2012-02

Runtime monitoring and verification of systems with hidden information

Drusinsky, Doron

Monterey, California. Naval Postgraduate School, Department of Computer Science

NPS-CS-12-001

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

RUNTIME MONITORING AND VERIFICATION OF SYSTEMS WITH HIDDEN INFORMATION

by

Doron Drusinsky

February 2012

Approved for public release; distribution is unlimited

Prepared for: Defense Threat Reduction Agency (DTRA) -
8725 John J Kingman Rd., Stop 6201 (RD-BAT), Fort Belvoir
VA 22060-6201

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver
President

Leonard A. Ferrari
Executive Vice President and
Provost

The report entitled *"Runtime Monitoring and Verification of Systems with Hidden Information"* was prepared for and funded by the Defense Threat Reduction Agency (DTRA).

Further distribution of all or part of this report is authorized.

This report was prepared by:

Doron Drusinsky
Associate Professor
Department of Computer Science

Reviewed by:

Peter Denning
Chairman
Department of Computer Science

Released by:

Douglas Fouts
Interim Vice President and Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
February 2012	Technical Report	8 August 2011 – 7 August. 2012

4. TITLE AND SUBTITLE
Runtime Monitoring and Verification of Systems with Hidden

5a. CONTRACT NUMBER
11-2338M

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Doron Drusinsky

5d. PROJECT NUMBER
R6DA1

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
1411 Cunningham Road
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-CS-12-001

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Threat Reduction Agency, 8725 John J Kingman Rd., Stop 6201, Fort Belvoir VA 22060-6201

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT

This paper describes a technique for Run-time Monitoring (RM) and Runtime Verification (RV) of systems with invisible events and data artifacts. Our approach combines well-known Hidden Markov Model (HMM) techniques for learning and subsequent identification of hidden artifacts, with run-time monitoring of probabilistic formal specifications. The proposed approach entails a process in which the end-user first develops and validates deterministic formal specification assertions, s/he then identifies hidden artifacts in those assertions. Those artifacts induce the state set of the identifying HMM. HMM parameters are learned using standard frequency analysis techniques. In the verification or monitoring phase, the system emits visible events and data symbols, used by the HMM to deduce invisible events and data symbols, and sequences thereof; both types of symbols are then used by a probabilistic formal specification assertion to monitor or verify the system.

15. SUBJECT TERMS
Runtime verification, Hidden data, Hidden Markov models, Formal specifications

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					Doron Drusinsky
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	UU	27	831 656 2168

THIS PAGE INTENTIONALLY LEFT BLANK

1. INTRODUCTION

A Hidden Markov Model (HMM) can be considered a state machine in which state transitions and state outputs, or observations, are probabilistic. HMM's are used to learn and classify sequences of observables. HMM technology has been used successfully in a diverse set of applications, such as speech recognition [Da, Pi], Gene prediction [Rä], and Cryptanalysis [Si].

Because of the probabilistic nature of the underlying process being observed by HMM's, they are not used often to recognize long-periodic sequences. Rather, they are mostly used as discriminators, to determine whether one HMM is better than another. For example, an HMM-based speech recognition system may have each HMM represent a word, with run time voice recognition choosing the HMM that best fits the incoming sequence of speech features.

This is in contrast with Deterministic Finite Automata (DFA) [HWU], Finite State Machines (FSM's) [KJ], or Harel-Statecharts [Ha, D1, D2], which are often used to identify and classify individual sequences. Stated differently, because HMM's identify individual sequences of external observables with a relatively low probability, it is usually not perceived as convincing evidence of the occurrence of a particular sequence.

Run-time Verification (RV) of formal specification assertions (RV), also known as Run-time Execution Monitoring (REM), is a class of methods for monitoring the sequencing and temporal behavior of an underlying application and comparing it to the correct behavior as specified by a formal specification.

Some published RV tools and techniques are: the TemporalRover/DBRover [D3], PaX [HR] and RT-Mac [SLS], all of which use extensions and variants of Propositional Linear-time Temporal Logic (PLTL) as the specification language of choice, and the StateRover [SR] that uses deterministic and non-deterministic statechart diagrams as its specification language. In [D2], Drusinsky describes the application of RV using statechart assertions to the verification of DoD and NASA applications, and to those of the Brazilian Space agency

Execution-based Model Checking (EMC) is a combination of RV and Automatic Test Generation (ATG). With EMC, a large volume of automatically generated tests are used to exercise the program or System Under Test (SUT), using RV on the other end to check the SUT's conformance to the formal specification. Some ATG tools that, when combined with RV tools, create an EMC technique are the StateRover's white-box automatic test-generator [SR] and NASA's Java Path Finder (JPF) [HP].

Runtime Monitoring (RM) is a technique for monitoring system behavior with respect to formally specified properties, but for purposes other than verification, such as performance or statistical analysis. In the remainder of this paper we refer to RV as the union of RV and RM.

In [DMS], the authors present a visual tradeoff space, called the Formal Validation and Verification (FV&V) tradeoff cuboid, which qualitatively compares three categories of FV&V techniques: Model Checking (MC), Theorem Proving (TP), and RV

combined with automatic Test Generation (ATG). The tradeoff space compares the cost and test-space coverage associated with these three categories of techniques. This tradeoff space highlights the wide spectrum of systems for which RV has a favorable cost-performance ratio.

In this paper, we use HMM's to identify hidden events and sequences thereof, for the purpose of subsequent RV. We will not be using the (rather small) probability of an observable sequence, but rather the probability of a hidden state being reached *given* a sequence of observables. Hence, the technique identifies hidden events with a relatively high probability.

This paper describes an extended RV technique suitable for systems in which not all artifacts are necessarily observable. The technique is a novel combination of Hidden Markov Models (HMM's) with probabilistic RV of formal specification assertions. Throughout the paper, we will be using the Statechart assertion formal specification language of [D1, D2]. We will show a probabilistic variant of this formalism suitable for RV of systems with hidden inputs.

Our proposed technique is suitable for the verification of complex systems in which visible data does not necessarily contain all the information required for monitoring the systems health or for verifying its behavior, as in the case of telemetry files of space missions. It is also suitable for monitoring the behavior of systems that are not fully accessible, such as a nuclear facility or distant unmanned vehicle, and for forensic applications, such behavioral analysis of a post-accident aircraft or automotive system using black-box information.

The rest of the paper is organized as follows. Section 2 provides an overview of RV using UML-based statechart assertions. Section 3 provides an overview of HMM's and HMM related algorithms. Section 4 describes our proposed *extended-RV* architecture and process that uses a combination of hidden and visible data, using an HMM connected to a special formal specifications monitor. Sections 5, 6 and 8 provide specific details of the two key components of this process: section 5 describes the HMM component, section 6 describes the operation of the formal specifications monitor, and section 8 describes three techniques for computing the probability distribution used by that monitor. While sections 5 and 6 focus on formal specification assertions with hidden data - manifested as UML statechart conditions, section 7 extends the technique to formal specification assertions with hidden events. Section 9 extends the technique to assertions with hidden continuous data. Finally, section 10 compares our suggested extended-RV architecture with two alternative architectures.

2. RV OF (DETERMINISTIC) FORMAL SPECIFICATION ASSERTIONS – AN OVERVIEW

Runtime Verification (RV) is a light-weight formal verification technique in which the runtime execution of a system is monitored and compared to an executable version of the system's formal specification. In other words, RV behaves as an automated observer of the program's behavior and compares that behavior with the expected behavior per the formal specification.

The following formal specification example will be used throughout the rest of the paper.

Consider the following Traffic Light Controller (TLC) requirement R1: *whenever vehicle speed in the Main direction is greater than 42km/h for more than 2 consecutive minutes while lights in the Main direction are green, then lights in that direction should turn red within 30 seconds afterwards.*

Figure 1 depicts a statechart-assertion for R1. As described in [D1,D2], a statechart-assertion is a UML state-machine augmented with a Java action language and a built in Boolean flag named *bSuccess*, whose value indicates whether the assertion is succeeding (e.g., the input scenario conforms to R1) or failing (e.g., the input requirement violates R1).

The statechart-assertion of Fig. 1 starts-up in the top-level *Init* state. When lights turn green (*lightsTurnedGreen* event) it transitions to the *Init* state of the *OnGoing* sub-state of the *Green* super-state, where it polls until the *Speed* variable becomes HIGH (using a 1Hz clock tick event named *clockTick*); the assertion then transitions to the *SpeedHigh* state. It then polls for *Speed* to become non-HIGH within 2 minutes. If *Speed* value is or becomes not HIGH then the assertion waits in *Green.OnGoing.Init* until *Speed* turns HIGH again. If two minutes have elapsed then the assertion waits for an additional 30 seconds, during which it checks whether lights have turned red as required. If so, then the process restarts in the top-level *Init* state. Otherwise, R1 has been violated and the assertion transition's to the Error state where it sets the *bSuccess* flag to false. This flag indicates that the assertion has failed.

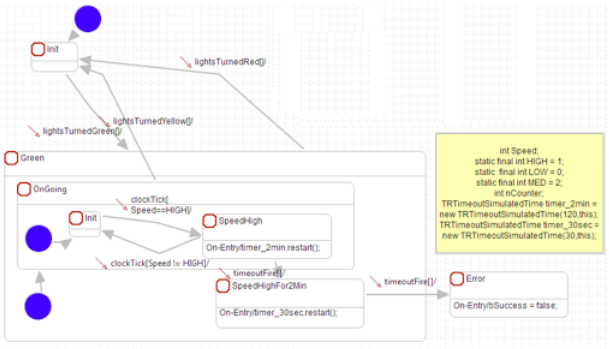


Figure 1. A statechart-assertion for requirement R1.

Fig. 2 illustrates the conventional RV architecture: an executable formal specification assertion observes inputs and outputs of the SUT (the TLC in our example), and compares those sequences to the expected behavior; whenever that actual behavior violates the requirement the specification announces a failure.

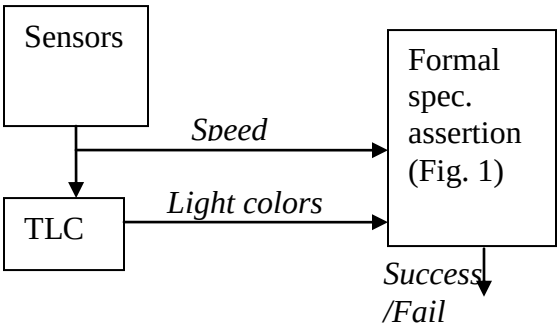
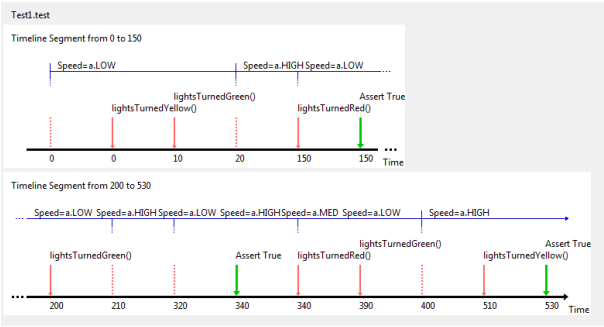


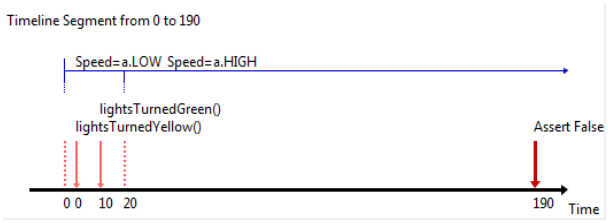
Figure 2. The RV architecture for the TLC and requirement R1.

Fig. 3 depicts two timeline diagrams of validation tests for the assertion of Fig. 1, i.e., tests that assure the statechart-assertion correctly implements the natural language requirement R1. Fig. 3a depicts a test scenario that conforms to R1 – checking that the assertion succeeds for this scenario, as expected. Fig. 3b depicts a test scenario that violates R1 – checking that the assertion fails for this scenario, as expected.

Validation testing is an important step in the process because the formal-specification assertion is to be trusted to represent requirement R1 in the subsequent automated verification phase, discussed below[1].



a. Timeline diagram for validation test Test1.



b. Timeline diagram for validation test Test2. R1 is violated by this scenario (as indicated by the JUNit *Assert False* arrow) because Speed is HIGH for more than two minutes while lights are green, yet lights didn't turn red as required.

Figure 3. Timeline diagrams for two validation tests for the statechart-assertion of Fig. 1.

Verification is performed by comparing a trace of the system (e.g., as captured by a log file) to the behavior of the assertion set. The StateRover tool does so using a two

[1] Further details about validation testing is available in [D2].

step process. First, the log file is converted into an equivalent JUnit test [JU], and the assertion is code-generated into an equivalent Java class (details about this code generator are available in [D1]). Next comes the RV step, the JUnit test is executed, thereby checking that the log-file trace conforms to the requirement as manifested by the assertion.

The extended-RV technique suggested in this paper uses the same process for the development and validation of assertions, i.e., assertions are developed as deterministic assertions. However, rather than performing deterministic RV by the virtue of using an assertion code generator that generates a deterministic implementation, our technique performs probabilistic RV using a special assertion code generator that generates a probabilistic, weighted implementation. Specific details are provided in section 6.

3. HIDDEN MARKOV MODELS

A (discrete) hidden Markov model (HMM) is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved, or hidden states. while in a regular Markov model, the state is directly visible to the observer, in a hidden Markov model the state is not directly visible, while the output, dependent on the state, is visible.

The parameters of a simple HMM are [Ra]:

- N, the number of states in the model. Individual states are denoted $S = \{s_1, s_2,...s_N\}$, and the state at time t as q_t.
- M, the number of distinct observation symbols. Individual states are denoted $V = \{v_1, v_2,...v_M\}$.
- The state transition probability distribution $A = \{a_{ij}\}$ where $a_{ij} = P[q_{t+1} = s_j | q_t = s_i]$, $1 \le i,j \le N$. Clearly, $\forall i$, $1 \le i \le N$, $\sum_{1 \le j \le N} a_{ij} = 1$.
- The observation symbol probability distribution in state j, $B=\{b_j(k)\}$, where $b_j(k) = P[v_k \text{ at } t \mid q_t = s_j]$, $1 \le j \le N$, $1 \le k \le M$.
- The initial state distribution $\pi = \{\pi_i\}$, where $\pi_i = P[q_1 = s_i]$, $1 \le i \le N$.

Rabiner [Ra] describes the following three primary problems associated with HMM's:

1. Given the observation sequence $O = O_1O_2...O_T$, and an HMM model $\lambda = (A,B, \pi)$, how do we efficiently compute $P(O|\lambda)$?
2. Given the observation sequence $O = O_1O_2...O_T$, and an HMM model $\lambda = (A,B, \pi)$, how do we choose an optimal state sequence $Q = q_1 q_2...q_T$?
3. How do we calculate the model parameters $\lambda = (A,B, \pi)$ to maximize $P(O|\lambda)$?

The most well known algorithms used to solve these problems are:

1. The *forward algorithm*, for calculating the *forward variable* $\alpha_t(i) = P(O_1O_2...O_t, q_t = s_i \mid \lambda)$. The forward algorithm is a dynamic programming algorithm based on the recurrence:

   $\alpha_{t+1}(j) = [\sum_{i=1..N} \alpha_t(i) a_{ij}] \, b_j(O_{t+1})$, $1 \le t \le T-1$, $1 \le j \le N$,

   with the initialization:

$\alpha_1(j) = \pi_j\, b_j(O_1)$.

Note that $P(O_1O_2...O_t|\lambda)=\sum_{i=1..N}\alpha_t(i)$.

$\alpha`$ is the normalized version of α:

$\alpha`_t(j)=P(q_t=s_i|O_1O_2...O_t, \lambda)$, calculated recursively as: $\alpha`_{t+1}(j)=\alpha_{t+1}(j)/P(O_1O_2...O_t|\lambda)$.

2. The *backward algorithm*, for calculating the *backward variable* $\beta_t(i) = P(O_{t+1}O_{t+2}...O_T |q_t = s_i, \lambda)$. The algorithm is a dynamic programming algorithm based on the recurrence:

$\beta_t(i) = \sum_{j=1..N} a_{ij}\ b_j(O_{t+1})\ \beta_{t+1}(j)$, for

$t = T-1, T-2, ..., 1$, and $1 \le i \le N$,

with the initialization:

$\beta_T(i) = 1$, for $1 \le i \le N$.

3. The *forward-backward algorithm*, for calculating the *forward-backward variable*

$\gamma_t(i) = P(q_t = s_i \mid O_1...O_T, \lambda)$.

γ is also:

$\gamma_t(i)=(\alpha_t(i)\ \beta_t(i))/ P(O_1O_2...O_T|\lambda)$

γ can also be expressed as:

$\gamma_t(i)=\sum_{1\le j\le N}\xi_t(i,j)$ where: $\xi_t(i,j) = (\alpha_t(i)\ a_{ij}\ b_j(O_{t+1})\ \beta_{t+1}(j))/P(O_1O_2...O_T|\lambda)$.

4. The Viterbi algorithm, for calculating the best state sequence that explains an observation sequence, $\delta_T(O_1O_2...O_T \mid \lambda)$. The algorithm defines:

$\delta_t(i)=\max[q_1,q_2,... q_{t-1}]\ P(q_1,q_2,...q_t=s_i, O_1O_2...O_t \mid \lambda)$,

and uses the following recursive formula:

$\delta_t(j) = \max_{1\le i\le N}[\delta_{t-1}(i)\ a_{ij}]\ b_j(O_t)$

along with the following formula, used to recover the actual most probable state sequence:

$\psi_t(j) = \text{argmax}_{1\le i\le N}[\delta_{t-1}(i)\ a_{ij}]$, where $\psi_1(j)=0$;

The Viterbi algorithm is essentially the forward algorithm with a recurrence in which a *max* operator is used instead of the sum. The probability of best state sequence $\delta_T(O_1O_2...O_T \mid \lambda)$ is then the maximal $\delta_T(i)$, $1 \le i \le N$, and $q_T = \text{argmax}_i\, \delta_T(i)$, $1 \le i \le N$.

The most probable state sequence $q_1,q_2,...q_T$ is calculated in a backward manner, using $q_{t-1} = \psi_t(q_t)$.

4. RV OF SYSTEMS WITH HIDDEN STATES

Suppose our TLC is being monitored or verified. Suppose also that, as assumed by the statechart-assertion of Figure 1, it emits 3 color change events: (*lightTurnedRed*, *lightTurnedGreen*, and *lightTurnedYellow*), but it not have a *Speed* input or output. Instead, the TLC has input sensors that measure the frequency of cars going through the junction in a particular direction (e.g., in the Main direction). In other words, frequency is an *observable* whereas speed is a *hidden* artifact.

To enable RV of the TLC with respect to R1 and its corresponding statechart-assertion, we modify the architecture of Fig. 2 as depicted in Fig. 4. This architecture differs from the conventional RV architecture of Fig.1 in three main aspects:

1. It contains a Hidden Markov Model (HMM), used to decode the probability of occurrence of sequences of hidden *Speed* states given sequences of the *frequency* observable. This HMM provides a plurality of weighted *Speed* inputs to the statechart-assertion, instead of a unique un-weighted *Speed* input used in Fig. 1. Detailed of the HMM are discussed below.
2. It uses a special code generator that generates a probabilistic implementation for the statechart assertion(s), one that operates on the weighted inputs from the HMM.
3. It evaluates the assertion using a success score in the range [0,1].

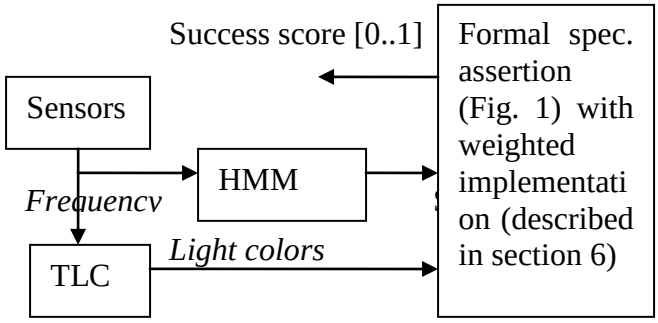


Figure 4. The RV architecture for the TLC and requirement R1 when the Speed input is hidden.

In our example, visible frequency measurement pertains to a sensor under the Main Street that measures the frequency of cars driving over the sensor. The sensor produces symbols, $f_1, f_2, \ldots, f_5$ where f_d represents a measured frequency in the range of $(d-1, d]$ cars per second, for all d [2]. Loosely speaking, using a 4 meter per car metric (including car to car spacing), $Speed = 14.4 * f$ *km/h*. We categorize 3 ranges of speeds for cars going over the sensor, as follows: (i) HIGH: cars speed is above 40 km/h, (ii) LOW: car speeds below 15 km/h, and (iii) MED: for all other possibilities.

While we could use the above-mentioned stationary process do deduce the hidden *Speed* value-range from the visible frequency measurement, it does not account for dynamic aspect of the system. First, it does not account for the fact that distances between cars change with car-speed, rendering the 4 meters per car estimate inaccurate. Also, it is expected for *Speed* to seldom change from HIGH to LOW directly.

Consequently, we use an HMM to model this random process. Figure 5 depicts an HMM for the TLC example. Its parameters are:

- The state set *Q* consists of three states that correspond with *Speed*, namely, HIGH, MED, and LOW, also denoted as states 0, 1, and 2, respectively. Note that it is not a coincident that the HMM states capture the hidden variable in the assertion of Fig. 1; we will discuss this relationship in section 5.
- An observable O, which takes on one of the f_d symbols discussed earlier.

[2] We assume that frequencies above 5 cars/sec are measured as 5 cars/sec.

- Transition probabilities are indicated along the edges of Fig. 5.

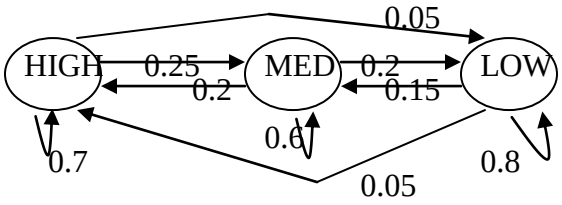


Figure 5. *Speed* random variable HMM states and transition.

- $b_s(O)$, the probability of an observable O being observed in state s, is listed in Table 1.

O\state	HIGH	MED	LOW
f_1	0.02	0.18	0.63
f_2	0.22	0.53	0.26
f_3	0.47	0.17	0.11
f_4	0.2	0.11	0
f_5	0.1	0.01	0

Table 1. Probability of observation O in TLC state s

- The initial state distribution is [0.3, 0.5, 0.2] for HIGH, MED, and LOW, respectively.

RV now proceeds according to the process illustrated in Fig. 4, as follows. Sampled frequency values are periodically fed into the HMM, which then executes a probability estimation algorithm, such as the forward-algorithm for the current iteration (section 8 discusses three probability estimation techniques). These probability values represent probabilities of the HMM being in states HIGH, MED, and LOW, respectively. This vector of symbols and corresponding probabilities is passed to the assertion's implementation code, which executes a weighted version of a state-machine state change, detailed in section 6. Finally, as discussed in section 6, the assertion announces the probability it detected a requirement violation.

A more realistic HMM for deducing car speed is one in which the observable frequency is a continuous random variable (called *Frequency*), e.g., with a normal distribution whose probability density function (PDF) is $f_O(o) \sim N(\mu,\sigma^2)$, rather than a Categorical distribution (as the case for TLC-example, whose distribution is listed in Table 2). Using the TLC example again, the probability estimation algorithm of choice (elaborated in section 8) will use $f_{Frequency}(frequency, j)$, the *Frequency* PDF in state j, instead of $b_j(frequency)$.

State :	IGH	ED	OW
μ *(cars/sec)*	3.125	2	0.55
σ *(cars/sec)*	0.35	1	0.50

Table 2. Normal distribution parameters of observation O in TLC states.

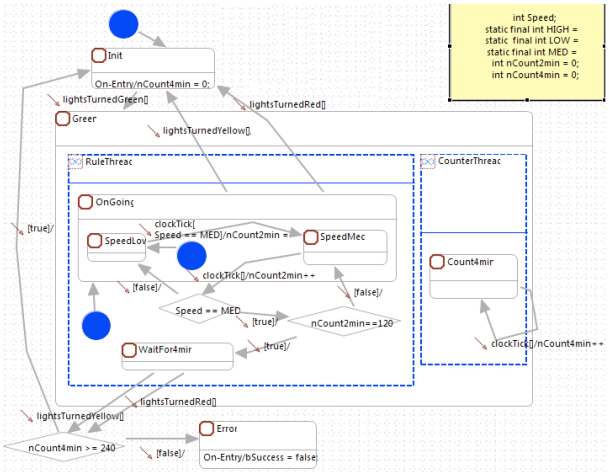
5. FROM ASSERTIONS TO HMM PARAMETER ESTIMATION

HMM parameter estimation, i.e., estimating the transition probability and probability of state observations, is a difficult problem. In particular, it is difficult to estimate the number of HMM states, the extreme cases being using one state (i.e., reducing the HMM to a stationary process) or *n* states, *n* being the length of the observation sequence.

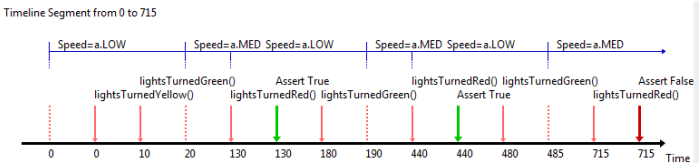
In our case however, HMM states are known; they are directly related to the hidden artifacts in the assertions. For example, in the TLC case, the three hidden symbols pertain to Speed values HIGH, MED, LOW, which are derived from Fig. 1 and its requirement R1, as well as from an assertion for the following requirement:

R2: *if vehicle speed in the Main direction is between 15 and 30 km/h for more than 2 consecutive minutes while lights in the Main direction are green, then lights should remain green for a total of 4 minutes.*

Fig. 6a depicts a statechart assertion for requirement R2, and Fig. 6b depicts a timeline diagram for a validation test for this assertion.



a. Statechart assertion and validation test for requirement R2



b. A timeline diagram of a validation test for the statechart-assertion of (a)

Figure 6. Statechart assertion and validation test for requirement R2.

Our use-case for HMM's is simpler than usual in one additional aspect: calculating transition and observable probabilities. Because HMM states relate to real world artifacts (e.g., car speed values), we can conduct learning-phase experiments which measure relative frequencies, such as one in which all speeds and sensor frequencies are measured on a 1-second period basis; all HMM probabilities follow trivially. This is the

case whether observables are distributed using a Categorical distribution or some continuous distribution.

Consequently, we can deduce the workflow for developing the components of the architecture of Fig. 4, as depicted in Fig. 7.

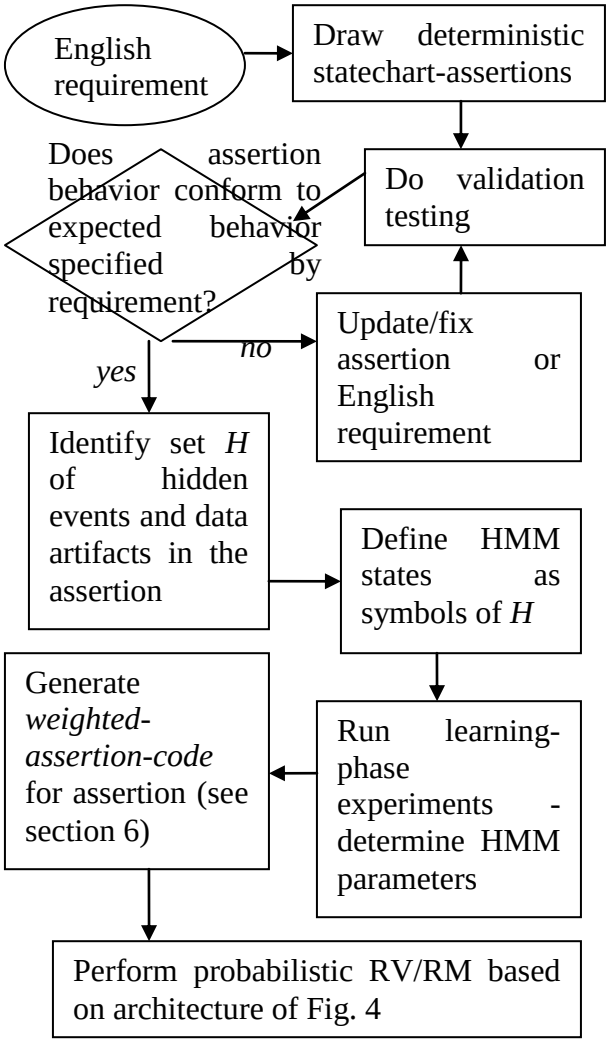


Figure 7. Workflow for developing the RV components of Fig. 4.

6. RV OF ASSERTIONS WITH PROBABILISTIC INPUTS

Using the architecture of Fig. 4, the formal specification assertion module observes sequences that consist of visible as well as hidden artifacts; in Fig. 1 for example, *lightsTurnedRed*, *lightsTurnedYellow*, *lightsTurnedGreen*, *timeoutFire*, and *clockTick* event are visible, while *Speed* is hidden. Hidden artifacts have an associated probability distribution which we call the *probability-of-occurrence distribution (POD)*, such as *POD*-1: *Speed=HIGH, MED, LOW at time 5 occurs with probability 0.72, 0.2, 0.08, respectively*. Section 8 describes three techniques, called α`, γ, and δ``, for computing the cycle-by-cycle POD, based on α, γ, and δ, respectively. We consider a visible artifact to have a probability of occurrence of 1.

A weighted/probabilistic implementation of the statechart assertion module of Fig. 4 responds to an input sequence $I = <S_1, P_1>, <S_2, P_2>,.., <S_T, P_T>$, where S_t is a visible or hidden artifact (i.e., event such a *clockTick*, or data artifact, i.e., variable, such as *Speed*, both in Fig. 1), and P_t is the *POD* of S_t.

We use the UML notation for S_t, S_t=*event*$_t$[*condition*$_t$], where *condition*$_t$ is optional; *event*$_t$ and *condition*$_t$ can either or both be visible or hidden.

An assertion's implementation consists of a collection ***C*** of instances, or copies, of the assertion, called *configurations*. Each configuration executes as a standalone assertion and preserves its own present-state. Each configuration *Con* has a probability measure *P*(*Con*), called the *Configuration Probability Measure* (CPM), that measures the probability the assertion is behaving as suggested by *Con*, i.e., that its present-state is *Con*'s present state. Upon startup, ***C*** consists of a single configuration $Con_{default}$ whose present-state, denoted $PS(Con_{default})$, is the assertion's default state (e.g., state *Init* in Fig. 1), and having $P(Con_{default})=1$.

All configurations of ***C*** respond to a pair $<S_t,P_t>$ of *I*, as follows. If $P_t = 1$ then the configuration performs a conventional state machine state change upon input S_i, such as *SpeedHigh* $\rightarrow_{\text{timeoutFire}}$ *SpeedHishFor2Min*, in Fig. 1. Otherwise, either *event*$_t$ or *condition*$_t$ are hidden. In this case the configuration *Con* is replaced with two configurations: *Con1* and *Con2*, whose present-state probabilities are calculated as follows:

- If *event*$_t$ is hidden (as discussed in section 7) then $P(Con1) = P(Con)*P_t$ and $P(Con2) = P(Con)*(1-P_t)$.
- If *condition*$_t$ is hidden, then we calculate $P(condition_t)$, the probability of the condition, as a function of the probabilities of its constituent variables using standard probability. For example, if *condition*$_t$ is *Speed = HIGH || Speed = MED* then $P(condition_t) = P(Speed = HIGH) + P(Speed = MED)$, where each term is taken from the POD at time *t*, such as 0.72 and 0.2 respectively, using *POD*-1.

  We set $P(Con1)=P(Con)P(condition_t)$, and $P(Con2)=P(Con)(1-P(condition_t))$.

Let *PS*(*Con*) denote *Con*'s present-state. *PS*(*Con1*) and *PS*(*Con2*) are determined as follows:

- If $event_t$ is hidden (as discussed in section 7) then *PS*(*Con1*) is the next state determined by the assertion's transition out of *PS*(*Con*), under the assumption that the event fired, and *PS*(*Con2*)= *PS*(*Con*).
- If $condition_t$ is hidden (e.g., *Speed*==HIGH condition in Fig. 1), then *PS*(*Con1*) is calculated assuming $condition_t$=*true* and *PS*(*Con2*) is calculated assuming $condition_t$=*false*,

For the sake of simplicity we disallow assertions in which both $event_t$ and $condition_t$ are hidden.

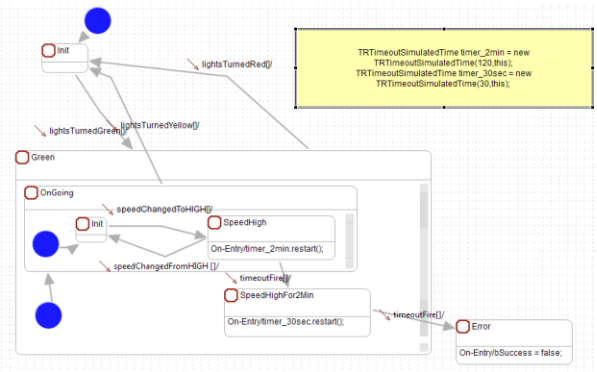
C configurations are routinely (i..e, every cycle *t*) managed as follows. All configurations *Con*` with the same present-state[3] are merged into a single configuration $Con_{merged}$, using the sum of all P(*Con*`) as $P(Con_{merged})$.

The statechart assertion declares a *probability of failure (POF)*, i.e., the probability its corresponding requirement has been violated, on a cycle by cycle basis, being the sum of all *P*(*Con*) for all configurations *Con* such that *PS*(*Con*) is an error state.

Note that statechart assertions typically have error states that are sink states, i.e., states with no outgoing transitions. For such assertions, the POF is monotonically increasing with time.

7. RV OF ASSERTIONS WITH HIDDEN EVENTS

UML statecharts, message sequence diagrams (MSC's), and other formalisms are intrinsically event driven. In fact, the statechart assertions of Figures 1 and 6a are event driven, using events such as *lightsTurnedRed* and the 1Hz *clockTick* event. However, as presented in section 4, HMM symbols are propositional in nature - , manifested as the states of the HMM, such as the *Speed* variable. Consequently, the assertions of Figures 1 and 6a must *poll* the *Speed* variable using the 1Hz *clockTick* event. In contrast, Fig. 8 depicts an *event driven* assertion for requirement R1; it uses hidden events *speedChangedToHIGH* and *speedChangedFromHIGH*.



[3] More accurately, *PS*(*Con*) is an extended state vector, that includes the state variable and the states of all local variables, such as the timer state and the *bSuccess* flag.

Figure 8. An event driven statechart assertion for requirement R1 that uses hidden events

The probability of these two events is induced by the probability of an HMM transition from state *i* to state *j* being traversed at time *t*, i.e., by $\xi_t(i,j)$. Hence, their probabilities are:

1. *P*(event *speedChangedToHIGH* occurring at time t $|O,\lambda) = \sum_{1\le i\le N} \xi_t(i,0)$.
2. *P*(event *speedChangedFromHIGH* occurring at time t $|O,\lambda) = \sum_{1\le j\le N} \xi_t(0,j)$.

8. GENERATING THE PROBABILITY OF OCCURRENCE OF A HIDDEN ARTIFACT

We propose three techniques for estimating the POD at time *t*: the *alpha*, *gamma*, and *delta* methods, as follows.

- The *alpha* method, which uses *N* values of $\alpha`_t(i)=P(q_t=s_i|O_1O_2...O_t, \lambda)$, one per symbol $s_i$, $1\le i\le N$. Note that $\sum_{1\le i\le N} \alpha`_t(i) = 1$.
- The *gamma* method, which uses *N* values of $\gamma_t(i)=P(q_t=s_i|O_1O_2...O_T, \lambda)$, one per symbol s_i, $1\le i\le N$. Note that $\sum_{1\le i\le N} \gamma_t(i) = 1$.
- The *delta* method, which uses *N* values of:

  $\delta_t``(i) = \delta_t`(i)/\sum_{1\le i\le N} \delta_t`(i)$, where

  $\delta_t`(i) = \max[q_1,q_2,... q_{t-1}]P(q_1,q_2,...q_t=s_i| O_1O_2...O_t,\lambda)$, where

  $P(q_1,q_2,...q_t=s_i| O_1O_2...O_t,\lambda)=\delta_t(i)/P(O_1O_2...O_t)$. In other words, $\delta_t``(i)$ is a normalized version of $\delta_t`(i)$, which in turn is the probability of the HMM generating symbol s_i at time *t*, via the most probable state sequence, given the observation.

The gamma method is a backward-forward algorithm; it therefore requires the entire observable sequence $O_1O_2...O_T$ for the evaluation of $\gamma_t(i)$ for $t \le T$. The alpha and delta methods on the other hand, are forward algorithms and therefore do not require future information for the calculation of $\alpha`_t(i)$ and $\delta_t``(i)$. Nevertheless, scaling issues discussed below effectively imply that no matter what method is used, it can only be used verbatim with a limited number of observables. In section 10 we suggest a remedy to this limitation.

When the HMM contains transitions with probability 0, then all three methods might induce sequences of symbols that cannot be physically generated. For example, consider an an HMM with N=3 and $a_{1,2}$=0, and suppose $\gamma_t(1)$=0.3 and $\gamma_{t+1}(2)$=0.2; The assertion then cosiders the sequence s_1, s_2 as possible, having a positive probability of 0.06.

All three methods suffer from inherent scaling problems, because the calculation of $\alpha`_t(i)$, $\gamma(i)$, and $\delta_t``(i)$ generate values that scale down geometrically with *t*. There are published numerical techniques designed to mitigate this problem [Ma]; nevertheless, this constraint limits the length of the sequence of observables $O_1O_2...O_T$, and its corresponding sequence $I = <S_1, P_1>, <S_2, P_2>,.., <S_T, P_T>$ of assertion inputs. Meanwhile, the RV process, in of as itself, is not necessarily limited in duration, and might continue working for intervals longer than *T*.

A straight-forward solution to the scaling problem is to perform RV using a sequence of frames of observables of length *T*, where the probability measurement values computed at time *T* (i.e., $\alpha`_T(i)$, $\gamma_T(i)$, or $\delta_T``(i)$) of a certain frame are used as π_i for the following frame. This approach however, introduces an error or noise every time we reload the frame buffer.

To circumvent this problem, we propose the following smoothing approach in which we use two partially overlapping buffers of observables of length *T*. Buffer B_1 contains observables $O_{n+1}O_{n+2}...O_{n+T}$, while buffer B_2 contains observables $O_{m+1}O_{m+2}...O_{m+T}$, where $m=n+T/2$; in other words $B_1[t]=B_2[t+T/2]$ if $t \leq T/2$ and $B_1[t]=B_2[t-T/2]$ otherwise. This is applied repeatedly for frames n=0,1,2,.., where the roles of B1 and B2 alternate. Now suppose we are using the gamma method; we apply it to each buffer, resulting in $\gamma^1_t(t)$ and $\gamma^2_{t+T/2}(t)$ if $t \leq T/2$, and $\gamma^1_t(t)$ and $\gamma^2_{t-T/2}(t)$ otherwise. Finally use the average of these two γ values as our actual $\gamma_t(t)$ using a weighted average that weighs the $\gamma^j_t(t)$ value that is closer to the center of its buffer more that the one that is farther away from the center of its buffer:

$\gamma_t(t) = (2t\,\gamma^1_t(t) + (T-2t)\,\gamma^2_{t+T/2}(t))/T$, if $t \leq T/2$

$\gamma_t(t) = ((2T-2t)\,\gamma^1_t(t) + (2t-T)\,\gamma^2_{t+T/2}(t))/T$, otherwise.

In future research we will conduct experiments that measure the deviation of $\alpha`_T(i)$, $\gamma_T(i)$, and $\delta_T``(i)$ from their true values when this method is used.

9. RV OF ASSERTIONS WITH HIDDEN CONTINUOUS DATA

While requirement R1 asserts about vehicle speed greater than 42km/h in the Main direction, the matching statechart assertion of Fig. 1 asserts about *Speed* values being one of the symbols (HMM states) HIGH, MED, or LOW; as a consequence, the task of matching HMM states to vehicle speed values becomes the TLC's HMM designer's responsibility, while this is actually a requirement vs. assertion matching issue.

An additional drawback of this approach is that the random variable being asserted about (*Speed*, in the TLC case) typically has a more complex distribution than the simplistic Categorical distribution.

Suppose that TLC *Speed* is not the HMM state, but a random variable associated with the state, one with a continuous distribution such as a normal distribution. The Table 3 example lists the parameters of the *Speed* random variable distribution for the TLC example.

State:	HIGH	MED	LOW
Name of distribution	F_0	F_1	F_2
μ *(km/h)*	40	28	14
σ *(km/h)*	12	15	10

Table 3. Normal distribution parameters of the *Speed* random variable TLC states.

Using this framework, we can now use a variant of the assertion of Fig. 1 that uses transition conditions *Speed*>42 and *Speed*≤42 instead of *Speed*=HIGH and *Speed*!=HIGH, respectively, thus addressing the letter of requirement R1.

Let *Speed(t,i)* denote a random variable (r.v.) representing *Speed* when the HMM is in state *i* at time *t*. We assume its distribution is time independent, and therefore write *Speed(i);* its cumulative distribution-function (CDF) is $F_{Speed(i)}(speed) = P(Speed \leq speed \mid q_t=s_i, \lambda)$. We also make the following counter-intuitive assumption: *Speed(i)* is independent of the observables (sensor *frequency* measurements), given the present state s_i. It is counter intuitive because after-all, vehicle speed seem to depend on those frequencies. Nevertheless, the dependence is totally manifested by the $q_t=s_i$, and given that, *Speed(i)* is independent of the observables.

We now define modified variables α, β, and γ, as expressions rather than literal numbers, as follows:

- $\alpha^{\Diamond}_{Speed,\,t}(speed, i)=P(O_1O_2...O_t, Speed \leq speed, q_t=s_i|\lambda)$.
  Clearly,
  $\alpha^{\Diamond}_{Speed,\,t}(speed, i)=$
  $P(O_1O_2...O_t, q_t = s_i \mid \lambda)\ P(Speed \leq speed \mid O_1O_2...O_t, q_t=s_i, \lambda) =$
  $\alpha_t(i)\ P(Speed \leq speed \mid q_t = s_i, \lambda)$,
  the last equality results from *Speed(i)* being independent of the observables.
  Hence:
  $\alpha^{\Diamond}_{Speed,\,t}(speed, i) = \alpha_t(i)\ F_{Speed(i)}(speed)$.
  The normalized version, $\alpha^{\Diamond`}$ is:
  $\alpha^{\Diamond`}_{Speed,\,t}(speed, i) = \alpha`_t(i)\ F_{Speed(i)}(speed)$.
- $\beta^{\Diamond}_{Speed,t}(speed,i) = P(O_{t+1}O_{t+2}...O_T, Speed(i) \leq speed \mid q_t=s_i, \lambda)$. As in the case of α, $\beta^{\Diamond}_{Speed,t}(speed,i)=\beta_t(i)F_{Speed(i)}(speed)$.
- $\gamma^{\Diamond}_{Speed,t}(i)= P(Speed(i) \leq speed, q_t = s_i \mid O_1...O_T, \lambda) = \alpha_t(i)\ \beta_t(i)\ F_{Speed(i)}(speed) / P(O_1...O_T)$.

Rhe RV process of section 6 is modified as follows. In addition to using the alpha or gamma methods to calculate a Categorical POD for HMM states such as *POD*-1, we calculate $\alpha^{\Diamond}$ or $\gamma^{\Diamond}$, respectively, using an instance value of *speed* (e.g., *speed* = 42) based on the assertion. More specifically, given an RV computation *Con*, the calculation of *P(Con1)* and *P(Con2)* discussed in section 6 is modified as follows:

- If $event_t$ is hidden, the calculation is unchanged, because the probability of a transition being traversed only depends on states and observations, not on the *Speed* variable. In other words, *Speed* only pertains to conditions in the assertion statecharts, not events.
- If $condition_t$ is hidden, as in *Speed*≤42 in the modified assertion of Fig. 1, then we calculate $P(condition_t)$, the probability of the condition, by evaluating the expected value of $\alpha^{\Diamond`}_t(42, i)$ namely,
  $\sum_{1 \leq i \leq N} \alpha`_t(i)\ F_{Speed(i)}(42)$ for the alpha method, or the expected value of $\gamma^{\Diamond}_t(42, i)$ namely, $\sum_{1 \leq i \leq N} \gamma_t(i)\ F_{Speed(i)}(42)$, for the gamma method.

We set $P(Con1)=P(Con)P(condition_t)$, and $P(Con2)=P(Con)$ $(1-P(condition_t))$, as in section 6.

10. A COMPARISON OF RV ARCHITECTURES

We considered the following two architectures for RV of systems with hidden information, in addition to the *weighted-probabilistic assertion architecture* of Fig. 4:

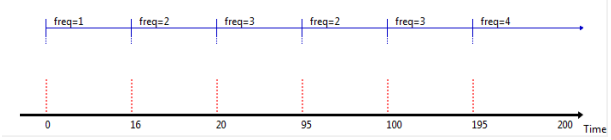
The first alternative architecture, denoted the *deterministic assertion architecture*, resembles that of Fig. 4, but has the HMM connected to a purely deterministic formal-specification assertion, instead of a weighted probabilistic module described in section 6. In other words, this architecture is the architecture of Fig. 4 where the *Formal Specification Assertion* block implements assertions using a conventional deterministic implementation, such as the one described in [D1].

Because this approach uses a deterministic assertion, it can only use a single sequence of input symbols from the HMM, such as the sequence $a_1,a_2,\ldots,a_T$ where $a_t=\max_{1\le i\le N}(\delta_t(i))$. However, the following example demonstrates the weakness of this approach.

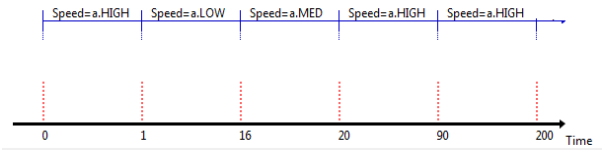
Consider the TLC scenario depicted in Fig. 9a. Using the above mentioned single sequence method it induces the sequence seq_1 of hidden states depicted in Fig. 9b, with probability $P1=\delta_T(seq_1)=9.677258147046034E\text{-}7$. This sequence conforms to requirement R1 because it does not contain two consecutive minutes of *Speed*=HIGH while lights are green.

In contrast, the sequence seq_2 of state symbols depicted in Fig. 9c violates R1. It is not generated by the single sequence method because its probability is $P2=\delta_T(seq_2)=4.639731359753094E\text{-}11$ is smaller than $P1$.

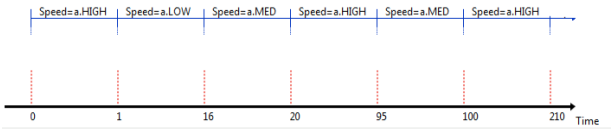
The *alpha* and *gamma* methods are capable of generating the later sequence, thereby enabling our suggested weighted-assertion architecture to detect the violation of R1 failure with a non zero probability. Fig. 10 shows the distribution of $\alpha`$ and $\gamma$ for $seq_1$ and $seq_2$. Note how the $\alpha`$ method generates identical distributions when the observation sequences seq_1 and seq_2 agree, because when the observation sequences agree on the HMM state $q_t=s_t$, then given that state, the observation O_t is independent of prior observations and states. In contrast, the γ method depends on future observations too, and O_t is not necessarily independent of those.



a. Timeline diagram of sequence of a observables O

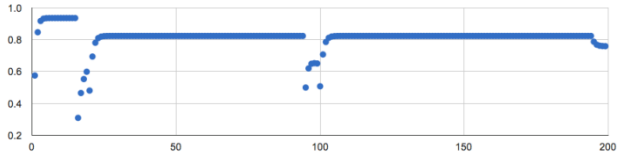


b. Timeline diagram of seq_1, the most probable state sequence that explains O according to the single sequence method. This sequence conforms to requirement R1

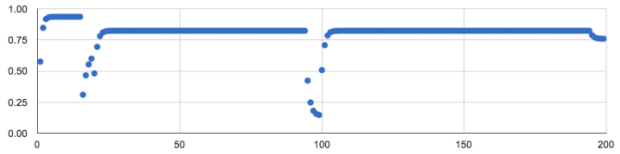


c. Timeline diagram of seq_2, a less probable state sequence due to the time interval [95,99]; this sequence violates requirement R1

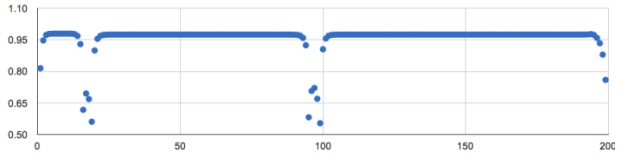
Figure 9. Scenarios that discriminate between the weighted-probabilistic assertion architecture and the deterministic assertion architecture



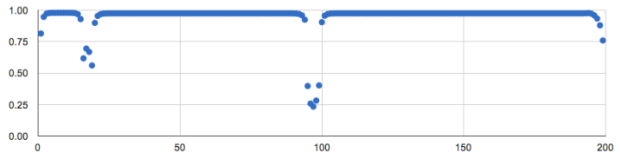
a. The distribution of $\alpha`$ for seq_1



b. The distribution of $\alpha`$ for seq_2



c. The distribution of γ for seq_1



d. The distribution of γ for seq_2

Figure 10. The distribution of $\alpha`$ and γ for seq_1 and seq_2

The second alternative architecture, denoted the *monolithic architecture*, contains no standalone RV module. Rather, the HMM itself, being a probabilistic state machine, performs the RV tasks.

With this approach, the assertions are combined with the symbol decoding HMM inducing a much larger HMM.

Two primary drawbacks of this approach are:

a. The overall RV system is hard to read and maintain; with no separation of concerns within the HMM, it is effectively performing two distinct jobs: (i) decoding hidden symbols from visible ones, and (ii) monitoring or verifying a requirement such as R1 or R2.

b. The HMM is the monolithic architecture, being larger and harder to read, will be harder to learn using the experimental approach discussed in section 5.

11. CONCLUSION

We have demonstrated a technique for performing RV in the presence of hidden evidence. We plan on applying this technique to the verification of aerospace applications, in which the evidence is provided as telemetry files that often do not contain the artifacts asserted about by the formal specifications. We also plan on applying this technique to automatic pattern detection within large volumes of cyber data, in an effort to identify malicious or dangerous behavioral patterns.

We are currently building a special StateRover code-generator that generates weighted/probabilistic implementation code for statechart assertions.

Considering the TLC example, one might wonder how the TLC itself is implemented to conform with requirement R1, given that the *Speed* variable is hidden. In other words, wouldn't TLC developers face the same difficulties when implementing the TLC as the quality assurance team faces when asserting about it? Indeed, in on-going research we are investigating the use of the proposed technique for controllers that operate in difficult environments where some of the inputs are not directly observable, as often the case in hostile environment.

12. REFERENCES

[Da] K.H. Davies, R. Biddulph, and S. Balashek, (1952) Automatic Speech Recognition of Spoken Digits, J. Acoust. Soc. Am. 24(6) pp.637 - 642

[D1] D. Drusinsky, *Modeling and Verification Using UML Statecharts – A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*, Elsevier, 2006.

[D2] D. Drusinsky, *Practical UML-based Specification, Validation, and Verification of Mission-critical Software*, DogEar Publishing, 2011.

[D3] D. Drusinsky, "The Temporal Rover and the ATG Rover," in Proc. SPIN 2000 Workshop, 2000, LNCS 1885, Springer-Verlag, pp. 323-329.

[DMS] D. Drusinsky, J.B. Michael and M. Shing, "A Visual Tradeoff Space for Formal Verification and Validation Techniques," IEEE Systems Journal, Vol. 2, No. 4, Dec. 2008, pp. 513-519.

[DMOS] D. Drusinsky, B. Michael, T. Otani, M. Shing, Validating UML Statechart-Based Assertions Libraries for Improved Reliability and Assurance,. Proceedings of

the Second International Conference on Secure System Integration and Reliability Improvement (SSIRI 2008), Yokohama, Japan, 14-17 July 2008, pp. 47-51. Best paper award.

[DS] D. Drusinsky and M. Shing, "Verification of Timing Properties in Rapid System Prototyping", Proc.14th IEEE International Workshop in Rapid Systems Prototyping, 9-11 June 2003, pp. 47-53.

[DDS] K. Demir, D. Drusinsky, and M. Shing, "Creation and Validation of Embedded Assertions Statecharts", *Proc 17th IEEE International Workshop on Rapid Systems Prototyping*, Chania, Greece, June 2006, 17-23.

[Ha] D. Harel, Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3):231–274, June 1987.

[E] S. Easterbrook, R. Lutz, R. Covington, J. Kely, Y. Ampo and D. Hamilton, "Experiences using lightweight formal methods for requirements modeling", *IEEE Transactions on Software Engineering*, 24(1), pp. 4-11, Jan 1998.

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

[HP] K. Havelund and T. Pressburger, "Model Checking Java Programs using Java PathFinder," Int'l J. Software Tools for Technology Transfer, vol. 2, no. 4, pp. 366-381, 2000.

[HR] K. Havelund and G. Rosu, "An Overview of the Runtime Verification Tool Java PathExplorer," Formal Methods in System Design, vol. 24, Springer Netherlands, pp. 189-215, 2004.

[HWU] J. E. Hopcrofthttp://www.amazon.com/Introduction-Automata-Theory-Languages-Computation/dp/0201441241, R. Motwanihttp://www.amazon.com/Introduction-Automata-Theory-Languages-Computation/dp/0201441241, J. D. Ullmanhttp://www.amazon.com/Introduction-Automata-Theory-Languages-Computation/dp/0201441241, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 2006.

[JU] JUnit, http://www.junit.org

[KJ] Z. Kohavi and N. K. Jha, *Switching and Finite Automata Theory*, Cambridge University Press, 2009.

[Ma] T.P. Mann. Numerically stable hidden markov model implementation. An HMM scaling tutorial, 2006.

[Pi] John Pierce (1969). *Whither Speech Recognition*. Journal of the Acoustical Society of America.

[Ra] L. W. Rabiner, A Tutorial on Hidden Markov models and Selected Applications in Speach Recognition, Proc. of the IEEE, Vol 77, No. 2, 1989.

[Rä] Rätsch, Gunnar; Sonnenburg, S; Srinivasan, J; Witte, H; Müller, KR; Sommer, RJ; Schölkopf, B (2007-02-23). "Improving the C. elegans genome annotation using machine learning". PLoS Computational Biology 3 (2): e20. doi:10.1371/journal.pcbi.0030020. PMC 1808025. PMID 17319737.

[SLS] U. Sammapun, I. Lee, and O. Sokolsky, “RT-MaC: Runtime Monitoring and Checking of Quantitative and Probabilistic Properties,” in Proc. 11th IEEE Int’l Conf. Embedded and Real-Time Computing Systems and Applications, 2005, IEEE, pp. 147-153.

[SR] The StateRover, http://www.time-rover.com

[Si] S. Singh (1999). The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography. London: Fourth Estate. pp. 143–189. ISBN 1-85702-879-1.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943

4. Dr. Robert Kehlet,
   Defense Threat Reduction Agency (DTRA) -
   8725 John J Kingman Rd., Stop 6201 (RD-BAT),
   Fort Belvoir VA 22060-6201

5. Professor Doron Drusinsky
   Naval Postgraduate School
   Monterey, California