TIOA Model Checker User Guide and Reference Manual

December 9, 2006

1 What is TIOA Model Checker?

The TIOA model checker will be based on TIOA2XTA, a translator from the TIOA specification language to XTA, the input language of the UPPAAL model checker for Timed Automata. TIOA2XTA is based on the new TIOA front-end, whose development effort is led by Laurent Michel of UConn. TIOA2XTA utilizes the front-end as follows. After a TIOA specification is parsed and the corresponding abstract syntax tree(AST) generated by the front-end, the TIOA2XTA processes the AST to generate the corresponding UPPAAL (i.e., Timed Automaton) specification. Once the XTA code is generated by TIOA2XTA, TIOA model checker will automatically invoke UPPAAL tool-set and feed in the generated XTA code. Then the UPPAAL tool-set can do the model checking on the XTA code, and model check the original TIOA specification in essence.

2 Language features of TIOA supported by TIOA2XTA

1. Primitive TIOA automaton
2. Composite TIOA automaton
3. Shared-variable (read-write) style specifications of distributed TIOA automata
4. Instantiation of several processes from the same TIOA automaton

3 Restrictions imposed on TIOA specifications by the translator TIOA2XTA

1. All the declared state variables could be only Int, Nat, Bool, Enumeration and array of these four types. Besides, each automaton can only have

"Time" variables declared as Real. Set, map, tuple, sequence, union etc. are all not allowed so far. Imported vocabulary can only be Enumeration. Also, const, type and local keywords are not allowed.

2. The supported operators are operators working with Int, Nat, Bool, Enumeration and array of these four types except abs() for Int and constant() and assign() for array.

3. where, let, choose, initially, urgent when, ensuring clauses are not allowed.

4. exist and for all quantifiers are not allowed so far.

5. Internal actions cannot have parameters.

6. Guard statements are restricted to conjunctions.

7. For input actions, effect statements are restricted to single If-Then statement or any combination of Assignments and For-Loop statements.

8. One must define a variable "mode" with the enumeration type "Location" as a program counter to specify the state information in the generated UPPAAL specification.

9. Trajectory evolve clauses are limited to d(t)=1 (time evolves at constant rate 1) where t in d(t) can only be the Real variable. If trajectories have the clause "invariant x = v1 stop when t = v2", then x is fixed to be the variable "mode", v1 should be one of the enumeration value of the Location type. t should be the declared Real variable expressing the time. v2 should be constant literal 1.

10. tasks block, automaton invariant statement, schedule block and simulation (forward simulation, backward simulation) statement are ignored.

4 Translation scheme

1. TIOA state variables are translated to variables in UPPAAL.

2. TIOA actions are translated to transitions in UPPAAL. TIOA preconditions map to guard clauses of UPPAAL transitions, TIOA effects map to assign clauses of UPPAAL transitions.

3. In the case of composite TIOA, actions in different primitive automata with the same name are translated to synchronized transitions in UPPAAL.

4. If run the model checker with the option -collapse, TIOA trajectories are used to define states in UPPAAL. Therefore, a trajectory may be of the form "invariant mode = ... stop when time = ..." which defines an urgency condition for the UPPAAL state named in the invariant clause. As

such, the number of the trajectories in the TIOA specification determines the number of states in the generated UPPAAL specification. Since the number of trajectories may be less than the specified number of modes, some modes may collapse to a single UPPAAL "DEFAULT" state.

5. The first value of enumeration type "Location" determines the initial state in UPPAAL.

6. Shared variables (read-write) are modeled in TIOA as automata with parameterized transitions that allow other automata in the system to read and write the value of the shared variables in question. During the translation process, the transition parameters are translated to global variables in UPPAAL and the transitions are translated to the corresponding read and write operations on those global variables. The name of shared variable automaton should include "SharedVar".

5 Integration with new front-end

1. The translator is integrated into the new front-end as a plug-in.

2. The translator is integrated into the Eclipse UI.

6 UPPAAL installation

The TIOA model checker could invoke UPPAAL (4.0) tool-set and load in the generated UPPAAL code automatically provided that UPPAAL tool-set is installed in the same logic disk as the TIOA tool-set. You can visit http://www.uppaal.com to download UPPAAL (4.0) tool-set. Also please remember to add UPPAAL tool-set to the PATH variable. Note: after you unzip the UPPAAL tool-set to your machine, please make sure that the files in directories bin-Linux, bin-SunOS have their execution bit set.

7 Examples

Along with the release, TIOA model checker includes following examples.

1. Train: This example models a train approaching a crossing. Each of the actions takes place with a certain urgency.

2. Door: This example represents an impatient person ringing a door bell. There are 4 actions: ringing the bell, door opening for him to go in, him leaving and him getting impatient while waiting.

3. NodeChannel: This example consists of a pair of nodes, communicating through two one-way channels.

4. Fischer protocol (buggy version): This example models fischer protocol which doesn't satisfy the mutual exclusion.

5. Fischer portocol (correct version): This example models fischer protocol which satisfies the mutual exclusion.

6. TrainCrossing: This example models trains crossing one gate satisfying without collision and starvation.

The first three examples are not complete examples for model checking. They are used to demonstrate the translation schemes. The last three are complete examples and could be used to check properties. Along with the last three examples, there are .q files to express the desired properties.

8 Execution of the Fischer example

Here we give the detailed steps to execute TIOA model checker with the buggy fischer example (buggy_fischer.tioa).

1. Run the tioa tool-set with the command: java -jar tioa.jar -plugin=uppaal "path to buggy_fischer.tioa"

2. You will see UPPAAL tool-set is invoked and the generated buggy_fischer.xta code is fed in the UPPAAL. Click Verifier tab, select the property you want to check, then click Check button, you will see "Property is not satisfied".

3. If you want to see the counterexample, click Options, choose Diagnostic Trace → Shortest, then click Check again. A window will pop up saying "Storing the new trace in the simulator will destroy the old one. Continue?", click yes. Then click Simulator tab, the bottom right corner is the message sequence chart (MSC) of the counterexample. You can analyze the MSC to find out the scenario which violates the property.