
t_3	COFFEE, IDLE, BUSY,	IDLE, COFFEE, ON, CVM
t_4	COFFEE, IDLE, BUSY	BUSY, COFFEE, ON, CVM
t_5	MONEY, EMPTY, NOTEMPTY	EMPTY, MONEY, ON, CVM
t_6, t_7, t_8	MONEY, EMPTY, NOTEMPTY	NOTEMPTY, MONEY, ON, CVM

In addition to variables, UML state diagrams introduce a new type of data flow, which we call data flow through states. In UML state diagrams, both states and variables can affect the occurrence of transitions in the same way. That is, the precondition and postcondition of a transition are defined in terms of its source and target states as well as the values of variables. Precisely, data flow through states is defined as follows:

- A state s is *defined* in a transition t if s is a descendant of $scope(t)$, i.e. $s \in \rho^*(scope(t))$.
- A state is *used* in a transition t if s is an ancestor of the source state of t, i.e. $source(t) \in \rho^*(s)$.

Table 2 shows the definitions and uses of the states in Fig. 1. Consider the transition t_3 in Fig. 1. Since $scope(t_3) =$ COFFEE, we say that COFFEE, IDLE, and BUSY are defined by t_3. Intuitively, COFFEE and its children can be changed by t_3 while the states outside COFFEE are not changed by t_3. We say that IDLE, COFFEE, ON, and CVM are used by t_3, because $source(t_3) =$ IDLE. The source states and all of their ancestors are the precondition of transitions, i.e. they should be included in the current configuration for the transition to occur.

Let $\langle GStates, C_0, GTrans\rangle$ be an EFSM. Let gt be a global transition representing a set of transitions T in UML state diagrams.

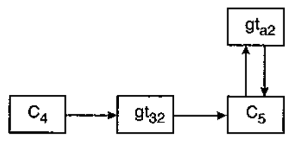
- We say that a state s is defined in $gt \in GTrans$ if x is defined in at least one transition $t \in T$.
- We say that a state s is used in $gt \in GTrans$ if x is used in at least one transition $t \in T$.

Now we transform EFSMs into flow graphs. A flow graph is a tuple $\langle V, N, E, def, use\rangle$ such that

- V is a set of variables and states in UML state diagrams.
- $N = N_s \cup N_t$ is a set of nodes. Each node in N_s is called an s-node and each node in N_t is called a t-node.
- $E = E_{st} \cup E_{ts}$ is a set of edges such that $E_{st} \subseteq \{(s, t) \mid s \in N_s, t \in N_t\}$ and $E_{ts} \subseteq \{(t, s) \mid t \in N_t, s \in N_s\}$.
- def: $N \to 2^V$ identifies the variables and states defined for each node.
- use: $N \to 2^V$ identifies the variables and states used for each node.

From a given EFSM $\langle GStates, C_0, GTrans\rangle$, we identify the flow graph as follows:

- $N_s = GStates$.
- $N_t = GTrans$.• For each global transition $gt_i = (C_i, e_i, g_i, a_i, C_i') \in GTrans$, we have two edges e_1 and e_2 such that $e_1 = (C_i, gt_i) \in E_{st}$ and $e_2 = (gt_i, C_i') \in E_{ts}$.
- For $s \in N_s$, $def(s) = \phi$.
- For $t \in N_t$, $def(t)$ is the set of all the variables and states defined in t.
- For $s \in N_s$, $use(s) = \phi$.
- For $t \in N_t$, $use(t)$ is the set of all the variables and states used in t.



Nodes	def	use
gt_{32}	{light} ∪ {COFFEE, IDLE, BUSY}	{money} ∪ {IDLE, SUPPLY, ON, CVM}
gt_{a2}	{light} ∪ {COFFEE, IDLE, BUSY}	{light} ∪ {BUSY, SUPPLY, ON, CVM}

Fig. 4 *Part of a flow graph*

Fig. 4 shows the part of the flow graph which is identified by considering C_4, C_5, gt_{32}, and gt_{a2} in Fig. 2.

Now we can readily generate test cases based on data flow from UML state diagrams by apply conventional data flow analysis techniques to the resulting flow graphs. We can also reuse a number of coverage criteria that have been extensively studied and compared in the testing literature such as *all-definition, all-use,* and *all def-use paths* coverage [25, 26]. By applying existing data flow techniques and coverage criteria, we can generate test cases as a set of paths that cover the associations between definitions and uses of each variable and state in UML state diagrams. For example, in Fig. 4 we can identify two def–use associations of *light:* (gt_{32}, gt_{a2}) and (gt_{a2}, gt_{a2}). That is, the definitions in gt_{32} and gt_{a2} can reach to the use in gt_{a2}. A path (C_1, gt_1), (C_2, gt_{51}), (C_4, gt_{32}), (C_5, gt_{a2}), (C_5, gt_{a2}), which corresponds to the sequence of $(power\text{-}on, inc, coffee, a_2, a_2)$ in Fig. 1, can be used as the test case that covers these associations. Table 3 shows all the def–use associations of

Table 3: Def–use associations of the variables in Fig. 1

Variables	Defs	Uses
light	gt_{31}	gt_{a1}
	gt_{a1}	gt_{a1}
	gt_{32}	gt_{a2}
	gt_{a2}	gt_{a2}
money	gt_1	gt_{31}
	gt_{51}	gt_{32}, gt_{61}, gt_{71}, gt_{81}, gt_{62}, gt_{72}, gt_{82}
	gt_{61}	gt_{32}, gt_{61}, gt_{71}, gt_{81}, gt_{62}, gt_{72}, gt_{82}
	gt_{71}	gt_{32}, gt_{61}, gt_{71}, gt_{81}, gt_{62}, gt_{72}, gt_{82}
	gt_{81}	gt_{31}
	gt_{52}	gt_{61}, gt_{71}, gt_{81}, gt_{62}, gt_{72}, gt_{82}
	gt_{62}	gt_{61}, gt_{71}, gt_{81}, gt_{62}, gt_{72}, gt_{82}
	gt_{72}	gt_{61}, gt_{71}, gt_{81}, gt_{62}, gt_{72}, gt_{82}
	gt_{82}	gt_{31}

light and *money* in Fig. 1. We can identify the def–use associations of *money* such as (gt_1, gt_{31}) that occur because of the hierarchical structure of states and (gt_{51}, gt_{32}) that occur because of the concurrent structure of states in UML state diagrams.

6 Conclusions and future work

We have presented a specification-based approach to class testing using UML state diagrams. We have proposed a transformation method from UML state diagrams into EFSMs and flow graphs and showed that conventional flow analysis techniques can be applied to test cases generation from UML state diagrams. Using the transformation we can flatten the hierarchical and concurrent structure of states and eliminate broadcast communications, while preserving both control and data flow in UML state diagrams. The resulting set of test cases provides the capability of checking that classes are correctly implemented against specifications written in UML state diagrams by testing whether class implementations establish the desired control and data flow specified in the specifications.

There are several areas that we are currently working on. In [23, 27, 28], tool support for object-oriented testing is discussed including specification editing, test cases generation, and test cases execution and validation. This paper discusses a method for the generation of test cases only and thus an automated environment would be needed to support the total process of class testing. In particular, when executing test cases and validating test results, we should resolve two technical issues of controllability and observability of class implementation states [29].

Second, this paper focuses on unit testing of classes and does not consider interrelationships between classes. In [30, 31], object-oriented integration testing techniques are discussed in which object-oriented testing is partitioned mainly into the following levels: classes, clusters, subsystems, and systems. In UML, state diagrams can be used as specifications in all of the four levels. We are planning to extend the work here to support object-oriented integration testing using UML state diagrams. In addition, UML provides three diagrams to specify communications between classes: sequence, collaboration, and activity diagrams. Testing techniques using these diagrams should be developed to complete the testing of dynamic behaviour specified in UML.

The final issue involves the testing of generalisation and specialisation of classes through inheritance. In [32], McGregor and Dyer discussed how to incrementally build Statechart-like specifications for a class from the specifications of its super classes. In [33, 34], the authors' exploited the hierarchical nature of the inheritance relation to test related groups of classes by reusing the testing information for a super class to guide the testing of a subclass. Combining this work with our work would provide a method appropriate for testing derived classes obtained by inheritance using UML state diagrams.

7 References

1 BOUGE, L., CHOQUET, N., FRIBOUR, L., and GAUDEL, M.C.: 'Test sets generation from algebraic specifications using logic programs', *J. Syst. Softw.* 1986, **6**, pp. 343–360

2 DOONG, R., and FRANKL, P.G.: 'Case studies on testing object–oriented programs', Proceedings of the 4th Symposium on *Software Testing, Analysis and Verification*, 1991, pp. 165–177

3 GANNON, J., MCMULLIN, P., and HAMLET, R.: 'Data-abstraction implementation, specification, and testing', *ACM Trans. Program. Lang. Syst.*, 1981, **31**, (3), pp. 211–223

4 PARRISH, A.S., BORIE, R.B., and CORDES, D.W.: 'Automated flow graph-based testing of object-oriented software modules', *J. Syst. Softw.*, 1993, **23**, pp. 95–109

5 ZWEBEN, S., HEYM, W., and KIMMICH, J.: 'Systematic testing of data abstractions based on software specifications', *J. Softw. Testing, Verif., Reliab.*, 1992, **1**, (4), pp. 39–55

6 HONG, H.S., KWON, Y.R., and CHA, S.D.: 'Testing of object-oriented programs based on finite state machines', Proceedings of Asia-Pacific *Software Engineering* Conference, 1995, pp. 234–241

7 KUNG, D., SUCHAK, N., GAO, J., HSIA, P., TOYOSHIMA, Y., and CHEN, C.: 'On object state testing', Proceedings of *Computer Software and Applications* Conference, 1994, pp. 222–227

8 MCGREGOR, J.D., and DYER, D.M.: 'Selecting functional test cases for object-oriented software', Proceedings of the Pacific Northwest *Software Quality* Conference, 1993

9 TURNER, C.D., and ROBSON, D.J.: 'The state-based testing of object-oriented programs', Proceedings of Conference on *Software Maintenance*, 1993, pp. 302–310

10 UML Proposal to the Object Management Group Version 1.1, Rational Corporation, Sept. 1997 (http://www.rational.com/uml/)

11 BOOCH, G.: 'Object oriented design with applications' (Benjamin Cummings, 1991)

12 RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., and LORENSEN, W.: 'Object oriented modeling and design' (Prentice Hall, 1991)

13 JACOBSON, I., CHRISTERSON, M., JONSSON, P., and OVERGGARD, G.: 'Object-oriented software engineering' (Addison-Wesley, 1992)

14 HAREL, D.: 'Statecharts: a visual formalism for complex systems', *Sci. Comput. Program.*, 1987, **8**, pp. 231–274

15 HECHT, M.S., 'Flow analysis of computer programs' (Elsevier North-Holland, 1977)

16 BOCHMAN, G.v., and PETRENKO, A.: 'Protocol testing: review of methods and relevance for software testing', Proceedings of the 1994 International Symposium on *Software Testing and Analysis*, 1994, pp. 109–124

17 ODA, T., and ARAKI, K.: 'Specification slicing in formal methods of software development', Proceedings of the 17th Annual International *Computer Software and Applications* Conference, 1993, pp. 313–319

18 CHANG, J., and RICHARDSON, D.J.: 'Static and dynamic specification slicing', Proceedings of the Fourth Irvine Symposium on *Software*, 1994

19 HEIMDAHL, M.P. E., and WHALEN, M.W.: 'Reduction and slicing of hierarchical state machines', Proceedings of the Fifth ACM SIGSOFT Symposium on the *Foundations of Software Engineering*, 1997

20 BEIZER, B.: 'Software testing techniques' (Van Nonstrand Reinhold, 1990)

21 CHOW, T.: 'Testing software design modeled by finite-state machines', *IEEE Trans. Softw. Eng.*, 1978, **4**, (3), pp. 178–187

22 BOSIK, B., and UYAR, M.Ü.: 'Finite state machine based formal methods in protocol conformance testing: from theory to implementation', *Comput. Netw. ISDN Syst.*, 1991, **22**, pp. 7–33

23 HOFFMAN, D., and STROOPER, P.: 'ClassBench: a framework for automated class testing', *Softw. –Pract. Exp.*, 1997, **27**, (5), pp. 573–597

24 PNUELI, A., and SHALEV, M.: 'What is in a step: on the semantics of Statecharts', Proceedings of International Conference on *Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science, (Springer-Verlag, 1991), **298**, pp. 245–264

25 FRANKL, P.G., and WEYUKER, E.J.: 'An applicable family of data flow testing criteria', *IEEE Trans. Softw. Eng.*, 1988, **14**, (10), pp. 1483–1498

26 NTAFOS, S.C.: 'A comparison of some structural testing strategies', *IEEE Trans. Softw. Eng.*, 1988, **14**, (6), pp. 868–874

27 MURPHY, G.C., TOWNSEND, P., and WONG, P.S.: 'Experiences with cluster and class testing', *Commun. ACM*, 1994, **37**, (9), pp. 39–47

28 POSTON, R.M.: 'Automated testing from object models', *Commun. ACM*, 1994, **37**, (9), pp. 48–58

29 WEIDE, B.W., EDWARDS, S.H., HEYM, W.D., LONG, T.J., and OGDEN, W.F.: 'Characterizing observability and controlability of software components', Proceedings of the 4th International Conference on *Software Reuse*, 1994, pp. 62–71

30 JORGENSEN, P.C., and ERICKSON, C.: 'Object-oriented integration testing', *Commun. ACM*, 1994, **37**, (9), pp. 30–38

31 MCGREGOR, J.D., and KORSON, T.D.: 'Integrating object-oriented testing and development processes', *Commun. ACM*, 1994, **37**, (9), pp. 59–77

32 MCGREGOR, J.D., and DYER, D.M.: 'A note on inheritance and state machines', *ACM SIGSOFT Softw. Eng. Notes*, 1993, **18**, (4), pp. 61–69

33 HARROL, M.J., MCGREGOR, J.D., and FITZPATRICK, K.J.: 'Incremental testing of object-oriented class structures', Proceedings of the 14th International Conference on *Software Engineering*, 1992, pp. 68–80

34 TURNER, C.D., and ROBSON, D.J.: 'The state-based testing and inheritance', Technical Report No. 1/93, 1993, School of Engineering and Computer Science, University of Durham, UK