A Temporal Logic Based Theory of Test Coverage and Generation[*]

Hyoung Seok Hong[1**], Insup Lee[1], Oleg Sokolsky[1], and Hasan Ural[2]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] School of Information Technology and Engineering, University of Ottawa

Abstract. This paper presents a theory of test coverage and generation from specifications written in EFSMs. We investigate a family of coverage criteria based on the information of control flow and data flow and characterize them in the branching time temporal logic CTL. We discuss the complexity of minimal cost test generation and describe a method for automatic test generation which employs the capability of model checkers to construct counterexamples. Our approach extends the range of applications of model checking from formal verification of finite state systems to test generation from finite state systems.

1 Introduction

Testing has always been an essential activity for validating the correctness of software and hardware systems. Although testing cannot provide an absolute guarantee on correctness as is possible with formal verification, a disciplined use of testing can greatly increase the effectiveness of system validation, especially when performed by suitable tools. In this paper, we study the problem of test coverage and generation from specifications written in extended finite state machines (EFSMs). EFSMs extend finite state machines with variables and operations on them and are widely used as an underlying model of many specification languages such as SDL[2], Estelle[4], and Statecharts[12]. Because an EFSM specification typically allows an infinite number of executions, it is not possible to determine whether an implementation under test conforms to its specification by considering all executions of the specification. In the last two decades, a number of methods and tools have been proposed for test generation from EFSMs (for survey, see [3,8]) and most of them focus on a family of coverage criteria based on the information of control flow (e.g, states and transitions) and data flow (e.g., definitions and uses of variables).

We show that the problem of test generation from EFSMs based on control flow and data flow oriented coverage criteria can be formulated as a model checking problem. Given a system model and a temporal logic formula, model checking

[*] This research was supported in part by NSF CCR-9988409, NSF CCR-0086147, NSF CISE-9703220, ARO DAAD19-01-1-0473, and DARPA ITO MOBIES F33615-00-C-1707.

[**] Partially supported by the Advanced Information Technology Research Center (AITrc) at Korea Advanced Institute of Science and Technology (KAIST).

J.-P. Katoen and P. Stevens (Eds.): TACAS 2002, LNCS 2280, pp. 327–341, 2002.
© Springer-Verlag Berlin Heidelberg 2002

establishes whether the model satisfies the formula. If so, model checkers are capable of supplying a witness that explains the success of the formula. Conversely, if the model fails to satisfy the formula, a counterexample is produced. In our approach, each coverage criterion is associated with a set of temporal logic formulas and the problem of test generation satisfying the criterion is formulated as finding witnesses for every formula in the set with respect to a given EFSM. The capability of model checkers to construct witnesses and counterexamples allows test generation to be automatic.

We illustrate our approach using the temporal logic CTL[7]. First we define the semantics of EFSMs in terms of Kripke structures. We then describe how to express each coverage criterion as a set of formulas in CTL, parameterized with the propositions of a given EFSM. Each formula is defined such that the formula is satisfied by the EFSM if and only if the EFSM has an execution that covers the entity described by the formula such as a specific state, transition, or definition-use association[21]. If the entity can be covered in the EFSM, a witness for the corresponding formula is constructed. A test suite is a set of finite executions of the EFSM such that for every formula, the test suite includes a finite execution which is a witness for the formula. In addition to the coverage criteria that cover states, transitions, and definition-use associations, we also consider more complex ones that are based on the affect relation in program slicing[24] and are applied to protocol conformance testing[22]. They deal with data flow from input variables to output variables through an arbitrary number of definition-use associations between local variables. Hence they cannot be characterized as CTL formulas and we characterize them as least fixpoints of predicate transformers over CTL formulas. Witnesses for such least fixpoints can be constructed in the way similar to CTL formulas.

We then discuss the problem of minimal test generation. Typically, a CTL formula can be represented by several different witnesses. By selecting the right witness for each formula, one can minimize the size of the test suite according to two costs: the number of test sequences in the suite or the total length of test sequences in the suite. We show that these optimization problems are NP-hard and describe a simple heuristic similar to the test generation method in [10], which enables the application of existing CTL model checkers such as SMV[19] to automatic test generation.

Related Work. Widely-used system models in the testing literature include finite state machines (FSMs) and labelled transition systems (LTSs), especially in hardware testing and protocol conformance testing. Testing methods based on such models primarily focus on control flow oriented test generation (for survey, see [3,8,17]). Although these methods are well-suited for hardware circuits and control portions of communication protocols, they are not powerful enough to test complex data-dependent behaviors.

EFSMs extend FSMs with variables to support the succinct specification of data-dependent behaviors. If the state space of an EFSM is finite, one can construct the equivalent FSM by unfolding the values of variables. Thus, EFSM-based testing with finite state space can be reduced in principle to ordinary

FSM-based testing. Of course, this approach suffers from the well-known state explosion problem which makes test generation often impractical. Even when test generation is feasible, this approach is often impractical because of the test explosion problem, i.e., the number of generated tests might be too large to be applied to implementations. A promising alternative is to apply conventional software testing techniques to test generation from EFSMs [22]. In this approach, an EFSM is transformed into a flow graph that models the flow of both control and data in the EFSM and tests are generated from the graph by identifying the flow information. The approach abstracts the values of variables when constructing flow graphs and hence it can be applicable even if the state space is infinite. However, it requires posterior analysis such as symbolic execution or constraint solving to determine the executability of tests and for the selection of variable values which make tests executable.

The approach we advocate here is based on constructing Kripke structures from EFSMs and hence also suffers from state explosion. Our approach, however, enables the use of symbolic model checking[5] that has been shown to be effective for controlling state explosion for certain problem domains. Second, our approach overcomes the test explosion problem by using control and data flow information of EFSMs like the flow-graph approach. Finally, our approach can be seen as complementary to the flow-graph approach. In particular, flow graphs can be constructed from system models whose state space is infinite, whereas our approach has the advantage that only executable tests are generated which obviates the need of posterior analysis. Ideally, one would eventually like to be able to combine these two approaches.

Recently, connection between test generation and model checking has been considered in the testing literature. [11,20] use binary decision diagrams (BDDs) to represent EFSMs and describe symbolic approaches to test generation for state and transition coverage criteria. [14] describes a test generation method by adapting local or on-the-fly model checking algorithms. [23] describes an on-the-fly test generation method which utilizes SPIN[13] to generate the information necessary for test generation. Test generation using the capability of model checkers to construct counterexamples has been applied in several contexts. [1] describes the application of model checking to mutation analysis. [6,9] generate tests by constructing counterexamples for user-specified temporal formulas. No consideration is given to coverage criteria. [10] generates tests from SCR specifications using two model checkers SMV and SPIN for control flow oriented coverage criteria, which are similar to transition coverage criterion. We are not aware of any work that considers the model checking approach to both control flow and data flow oriented coverage criteria.

2 Logic: CTL

Syntax. CTL[7] is a branching time temporal logic widely-used for symbolic model checking. Formulas in CTL are built from atomic propositions, boolean connectives, path quantifiers **A** (for all paths) and **E** (for some path), and modal operators $\mathbf{X}$ (next time), $\mathbf{U}$ (until), $\mathbf{F}$ (eventually), and $\mathbf{G}$ (always). Formally, CTL is the set of state formulas defined as follows:

– Every atomic proposition is a state formula,
– If f and g are state formulas, then $\neg f$, $f \wedge g$ are state formulas,
– If f and g are state formulas, then $\mathbf{X}f$, $f\mathbf{U}g$, and $\mathbf{G}f$ are path formulas,
– If f is a path formula, then $\mathbf{E}f$ is a state formula.

The remaining formulas are defined by: $\mathbf{EF}f \equiv \mathbf{E}[true\mathbf{U}f]$, $\mathbf{AX}f \equiv \neg\mathbf{EX}\neg f$, $\mathbf{A}[f\mathbf{U}g] \equiv \neg\mathbf{E}[\neg g\mathbf{U}\neg f\wedge\neg g] \wedge \neg\mathbf{EG}\neg g$, $\mathbf{AF}f \equiv \neg\mathbf{EG}\neg f$, $\mathbf{AG}f \equiv \neg\mathbf{EF}\neg f$.

Semantics. The semantics of CTL is defined with respect to a *Kripke structure* $M = (Q, Q_0, L, R)$ where Q is a finite set of states; $Q_0 \subseteq Q$ is the set of initial states; L: $Q \rightarrow 2^{AP}$ is the function labeling each state with a set of atomic propositions in AP; and $R \subseteq Q \times Q$ is the transition relation. A sequence q_0, q_1, q_2, ... of states is a *path* if $(q_i, q_{i+1}) \in R$ for all $i \geq 0$. Given a path π and an integer i, $\pi(i)$ denotes the i-th state of π. The satisfaction relation $\models$ is inductively defined as follows:

– $M, q \models p$ if $p \in L(q)$;
– $M, q \models \neg\phi$ if $\neg(q \models \phi)$;
– $M, q \models \phi \wedge \phi'$ if $q \models \phi$ and $q \models \phi'$;
– $M, q \models \mathbf{E}f$ if $\pi \models f$ for some path π such that $\pi(0) = q$;
– $M, \pi \models \mathbf{X}f$ if $\pi(1) \models f$;
– $M, \pi \models f\mathbf{U}g$ if $\pi(i) \models g$ for some $i \geq 0$ and $\pi(j) \models f$ for all $0 \leq j < i$;
– $M, \pi \models \mathbf{G}f$ if $\pi(i) \models f$ for all $i \geq 0$.

We write $M \models f$ if $M, q_0 \models f$ for every initial state $q_0 \in Q_0$.

Witnesses. One of the important features of model checking is the ability to generate witnesses and counterexamples. If a formula $\mathbf{E}f$ is true, we can demonstrate the success of the formula by finding a witness which is a path π such that $\pi \models f$. Likewise, if a formula $\mathbf{A}f$ is false, there is a counterexample π such that $\pi \models \neg f$. We observe that a witness for a formula of the form $\mathbf{E}f$ is also a counterexample for its negation $\neg\mathbf{E}f$. In general, a witness or counterexample is a set of infinite paths. For example, to demonstrate the success of $\mathbf{EG}p_1 \wedge \mathbf{EG}p_2$ or the failure of $\mathbf{AF}\neg p_1 \vee \mathbf{AF}\neg p_2$, we must find two infinite paths π_1 and π_2 such that $\pi_1 \models \mathbf{G}p_1$ and $\pi_2 \models \mathbf{G}p_2$. However, if we consider a subclass of CTL, which we call WCTL, then it is guaranteed that every witness is a finite path. A CTL formula f is a WCTL formula if (i) f is in positive normal form, i.e., every negation in f is applied only to atomic propositions, (ii) f contains only $\mathbf{EX}$ and $\mathbf{EU}$, and (iii) for every subformula of f of the form $f_1 \wedge ... \wedge f_n$, every conjunct f_i except at most one is an atomic proposition. For example, $\mathbf{EF}(p_1 \wedge \mathbf{EF}p_2)$ is a WCTL formula, while $\mathbf{EF}(\mathbf{EF}p_1 \wedge \mathbf{EF}p_2)$ is not.

For a WCTL formula f and a Kripke structure M such that $M \models f$, we define the set of *witnesses* for f with respect to M, denoted by $\mathcal{W}(M, f)$, as follows:

- $W(M, true) = Q$,
- $W(M, p \wedge f) = \{q_0 \mid q_0 \models p\} * W(M, f)$,
- $W(M, f \vee g) = W(M, f)$ or $W(M, f \vee g) = W(M, g)$,
- $W(M, \mathbf{EX}f) = \{q_0 q_1 \mid q_1 \models f\} * W(M, f)$,
- $W(M, \mathbf{E}[f\mathbf{U}g]) = \{q_0 q_1 ... q_n \mid q_i \models f$ for all $0 \leq i < n$ and $q_n \models g\} * W(M, g)$,
- $\mathcal{W}(M, f) = \{\pi \in W(M, f) \mid \pi(0) \in Q_0\}$,

where $\Pi_1 * \Pi_2 = \{\pi \mid \exists i : \pi_i \in \Pi_1, \pi^i \in \Pi_2\}$, π_i denotes the prefix of π ending at $\pi(i)$, and π^i denotes the suffix of π starting from $\pi(i)$. We extend the notion of witnesses to a set of WCTL formulas. A set Π of finite paths is a *witness-set* for a set F of WCTL formulas with respect to M if, for every formula f in F such that $M \models f$, there exists a finite path π in Π that is a witness for f. Note that Π is a witness-set for F with respect to M if and only if it is a witness-set for $\{f \in F \mid M \models f\}$, or equivalently $F \backslash \{f \in F \mid M \not\models f\}$.

3 Model: EFSM

Syntax. An *extended finite state machine* (EFSM) is a tuple $G = (S, S_0, E, V, T)$ where S is a finite set of states; $S_0 \subseteq S$ is the set of initial states; E is a finite set of events; V is a finite set of variables partitioned into three disjoint subsets V_I, V_L, and V_O comprising input, local, and output variables, respectively; T is a finite set of transitions. A transition is a tuple (s, e, g, A, s') where $s, s' \in S$, $e \in E$, g is a predicate on $V_I \cup V_L$ and A is a set of assignments to $V_L \cup V_O$. In this paper, we consider only deterministic EFSMs. An EFSM is *deterministic* if, for every state s and event e, $g_i \wedge g_j = \mathit{false}$ for all $1 \leq i, j \leq n, i \neq j$, where g_1, ..., g_n are the guards of the transitions whose source state is s and event is e. Figure 1 shows a simple coffee vending machine which has $S = \{$IDLE, BUSY$\}$, $S_0 = \{$IDLE$\}$, $E = \{$*insert*, *coffee*, *done*, *display*$\}$, $V_I = \{x\}$, $V_L = \{m\}$, and $V_O = \{y\}$. We assume x, m, and y are of integer subrange [0..5].

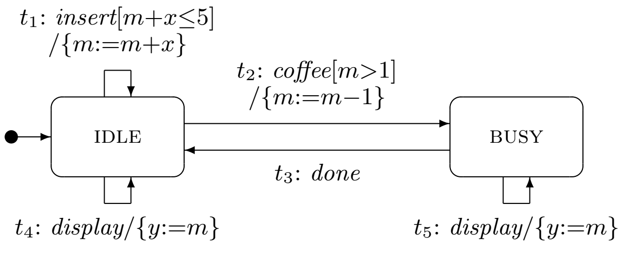
t_1: *insert*[m+x≤5]
/{m:=m+x}
t_2: *coffee*[m>1]
/{m:=m−1}
IDLE
BUSY
t_3: *done*
t_4: *display*/{y:=m}
t_5: *display*/{y:=m}

Fig. 1. An example of EFSMs

Local variables can be defined and used by the EFSM while input variables can only be used and output variables can only be defined. Formally, a variable

v is *defined* at a transition $t = (s, e, g, A, s')$, denoted by d_t^v, if v occurs in the left hand side of an assignment in A, and v is *used* at t, denoted by u_t^v, if v occurs in the guard g or in the right hand side of an assignment in A. For two variables v, v', and a transition t, we say that u_t^v *directly affects* $d_t^{v'}$ at t, denoted by $da_t^{v,v'}$, if v occurs in the guard of t or in the right hand side of the assignment of t whose left hand side is v'. For a transition t, define $DEF(t)$, $USE(t)$, and $DA(t)$ as the sets of definitions, uses, and directly affects occurring at t, respectively. Define $DEF(G)$, $USE(G)$, and $DA(G)$ as $\bigcup_{t \in T} DEF(t)$, $\bigcup_{t \in T} USE(t)$, and $\bigcup_{t \in T} DA(t)$, respectively. Table 1 shows the classification of the variables in Figure 1 as definitions, uses, and directly affects.

Table 1. The definitions, uses, and directly affects in the coffee vending machine

transitions	$DEF(t)$	$USE(t)$	$DA(t)$
t_1	$\{d_{t_1}^m\}$	$\{u_{t_1}^x,\, u_{t_1}^m\}$	$\{da_{t_1}^{x,m},\, da_{t_1}^{m,m}\}$
t_2	$\{d_{t_2}^m\}$	$\{u_{t_2}^m\}$	$\{da_{t_2}^{m,m}\}$
t_3	$\emptyset$	$\emptyset$	$\emptyset$
t_4	$\{d_{t_4}^y\}$	$\{u_{t_4}^m\}$	$\{da_{t_4}^{m,y}\}$
t_5	$\{d_{t_5}^y\}$	$\{u_{t_5}^m\}$	$\{da_{t_5}^{m,y}\}$

Semantics. For a set V of variables, a valuation σ over V is a function mapping variables to their values. The set of valuations over V is denoted by Σ_V. For a set A of assignments, $A(\sigma)$ denotes the valuation defined by $A(\sigma)(v) = \mathit{value}$ if there exists an assignment of the form $v{:=}\mathit{exp}$ in A and value is the value of exp evaluated over σ, and $A(\sigma)(v) = \sigma(v)$ otherwise.

We view EFSMs as Kripke structures to characterize the problems of test coverage and generation in CTL. We call each element in Q of a Kripke Structure a *global state* to distinguish it from a state of EFSMs. Similarly, we call each element in R a *global transition*. The Kripke structure corresponding to an EFSM G is $(S \times E \times \Sigma_V \times (T \cup \{\emptyset\}),\ S_0 \times E \times \Sigma_V \times \{\emptyset\},\ L,\ R)$ where

- for every $(s, e, \sigma, t) \in S \times E \times \Sigma_V \times (T \cup \{\emptyset\})$, $L((s, e, \sigma, t)) = \{s\} \cup \{e\} \cup \{v{=}\sigma(v) \mid v \in V\} \cup \{t\} \cup \{d_t^v \mid d_t^v \in DEF(t)\} \cup \{u_t^v \mid u_t^v \in USE(t)\} \cup \{da_t^{v,v'} \mid da_t^{v,v'} \in DA(t)\}$,
- $((s, e, \sigma, t),\ (s', e', \sigma', t')) \in R$ if and only if there exists a transition $t' = (s, e, g, A, s')$ satisfying $\sigma \models g$ and $\sigma' = A(\sigma)$.

A global state (s, e, σ, t) captures (i) the current state in which the EFSM is, (ii) the event generated, (iii) the values of variables, and (iv) the transition taken. A global transition $((s, e, \sigma, t),\ (s', e', \sigma', t'))$ represents the execution of its corresponding transition t'.

Test Sequences. Since it is impossible to test infinite executions, we define a *test sequence* of an EFSM as a finite path of its Kripke structure. A *test suite* is a finite set of test sequences. Moreover, we require that the execution of every test sequence end at a specific state, if the state is designated by a tester as the *exit* state of the EFSM. If an initial state of an EFSM is reachable from every state, we oftern require a test sequence end at the initial state because it is convenient to execute another test sequence without resetting an implementation under test into the initial state. In general, a tester may designate an arbitrary state as the exit state and distinguish test sequences ending at that state from others by interpreting the sequences as completed tasks of the EFSM.

4 Test Coverage

This section investigates a family of coverage criteria for EFSMs and characterizes them in terms of witness-sets. For the remainder of the paper, we fix an EFSM G with exit condition *exit*, denoted by $\langle G, exit\rangle$. The condition *exit* is defined as s_e if s_e is the exit state designated by a tester, and *true* otherwise.

4.1 Control Flow Oriented Coverage Criteria

Obviously, the strongest coverage criterion for determining the conformance of an implementation to its EFSM specification is *path coverage* which requires that all paths of the Kripke structure corresponding to the EFSM be traversed. Because there is an infinite number of paths, it is impossible to achieve exhaustive testing and we need to have coverage criteria that select a reasonable and finite number of test sequences. Included are control flow oriented coverage criteria that require that every state or transition be traversed at least once during testing.

State Coverage. A state s of $\langle G, exit\rangle$ is *testable* if there exists a test sequence $q_0 ... q_n$ such that $q_i \models s$ for some i and $q_n \models exit$. In this case, the test sequence is said to *cover s*. It is easy to see that a test sequence covers s if and only if it is a witness of $\mathbf{EF}(s \wedge \mathbf{EF} exit)$, because the set of witnesses for the formula is $\{q_0 ... q_n \mid q_i \models s \wedge \mathbf{EF}(exit)$ for some i and $q_n \models exit\}$.

A test suite Π of $\langle G, exit\rangle$ satisfies *state coverage criterion* if every testable state is covered by a test sequence in Π. We characterize test suites satisfying state coverage criterion as follows. A test suite Π of $\langle G, exit\rangle$ satisfies state coverage criterion if and only if it is a witness-set for

$$\{\mathbf{EF}(s \wedge \mathbf{EF} exit) \mid s \in S\}$$

Note that Π is a witness-set for $\{\mathbf{EF}(s \wedge \mathbf{EF} exit) \mid s \in S\}$ if and only if it is a witness-set for $\{\mathbf{EF}(s \wedge \mathbf{EF} exit) \mid s \in S$ and s is testable$\}$.

Transition Coverage. A transition t of $\langle G, exit\rangle$ is *testable* if there exits a test sequence $q_0 ... q_n$ such that $q_i \models t$ for some i and $q_n \models exit$. In this case, the test sequence is said to *cover t*. A test suite Π of $\langle G, exit\rangle$ satisfies *transition coverage criterion* if every testable transition is covered by a test sequence in Π. A test suite Π satisfies transition coverage criterion if and only if it is a witness-set for

$$\{\mathbf{EF}(t \wedge \mathbf{EF} exit) \mid t \in T\}$$

4.2 Data Flow Oriented Coverage Criteria

Data flow oriented coverage criteria establish associations between definitions and uses of variables and require that these associations are examined at least once during testing. We consider two types of associations: definition-use pairs and affect pairs that are centeral notions in data flow analysis and program slicing, respectively.

Data Flow among Local Variables. For a definition d^v_t and use u^v_t of the same variable v, we say that $(d^v_t, u^v_{t'})$ is a *definition-use pair* (in short, du-pair) if there exists a test sequence $q_0 ... q_n$ such that $q_i \models d^v_t$ and $q_j \models u^v_{t'}$ for some $0 \leq i < j \leq n$, and $q_k \models \neg def(v)$ for all $i < k < j$, where $def(v) = \bigvee_{d^v_t \in DEF(G)} d^v_t$. In addition, if $q_n \models exit$, the du-pair is *testable.* In this case, the test sequence is said to *cover* $(d^v_t, u^v_{t'})$ and the subpath $q_i ... q_j$ is called a *definition-clear path* of $(d^v_t, u^v_{t'})$. It can be shown that a test sequence covers $(d^v_t, u^v_{t'})$ if and only if it is a witness of $\mathbf{EF}(d^v_t \wedge \mathbf{EXE}[\neg \mathit{def}(v) \mathbf{U}(u^v_{t'} \wedge \mathbf{EF} \mathit{exit})])$. Table 2 shows the du-pairs in Figure 1. For example, $(d^m_{t_1},\, u^m_{t_4})$ is a du-pair whereas $(d^m_{t_1},\, u^m_{t_5})$ is not because there is no definition-clear path with respect to m from t_1 to t_5.

Table 2. The du-pairs in the coffee vending machine

variables	du-pairs
m	$(d^m_{t_1},\, u^m_{t_1})$, $(d^m_{t_1},\, u^m_{t_2})$, $(d^m_{t_1},\, u^m_{t_4})$, $(d^m_{t_2},\, u^m_{t_1})$, $(d^m_{t_2},\, u^m_{t_2})$, $(d^m_{t_2},\, u^m_{t_4})$, $(d^m_{t_2},\, u^m_{t_5})$

All-def Coverage. A test suite Π of $\langle G, exit \rangle$ satisfies *all-def coverage criterion* if, for every definition d^v_t, some testable du-pair $(d^v_t, u^v_{t'})$ is covered by a test sequence in Π. A test suite Π satisfies *all-def coverage criterion* if and only if it is a witness-set for

$$\{ \bigvee_{u^v_{t'} \in USE(G)} \mathbf{EF}(d^v_t \wedge \mathbf{EXE}[\neg \mathit{def}(v) \mathbf{U} u^v_{t'} \wedge \mathbf{EF} \mathit{exit})]) \mid d^v_t \in DEF(G) \}$$

All-use Coverage. A test suite Π of $\langle G, exit \rangle$ satisfies *all-use coverage criterion* if, for every definition d^v_t, every testable du-pair $(d^v_t, u^v_{t'})$ is covered by a test sequence in Π. A test suite Π satisfies *all-use coverage criterion* if and only if it is a witness-set for

$$\{ \mathbf{EF}(d^v_t \wedge \mathbf{EXE}[\neg \mathit{def}(v) \mathbf{U}(u^v_{t'} \wedge \mathbf{EF} \mathit{exit})]) \mid d^v_t \in DEF(G), u^v_{t'} \in USE(G) \}$$

Data Flow among Input and Output Variables. For a use u^v_t of variable v and a definition $d^{v'}_{t'}$ of variable v', we say that u^v_t *affects* $d^{v'}_{t'}$ if (i) either $t = t'$ and u^v_t directly affects $d^{v'}_t$, or (ii) there exists a du-pair $(d^{v''}_t, u^{v''}_{t''})$ such that u^v_t directly affects $d^{v''}_t$ and $u^{v''}_{t''}$ affects $d^{v'}_{t'}$. We say that $(u^v_t,\, d^{v'}_{t'})$ is an *affect-pair* if u^v_t affects $d^{v'}_{t'}$. A *data-flow chain* (in short df-chain) of an affect-pair $(u^v_t,\, d^{v'}_{t'})$ is a sequence of du-pairs $(d^{v_1}_{t_1},\, u^{v_1}_{t_2})$ $(d^{v_2}_{t_2},\, u^{v_2}_{t_3})$, ..., $(d^{v_n}_{t_n},\, u^{v_n}_{t_{n+1}})$, $n \geq 0$, such that

- $t_1 = t$ and $u^v_{t_1}$ directly affects $d^{v_1}_{t_1}$, $t_{n+1} = t'$ and $u^{v_n}_{t_{n+1}}$ directly affects $d^{v'}_{t_{n+1}}$, and for every $1 \leq i < n$, $u^{v_i}_{t_{i+1}}$ directly affects $d^{v_{i+1}}_{t_{i+1}}$,
- there exists a test sequence $q_0...q_m$ such that for every $1 \leq i \leq n$, there exists a subpath π_i of $q_0...q_m$ satisfying $\mathit{last}(\pi_i) = \mathit{first}(\pi_{i+1})$ and π_i is a definition-clear path of $(d^{v_i}_{t_i}, u^{v_i}_{t_{i+1}})$.

In addition, if $q_m \models \mathit{exit}$, the affect-pair $(u^v_t, d^{v'}_{t'})$ is *testable*. In this case, the test sequence is said to *cover* $(u^v_t, d^{v'}_{t'})$. Table 3 shows the affect-pairs in Figure 1. For example, from the affect-pair $(u^x_{t_1}, d^y_{t_4})$, we observe that the use of x at t_1 affects the definition of y at t_4 through a df-chain, say $(d^m_{t_1}, u^m_{t_4})$.

Table 3. The affect-pairs in the coffee vending machine

variables	affect-pairs
x, m	$(u^x_{t_1}, d^m_{t_1})$, $(u^x_{t_1}, d^m_{t_2})$,
x, y	$(u^x_{t_1}, d^y_{t_4})$, $(u^x_{t_1}, d^y_{t_5})$,
m, m	$(u^m_{t_1}, d^m_{t_1})$, $(u^m_{t_1}, d^m_{t_2})$, $(u^m_{t_2}, d^m_{t_1})$, $(u^m_{t_2}, d^m_{t_2})$,
m, y	$(u^m_{t_1}, d^y_{t_4})$, $(u^m_{t_1}, d^y_{t_5})$, $(u^m_{t_2}, d^y_{t_4})$, $(u^m_{t_2}, d^y_{t_5})$, $(u^m_{t_4}, d^y_{t_4})$, $(u^m_{t_5}, d^y_{t_5})$

In contrast to du-pairs, affect-pairs cannot be characterized in terms of WCTL formulas because they require an arbitrary number of du-pairs. Instead, we characterize them using a least fixpoint of an appropriate predicate transformer over WCTL formulas. Note that the computation of fixpoints can be implemented efficiently in symbolic model checking.

For a testable affect-pair $(u^v_t, d^{v'}_{t'})$, we use $Q(u^v_t, d^{v'}_{t'})$ to denote the set of global states q_0 such that $q_0 \models u^v_t$ and there exists a test sequence $q_0 q_1...$ covering the affect-pair. By the definition of affect-pairs, we have the following equation.

$$Q(u^v_t, d^{v'}_{t'}) = (u^v_t \wedge da^{v,v'}_t \wedge \mathbf{EF}\mathit{exit}) \vee$$

$$(u^v_t \wedge \bigvee_{v'' \in DA(t,v)} \mathbf{EXE}[\neg def(v'') \; \mathbf{U} \bigvee_{u^{v''}_{t''} \in USE(G)} Q(u^{v''}_{t''}, d^{v'}_{t'})])$$

where $DA(t, v)$ is the set of variables directly affected by v at t.

We identify every WCTL formula f with the predicate $\{q \mid M, q \models f\}$ in 2^Q. Let $\tau : 2^Q \rightarrow 2^Q$ be a predicate transformer defined as follows.

$$\tau(Z) = (u^v_t \wedge da^{v,v'}_t \wedge \mathbf{EF}\mathit{exit}) \vee$$

$$(u^v_t \wedge \bigvee_{v'' \in DA(t,v)} \mathbf{EXE}[\neg def(v'') \; \mathbf{U} \bigvee_{u^{v''}_{t''} \in USE(G)} Z[v''/v, t''/t]])$$

where $Z[v''/v, t''/t']$ is the formula obtained by replacing each occurrence of v and t in Z by v'' and t'', respectively.

Theorem 1 *$Q(u_t^v,\, d_{t'}^{v'})$ is the least fixpoint of τ.*

Proof It is easy to see that τ is monotonic.

Let Z_f be $Q(u_t^v,\, d_{t'}^{v'})$. Suppose that $q_0 \models \tau(Z_f)$, then there exists a path $q_0q_1...$ such that either $q_0 \models (u_t^v \wedge da_t^{v,v'} \wedge \mathbf{EF}\mathit{exit})$, that is, u_t^v directly affects $d_{t'}^{v'}$, or $q_0 \models (u_t^v \wedge \bigvee_{v'' \in DA(t,v)} \mathbf{EXE}[\neg def(v'')\mathbf{U} \bigvee_{u_{t''}^{v''} \in USE(G)} Z_f[v''/v, t''/t]])$, that is, there exists a du-pair $(d_t^{v''}, u_{t''}^{v''})$ such that u_t^v directly affects $d_t^{v''}$ and $u_{t''}^{v''}$ affects $d_{t'}^{v'}$. Hence, $q_0 \models u_t^v$ and $q_0q_1...$ covers $(u_t^v,\, d_{t'}^{v'})$, that is, $q_0 \models Z_f$. Therefore, we have $\tau(Z_f) \subseteq Z_f$. Similarly, we can show that if $q_0 \models Z_f$, then $q_0 \models \tau(Z_f)$. Consequently, Z_f is a fixpoint of τ.

To prove that Z_f is the least fixpoint of τ, it is sufficient to show that $Z_f = \cup_i \tau^i(\mathit{false})$, where $\tau^0(Z) = Z$ and $\tau^{i+1}(Z) = \tau(\tau^i(Z))$. It is easy to show by induction on i that for every i, $\tau^i(\mathit{false}) \subseteq Z_f$. Hence, we have the first direction $\cup_i \tau^i(\mathit{false}) \subseteq Z_f$. The other direction, $Z_f \subseteq \cup_i \tau^i(\mathit{false})$, is shown by induction on the number of du-pairs of the df-chain of $(u_t^v,\, d_{t'}^{v'})$. Suppose that $q_0 \models Z_f$, then there exists a path $q_0q_1...$ covering $(u_t^v,\, d_{t'}^{v'})$. Let $j \geq 0$ be the number of du-pairs of in the df-chain of $(u_t^v,\, d_{t'}^{v'})$. We show by induction on j that for every $j \geq 0$, $q_0 \in \tau^{j+1}(\mathit{false})$. For the base case, suppose that $j = 0$, that is, u_t^v directly affects $d_{t'}^{v'}$. Then $q_0 \models (u_t^v \wedge da_t^{v,v'} \wedge \mathbf{EF}\mathit{exit})$ and hence $q_0 \in \tau^1(\mathit{false})$. For the inductive step, suppose that $q_0 \in \tau^{j+1}(\mathit{false})$ for $j = n$. Let $j = n+1$ and q_k be the global state in the path $q_0q_1...$ at which the first du-pair in the df-chain ends. Hence, there exist n du-pairs from q_k and we have that $q_k \in \tau^{n+1}(\mathit{false})$ by the induction hypothesis. Therefore, $q_0 \in (u_t^v \wedge \bigvee_{v'' \in DA(t,v)} \mathbf{EXE}[\neg def(v'')\ \mathbf{U} \bigvee_{u_{t''}^{v''} \in USE(G)} \tau^{n+1}(\mathit{false})[v''/v, t''/t]])$ and $q_0 \in \tau^{n+2}(\mathit{false})$. □

Among the particular affect-pairs of interest to our coverage criteria are those starting with an input variable and ending with an output variable. We say that an affect-pair $(u_t^i,\, d_{t'}^o)$ is an *io-pair* if i is an input variable and o is an output variable. For example, in Table 3, there are two io-pairs $(u_{t_1}^x,\, d_{t_4}^y)$ and $(u_{t_1}^x,\, d_{t_5}^y)$, that is, the use of x at t_1 affects the definition of y at t_4 and t_5. The rationale here is to identify functionality specified by the EFSM in terms of the effects of input variables accepted from its environment on output variables offered to its environment.

All-input Coverage. A test suite Π of $\langle G, exit\rangle$ satisfies *all-input coverage criterion* if, for every use u_t^i of every input variable i, some testable io-pair $(u_t^i, d_{t'}^o)$ is covered by a test sequence in Π. A test suite Π satisfies *all-input coverage criterion* if and only if it is a witness-set for

$$\{ \bigvee_{d_{t'}^o \in DEF(G), o \in V_O} \mathbf{EF}Q(u_t^v, d_{t'}^{v'}) \mid u_t^i \in USE(G), \mid i \in V_I \}$$

All-output Coverage. A test suite Π of $\langle G, exit\rangle$ satisfies *all-output coverage criterion* if, for every use u_t^i of every input variable i, every testable io-pair

$(u^i_t, d^o_{t'})$ is covered by a test sequence in Π. A test suite Π satisfies *all-output coverage criterion* if and only if it is a witness-set for

$$\{\mathbf{EF}Q(u^v_t, d^{v'}_{t'}) \mid u^i_t \in USE(G), i \in V_I, d^o_{t'} \in DEF(G), o \in V_O\}$$

5 Test Generation

This section defines two optimization problems of minimal cost test generation. They are shown to be NP-hard and a heuristic algorithm is described.

5.1 Complexity

To generate a test suite for a given EFSM and coverage criterion, we construct a Kripke structure M corresponding to the EFSM and a set F of WCTL formulas (or WCTL formulas with a least fixpoint operator). We wish to generate a minimal test suite Π with respect to one of the two costs: (i) the number of test sequences in Π or (ii) the total length of test sequences in Π. After finishing the execution of a test sequence, an implementation under test should be reset into its initial state from which another test sequence can be applied. It is appropriate to use the first cost if the reset operation is expensive, and the second one otherwise.

Let $\mathcal{W}_f$ be the set of witnesses for a formula f in F. First we consider the Minimal Number Test Generation (MNTG) problem which is an optimization problem defined by: given a collection of sets $\mathcal{W}_f$, generate a minimal witness-set Π in the number of witnesses in Π. We show this problem to be NP-hard by considering its corresponding decision problem MNTG$'$: given a collection of $\mathcal{W}_f$ and positive integer k, is there a witness-set Π with $|\Pi| \leq k$? We prove that MNTG$'$ is NP-complete by reducing the Hitting Set problem, which is known to be NP-complete[15], to MNTG$'$. The Hitting Set problem is defined by: given a collection of subsets C_i of a finite set S and positive integer k, is there a subset $S' \subseteq S$, called *hitting set*, such that $|S'| \leq k$ and S' contains at least one element from each C_i?

Theorem 2 *MNTG$'$ is NP-complete.*

Proof It is easy to show that MNTG$'$ is in NP. Given an instant of the Hitting Set problem, we construct a Kripke structure (Q, Q_0, L, R) such that $Q = \{q_0\} \cup \{q_c \mid c \in \bigcup C_i\}$, $Q_0 = \{q_0\}$, and $R = \{(q_0, q_c) \mid c \in \bigcup C_i\}$. This reduction is linear in the size of S. For every subset C_i, we construct a set $\mathcal{W}_i$ of witnesses as follows: $q_0 q_c$ is in $\mathcal{W}_i$ if and only if $c \in C_i$. Clearly, there exists a hitting set S' with $|S'| \leq k$ for the collection of C_i if and only if there exists a witness-set $\Pi = \{q_0 q_s \mid s \in S'\}$ with $|\Pi| \leq k$ for the collection of W_i. ☐

Second we consider the Minimal Length Test Generation (MLTG) problem defined by: given a collection of $\mathcal{W}_f$, generate a minimal witness-set Π in the total length of witnesses in Π. Its corresponding decision problem MLTG$'$ is defined by: given a collection of sets $\mathcal{W}_f$ and positive integer k, is there a witness-set Π such that $\sum_{\pi \in \Pi} |\pi| \leq k$?

Theorem 3 *MLTG$'$ is NP-complete.*

Proof It is easy to show that MLTG$'$ is in NP. We use the same reduction used as in Theorem 2. Since all paths in Q are of length one, the minimum total-length of the witness-set Π is achieved when Π contains the minimum number of witnesses. Therefore, a solution for the MLTG problem in this case will yield the same witness-set which also is a solution to the MNTG problem. Hence there exists a hitting set S' with $|S'| \leq k$ if and only if there exists a witness-set Π with $\sum_{\pi \in \Pi} |\pi| \leq k$. □

5.2 Heuristic

Because of NP-hardness of the problems, we do not expect optimal solutions to them. Instead we describe a greedy algorithm which can be applied to both MNTG and MLTG problems. Figure 2 shows how the greedy algorithm is applied to state coverage criterion. The algorithm can also be applied to other coverage criteria by changing the set of covered entities to transitions, du-pairs, or io-pairs.

```
Input: a set $F$ of formulas and a Kripke structure $M$
Output: a test suite $\Pi$ satisfying state coverage criterion

mark every state in $\mathcal{S}$ as uncovered;
$\Pi := \emptyset$;
repeat
    choose a state $s \in \mathcal{S}$ marked as uncovered;
    model check the negation of $f = \mathbf{EF}(s \wedge \mathbf{EF}exit)$ in $F$ against $M$;
    if $M \models \neg f$
        mark $s$ as untestable;
    else /* $M \not\models \neg f$ */
        let $\pi$ be the counterexample for $\neg f$ (equivalently the witness for $f$);
        let $S_\pi$ be the set of states covered by $\pi$;
        mark every state in $S_\pi$ as covered;
        $\Pi := \Pi \cup \{\pi\}$;
        for all $\pi' \in \Pi$ such that $S_{\pi'} \subset S_\pi$
            $\Pi := \Pi - \{\pi'\}$;
until every state in $\mathcal{S}$ is marked as covered or untestable
return $\Pi$;
```

Fig. 2. A greedy algorithm for state coverage criterion

In the algorithm, we directly employ the capability of model checkers to construct counterexamples because a witness for a WCTL formula or a formula of the form $\mathbf{EF}Q(u_t^v,\, d_{t'}^{v'})$ is also a counterexample for its negation. Basically we generate a witness for every formula f in F by model checking the negation $\neg f$ and constructing its counterexample. The resulting set of witnesses constitutes a test suite. This naive method would generate a number of redundant witnesses because a witness may cover more than one state at the same time. We remove such redundant witnesses by considering only states which are not already covered by an exiting witness (Line 4) and by removing an existing witness if all the states covered by it are also covered by a new witness (Line 13 and 14).

6 Conclusion and Future Work

We have presented a temporal logic based approach to automatic test generation from specifications written in EFSMs. Our approach considers a family of coverage criteria based on the information of both control flow and data flow. We associate each coverage criterion with a set of CTL formulas and generate a test suite by finding a set of witnesses for each formula in the set. The resulting test suite provides the capability of determining whether an implementation establishes the required flow of control and data prescribed in its EFSM specification. We show that the optimization problems of finding minimal test suites are NP-hard and describe a method for automatic test generation.

Our ultimate goal is to develop an integrated environment for testing reactive systems. Testing reactive systems is a hard multi-faceted problem. We have just touched the surface of the wealth of issues associated with it. Listed below are some possible extensions that we plan to explore.

Nondeterminism. This paper considered only deterministic EFSMs. In the case of non-deterministic EFSMs, there may be more than one possible execution for a given input event sequence. In this situation, a single witness constructed by model checkers is not enough for the input event sequence, since it identifies only one execution among all possible ones. One possible solution to this problem is to treat the witness as prescribing only the input event sequence. An extra step is then necessary to find all executions corresponding to this input event sequence. If we have a model checker that produces multiple (or all) witnesses to a formula, we can express the input event sequence as a formula and give it to the model checker. The resulting set of witnesses constructed by the model checker will contain all possible executions.

Other Coverage Criteria. A number of other coverage criteria based on control and data flow have been proposed in the software testing literature (for example, see [21]). Some of these coverage criteria require that all paths that cover a certain entity be considered as test sequences. For example, all-du-path coverage criterion requires that all definition-clear paths for every definition-use pair be examined. To generate tests for this criterion in our approach, we need to obtain all witnesses to a CTL formula instead of only one.

Other Formalisms. Our characterization of coverage criteria as collections of CTL formulas is language-independent and is applicable with minor modifications to any kind of specification languages based on EFSMs, e.g., SDL, Estelle, and Statecharts. In fact, semantic differences in such languages affect only the way these models are transformed into input to model checkers. However, when we allow a specification language to express concurrent EFSMs, a number of complications arise. First, the construction of a single Kripke structure from several concurrent EFSMs may result in state explosion. Second, the resulting Kripke structure will likely be nondeterministic due to the interleaving of concurrent events. Often, these interleavings are not controllable by testers.

Other Logics. We showed that CTL is not capable of expressing the coverage criteria based on the affect relation and resolved this problem by extending CTL with least fixpoints of specific predicate transformers so that they can be implemented efficiently in symbolic model checking. However, a more elegant way may be to employ a more expressive temporal logic than CTL. We are currently working with a subset of μ-calculus [16]. The presence of explicit fixpoint operators in μ-calculus makes it possible to characterize all coverage criteria considered in this paper in a more uniform way.

References

1. P. Ammann, P. Black, and W. Majurski, “Using Model Checking to Generate Tests from Specifications,” in *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods*, pp. 46-54, 1998.
2. F. Belina and D. Hogrefe, “The CCITT-Specification and Description Language SDL,” *Computer Networks and ISDN Systems*, Vol. 16, pp. 311-341, 1989.
3. G.v. Bochmann and A. Petrenko, “Protocol Testing: Review of Methods and Relevance for Software Testing,” in *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pp 109-124, 1994.
4. S. Budkowski and P. Dembinski, “An Introduction to Estelle: a Specification Language for Distributed Systems,” *Computer Networks and ISDM Systems*, Vol. 14, No. 1, pp. 3-24, 1991.
5. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang, “Symbolic Model Checking: 10^{20} States and Beyond,” *Information and Computation*, Vol. 98, No. 2, pp. 142-170, June 1992.
6. J. Callahan, F. Schneider, and S. Easterbrook, “Specification-based Testing Using Model Checking,” in *Proceedings of 1996 SPIN Workshop*, also Technical Report NASA-IVV-96-022, West Virginia Univeristy, 1996.
7. E.M. Clarke, E.A. Emerson, and A.P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications,” *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 2, pp. 244-263, Apr. 1986.
8. R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, “Test Development for Communication Protocols: towards Automation,” *Computer Networks*, Vol. 31, Issue 7, pp. 1835-1872, June 1999.

9. A. Engels, L. Feijs, and S. Mauw, "Test Generation for Intelligent Networks Using Model Checking," in *Proceedings of TACAS '97*, Lecture Notes in Computer Science, Vol. 1217, pp. 384-398, Springer-Verlag, 1997.
10. A. Gargantini and C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," in *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 6-10, 1999.
11. D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal, "Coverage-Directed Test Generation Using Symbolic Techniques," in *Proceedings of Formal Methods in Computer Aided Design*, 1996.
12. D. Harel, "Statecharts: a Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, pp. 231-274, 1987.
13. G.J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997.
14. T. Jeron and P. Morel, "Test Generation Derived From Model Checking," in *Computer Aided Verification '99*, Lecture Notes in Computer Science, Vol. 1633, pp. 108-121, Springer-Verlag, 1999.
15. R.M. Karp, "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher Eds., Plenum Press, pp. 85-103, 1972.
16. D. Kozen, "Results on the Propositional Mu-Calculus," *Theoretical Computer Science*, Vol. 27, pp. 333-354, 1983.
17. D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines - A Survey," *Proceedings of the IEEE*, Vol. 84, No. 8, pp. 1090-1123, Aug. 1996.
18. D. Lee and R. Hao, "Test Sequence Selection," in *Formal Techniques for Networked and Distributed Systems*, pp. 269-284, Kluwer Academic Publishers, 2001.
19. K.L. McMillan, *Symbolic Model Checking – an Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.
20. D. Moundanos, J.A. Abraham, and Y.V. Hoskote, "Abstraction Techniques for Validation Coverage Analysis and Test Generation," in *IEEE Transactions on Computers*, Vol. 47, No. 1, pp. 2-14, Jan. 1998.
21. S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, Vol. 11, No. 4, pp. 367-375, Apr. 1985.
22. H. Ural and B. Yang, "A Test Sequence Generation Method for Protocol Testing," *IEEE Transactions on Communications*, Vol. 39, No. 4, pp. 514-523, Apr. 1991.
23. R. de Vries and J. Tretmans, "On-the-Fly Conformance Testing Using SPIN," *International Journal on Software Tools for Technology Transfer*, Vol. 2, Issue 4, pp. 382-393, 2000.
24. M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, pp. 352-357, Apr. 1984.