Mapping Big-Step Modeling Languages to SMV

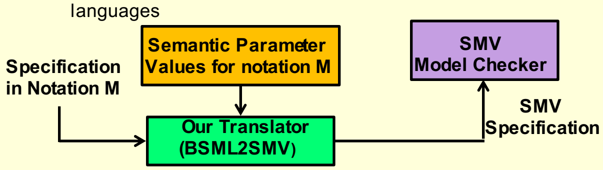
Formal Verification for Model-Driven Engineering

Fathiyeh Faghih and Nancy A. Day
University of Waterloo
Waterloo, ON, Canada

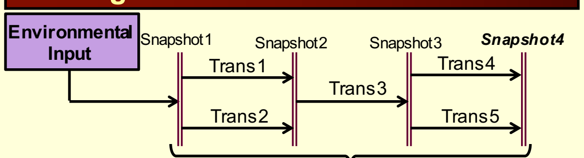
For more information, contact:
Fathiyeh Faghih
E-mail: ffaghihe@cs.uwaterloo.ca
http://www.cs.uwaterloo.ca/~ffaghihe

1- Introduction

- Different behavioural modeling languages are used in model-driven methodologies
  - ✓ How do we verify properties of models designed in different languages?
  - ✓ Solution: transform a design model notation to the input language of an analyzer, such as a model checker
- Our contribution is a semantics-based translator from the family of big-step modeling languages (BSMLs) to the input language of SMV.
- A BSML is described using parameter values for semantic aspects.
  - ✓ Using our translator, different combinations of options for semantic aspects will lead to generating new translators for specific languages

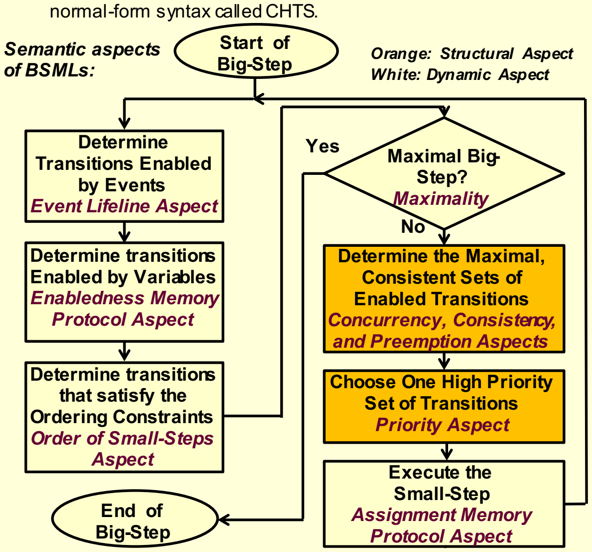


2- Background: BSMLs

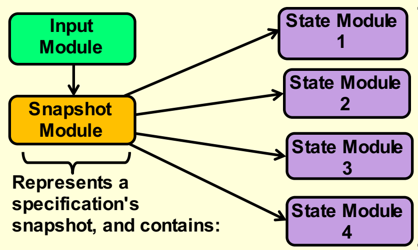


***Big-Step*: Reaction of model to an environmental Input, which consists of a sequence of small-steps**

- Many BSMLs exist with different syntaxes and semantics, such as Statecharts, Argos, Reactive Modules, and SCR, so we use a normal-form syntax called CHTS.



3- Module Structure in SMV



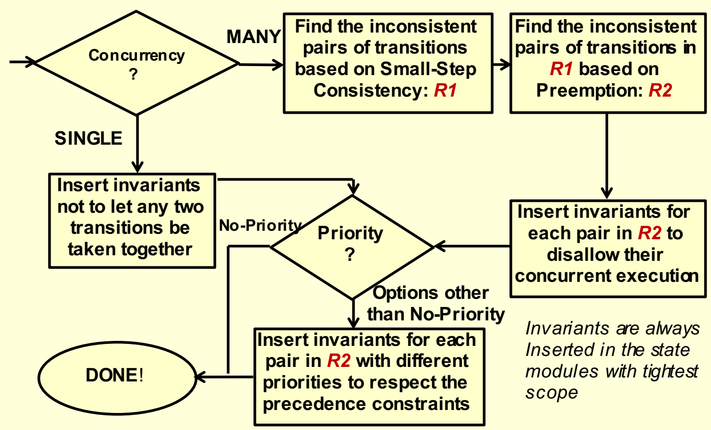
Represents a specification's snapshot, and contains:

- ✓ A Boolean variable for each basic state and a macro for every non-basic state representing whether they are active or not
- ✓ A Boolean variable and a variable with the appropriate type for each event and each variable in the original model
- ✓ A Boolean variable for each transition in the model: execution variables

One for each non-basic state, and includes:

- ✓ Enabledness flag for each transition in the state's scope
- ✓ An instance of the state modules corresponding to the state's direct children
- ✓ Two flags (macros) for the state enabledness and execution

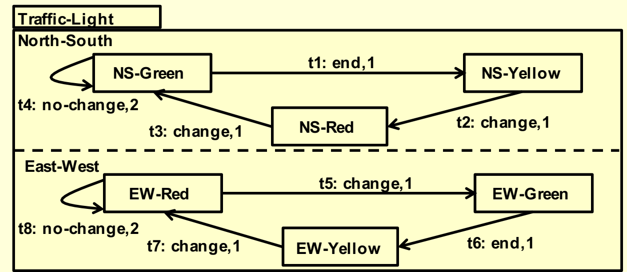
4- Translating the Structural Semantic Aspects

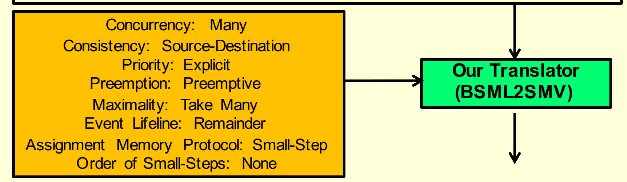


5- Translating the Dynamic Semantic Aspects

- **Maximality**
  - ✓ Using a boolean variable to indicate when the big-step concludes and the next input should be read from the Input module
- **Event Lifeline**
  - ✓ Reflected in the next statement of events in the Snapshot module
- **Assignment Memory Protocol and Enabledness Memory Protocol**
  - ✓ If the option is Big-step, a copy of variable value at the beginning of the big-step is needed and will be used accordingly.
- **Order of Small-Steps**
  - ✓ If the option is Explicit, invariants are used to impose that a transition is enabled, only if none of its predecessors are enabled.

6- Example






```
MODULE snapshot(){
VAR
--basic states
NS_Green,NS_Yellow,NS_Red,
EW_Green,EW_Yellow,EW_Red:boolean;
--events
end,change : boolean;
--execution variables of transitions
t1_exe,t2_exe,t3_exe,t4_exe,t5_exe
t6_exe,t7_exe,t8_exe : boolean;
DEFINE
North-South :=
NS_Green|NS_Yellow|NS_Red;
East-West :=
EW_Green|EW_Yellow|EW_Red;
--next statements
default
{next(no-change)  := no-change;}
in{if(stable)
     next(no-change) := input.no-change;}
--similar statements for "end" and "change"
next(NS_Green) := case {
t3_execute | t4_execute : 1;
t1_execute : 0;
1 : NS_Green;
};
--similar statements for all basic states}
MODULE main(){
--instance of the module snapshot
ss : snapshot;
model_root : Traffic_Light(ss);}
MODULE Traffic_Light(ss){
North_South : North_South(ss);
East_West : East_West(ss);
enabled := North_South.enabled |
East_West.enabled;
execute := North_South.execute |
East_West.execute;
ss.stable := ~enabled;
--invariants
enabled -> execute;}
MODULE North_South(ss){
t1_enabled := ss.NS_green & end;
t2_enabled := ss.NS_yellow & change;
t3_enabled := ss.NS_red & change;
t4_enabled:=ss.NS_green& no-change;
enabled := t1_enabled | t2_enabled |
t3_enabled | t4_enabled;
execute := ss.t1_execute |
ss.t2_execute | ss.t3_execute |
ss.t4_execute;
--invariants
~(ss.t1_execute & ss.t4_execute);
execute -> ((t1_enabled&~t4_execute)
-> t1_execute);
execute -> ((t4_enabled&~t1_execute)
-> t4_execute);
execute -> (t2_enabled-> t2_execute);
execute -> (t3_enabled-> t3_execute)
--priority invariant
t4_enabled -> ~t1_execute;}
MODULE East_West(ss){
t5_enabled := ss.EW_red & change;
t6_enabled := ss.EW_green & end;
t7_enabled := ss.EW_yellow & change;
t8_enabled := ss.EW_red & no-change;
enabled := t5_enabled | t6_enabled |
t7_enabled | t8_enabled;
execute := ss.t5_execute |
ss.t6_execute | ss.t7_execute |
ss.t8_execute;
--invariants
~(ss.t5_execute & ss.t8_execute);
execute ->((t5_enabled&~t8_execute)
-> t5_execute);
execute ->((t8_enabled&~t5_execute)
-> t8_execute);
execute -> (t6_enabled-> t6_execute);
execute -> (t7_enabled-> t7_execute)
--priority invariant
t8_enabled -> ~t5_execute;
}
```