CS392F P1 Design Description and Instructions

Writing Model-to-Text Transformations with VM2T

Jianyu Huang

Xiaohui Chen

# 0 Running the examples

Please execute the bash-script in Cygwin.

|  |
| --- |
| 1. Make the the script executable  *chmod u+x ./run.script.sh*  2. Run the script  *./run.script.sh [the parts you want to run(optional)]*  *e.g*  *./run.script.sh ---- run all parts (Part 1,2&3)*  *./run.script.sh 3 1 ---- run part 3 and part 1* |

# 1 Part 1

## 1.1 Design

The *vm* code could generate *fsm.java* with methods “*goto[node name]()*”. The only private variable of *fsm* class is *currentState*, which is of type *State*. The reason for this variable to be *private* is to obey the rule of encapsulation in object-oriented programming. The “*goto*” methods simply calls *currentState*. “*gotoXXX*” and returns a state. If the state is null, that means the transition between the two nodes is not possible. Otherwise a new state is returned and *currentState* is assigned to the new state. The transition status would be printed according to the project requirements. The *getName()* method would call the *currentState.getName()* and return the node name.

Our *vm* code for part 1 is shown as the follows,

|  |
| --- |
| *#set($MARKER="//----")*  *${MARKER}fsm.java*  *package myfsm;*  *public class fsm{*  *private State currentState;*  *public fsm()*  *{*  *#foreach($node in $nodeS)*  *#if(${node.type}=="start")*  *currentState=new ${node.nodeid}();*  *#end*  *#end*  *}*  *#foreach($node in $nodeS)*  *public void goto${node.name}()*  *{*  *State tmpState=currentState.goto${node.name}();*  *if(tmpState!=null)*  *{*  *System.out.println("go to ${node.name}");*  *currentState=tmpState;*  *}else*  *{*  *System.out.println("ignoring transition to ${node.name}");*  *}*  *}*  *#end*  *public String getName()*  *{*  *return currentState.getName();*  *}*  *}*  *${MARKER}State.java*  *package myfsm;*  *public interface State*  *{*  *#foreach($node in $nodeS)*  *State goto${node.name}();*  *#end*  *String getName();*  *}*  *#foreach($node in $nodeS)*  *${MARKER}${node.nodeid}.java*  *package myfsm;*  *public class ${node.nodeid} implements State*  *{*  *private String name;*  *private String type;*  *public ${node.nodeid}()*  *{*  *name="${node.name}";*  *type="${node.type}";*  *}*  *#foreach($tmpnode in $nodeS)*  *public State goto${tmpnode.name}()*  *{*  *#set($hastrans=0)*  *#foreach($transition in $transitionS)*  *#if(${node.nodeid}==${transition.startsAt})*  *#if(${tmpnode.nodeid}==${transition.endsAt})*  *#set($hastrans=1)*  *#end*  *#end*  *#end*  *#if(${hastrans}==1)*  *return new ${tmpnode.nodeid}();*  *#else*  *return null;*  *#end*  *}*  *#end*  *public String getName()*  *{*  *return name;*  *}*  *}*  *#end* |

## 1.2 Generated code

*State.java* is simply a java interface and all the methods there are public abstract methods.

The *nXXX.java* are nodes with *nodeid* as class names. In the “*goto*” methods in each class, the tuples in transition tables are inspected. If a transition is possible, then the state in *endsAt* would be returned. Otherwise *null* would be returned.

# 2 Part 2

## 2.1 Design

Part 2 has the same prolog database as that of part 1, but the *fsm* is more abstract. Here, *fsm.java* uses *enum* to include all the possible states. Also, in each “*gotoXXX*” method, the *vm* evaluated the tuples in transition table. In this case a switch statement is used to test whether the transition from *currentState* to state XXX is possible. Therefore it would print out exactly the same results as that of part 1.

Our *vm* code for part 2 is shown as the follows,

|  |
| --- |
| *#set($MARKER="//----")*  *${MARKER}fsm.java*  *package myfsm;*  *public class fsm {*  *public enum states {#set($comma="")#foreach($node in $nodeS)$comma ${node.name}#set($comma=",")#end }*  *#foreach($node in $nodeS)*  *#if(${node.type}=="start")*  *states currentState = states.${node.name};*  *#end*  *#end*  *public String getName() { return currentState.toString(); }*  *#foreach($node in $nodeS)*  *public void goto${node.name}() {*  *switch(currentState) {*  *#set ($transFlag=0)*  *#foreach($transition in $transitionS)*  *#if (${transition.endsAt} == ${node.nodeid})*  *#set ($transFlag=1)*  *#foreach($node in $nodeS)*  *#if (${node.nodeid} == ${transition.startsAt})*  *case ${node.name} :*  *#end*  *#end*  *#end*  *#end*  *#if (${transFlag} == 1)*  *System.out.println("going to ${node.name}");*  *currentState = states.${node.name};*  *break;*  *#end*  *default :*  *System.out.println("ignoring transition to ${node.name}");*  *}*  *}*  *#end*  *}* |

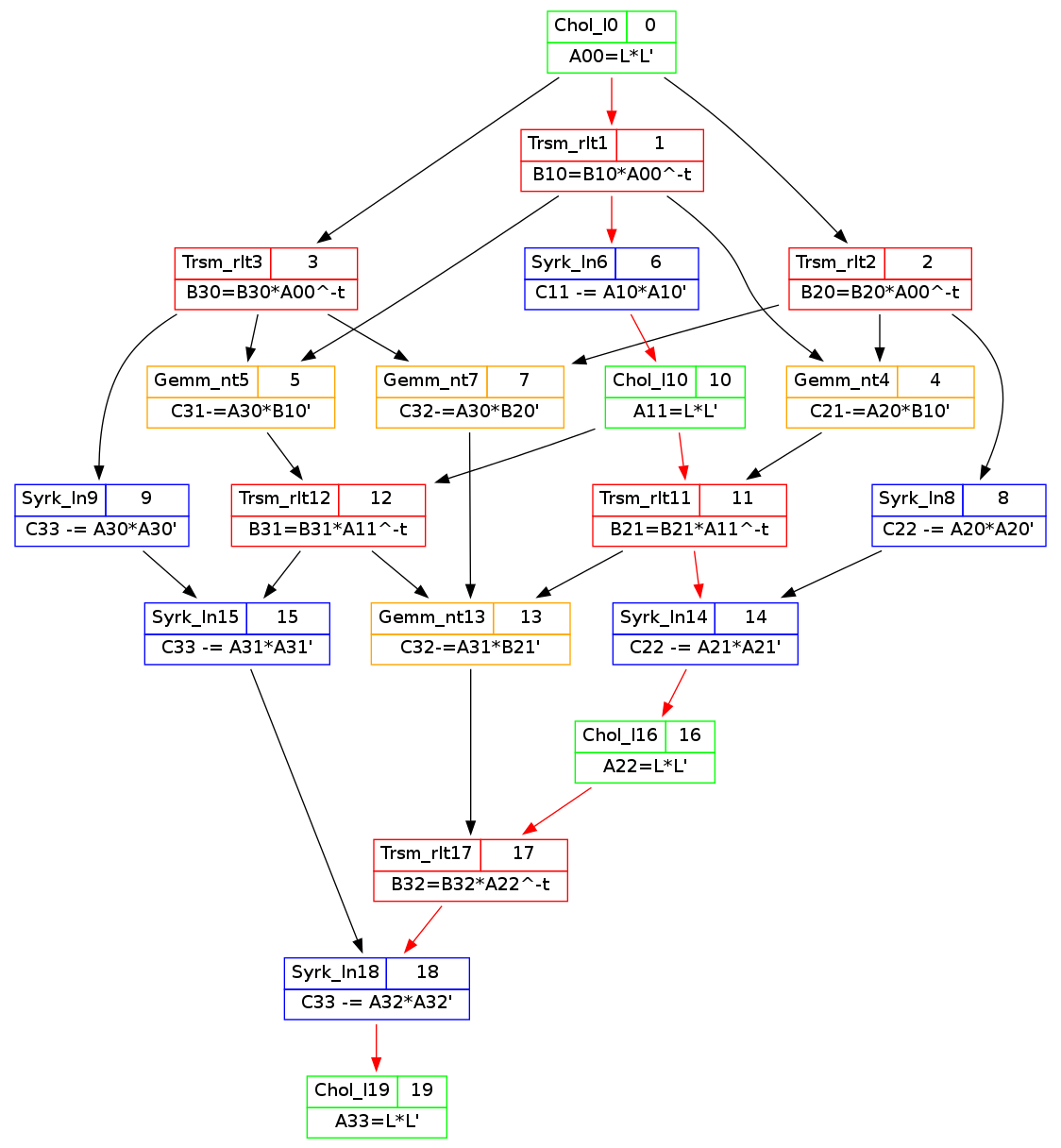
## 2.2 Generated code

*fsm.java* is an all-in-one class wrapping up all “gotoXXX” methods with the *enum* states.

# 3 Part 3

## 3.1 FSM example

We want to show a FSM (finite state machine) example in the correctness verification for *Supermatrix* run-time system in the *DLA (dense linear algebra)* domain. *Supermatrix* is a run-time system for task scheduling. In the first stage of *Supermatrix* run-time system, we need to generate the *DAG (directed acyclic graph)* for the dependency relations for the tasks of *linear algebra subroutine*. The following DAG is for a 4x4 *Cholesky* decomposition.



FSM for a correct dependency path in the above *Cholesky* decomposition should be



## 3.2 Prolog database definition

We use Prolog to represent the above FSM:

|  |
| --- |
| *%dbase(fsm,[node,transition]).*  *%table(node,[nodeid,name,type]).*  *node(nStart, start, start).*  *node(nChol, CHOL, state).*  *node(nTrsm, TRSM, state).*  *node(nSyrk, SYRK, state).*  *node(nGemm, GEMM, state).*  *node(nStop, stop, stop).*  *%table(transition,[transid,startsAt,endsAt]).*  *transition(t1, nStart, nChol).*  *transition(t2, nChol, nTrsm).*  *transition(t3, nTrsm, nGemm).*  *transition(t4, nGemm, nGemm).*  *transition(t5, nGemm, nTrsm).*  *transition(t6, nTrsm, nSyrk).*  *transition(t7, nSyrk, nSyrk).*  *transition(t8, nSyrk, nChol).*  *transition(t9, nChol, nStop).* |

## 3.3 Verification

We use this FSM to verify the correctness of one specific dependency path (the red path in the DAG). The *app.java* is as the follows,

|  |
| --- |
| *import myfsm.\*;*  *public class app {*  *public static void main(String[] args) {*  *System.out.println("----");*  *paces( new fsm() );*  *System.out.println("----");*  *}*  *public static void paces( fsm f ) {*  *f.gotoCHOL();*  *f.gotoTRSM();*  *f.gotoSYRK();*  *f.gotoSYRK();*  *f.gotoCHOL();*  *f.gotoTRSM();*  *f.gotoSYRK();*  *f.gotoCHOL();*  *f.gotostop();*  *System.out.println(f.getName());*  *}*  *}* |

With the help of VM2T tools and our general *vm* files (model-to-text mappings), we can easily generate the code for our FSM with the *vm* files in either part 1 or part 2, thus we can verify the correctness of the red path in DAG.

|  |
| --- |
| *----*  *go to CHOL*  *go to TRSM*  *go to SYRK*  *go to SYRK*  *go to CHOL*  *go to TRSM*  *go to SYRK*  *go to CHOL*  *go to stop*  *stop*  *----* |

There is no “ignoring transition to …” message in the output. So we can verify that that specific task dependency path is correct.