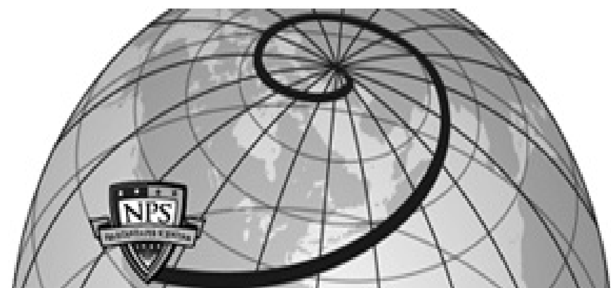
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers' Publications

2017-08-10

Reverse engineering concurrent UML state machines using black box testing and genetic programming

Drusinsky, Doron

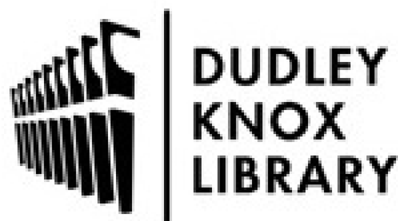
SpringerLink

Drusinsky, Doron. "Reverse engineering concurrent UML state machines using black box testing and genetic programming." Innovations in Systems and Software Engineering 13.2-3 (2017): 117-128.

http://hdl.handle.net/10945/59207

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

Innovations Syst Softw Eng (2017) 13:117–128
DOI 10.1007/s11334-017-0299-9

ORIGINAL PAPER

Reverse engineering concurrent UML state machines using black box testing and genetic programming

Doron Drusinsky[1]

Received: 2 June 2017 / Accepted: 26 July 2017 / Published online: 10 August 2017
© Springer-Verlag London (outside the USA) 2017

Abstract This paper presents a technique for reverse engineering, a software system generated from a concurrent unified modeling language state machine implementation. In its first step, a primitive sequential finite-state machine (FSM) is deduced from a sequence of outputs emitted from black box tests applied to the systems' input interface. Next, we provide an algorithmic technique for decomposing the sequential primitive FSM into a set of concurrent (orthogonal) primitive FSMs. Lastly, we show a genetic programming machine learning technique for discovering local variables, actions performed on local and non-binary output variables, and two types of intra-FSM loops, called counting-loops and while-loops.

Keywords Machine Learning · Concurrent UML state machines · Concurrency decomposition · Genetic programming · Reverse engineering

1 Introduction

A finite state machine (FSM) is a mathematical model of computation that consists of a finite set of states and interconnection state transitions. An FSM transitions from one state to another in response to an external input event. An FSM also has (optional) actions that are executed within states or as transitions are traversed; actions are in the form of binary assignments to output variables. The software implementation of an FSM is straightforward; it typically consists of a single state variable to store the present state, and a set of if-statements, one per transition of the FSM.

There is ample motivation to reverse engineer FSMs, with applications ranging from security [4,13] and verification [3,5] to the representation of client-side behavior of rich Internet applications [2].

Unified modeling language (UML) state machines extend FSM basic behavior with features such as: state nesting, state machine concurrence (orthogonality), local variables, transition guards, flowcharts within state machines, non-binary outputs, and action specified using a textual action language. UML state machines and corresponding software implementation techniques are reviewed in Sect. 2.

This paper is concerned with the reverse engineering UML state machine software implementations from black box test evidence.

In [9], the author proposed a white-box technique for extracting the underlying UML statechart structure of an FSM. This technique does not discover local variables, transition guards, flowcharts within state machines, non-binary outputs, or textual actions. In contrast, in addition to discovering these artifacts, our technique assumes no a priori know-how of the internal structure of the FSM.

Angluin's well-known L* algorithm [1] learns an unknown regular language over a known alphabet and produces a deterministic finite-state automaton (DFA) that accepts it. DFAs have binary *accept/reject* outputs manifested by the DFA's final states. DFAs have any number final and non-final states, where states are used as a form of internal memory, typically for counting purposes. In contrast, our technique is applied to UML state machines where binary and/or non-binary outputs as well as local variables are used

This research was funded by a Grant from the Office of Naval Research (ONR). The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

✉ Doron Drusinsky
ddrusins@nps.edu

[1] Computer Science Department, Naval Postgraduate School, Monterey, CA 93943, USA

for memorization/counting. We apply machine learning to learn counting-related attributes.

Genetic programming (GP) is a technique whereby computer programs are encoded as a set of genes that are then modified (evolved) using an evolutionary algorithm. GP is inspired by biological evolution and its fundamental mechanisms; GP software systems implement an algorithm that uses random mutation, crossover, a fitness function, and multiple generations of evolution to resolve a user-defined task. GP can be used to discover a functional relationship between features in data (symbolic regression), to group data into categories (classification), and to assist in various AI applications, such as the design of electrical circuits, antennae, or quantum algorithms. GP is applied to software engineering through code synthesis, genetic improvement, automatic bug-fixing, in developing game-playing strategies, and more [7]. GP is overviewed in Sect. 2.3.

In this paper, we describe a technique for reverse engineering UML state machine software implementations given black box test evidence. The proposed technique consists of three main parts. In Sect. 3, we reverse engineer a primitive underlying FSM using black box testing of the underlying UML system under test (SUT). In Sect. 4, we describe a technique for decomposing that primitive FSM into a collection of orthogonal FSMs; after doing so we test each FSM independently of the others and apply white-box testing to generate data for the final, machine learning, step. In Sect. 5, we describe a genetic programming technique for reverse engineering intra-FSM local variables, complex actions performed on non-binary outputs, and two types of loops, called counting-loops and while-loops.

2 Background: concurrent UML state machines and genetic programming

2.1 Concurrent UML state machines

UML state machines are state diagrams augmented with state hierarchy, flowcharts within state diagrams, event-guard annotation of state transitions, local variables, binary and non-binary variables (i.e., local and I/O variables), and actions assigning values to local and I/O variables using a local, text-based computation.

UML state machines also cater for concurrency (also referred to as orthogonality) within any state of the state machine. This paper is concerned with concurrent UML state machines where concurrency is allowed only on the top most level of state hierarchy.

Figure 1 illustrates the behavior (Fig. 1a) and input/output (I/O) interface (Fig. 1b) of a concurrent UML for a car's body logic. The input interface consists of a set Σ of input events, $\Sigma = \{$*engineOn, engineOff, timer_fire*, *radioOn, radioOff, changeVol*$\}$. The output interface consists of a set of output variables O = $\{$*radioOff, volume, lock, unlock, doCalibrate*$\}$, where all but *volume* are considered binary command (event) outputs, such as $lock = 1$ meaning "lock the car." The distinction between binary and non-binary outputs is important because, in the first step of our algorithm (Sect. 3), it produces a primitive FSM using binary output information only. This assumption is relaxed in later steps. In Sect. 3, we also consider the earliest assignment made to a non-binary output as a binary output; for example, the action *setVolume*(MIN) depicted in state *RadioOff* shown in Fig. 1a is exhibited as $volume = 10$ in test outputs; hence, the first assignment made to volume within the test ($volume = 10$) is considered a binary output.

The UML state machine shown in Fig. 1a consists of two orthogonal FSMs, one named *Engine* and the other named *Radio*. These two FSMs are indeed *orthogonal* because the state transitions in one do not depend on states in the other.

The *Engine* FSM implements the following functionality:

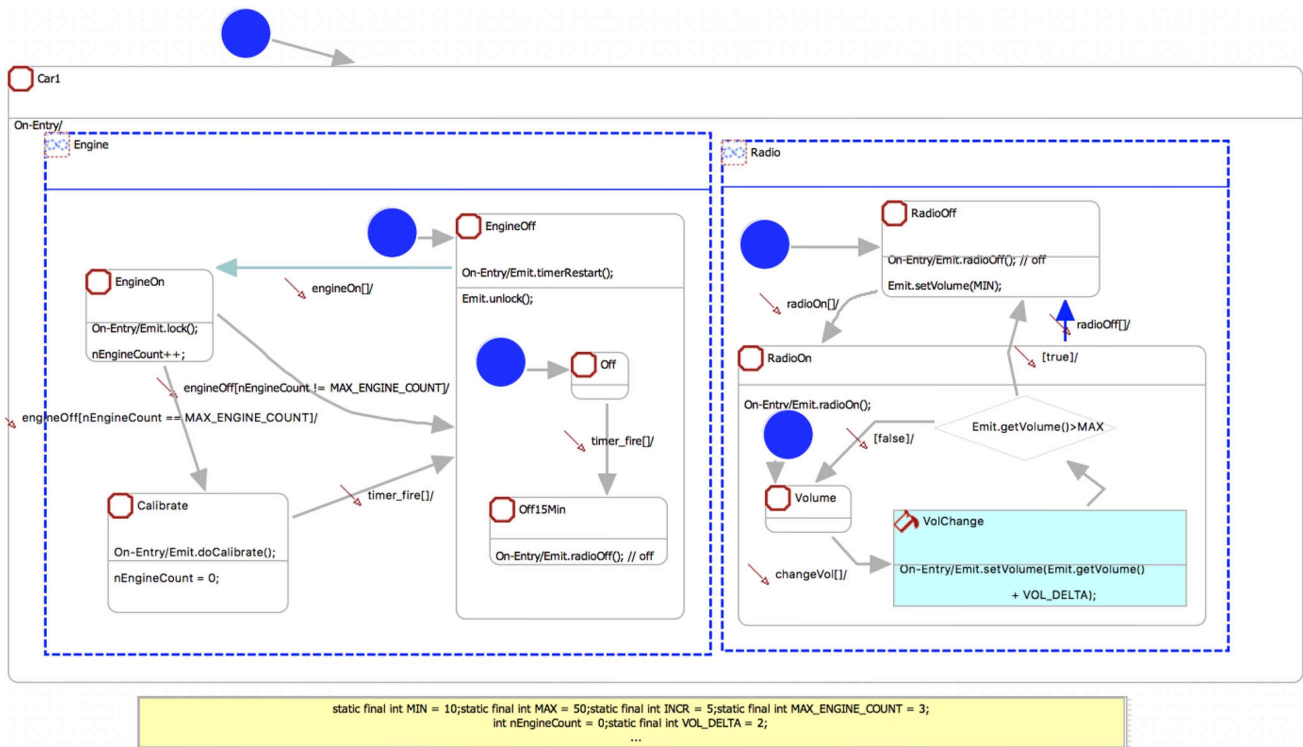
- It counts the number of successive times the engine has been turned on and off; when that count reaches a certain number (MAX_ENGINE_COUNT) the state machine issues a binary calibration output command (*doCalibrate*).
- When the engine is turned on (off), the doors are locked (unlocked), using a binary *lock* and *unlock* output commands, respectively.
- The radio is automatically turned off some time (at least 15 min) after the engine had been turned off. This is done using a binary *radioOff* output command.

Note that *engineOn* and *engineOff* are events, *nEngineCount* is a local (non-binary) variable. Note how UML transitions are annotated using the *event*[*guard*] notation, such as *engineOff* [*nEngineCount==MAX_ENGINE_COUNT*], which means that the transition fires when event *engineOff* occurs, but only under the condition that the (local) variable *nEngineCount* equals the constant value *MAX_ENGINE_COUNT.*

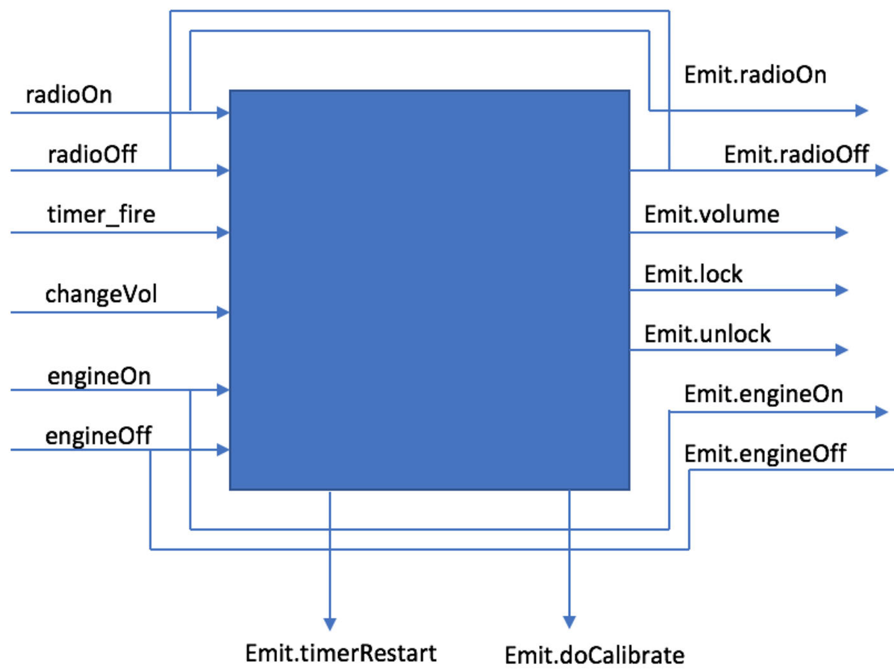
Using the StateRover [12] notation, local variables such as *nEngineCount* are declared in a local variables box, depicted in the bottom of Fig. 1.

The Radio FSM controls the radio's volume as follows. The car contains a *volume* button that emits an event called *changeVol*. When the radio is on, then every time the *changeVol* event occurs, volume is incremented as specified by a computation that takes place within the *VolChange* flowchart activity box. The FSM then proceeds to one of the two possible next states via the flowchart decision polygon.

Often, state-changing event sets in each FSM are mutually exclusive. For example, in the car example shown in Fig. 1, the event set for the *Engine* FSM is $\{$*engineOn, engineOff, timer_fire*$\}$ whereas the event set for the *Radio* FSM is



a Car's internal behavior: UML state diagram



b The Car's Input-Output (I/O) interface.

Fig. 1 A concurrent UML state machine for car: internal behavior and I/O interface. **a** car's internal behavior: UML state diagram. **b** The car's input/output (I/O) interface

{*radioOn, radioOff, changeVol*}. Each event set is referred to as *FSM event set*, or *FSM events*. The concurrence discovery algorithms of Sect. 4 assumes that the intra-FSM event sets are mutually exclusive.

In the sequel, we will distinguish between a state machines' *basic behavior* and its *extended UML behavior*. Basic behavior consists of states, transitions labeled with symbols (events), and state actions that assign binary outputs. Extended UML behavior extends basic behavior by enabling transition guards, local variables, and a textual action language, all discussed below. Sections 3 and 4 are concerned with reverse engineering an FSM's basic behavior, whereas Sect. 5 is concerned with reverse engineering its extended UML behavior.

Note that the car system contains two guards-dependent loops, one in each member FSM. The *Engine* FSM loop (two loops actually) is conditioned on the local integer variable *nEngineCount* to decide whether to calibrate or not. The *Radio* FSM depends on an output non-binary variable (*volume*). In Sect. 5, we will use genetic programming to discover the mathematical formulae assigned to these variables as well as the associated guards.

2.2 Concurrent UML state machine implementation

A trivial and highly inefficient implementation strategy for concurrent UML state machines is to convert such a machine into a single equivalent sequential finite state machine (FSM). This approach induces an FSM state set that is the Cartesian product of the states in each set, an inefficiency caused by essentially ignoring the design information available in the original concurrent state machine.

Alternatively, there are two prevailing implementations that preserve the UML diagrams' concurrence within the implementation code, as follows:

1. The implementation suggested by the author [6] represents concurrence as an array of state variables, with intra-FSM transitions inducing state changes to their respective state variables.
2. The object-oriented approach of [14] uses classes to implements state hierarchy (super states), such as the *EngineOff* state of the *Engine* FSM in Fig. 1. More specifically, each superstrate, along with its intra-state transitions, is implemented as an object in memory.

With both approaches, an input sequence such as $seq = engineOn, radioOn, changeVol, engineOff$ induces a sequence of *interleaved state changes*. For example, given the present state pair of *<EngineOff/Off, RadioOff>* (in the *Engine* FSM and *Radio* FSM, respectively), the state change sequence induced by *seq* is: *<EngineOff, RadioOff>*, *<EngineOn, RadioOff>*, *<EngineOn, RadioOn/Volume>* , *<EngineOn, RadioOn/Volume>*, *<EngineOff, RadioOn/Volume>*. Note that the transition *RadioOn/Volume* to *RadioOn/Volume* occurred via the flowchart box and flowchart decision polygon and was accompanied by a change to the *volume* output variable.

Clearly, as shown by the above example, the state change sequence induced by an input test is an *interleaving* of the FSM states changes for all FSMs within the UML state machine.

2.3 Genetic programming

GP is a machine learning technique whereby a computer program, algebraic function, or some other learning object is encoded as a set of genes that are then evolved using an evolutionary genetic algorithm (GA) [11]. In this paper, we apply GP to learn UML state machine extended behavior, including variables, actions, and transition guards related to a reversed-engineered UML state machine; details of these learning objectives are provided in Sect. 5.2. Using this domain of discourse as an example, a genetic programming algorithm consists of all or most of the following steps [8]:

1. Randomly create an initial population of individual FSMs, each being a primitive FSM whose basic behavior is augmented with the machine learning objective entities pertaining to an extended UML behavior, namely: local variables, actions, and transition guards. Using GP terminology, we also refer to these learning objectives as *genes*.
2. Iteratively perform the following sub-steps on the population, until the termination criterion is satisfied:
   a. Test each FSM in the population and determine its fitness; the fitness criterion is defined in Sect. 5.
   b. Select a subset of FSMs from the population with a probability based on fitness to participate in the genetic operations of (c).
   c. Create new individual FSMs for the population by applying the following genetic operations with specified probabilities:
      i. Reproduction: copy the selected individual FSM to the new population.
      ii. Crossover: create new offspring FSMs for the new population by recombining randomly chosen learning objective entities from two selected FSMs.
      iii. Mutation: create one new offspring FSM for the new population by randomly mutating a randomly chosen learning objective entities of one selected FSM.

After the termination criterion is satisfied, the single best individual FSM in the population produced during the run is designated as the output result.

3 Synthesis of a primitive underlying FSM using black box testing

Black box testing is the process of repeatedly injecting the SUT with input events declared in its input interface (e.g., the interface shown in Fig. 1b) without assuming knowledge of the SUT's internal design or behavior. In this section, we reverse engineer a primitive FSM from the observations generated by executing a black box test suite. The resulting FSM will be primitive in the following sense:

1. It will be flat and sequential, i.e., it will contain no concurrence or nested states.
2. It will contain no local variables.
3. All output variables will be binary. State actions on those outputs will consist of binary assignments only.
4. State transitions will be annotated with events from the input interface.

Listing 1 contains a snippet of the output generated by black box testing of the UML state machine shown in Fig. 1. For example, at time $t = 0$ the machine emits the binary outputs *timerRestart*, *unlock*, and *radioOff*, and assigns of the initial value 10 to the non-binary output variable volume. Next, at time $t = 1$, following the event *engineOn*, the machine emits the binary output *lock*.

```
t=0
Emit.timerRestart
Emit.unlock
Emit.radioOff
Emit.volume=10
t=1
Event=engineOn
Emit.lock
t=10
Event=radioOn
Emit.radioOn
t=20
Event=engineOff
Emit.timerRestart
Emit.unlock
t=30
Event=engineOn
Emit.lock
t=40
Event=radioOff
Emit.radioOff
Emit.volume=10
```

Listing 1. A snippet of the snippet of the output generated by black-box testing of the UML state machine of Figure 1.

The primitive FSM induced by such a test output is straightforward, as depicted in Fig. 2. Each set of binary output assignments emitted at a certain time stamp is considered a state, such as the output set {*timerRestart*, *unlock*} being emitted at time $t = 20$ in Listing 1, and the output set {*lock*} being emitted at time $t = 30$; in Fig. 2, these induced states are denoted as *timerRestart*$*unlock* and *lock*, respectively.[1] State outputs in the primitive FSM are obvious. State transitions are also deduced explicitly from the test output: a pair of successive induced states in the test, such as *timerRestart*$*unlock* and *lock* in Listing 1, induces a transition the primitive FSM; the transition event is the event listed in the test (e.g., at time $t = 30$ the event is *engineOn*).

Formally, the generated primitive FSM consists of the FSM states and transitions of a classical FSM; i.e.,

- A set of primitive states: S, where a state $s \in S$ contains a set $\underline{o}(s)$ of binary assignments to output variables.
- A set of state transitions: T, where a transition $t \in T$ is annotated with an event of Σ.

Since our SUT is an implementation of a concurrent UML state machine, tests consist of sequences of interleavings of intra-FSM events. There are several test generation approaches that can help unveil the primitive FSM representation of the SUT; the following a few brief summaries of two such techniques:

1. *Perfect Interleaving Testing (PIT)*. The PIT approach generates a test suite that contains all possible interleavings of events declared in the SUT's input interface; in Sect. 4 we also use PIT to perform FSM decomposition, i.e., to discover concurrence within the primitive FSM. Suppose the maximal length of any test in the suite is n, then given an input interface Σ, perfect interleaving test suite consists of $|\Sigma|^n$ test sequences. Clearly, such exponential growth is unmanageable even for moderate sized n's. Hence, to use PIT, one must use very short tests, such as using $n = |S|$. If $|\Sigma|^n$ is nevertheless prohibitively large, the FSM decomposition technique discussed in Sect. 4 can nevertheless be applied with non-perfect interleaving testing.
2. A random *black box testing* based technique.
   a. Use random testing to induce a primitive FSM S.
   b. Discover new states, as follows:
      i. For each state $s \in S$, use graph algorithm such as depth-first search to discover all simple paths in S that lead to s. Each such transition-sequence $p = t_1, t_2, \ldots, t_m$ induces a sequence of input events, $e = e_1, e_2, \ldots, e_m$, where e_i is the event for transition t_i.

[1] The initial state of the primitive FSM is the only state that is not induced by outputs of a *one* constituent FSM within the UML statechart, but rather is induced by outputs of *all* constituent FSMs.

ii. For each such sequence e:
   1. Derive $|\Sigma|$ new tests from e, each test being an extension of e with an event from Σ.
   2. Test S uses these $|\Sigma|$ tests. Given a test $test_i$, let o be the output set generated by the last step of the $test_i$. If for all $s \in S$, $\underline{o}(s) \neq o$, it means that a new state s' has just been discovered, where $\underline{o}(s') = o$; add s' to S and proceed to (i).

Note how the primitive FSM shown in Fig. 2 is non-deterministic. For example, state *lock* has two outgoing transitions labeled *engineOff*. The reason for this non-determinism is that the original UML state machine had two transitions labeled *engineOff*[*cond*] and *engineOff*[!*cond*], respectively, where *cond* is a Boolean guard; such transitions usually control loops within the UML state machine. In Sect. 5, we describe a procedure for discovering loop-controlling UML state transition guards.

We classify every primitive FSM transition $t \in T$ as one of two types:

1. *Original transition* A transition that exists in the original concurrent UML state machine, i.e., in one of its member FSMs. For example, the transition *timerRestart*$*unlock* $\rightarrow_{engineOn}$ *lock* shown in Fig. 2 is an original transition because it is actually the transition *Off* $\rightarrow_{engineOn}$ *EngineOn* in the Engine FSM shown in Fig. 1.
2. *Interleaved transition* A transition results from the order of output-emissions in the interleaved testing. Such transition does not exist in the original concurrent/orthogonal UML state machine; it actually connects two states that reside in orthogonal FSMs. For example, the transition *doCalibrate* $\rightarrow_{radioOff}$ *radioOff*$*volume = 10* shown in Fig. 2 is an interleaved transition.

Note that given our black box approach, we do not have a priori knowledge of this information. Nevertheless, in Sect. 3 we will be using the following property.

The interleaved transitions property: consider a concurrent UML state machine with c_{conc} concurrent, member FSMs: FSM_1, FSM2, ..., FSM_{cconc}. Let events (FSM_i) be the events triggering transitions of FSM_i, and let $A_i \rightarrow_{evt} B_i$ be an original transition in FSM_i. Under the perfect interleaved testing assumption, an induced primitive FSM will contain an interleaved transition labeled *evt* from every state of every FSM_j, $j \neq i$, to B_i.

4 Discovering concurrent FSMs within a primitive FSM

In this section, we present a technique for discovering a decomposition of the primitive FSM into concurrent/orthogonal FSMs. In the first step, we create a data structure called the *1-hot transition table* (abbreviated as 1HTT). It is a straightforward table with $|S| + 2$ columns as follows:

- $|S|$ columns called *source-state columns*, one per state of the primitive FSM.
- An *event column*.
- A *target state* column.

Each row of the 1HTT represents a set of states of the primitive FSM, denoted $\chi(evt.s_2)$, that share the same event (evt) and target state (s_2); formally $\chi(evt.s_2) = \{s_1 \in S | \exists t \in T\, s.t.\, t = (s_1 \rightarrow_{\text{evt}} s_2)\}$. In the (one and only one) 1HTT row for the <event, target state> pair, <*evt*, s_2 >, the 1HTT contains a 1 per state in $\chi(evt.s_2)$. Figure 3 depicts the 1HTT for the primitive FSM shown in Fig. 1.

Let the dimensions of the 1HTT be: $w_{1\text{HTT}}$ columns and $h_{1\text{HTT}}$ rows; for Fig. 3 these dimensions are $w_{1\text{HTT}} = 8$ and $h_{1\text{HTT}} = 9$. The last 2 columns of the 1HTT are the event and target state columns; the number of source-state columns is denoted $ws_{1\text{HTT}}$, where $ws_{1\text{HTT}} = w_{1\text{HTT}} - 2$.

To discover the decomposition of the primitive FSM into concurrent FSMs, we will decompose its 1HTT in a manner that reveals interleaved transitions based on the interleaved transitions property. Hence, we define the following decomposition of a 1HTT:

Perturb rows and columns of the 1HTT such that:[2]

a. There exists integers m, n such that: $2 < m < ws_{1\text{HTT}} - 1$, and $2 < n < h_{1\text{HTT}}$-1.
b. All 1HTT cells $[1 : m, 1 : n]$, i.e., the top left corner, contain 1's.
c. All 1HTT cells $[m+1 : ws_{1\text{HTT}}, n+1 : h_{1\text{HTT}}]$, i.e., the bottom right corner, contain 1's.
d. For every row of each such rectangle, the target state is not one of the source states within that rectangle.
e. The event sets related to the rows of one rectangle are mutually exclusive to those of the other rectangle.

When the 1HTT is decomposable with some pair of numbers m, n we say that m and n are the *1HTT decomposition parameters*.

An alternate decomposition discovers top right/bottom left rectangles of 1's. All decomposition algorithms presented in the sequels search for either type.

The *generic 1HTT transformation* is to shuffle the rows and columns until the 1HTT decomposition condition is satisfied.

[2] Row and column numbers are 1-based.

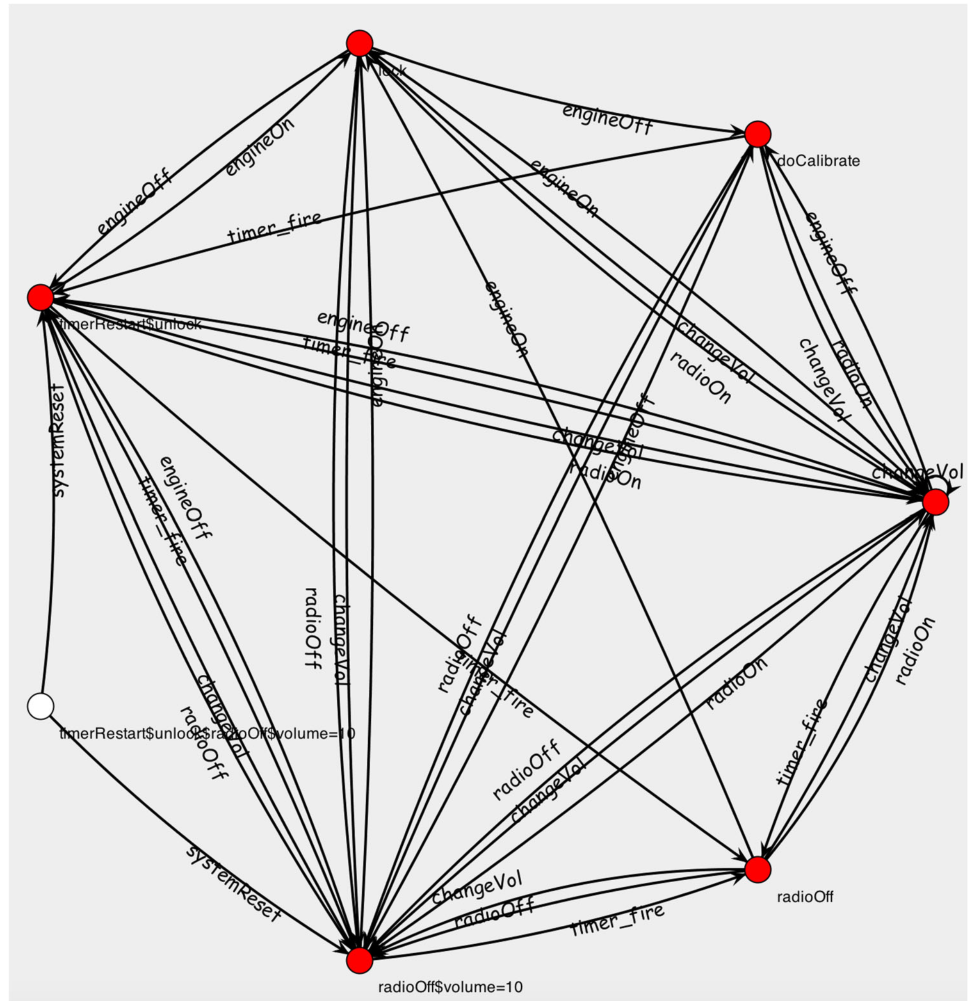


Fig. 2 Primitive FSM induced by perfect interleaved testing of the input UML state machine. The white state is the initial state. The state without a name is such because it has no outputs

Fig. 3 1HTT for the primitive FSM shown in Fig. 2. The source and target state with an empty name corresponds to a state shown in Fig. 1 with no output actions. Note that the table does not contain rows for transitions from the initial state shown in Fig. 2

Source state	timerRestart$unlock	radioOff	lock	doCalibrate	radioOff$volume=10		Event	Target-State
	1	1			1	1	engineOn	lock
			1		1	1	engineOff	doCalibrate
			1		1	1	engineOff	timerRestart$unlock
				1	1	1	timer_fire	timerRestart$unlock
	1				1	1	timer_fire	radioOff
	1	1	1	1		1	radioOff	radioOff$volume=10
	1	1	1	1	1		radioOn	
	1	1	1	1		1	changeVol	radioOff$volume=10
	1	1	1	1		1	changeVol	

Fig. 4 A 1HTT transformation version of the 1HTT shown in Fig. 3; the rectangles depict a 1HTT decomposition

radioOff$volume=10		timerRestart$unlock	radioOff	lock	doCalibrate	Event	Target-State
1	1	1	1			engineOn	lock
1	1			1		engineOff	doCalibrate
1	1			1		engineOff	timerRestart$unlock
1	1				1	timer_fire	timerRestart$unlock
1	1	1				timer_fire	radioOff
	1	1	1	1	1	radioOff	radioOff$volume=10
1		1	1	1	1	radioOn	
	1	1	1	1	1	changeVol	radioOff$volume=10
	1	1	1	1	1	changeVol	

Figure 4 depicts the transformed and decomposed version of the 1HTT shown in Fig. 3. For example, the *lock* target state of row 1 is not a source state within the top left triangle, complying with condition (d).

Condition (e) corresponds to the assumption made in Sect. 2, that intra-FSM event sets are mutually exclusive. 1HTT decomposition is a decomposition of the states and

events of the primitive FSM into two sets, corresponding to the states and events of two orthogonal FSMs.

Conditions (a–d) reflect the interleaved transitions property, as follows. Consider an arbitrary cell in row *r* of one of the rectangles. It maps a source state in one orthogonal FSM to a target state *s* in another FSM via a transition whose event is *evt*; given the interleaved transitions property, all source states in the same FSM should have a transition with event *evt* and target state *s—indeed, according to the 1HTT decomposition condition, the entire portion of row r contained in that rectangle contains 1's.*

The 1HTT transformation induces two orthogonal FSMs, as follows:

- Target states associated with the top (bottom) rectangle are target states of the first (second) FSM.
- All 1's outside the two rectangles correspond to local transitions within the corresponding FSMs.
- In each FSM, the state that appears the earliest in the test output is declared as the initial state of that FSM.

Figure 5 depicts the two concurrent FSMs induced by the 1HTT shown in Fig. 4.

Given a decomposition into two FSMs, the same procedure can be applied recursively to each FSM, further decomposing it.

4.1 1HTT transformation algorithms

The following two concrete algorithms implement the generic 1HTT transformation.
Algorithm A: a brute force search.

- Algorithm A.1; used if $|\Sigma| \leq |S|$. There are $2^{|\Sigma|}$ subsets of Σ; For each $\Sigma' \subseteq \Sigma$; do:
    1. Move the rows of the 1HTT such that the rows whose event column values belong to Σ' reside above rows whose event column values belong to $\Sigma - \Sigma$'.
    2. Search for a column number k, $2 < k < ws_{1\text{HTT}} - 1$ such that k and $|\Sigma'|$ are parameters of a valid 1HTT decomposition. If such a k exists then stop; a valid 1HTT decomposition has been discovered.
- Algorithm A.2; used if $|S| \leq |\Sigma|$. There are $2^{|S|}$ subsets of S; For each subset $S' \subseteq S$ do:
    1. Move the columns of the 1HTT such that the columns whose source-state column values belong to S' reside to the left of columns whose source-state column values belong to $S - S$'.
    2. Search for a column number k, $2 < k < h_{1\text{HTT}} - 1$ such that k and $|S'|$ are valid 1HTT decomposition condition parameters. If such a k exists then stop; a valid 1HTT decomposition has been discovered.

Algorithm B:

- Algorithm B.1:
    1. Shuffle the 1HTT rows so that they are sorted by a descending total number of 1's in the source-state entries of each row.
    2. Shuffle the 1HTT columns so that all 1's of the first row are packed on the left.
    3. Perform step 2 of Algorithm A.2.
- Algorithm B.2:
    1. Shuffle the 1HTT columns so that they are sorted by the descending total number of 1's in the source-state entries of each column.
    2. Shuffle the 1HTT rows so that all 1's of the first column are packed on the top.
    3. Perform step 2 of Algorithm A.1.

Step 3 of these algorithms is not guaranteed to discover a valid 1HTT decomposition. However, these algorithms are much faster than the brute force Algorithm A.

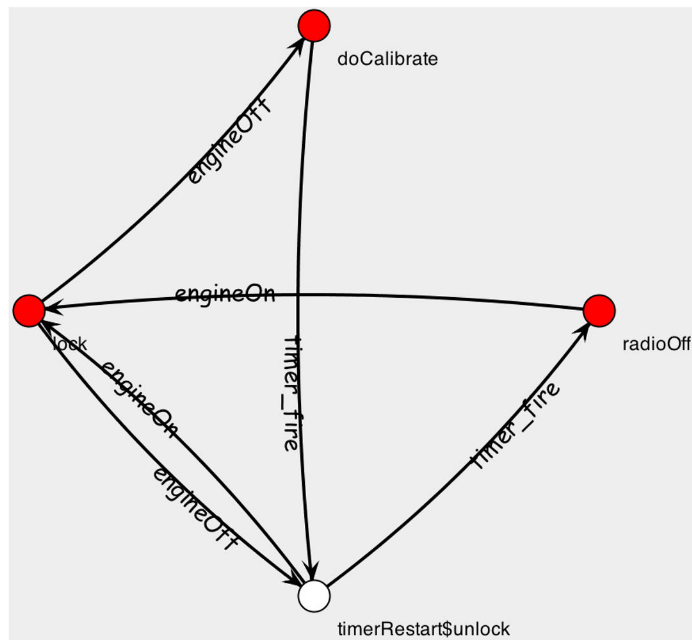
This algorithm is expected to be effective when the number of states in one of the orthogonal FSMs is greater than the combined number of states in all others. For example, there are 4 versus 2 states in the two FSMs shown in Fig. 5, respectively. Hence, every row of the 1HTT whose <event,target state> pairs belong to the FSM shown in Fig. 5b is expected to have more 1's than rows of whose <event,target state> pairs belong to the FSM shown in Fig. 5a.

4.2 Using imperfect interleaving testing

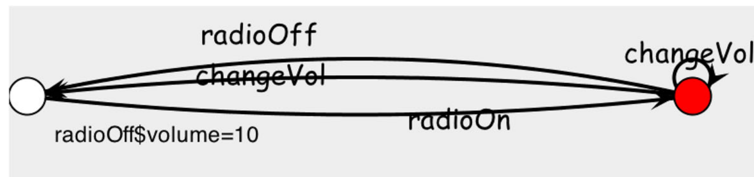
The 1HTT decomposition condition and resulting algorithms assumes the availability of perfect interleaving testing data. When such testing does exist, then the rectangles of the 1HTT table are expected to be incomplete. A workaround is to apply the following changes to relax the methodology:

1. The 1-HTT table entries in source-state columns will be one of 1, 0, or *empty*, as follows:
    a. An entry for a transition $s_1 \rightarrow_{\text{evt}} s_2$, i.e., the cell whose source-state column is s_1 and row has event *evt* and target state s_2, will contain a 1 (as was the case so far).
    b. A cell whose source-state column is s_1, and its row contains event *evt* and target state s_3, will contain a 0

Fig. 5 Two concurrent FSMs induces by the 1HTT transformation depicted in Fig. 4. **a** First FSM of two concurrent FSMs induces by the 1HTT transformation depicted in Fig. 4. **b** Second FSM of two concurrent FSMs induces by the 1HTT transformation depicted in Fig. 4. The state with no name corresponds to a state shown in Fig. 1 with no output actions



a First FSM of two concurrent FSM's induces by the 1HTT transformation depicted in Figure 4.



b Second FSM of two concurrent FSM's induces by the 1HTT transformation depicted in Figure 4. The state with no name corresponds to a state of Figure 1 with no output actions.

if there exists a cell in the same source-state column (s_1) that contains a 1, and whose row contains event *evt* and target state $s_2, s_2 \neq s_3$. Such entry means that the test suite contains a test that injected event *evt* when the primitive FSM was in state s_1 yet that test lead the FSM to state s_2, not s_3.

c. All other cells (i.e., cells whose source-state columns have not been populated in (a) or (b)) remain unpopulated, i.e., *empty*.

2. Refine the 1HTT decomposition requirement, so that rather than requiring the two rectangles contain 1's in *all* entries, require that: (i) they contain either 1's or be *empty*, and (ii) $P\%$ of each rectangle must contain 1's. We call this decomposition *1HTT-decomposition*($P\%$)

All above-mentioned algorithms can easily be redefined to find a valid 1HTT-decomposition($P\%$) with the highest P using a binary search strategy. Once a valid 1HTT-decomposition($P\%$) is discovered, the test suite should be augmented to test transitions that correspond with empty cells in either rectangle. If those empty cells all become "1" after this additional testing, then a valid 1HTT-decomposition has been discovered.

5 Using genetic programming to discover non-binary variables, loop conditions, and actions

After decomposing a primitive FSM into a set of orthogonal, primitive FSMs, each FSM is primitive because it contains none of the following UML features:

- Local variables
- Assignments to non-binary variables (local or output)
- Internal actions (code snippets associated with variable assignments)
- Transition guards, i.e., transitions are triggered by events without any restricting condition.

In this section, we introduce a genetic programming-based machine learning technique for discovering intra-FSM code details (variables, actions, and transition guards) associated with two types of loops, thereby eliminating abovementioned non-determinism. The two types are:

1. *Counting-loops* A counting-loop is controlled by a pair of transitions whose guards are in the form of $cVar < M$, and $cVar \geq M$, as depicted in Fig. 6a. *cVar* is a local (integer) variable acting as a counter, and *M* is the loop limit. Counting-loops are akin to simple *for*-loops in a textual programming language such as Java or C. Clearly, once the location of a counting-loop within an FSM is known, the machine learning objectives are: the increment value *INCR* and the loop limit M. Note that the purpose of the assignment $cVar = M \ mod \ INCR$ in Fig. 6a is to allow the machine learning algorithm distinguish between all solutions where $M \ mod \ INCR = 0$, such as distinguishing between $M = 3$, $INCR = 1$ and $M = 7$, $INCR = 2$.
2. *While-loops* A while-loop is controlled by a similar pair of transitions whose guards are in the form of $nbVar < M$ and $nbVar \geq M$. The structure of a while-loop is depicted in Fig. 6b. A while-loop differs from a counting-loop in the following ways:

   a. The loop control variable *nbVar* represents is a non-binary *output* variable rather than a local counter variable.
   b. Unlike the simple incremental update of *cVar*, *nbVar* is assigned from a machine learnable function (e.g., a polynomial function of one or more non-binary outputs). This assignment models the fact that UML state machines use an underlying textual action language, such as UAL [YMP] or Java [SR], that enable a mix of conventional textual code within a state-based model.

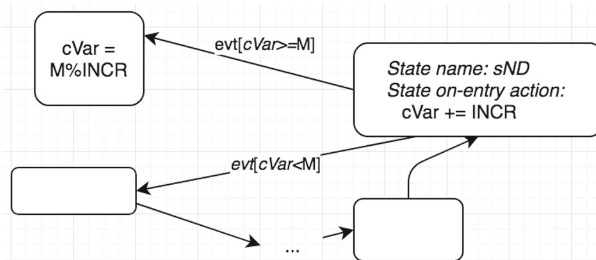
Counting-loops are simpler than while-loops but depend on hidden, local variables, whereas while-loops depend on output variables whose values are observable for subsequent machine learning.

The remainder of this section is devoted to:

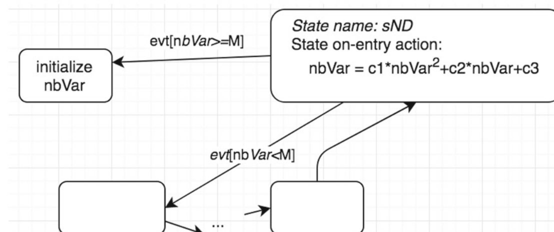
1. Discovering the location of counting and while-loops within a given FSM.
2. Using GP to reverse engineer counting and while-loops.

5.1 Discovering the location of counting and while-loops

Our machine learning technique for reverse engineering such loops within an FSM F is preceded by the following steps.



a The structure of a counting-loop. The variable *cVar* is local, i.e., not present in test results



b The structure of a while-loop. The variable nbVar is a non-binary output variable.

Fig. 6 Counting- and while-loops. **a** The structure of a counting-loop. The variable *cVar* is local, i.e., not present in test results. **b** The structure of a while-loop. The variable nbVar is a non-binary output variable

1. Identify the location of loops to be learned within F, as follows:

   a. Search F for a state (denoted *sND*) that has two outgoing transitions (called the *ND-transition pair*) triggered by the same event. For example, state *lock* shown in Fig. 5a has two outgoing transitions that are triggered by the event *engineOff*; likewise, the *empty-name* state shown in Fig. 5b has two outgoing transitions that are triggered by the event *changeVol.*
   b. Find the shortest simple cycle from *sND* to itself. For example, the simple cycle *lock* $\rightarrow_{\text{engineOff}}$ *timerRestart$unlock* $\rightarrow_{\text{engineOn}}$ *lock* in Fig. 5a, or the simple cycle *changeVol* $\rightarrow_{\text{changeVol}}$ *changeVol* in Fig. 5b.

2. Create and apply a white-box[3] test that traverses the abovementioned simple cycle (from *sND* to itself); the test should be sufficiently long to cater for the largest number times F could possibly repeat the cycle before breaking out of the loop. Since this test is merely a repeated cycle, it can be written manually or by a simple script. We denote this test as *WBTS(sND).*

[3] The test is actually black box in terms of the original SUT, because the SUT's internal structure or behavior is not assumed to be known a priori. However, after deducing its constituent internal FSMs such a test is white-box with respect to each such FSM.

After performing steps 1–3, while-loop and counting-loops within an FSM F have been identified. In both cases, the loop is characterized by its *sND* state and the corresponding ND-transition-pair.

5.2 Using genetic programming to reverse engineer counting- and while-loops

The initial GP population consists of individual FSMs, each being a primitive FSM whose basic structure (states and transitions) is augmented with genes pertaining to machine learning objective entities: variables, actions, and transition guards.

More specifically, an FSM with a counting-loop has the following machine learning objective genes, shown in Fig. 6a:

- The loop control constant, M.
- The counting-increment delta INCR.

An FSM with a while-loop has the following machine learning objective genes, shown in Fig. 6b:

- The loop control constant, M.
- Coefficients of the polynomial whose value is assigned to *nbVar*.

Consider the generic GP algorithm presented in Sect. 2. Given the abovementioned genes, the following settings provide specific details required for a concrete GP algorithm implementation:

- *Random generation of an individual FSM* Create random instances of individual genes where each gene has an associated range, such as a loop guard M value being anywhere between 0 and 1000.
- *Fitness criterion for an individual FSM F*
  - Execute the test suite *WBTS(sND)* on F and on the SUT; for each *test* in *WBTS(sND)* let *cnt*(*test*) be the number of time stamps in which the two test-responses (the SUT's repose and F's response) differ.
  - The sum of all *cnt*(*test*) values is a decreasing fitness criterion, i.e., a perfect fit is manifested by a sum of 0.
- *Crossover* Given two individual FSMs, F_1 and F_2, a new individual FSMs F_3 is created by cloning F_1 and substituting some of its genes with corresponding genes from F_2, such as substituting a loop counter M in $F_1$with the value of the same loop counter in $F_{2,}$or substituting a polynomial coefficient in F_1 with the corresponding coefficient in $F_{2.}$
- *Mutation* Given an individual FSM F_1, randomly selected genes (e.g., counting-increment delta INCR, or while-loop polynomial coefficients) are randomly mutated by replacing them with random values using the same procedure that was used when an individual FSM is created.

6 Conclusion

We described a technique for reverse engineering a concurrent UML statechart using black box testing. The technique consists of two primary phases: an algorithmic phase for discovering the internal composition of its constituent, concurrent, FSMs, and a machine learning phase for learning the parameters of internal counting-loops and while-loops.

The novelty of the suggested technique is threefold:

1. A reverse engineering technique for concurrent UML statecharts.
2. A technique for decomposing an FSM into concurrent/orthogonal FSMs.
3. A machine learning technique for discovering inner loops, actions, and local variables within UML state machines.

Additional research is needed to investigate machine learning of more complex UML actions within UML statecharts.

References

1. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106
2. Amalfitano D, Fasolino AR, Tramontana P (2008) Proceedings of the 15th working conference on reverse engineering, Antwerp, Belgium, Oct 15–18, 2008
3. Brutscheck M (2009) Systematic analysis of unknown integrated circuits. Ph.D. dissertation, Dublin Institute of Technology
4. Brutscheck M, Franke M, Schwartzbacher A, Becker S (2008) Investigation and implementation of test vectors for efficient IC analysis. In: Signals and systems conference. Becker
5. Brutscheck M, Schmidt B, Franke M, Schwartzbacher A, Becker S (2010) Identification of deterministic sequential finite state machines in unknown cmos ICs. In: International solid state circuits conference. Becker
6. Drusinsky-Yoresh D (1991) A state assignment for single-block implementation of statecharts. IEEE Trans Comput Aided Des Integr Circuits Syst 10(12):1569–1575
7. Genetic programming. http://geneticprogramming.com/
8. Koza J, A genetic programming tutorial. http://geneticprogramming.com/tutorial/
9. Kumar A (2008) A novel technique to extract Statechart representations of FSMs. Master thesis, Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur
10. Yang M, Michaelson GJ, Pooley RJ (2008) Formal action semantics for a UML action language. J Univers Comput Sci 14(21):3608–3624

11. Genetic programming. https://en.wikipedia.org/wiki/Genetic_programming
12. Drusinsky D (2006) Modeling and verification using UML statecharts. Elsevier, Amsterdam ISBN: 978-0-7506-7949-7
13. Smith J, Oler K, Miller C, Manz D (2017) Reverse engineering integrated circuits using finite state machine analysis. In: 50th Hawaii international conference on system sciences, pp 2906–2914
14. Spinke V (2013) An object-oriented implementation of concurrent and hierarchical state machines. Inf Softw Technol 55:1726–1740