SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2000; **10**: 203–227

A test sequence selection method for statecharts

Hyoung Seok Hong[1,*], Young Gon Kim[1], Sung Deok Cha[1,*], Doo Hwan Bae[1] and Hasan Ural[2]

[1]*Division of Computer Science, Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 373-1, Kusong-dong, Yusong-gu, 305-701, Taejon, Korea*
[2]*School of Information Technology and Engineering, University of Ottawa, Ottawa, Ontario, K1N 6N5, Canada*

SUMMARY

This paper presents a method for the selection of test sequences from statecharts. It is shown that a statechart can be transformed into a flow graph modelling the flow of both control and data in the statechart. The transformation enables the application of conventional control and data flow analysis techniques to test sequence selection from statecharts. The resulting set of test sequences provides the capability of determining whether an implementation establishes the desired flow of control and data expressed in statecharts. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: software testing and analysis; specification-based testing; test sequence selection; statecharts; data flow analysis

1. INTRODUCTION

The statechart formalism [1] is a graphical language that has been successfully used for specifying reactive and real-time systems. Basically, statecharts can be regarded as extended finite state machines (EFSMs) augmented with several concepts that enable the succinct specification of complex systems such as the hierarchical and concurrent structure on states and the communications mechanism through events broadcasting. Many variants of semantics have been proposed for statecharts [2] and one main difference between them lies in the ways of constructing steps. Intuitively, a step represents the response of a system to the events generated externally by the environment or internally by the system itself. The

* Correspondence to: Hyoung Seok Hong or Sung Deok Cha, Division of Computer Science and AITrc, Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology (KAIST), 373-1, Kusong-dong, Yusong-gu, 305-701, Taejon, Korea
Contract/grant sponsor: KOSEF through AITrc

Received 30 March 2000
Accepted 15 September 2000

behaviour of a statechart is then characterized by the set of all runs, each consisting of a sequence of steps.

Because there may be an infinite number of runs in a statechart, it is impossible to determine whether an implementation conforms to the required behaviour expressed in a statechart by considering all runs of the statechart. Therefore, exhaustive testing is impossible to achieve and it is necessary to have systematic coverage criteria that select a reasonable number of runs satisfying certain conditions. In recent years, several testing methods have been proposed for statecharts [3–7]. All of them consider statecharts without variables and discuss the application of test selection criteria such as state and transition coverage criteria to the control flow oriented test selection from statecharts. Clearly both the control and data flow aspects of a system must be tested and the tests generated by the data flow oriented criteria are complementary to those constructed by control flow oriented criteria [8,9]. Hence, for the generation of a comprehensive set of complementary tests, both types of test selection criteria must be used.

This paper presents a method that involves the application of conventional data flow analysis techniques to the selection of test sequences from statecharts. First it is shown that the behaviour of a statechart can be represented by an EFSM in a conservative way. In the approach presented here an EFSM is called a normal form specification of a statechart if the behaviour of the statechart is preserved in the resulting EFSM, i.e. each run of the statechart is also a run of the EFSM. Of course, there are an infinite number of EFSMs that are normal form specifications for a given statechart. A class of EFSMs is identified in this paper that can be used as a representative of all possible normal form specifications. The basic idea of the approach is to obtain a normal form specification for a statechart by flattening the hierarchical and concurrent structure on states and eliminating the broadcast communications in the statechart. This paper presents results based on the STATEMATE semantics of statecharts by Harel and Namaad [10].

The main benefit of the transformation from statecharts into EFSMs is that the existing testing methods and tools developed for EFSMs can now be reused for statecharts. A number of test selection methods have been proposed for EFSMs (for a survey, interested readers are referred to Reference [11]). In general, these methods can be divided into two classes depending on the test selection criteria employed: control flow oriented test selection [12–15] and data flow oriented test selection [16–20]. Among them, this paper considers the methods of Ural *et al*. [18–20] for the selection of test sequences from specifications written using formal description techniques [21]: SDL, Estelle and Lotos. Their methods use EFSMs as underlying models of such languages and select test sequences by transforming EFSMs into flow graphs and then applying data flow analysis techniques to the flow graphs. This paper shows that statecharts can be transformed into flow graphs modelling the flow of both control and data in statecharts by combining the transformation method from statecharts into EFSMs with the methods of Ural *et al*. This enables the application of data flow analysis techniques to the selection of test sequences from statecharts. The resulting set of test sequences provides the capability of determining whether an implementation establishes the desired flow of both control and data expressed in statecharts.

The remainder of the paper is organized as follows: Section 2 reviews preliminaries of EFSMs and statecharts. Section 3 gives a formal definition for the STATEMATE semantics, which was originally described informally in Reference [10]. The formalization is needed to

provide a formal foundation of the transformation method from statecharts into EFSMs presented in Section 4. Section 5 describes the application of data flow analysis techniques to the selection of test sequences from statecharts. Finally, Section 6 gives concluding remarks.

2. PRELIMINARIES

This section provides a brief introduction to EFSMs and statecharts.

2.1. EFSMs

An *extended finite state machine* (EFSM) M is a tuple $M = (Q, q_0, I, O, V, \Theta, \delta)$, where Q is a finite set of states; $q_0 \in Q$ is the initial state; I, O and V are finite sets of input symbols, output symbols and variables, respectively; Θ is an interpretation of V that assigns the initial value for each variable in V; δ is a finite set of transitions. Each transition in δ is a tuple (q, i, o, g, a, q'), where $q \in Q$, $i \in I$, $o \in O$, $q' \in Q$, g is a predicate on variables in V, and a is a set of assignments to variables in V. If the choice of transition in δ is not unique with respect to q, i and g, then the EFSM is non-deterministic.

In the approach here, state machines are used as semantic models for both EFSMs and statecharts. A *state machine* N is a tuple $N = (Q, q_0, I, O, \delta)$, where Q is a (possibly infinite) set of states; q_0 is the initial state; I and O are finite sets of input symbols and output symbols, respectively; δ is a (possibly infinite) set of transitions. Each transition in δ is a tuple (q, i, o, q'), where $q \in Q$, $i \in I$, $o \in O$ and $q' \in Q$.

An element in Q of state machine N is called a *global state* to distinguish it from a state in EFSMs and statecharts. Note that state machines are allowed to have an infinite number of states (and hence an infinite number of transitions) so that they can be used in the formalization of EFSMs and statecharts with infinite state space. Each element in δ of state machine N is called a *global transition*. Note that δ is defined as a relation rather than a function to model the non-deterministic behaviour of EFSMs and statecharts. A sequence $q_0 \xrightarrow{i_0/o_0} q_1 \xrightarrow{i_1/o_1} q_2 \xrightarrow{i_2/o_2} \cdots$ is a *path* of state machine N if q_0 is the initial state and $(q_j, i_j, o_j, q_{j+1}) \in \delta$, for all $j \geq 0$.

An interpretation σ of a set of variables V is a mapping that assigns to each variable $v \in V$ a value. An interpretation σ satisfies a predicate g, written as $\sigma \models g$, if and only if the value obtained by evaluating g using the value $\sigma(v)$ for each variable v appearing in g is true. For a set of assignments a, $a(\sigma)$ denotes the interpretation obtained by executing the assignments in a over σ. That is, $a(\sigma) = \sigma[v_1 \mapsto e_1, v_2 \mapsto e_2, \ldots, v_n \mapsto e_n]$ where $v_i := exp_i$ is in a and e_i is the value of expression exp_i evaluated over σ. A formal semantics of EFSMs is defined in terms of state machines as follows.

Definition 1: Let $M = (Q, q_0, I, O, V, \Theta, \delta)$ be an EFSM and Σ be the set of all interpretations of V. The *reachability graph* $G(M)$ for M is the state machine

$$G(M) = (Q \times \Sigma, (q_0, \Theta), I, O, \delta')$$

such that $((q, \sigma), i, o, (q', \sigma')) \in \delta'$ if there exists a transition $(q, i, o, g, a, q') \in \delta$ satisfying $\sigma \models g$ and $\sigma' = a(\sigma)$.

The behaviour of an EFSM M is characterized by the paths in its reachability graph $G(M)$. Each path in $G(M)$ is called a *run* of M and $R(M)$ is used to denote the set of all runs of M.

2.2. Statecharts

A *statechart* Z is a tuple (S, Π, V, Θ, T) where S, Π, V and T are sets of states, events, variables and transitions, respectively. Θ is an interpretation of V that assigns the initial value for each variable $v \in V$. Figure 1 shows an example that demonstrates the main features of statecharts. The variable m in Figure 1 is used as a synonym for *money* and the initial value for m is defined by $\Theta(m) = 0$.

A state of a statechart is either a basic state or a composite state containing 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, and it has exactly one default substate. For example, the OR-state CVM in Figure 1 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 related by an and-relation. Being in the AND-state ON implies being in COFFEE and MONEY simultaneously.

A *configuration* is a maximal set of states in which a system can be simultaneously.

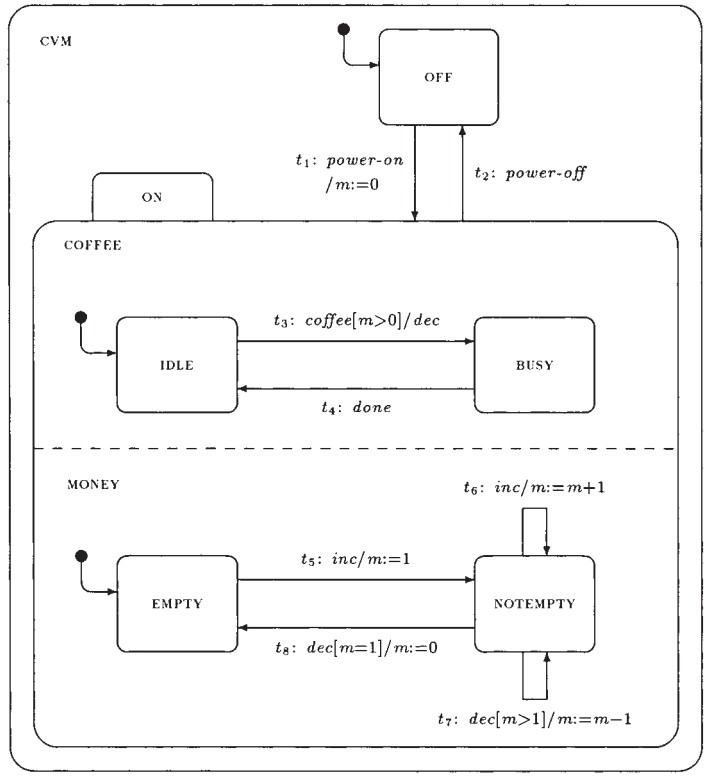


Figure 1. A simple coffee vending machine.

Precisely, $C \subseteq S$ 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 statechart shown in Figure 1 has the following configurations with C_1 as its initial configuration: $C_1 = \{$CVM, OFF$\}$, $C_2 = \{$CVM, ON, *coffee*, idle, money, empty$\}$, $C_3 = \{$CVM, ON, COFFEE, BUSY, MONEY, EMPTY$\}$, $C_4 = \{$CVM, ON, COFFEE, IDLE, MONEY, NOTEMPTY$\}$ and $C_5 = \{$CVM, ON, COFFEE, BUSY, MONEY, NOTEMPTY$\}$.

A transition of a statechart is a tuple (s, l, s'), where $s, s' \in S$ are the source state and target state, respectively. The label l is defined as $e[g]/a$ where e is called a trigger and is a Boolean expression on primitive events in Π; g is called a guard and is a predicate on variables in V; and a is called an action and is a set of $a_1, a_2, \ldots, a_n$ in which each a_i is either a primitive event in Π or an assignment to variables in V. For a transition $t \in T$, *source*(t), *target*(t), *trigger*(t) and *guard*(t) denote the source state, target state, trigger and guard of t, respectively, and *generated*(t) and *assignments*(t) denote the set of events and the set of assignments in the action a of t, respectively.

Two transitions *conflict* if there exists a configuration that includes their source states. For example, t_2 and t_3 in Figure 1 conflict because there exist configurations that include both ON and IDLE, say C_2 and C_4 . For a transition t, *Exits*(t) (respectively, *Enters*(t)) denotes the set of states that a system exits (respectively, enters) on taking transition t. In Figure 1, $Exits(t_1) = \{$OFF$\}$ and $Enters(t_1) = \{$ON, COFFEE, IDLE, MONEY, EMPTY$\}$. The formal definitions for *Exits*(t) and *Enters*(t) can be found in Reference [22].

3. A FORMAL DEFINITION FOR THE STATEMATE SEMANTICS

The central notion in the STATEMATE semantics is a step. Informally, a step is a maximal set of enabled transitions that are triggered by an input and are mutually non-conflicting. The input is a set of primitive events generated externally by the environment or internally by the system itself. Once a step is determined, the transitions in a step are executed simultaneously. Let $Z = (S, \Pi, V, \Theta, T)$ be a statechart, *Config* be the set of all configurations of Z, and Σ be the set of all interpretations of V.

Definition 2: Let $C, C' \in Config$ and $\sigma, \sigma' \in \Sigma$. Let $\tau \subseteq T$ be a set of transitions and $i, o \subseteq \Pi$ be sets of primitive events, called *input* and *output*, respectively. A tuple (i, τ, o) is a *step* from (C, σ) to (C', σ'), denoted by $(C, \sigma) \xrightarrow{(i,\tau,o)} (C', \sigma')$, if

- each transition in τ is triggered by i, i.e. *trigger*(t) evaluates to true for i;
- each transition in τ is enabled in (C, σ), i.e. $source(t) \in C$ and $\sigma \models guard(t)$;
- no two transitions in τ conflict;
- τ is maximal, i.e. each transition not in τ but triggered by i and enabled in (C, σ) conflicts with some transition in τ;
- $C' = (C - \cup_{t\in\tau} Exits(t)) \cup \cup_{t\in\tau} Enters(t)$;
- $o = \cup_{t\in\tau} generated(t)$;
- $\sigma' = a(\sigma)$, where $a = \cup_{t\in\tau} assignments(t)$.

If the choice of τ is not unique with respect to (C, σ) and i, then the statechart is non-deterministic.

3.1. The synchronous time model

The STATEMATE semantics provides two models of time: synchronous and asynchronous. The two time models use the same step construction method defined in Definition 2 and differ only in the ways external events can be introduced to a system. In the synchronous time model, external events can be introduced to a system after termination of each step. This implies that each step (i, τ, o) in the synchronous time model has a constraint on the input i such that $i = i_{ex} \cup i_{in}$ where i_{ex} is a set of external events generated by the environment and i_{in} is a set of internal events generated in the previous step.

Definition 3: The *reachability graph* $G_S(Z)$ for a statechart $Z = (S, \Pi, V, \Theta, T)$ with the synchronous time model is the state machine

$$G_S(Z) = (Config \times 2^{\Pi} \times \Sigma, (C_0, \varnothing, \Theta), 2^{\Pi}, 2^{\Pi}, \delta)$$

such that $((C, E, \sigma), i, o, (C', E', \sigma')) \in \delta$ and only if there exists a step (i, τ, o) from (C, σ) to (C', σ') satisfying (i) $E \subseteq i$ and (ii) $E' = o$.

A global state (C, E, σ) of state machine $G_S(Z)$ represents all the relevant status of statechart Z: (i) the states that the system is in; (ii) the events generated internally in the previous step; (iii) the values of variables. The initial global state is defined as $(C_0, \varnothing, \Theta)$, where C_0 is the initial configuration and $\varnothing$ states that there is no internal event generated at the initialization. For example, the initial global state in Figure 1 is $(\{\text{CVM}, \text{OFF}\}, \varnothing, [m \mapsto 0])$. The requirement '$E \subseteq i$' in the above definition states the constraint on the input i of a step (i, τ, o) such that any external event can be included in i as far as i contains the internal events generated in the previous step. The requirement '$E' = o$' states that the events generated in the current step are stored in the next global state.

Like EFSMs, the behaviour of a statechart is characterized by the paths in its reachability graph. Each path in the reachability graph $G_S(Z)$ of a statechart Z is called a *run* of Z with the synchronous time model. A run of a statechart represents a non-terminating computation that maintains an ongoing interaction with the environment. As an example, the following shows a run of the coffee vending machine in Figure 1. The run occurs when the sequence of steps $(\{t_1\}, \{t_5\}, \{t_3\}, \{t_8\}, \{t_4\})$ is taken.

$$(C_1, \varnothing, [m \mapsto 0]) \xrightarrow{\{power\text{-}on\}/\varnothing} (C_2, \varnothing, [m \mapsto 0]) \xrightarrow{\{inc\}/\varnothing}$$
$$(C_4, \varnothing, [m \mapsto 1]) \xrightarrow{\{coffee\}/\{dec\}}$$
$$(C_5, \{dec\}, [m \mapsto 1]) \xrightarrow{\{dec\}/\varnothing} (C_3, \varnothing, [m \mapsto 0])$$
$$\xrightarrow{\{done\}/\varnothing} (C_2, \varnothing, [m \mapsto 0]) \cdots$$

3.2. The asynchronous time model

In contrast with the synchronous time model, the asynchronous time model assumes that a system can accept external events only when the system is stable. Once external events are accepted, a sequence of steps is executed until the system becomes stable again. Stable means that there are no internal events generated in the previous step and there is no transition enabled and thus further steps are impossible without new external events. Formally, a global state (C, E, σ) is *stable* if $E = \varnothing$ and $(\varnothing, \varnothing, \varnothing)$ is the only possible step from (C, σ). A sequence of steps between two stable global states is referred to as a *super-step*. Each step (i, τ, o) in the asynchronous time model has a constraint on the input i such that $i = i_{ex}$, when the step occurs in a stable global state, and $i = i_{in}$ otherwise.

Definition 4: The *reachability graph* $G_A(Z)$ for a statechart $Z = (S, \Pi, V, \Theta, T)$ with the asynchronous time model is the state machine

$$G_A(Z) = (Config \times 2^{\Pi} \times \Sigma, (C_0, \varnothing, \Theta), 2^{\Pi}, 2^{\Pi}, \delta)$$

such that $((C, E, \sigma), i, o, (C', E', \sigma')) \in \delta$ if and only if there exists a step (i, τ, σ) from (C, σ) to (C', σ') satisfying (i) $E = i$ if (C, E, σ) is not stable and (ii) $E' = o$.

The requirement (i) states that only internal events generated in the previous step can be introduced to a system when the system is not stable. It also states that any external events can be introduced when the system is stable.

As mentioned before, the synchronous and asynchronous time models differ only in the ways external events can be introduced to a system. Because external events can be introduced at any step in the former model, the latter can be regarded as a restricted version of the former. $R_S(Z)$ (respectively, $R_A(Z)$) are used to denote the set of all runs of a statechart Z with the synchronous (respectively, asynchronous) time model.

Theorem 1: *Let Z be a statechart. $R_A(Z) \subseteq R_S(Z)$.*
Proof

Suppose that $(C_0, E_0, \sigma_0) \xrightarrow{i_0/o_0} (C_1, E_1, \sigma_1) \xrightarrow{i_1/o_1} \cdots$ is a run in $R_A(Z)$. When (C_j, E_j, σ_j) is stable, $E_j = \varnothing$ and thus $E_j \subseteq i_j$. When (C_j, E_j, σ_j) is not stable, $E_j = i_j$ and thus $E_j \subseteq i_j$. Therefore, the run is also a run in $R_S(Z)$.

4. NORMAL FORM SPECIFICATIONS FOR STATECHARTS

In the approach presented here, any EFSM that preserves the behaviour of a statechart is called a normal form specification (NFS) of the statechart. Precisely, an EFSM M is a *normal form specification* for a statechart Z with the synchronous (respectively, asynchronous) time model if $R_S(Z) \subseteq R(M)$ (respectively, $R_A(Z) \subseteq R(M)$). By Theorem 1, if an EFSM M is a normal form specification for a statechart Z with the synchronous time model, then it is also a normal form specification for Z with the asynchronous time model. That is, $R_S(Z) \subseteq R(M)$ implies that $R_A(Z) \subseteq R(M)$.

4.1. Transforming statecharts into EFSMs

For a given statechart, it is possible to obtain an infinite number of EFSMs that are normal form specifications for the statechart. For example, a simple way to obtain a normal form specification for a statechart is to generate the reachability graph of the statechart under the assumption that each variable of the statechart has a finite domain. The major problem is the data explosion, i.e. it is often impractical to generate such reachability graphs when the state space is large and it is impossible when the state space is infinite. The approach here avoids the data explosion by obtaining a normal form specification without expanding the values of variables of statecharts. The basic idea is such that the hierarchical and concurrent structure on states is flattened by (i) using the configurations of a statechart Z as the states of an EFSM M and (ii) using the possible steps of Z as the transitions of M.

Definition 5: The *normal form specification* (NFS) for a statechart $Z = (S, \Pi, V, \Theta, T)$, denoted by NFS($Z$), is the EFSM

$$(Config \times 2^{\Pi}, (C_0, \varnothing), 2^{\Pi}, 2^{\Pi}, V, \Theta, \delta)$$

such that $((C, E), i, o, g, a, (C', E')) \in \delta$ if and only if

- $E \subseteq i$;
- $E' = o$;
- there exists a set of transitions $\tau \subseteq T$ satisfying
  - — each transition in τ is triggered by i;
  - — the source state of each transition in τ is in C;
  - — no two transitions in τ conflict;
  - — $g = \bigwedge_{t \in \tau} guard(t)$;
  - — $a = \bigcup_{t \in \tau} assignments(t)$;
  - — $C' = (C - \bigcup_{t \in \tau} Exits(t)) \cup \bigcup_{t \in \tau} Enters(t)$;
  - — $o = \bigcup_{t \in \tau} generated(t)$.

A state of NFS(Z) is a pair (C, E) where C is a configuration and E is a set of primitive events and is used to store the internal events generated in the previous step. Hence, the state space of NFS(Z) is equivalent to that of statechart Z. A transition of NFS(Z) corresponds to a set τ of transitions of Z and represents a set of steps that may (but not necessarily do) occur in Z. Note that the requirements for τ in the above definition are similar to those in the step construction method except that the second, fourth and seventh items concerning the values of variables from Definition 2 are removed. The following lemma is a direct consequence of Definitions 2 and 5.

Lemma 1: *Let* $Z = (S, \Pi, V, \Theta, T)$ *be a statechart and* $E, E' \subseteq \Pi$. *For each step* (i, τ, o) *from* (C, σ) *to* (C', σ') *in* Z *satisfying* $E \subseteq i$ *and* $E' = o$, *there is a transition* $((C, E), i, o, g, a, (C', E'))$ *in* NFS(Z) *such that* $\sigma \models g$ *and* $\sigma' = a(\sigma)$.

Theorem 2: *Let* $Z = (S, \Pi, V, \Theta, T)$ *be a statechart.* $R_S(Z) \subseteq R(\text{NFS}(Z))$.

Proof

Suppose that $(C_0, E_0, \sigma_0) \xrightarrow{i_0/o_0} (C_1, E_1, \sigma_1) \xrightarrow{i_1/o_1} \cdots$ is a run in $R_S(Z)$. By Definition 3, (i_j, τ_j, o_j) is a step from (C_j, σ_j) to (C_{j+1}, σ_{j+1}) of Z such that $E_j \subseteq i_j$ and $E_{j+1} = o_j$, for all $j \geq 0$. By Lemma 1, $((C_j, E_j), i_j, o_j, g_j, a_j, (C_{j+1}, E_{j+1}))$ is a transition of NFS(Z) such that $\sigma_j \models g_j$ and $\sigma_{j+1} = a_j(\sigma_j)$, for all $j \geq 0$. By Definition 1, $((C_0, E_0), \sigma_0) \xrightarrow{i_0/o_0} ((C_1, E_1), \sigma_1) \xrightarrow{i_1/o_1} \cdots$ is a run of R(NFS(Z)).

The normal form specification for the coffee vending machine is shown in Figure 2 in which the events *power-on* and *power-off* are abbreviated as *pon* and *poff*. Each transition in

$$\{\tau_1, \tau_{21}, \tau_{22}, \tau_{23}, \tau_{24}, \tau_{31}, \tau_{32}, \tau_{41}, \tau_{42}, \tau_{51}, \tau_{52}, \tau_{61}, \tau_{62}, \tau_{71}, \tau_{72}, \tau_{73}, \tau_{81}, \tau_{82}, \tau_{83}, \tau_{\varnothing}\}$$

represents a set of possible steps that may occur when one event occurs. For example, consider the global state $(C_1, \varnothing, [m \mapsto 0])$ and the event *power-on* in Figure 1. The step $(\{power\text{-}on\}, \{t_1\}, \varnothing)$ from $(C_1, [m \mapsto 0])$ to $(C_2, [m \mapsto 0])$ is represented by the transition τ_1 in Figure 2. Each transition in

$$\{\tau_{3\|5}, \tau_{3\|6}, \tau_{3\|7}, \tau_{3\|8}, \tau_{4\|5}, \tau_{4\|6}, \tau_{4\|7}, \tau_{4\|8}, \tau_{\varnothing\|4}, \tau_{\varnothing\|5}, \tau_{\varnothing\|6}, \tau_{\varnothing\|7}, \tau_{\varnothing\|8}, \tau'_{4\|7}, \tau'_{4\|8}\}$$

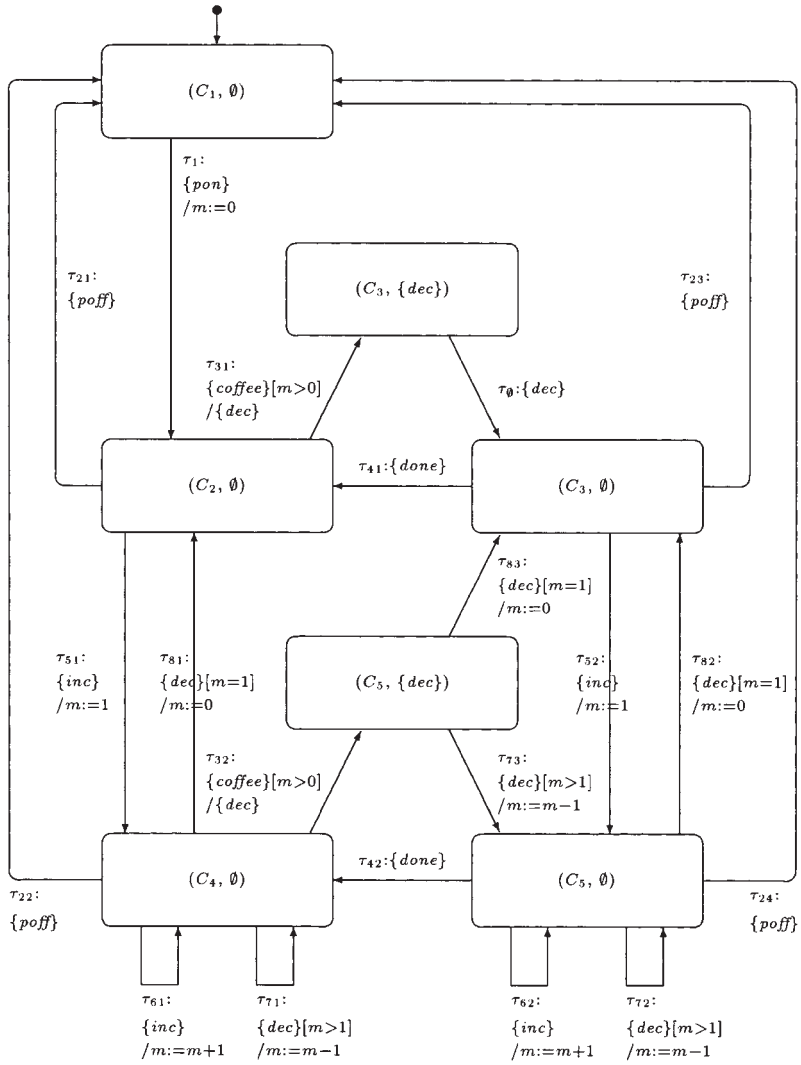
represents a set of possible steps that may occur when two events occur.

For legibility, certain implicit transitions are omitted from Figure 2 that are classified into two types. Each implicit transition of the first type represents steps consisting of an empty set of transitions and whose source state and target state are equivalent. For example, the step $(\{power\text{-}off\}, \varnothing, \varnothing)$ from $(C_1, [m \mapsto 0])$ to $(C_1, [m \mapsto 0])$ in Figure 1 is represented by the implicit transition $((C_1, \varnothing), \{power\text{-}off\}, \varnothing, true, \varnothing, (C_1, \varnothing))$ in Figure 2; 2^5 such implicit transitions are omitted from $(C_1, \varnothing)$ to $(C_1, \varnothing)$ in Figure 2, i.e. those whose input symbol corresponds to each subset of {*power-off, coffee, done, inc, dec*}. Each implicit transition of the second type represents steps whose transitions are triggered by an input containing more events than necessary. For example, the step $(\{power\text{-}on, power\text{-}off\}, \{t_1\}, \varnothing)$ from $(C_1, [m \mapsto 0])$ to $(C_2, [m \mapsto 0])$ is represented by $((C_1, \varnothing), \{power\text{-}on, power\text{-}off\}, \varnothing, \text{true}, m{:=}1, (C_2, \varnothing))$; 2^5 such implicit transitions are omitted from $(C_1, \varnothing)$ to $(C_2, \varnothing)$, i.e. those whose input symbol corresponds to each superset of {*power-on*} in {*power-on, power-off, coffee, done, inc, dec*}.

Now consider how exactly NFS(Z) represents the behaviour of the original statechart Z. In general, the converse of Theorem 2 does not hold because the values of variables are discarded when defining transitions of NFS(Z). The following shows a counterexample that falsifies the converse of Theorem 2. Suppose that events *done* and *inc* occur simultaneously at $(C_3, \varnothing, [m \rightarrow 0])$ in Figure 1, then

$$(C_3, \varnothing, [m \mapsto 0]) \xrightarrow{\{done,inc\}/\varnothing} (C_4, \varnothing, [m \mapsto 1]) \cdots$$

is the only possible run. However, there exist three runs from $(C_3, \varnothing, [m \rightarrow 0])$ in Figure 2 including the above one and the following two runs:



$$
\begin{aligned}
\tau_{3\|5} &= ((C_2, \emptyset), \{coffee, inc\}, \{dec\}, m{>}0, m{:=}1, (C_5, \{dec\}))\\
\tau_{3\|6} &= ((C_4, \emptyset), \{coffee, inc\}, \{dec\}, m{>}0, m{:=}m{+}1, (C_5, \{dec\}))\\
\tau_{3\|7} &= ((C_4, \emptyset), \{coffee, dec\}, \{dec\}, m{>}0 \wedge m{>}1, m{:=}m{-}1, (C_5, \{dec\}))\\
\tau_{3\|8} &= ((C_4, \emptyset), \{coffee, dec\}, \{dec\}, m{>}0 \wedge m{=}1, m{:=}0, (C_3, \{dec\}))\\
\tau_{4\|5} &= ((C_3, \emptyset), \{done, inc\}, \emptyset, true, m{:=}1, (C_4, \emptyset))\\
\tau_{4\|6} &= ((C_5, \emptyset), \{done, inc\}, \emptyset, true, m{:=}m{+}1, (C_4, \emptyset))\\
\tau_{4\|7} &= ((C_5, \emptyset), \{done, dec\}, \emptyset, m{>}1, m{:=}m{-}1, (C_4, \emptyset))\\
\tau_{4\|8} &= ((C_5, \emptyset), \{done, dec\}, \emptyset, m{=}1, m{:=}0, (C_2, \emptyset))\\
\tau_{\emptyset\|4} &= ((C_3, \{dec\}), \{done, dec\}, \emptyset, true, \emptyset, (C_2, \emptyset))\\
\tau_{\emptyset\|5} &= ((C_3, \{dec\}), \{inc, dec\}, \emptyset, true, m{:=}1, (C_5, \emptyset))\\
\tau_{\emptyset\|6} &= ((C_5, \{dec\}), \{inc, dec\}, \emptyset, true, m{:=}m{+}1, (C_5, \emptyset))\\
\tau_{\emptyset\|7} &= ((C_5, \{dec\}), \{inc, dec\}, \emptyset, m{>}1, m{:=}m{-}1, (C_5, \emptyset))\\
\tau_{\emptyset\|8} &= ((C_5, \{dec\}), \{inc, dec\}, \emptyset, m{=}1, m{:=}0, (C_3, \emptyset))\\
\tau'_{4\|7} &= ((C_5, \{dec\}), \{done, dec\}, \emptyset, m{>}1, m{:=}m{-}1, (C_4, \emptyset))\\
\tau'_{4\|8} &= ((C_5, \{dec\}), \{done, dec\}, \emptyset, m{=}1, m{:=}0, (C_2, \emptyset))
\end{aligned}
$$

Figure 2. The normal form specification for the coffee vending machine.

$$(C_3, \emptyset, [m \mapsto 0]) \xrightarrow{\{done,inc\}/\emptyset} (C_2, \emptyset, [m \mapsto 0]) \cdots$$
$$(C_3, \emptyset, [m \mapsto 0]) \xrightarrow{\{done,inc\}/\emptyset} (C_5, \emptyset, [m \mapsto 1]) \cdots$$

These two runs do not satisfy the fourth requirement of maximality in the step construction method in Definition 2. The transformation from a statechart Z into NFS(Z) induces these runs because the requirement of maximality cannot be fulfilled without considering the values of variables.

4.2. Discussion

4.2.1. Other STATEMATE constructs

The above transformation method does not exhaust other important STATEMATE constructs such as actions associated with states, transitions with multiple source and target states, compound transitions, histories and priorities. However, it is fairly simple to extend the transformation method once a formal definition is obtained for the step construction method for these constructs. For example, actions associated with states can be integrated into the step construction method as follows: let *entry*(s) (respectively, *exit*(s)) be the set of entry actions (respectively, exit actions) associated with state s. Now the seventh item in Definition 2, which concerns the action part of a transition, is replaced by

- $a = \cup_{t \in \tau} A_1(t) \cup A_2(t) \cup A_3(t)$, where
  - $A_1(t) = \cup_{s \in Exits(t) \cap C}$ *exit*(s);
  - $A_2(t) =$ *assignments*(t);
  - $A_3(t) = \cup_{s \in Enters(t) \cap C'}$ *entry*(s).

It is straightforward to modify the transformation method in order to reflect the above change in the definition of steps.

4.2.2. An alternative definition for normal form specifications

Another class of EFSMs could be identified as a normal form specification for a statechart by replacing ($Config \times 2^{\Pi}$, (C_0, $\emptyset$), 2^{Π}, 2^{Π}, V, Θ, δ) by ($Config$, C_0, 2^{Π}, 2^{Π}, V, Θ, δ) and removing the requirements '$E \subseteq i$' and '$E = o$' from Definition 5. A state of the alternative NFS corresponds to a configuration of a statechart. Therefore, each global state of the alternative NFS is of the form (C,σ) and does not contain any information about the internal events generated in the previous step. Like NFS(Z), the values of variables are discarded when defining transitions of the alternative NFS and hence the requirement of maximality is also not fulfilled. Moreover, the transformation from a statechart Z into the alternative NFS induces another type of run that cannot be a run of Z because the alternative NFS does not model the broadcast communications in Z at all. As an example, consider the following run of the alternative NFS:

$$(C_4, [m \mapsto 1]) \xrightarrow{\{coffee\}/\{dec\}} (C_5, [m \mapsto 1]) \xrightarrow{\{inc\}/\emptyset} (C_5, [m \mapsto 2]) \cdots$$

There exists no run in Figure 1 that corresponds to the above run because $\{dec\} \not\subseteq \{inc\}$, i.e. the above run does not satisfy the constraint on input such that each input should contain the internal events generated in the previous step.

The set of states of the alternative NFS has a size equivalent to the number of configurations of statechart Z and hence is much smaller than that of NFS(Z). On the other hand, there exist a large number of runs of the alternative NFS that cannot be a run of Z when communications through events broadcasting occur frequently in Z, while these runs cannot occur in NFS(Z) because NFS(Z) models the broadcast communications accurately.

4.2.3. Complexity

It is simple to show that the transformation from a statechart Z to NFS(Z) is exponential. Let $C \in Config$ be a configuration of Z and $T(C) \subseteq T$ be the set of transitions of Z defined as $\{t \in T \mid source(t) \in C \text{ and } generated(t) \neq \emptyset\}$. The set of states of NFS(Z) has the size linear to $\Sigma_{C \in Config}\, 2^{|T(C)|}$, which grows exponentially with the size of constituent AND-states. This is the well-known state explosion problem inherent in the static analysis methods based on the construction of FSMs or EFSMs from a set of communicating state machines. Although the approach advocated here also suffers from the state explosion problem, it has a novelty such that EFSMs are constructed from statecharts without expanding the values of variables. Thus the approach has complexity independent of the number of variable values and can be applicable even if the state space of statecharts is infinite.

Another problem of the static analysis methods for statecharts is the transition explosion. Most FSM and EFSM models in the testing literature are based on interleaving semantics that sequentializes simultaneous transitions in an arbitrary order so that at most one transition has to be analysed at a time. However, many statecharts semantics, including the STATEMATE semantics, do not adopt interleaving semantics and normally allow the occurrences of multiple transitions at a time. Hence the transition explosion problem is inherent in statecharts and their normal form specifications. Precisely, in NFS(Z) there exist $|2^{\Pi}|$ transitions starting from each state of the form $(C, \emptyset)$ and $|2^{\pi - E}|$ transitions from each state of the form (C, E), where Π is the set of primitive events and $E \subseteq \Pi$.

4.2.4. Controlling the transition explosion

A possible way to alleviate the transition explosion is to identify transitions of normal form specifications that are not worthy of analysis, e.g. the implicit transitions omitted in Figure 2. When selecting test sequences from statecharts using data flow analysis techniques, such implicit transitions need not be considered at all. As an example, consider the transitions t_5 and t_3 in Figure 1 in which the variable m is defined and used, respectively. A fundamental question in data flow analysis is one such as 'is there a definition-clear path with respect to m from t_5 to t_3?' When answering this question, implicit transitions of the first type are of no importance because such transitions represent steps consisting of an empty set of transitions, i.e. no definition and use of variables can occur in the steps. Implicit transitions of the second type can also be discarded because, for each implicit transition of the second type, there always exists a transition that has the same source and target state and the same set of definitions and uses of variables.

Theorem 2 together with Theorem 1 implies that NFS(Z) is also a normal form specification for a statechart Z with the asynchronous time model. Moreover, it is possible to obtain a more accurate normal form specification for a statechart with the asynchronous time model by strengthening the constraint on input i in Definition 5, i.e. replacing '$E \subseteq i$' by '$E = i$ when $E \neq \emptyset$'. Then, at the state (C_3, {*dec*}) in Figure 2, only the transition $\tau_{\emptyset} = ((C_3, \{dec\}), \{dec\}, \emptyset, true, \emptyset, (C_3,\emptyset))$ is allowed. Similarly, at the state (C_5, {*dec*}), only the transitions $\tau_{73} = ((C_5, \{dec\}), \{dec\}, \emptyset, m > 1, m := m-1, (C_5,\emptyset))$ and $\tau_{83} = ((C_5, \{dec\}), \{dec\}, \emptyset, m = 1, m := 0, (C_3,\emptyset))$ are allowed. Thus the following set of transitions are removed from Figure 2.

$$T_1 = \{\tau_{\emptyset\|4}, \tau_{\emptyset\|5}, \tau_{\emptyset\|6}, \tau_{\emptyset\|7}, \tau_{\emptyset\|8}, \tau'_{4\|7}, \tau'_{4\|8}\}$$

The transition explosion can be further alleviated by adopting several assumptions widely-used in specification languages for reactive systems. Included is the partition of the set Π of primitive events into two disjoint subsets $\Pi_{external}$ and $\Pi_{internal}$ comprising the external and internal events, respectively. With this assumption, a primitive event is either generated externally by the environment or internally by the system itself, but not both. Another interesting assumption is to partition the set Π into Π_{input}, Π_{output} and Π_{local} comprising the input, output and local events, respectively. These assumptions are often used to support the modular specification of reactive systems. In Figure 1, assume that $\Pi_{external}$ = {*power-on*, *power-off*, *coffee*, *done*, *inc*} and $\Pi_{internal}$ = {*dec*}. This partition can be used to reduce the number of transitions of NFS(Z), because *dec* cannot be introduced to a system when the system is in state (C,E) such that $E = \emptyset$. Hence the following set of transitions is removed from Figure 2.

$$T_2 = \{\tau_{71}, \tau_{72}, \tau_{81}, \tau_{82}, \tau_{3\|7}, \tau_{3\|8}, \tau_{4\|7}, \tau_{4\|8}\}$$

The number of transitions can be further reduced by the assumptions that restrict the simultaneous occurrences of external events. One such assumption is the input relation of ESTEREL [23]. For example, the input relation *coffee#inc* describes the incompatibility between the events *coffee* and *inc*, i.e. they cannot occur at the same time. Using the relation, the following set of transitions is removed from Figure 2.

$$T_3 = \{\tau_{3\|5}, \tau_{3\|6}\}$$

A similar assumption can be found in the SCR method [24], called the one-input assumption, which states that exactly one external event occurs at any instant of time. The one-input assumption is equivalent to the input relation in ESTEREL in which all pairs of external events are declared to be incompatible. The semantics used in Unified Modeling Language (UML) [25] also assumes that only one external event can be introduced at any instant of time. In the UML semantics, events generated externally by the environment of an object are acepted by an event queue for the object. The semantics assumes that the events in the queue are processed in sequence one at a time.

In summary, Figure 3 shows the revised normal form specification for the coffee vending

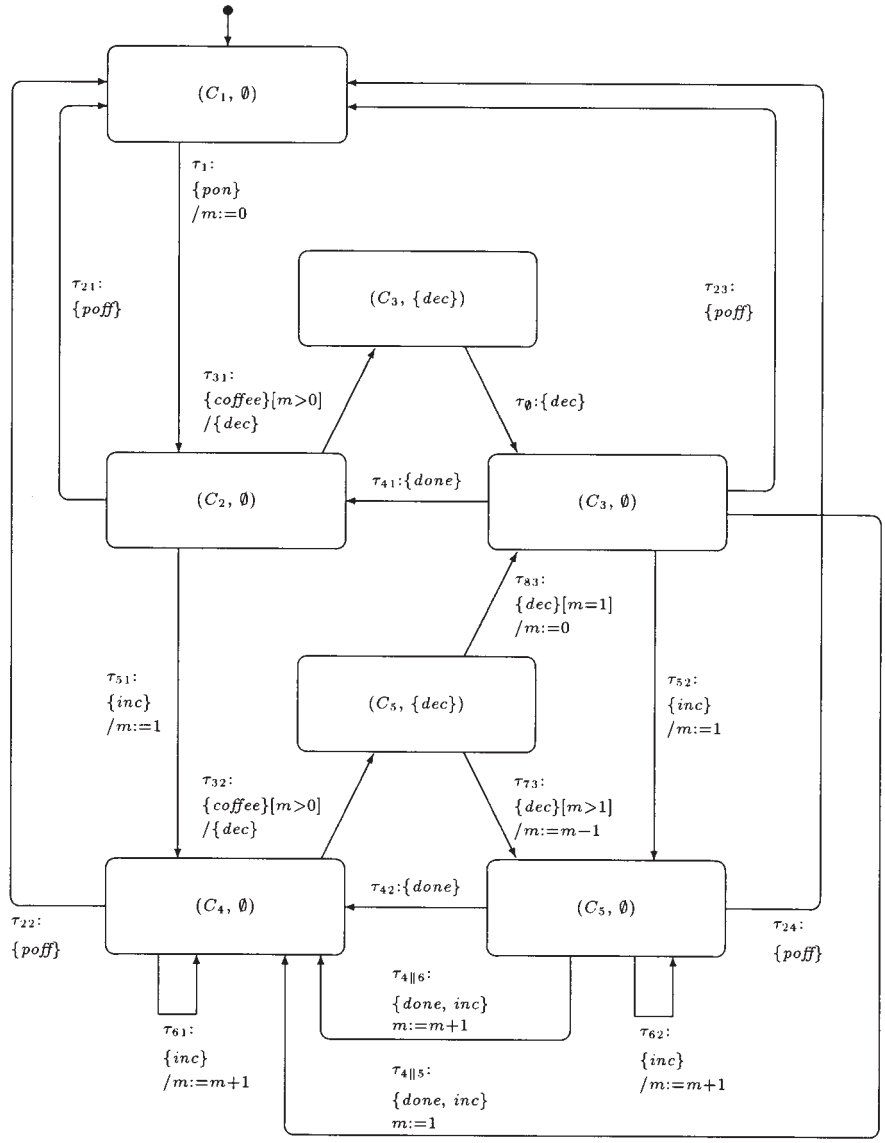


Figure 3. The revised normal form specification for the coffee vending machine.

machine obtained by removing the implicit transitions and the transitions in $T_1 \cup T_2 \cup T_3$ from Figure 2.

5. SELECTING TEST SEQUENCES FROM STATECHARTS BASED ON DATA FLOW ANALYSIS

Briefly, the selection of test sequences from a given statechart consists of the following steps.

Step 1. A normal form specification for the given statechart is constructed by following the method in the previous section.
Step 2. A flow graph for the normal form specification is constructed by following the method of Ural *et al*. [18–20].
Step 3. Each variable occurrence in the flow graph is classified as being a definition, computation use or predicate use.
Step 4. Associations between definitions and uses of each variable are identified on the flow graph.
Step 5. A set of paths satisfying certain data flow testing coverage criteria is selected from the flow graph.
Step 6. Each selected path is mapped into a test sequence of the given statechart.

Steps 3 to 5 are performed by conventional data flow analysis techniques with several modifications that aim to reflect the semantic differences between statecharts and procedural languages on which data flow analysis techniques are based.

5.1. Transforming EFSMs into flow graphs

In References [18–20], Ural *et al*. showed that data flow analysis techniques can be applied to the selection of test sequences from EFSMs by transforming EFSMs into a special class of flow graphs having the following structural characteristics. A *flow graph G* is a digraph $G = (N, en, E)$, where $N = \{n \mid n$ is a q-node, i-node or t-node$\}$; $en \in N$ is the entry node; $E = \{e \mid e$ is a qi-edge, it-edge or tq-edge$\}$.

Intuitively, the transformation of an EFSM $M = (Q, q_0, I, O, V, \Theta, \delta)$ into a flow graph $G = (N, \text{en}, E)$ is such that each state $q \in Q$ is represented by a q-node, each input symbol i in a transition $t \in \delta$ is represented by an i-node, and each transition $t \in \delta$ is represented by a t-node. Since a predicate affects the control flow in an EFSM, each predicate g in a transition (q, i, o, g, a, q') is associated with an it-edge. Note that qi-edges and tq-edges are used for completing the control flow of the EFSM.

Let $M = (Q, q_0, I, O, V, \Theta, \delta)$ be an EFSM, $s \in$ Q be a state, and $in \in I$ be an input symbol. Let $T_s = \{(q, i, o, g, a, q') \in \delta \mid q = \text{s}\}$, $T_{s,in} = \{(q, i, o, g, a, q') \in T_s \mid i = in\}$, and $W_s = \{in \in I \mid T_{s,in} \neq \varnothing\}$. As an example, consider the revised NFS in Figure 3 and fix s to the state $(C_5, \{dec\})$. It can be observed that $T_s = \{\tau_{73}, \tau_{83}\}$, $T_{s,\{dec\}} = \{\tau_{73}, \tau_{83}\}$, and $W_s = \{\{dec\}\}$.

Definition 6: The *flow graph G* for an EFSM $M = (Q, q_0, I, O, V, \Theta, \delta)$ is the digraph

$$G = (N, q_0, E)$$

defined by the mapping such that, for each $q \in Q$ of M, G consists of

- one q-node q;

- one i-node for each $i \in W_q$;
- one t-node for each $t \in T_q$;
- one qi-edge from the q-node q to each i-node $i \in W_q$;
- one it-edge from the i-node $i \in W_q$ to each t-node $t \in T_{q,i}$;
- one tq-edge from the t-node $t \in T_q$ to the q-node q' such that $t = (q, i, o, g, a, q')$.

Figure 4 shows the flow graph for the revised normal form specification in Figure 3. With state $(C_5, \{dec\})$ and input symbol $\{dec\}$, the following can be observed in the flow graph:

q-nodes: $(C_5, \{dec\})$, $(C_3, \varnothing)$, $(C_5, \varnothing)$
i-nodes: $\{dec\}$
t-nodes: τ_{73}, τ_{83}
qi-edges: $((C_5, \{dec\}), \{dec\})$
it-edges: $(\{dec\}, \tau_{73})$, $(\{dec\}, \tau_{83})$
tq-edges: $(\tau_{73}, (C_5, \varnothing))$, $(\tau_{83}, (C_3, \varnothing))$

5.2. Identifying definitions and uses of each variable

Each variable occurrence in a flow graph G for an EFSM M is classified as being a definition (def), computational use (c-use) or predicate use (p-use). The following conventions are used to identify definitions, c-uses and p-uses of each variable.

- A variable v is said to be *defined* at a t-node t if a of the EFSM's transition $t = (q,i,o,g,a,q')$ contains an assignment that defines v.
- A variable v is said to be *c-used* at a t-node t if a of the EFSM's transition $t = (q,i,o,g,a,q')$ contains an assignment that references v.
- A variable v is said to be *p-used* at an it-edge (i,t) if g of the EFSM's transition $t = (q,i,o,g,a,q')$ references v.

Based on the above classification, a pair of def and c-use sets is created for each t-node and a p-use set for each it-edge. A def set $def(t)$ (respectively, a c-use set $c\text{-}use(t)$) is the set of variables defined (respectively, used) at node t. A p-use set $p\text{-}use((i, t))$ is the set of variables used at edge (i, t). Table I shows the def sets and c-use sets for the flow graph in Figure 4. The p-use sets are shown in Table II.

5.3. Identifying associations between definitions and uses

A sequence of nodes $(n_1, n_2, \ldots, n_m)$, $m \geq 2$, is a path of flow graph $G = (N, en, E)$ if $(n_i, n_{i+1}) \in E$, for all $1 \leq i < m$. A path $(i, n_1, n_2, \ldots, n_m, j)$ is a *definition-clear path* with respect to variable v from node i to node j, if the nodes in the subpath $(n_1, \ldots, n_m)$ contain no definition of v. A path $(i, n_1, n_2, \ldots, n_m, j, k)$ is a *definition-clear path* with respect to variable v from node i to edge (j,k), if the nodes in the subpath $(n_1, \ldots, n_m, j)$ contain no definition of v.

Based on def, c-use and p-use sets, associations are identified between definitions and c-uses and between definitions and p-uses of each variable as follows. Let i be a node and v be a variable such that $v \in def(i)$.

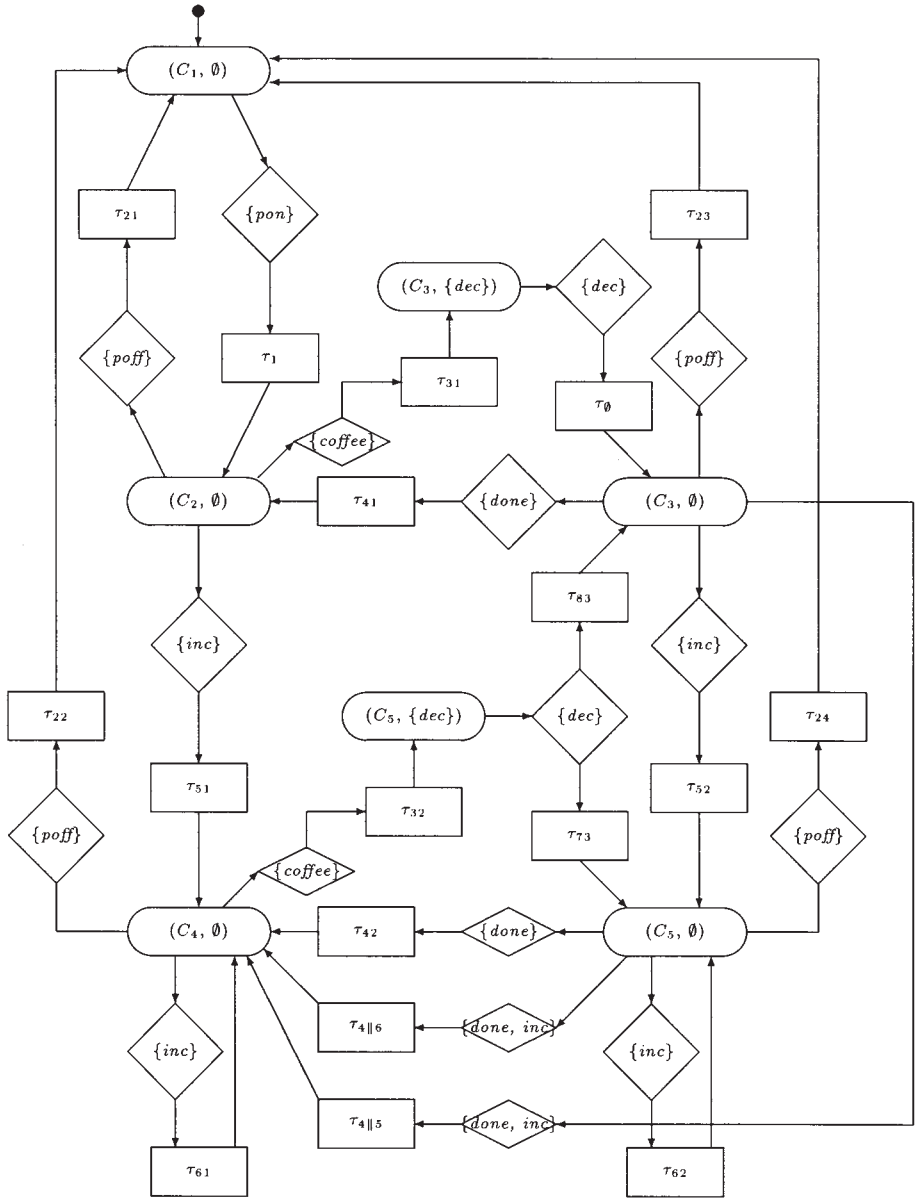


Figure 4. The flow graph for the revised normal form specification in Figure 3.

- *dcu*(v, i) is the set of all nodes j such that $v \in$ *c-use*(j) and there exists a definition-clear path with respect to v from i to j. A triple (v, i, j) is called a *def–c-use association* if $j \in dcu(v, i)$.
- *dpu*(v, i) is the set of all edges (j, k) such that $v \in$ *p-use*((j, k)) and there exists a

Table I. The def sets and c-use sets in Figure 4.

t-node	def set	c-use set
τ_1	m	$\varnothing$
τ_{51}	m	$\varnothing$
τ_{52}	m	$\varnothing$
τ_{61}	m	m
τ_{62}	m	m
τ_{73}	m	m
τ_{83}	m	$\varnothing$
$\tau_{4\|5}$	m	$\varnothing$
$\tau_{4\|6}$	m	m
The other *t*-nodes	$\varnothing$	$\varnothing$

Table II. The p-use sets in Figure 4.

it-edge	p-use set
({*coffee*}, τ_{31})	m
({*coffee*}, τ_{32})	m
({*dec*}, τ_{73})	m
({*dec*}, τ_{83})	m
The other *it*-edges	$\varnothing$

definition-clear path with respect to v from i to (j,k). A triple $(v, i, (j, k))$ is called a *def–p-use association* if $(j, k) \in dpu(v, i)$.

- A *def–use association* is either a def–c-use association or a def–p-use association.

Table III shows the dcu and dpu sets for the flow graph in Figure 4. For example, consider

Table III. The dcu sets and dpu sets in Figure 4.

Node	dcu set	dpu set
τ_1	$\varnothing$	({*coffee*}, τ_{31})
τ_{51}	τ_{61}, τ_{73}	({*coffee*},τ_{32}), ({*dec*},τ_{73}), ({*dec*},τ_{83})
τ_{52}	τ_{61}, τ_{62}, τ_{73}, $\tau_{4\|6}$	({*coffee*},τ_{32}), ({*dec*},τ_{73}), ({*dec*},τ_{83})
τ_{61}	τ_{61}, τ_{73}	({*coffee*},τ_{32}), ({*dec*},τ_{73}), ({*dec*},τ_{83})
τ_{62}	τ_{61}, τ_{62}, τ_{73}, $\tau_{4\|6}$	({*coffee*},τ_{32}), ({*dec*},τ_{73}), ({*dec*},τ_{83})
τ_{73}	τ_{61}, τ_{62}, τ_{73}, $\tau_{4\|6}$	({*coffee*},τ_{32}), ({*dec*},τ_{73}), ({*dec*},τ_{83})
τ_{83}	$\varnothing$	({*coffee*},τ_{31})
$\tau_{4\|5}$	τ_{61}, τ_{73}	({*coffee*},τ_{32}), ({*dec*},τ_{73}), ({*dec*},τ_{83})
$\tau_{4\|6}$	τ_{61}, τ_{73}	({*coffee*},τ_{32}), ({*dec*},τ_{73}), ({*dec*},τ_{83})

the def–c-use association (m,τ_{52},τ_{61}). The definition of m at the t-node τ_{52} can reach the use of m at the t-node τ_{61} through the definition-clear path (τ_{52}, (C_5, $\varnothing$), {*done*}, τ_{42}, (C_4, $\varnothing$), {inc}, τ_{61}).

In general, there exist three types of associations between definitions and uses in flow graphs constructed from statecharts. The first type includes associations between a definition at a t-node t_1 and a use at a t-node t_2 such that the transitions t_1 and t_2 represent steps occurring within the same OR-state in the original statechart, e.g. the def–c-use association (m, τ_{51}, τ_{61}) in Table III. The t-node τ_{51} represents the statechart's transition 't_5: *inc*/*m*:=1' in Figure 1 and τ_{61} represents 't_6: *inc*/*m*:=*m* + 1'. Associations between definitions and uses occurring in ordinary EFSMs belong to this type. The hierarchical and concurrent structure on states introduces two additional types of association. The second type is caused by the hierarchical structure on states, e.g. the def–p-use association (m, τ_1, ({*coffee*}, τ_{31})) in Table III. The t-node τ_1 represents 't_1: *power-on*/*m*:=0' and τ_{31} represents 't_3: *coffee*[*m*>0]/*m*:=0'. The definition of m at t_1 can reach the p-use at t_3 because of the hierarchical structure on states. The third type is caused by the concurrent structure on states, e.g. the def–p-use association (m, τ_{51}, ({*coffee*}, τ_{32})) in Table III. The concurrent structure on states allows that the definition of m at t_5 can reach the p-use at t_3.

5.4. Selecting a set of complete paths

In conventional data flow analysis, flow graphs usually contain a set of exit nodes to model the terminating behaviour of programs written in procedural languages. A set of complete paths is then selected to cover associations between definitions and uses. A path in such flow graphs is called *complete* if the first node of the path is the entry node and the last node is an exit node. On the other hand, there is no exit node in flow graphs constructed from statecharts because the behaviour of statecharts is characterized by their non-terminating runs. This leads to a modified definition for complete paths as follows. A path of a flow graph is complete if its first node is the entry node and its last node is a q-node. That is, each q-node is regarded as a pseudo exit node.

A number of coverage criteria have been proposed based on data flow analysis in the software testing literature. Rapps and Weyuker [9] proposed a family of data flow testing coverage criteria: all-nodes, all-edges, all-defs, all-p-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses, all-uses, all-du-paths, all-paths. It is clear that the all-nodes, all-edges, and all-paths criteria represent the well-known statement, branch and path coverage criteria, respectively. The all-defs, all-p-uses, all-c-uses/some-p-uses and all-p-uses/some-c-uses criteria in general yield a set of complete paths that do not necessarily cover all associations between definitions and uses of each variable. On the other hand, all-uses and all-du-paths criteria cover all associations between definitions and uses of each variable. This paper shows the application of the all-uses criterion to flow graphs constructed from statecharts. Let $G = (N, en, E)$ be a flow graph and P be a set of complete paths in G. P satisfies the all-uses criterion if, for each node $n \in N$ and each $v \in def(n)$, P includes a definition-clear path with respect to v from the node n to each element of $dcu(v,n)$ and $dpu(v,n)$.

Table IV shows a set of complete paths selected by the application of the all-uses criterion to the flow graph in Figure 4. As an example, the complete path p_1 covers the following def–use associations:

Table IV. A set of complete paths satisfying the all-uses criterion in Figure 4.

p_1	p_2	p_3	p_4	p_5	p_6	p_7
$(C_1,\varnothing)$	$(C_1,\varnothing)$	$(C_1,\varnothing)$	$(C_1,\varnothing)$	$(C_1,\varnothing)$	$(C_1,\varnothing)$	$(C_1,\varnothing)$
{*pon*}	{*pon*}	{*pon*}	{*pon*}	{*pon*}	{*pon*}	{*pon*}
τ_1	τ_1	τ_1	τ_1	τ_1	τ_1	τ_1
$(C_2,\varnothing)$	$(C_2,\varnothing)$	$(C_2,\varnothing)$	$(C_2,\varnothing)$	$(C_2,\varnothing)$	$(C_2,\varnothing)$	$(C_2,\varnothing)$
{*coffee*}	{*coffee*}	{*inc*}	{*inc*}	{*inc*}	{*inc*}	{*inc*}
τ_{31}	τ_{31}	τ_{51}	τ_{51}	τ_{51}	τ_{51}	τ_{51}
$(C_3,\{dec\})$	$(C_3,\{dec\})$	$(C_4,\varnothing)$	$(C_4,\varnothing)$	$(C_4,\varnothing)$	$(C_4,\varnothing)$	$(C_4,\varnothing)$
{*dec*}	{*dec*}	{*coffee*}	{*inc*}	{*coffee*}	{*coffee*}	{*inc*}
$\tau_\varnothing$	$\tau_\varnothing$	τ_{32}	τ_{61}	τ_{32}	τ_{32}	τ_{51}
$(C_3,\varnothing)$	$(C_3,\varnothing)$	$(C_5,\{dec\})$	$(C_4,\varnothing)$	$(C_5,\{dec\})$	$(C_5,\{dec\})$	$(C_4,\varnothing)$
{*done*}	{*inc*}	{*dec*}	{*coffee*}	{*dec*}	{*dec*}	{*coffee*}
τ_{41}	τ_{52}	τ_{83}	τ_{32}	τ_{83}	τ_{83}	τ_{32}
$(C_2,\varnothing)$	$(C_5,\varnothing)$	$(C_3,\varnothing)$	$(C_5,\{dec\})$	$(C_3,\varnothing)$	$(C_3,\varnothing)$	$(C_5,\{dec\})$
{*inc*}	{*done*}	{*inc*}	{*dec*}	{*inc*}	{*inc*}	{*dec*}
τ_{51}	τ_{42}	τ_{52}	τ_{73}	τ_{52}	τ_{52}	τ_{73}
$(C_4,\varnothing)$	$(C_4,\varnothing)$	$(C_5,\varnothing)$	$(C_5,\varnothing)$	$(C_5,\varnothing)$	$(C_5,\varnothing)$	$(C_5,\varnothing)$
{*coffee*}	{*coffee*}	{*done*}	{*done*}	{*done*}	{*done,inc*}	{*done*}
τ_{32}	τ_{32}	τ_{42}	τ_{42}	τ_{42}	$\tau_{4\|6}$	τ_{42}
$(C_5,\{dec\})$	$(C_5,\{dec\})$	$(C_4,\varnothing)$	$(C_4,\varnothing)$	$(C_4,\varnothing)$	$(C_4,\varnothing)$	$(C_4,\varnothing)$
{*dec*}	{*dec*}	{*inc*}	{*inc*}	{*coffee*}	{*inc*}	{*coffee*}
τ_{73}	τ_{73}	τ_{61}	τ_{61}	τ_{32}	τ_{61}	τ_{32}
$(C_5,\varnothing)$	$(C_5,\varnothing)$	$(C_4,\varnothing)$	$(C_4,\varnothing)$	$(C_5,\{dec\})$	$(C_4,\varnothing)$	$(C_5,\{dec\})$
{*done*}	{*inc*}	{*inc*}	{*coffee*}	{*dec*}	{*coffee*}	{*dec*}
τ_{42}	τ_{62}	τ_{61}	τ_{32}	τ_{83}	τ_{32}	τ_{83}
$(C_4,\varnothing)$	$(C_5,\varnothing)$	$(C_4,\varnothing)$	$(C_5,\{dec\})$	$(C_3,\varnothing)$	$(C_5,\{dec\})$	$(C_3,\varnothing)$
{*inc*}	{*done*}	{*coffee*}	{*dec*}	{*inc*}	{*dec*}	{*done,inc*}
τ_{61}	τ_{42}	τ_{32}	τ_{73}	τ_{52}	τ_{73}	$\tau_{4\|5}$,
$(C_4,\varnothing)$	$(C_4,\varnothing)$	$(C_5,\{dec\})$	$(C_5,\varnothing)$	$(C_5,\varnothing)$	$(C_5,\varnothing)$	$(C_4,\varnothing)$
{*coffee*}	{*coffee*}	{*dec*}	{*done*}	{*inc*}	{*done,inc*}	{*coffee*}
τ_{32}	τ_{32}	τ_{73}	τ_{42}	τ_{62}	$\tau_{4\|6}$	τ_{32}
$(C_5,\{dec\})$	$(C_5,\{dec\})$	$(C_5,\varnothing)$	$(C_4,\varnothing)$	$(C_5,\varnothing)$	$(C_4,\varnothing)$	$(C_5,\{dec\})$
{*dec*}	{*dec*}	{*inc*}	{*coffee*}	{*done,inc*}	{*poff*}	{*dec*}
τ_{83}	τ_{83}	τ_{62}	τ_{32}	$\tau_{4\|6}$	τ_{22}	τ_{83}
$(C_3,\varnothing)$	$(C_3,\varnothing)$	$(C_5,\varnothing)$	$(C_5,\{dec\})$	$(C_4,\varnothing)$	$(C_1,\varnothing)$	$(C_3,\varnothing)$
{*done*}	{*inc*}	{*inc*}	{*dec*}	{*coffee*}		{*done,inc*}
τ_{41}	τ_{52}	τ_{62}	τ_{83}	τ_{32}		$\tau_{4\|5}$
$(C_2,\varnothing)$	$(C_5,\varnothing)$	$(C_5,\varnothing)$	$(C_3,\varnothing)$	$(C_5,\{dec\})$		$(C_4,\varnothing)$
{*coffee*}	{*done,inc*}	{*done*}	{*poff*}	{*dec*}		{*inc*}
τ_{31}	$\tau_{4\|6}$	τ_{42}	τ_{23}	τ_{73}		τ_{61}
$(C_3,\{dec\})$	$(C_4,\varnothing)$	$(C_4,\varnothing)$	$(C_1,\varnothing)$	$(C_5,\varnothing)$		$(C_4,\varnothing)$
{*dec*}	{*coffee*}	{*inc*}		{*poff*}		{*poff*}
$\tau_\varnothing$	τ_{32}	τ_{61}		τ_{24}		τ_{22}
$(C_3,\varnothing)$	$(C_5,\{dec\})$	$(C_4,\varnothing)$		$(C_1,\varnothing)$		$(C_1,\varnothing)$
{*done,inc*}	{*dec*}	{*poff*}				
$\tau_{4\|5}$	τ_{83}	τ_{22}				
$(C_4,\varnothing)$	$(C_3,\varnothing)$	$(C_1,\varnothing)$				
{*coffee*}	{*poff*}					
τ_{32}	τ_{23}					
$(C_5,\{dec\})$	$(C_1,\varnothing)$					
{*dec*}						
τ_{73}						
$(C_5,\varnothing)$						
{*poff*}						
τ_{24}						
$(C_1,\varnothing)$						

$(\tau_1, (\{coffee\},\tau_{31}))$, $(\tau_{51}, (\{dec\},\tau_{73}))$, (τ_{51},τ_{73}), $(\tau_{61}, (\{dec\},\tau_{83}))$, $(\tau_{83}, (\{coffee\},\tau_{31}))$, $(\tau_{4\|5}, (\{dec\},\tau_{73}))$, $(\tau_{4\|5},\tau_{73})$

5.5. Mapping complete paths into test sequences

Like the FSM and EFSM based testing methods, the approach here uses a sequence of inputs of a statechart as a test sequence. In deterministic FSMs and EFSMs, a sequence of inputs corresponds to exactly one path in such models. However, the information of states and outputs is additionally necessary to determine each path uniquely in non-deterministic models. Because statecharts are also non-deterministic, it is required that a test sequence of a statechart contains two more sequences of configurations and outputs in addition to a sequence of inputs. A sequence $C_0 \xrightarrow{in_0/out_0} C_1 \xrightarrow{in_1/out_1} \cdots \xrightarrow{in_{m-1}/out_{m-1}} C_m$ is a *test sequence* of a statechart $Z = (S, \Pi, V, \Theta, T)$ if C_j is a configuration of Z, for all $0 \leq j \leq m$, and $in_j \subseteq \Pi$ is an input of Z and $out_j \subseteq \Pi$ is an output of Z, for all $0 \leq j \leq m-1$. It is easy to map each complete path of a flow graph into a test sequence by noting the configurations in the *q*-nodes, inputs in the *i*-nodes, and outputs in the *t*-nodes of the complete path. Table V shows a set of test sequences of the coffee vending machine. Each test sequence in Table V corresponds to a complete path in Table IV.

A test sequence $C_0 \xrightarrow{in_0/out_0} C_1 \xrightarrow{in_1/out_1} \cdots \xrightarrow{in_{m-1}/out_{m-1}} C_m$ of a statechart Z with the synchronous time model (respectively asynchronous time model) is *executable* or *feasible* if there exists a run $(C_0, E_0, \sigma_0) \xrightarrow{i_0/o_0} (C_1, E_1, \sigma_1) \xrightarrow{i_1/o_1} (C_2, E_2, \sigma_2) \xrightarrow{i_2/o_2} \cdots$ in $R_S(M)$ (respectively $R_A(M)$) such that $i_j = in_j$ and $o_j = out_j$, for all $0 \leq j \leq m-1$. Such a run is called an *accepting run* for the test sequence. The rest sequences ts_3, ts_4, ts_5, ts_6 and ts_7 in Table V are executable and the following shows an accepting run for ts_3.

$$
\begin{aligned}
&(C_1, \varnothing, [m \mapsto 0]) \xrightarrow{\{power\text{-}on\}/\varnothing} (C_2, \varnothing, [m \mapsto 0]) \xrightarrow{\{inc\}/\varnothing} (C_4, \varnothing, [m \mapsto 1]) \xrightarrow{\{coffee\}/\{dec\}} \\
&(C_5, \{dec\}, [m \mapsto 1]) \xrightarrow{\{dec\}/\varnothing} (C_3, \varnothing, [m \mapsto 0]) \xrightarrow{\{inc\}/\varnothing} (C_5, \varnothing, [m \mapsto 1]) \xrightarrow{\{done\}/\varnothing} \\
&(C_4, \varnothing, [m \mapsto 1]) \xrightarrow{\{inc\}/\varnothing} (C_4, \varnothing, [m \mapsto 2]) \xrightarrow{\{inc\}/\varnothing} (C_4, \varnothing, [m \mapsto 2]) \xrightarrow{\{coffee\}/\{dec\}} \\
&(C_5, \{dec\}, [m \mapsto 2]) \xrightarrow{\{dec\}/\varnothing} (C_5, \varnothing, [m \mapsto 1]) \xrightarrow{\{inc\}/\varnothing} (C_5, \varnothing, [m \mapsto 2]) \xrightarrow{\{inc\}/\varnothing} \\
&(C_5, \varnothing, [m \mapsto 3]) \xrightarrow{\{done\}/\varnothing} (C_4, \varnothing, [m \mapsto 3]) \xrightarrow{\{inc\}/\varnothing} (C_4, \varnothing, [m \mapsto 4]) \xrightarrow{\{power\text{-}off\}/\varnothing} \\
&(C_1, \varnothing, [m \mapsto 4]) \xrightarrow{\{power\text{-}on\}/\varnothing} \cdots
\end{aligned}
$$

On the other hand, there is no accepting run for the test sequences ts_1 and ts_2 in Table V. Consider the test sequence $C_1 \xrightarrow{\{power\text{-}on\}/\varnothing} C_2 \xrightarrow{\{coffee\}/\{dec\}} C_3 \xrightarrow{\{dec\}/\varnothing} C_3$, which is a subsequence of both ts_1 and ts_2 and covers the def–use association $(\tau_1, (coffee,\tau_{31}))$. The variable m in Figure 1 is assigned the value of 0 by the step $(\{power\text{-}on\}, \{t_1\}, \varnothing)$ and subsequently the step $(\{coffee\},\{t_3\},\varnothing)$ cannot be taken because the guard of t_3, $[m > 0]$, is not satisfied.

The guard is satisfied only on those test sequences that include at least one occurrence of *inc* preceding the first occurrence of *coffee*. The other def–use associations covered by

Table V. Test sequences corresponding to the complete paths in Table IV.

ts_1	$C_1 \xrightarrow{\{pon\}/\varnothing} C_2 \xrightarrow{\{coffee\}/\{dec\}} C_3 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{done\}/\varnothing} C_2 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_5 \xrightarrow{\{done\}/\varnothing} C_4 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{done\}/\varnothing} C_2 \xrightarrow{\{coffee\}/\{dec\}} C_3 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{done,inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_5 \xrightarrow{\{poff\}/\varnothing} C_1$
ts_2	$C_1 \xrightarrow{\{pon\}/\varnothing} C_2 \xrightarrow{\{coffee\}/\{dec\}} C_3 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{inc\}/\varnothing} C_5 \xrightarrow{\{done\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_5 \xrightarrow{\{inc\}/\varnothing} C_5 \xrightarrow{\{done\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{inc\}/\varnothing} C_5 \xrightarrow{\{done,inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{poff\}/\varnothing} C_1$
ts_3	$C_1 \xrightarrow{\{pon\}/\varnothing} C_2 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{inc\}/\varnothing} C_5 \xrightarrow{\{done\}/\varnothing} C_4 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_5 \xrightarrow{\{inc\}/\varnothing} C_5 \xrightarrow{\{done\}/\varnothing} C_4 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{poff\}/\varnothing} C_1$
ts_4	$C_1 \xrightarrow{\{pon\}/\varnothing} C_2 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_5 \xrightarrow{\{done\}/\varnothing} C_4 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_5 \xrightarrow{\{done\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_5 \xrightarrow{\{poff\}/\varnothing} C_1$
ts_5	$C_1 \xrightarrow{\{pon\}/\varnothing} C_2 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{inc\}/\varnothing} C_5 \xrightarrow{\{done\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{done\}/\varnothing} C_3 \xrightarrow{\{inc\}/\varnothing} C_5 \xrightarrow{\{done,inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_5 \xrightarrow{\{poff\}/\varnothing} C_1$
ts_6	$C_1 \xrightarrow{\{pon\}/\varnothing} C_2 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{inc\}/\varnothing} C_5 \xrightarrow{\{done,inc\}/\varnothing} C_4 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_5 \xrightarrow{\{done,inc\}/\varnothing} C_4 \xrightarrow{\{poff\}/\varnothing} C_1$
ts_7	$C_1 \xrightarrow{\{pon\}/\varnothing} C_2 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{inc\}/\varnothing\}} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_5 \xrightarrow{\{done\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{done,inc\}/\varnothing} C_4 \xrightarrow{\{coffee\}/\{dec\}} C_5 \xrightarrow{\{dec\}/\varnothing} C_3 \xrightarrow{\{done,inc\}/\varnothing} C_4 \xrightarrow{\{inc\}/\varnothing} C_4 \xrightarrow{\{poff\}/\varnothing} C_1$

the unexecutable test sequences ts_1 and ts_2 cannot also be covered by any executable test sequence. Such def–use associations are called unexecutable, and thus will not be covered by the selected test sequences. In fact, the coverage of the executable def–use associations is advocated by Frankl and Weyuker [8] in the definition of the applicable versions of the data flow oriented test selection criteria of Rapps and Weyuker [9].

As a technical convenience, the method described here assumed that each variable appearing in statecharts has only one initial value. It is straightforward to deal with statecharts with multiple initial states, because the transformation methods from statecharts to EFSMs and from EFSMs to flow graphs do not depend on the values of variables. Now a test sequence of statecharts requires the selection of initial values for variables as well as the identification

of input, output and configuration sequences. The selection of initial values can be carried out by applying symbolic execution techniques to statecharts. Symbolic execution derives path conditions that consist of a system of inequalities on variables and hence can be used to find initial values that will drive a statechart along a particular test sequence. Although the solution to the system of inequalities provides a set of ranges of values for the variables occurring in the inequalities, specific values for these variables must be determined for the construction of an executable test sequence. Clearly, this is a relatively easy activity. However, in order to increase the error detection capability of a collection of test sequences and to obtain a definite fault coverage, the selection of values for variables must be based on a fault model. Such fault models must be formed for particular application domains, which is still an open problem. Moreover, the fault detection ability of particular collections of test data must be studied. Such studies can be formulated on the basis of the work proposed in Reference [26] that addresses the subdomain based test selection criteria. That is, a test selection criterion is viewed as a means of identifying subdomains of the input domain of a system where each test unit that must be covered identifies a subdomain of the input domain. The test selection criteria are then compared for relative effectiveness of fault coverage on the basis of their respective collections of subdomains.

6. CONCLUSIONS AND FUTURE WORK

This paper has demonstrated that conventional data flow analysis techniques can be applied to the selection of test sequences from specifications written in statecharts. A method is proposed that transforms a given statechart to an EFSM, called a normal form specification, containing all the possible runs of the statechart. The transformation method combined with the existing method of Ural *et al*., which transforms an EFSM into a flow graph, enables the association of a flow graph with a statechart that models the flow of both control and data in the statechart. A set of test sequences is selected from the resulting flow graph to cover all associations between definitions and uses of each variable appearing in the original statechart. These test sequences allow one to determine whether an implementation establishes the desired associations between definitions and uses expressed in the statechart.

The test sequence selection method can be fully automated. The research group of Ural *et al*. has already developed a toolset that automates the construction of flow graphs from EFSMs and the test sequence selection from flow graphs. Therefore, the only remaining task is the automated construction of EFSMs from statecharts. For that purpose the authors are currently investigating the use of library functions in the STATEMATE toolset for information retrieval of statecharts written using the STATEMATE editor. In addition to the test sequence selection, the determination of the executability of test sequences and variable values that make test sequences executable are mandatory for the whole test selection for statecharts. The application of symbolic execution techniques to statecharts is a prerequisite for supporting the determination.

Future work includes the development of methods for the selection of test sequences from statecharts with the complete set of language constructs provided in STATEMATE and with other semantics. The RSML semantics proposed by Leveson *et al*. [27] is close to the asynchronous time model of the STATEMATE semantics. Slight modifications to the test

sequence selection method proposed in this paper would suffice for the RSML semantics. For other statecharts semantics, significant changes would be necessary to reflect the different characteristics of the step construction method in these semantics. Other future work includes the development of normal form specifications for statecharts with real-time features such as timeout events and scheduled actions [10] and transitions with time intervals [28].

An interesting future research issue is the granularity of the single execution unit of statecharts. In general, there are three candidates that may be used as the single execution unit: transition, step and super-step. This paper implicitly assumed that step is the single execution unit and the flow of control and data was traced between steps. Alternatively, super-step may be used as the single execution unit. Many statecharts semantics are based on the synchrony hypothesis [23], which assumes that the response of a system to an external input is completed before another external input arrives. The asynchronous time model of the STATEMATE semantics fulfils the synchrony hypothesis because external events can be introduced only in stable states, while the synchronous one does not. It may be more natural to use super-step in the asynchronous time model rather than step as the single execution unit if one is only interested in tracing the flow of control and data between a system and its environment. Transition cannot be used as the single execution unit in the STATEMATE semantics. However, in statecharts with interleaving semantics [28] or the UML semantics [25], at most one transition may occur at any instant of time, and hence transition is the single execution unit. Statecharts with such semantics are of special interest because the transition explosion problem can be reduced in them.

ACKNOWLEDGEMENTS

This work was partially supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

REFERENCES

1 Harel D, Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 1987; **8**(3): 231–274.
2 von der Beeck M. A comparison of statecharts variants. In *Formal Techniques in Real-time Fault-tolerant Systems*, Lecture Notes in Computer Science, vol. 863, Springer Verlag, 1994; 128–148.
3 Bader A, Sajeev A, Ramakrishnan S. Testing concurrency and communication in distributed objects. In *Proceedings of the 5th International Conference on High Performance Computing* 1999; 422–428.
4 Bogdanov K, Holcombe M, Singh H. Automated test set generation for statecharts. In *Applied Formal Methods*, Lecture Notes in Computer Science, vol. 1641, Springer-Verlag, 1999; 107–121.
5 Kung D, Suchak N, Gao J, Hsia P, Toyoshima Y, Chen C. On object state testing. In *Proceedings of Computer Software and Applications Conference* 1994; 222–227.
6 Li L, Qi Z. Test selection from UML statecharts. In *Proceedings of the 31st International Conference on Technology of Object-oriented Languages and Systems*, 1999; 273–279.
7 Offutt J, Abdurazik A. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on the Unified Modeling Language*, 1999.
8 Frankl PG, Weyuker EJ. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 1988; **14**(10): 1483–1498.
9 Rapps S, Weyuker EJ. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 1985; **11**(4): 367–375.
10 Harel D, Naamad A. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 1996; **5**(4): 293–333.

11 von Bochmann G, Petrenko A. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis* 1994; 109–124.
12 Fernandez J-C, Jard C, Jéron T, Nedelka L, Viho C. Using on-the-fly verification techniques for the generation of test suites. In *Computer Aided Verification '96*, Lecture Notes in Computer Science, vol. 1102, Springer Verlag, 1996; 348–359.
13 Grabowski J, Hogrefe D, Scheurer R, Dai ZR. Applying SAM-STAG to the B-ISDN protocol SSCOP. In *Testing of Communicating Systems*, vol. 10, Chapman & Hall, 1997.
14 Huang S, Lee D, Staskauskas M. Validation based test sequence generation for networks of EFSMs. In *Proceedings of IFIP FORTE/PSTV '96.*
15 Kerbrat A, Jeron T, Groz R. Automated test generation from SDL specifications. In *Proceedings of SDL Forum* 1999; 135–151.
16 Miller RE, Paul S. Generating conformance test sequences for combined control and data flow of communication protocols. In *Proceedings of PSTV'92*; 13–27.
17 Sarikaya B, von Bochmann G, Cerny E. A test design methodology for protocol testing. *IEEE Transactions on Software Engineering*, 1987; **13**(5): 518–531.
18 Ural H. Test sequence selection based on static data flow analysis. *Computer Communications*, 1987; **10**(5): 234–242.
19 Ural H, Yang B. A test sequence selection method for protocol testing. *IEEE Transactions on Communications*, 1991; **39**(4): 514–523.
20 Ural H, Williams A, Test generation by exposing control and data dependencies within system specifications in SDL. In *Proceedings of IFIP 6th International Conference on Formal Description Techniques, FORTE'93*; 339–354.
21 ITU-T, Recommendation X. 904 — Information Technology — Open Distributed Processing — Reference Model: Architectural Semantics, December 1997.
22 Chan W, Anderson RJ, Beame P, Burns S, Modugno F, Notkin D, Reese JD. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 1998; **24**(7): 498–520.
23 Berry G, Gonthier G. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 1992; **19**(2): 87–152.
24 Heitmeyer CL, Jeffords RD, Labaw BG. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 1996; **5**(3): 231–261.
25 UML Proposal to the Object Management Group Version 1.1, Rational Corporation, September 1997. http://www.rational.com/uml/
26 Frankl PG, Weyuker EJ. A formal analysis of the fault detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 1993; **19**(3): 202–213.
27 Leveson NG, Heimdahl MPE, Hildreth H, Reese JD. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 1994; **30**(9): 684–707.
28 Kesten Y, Pnueli A. Timed and hybrid statecharts and their textual representation. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, vol. 571, Springer Verlag, 1992; 591–620.

