A Hidden Markov Model based Runtime Monitoring tool

Drusinsky, Doron
Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/47575

Downloaded from NPS Archive: Calhoun
NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

A HIDDEN MARKOV MODEL BASED RUNTIME MONITORING TOOL

by

Doron Drusinsky

January 2016

Approved for public release; distribution is unlimited

Prepared for: Defense Threat Reduction Agency
## A Hidden Markov Model based Runtime Monitoring Tool

Runtime Monitoring (RM), also known as Runtime Verification (RV), is the process of monitoring and verifying the sequencing and temporal behavior of an underlying application and comparing it to the correct behavior as specified by a formal specification pattern. Hidden Markov Model (HMM) based RM enables the monitoring of systems with both visible and hidden data, using the same formal specifications used by deterministic RM. Hence, with HMM-based RM, formal specifications need not contain probability measures. This report details the process and instructions for using the newly developed tool kit for HMM-based RV.
The report entitled "A Hidden Markov Model based Runtime Monitoring Tool" was prepared for and funded by 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
Computer Sciences Department

Reviewed by:  Released by:

Peter Denning, Chairman  Jeffrey D. Paduan
Computer Sciences Department  Dean of Research
Runtime Monitoring (RM), also known as Runtime Verification (RV), is the process of monitoring and verifying the sequencing and temporal behavior of an underlying application and comparing it to the correct behavior as specified by a formal specification pattern. Hidden Markov Model (HMM) based RM enables the monitoring of systems with both visible and hidden data, using the same formal specifications used by deterministic RM. Hence, with HMM-based RM, formal specifications need not contain probability measures. This report details the process and instructions for using the newly developed tool kit for HMM-based RV.
TABLE OF CONTENTS

I. OVERVIEW .................................................................................................................. 9
II. PREREQUISITES: ......................................................................................................... 11
III. STEP 1: (DETERMINISTIC) RULE DEVELOPMENT .............................................. 12
IV. STEP 2: STANDARD STATEROVER RULE CREATION AND CODE GENERATION ......................................................................................................................... 16
V. STEP 3: GENERATE HMM FROM YOUR LEARNING-PHASE CSV FILE. 21
VI. STEP 4: GENERATE SPECIAL JAVA CODE FOR PROBABILISTIC RM. 23
VII. STEP 5: RUN THE ALPHA METHOD ........................................................................... 25
VIII. STEP 6: RUNTIME MONITORING ............................................................................ 26
IX. PYTHON EXAMPLE SCRIPT ....................................................................................... 27
LIST OF REFERENCES ...................................................................................................... 29
INITIAL DISTRIBUTION LIST .......................................................................................... 31
THIS PAGE INTENTIONALLY LEFT BLANK
I. OVERVIEW

Run-time Verification (RV) of formal specification assertions 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 pattern. In [D3, D4], 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. The StateRover [SR] is a set of Eclipse IDE plugins for UML-statechart based modeling, specification, validation, and runtime verification.

Rules4business [R4B] is an online rule specification and validation service. It contains a library of two-dozen generic rules that enable the specification of sequencing and temporal rules. For example, generic Rule 11 on rules4business.com is: “Flag whenever event P occurs with eventual event Q occurring within time T after P”. After selecting this generic rule the end user will customize P, Q, and T. Each such natural language rule is accompanied by its formal specification in the form of a statechart assertion diagram (e.g., Fig. 1).

The rules4business monitoring domain is the end-user’s credit-card statement, bank account statement, or any other a comma separated delimited file (CSV) file. A generic event (e.g., P or Q) is customized by the end user using Python or Javascript by asserting on columns of the CSV file; for example: P=description.indexOf("Acme Inc.")>=0 means “Acme Inc.” appears in the description column of the CSV.

Rules4business monitors (checks) an uploaded CSV file against the rules the end-user created, thereby effectively performing RM using formal statechart-assertion specifications. The end-user can visually witness how each rule arrived at its Flag or no Flag conclusion by animating the statechart assertions and the upload CSV file on a step-by-step basis; the end-user can move forwards and backwards in time while doing so.

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 [Pi], Gene prediction Rä], and Cryptanalysis [Si].

In [D1, D2] the author describes an RV technique and algorithm for systems with hidden and visible inputs and outputs. The technique uses deterministic Statechart assertions for formal specification; these specifications contain no probability measures. In other words, the end-user need to be concerned with obtaining a sample space for the requirement set, nor does s/he need to obtain probabilities for the specifications. Rather, the end-user needs to create a spreadsheet representation of the HMM outputs as a function of visible artifacts; the tool set then automatically generates the HMM from this spreadsheet.

This report provides instructions for using the RV tool-kit for HMM-based verification. As described in the report, tool is applied to statechart assertions that are customized using rules4business.com, and whose deterministic implementation is code generated by the StateRover’s code-generator.
II. PREREQUISITES:

1. Java 1.7 or higher
2. From eclipse.org: Eclipse package called “Eclipse Modeling Tools (and not any other package of Eclipse)
3. The StateRover Eclipse plugin. Get the update site files from Doron, copy them to your machine (I call your copy the “repository”), go to Eclipse: Help → New Software → Add → Local → point to the folder named “update” in that repository. Uncheck “Group Items by Category”, select all plugins, and then check OK till the end. EClipse will restart.
III. **STEP 1: (DETERMINISTIC) RULE DEVELOPMENT**

1. Use rules4business.com to choose and customize your statechart assertion(s). However, your rules must all be deterministic assertions (vs. non-deterministic ones).

I created two rules, Rule 9 and Rule 21, whose rules4business.com instances are depicted below.

- **Rule 9:** Flag when two suspicious (HiddenState) payments are less than 4 weeks apart. Rules4business parameters are as follows. Events: $E=\text{HiddenState}==\text{"S"}$, Time bounds: $T=4$, Time units: weeks
- **Rule 21:** Flag when there are two or more suspicious or flagged (HiddenState) events between any two Maya software salary deposits. Rules4business parameters are as follows. Events: $E=\text{description.indexOf("Maya")}\geq0$, $Q=E=\text{HiddenState}==\text{"S"}$ or $E=\text{HiddenState}==\text{"F"}$, Count Limits: $N=1$

\[\text{Figure 1. Rule 9 in rules4business.com}\]
2. Validate the assertion by uploading a CSV file and checking that the rule flags precisely when you expect it too. Using our running example set, each rule was validated using Validation.csv, i.e., by uploading Validation.csv to rules4business. For example, the images below depict the validation of expected rule Flagging.

Figure 3. Validation of Rule 9, using rules4business.com. A HiddenState==”S” is discovered twice in a 4 week period ending on 04/12/2013.
Figure 4. Validation of Rule 21, using rules4business.com. Two HiddenState=="S" or "F" are discovered twice between two “MAYA” payroll events, the last of which is on 03/15.

Note that all columns of the Validation.csv spreadsheet are visible. Hence, for columns that are eventually going to be hidden, validate the rule as if they are visible.

For example, Rule 9 has an event E being the condition $\text{HiddenState}=="S"$ (i.e., the hiddenState column value equals “S”, i.e., suspiciousActivity, (as explained below, HiddenState is the name of the hidden column); note how the event E is a key-value pair: $E=\text{HiddenState}=="S"$; as depicted below, the HiddenState column of Validation.csv is populated for validation testing. Stated differently, we validate using CSV files that are later (step-3) be used for HMM training/learning.

Figure 5. A snippet of Validation.csv. Note how (for Validation and Learning phase purposes only), the HiddenState column is populated.

3. Create a properties file named <rule_name>events.properties. This property file should contain key-value pairs you specified in step-2, such as $P=\text{amount}==\text{amountGT500}$. The following two files are the events.properties files for Rules 9 and 21 of our running example.

```
#Rule9_Events.properties
E=HiddenState=="S"
```

```
#Rule21_Events.properties
E=HiddenState=="S"
```
T=4 weeks

#Rule21_Events.properties
E=description.indexOf("Maya")>=0
Q=HiddenState==="S" or HiddenState==="F"
N=1
IV.  STEP 2: STATEROVER RULES: CREATION AND CODE GENERATION

In this phase we will convert the rules4business diagrams to StateRover diagrams.

1. Create a Java project named Rules. Implement each StateRover rule in a separate package. In my running example I therefore have two packages: Rule9 and Rule21.
2. Select that package. Create a statechart assertion using the StateRover: File→New→TimeRover→“Statechart Assertion Diagram”; name it, Next, Next, Finish.
3. In the package explorer, you will see 3 files inside the package you created: a .statechart file, a .statechart_diagram file, and a .properties file. Do not edit the .statechart file.
4. Go to Eclipse’s Preferences→Time Rover→Code Generation, check the second check box labeled “Force…”.
5. Switch to the “Statechart Diagram” perspective.
6. Open the .statechart_diagram file. Draw the diagram(s) you used in rules4business.com (Step 1). The following two screenshots are of the two diagrams (Rule 9 and Rule 21, respectively), as drawn using the StateRover.

Figure 6. The StateRover version of Rule 9
When creating StateRover statechart-assertion diagrams please note the following:

1. Use **Compound States** in the StateRover pallet for states. Use **Flowchart Action Box**’es for flowchart boxes. Use a **Variable box** to declare variables, a **Comment box** for comments.

2. Actions within states and flowchart box are **On Entry** actions (see **Properties view**, under the **StateChart Diagram** perspective).

3. In the **Flag** state: instead of **Flag=true**, write **bSuccess=false**; The StateRover flags when **bSuccess** (and the corresponding **isSuccess** Boolean method) is **false**.

4. As depicted for Rule 9, a timer object is declared as follows:

   ```java
   TRTimeoutSimulatedTime timer = new TRTimeoutSimulatedTime (T, this);
   ```

5. The constant T for the timer must be declared as: **TRShared int T**; (or T1, T2, depending on the diagram specifics in rules4business.com). The counting constant must be declared as: **TRShared int N**;

6. Edit the .statechar_properties file; select “**Code Generation Properties**” and in the **Properties view** select the code generator named “Java – Non-deterministic”.

---

**Figure 7. The StateRover version of Rule 21**
Figure 8. Setting the code generator to Non-deterministic Java

Note that there is no conflict between line-item 1 of step-1 (where it says you should only use deterministic assertions) and choosing the “Java non-deterministic” code generator.

7. In the Package Explorer, select the .statechart_diagram file, right click and check “Enable Code Generation”.

8. Generate code by simply saving your statechart_diagram file - which induces code generation inside your package. Make sure that your package now includes a newly created Java file whose name is the name of your statechart_diagram file (e.g., foo.statechart_diagram induces foo.java).

9. Add TReclipseAnimation.jar and stateroverifacesrc.jar to the Rules projects’ Build Path.

10. To assure that you write a small JUnit sanity test case to assure that the generated code compiles are runs ok. The following are two JUnit test cases for our running example set.

```java
package Rule9;

import junit.framework.TestCase;
import org.junit.Before;
import org.junit.Test;

public class SanityTest {

    Rule9 rule;

    @Before
    public void setup() {
        rule = new Rule9();
        rule.T=30;
        rule.execTRreset(); // enforce the setting of T
    }
}
```
@Test
public void test() {
    rule.incrTime(10);
    rule.E();
    rule.incrTime(35);
    rule.E();
    TestCase.assertTrue(rule.isSuccess()); // E's are more than 30 units apart
}

@Test
public void test1() {
    rule.incrTime(10);
    rule.E();
    rule.incrTime(15);
    rule.E();
    TestCase.assertNull(rule.isSuccess()); // E's are less than 30 units apart
}
}


package Rule21;

import junit.framework.TestCase;
import org.junit.Before;
import org.junit.Test;

public class SanityTest {
    Rule21 rule;
    @Before
    public void setup() {
        rule = new Rule21();
        rule.N=1;
    }
    @Test
    public void test() {
        rule.E();
        rule.Q_and_notE();
        rule.E();
        rule.E();
        TestCase.assertTrue(rule.isSuccess());
    }
    @Test
    public void test1() {
        rule.E();
        rule.Q_and_notE();
        rule.Q_and_notE();
        rule.E();
        rule.E();
        TestCase.assertNull(rule.isSuccess());
    }
}
Quantization: The following explains quantitation:

The learning phase CSV columns (and matching columns in run-time CSV files) events must relate to quantized data values. For example, while rules4business supports a custom event \( P=amount>0 \), this kind of open-ended range is an overkill for an HMM. Hence, you need to quantize \( amount \) into a set of completing yet non-overlapping ranges (typically up to 4 ranges), such as the following 3 ranges: (i) \( amount\leq 0 \) denoted \( amountLE0 \), (ii) \( 0<amount\leq 500 \) denoted \( amount0to500 \), and (iii) \( 500<amount \) denoted \( amountGT500 \).

This is done to inform our tool set of the mapping between generic names (e.g, \( P, E, Q, R \)) and the csv columns. Hence, for this quantization you could write: 

\[ P=amount==amountLE0, \ P=amount==amount0to500, \text{ or } P=amount==amountGT500. \]

Suppose for a moment that \( P=amount==amountGT500 \); this informs our tools that the runtime values of statechart-assertion event \( P \) are taken from csv column named \( amount \), and that event \( P \) will fire whenever that column has a value of: \( amountGT500 \)

1. The learning csv file is a single learning-phase file referred to in step 1. If you have more than one such file then append them to create one file. Lets call all csv files you appended together - “parts” of the resulting single csv file.

2. Make sure the learning-phase csv file has a special row No. 1 - with column names in it.

3. Add a special column (usually on the left, i.e., the first column) named \( InitialState \). In this column, write the letter \( Y \) for every row that was the first row of a “part” file. You need not write anything elsewhere.

4. The HMM generator requires a column named \( HiddenState \). Since this is the learning phase, you now play the role of an expert: fill in the cells of the \( HiddenState \) column values from the set of possible hidden states.

5. Column names (except \( InitialState \) and \( HiddenState \) ) must correspond to the r.h.s (value) of the key-value pair entries in the \(<rule_name>events.properties \) of Step-1 and 2. For example if you have \( P=amount==amountGT500 \) as a key-value pair then the csv file must contain a column named \( amount \) whose entries (possible values) are the quantized values you decided about in step-1 \( (amountLE0, amount0to500, amountGT500) \). The inverse also holds; ex: a column named \( foo \) must correspond to some implied column in the properties file (implied by the existence of some customization like \( Q=foo==fooGT20 \)).

The following is an example: Suppose Rule1 contains a transition labeled as event \( P \). In rules4business (step-1) you map \( P=amount==amountGT500 \); The following table depicts a learning-phase csv. The Rule1 events.properties file should contain \( P=amount==amountGT500 \).

<table>
<thead>
<tr>
<th>InitialState</th>
<th>amount</th>
<th>…</th>
<th>HiddenState</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>amount0to500</td>
<td>…</td>
<td>Benign</td>
</tr>
</tbody>
</table>
6. Run the command: `java -jar dtrahmm.jar <arguments>`, where the arguments list is:
   - Path to the *learning-phase CSV file*, no spaces are allowed in path string.
   - Path to the folder containing *quantization.properties* file – a file that informs tool how to quantize columns.
   - (Optional) the letter D: used whenever your CSV file is in descending order (i.e., first row is column headers, then the most recent data row by Date, then the second most recent row, etc.)

In the running example case I executed:
```
java -jar dtrahmm.jar LearningPhase.csv PythonQuantizationScripts d
```

The output from this file is an HMM parameters file named *hmm.json*; you will be using it in the RM step. It will be generated as a sibling file to the input file.

7. Note that all information in the csv-file is case **insensitive**.
VI. STEP 4: GENERATE SPECIAL JAVA CODE FOR PROBABILISTIC RM.

1. Run the command: `java -jar dtracg.jar <arguments>` where the arguments list is “?” for usage instructions, or (in this particular order):
   - **Java file** – this is a path to the Java file created in step 2.9. No spaces are allowed in path string.
   - `<rule_name>events.properties file` - this is a path to the file created in step 1.3

   This step results in a new sibling file Java file called `inFile_DTRA.java`, where `inFile` is the name of the input file you provided as an argument above.

2. We will now perform sanity testing on the resulting Java code.
   a. Similarly to Step2.1: create a Java project named `DTRA_Rules`, and in it create a Java a package per rule (i.e., Rule9 and Rule21 in my example).
   b. Move or copy the `inFile_DTRA.java` files from the location in which they were generated (likely the Rules project) to the corresponding folder in the DTRA_Rules project.
   c. Copy the unit test files you created in step 2.10 to their respective in the DTRA_Rules project.
   d. Add the source code for the package `com.timerover.staterover.ifacesrc` to DTRA_Rules project.

   The resulting file Package Explorer image for my DTRA_Rules project with Rule9 package is illustrated below:

   e. Modify your unit tests to incorporate probability – see lines and comments marked in red.

   ```java
   Rule9_DTRA rule;
   @Before
   public void setup() {
       rule = new Rule9_DTRA();
       rule.N=1;
   }
   @Test
   public void test() {
       rule.incrTime(10);
   }
   ```
rule.E(1.0); //add probability 1 to events
rule.incrTime(35);
rule.E(1.0); //add probability 1 to events
double d = rule.getProbabilityOfSuccess();
System.out.println("in test: d="+d);
TestCase.assertTrue(rule.isSuccess()); // E's are
more than 30 units apart
TestCase.assertEquals(1-d, 0.0); // probability 1
of success = probability 0 of flagging
}

@Test
public void test1() {
    rule.incrTime(10);
    rule.E(1.0); //add probability 1 to events
    rule.incrTime(15);
    rule.E(1.0); //add probability 1 to events
double d = rule.getProbabilityOfSuccess();
    System.out.println("d="+d);
    TestCase.assertFalse(rule.isSuccess()); // E's
    are less than 30 units apart
    TestCase.assertEquals(1-d, 1.0); // probability 0
    of success = probability 1 of flagging
}

Listing 3. Modified unit tests

3. Bundle the entire DTRA_Rules folder as a jar file. For example, use the Eclipse
Export command and Export as a jar file. Let the jar file name be
DTRA_Rules.jar.
VII. STEP 5: RUN THE ALPHA METHOD

Run the command: \texttt{java -jar dtraalpha.jar <arguments>} where the arguments list is “?” for usage instructions, or (in this particular order):

- Path to the run-time CSV file, no spaces are allowed in path string.
- Path to the \texttt{hmm\.json} file that was created in step 3.6.
- Path to the \texttt{folder} containing \texttt{quantization\.properties} file – a file that informs tool how to quantize columns.
- (Optional) the letter D: used whenever your CSV file is in descending order (i.e., first row is column headers, then the most recent data row by Date, then the second most recent row, etc.)

In the running example case I executed:

\texttt{java -jar dtraalpha.jar Checking1.csv hmm.json PythonQuantizationScripts d}

The output from this step is a file \texttt{alpha\.json} in the folder of the input csv file.
VIII. STEP 6: RUNTIME MONITORING

1. Run the command: `java -jar dtrarm.jar <arguments>` where the arguments list is “?” for usage instructions, or (in this particular order):
   - Path to the run-time CSV file; this must be the same file used in step 5.
   - Path to the alpha.json file the file generated in step 5.
   - Path to the DTRA_Rules.jar file you created in step 4.3.
   - Name of the assertion you are monitoring, such as Rule21, or Rule9.
   - `<rule_name>events.properties` file - this is a path to the file created in step 1.3 and same file used in step 4.1.
   - `[D][nnn]`: optional `d` is used whenever your CSV file is in descending order (i.e., first row is column headers, then the most recent data row by Date, then the second most recent row, etc.). Use this parameter if-and-only-if it was used in step 5. `nnn` is an optional number, like 150, used to stop monitoring after 150 cycles.

The output this `dtrarm` execution is a list of probabilities, one per cycle (i.e., per CSV row being monitored – but remember that you might have specified the CSV to be in descending order, the printout is in the order you prescribed). Each probability measure represents the likelihood of reaching the Flag state of the assertion being monitored, in that cycle, as per the Alpha method in [D1].
IX. PYTHON EXAMPLE SCRIPT

The script allsteps.py (in the distrib/Example folder) automates steps 3-6. Note however that the use-case for the tool set is such that creating the HMM is done less frequently than performing RM using that HMM, because RM is performed using many possible CSV files.
LIST OF REFERENCES


[R4B] www.rules4business.com

[SR] www.time-rover.com


INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia
   Attn: Paul Tandy

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

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