View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarlyCommons@Penn

University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS)

Department of Computer & Information Science

1-1-2001

# Automatic Test Generation From Statecharts Using Model Checking

Hyoung Seok Hong
*University of Pennsylvania*

Insup Lee
*University of Pennsylvania*, lee@cis.upenn.edu

Oleg Sokolsky
*University of Pennsylvania*, sokolsky@cis.upenn.edu

Follow this and additional works at: https://repository.upenn.edu/cis_reports

## Recommended Citation

Hyoung Seok Hong, Insup Lee, and Oleg Sokolsky, "Automatic Test Generation From Statecharts Using Model Checking", . January 2001.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-01-07.

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/90
For more information, please contact repository@pobox.upenn.edu.

# Automatic Test Generation From Statecharts Using Model Checking


## Abstract

This paper describes a method for automatic generation of tests from specifications written in Statecharts. These tests are to be applied to an implementation to validate the consistency of the implementation with respect to the specification. For test coverage, we adapt the notions of control-flow coverage and data-flow coverage used traditionally in software testing to Statecharts. In particular, we redefine these notions for Statecharts and formulate test generation problem as finding a counterexample during the model checking of a Statecharts specification. The ability to generate a counterexample allows test generation to be automatic.

To illustrate our approach, we show how to translate Statecharts to SMV, after defining the semantics of Statecharts using Kripke structures. We, then, describe how to formulate various test coverage criteria in CTL, and show how the SMV model checker can be used to generate only executable tests.



## Keywords

software testing and analysis, specification-based testing, test generation, control and data flow analysis, Statecharts, model checking, CTL, SMV

## Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-01-07.



This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/90

# Automatic Test Generation from Statecharts Using Model Checking *

Technical Report MS-CIS-01-07
Department of Computer and Information Science
University of Pennsylvania

**Hyoung Seok Hong, Insup Lee, Oleg Sokolsky**
Department of Computer and Information Science
University of Pennsylvania
{hshong,lee,sokolsky}@saul.cis.upenn.edu
**Sung Deok Cha**
Division of Computer Science and AITrc
Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology
cha@salmosa.kaist.ac.kr

**Abstract**

This paper describes a method for automatic generation of tests from specifications written in Statecharts. These tests are to be applied to an implementation to validate the consistency of the implementation with respect to the specification. For test coverage, we adapt the notions of control-flow coverage and data-flow coverage used traditionally in software testing to Statecharts. In particular, we redefine these notions for Statecharts and formulate test generation problem as finding a counterexample during the model checking of a Statecharts specification. The ability to generate a counterexample allows test generation to be automatic.

To illustrate our approach, we show how to translate Statecharts to SMV, after defining the semantics of Statecharts using Kripke structures. We, then, describe how to formulate various test coverage criteria in CTL, and show how the SMV model checker can be used to generate only executable tests.

**Keywords**: software testing and analysis, specification-based testing, test generation, control and data flow analysis, Statecharts, model checking, CTL, SMV

*This research was supported in part by NSF CCR-9988409, NSF CISE-9703220, ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, DARPA ITO MOBIES F33615-00-C-1707, and ONR N00014-97-1-0505 (MURI).

# 1 Introduction

Testing is the most common validation technique and is one of the most time-consuming activities in the development of software or hardware systems. Therefore, techniques for automated construction of high-quality test suites are important to ascertain the correctness of implementations with respect to specifications and decrease the development time and cost.

In this paper, we address the problem of test generation from formal specifications written in Statecharts [13] that have been widely used for specifying reactive systems. Statecharts can be regarded as extended finite state machines (EFSM) that support the hierarchical and concurrent structure on states and the communication mechanism through event broadcasting. Among several variants of Statecharts considered in the literature [3], this paper concentrates on the STATEMATE semantics for Statecharts [14]. Our approach, however, can be immediately applied to other variants of Statecharts semantics, for example, the UML Statecharts [28].

A Statecharts specification typically allows an infinite number of executions and hence exhaustive testing is impossible, which requires all the possible executions be performed. The prevalent testing practice is to construct a test suite, that is, a finite set of test sequences according to some selection criteria, called coverage criteria. This paper considers a family of test coverage criteria based on the flow information of both control and data described in Statecharts. Since manual construction of test suites is time-consuming and error-prone, automatic generation of test suites is desirable to improve the quality and productivity of testing. This paper presents an approach that involves the application of the temporal logic CTL [9] and its symbolic model checker SMV [23] to test generation from Statecharts. Given a Kripke structure model of a system and a temporal logic formula, CTL model checking provides either a claim that the formula is satisfied in the model or else a counterexample falsifying the formula.

An overview of our approach is shown in Figure 1. The problem of test generation is formulated as a CTL model checking problem. A given coverage criterion is expressed as a parameterized collection of formulas in CTL that are instantiated for a given Statecharts specification. Each formula describes a test sequence in abstract terms in such a way that the formula is true if and only if a statechart specification does *not* allow the test sequence. If the specification has finite state space, we can use a model checking tool to check each formula against the specification. If the test sequence described by the formula can be performed by the specification, model checking will fail and the tool will generate a counterexample giving an execution sequence that explains why the formula cannot be satisfied. This counterexample is easily mapped into the test sequence by projecting it onto the observable events of the specification.

The contributions of this paper can be summarized as follows. We give a formal semantics for Statecharts consistent with the STATEMATE informal interpretation. We apply a family of control-flow and data-flow coverage criteria to Statecharts and give a CTL characterization of each coverage criterion. Finally, we demonstrate how to use SMV, an off-the-shelf CTL model checking tool, for the purpose of automatic generation of test suites from a statechart specification with respect to a given coverage criterion.

**Related work.** Widely-used models for reactive systems found in the testing literature include finite state machines (FSM), especially in hardware testing and protocol conformance testing. FSM-based testing methods primarily focus on the control-flow oriented test generation such as transition tour, unique-input-output sequence, distinguishing sequence, and characterizing sequence (see [4, 20] for survey). In protocol conformance testing, these methods have been extensively applied to

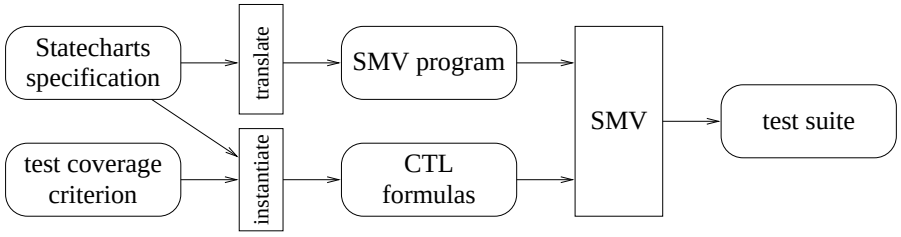


Figure 1: Overview of test generation

formal description techniques [17] such as SDL and Estelle, and a number of automated tools have been developed (for survey, see [10]). Compared to these methods, we are able to provide explicit coverage criteria for data-dependent behaviors.

EFSMs extend FSMs with variables to support the specification of a system with data variables and operations on them. If the state space of an EFSM is finite, one can obtain the equivalent FSM by unfolding the EFSM. Thus testing based on EFSMs with finite state space is reduced in principle to testing based on ordinary FSMs. This approach, however, 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 constructed tests might be too huge to be applied to implementations under test.

A promising alternative is to apply conventional software testing techniques to the generation of tests from EFSMs. In this approach, an EFSM is transformed into a flow graph that models both the control and data flow of the EFSM and then tests are generated by identifying control and data flow information such as definitions and uses of variables in the flow graph [30]. The flow-graph test generation method is also applied to Statecharts [16]. This approach abstracts from the values of variables and hence it can be applicable even if the state space is infinite. The approach, however, requires posterior analysis such as symbolic execution or constraint solving to determine the executability of tests and for the the selection of variable values which make tests executable.

The approach we advocate here is based on translating Statecharts into Kripke structures and also suffers from the state explosion problem. However, the formulation of test generation as model checking in our approach enables the use of symbolic model checking [6], a technique that has been proven successful for controlling the state explosion problem. Second, our approach overcomes the test explosion problem by using the flow information of both control and data described in specifications like the flow-graph approach. Finally, our approach can be seen as complementary to the flow-graph approach. On the one hand, flow graphs can be constructed for systems that are not finite-state. On the other hand, our approach has the advantage that only executable tests are produced.

Connections between test generation and model checking has been considered previously in the literature. A tool that uses test generation algorithms inspired by model checking algorithms is described in [18]. Test generation using counterexamples constructed by a model checker has been applied in several contexts. Mutation analysis is used in the approach of [1]. In [7], test generation is performed from user-specified temporal formulas, while in [11] testing purposes are used to generate tests. No consideration is given to coverage criteria. Several control-flow coverage criteria are considered in [12]. We are not aware of any work that considers the model checking approach to data-flow oriented test generation.

**Organization of the paper.** Section 2 reviews preliminaries of Statecharts and CTL model checking. Section 3 gives a formal definition of the STATEMATE semantics for Statecharts in terms of Kripke structures. Section 4 introduces the notion of test sequences for Statecharts and Section 5 discusses a family of coverage criteria suitable for the generation of test sequences from Statecharts. Section 6 introduces a method to generate tests suites for a given Statecharts specification for several coverage criteria. Finally, Section 7 concludes the paper with a description of future work.

# 2 Preliminaries

This section provides a brief introduction to statecharts and CTL model checking.

## 2.1 Statecharts

A *statechart* is a tuple $Z = (S, \Pi, V, \Theta, T)$ where $S$, $\Pi$, $V$, and $T$ are sets of states, events, variables, and transitions. $\Theta$ is an interpretation of $V$ which assigns to each variable its initial value. The statechart in Figure 2, which specifies a simple coffee vending machine, will be used as the running example in this paper. The variable $m$ in Figure 2 is of integer subrange [0,10] and its initial value is defined by $\Theta(m) = 0$.

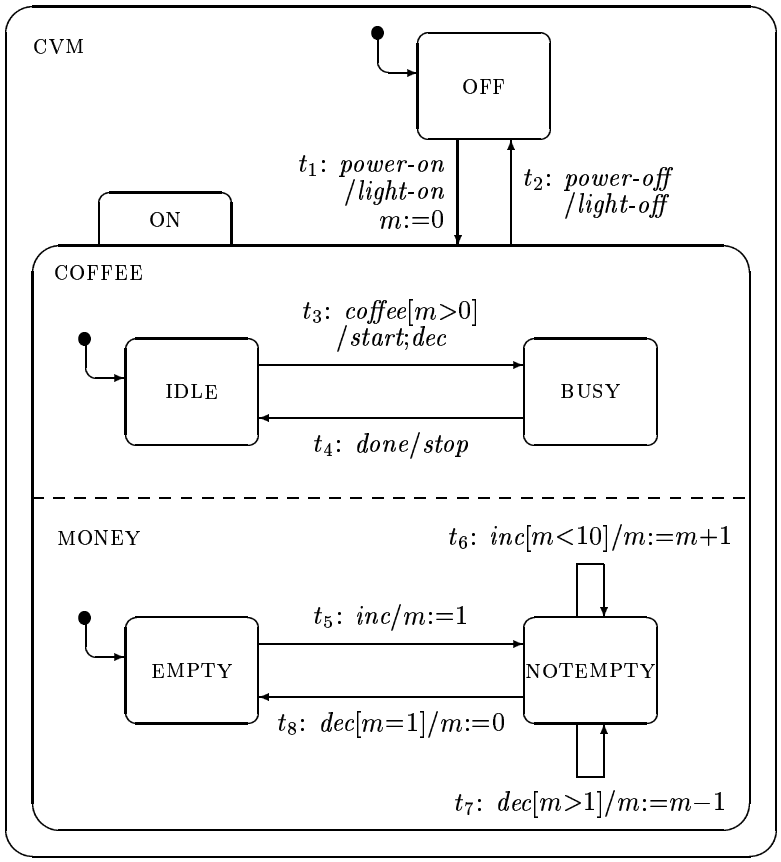


Figure 2: Example of statecharts

One of the main features of statecharts is the hierarchical and concurrent structure on states. A

state is either basic or composite. A composite state is classified as either OR-state or AND-state. An OR-state has substates related by exclusive-or-relation and has exactly one default substate. For example, the OR-state CVM in Figure 2 consists of OFF and ON with OFF as its default substate. Being in CVM implies being in OFF or in ON, but not in both. An AND-state has substates related by and-relation. Being in the AND-state ON implies being in COFFEE and MONEY at the same time. There is a unique state called the *root* at the highest level on the state hierarchy, say CVM.

For a state $s$, define *children*($s$) as the set of substates of $s$ and *children*$^*$ as the reflexive-transitive closure of *children*. For two states $s_1$ and $s_2$, $s_1$ is an *ancestor* of $s_2$ if $s_2 \in children^*(s_1)$. If, in addition, $s_1 \neq s_2$, we say that $s_1$ is a *strict ancestor* of $s_2$.

A *configuration* is a maximal set of states in which a system can be simultaneously. 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 $C$. Each configuration can be uniquely characterized by its basic states. In Figure 2, we have the following configurations with {OFF} as the initial configuration: {OFF}, {IDLE, EMPTY}, {IDLE, NOTEMPTY}, {BUSY, EMPTY}, {BUSY, NOTEMPTY}.

Similar to I/O automata [22] and reactive modules [2], we partition the set $\Pi$ of events into three disjoint subsets $\Pi_I$, $\Pi_L$ and $\Pi_O$ comprising *input*, *local*, and *output* events, respectively. The occurrence of input events is determined by the environment of a system while local and output events are generated by the system itself. Local events are used for internal communications and are assumed to be invisible to the environment. Input and output events are visible to the environment and constitute the *observables* of a statechart. In Figure 2, we have $\Pi_I$ = {*power-on*, *power-off*, *coffee*, *done*, *inc*}, $\Pi_L$ = {*dec*}, and $\Pi_O$ = {*light-on*, *light-off*, *start*, *stop*}.

A transition $t$ is a tuple (*source*($t$), *trigger*($t$) *guard*($t$), *action*($t$), *target*($t$)) where *source*($t$), *target*($t$) $\in S$, *trigger*($t$) is a predicate on $\Pi$, *guard*($t$) is a predicate on $V$, *action*($t$) consists of a set of assignments to $V$, denoted by *assignments*($t$), and a set of events in $\Pi_L \cup \Pi_O$, denoted by *generated*($t$).

For a transition $t$, define *Exits*($t$) (resp. *Enters*($t$)) as the set of states that a system exits (resp. enters) on taking transition $t$. For example, we have that *Exits*($t_1$) = {OFF} and *Enters*($t_1$) = {ON, COFFEE, IDLE, MONEY, EMPTY}. The formal definition for *Exits*($t$) and *Enters*($t$) can be found in [8]. The *scope* of a transition $t$, denoted by *scope*($t$), is defined as the lowest OR-state in the state hierarchy that is a proper ancestor of both *source*($t$) and *target*($t$). For example, *scope*($t_2$) = CVM and *scope*($t_3$) = COFFEE. Two transitions $t$ and $t'$ *conflict* if *scope*($t$) is an ancestor of *scope*($t'$)). For example, $t_2$ and $t_3$ conflict because CVM is an ancestor of COFFEE.

## 2.2 CTL Model Checking

Symbolic model checking [6] is a proven successful technique for the automatic verification of finite state systems. A widely-used temporal logic for symbolic model checking is the branching time temporal logic CTL [9]. Let $AP$ be the underlying set of atomic propositions. The syntax for CTL is defined by the following grammar:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi' \mid EX\phi \mid AX\phi \mid E[\phi U \phi'] \mid A[\phi U \phi']$$

where $p \in AP$ and $\phi, \phi'$ range over CTL formulas. The remaining temporal operators are defined by the equivalence rules: $EF\phi \equiv E[trueU\phi]$; $AF\phi \equiv A[trueU\phi]$; $EG\phi \equiv \neg AF(\neg\phi)$; $AG\phi \equiv \neg EF(\neg\phi)$.

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 state-labeling function; and $R \subseteq Q \times Q$ is the set of transitions. A sequence $q_0$, $q_1$, $q_2$, ... of states is a *path* if $(q_i, q_{i+1}) \in R$ for all $i \geq 0$. A path $\rho$ is a $q$-path if $\rho(0) = q$. The satisfaction relation $\models$ is inductively defined as follows:

- $q \models p$ iff $p \in L(q)$; $q \models \neg\phi$ iff $\neg(q \models \phi)$; $q \models \phi \wedge \phi'$ iff $q \models \phi$ and $q \models \phi'$;
- $q \models EX\phi$ (resp. $AX\phi$) iff for some (resp. all) $q$-path $\rho$, $\rho(1) \models \phi$;
- $q \models E[\phi U \phi']$ (resp. $A[\phi U \phi']$) iff for some (resp. all) $q$-path $\rho$, there exists $i \geq 0$ such that $\rho(i) \models \phi'$ and $\rho(j) \models \phi$ for all $0 \leq j < i$.

SMV [23] is a symbolic model checker for CTL which represents the state space and transition relation of Kripke structures using OBDDs [5]. An SMV program contains a set of variable declarations to determine its state space and descriptions of the initial states and transition relation, as well as a list of CTL formulas to be checked. Given a system model and a CTL formula, SMV automatically provides either a claim that the formula is satisfied in the system model or else a counterexample falsifying the formula.

Let $V$ be a set of variables. We call $v'$ as the primed version of a variable $v \in V$ and use $V'$ to denote the set of primed versions of all variables in $V$. For a predicate $g$ on $V$, we denote by $g'$ the predicate obtained by replacing each variable $v$ in $g$ by $v'$. We define a *SMV program* as a tuple ($V$, *Init*, *Trans*) where $V$ is a finite set of variables; *Init* is a predicate on $V$; and *Trans* is a predicate on $V \cup V'$.

Let $\Sigma(V)$ be the set of all interpretations of $V$. We say that a SMV program ($V$, *Init*, *Trans*) defines the Kripke structure ($Q$, $Q_0$, $L$, $R$) such that

- $Q = \Sigma(V)$;
- $Q_0 = \{\sigma \in \Sigma(V) \mid \sigma \models \mathit{Init}\}$;
- $L(\sigma) = \{v{=}\sigma(v) \mid v \in V\}$, for each $\sigma \in \Sigma(V)$;
- $(\sigma, \sigma') \in R$ if and only if $\langle \sigma, \sigma' \rangle \models \mathit{Trans}$, where $\langle \sigma, \sigma' \rangle$ is the joint interpretation that assigns $\sigma(v)$ to $v \in V$ and $\sigma'(v)$ to $v' \in V'$.

# 3 A Formal Definition of the STATEMATE Semantics

This section formally defines a statechart $Z$ as a Kripke structure $M(Z)$ based on the STATEMATE semantics. Several formalizations of the STATEMATE semantics have been given in terms of labeled transition systems [24, 25]. We chose an alternative approach, since we use CTL model checking to generate tests. We call each element in $Q$ of a Kripke structure ($Q$, $Q_0$, $L$, $R$) a *global state* to distinguish it from a state of Statecharts. Similarly we call each element in $R$ as a *global transition*. The formalization is used as the semantic foundation of the test coverage criteria and test generation method presented in the following sections.

The STATEMATE semantics uses the set of nonnegative integers $\mathsf{N}$ as the time domain and provides two models of time: synchronous and asynchronous. The main notion of the STATEMATE semantics is a step. Intuitively, a step represents the response of a system to the input events

generated by the environment or the local events generated by the system itself. Both time models assume that the execution of a step takes zero time and differ in the way of how time is advanced relative to the execution of steps. In the former model, time is incremented by one time unit after the execution of each step. This time model is mainly used for highly synchronous systems such as synchronous circuits. In the latter, several steps are allowed to take place within a single point in time and time is incremented only when the system becomes stable. Intuitively, stability means that further steps are impossible without new input events. This paper focuses on the asynchronous time model.

### 3.1 State Space

We give a set of rules that identify each component of a Kripke structure from a given statechart. First we represent the state space of a statechart $Z = (S, \Pi, V, \Theta, T)$ using the following set of global states.

$$Q = \mathit{Config} \times \Sigma(V) \times 2^{\Pi} \times 2^{T \cup IT}$$

where *Config* is the set of all configurations of $Z$, $\Sigma(V)$ is the set of all interpretations of $V$, and $IT$ is the set of implicit transitions which shall be discussed in the next section. The set $Q$ of global states captures the following information about a statechart: (i) the states in which a system is; (ii) the values of variables; (iii) the events generated; (iv) the transitions taken.

The set of initial global states is defined as follows: $(C_0, \sigma_0, E_0, \tau_0) \in Q_0$ if $C_0$ is the initial configuration, $\sigma_0 = \Theta$, $E_0 \in \Pi_I$, and $\tau_0 = \emptyset$. The requirement of "$E_0 \in \Pi_I$" states that only input events can be generated and "$\tau_0 = \emptyset$" states that there is no transition taken at the system initialization.

The definition of the label of each global state $(C, \sigma, E, \tau)$ is straightforward:

$$L((C, \sigma, E, \tau)) = in(C) \cup \{v{=}\sigma(v) \mid v \in V\} \cup E \cup \tau$$

where $in(C)$ is a set of propositions defined as $\{in(s) \mid s \in C\}$.

### 3.2 Transition Relation

In the asynchronous time model, input events can be introduced to a system only when the system becomes stable. Once input events are introduced, a sequence of steps is executed until the system becomes stable again. A global state $(C, \sigma, E, \tau)$ is *stable* if there exists no generated input or local event, i.e., $E \cap (\Pi_I \cup \Pi_L) = \emptyset$, and there exists no transition that may occur at that state.

We represent the transition relation of a statechart by the set of global transitions $R = R_1 \cup R_2$, where each global transition in $R_1$ (resp. $R_2$) is called *step* (resp. *tick*). A step transition starts from a non stable global state and manipulates configurations, variables and local and output events, while a tick transition start from a stable global state and manipulates input events.

**Definition 3.1** Let $(C, \sigma, E, \tau)$ and $(C', \sigma', E', \tau')$ be global states. We say that $((C, \sigma, E, \tau), (C', \sigma', E', \tau'))$ is a *step transition* if and only if

1. (not stable) $(C, \sigma, E, \tau)$ is not stable;
2. (configurations) $C' = (C - \bigcup_{t \in \tau'} \mathit{Exits}(t)) \cup \bigcup_{t \in \tau'} \mathit{Enters}(t)$;

3. (variables) $\sigma' = a(\sigma)$, where $a = \bigcup_{t \in \tau'} \mathit{assignments}(t)$;
4. (events) $E' = \bigcup_{t \in \tau'} \mathit{generated}(t)$;
5. (transitions)
   (a) (may occur) each transition $t \in \tau'$ is enabled at $(C, \sigma)$, i.e., $\mathit{source}(t) \in C$ and $\sigma \models \mathit{guard}(t)$, and is triggered by $E$, i.e., $\mathit{trigger}(t)$ evaluates to true for $E$;
   (b) (no conflict) no two transitions in $\tau'$ conflict;
   (c) (maximal) $\tau'$ is maximal, i.e., each transition not in $\tau'$ but triggered by $E$ and enabled at $(C, \sigma)$ conflicts with some transition in $\tau'$.

Intuitively, a step transition $((C, \sigma, E, \tau), (C', \sigma', E', \tau'))$ corresponds to the execution of the transitions in $\tau'$.

**Definition 3.2** Let $(C, \sigma, E, \tau)$ and $(C', \sigma', E', \tau')$ be global states. We say that $((C, \sigma, E, \tau), (C', \sigma', E', \tau'))$ is a *tick transition* if and only if

1. (stable) $(C, \sigma, E, \tau)$ is stable;
2. (configurations) $C' = C$;
3. (variables) $\sigma' = \sigma$;
4. (events) $E' \subseteq \Pi_I$;
5. (transitions) $\tau' = \emptyset$.

A tick transition corresponds to the introduction of input events to a system.

# 4 Test Sequences for Statecharts

This section introduces some necessary terminology.

**Runs.** Let $Z = (S, \Pi, V, \Theta, T)$ be a statechart. We refer to a finite word $\bar{i} = i_1 ... i_n$ over $2^{\Pi_I}$ as *input sequence* and a finite word $\bar{o} = o_1 ... o_n$ over $2^{\Pi_O}$ as *output sequence*. A simple approach to characterizing the behavior of a statechart is to use all the finite paths of its Kripke structure $M(Z)$. This approach, however, is of little use because a path ending at a non stable global state may not provide the information of the output sequence that is supposed to be generated as the response to an input sequence. Therefore we are concerned about only finite paths ending at a stable global state, which we call *runs*. Figure 3 shows a run of the coffee vending machine in which double rectangles represent stable global states. The run corresponds to the execution of the transition sequence $t_1$, $t_5$, $t_3$, $t_8$ in Figure 2.

A subsequence $\langle (C_i, \sigma_i, E_i, \tau_i), ..., (C_j, \sigma_j, E_j, \tau_j) \rangle$ of a run $\langle (C_0, \sigma_0, E_0, \tau_0), ..., (C_n, \sigma_n, E_n, \tau_n) \rangle$ is a *superstep* if $(C_{i-1}, \sigma_{i-1}, E_{i-1}, \tau_{i-1})$ is stable, $(C_k, \sigma_k, E_k, \tau_k)$ is not stable for $i \leq k < j$, and $(C_j, \sigma_j, E_j, \tau_j)$ is stable. We refer to $E_i$ as the input of the superstep and $\Pi_O \cap \bigcup_{i<k\leq j} E_k$ as the output of the superstep. For example, the following shows the three supersteps in Figure 3.

| superstep | input | output |
|---|---|---|
| $gs_1$, $gs_2$ | {*power-on*} | {*light-on*} |
| $gs_3$, $gs_4$ | {*inc*} | $\emptyset$ |
| $gs_5$, $gs_6$, $gs_7$ | {*coffee*} | {*start*} |

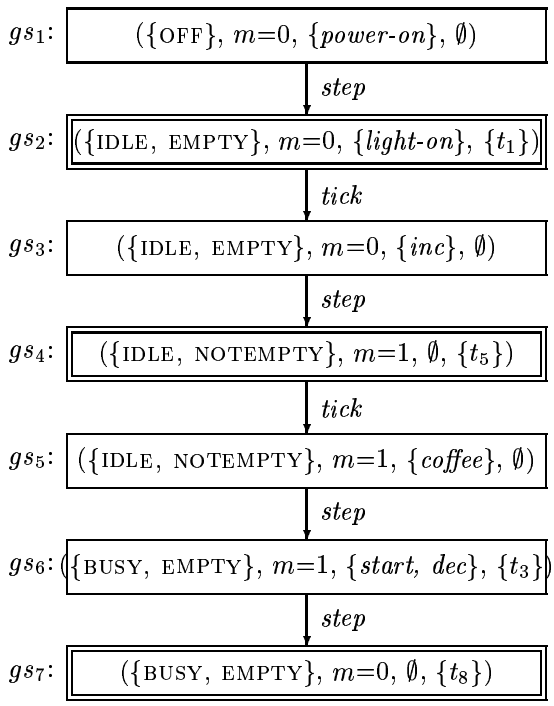


Figure 3: A run for ⟨{*power-on*},{*inc*},{*coffee*} / {*light-on*},∅,{*start*}⟩

**Traces.** We say that a pair of input and output sequences, written as $\bar{i}/\bar{o}$, is a *trace* if there is a run such that $i_j$ and $o_j$ are the input and output of the $j$-th superstep of the run, respectively. For example, from the run in Figure 3 we have the trace ⟨{*power-on*},{*inc*},{*coffee*} / {*light-on*},∅,{*start*}⟩.

**Completion.** Often we need to test that the system does *not* produce any output in response to some input. For example, the coffee vending machine in configuration {IDLE, EMPTY} when $m = 0$ does not respond to input {*coffee*} simply because there are no enabled transitions in the corresponding global state. However, if we want to generate a test for such quiescent behavior using the same technique as for observable behaviors, we need to make the absence of output explicit. For this, we extend the set of transitions of the statechart with *implicit transitions*. Implicit transitions are always self-loops with the empty set of output events.

The following shows the implicit transitions for the coffee vending machine.

$it_1 = it(\text{OFF},\textit{power-off}) = (\text{OFF},\ \textit{power-off},\ \textit{true},\ \emptyset,\ \text{OFF})$
$it_2 = it(\text{OFF},\textit{coffee}) = (\text{OFF},\ \textit{coffee},\ \textit{true},\ \emptyset,\ \text{OFF})$
$it_3 = it(\text{OFF},\textit{done}) = (\text{OFF},\ \textit{done},\ \textit{true},\ \emptyset,\ \text{OFF})$
$it_4 = it(\text{OFF},\textit{inc}) = (\text{OFF},\ \textit{inc},\ \textit{true},\ \emptyset,\ \text{OFF})$
$it_5 = it(\text{OFF},\textit{dec}) = (\text{OFF},\ \textit{dec},\ \textit{true},\ \emptyset,\ \text{OFF})$
$it_6 = it(\text{ON},\textit{power-on}) = (\text{ON},\ \textit{power-on},\ \textit{true},\ \emptyset,\ \text{ON})$
$it_7 = it(\text{IDLE},\textit{coffee}) = (\text{IDLE},\ \textit{coffee},\ \neg(m > 0),\ \emptyset,\ \text{IDLE})$
$it_8 = it(\text{BUSY},\textit{coffee}) = (\text{BUSY},\ \textit{coffee},\ \textit{true},\ \emptyset,\ \text{BUSY})$
$it_9 = it(\text{IDLE},\ \textit{done}) = (\text{IDLE},\ \textit{done},\ \textit{true},\ \emptyset,\ \text{IDLE})$
$it_{10} = it(\text{NOTEMPTY},\ \textit{inc}) = (\text{NOTEMPTY},\textit{inc},\ \neg(m < 10),\ \emptyset,\ \text{NONEMPTY})$

$it_{11} = it(\text{EMPTY}, dec) = (\text{EMPTY}, dec,\ true,\ \emptyset,\ \text{EMPTY})$
$it_{12} = it(\text{NOTEMPTY},\ dec) = (\text{NOTEMPTY}, dec,\ \neg(m > 1 \vee m = 1),\ \emptyset,\ \text{NONEMPTY})$

An input sequence $\bar{i}$ is *explicit* if there exists a trace $\bar{i}/\bar{o}$ such that each step transition of its run corresponds to the execution of an explicit transition. Otherwise, it is *implicit*. For example, the input sequence $\langle\{power\text{-}on\},\ \{inc\},\ \{coffee\}\rangle$ is explicit (see Figure 3), while $\langle\{power\text{-}on\},\ \{coffee\}\rangle$ is implicit because the step transition $(gs_3, gs_4)$ in Figure 4 corresponds to the execution of $it_7$.

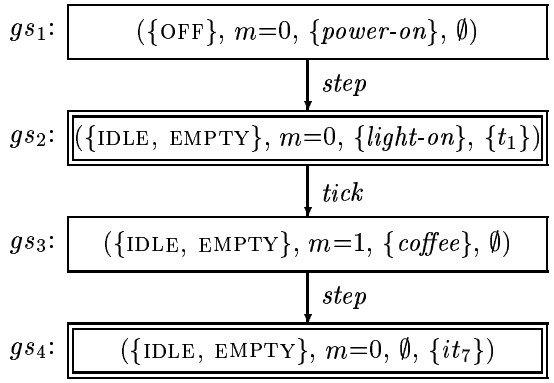


Figure 4: A run for $\langle\{power\text{-}on\},\{coffee\}\ /\ \{light\text{-}on\},\emptyset\rangle$

**Nondeterminism.** A statechart $Z$ is *deterministic with respect to* an input sequence $\bar{i}$ if there exists only one output sequence $\bar{o}$ such that $\bar{i}/\bar{o}$ is a trace of $Z$. Otherwise, it is *nondeterministic with respect to* $\bar{i}$. For example, the coffee vending machine is deterministic with respect to $\langle\{power\text{-}on\},\{inc\},\{coffee\}\rangle$, while it is nondeterministic w.r.t. $\langle\{power\text{-}on\},\{inc\},\{power\text{-}off, coffee\}\rangle$ because there are two possible output sequences: $\langle\{light\text{-}on\},\emptyset,\{light\text{-}off\}\rangle$ and $\langle\{light\text{-}on\},\emptyset,\{start\}\rangle$.

In order to resolve certain classes of nondeterminism, the STATEMATE semantics provides a priority scheme based on the scope of transitions. Let $t$ and $t'$ be transitions conflicting each other. If $scope(t)$ is a strict ancestor of $scope(t')$, then $t$ has priority over $t'$. If $scope(t)$ is equivalent to $scope(t')$, $t$ and $t'$ have the same priority. For example, in Figure 2 $t_2$ has priority over $t_3$, $t_4$, $t_5$, $t_6$, $t_7$, and $t_8$, while $t_6$ and $t_7$ have the same priority. With this priority scheme, the coffee vending machines becomes deterministic with respect to $\langle\{power\text{-}on\},\{inc\},\{power\text{-}off, coffee\}\rangle$ because now we have $\langle\{light\text{-}on\},\emptyset,\{light\text{-}off\}\rangle$ as the only possible output sequence.

**Test sequences.** For an input sequence $\bar{i}$ of a statechart $Z$, we denote by $Z\langle\bar{i}\rangle$ the set of all output sequences $\bar{o}$ such that $\bar{i}/\bar{o}$ is a trace of $Z$. Intuitively $Z\langle\bar{i}\rangle$ is the set of expected or required responses of implementations under test to the input sequence $\bar{i}$.

**Definition 4.1** A *test sequence* of a statechart $Z$, written as $\bar{i}/Z\langle\bar{i}\rangle$, is a pair of an input sequence $\bar{i}$ and its corresponding set $Z\langle\bar{i}\rangle$ of output sequences. A *test suite* is a set of test sequences.

We compare the nature of test sequences for reactive systems with those for transformational systems. Most of analysis and testing models for transformational systems, e.g., flow graphs[15] and program dependency graphs[19], contain a distinguished node called exit node to model the terminating behavior of such systems. Test sequences in such graphs are naturally defined in terms of paths whose first node is the entry node and last node is the exit node of the graphs. On

the other hand, there is no corresponding notion in reactive system models, e.g., FSMs, EFSMs, and statecharts, because the behavior of reactive systems is characterized by their non-terminating computations that maintain an ongoing interaction with the environment.

In Definition 4.1, we do not put any constraints on input sequences and thus allow the execution of test sequences may end at any stable global state. That is, we regard each stable global state as a pseudo-exit node. There are, however, more elaborate approaches to defining exit nodes. A widely used approach in FSM or EFSM-based testing is to require that the execution of test sequences end at an initial state from which another test sequence can be applied. In this approach a special input called reset, which takes a system to its initial state from any state of the system, is often used when the system is not fully connected.

In general, testers may want to designate an arbitrary state as an exit node. An interesting notion is marker state in the supervisory control theory by Ramadge and Wonham[26]. This theory distinguishes the paths ending at a state designated as a marker from others and interprets such paths as completed tasks of the modeled system. For example, a tester may want to designate the configuration {IDLE, EMPTY} as a marker for the coffee vending machine and require that the execution of every test sequence end at the marker. In this case, the input sequences in Figure 3 and Figure 4 can be extended with $\{done\}$ and $\{inc\}$, $\{coffee\}$, $\{done\}$, respectively, in order to guarantee that the machine ends at the marker.

Finally we present a conformance relation between specifications written in statecharts and implementations under test. For an implementation $I$, we denote by $I\langle\bar{i}\rangle$ the set of actual responses of $I$ to an input sequence $\bar{i}$.

**Definition 4.2** Let $Z$ be a statechart and $I$ be an implementation under test. We say that $I$ *weakly conforms* to $Z$ if for all explicit input sequences $\bar{i}$, $I\langle\bar{i}\rangle \subseteq Z\langle\bar{i}\rangle$. We say that $I$ *strongly conforms* to $Z$ if for all input sequences $\bar{i}$, $I\langle\bar{i}\rangle \subseteq Z\langle\bar{i}\rangle$.

# 5 Test Coverage Criteria for Statecharts

Obviously the strongest test coverage criteria is *path coverage* which requires that all the runs of a statechart be traversed, or equivalently all the input sequences of the statechart be applied to implementations under test. Because there is an infinite number of input sequences, it is impossible to achieve exhaustive testing and we need to have systematic coverage criteria that select a reasonable number of test sequences satisfying certain conditions. This paper proposes a family of test coverage criteria based on the flow information of both control and data in statecharts.

## 5.1 Control-Flow Oriented Test Coverage Criteria

**State coverage.** We say that a run *covers* a state $s$ if it contains a global state $(C, \sigma, E, \tau)$ such that $s \in C$. A test sequence $\bar{i}/Z\langle\bar{i}\rangle$ covers a state $s$ if there exists a trace $\bar{i}/\bar{o}$ whose run covers $s$. A test suite $P$ satisfies *state coverage* if each state is covered by a test sequence in $P$.

**Configuration coverage.** A stronger criterion than state coverage may be defined for statecharts which requires the traversal of each configuration. A test suite $P$ satisfies *configuration coverage* if each configuration is covered by a test sequence in $P$.

**Transition coverage.** We say that a run covers a transition $t$ if it contains a step transition corresponding to the execution of $t$. A test sequence $\bar{i}/Z\langle\bar{i}\rangle$ covers a transition $t$ if there exists a trace $\bar{i}/\bar{o}$ whose run covers $t$. A test suite $P$ satisfies *weak transition coverage* if each explicit transition is covered by a test sequence in $P$. A test suite $P$ satisfies *strong transition coverage* if each explicit and implicit transition is covered by a test sequence in $P$.

## 5.2 Data-Flow Oriented Test Coverage Criteria

We adopt the following convention which classifies each variable occurrence of a transition as being a definition, computation use (c-use), and predicate use (p-use). Let $v$ be a variable and $t$ be a transition. $v$ is *defined* at $t$ if *assignments*($t$) includes an assignment that defines $v$; $v$ is *c-used* at $t$ if *assignments*($t$) includes an assignment that references $v$; $v$ is *p-used* at $t$ if *guard*($t$) references $v$. We denote by *def*($v$), *c-use*($v$), and *p-use*($v$) the sets of transitions that define, c-use, and p-use $v$, respectively. Table 1 shows the def, c-use, and p-use sets for the variable $m$ of the coffee vending machine.

Table 1: The def, c-use, and p-use sets for the coffee vending machine

| *def*($m$) | *c-use*($m$) | *p-use*($m$) |
|---|---|---|
| $t_1$, $t_5$, $t_6$, $t_7$, $t_8$ | $t_6$, $t_7$ | $t_3$, $t_6$, $t_7$, $t_8$ |

Let $t$ and $t'$ be transitions and $gs_0, \ldots, gs_n$ be a run. Suppose that the step transitions ($gs_i$, $gs_{i+1}$) and ($gs_j$, $gs_{j+1}$) such that $0 \leq i < j < n$ correspond to the execution of $t$ and $t'$, respectively. The run is a *definition-clear run* with respect to $v$ from $t$ to $t'$ if each step transition ($gs_k$, $gs_{k+1}$) does not correspond to the execution of any transition at which $v$ is defined, for $i < k < j$.

We define associations between definitions and uses of a variable as follows: a tuple $(v, t, t')$ is a *def-c-use association* (resp. *def-p-use association*) if $t \in$ *def*($v$), $t' \in$ *c-use*($v$) (resp. $t' \in$ *p-use*($v$)), and there exists a definition-clear run with respect to $v$ from $t$ to $t'$. A *def-use association* is either a def-c-use association or def-p-use association. Table 2 shows the def-use associations for the coffee vending machine. Consider the def-p-use association $(m, t_5, t_3)$. The definition of $m$ at $t_5$ can reach the use of $m$ at $t_3$ through the definition-clear run shown in Figure 3.

Table 2: The def-use associations for the coffee vending machine

| def-c-use associations | def-p-use associations |
|---|---|
| ($m$, $t_5$, $t_6$) | ($m$, $t_5$, $t_3$), ($m$, $t_5$, $t_6$), ($m$, $t_5$, $t_8$) |
| ($m$, $t_6$, $t_6$), ($m$, $t_6$, $t_7$) | ($m$, $t_6$, $t_3$), ($m$, $t_6$, $t_6$), ($m$, $t_6$, $t_7$) |
| ($m$, $t_7$, $t_6$), ($m$, $t_7$, $t_7$) | ($m$, $t_7$, $t_3$), ($m$, $t_7$, $t_6$), ($m$, $t_7$, $t_7$), ($m$, $t_7$, $t_8$) |

Now we discuss the data flow caused by implicit transitions. Let $v$ be a variable and $it$ be an implicit transition. We say that $v$ is *implicitly p-used* at $it$ if *guard*($it$) references $v$ and denote by *implicit-p-def*($v$) the set of implicit transitions at which $v$ is p-used. Note that definitions and c-uses cannot occur at an implicit transition because the action of an implicit transition is defined to be empty, A tuple $(v, t, it)$ is an *implicit def-p-use association* if $t \in$ *def*($v$), $it' \in$ *p-use*($v$), and there exists a definition-clear run with respect to $v$ from $t$ to $it$. Table 3 shows the data flow caused

by implicit transitions the coffee vending machine. The run shown in Figure 3 is a definition-clear run for $(m, t_1, it_7)$.

Table 3: The implicit p-uses and def-p-use associations for the coffee vending machine

| *implicit-p-use*$(m)$ | *implicit def-p-use associations* |
|---|---|
| $it_7$, $it_{10}$, $it_{12}$ | $(m, t_1, it_7)$, $(m, t_6, it_{10})$, $(m, t_8, it_7)$ |

In general there are three types of associations between definitions and uses in statecharts. The first type includes associations which occur within an OR-state, e.g., $(m, t_5, t_6)$. Associations occurring in ordinary EFSMs belong to this type. The second type is caused by the hierarchical structure on states, e.g., $(m, t_1, it_7)$. The third type is caused by the concurrent structure on states, e.g., $(m, t_5, t_3)$.

We say that a test sequence $\bar{i}/Z\langle\bar{i}\rangle$ covers a def-use association $(v, t, t')$ if there exists a trace $\bar{i}/\bar{o}$ whose run is a definition-clear run with respect to $v$ from $t$ and $t'$.

**All-def coverage.** A test suite $P$ satisfies *weak all-def coverage* if for each variable $v$ and each transition $t$ such that $t \in$ *def*$(v)$, some def-use association $(v, t, t')$ is covered by a test sequence in $P$. A test suite $P$ satisfies *strong all-def coverage* if for each variable $v$ and each transition $t$ such that $t \in$ *def*$(v)$, some explicit def-use association $(v, t, t')$ or implicit def-use association $(v, t, it)$ is covered by a test sequence in $P$.

**All-use coverage.** A test suite $P$ satisfies *weak all-use coverage* if for each variable $v$ and each transition $t$ such that $t \in$ *def*$(v)$, each def-use association $(v, t, t')$ is covered by a test sequence in $P$. A test suite $P$ satisfies *strong all-use coverage* if for each variable $v$ and each transition $t$ such that $t \in$ *def*$(v)$, each explicit def-use association $(v, t, t')$ and implicit def-use association $(v, t, it)$ is covered by a test sequence in $P$.

# 6 Test Generation Methods for Statecharts

This section shows that test generation from Statecharts can be automatically performed by using the SMV's ability to generate counterexamples. Briefly, the generation of a test suite from a statechart and a test coverage criterion consists of the following steps.

- An SMV program is constructed from the statechart.
- A set of CTL formulas is constructed from the criterion.
- A test suite is constructed by model-checking the CTL formulas against the SMV program and projecting the obtained counterexamples onto the observable events of the trace.

The following subsections describe the details of the three steps.

## 6.1 Statecharts as SMV Programs

This section gives a method that translates a statechart $Z$ into a SMV program. The correctness of the translation method is shown by proving that the SMV program is isomorphic to the Kripke structure $M(Z)$.

### 6.1.1 State Space

Recall that a global state $(C, \sigma, E, \tau)$ consists of a configuration, an interpretation of $V$, and a set of events. A simple approach to encoding configurations in SMV is to use a Boolean variable for each state of a statechart. This requires a number of Boolean variables and can be improved by associating with an OR-states $s$ a Boolean variable that ranges over $children(s)$, the set of substates of $s$. For example, we use *CVM*: {*off, on*}, *COFFEE*: {*idle, busy*}, and *MONEY*: {*empty, notempty*} for the coffee vending machine. Note that each configuration of a statechart is uniquely characterized in terms of interpretations of such variables. For each state $s$, define an auxiliary predicate $in(s)$ to indicate whether a system is in $s$ or not.

| | |
|---|---|
| $in(s) ::= 1$ | if $s$ is the root state; |
| $in(s) ::= in(ps)$ | if $s$ is a substate of an AND-state $ps$ |
| $in(s) ::= in(ps) \wedge ps{=}s$ | if $s$ is a substate of an OR-state $ps$ |

We associate a Boolean variable with each event to represent the occurrence of the event. We also associate a Boolean variable with each explicit and implicit transition to represent the execution of the transition. In summary we declare and initialize the following SMV variables to encode the global states of a statechart.

| | |
|---|---|
| for each OR-state $s$, | `VAR` $s$: $children(s)$; `INIT` $s{=}default(s)$ |
| for each variable $v$, | `VAR` $v$: $range(v)$; `INIT` $v{=}\Theta(v)$ |
| for each input event $e$, | `VAR` $e$: boolean; `INIT` 1 |
| for each local or output event $e$, | `VAR` $e$: boolean; `INIT` $e{=}0$ |
| for each explicit transition $t$, | `VAR` $t$: boolean; `INIT` $t{=}0$ |
| for each implicit transition $it$, | `VAR` $it$: boolean; `INIT` $it{=}0$ |

### 6.1.2 Transition Relation

Recall that the transition relation of a statechart consists of two types of global transitions: step and tick. First we represent step transitions using the following predicate.

$$Step ::= StepExplicit \wedge StepImplicit \wedge StepConfig \wedge StepVariable \wedge StepEvent$$

The sub-predicate *StepExplicit* is used to select a set $\tau' \subseteq T$ of explicit transitions satisfying the requirements from 5.(a) to 5.(c) in Definition 3.1.

| | |
|---|---|
| $StepExplicit$ | $::= \bigwedge_{t \in T} StepExplicit(t)$ |
| $StepExplicit(t)$ | $::= [t' = mayoccur(t) \wedge \neg conflicting(t)]$ |
| $mayoccur(t)$ | $::= in(source(t)) \wedge guard(t) \wedge trigger(t)$ |
| $conflicting(t)$ | $::= \bigvee_{ct \in CT(t)} ct'$ where $CT(t)$ is the set of transitions that conflict $t$ and have priority over $t$. |

The correctness of *StepExplicit* can be understood as follows. $mayoccur(t)$ and $\neg conflicting(t)$ fulfill the requirement of 5.(a) and 5.(b), respectively. *StepExplicit* combines its sub-predicates by conjunction and hence fulfills the requirement of 5.(c), i.e., maximality.

We represent the execution of implicit transitions as follows:

$$\begin{aligned} &\mathit{StepImplicit} &&::= \textstyle\bigwedge_{it \in IT} \mathit{StepImplicit}(it) \\ &\mathit{StepImplicit}(it) &&::= [it' = mayoccur(it) \wedge in(target(it))'] \end{aligned}$$

The other three sub-predicates of *Step* are used to capture the changes of configurations, variables, and events, respectively. For an OR-state $s$, we denote by $T(s)$ the set of transitions $t$ such that $children(t) \cap Enters(t) \neq \emptyset$. The substates of $s$ can be changed by a transition $t \in T(s)$ and the structural characteristics of statecharts guarantees that $children(t) \cap Enters(t)$ is a singleton. For a variable $v \in V$, we denote by $T(v)$ the set of transitions $t$ such that $assignments(t)$ contains an assignment $v{:=}exp$. For an event $e \in \Pi$, we use $T(e)$ to denote the set of transitions $t$ such that $generated(t)$ contains $e$.

$$\begin{aligned} &\mathit{StepConfig} &&::= \textstyle\bigwedge_{s \in OS} \mathit{StepConfig}(s) \\ &\mathit{StepConfig}(s) &&::= [\textstyle\bigwedge_{t \in T(c)} (t'{=}1 \rightarrow s'{=}ns)] \wedge [(\textstyle\bigwedge_{t \in T(c)} t'{=}0) \rightarrow s'{=}s] \\ &&&\quad \text{where } children(s) \cup Enters(t) = \{ns\} \end{aligned}$$

$$\begin{aligned} &\mathit{StepVariable} &&::= \textstyle\bigwedge_{v \in V} \mathit{StepVariable}(v) \\ &\mathit{StepVariable}(v) &&::= [\textstyle\bigwedge_{t \in T(v)} (t'{=}1 \rightarrow v'{=}exp))] \wedge [(\textstyle\bigwedge_{t \in T(v)} t'{=}0) \rightarrow v'{=}v] \end{aligned}$$

$$\begin{aligned} &\mathit{StepEvent} &&::= \textstyle\bigwedge_{e \in \Pi} \mathit{StepEvent}(e) \\ &\mathit{StepEvent}(e) &&::= [e'{=}0] \text{ if } e \text{ is an input event} \\ &&&\quad [\textstyle\bigwedge_{t \in T(e)} (t'{=}1 \rightarrow e'{=}1))] \wedge [(\textstyle\bigwedge_{t \in T(e)} t'{=}0) \rightarrow e'{=}0] \text{ otherwise} \end{aligned}$$

Second we represent tick transitions using the following predicate.

$$\begin{aligned} &\mathit{Tick} ::= \textstyle\bigwedge_{t \in T}(t'{=}0) \wedge \bigwedge_{it \in IT}(it'{=}0) \wedge \bigwedge_{s \in OS}(s'{=}s) \wedge \bigwedge_{v \in V}(v'{=}v) \wedge \mathit{TickEvent} \\ &\mathit{TickEvent} ::= \textstyle\bigwedge_{e \in \Pi} \mathit{TickEvent}(e) \\ &\mathit{TickEvent}(e) ::= 1 \text{ if } e \text{ is an input event, } (e'{=}0) \text{ otherwise} \end{aligned}$$

Finally we define the transition relation *Trans* for a statechart using the predicates *Step* and *Tick* as follows:

$$\begin{aligned} &\mathit{Trans} ::= [\neg \mathit{Stable} \rightarrow \mathit{Step}] \wedge [\mathit{Stable} \rightarrow \mathit{Tick}] \\ &\mathit{Stable} ::= \textstyle\bigwedge_{e \in \Pi_I \cup \Pi_L} (e{=}0) \wedge \bigwedge_{t \in T} \neg mayoccur(t) \end{aligned}$$

For a statechart $Z = (S, \Pi, V, \Theta, V)$, we use $AP(Z)$ to denote the following set of atomic propositions.

$$\{in(s) \mid s \in S\} \cup \{v{=}\sigma(v) \mid v \in V,\, \sigma \in \Sigma(V)\} \cup E \cup T \cup IT$$

To illustrate the translation, we show the complate SMV program for the coffee vending machine specification in Appendix A.

Correctness of the translation is proved by the following argument. Let $Z = (S, \Pi, V, \Theta, V)$ be a statechart. We use $AP(Z)$ to denote the following set of atomic propositions.

$$\{in(s) \mid s \in S\} \cup \{v{=}\sigma(v) \mid v \in V,\, \sigma \in \Sigma(V)\} \cup E \cup T \cup IT$$

Let $M_1$ be the Kripke structure $M(Z)$ and $M_2$ be the Kripke structure defined by the SMV program ($V$, *Init*, *Trans*) for a statechart $Z$. We deifne $f$ to be the bijection from the set of global states of $M_1$ to the set of global states of $M_2$ such that for each atomic proposition $p$ in $AP(Z)$,

$$f(q_1) = q_2 \text{ if and only if } q_1 \models p \Leftrightarrow q_2 \models p.$$

**Lemma 6.1** *Let $q_1$ and $q_1'$ be states of $M_1$. Then ($q_1$, $q_1'$) is a step transition of $M_1$ if and only if $\langle f(q_1), f(q_1')\rangle \models \neg Stable \rightarrow Step$.*

**Lemma 6.2** *Let $q_1$ and $q_1'$ be states of $M_1$. Then ($q_1$, $q_1'$) is a tick transition of $M_1$ if and only if $\langle f(q_1), f(q_1')\rangle \models Stable \rightarrow Tick$.*

**Lemma 6.3** *Let $q_1$ and $q_1'$ be states of $M_1$. Then ($q_1$, $q_1'$) is a global transition of $M_1$ if and only if $\langle f(q_1), f(q_1')\rangle \models Trans$.*

**Proof**: a direct consequence of Lemma 1 and Lemma 2.

**Theorem 6.1** *$M_1$ is ismorphic to $M_2$.*

**Proof**:

- $q$ is an initial global state of $M$ iff $f(q)$ is an initial global state of $M(Z)$ (by definition).
- ($q$, $q'$) is a global transition of $M$ iff ($f(q)$, $f(q')$) is a global transition of $M(Z)$ (by Lemma 3).

## 6.2 Test Coverage Criteria as CTL Formulas

Each coverage criterion is represented as a set of CTL templates. For a given statechart, the set of templates is instantiated into a set of CTL formulas with the predicates defined for the statechart. The resulting set of CTL formulas captures exactly the coverage criterion for the given statechart.

### 6.2.1 CTL Formulas for Control-Flow Coverage Criteria

We begin with the state coverage criteria which requires that for each state $s$, there exists at least one run covering $s$. A Kripke structure $M$ has a run covering $s$ if and only if (i) there exists global state $gs_i$ which is reachable from an initial global state $gs_0$ and at which $in(s)$ is satisfied and (ii) there exists a global state $gs_j$ which is reachable from $gs_i$ and at which *stable* is satisfied (see Figure 5). We express the requirement using the CTL formula $EF\,(in(s) \wedge EF\,stable)$.

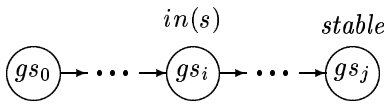


Figure 5: A run covering a state $s$

Now we take the negation of the above formula and run SMV against the negated formula $\neg EF\,(in(s) \wedge EF\,stable)$ because we are interested in generating runs covering $s$ instead of checking the satisfiability of the original formula. If there exists a run covering $s$, SMV generates a counterexample which corresponds to a run covering $s$. Otherwise, SMV provides the result of *true*. There are two cases in which SMV provides *true* against the negated formula. First, the global state $gs_i$ is not reachable from any initial global state, i.e., $EF\,in(s)$ is not satisfied. Second, a statechart does not terminate and hence cannot reach a global state at which *stable* is satisfied, i.e., $EF\,stable$ is not satisfied. For example, consider $\neg EF\,(in(\text{B}) \wedge EF\,stable)$ whose purpose is to cover the state B in Figure 6. Although B is reachable from an initial global state, there is no run covering B because there is an infinite sequence consisting of only step transitions

$$(\{\text{A}\}, m = 1, \{\alpha\}, \emptyset), (\{\text{B}\}, m = 1, \{\beta\}, \{t_1\}), (\{\text{A}\}, m = 1, \{\alpha\}, \{t_2\}), \ldots$$

and hence the statechart cannot reach a stable global state.

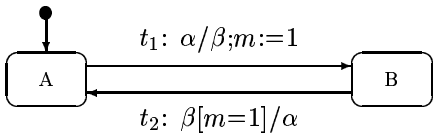


Figure 6: A non-terminating statechart

As mentioned before, it may be required that each run ends at an initial state or an arbitrary state marked by testers. Let *exit* be a predicate defined as *stable* if there is no marker, and *stable* $\wedge$ $in(C)$ if $C$ is a configuration designated as a marker. We can simply express the requirement of markers using $EF\,(in(s) \wedge EF\,exit)$.

Let $BS \subseteq S$ be the set of basic states. We note that a test suite covering all basic states covers all states because of the state hierarchy of Statecharts. To generate test sequences satisfying state coverage, we use the following set of CTL formulas.

**Definition 6.1** The CTL formulas for state coverage

$$\{\neg EF\,(in(s) \wedge EF\,exit) \mid s \in BS\}$$

We can define the CTL formulas for other control-flow oriented criteria in a similar way.

**Definition 6.2** The CTL formulas for configuration coverage

$$\{\neg EF\,(in(c) \wedge EF\,exit) \mid c \in Config\}$$

**Definition 6.3** The CTL formulas for weak transition coverage

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

**Definition 6.4** The CTL formulas for strong transition coverage

$$\{\neg EF\,(t \wedge EF\,exit) \mid t \in T\} \cup \{\neg EF\,(it \wedge EF\,exit) \mid it \in IT\}$$

### 6.2.2 CTL Formulas for Data-Flow Coverage Criteria

We use the following predicates to encode the information of definitions and uses of a variable $v$ in SMV.

$$d(v) ::= \bigvee_{t \in def(v)} t$$
$$u(v) ::= \bigvee_{t \in p\text{-}use(v) \cup c\text{-}use(v)} t$$
$$im\text{-}u(v) ::= \bigvee_{it \in implicit\text{-}p\text{-}use(v)} it$$

For example, for the coffee vending machine we have $d(m) ::= t_1 \vee t_5 \vee t_6 \vee t_7 \vee t_8$, $u(m) ::= t_3 \vee t_6 \vee t_7 \vee t_8$, and $im\text{-}u(m) ::= it_7 \vee it_{10} \vee it_{12}$.

The requirement for a def-use association $(v, t, t')$ can be stated as follows: (i) there exists a global state $gs_1$ which is reachable from an initial global state $gs_0$ and at which $t$ is satisfied; (ii) there exists a path $gs_2$ ... $gs_3$ which starts from a successor of $gs_1$, and contains no definition of $v$ until $gs_4$ at which $t'$ is satisfied; (iii) there exists a global state reachable $gs_5$ which is from the global state $gs_4$ and at which *exit* is satisfied (see Figure 7). We express this requirement using $EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(t' \wedge EF\,exit)])$.

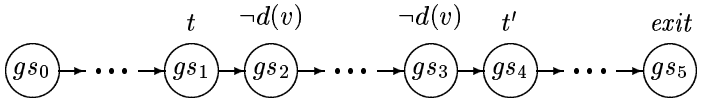


Figure 7: A run covering a def-use association $(v, t, t')$

We determine whether each tuple $(v, t, t')$ such that $t \in def(v)$ and $t' \in use(v)$ is a def-use association or not by associating the negation of the above formula $\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(t' \wedge EF\,exit)])$ with the tuple. If SMV generates the result of *true* against the negated formula, the tuple is not a def-use association. Otherwise, the counterexample generated by SMV corresponds to a run covering the def-use association $(v, t, t')$.

**Definition 6.5** The CTL formulas for weak all-def coverage

$$\{\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(u(v) \wedge EF\,exit)]) \mid v \in V, t \in def(v)\}$$

**Definition 6.6** The CTL formulas for strong all-def coverage

$$\{\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,((u(v) \vee im\text{-}u(v)) \wedge EF\,exit)]) \mid v \in V, t \in def(v)\}$$

**Definition 6.7** The CTL formulas for weak all-use coverage

$$\{\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(t' \wedge EF\,exit)]) \mid v \in V, t \in def(v), t' \in use(v)\}$$

**Definition 6.8** The CTL formulas for strong all-use coverage

$$\{\neg EF\,(t \wedge EX\,E\,[\neg d(v)\,U\,(t' \wedge EF\,exit)]) \mid v \in V, t \in def(v), t' \in use(v) \cup implicit\text{-}p\text{-}use(v)\}$$

### 6.3 Test Generation as CTL Model Checking

We consider deterministic statecharts for test generation. We can determine the determinacy of a statechart by model-checking $\bigwedge_{t \in T} AG\,(mayoccur(t) \rightarrow AX\,t)$ against the Kripke structure $M(Z)$. The following shows our test generation method for deterministic statecharts.

*TestSuite* = $\emptyset$;
for each CTL formula $f$ for a test coverage criterion
    model-check $f$ against $M(Z)$;
    if SMV generates a counterexample
        map the counterexample into a trace $\bar{i}/\bar{o}$;
        *TestSuite* := *TestSuite* $\cup$ $\{\bar{i}/\{\bar{o}\}\}$;
end;

It is straightforward to map a counterexample into a trace by projecting it on the input and output events of the statechart. For example, Appendix B shows the result of model-checking $\neg EF\,(t_3 \wedge EF\,stable)$ against the coffee vending machine. The counterexample is the symbolic representation of the run in Figure 3 and covers the transition $t_3$. When generating counterexamples, SMV describes an initial state by providing the values of all variables and predicates. Other states are described in terms of only the values that are changed from one state to the next. The counterexample is mapped into the trace {*power-on*},{*inc*},{*coffee*} / {*light-on*},$\emptyset$,{*start*}. Appendix C shows a set of test suites for the coffee vending machine.

## 7 Conclusions and Future Work

We described a method for automatic test generation for Statecharts specifications. Tests are selected according to a set of commonly used coverage criteria based on control and data flow in the specification. Each coverage criterion is described as a set of formulas in the temporal logic CTL. Each formula defines one test in such a way that the formula is satisfied by the specification if and only if the test is infeasible in the specification. Otherwise, the model checker produces a counterexample for the formula. The counterexample, projected onto the observable events of the specifications, yields a test.

The advantage of the approach is that only feasible tests are generated. We have applied the test generation method to several examples. Our future work includes larger case studies to assess effectiveness and scalability of the approach.

**Extensions.** In this paper, we did not consider several important features of Statecharts such as actions associated with states, transitions with multiple source and target states, compound transitions, histories, and real-time constructs such as timeout events and scheduled actions. However, it is fairly simple to extend our test generation method once we have a formal definition for these features in terms of Kripke structures.

Many variants of semantics have been proposed for Statecharts [3]. The RSML semantics proposed by Leveson et al. [21] is close to the asynchronous time model of the STATEMATE semantics. The method presented in this paper would apply to the RSML semantics with only slight modifications. For other Statecharts semantics, significant changes would be necessary to reflect the peculiarities of step construction methods in these semantics. However, the differences

between such semantics affect only the translation method from Statecharts into the input to a model checker. Our representation of test coverage criteria as collections of CTL formulas is language-independent and is applicable with only minor modifications to any kind of specification languages based on EFSMs including UML Statecharts and formal description techniques such as SDL and Estelle.

**Other coverage criteria.** A number of other coverage criteria based on control and data flow analysis have been proposed in the software testing literature (see, for example, [27]). Some of these coverage criteria cannot be handled using SMV, because their CTL properties contain universal path quantifiers. For example, all-du-paths coverage criterion requires that all paths for a definition-use pair, all definition-clear runs must be traversed. To generate tests for this criterion correctly, SMV would have to produce all counterexamples to each CTL formula. Such coverage criteria can be handled by extending SMV to produce multiple counterexamples for a formula, or by using a different model checker that has this facility.

**Nondeterminism.** In this work, we generated tests only from deterministic statecharts. If a statechart is non-deterministic, there may be more than one possible output sequence for a given input sequence. In this situation, a single counterexample produced by the model checker is not enough, since it will identify a single output sequence. A possible solution to this problem is to treat the counterexample as prescribing only the *input* sequence. An extra step is then needed to find all output sequences corresponding to this input sequence. If we have a model checker that produces multiple counterexamples to a formula, as discussed in the previous paragraph, we can express the input sequence as a CTL formula and give its negation to the model checker. The set of counterexamples produced by the model checker will contain all feasible output sequences.

**Symbolic Test Generation.** As a technical convenience, we implicitly assumed that each variable appearing in a statechart is local, i.e., it is defined and used only by the modeled system. Similar to the set of events, we can partition the set of variables into the set of input, local, and output variables. Or equivalently, we can associate parameters with input and output events. The values of local and output variables are determined by the system itself, while input variables can be assigned any value nondeterministically by the environment of a system. Therefore, we are interested in a set of possible values for an input variable instead of a specific value when the value of the input variable is changed by the environment.

This problem cannot be solved by the current implementaion of SMV which generates a counterexample by randomly selecting one value among all the possible values. Recently, Rusu et al. [29] discussed the problem of symbolic test generation from EFSMs in the realm of theorem proving. We believe that in the finite-state setting this problem can be formulated as a model checking by problem extending the SMV's ability to generate counterexamples so that it can provide a set of predicates over variables and parameters which describe the possible values instead of a specific value.

**Optimizations.** Often, the test suite produced by the described approach will contain redundant tests. For example, in the transition coverage test suite for the coffee vending machine, the test to cover transition $t_7$: $\langle\{power\text{-}on\},\{inc\},\{inc\},\{dec\}$ / $\{light\text{-}on\},\emptyset,\emptyset,\emptyset\rangle$ will also cover transitions $t_1, t_5$, and $t_6$. Therefore, we need to find heuristics to minimize the number of generated tests without sacrificing the coverage they provide.

## 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] R. Alur and T.A. Henzinger, "Reactive Modules," in *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pp. 207-218, 1996.

[3] M. von der Beeck, "A Comparison of Statecharts Variants," in *Formal Techniques in Real-Time Fault-Tolerant Systems*, Lecture Notes in Computer Science, Vol. 863, pp. 128-148, Springer-Verlag, 1994.

[4] 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.

[5] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 6, pp. 677-691, Aug. 1986.

[6] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang, "Symbolic Model Checking: $10^{20}$ States and Beyond," in *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, 1990.

[7] 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.

[8] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese, "Model Checking Large Software Specifications," *IEEE Transactions on Software Engineering*, Vol. 24, No. 7, pp. 498-520, July 1998.

[9] 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.

[10] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, "Test Development for Communication Protocols: towards Automation," *Computer Networks*, Vol. 31, Vol. 17, pp. 1835-1872, 1999.

[11] 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.

[12] 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.

[13] D. Harel, "Statecharts: a Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, pp. 231-274, 1987.

[14] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Transactions on Software Engineering and Methodologies*, Vol. 5, No. 4, pp. 293-333, Oct. 1996.

[15] M.S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.

[16] H.S. Hong, Y.G. Kim, S.D. Cha, D.H. Bae, and H. Ural, "A Test Sequence Selection Method for Statecharts," *Journal of Software Testing, Verification, and Reliability*, Vol. 10, No. 4, pp. 203-227, Dec. 2000.

[17] ITU-T, Recommendation X. 904 - Information Technology - Open Distributed Processing - Reference Model: Architectural Semantics, Dec. 1997.

[18] T. Jeron and P. Morel, "Test Generation Derived From Model Checking," in *Proceedings of CAV '99*, LNCS 1633, pp. 108-121, 1999.

[19] B. Korel, "The Program Dependence Graph in Static Program Testing," *Information Processing Letters*, Vol. 24, pp. 103-108, Jan. 1987.

[20] 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.

[21] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Transactions on Software Engineering*, Vol. 30, No. 9, pp. 684-707, Sept. 1994.

[22] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," *CWI Quarterly*, Vol. 2, No. 3, pp. 219-246, Sept. 1989.

[23] K.L. McMillan, *Symbolic Model Checking — an Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.

[24] E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel, "On formal semantics of Statecharts as supported by STATEMATE," in *Proceedings of 2nd BCS-FACS Northern Formal Methods Workshop*, Springer-Verlag, July 1997.

[25] C. Petersohn and L. Urbina, "A Timed Semantics for the STATEMATE Implementation os Statcharts," in *Proceedings of Formal Methods Europe '97*, 1997.

[26] P.J. Ramadge and W.M. Wonham, "Supervisory Control of a Class of Discrete Event Processes," *SIAM Journal of Control and Optimization*, Vol. 25, No. 1, pp. 206-230, Jan. 1987.

[27] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, pp. 367-375, Apr. 1985.

[28] J. Rumbaugh, G. Booch, and I. Jacobson, *The UML Reference Guide*, Addison Wesley Longman, 1999.

[29] V. Rusu, L. du Bousquet, and T. Jéron, "An Approach to Symbolic Test Generation," in *Proceedings of the International Conference on Integrating Formal Methods*, 2000.

[30] H. Ural, K. Sale, and A. Williams, "Test Generation Based on Control and Data Dependencies within System Specifications in SDL," *Computer Communications*, Vol. 23, p. 609-627, 2000.

# A The SMV Program for the Coffee Vending Machine

```
MODULE main

VAR
-- configurations
CVM: {off, on};
COFFEE: {idle, busy};
MONEY: {empty, notempty};
-- events
power-on: boolean;
power-off: boolean;
coffee: boolean;
done: boolean;
inc: boolean;
dec: boolean;
light-on: boolean;
light-off: boolean;
start: boolean;
stop: boolean;
-- variables
m: 0..10;
-- explicit transitions
t1: boolean;
t2: boolean;
t3: boolean;
t4: boolean;
t5: boolean;
t6: boolean;
t7: boolean;
t8: boolean;
-- implicit transitions
it1: boolean;
it2: boolean;
it3: boolean;
it4: boolean;
it5: boolean;
it6: boolean;
it7: boolean;
it8: boolean;
it9: boolean;
it10: boolean;
it11: boolean;
it12: boolean;

DEFINE
-- in(s)
in-cvm := 1;
in-off:= in-cvm & CVM=off;
in-on := in-cvm & CVM=on;
in-coffee := in-on;
in-idle := in-coffee & COFFEE=idle;
in-busy := in-coffee & COFFEE=busy;
in-money := in-on;
in-empty := in-money & MONEY=empty;
in-notempty := in-money & MONEY=notempty;
-- mayoccur(t)
mayoccur-t1 := in-off & 1 & power-on=1;
mayoccur-t2 := in-on & 1 & power-off=1;
mayoccur-t3 := in-idle & m > 0 & coffee=1;
mayoccur-t4 := in-busy & 1 & done=1;
mayoccur-t5 := in-empty & 1 & inc=1;
mayoccur-t6 := in-notempty & m < 10 & inc=1;
mayoccur-t7 := in-notempty & m > 1 & dec=1;
mayoccur-t8 := in-notempty & m = 1 & dec=1;
-- conflict(t)
conflict-t1 := 0;
conflict-t2 := 0;
conflict-t3 := next(t2)=1;
conflict-t4 := next(t2)=1;
conflict-t5 := next(t2)=1;
conflict-t6 := next(t2)=1 | next(t7)=1 | next(t8)=1;
conflict-t7 := next(t2)=1 | next(t6)=1 | next(t8)=1;
conflict-t8 := next(t2)=1 | next(t6)=1 | next(t7)=1;
-- stable
stable := power-on=0 & power-off=0 & coffee=0 &
    done=0 & inc=0 & dec=0 &
    mayoccur-t1=0 & mayoccur-t2=0 &
    mayoccur-t3=0 & mayoccur-t4=0 &
    mayoccur-t5=0 & mayoccur-t6=0 &
    mayoccur-t7=0 & mayoccur-t8=0;
-- def
def-m := t1 | t5 | t6 | t7 | t8;
-- cuse
cuse-m := t6 | t7;
-- puse
puse-m := t3 | t6 | t7 | t8 | it7 | it10 | it12;
-- use
use-m := cuse-m | puse-m;

ASSIGN
-- INIT
-- configurations
init(CVM) := off;
init(COFFEE) := idle;
init(MONEY) := empty;
-- events
init(dec) := 0;
init(light-on) := 0;
init(light-off) := 0;
init(start) := 0;
init(stop) := 0;
-- variables
init(m) := 0;
-- explicit transitions
init(t1) := 0;
init(t2) := 0;
init(t3) := 0;
init(t4) := 0;
init(t5) := 0;
init(t6) := 0;
init(t7) := 0;
init(t8) := 0;
-- implicit transitions
init(it1) := 0;
init(it2) := 0;
init(it3) := 0;
init(it4) := 0;
init(it5) := 0;
init(it6) := 0;
init(it7) := 0;
init(it8) := 0;
init(it9) := 0;
init(it10) := 0;
init(it11) := 0;
init(it12) := 0;

-- NEXT
-- configurations
next(CVM) := case
```

```
    next(t1)=1: on;
    next(t2)=1: off;
   1: CVM;
esac;
next(COFFEE) := case
    next(t1)=1: idle;
    next(t3)=1: busy;
    next(t4)=1: idle;
    1: COFFEE;
esac;
next(MONEY) := case
    next(t1)=1: empty;
    next(t5)=1: notempty;
    next(t6)=1: notempty;
    next(t7)=1: notempty;
    next(t8)=1: empty;
    1: MONEY;
esac;
-- events
next(power-on) := case
    stable=0: 0;
    1: {0,1};
esac;
next(power-off) := case
    stable=0: 0;
    1: {0,1};
esac;
next(coffee) := case
    stable=0: 0;
    1: {0,1};
esac;
next(done) := case
    stable=0: 0;
    1: {0,1};
esac;
next(inc) := case
    stable=0: 0;
    1: {0,1};
esac;
next(dec) := case
    next(t3)=1: 1;
    1: 0;
esac;
next(light-on) := case
    next(t1)=1: 1;
    1: 0;
esac;
next(light-off) := case
    next(t2)=1: 1;
    1: 0;
esac;
next(start) := case
    next(t3)=1: 1;
    1: 0;
esac;
next(stop) := case
    next(t4)=1: 1;
    1: 0;
esac;
-- variables
next(m) := case
    next(t1)=1: 0;
    next(t5)=1: 1;
    next(t6)=1 & m < 10: m + 1;
    next(t7)=1 & m > 0: m - 1;
    next(t8)=1: 0;
    1: m;
esac;
```

```
-- explicit transitions
next(t1) := case
    mayoccur-t1 & !conflict-t1: 1;
    1: 0;
esac;
next(t2) := case
    mayoccur-t2 & !conflict-t2: 1;
    1: 0;
esac;
next(t3) := case
    mayoccur-t3 & !conflict-t3: 1;
    1: 0;
esac;
next(t4) := case
    mayoccur-t4 & !conflict-t4: 1;
    1: 0;
esac;
next(t5) := case
    mayoccur-t5 & !conflict-t5: 1;
    1: 0;
esac;
next(t6) := case
    mayoccur-t6 & !conflict-t6: 1;
    1: 0;
esac;
next(t7) := case
    mayoccur-t7 & !conflict-t7: 1;
    1: 0;
esac;
next(t8) := case
    mayoccur-t8 & !conflict-t8: 1;
    1: 0;
esac;
-- implicit transitions
next(it1) := case
    in-off & 1 & power-off=1 &
        next(in-off): 1;
    1: 0;
esac;
next(it2) := case
    in-off & 1 & coffee=1 &
        next(in-off): 1;
1: 0; esac;
next(it3) := case
    in-off & 1 & done=1 &
        next(in-off): 1;
    1: 0;
esac;
next(it4) := case
    in-off & 1 & inc=1 &
        next(in-off): 1;
    1: 0;
esac;
next(it5) := case
    in-off & 1 & dec=1 &
        next(in-off): 1;
    1: 0;
esac;
next(it6) := case
    in-on & 1 & power-on=1 &
        next(in-on): 1;
    1: 0;
esac;
next(it7) := case
    in-idle & !(m > 0) & coffee=1 &
        next(in-idle): 1;
    1: 0;
esac;
```

```
next(it8) := case
    in-busy & 1 & coffee=1 &
        next(in-busy): 1;
    1: 0;
esac;
next(it9) := case
    in-idle & 1 & done=1 &
        next(in-idle): 1;
    1: 0;
esac;
next(it10) := case
    in-notempty & !(m < 10) & inc=1 &
        next(in-notempty): 1;
    1: 0;
esac;
next(it11) := case
    in-empty & 1 & dec=1 &
        next(in-empty): 1;
    1: 0;
esac;
next(it12) := case
    in-notempty & m <= 0 & dec=1 &
        next(in-notempty): 1;
    1: 0;
esac;
```

```
-- control flow oriented test coverage
-- state coverage
SPEC ! EF (in-off & EF stable)
SPEC ! EF (in-idle & EF stable)
SPEC ! EF (in-busy & EF stable)
SPEC ! EF (in-empty & EF stable)
SPEC ! EF (in-notempty & EF stable)
-- configuration coverage
SPEC ! EF (in-off & EF stable)
SPEC ! EF (in-idle & in-empty & EF stable)
SPEC ! EF (in-idle & in-notempty & EF stable)
SPEC ! EF (in-busy & in-empty & EF stable)
SPEC ! EF (in-busy & in-notempty & EF stable)
-- strong transition coverage
SPEC ! EF (t1 & EF stable)
SPEC ! EF (t2 & EF stable)
SPEC ! EF (t3 & EF stable)
SPEC ! EF (t4 & EF stable)
SPEC ! EF (t5 & EF stable)
SPEC ! EF (t6 & EF stable)
SPEC ! EF (t7 & EF stable)
SPEC ! EF (t8 & EF stable)
SPEC ! EF (it1 & EF stable)
SPEC ! EF (it2 & EF stable)
SPEC ! EF (it3 & EF stable)
SPEC ! EF (it4 & EF stable)
SPEC ! EF (it5 & EF stable)
SPEC ! EF (it6 & EF stable)
SPEC ! EF (it7 & EF stable)
SPEC ! EF (it8 & EF stable)
SPEC ! EF (it9 & EF stable)
SPEC ! EF (it10 & EF stable)
SPEC ! EF (it11 & EF stable)
SPEC ! EF (it12 & EF stable)
-- data flow oriented test coverage
-- strong all-def coverage
SPEC ! EF(t1 & EX E [!def-m U (use-m & EF stable)])
SPEC ! EF(t5 & EX E [!def-m U (use-m & EF stable)])
SPEC ! EF(t6 & EX E [!def-m U (use-m & EF stable)])
SPEC ! EF(t7 & EX E [!def-m U (use-m & EF stable)])
SPEC ! EF(t8 & EX E [!def-m U (use-m & EF stable)])
-- strong all-use coverage
SPEC ! EF(t1 & EX E [!def-m U (t3 & EF stable)])
SPEC ! EF(t1 & EX E [!def-m U (t6 & EF stable)])
SPEC ! EF(t1 & EX E [!def-m U (t7 & EF stable)])
SPEC ! EF(t1 & EX E [!def-m U (t8 & EF stable)])
SPEC ! EF(t1 & EX E [!def-m U (it7 & EF stable)])
SPEC ! EF(t1 & EX E [!def-m U (it10 & EF stable)])
SPEC ! EF(t1 & EX E [!def-m U (it12 & EF stable)])
SPEC ! EF(t5 & EX E [!def-m U (t3 & EF stable)])
SPEC ! EF(t5 & EX E [!def-m U (t6 & EF stable)])
SPEC ! EF(t5 & EX E [!def-m U (t7 & EF stable)])
SPEC ! EF(t5 & EX E [!def-m U (t8 & EF stable)])
SPEC ! EF(t5 & EX E [!def-m U (it7 & EF stable)])
SPEC ! EF(t5 & EX E [!def-m U (it10 & EF stable)])
SPEC ! EF(t5 & EX E [!def-m U (it12 & EF stable)])
SPEC ! EF(t6 & EX E [!def-m U (t3 & EF stable)])
SPEC ! EF(t6 & EX E [!def-m U (t6 & EF stable)])
SPEC ! EF(t6 & EX E [!def-m U (t7 & EF stable)])
SPEC ! EF(t6 & EX E [!def-m U (t8 & EF stable)])
SPEC ! EF(t6 & EX E [!def-m U (it7 & EF stable)])
SPEC ! EF(t6 & EX E [!def-m U (it10 & EF stable)])
SPEC ! EF(t6 & EX E [!def-m U (it12 & EF stable)])
SPEC ! EF(t7 & EX E [!def-m U (t3 & EF stable)])
SPEC ! EF(t7 & EX E [!def-m U (t6 & EF stable)])
SPEC ! EF(t7 & EX E [!def-m U (t7 & EF stable)])
SPEC ! EF(t7 & EX E [!def-m U (t8 & EF stable)])
SPEC ! EF(t7 & EX E [!def-m U (it7 & EF stable)])
SPEC ! EF(t7 & EX E [!def-m U (it10 & EF stable)])
SPEC ! EF(t7 & EX E [!def-m U (it12 & EF stable)])
SPEC ! EF(t8 & EX E [!def-m U (t3 & EF stable)])
SPEC ! EF(t8 & EX E [!def-m U (t6 & EF stable)])
SPEC ! EF(t8 & EX E [!def-m U (t7 & EF stable)])
SPEC ! EF(t8 & EX E [!def-m U (t8 & EF stable)])
SPEC ! EF(t8 & EX E [!def-m U (it7 & EF stable)])
SPEC ! EF(t8 & EX E [!def-m U (it10 & EF stable)])
SPEC ! EF(t8 & EX E [!def-m U (it12 & EF stable)])
```

# B The counterexample for $\neg EF(t_3 \wedge EF\, stable)$

```
-- specification !EF (t3 & EF stable) is false
-- as demonstrated by the following execution sequence
state 1.1:
stable=0
CVM=off COFFEE=idle MONEY=empty
in-notempty=0 in-empty=0 in-money=0 in-busy=0 in-idle=0 in-coffee=0 in-on=0 in-off=1 in-cvm=1
m=0
power-on=1 power-off=0 coffee=0 done=0 inc=0 dec=0 light-on=0 light-off=0 start=0 stop=0
conflict-t8=0 conflict-t7=0 conflict-t6=0 conflict-t5=0
conflict-t4=0 conflict-t3=0 conflict-t2=0 conflict-t1=0
mayoccur-t8=0 mayoccur-t7=0 mayoccur-t6=0 mayoccur-t5=0
mayoccur-t4=0 mayoccur-t3=0 mayoccur-t2=0 mayoccur-t1=1
t1=0 t2=0 t3=0 t4=0 t5=0 t6=0 t7=0 t8=0
it1=0 it2=0 it3=0 it4=0 it5=0 it6=0 it7=0 it8=0 it9=0 it10=0 it11=0 it12=0

state 1.2:
stable=1
CVM=on in-empty=1 in-money=1 in-idle=1 in-coffee=1 in-on=1 in-off=0
power-on=0 light-on=1
mayoccur-t1=0 t1=1

state 1.3:
stable=0
inc=1 light-on=0
mayoccur-t5=1 t1=0

state 1.4:
stable=1
MONEY=notempty in-notempty=1 in-empty=0
inc=0 m=1
mayoccur-t5=0 t5=1

state 1.5:
stable=0
coffee=1
mayoccur-t3=1 t5=0

state 1.6:
COFFEE=busy in-busy=1 in-idle=0
coffee=0 dec=1 start=1
mayoccur-t8=1 mayoccur-t3=0 t3=1

state 1.7:
stable=1
MONEY=empty in-notempty=0 in-empty=1
m=0
dec=0 start=0
mayoccur-t8=0 t3=0 t8=1

resources used:
user time: 0.36 s, system time: 0.03 s
BDD nodes allocated: 14070
Bytes allocated: 1376256
BDD nodes representing transition relation: 4496 + 6
```

# C Test Suites for the Coffee Vending Machine

### State Coverage

| state | result |
|---|---|
| OFF | $\emptyset/\emptyset$ |
| IDLE | $\{power\text{-}on\}$ / $\{light\text{-}on\}$ |
| BUSY | $\{power\text{-}on\}$,$\{inc\}$,$\{coffee\}$ / $\{light\text{-}on\}$,$\emptyset$,$\{start\}$ |
| EMPTY | $\{power\text{-}on\}$ / $\{light\text{-}on\}$ |
| NOTEMPTY | $\{power\text{-}on\}$,$\{inc\}$,$\{inc\}$ / $\{light\text{-}on\}$,$\emptyset$,$\emptyset$ |

### Configuration Coverage

| state | result |
|---|---|
| {OFF} | $\emptyset/\emptyset$ |
| {IDLE, EMPTY} | $\{power\text{-}on\}$ / $\{light\text{-}on\}$ |
| {IDLE, NOTEMPTY} | $\{power\text{-}on\}$,$\{inc\}$,$\{inc\}$ / $\{light\text{-}on\}$,$\emptyset$,$\emptyset$ |
| {BUSY, EMPTY} | $\{power\text{-}on\}$,$\{inc\}$,$\{coffee\}$ / $\{light\text{-}on\}$,$\emptyset$,$\{start\}$ |
| {BUSY, NOTEMPTY} | $\{power\text{-}on\}$,$\{inc\}$,$\{coffee\}$ / $\{light\text{-}on\}$,$\emptyset$,$\{start\}$ |

### Strong Transition Coverage

| transition | result |
|---|---|
| $t_1$ | $\{power\text{-}on\}$ / $\{light\text{-}on\}$ |
| $t_2$ | $\{power\text{-}on\}$,$\{power\text{-}off\}$ / $\{light\text{-}on\}$,$\{light\text{-}off\}$ |
| $t_3$ | $\{power\text{-}on\}$,$\{inc\}$,$\{coffee\}$ / $\{light\text{-}on\}$,$\emptyset$,$\{start\}$ |
| $t_4$ | $\{power\text{-}on\}$,$\{inc\}$,$\{coffee\}$,$\{done\}$ / $\{light\text{-}on\}$,$\emptyset$,$\{start\}$,$\{stop\}$ |
| $t_5$ | $\{power\text{-}on\}$,$\{inc\}$ / $\{light\text{-}on\}$,$\emptyset$ |
| $t_6$ | $\{power\text{-}on\}$,$\{inc\}$,$\{inc\}$ / $\{light\text{-}on\}$,$\emptyset$,$\emptyset$ |
| $t_7$ | $\{power\text{-}on\}$,$\{inc\}$,$\{inc\}$,$\{dec\}$ / $\{light\text{-}on\}$,$\emptyset$,$\emptyset$,$\emptyset$ |
| $t_8$ | $\{power\text{-}on\}$,$\{inc\}$,$\{dec\}$ / $\{light\text{-}on\}$,$\emptyset$,$\emptyset$ |
| $it_1$ | $\{power\text{-}off\}$ / $\emptyset$ |
| $it_2$ | $\{coffee\}$ / $\emptyset$ |
| $it_3$ | $\{done\}$ / $\emptyset$ |
| $it_4$ | $\{inc\}$ / $\emptyset$ |
| $it_5$ | *infeasible* |
| $it_6$ | $\{power\text{-}on\}$,$\{power\text{-}on\}$ / $\{light\text{-}on\}$,$\emptyset$ |
| $it_7$ | $\{power\text{-}on\}$,$\{coffee\}$ / $\{light\text{-}on\}$,$\emptyset$ |
| $it_8$ | $\{power\text{-}on\}$,$\{inc\}$,$\{coffee\}$,$\{coffee\}$ / $\{light\text{-}on\}$,$\emptyset$,$\{start\}$,$\emptyset$ |
| $it_9$ | $\{power\text{-}on\}$,$\{done\}$ / $\{light\text{-}on\}$,$\emptyset$ |
| $it_{10}$ | $\{power\text{-}on\}$,$\{inc\}$,$\{inc\}$,$\{inc\}$,$\{inc\}$,$\{inc\}$, $\{inc\}$,$\{inc\}$,$\{inc\}$,$\{inc\}$,$\{inc\}$,$\{inc\}$ / $\{light\text{-}on\}$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$ |
| $it_{11}$ | *infeasible* |
| $it_{12}$ | *infeasible* |

**Strong all-use coverage**

| tuple | result |
|---|---|
| $(m, t_1, t_3)$ | *infeasible* |
| $(m, t_1, t_6)$ | *infeasible* |
| $(m, t_1, t_7)$ | *infeasible* |
| $(m, t_1, t_8)$ | *infeasible* |
| $(m, t_1, it_7)$ | {*power-on*},{*coffee*}/{*light-on*},∅ |
| $(m, t_1, it_{10})$ | *infeasible* |
| $(m, t_1, it_{12})$ | *infeasible* |
| $(m, t_5, t_3)$ | {*power-on*},{*inc*},{*coffee*} / {*light-on*},∅,{*start*} |
| $(m, t_5, t_6)$ | {*power-on*},{*inc*},{*inc*} / {*light-on*},∅,∅ |
| $(m, t_5, t_7)$ | *infeasible* |
| $(m, t_5, t_8)$ | {*power-on*},{*inc*},{*dec*} / {*light-on*},∅,∅ |
| $(m, t_5, it_7)$ | *infeasible* |
| $(m, t_5, it_{10})$ | *infeasible* |
| $(m, t_5, it_{12})$ | *infeasible* |
| $(m, t_6, t_3)$ | {*power-on*},{*inc*},{*inc*},{*coffee*} / {*light-on*},∅,∅,{*start*} |
| $(m, t_6, t_6)$ | {*power-on*},{*inc*},{*inc*},{*inc*} / {*light-on*},∅,∅,∅ |
| $(m, t_6, t_7)$ | {*power-on*},{*inc*},{*inc*},{*dec*} / {*light-on*},∅,∅,∅ |
| $(m, t_6, t_8)$ | *infeasible* |
| $(m, t_6, it_7)$ | *infeasible* |
| $(m, t_6, it_{10})$ | {*power-on*},{*inc*},{*inc*},{*inc*},{*inc*},{*inc*}, {*inc*},{*inc*},{*inc*},{*inc*},{*inc*},{*inc*} / {*light-on*},∅,∅,∅,∅,∅,∅,∅,∅,∅,∅,∅ |
| $(m, t_6, it_{12})$ | *infeasible* |
| $(m, t_7, t_3)$ | {*power-on*},{*inc*},{*inc*},{*dec*},{*coffee*} / {*light-on*},∅,∅,∅,{*start*} |
| $(m, t_7, t_6)$ | {*power-on*},{*inc*},{*inc*},{*dec*},{*inc*} / {*light-on*},∅,∅,∅,∅ |
| $(m, t_7, t_7)$ | {*power-on*},{*inc*},{*inc*},{*inc*},{*dec*},{*dec*} / {*light-on*},∅,∅,∅,∅,∅ |
| $(m, t_7, t_8)$ | {*power-on*},{*inc*},{*inc*},{*dec*},{*dec*} / {*light-on*},∅,∅,∅,∅ |
| $(m, t_7, it_7)$ | *infeasible* |
| $(m, t_7, it_{10})$ | *infeasible* |
| $(m, t_7, it_{12})$ | *infeasible* |
| $(m, t_8, t_3)$ | *infeasible* |
| $(m, t_8, t_6)$ | *infeasible* |
| $(m, t_8, t_7)$ | *infeasible* |
| $(m, t_8, t_8)$ | *infeasible* |
| $(m, t_8, it_7)$ | {*power-on*},{*inc*},{*dec*},{*coffee*} / {*light-on*},∅,∅,∅ |
| $(m, t_8, it_{10})$ | *infeasible* |
| $(m, t_8, it_{12})$ | *infeasible* |