

Verifying Temporal Properties via Dynamic Program Execution

Zhenhua Duan
Xidian University, China

Main Points

- Background & Motivation
- MSVL and Compiler
- PPTL
- Unified Program Verification
- Tool Demo
- Conclusion and Future Work

Background & Motivation

We focus on Software model checking in code-level

(1) Rechability analysis of bad things

```
int* a;
int i=0;
…
if(a==0) goto Err;
i= *a;   //de-referencing a
…
Err:
```

However, verifiation of other temporal properties such as liveness etc. cannot be supported!

- Suitable for only safety property verification
- Two well known ways: CEGAR and bounded model checking
- Tools: SLAM, BLAST, CPAChecker and CBMC …

Background & Motivation

(2) Model checking temporal properties without executing code (static)

- Considering all possible behaviors makes small programs have large state-space
- Tools: Ultmate LTLAutomizer, T2, ...
- **Difficult to verify programs in large scale**
- **Poor in accuracy with lots of false positives**

Background & Motivation

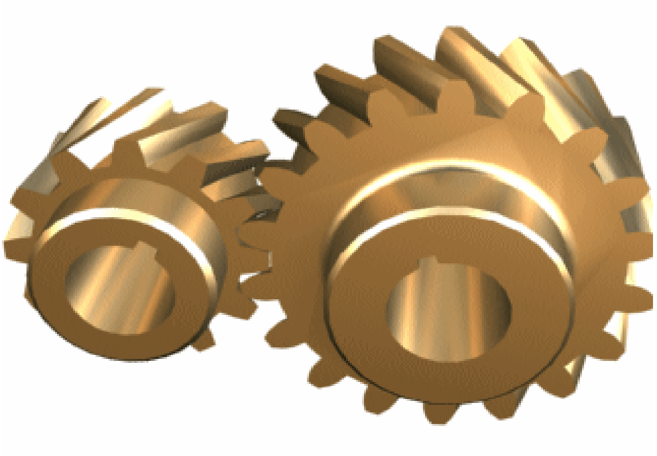
(3) Model checking temporal properties at run-time

- Extracting events while executing systems
- A monitor is designed in advance to check whether the trace violates the desired property
- Tools: Java PathExplorer, RiTHM, …
- **Interaction between systems and monitors incurs extra overhead**

Background & Motivation

Our approach:

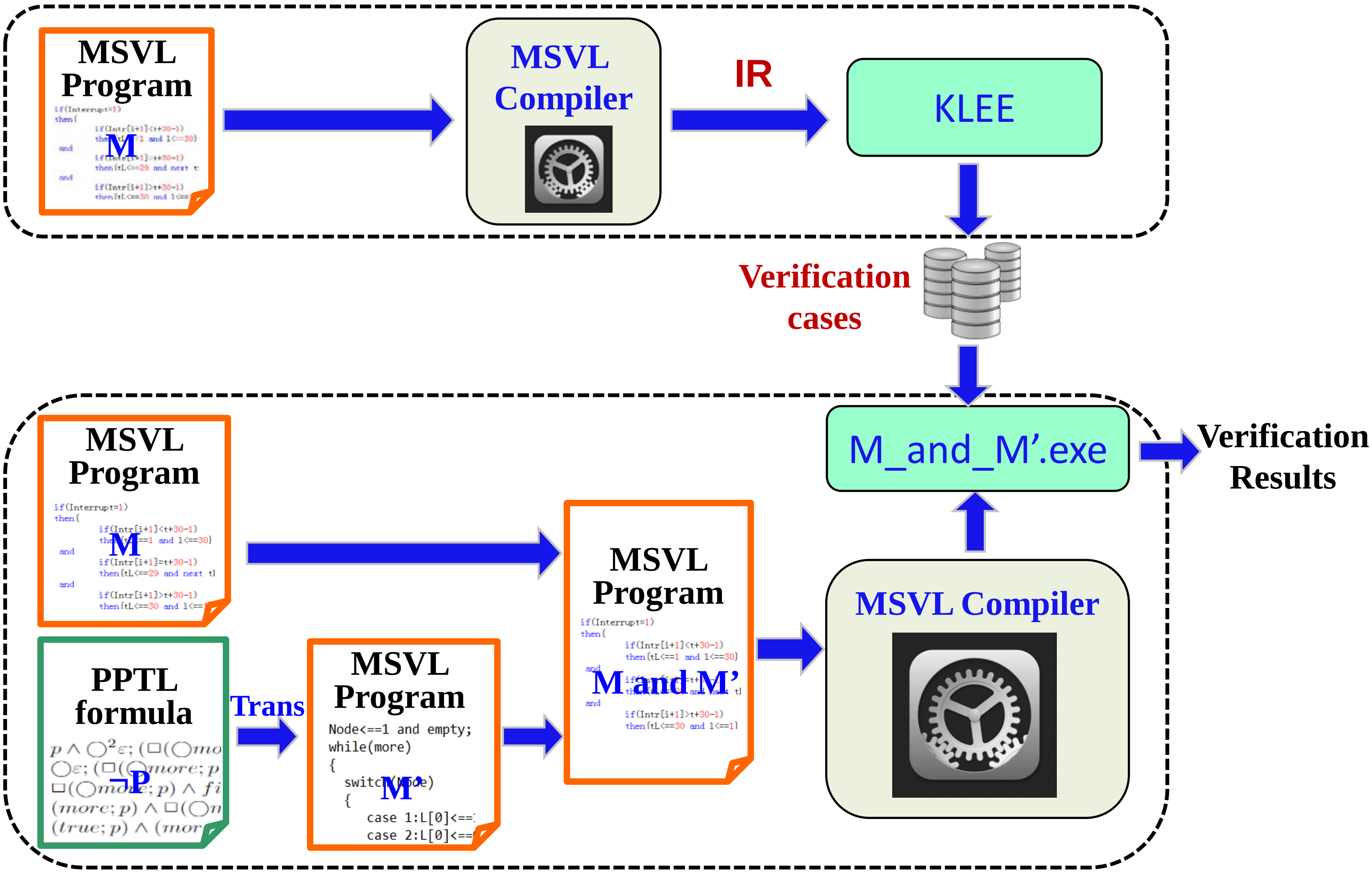
Verifying full-regular (temporal) properties of programs via dynamic program execution



Executing both the program and property

Background & Motivation

Approach Overview



Modeling Simulation and Verification Language

- Modeling Simulation and Verification Language (MSVL) is an executing subset of Projection Temporal Logic (PTL) with framing technique

Data Types:

(unsigned) int, float, (unsigned) char, string, array, pointer, struct, union

Syntax :

- Arithmetic expression

$$e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0 \ op \ e_1 (op ::= + \mid - \mid * \mid / \mid mod) \mid f(e_1, ..., e_n)$$

- Boolean expression

$$b ::= true \mid false \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \wedge b_1$$

Modeling Simulation and Verification Language

Two kinds of functions in MSVL programs

- External functions

  C standard library functions (strcat, strcmp, strlen, strcpy …)

- MSVL functions
  - MSVL standard library functions (int getline(int len,char s[ ]){…})
  - MSVL user-defined functions (void f(int x_1, x_2,…,x_n){…})

Two kinds of function calls in MSVL programs

- Black-box call (extern f(e_1,e_2,…,e_n))
- White-box call (only for MSVL functions: f(e_1,e_2,…,e_n))

Elementary Statements in MSVL

Termination:	empty	$\stackrel{def}{=}$	ε
State Assignment:	x<==e	$\stackrel{def}{=}$	$(x=e)\wedge p_x$
Assignment:	x:=e	$\stackrel{def}{=}$	$\bigcirc(x=e\wedge p_x)\wedge\bigcirc\varepsilon$
State Frame:	lbf(x)	$\stackrel{def}{=}$	$\neg p_x \rightarrow \exists b : (\ominus x = b \wedge x = b)$
Interval Frame:	frame(x)	$\stackrel{def}{=}$	$\Box(more \rightarrow \bigcirc lbf(x))$
Conjunction:	p and q	$\stackrel{def}{=}$	$p\wedge q$
Selection:	p or q	$\stackrel{def}{=}$	$p\vee q$
Next:	next p	$\stackrel{def}{=}$	$\bigcirc p$
Always:	always p	$\stackrel{def}{=}$	$\Box p$
Conditional:	if b then p else q	$\stackrel{def}{=}$	$(b \rightarrow p) \wedge (\neg b \rightarrow q)$
Local variable:	local x: p	$\stackrel{def}{=}$	$\exists x : p$
Projection:	(p1,…,pm) prj q	$\stackrel{def}{=}$	$(p_1, \ldots, p_m)\ \mathrm{prj}\ q$
Sequence:	p;q	$\stackrel{def}{=}$	$p;q$
While:	while b do p	$\stackrel{def}{=}$	$(p\wedge b)^* \wedge\Box(\varepsilon\rightarrow\neg b)$
Parallel:	p||q	$\stackrel{def}{=}$	$(p \wedge (q;true)) \vee (q \wedge (p;true)) \vee (p\wedge q)$
Await:	await(b)	$\stackrel{def}{=}$	$(frame(x_1)\wedge\cdots\wedge frame(x_h))\wedge\Box(\varepsilon\leftrightarrow b)$

MSVL

Normal form of MSVL programs

Present components

Future components

$$\phi \equiv (\bigvee_{i=1}^{k} \phi_{ei} \wedge \varepsilon) \vee (\bigvee_{j=1}^{h} \phi_{cj} \wedge \bigcirc \phi_{fj})$$

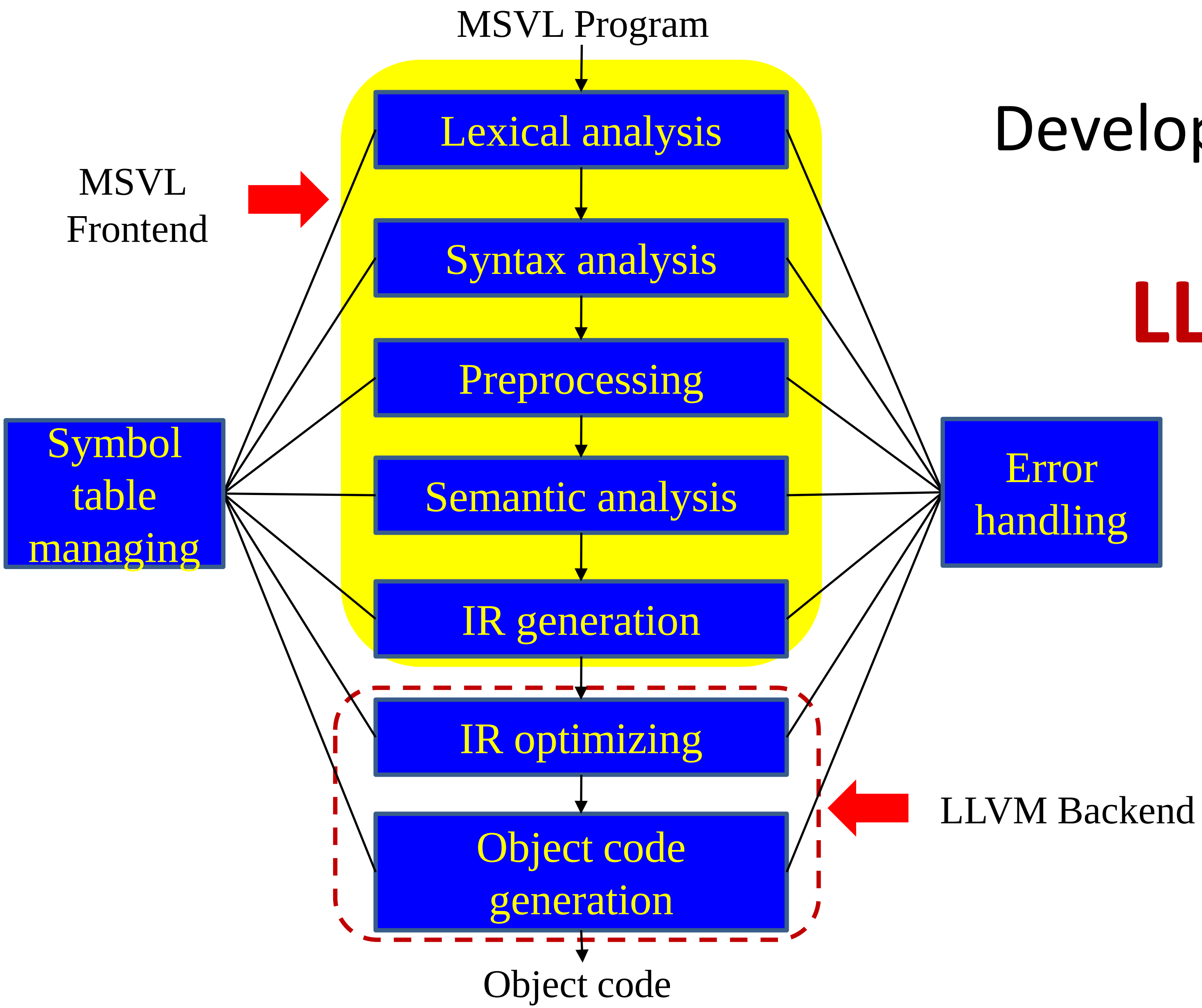
A program

Conjunction of state statements

An internal program where variables may refer to the previous states

Execution of MSVL programs is based on transforming programs into normal forms

MSVL Compiler



Developed based on **LLVM**

MSVL Compiler

Case Studies

[Dining philosophers problem]

[LTL2BA]

A program for translating LTL formulas to Büchi automata

[Simple CPU]

An adder including dereference, decode and execution

Propositional Projection Temporal Logic

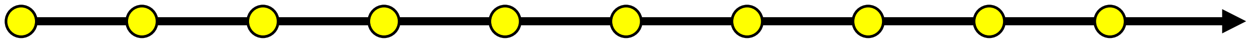
Propositional Projection Temporal Loigc (PPTL)

- Syntax

$$P ::= \quad p \mid \bigcirc P \mid \neg P \mid P \vee Q \mid (P_1, \cdots, P_m)\ prj\ P$$

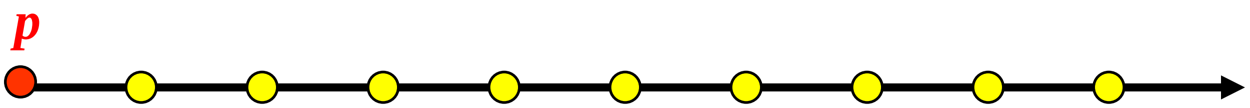
- Semantics

  An interval σ is a non-empty sequence of states, which can be finite or infinite.

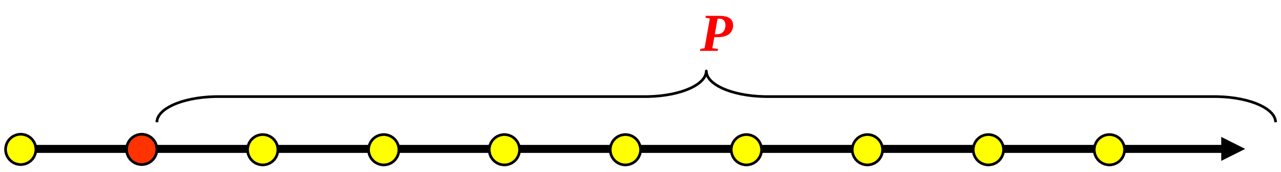


Propositional Projection Temporal Logic

- An interpretation is a triple $I = (\sigma, \boldsymbol{i}, \boldsymbol{j})$, where σ is an interval, $\boldsymbol{i}$ is an integer, and $\boldsymbol{j}$ an integer or $\boldsymbol{\omega}$.
- The satisfaction relation is inductively defined as follows:
  - $I \models p$ iff $s_i[p]$ = true, and $p \in Prop$ is an atomic proposition



  - $I \models \bigcirc P$ iff $i < j$ and $(\sigma, i+1, j) \models P$



  - $I \models \neg P$ iff $I \not\models P$
  - $I \models P \vee Q$ iff $I \models P$ or $I \models Q$

Propositional Projection Temporal Logic

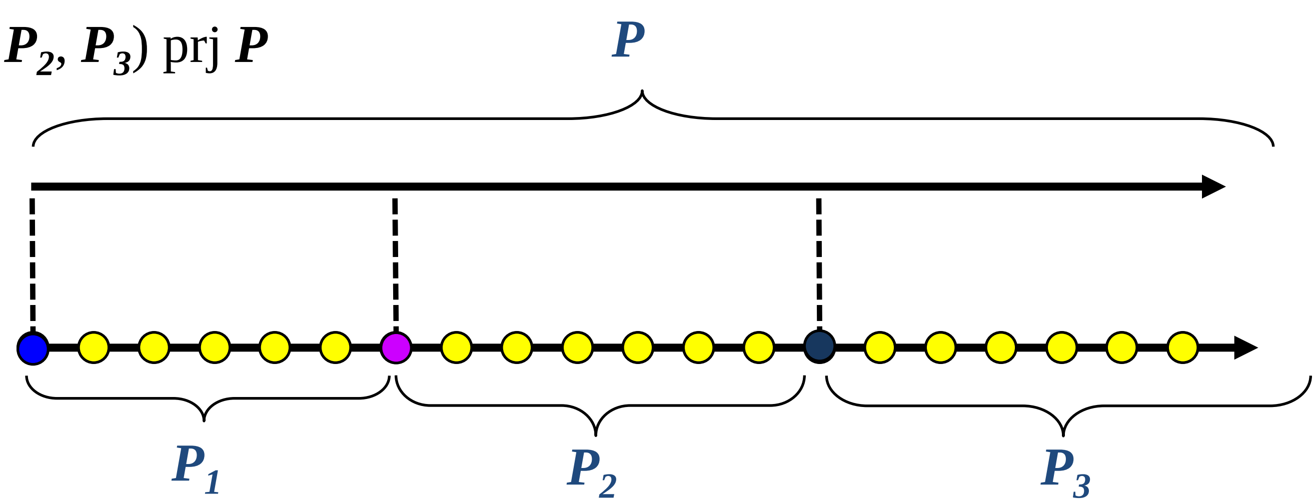
- $I \models (P_1, P_2, \ldots, P_m)$ prj P, if there exist integers $r_0 \leq r_1 \leq \cdots \leq r_m \leq j$ such that $(\sigma, r_{l-1}, r_l) \models P_l$, $1 \leq l \leq m$, and $(\sigma', 0, |\sigma'|) \models P$ for one of the following σ':

  (a) $r_m < j$ and $\sigma' = \sigma\downarrow(r_0, \ldots, r_m) \cdot \sigma(r_{m+1}, \ldots, j)$, or

  (b) $r_m = j$ and $\sigma' = \sigma\downarrow(r_0, \ldots, r_h)$ for some $0 \leq h \leq m$

$<s_0, s_1, s_2, s_3, s_4>\downarrow(0,0,2,2,2,3) = <s_0, s_2, s_3>$

(P_1, P_2, P_3) prj P



Derived formulas

$true \stackrel{\text{def}}{=} P \vee \neg P$ $\quad \varepsilon \stackrel{\text{def}}{=} \neg \bigcirc true$

$P;Q \stackrel{\text{def}}{=} (P,Q)prj\ \varepsilon$ $\quad \Diamond p \stackrel{\text{def}}{=} true;P$ $\quad \Box P \stackrel{\text{def}}{=} \neg\Diamond\neg P$

Normal Form of PPTL formulas

- A PPTL formula ***P*** is in normal form if,

$$P \equiv \bigvee_{i=1}^{l} P_{ei} \wedge \varepsilon \vee \bigvee_{j=1}^{t} P_{cj} \wedge \bigcirc P_{fj}$$

  - $\boldsymbol{P_{fj}}$ is a PPTL formula without disjuct being the main operator
  - $\boldsymbol{P_{ei}}$ and $\boldsymbol{P_{cj}}$ are true or state formulas of the form:

$$\bigwedge_{k=1}^{m} \dot{p}_k$$

$\dot{p}_k$ means p_k or $\neg p_k$ for each $p_k \in Prop$.

Theorem: Any PPTL formula can be equivalently transformed into its normal form.

Labeled Normal Form Graph

labeled normal form graphs (LNFG) are constructed based on normal form of PPTL formulas

- LNFG of a PPTL formula is a 4-tuple

$$LNFG = (CL, EL, V_0, \mathbb{L} = \{\mathbb{L}_1, \ldots, \mathbb{L}_k\})$$

  - *CL* : non-empty finite set of nodes
  - *EL*: set of directed edges among *CL*
  - V_0 : set of initial (root) nodes
  - $\mathbb{L}_i$: $\mathbb{L}_i \subseteq CL$, $1 \leq i \leq k$, set of nodes with l_i being the label.

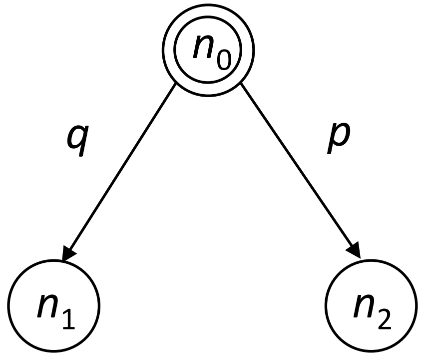
Inf(π): set of nodes which infinitely often occur in path π

A path is acceptable if it is finite, or infinite and all the nodes in ***Inf***(π) do not share a same label.

Labeled Normal Form Graph

- Example

LNFG of $\square(\bigcirc q) \wedge \square((p;q) \vee q)$



NF(n_0) =

$q \wedge \bigcirc(q \wedge \square(\bigcirc q) \wedge \square((p;q) \vee q)) \quad \vee$

$p \wedge \bigcirc(q \wedge \square(\bigcirc q) \wedge (true;q) \wedge \square((p;q) \vee q))$

n_0: $\square(\bigcirc q) \wedge \square((p;q) \vee q)$

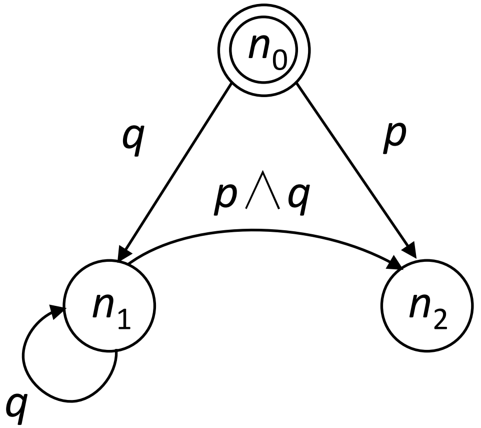
n_1: $q \wedge \square(\bigcirc q) \wedge \square((p;q) \vee q)$

n_2: $q \wedge \square(\bigcirc q) \wedge (\text{true};q) \wedge \square((p;q) \vee q)$

Labeled Normal Form Graph

- Example

LNFG of $\Box(\bigcirc q) \wedge \Box((p;q) \vee q)$



NF(n_1) =

$q \wedge \bigcirc(q \wedge \Box(\bigcirc q) \wedge \Box((p;q) \vee q)) \quad \vee$

$p \wedge q \wedge \bigcirc(q \wedge \Box(\bigcirc q) \wedge (true;q) \wedge \Box((p;q) \vee q))$

n_0: $\Box(\bigcirc q) \wedge \Box((p;q) \vee q)$

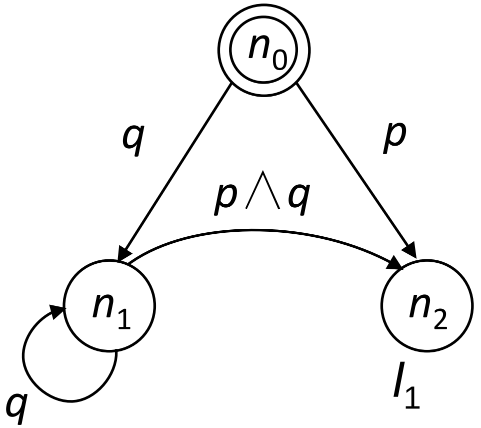
n_1: $q \wedge \Box(\bigcirc q) \wedge \Box((p;q) \vee q)$

n_2: $q \wedge \Box(\bigcirc q) \wedge (\text{true};q) \wedge \Box((p;q) \vee q)$

Labeled Normal Form Graph

- Example

LNFG of $\Box(\bigcirc q) \wedge \Box((p;q) \vee q)$



Rewrite n_2 with fin label

n_0: $\Box(\bigcirc q) \wedge \Box((p;q) \vee q)$

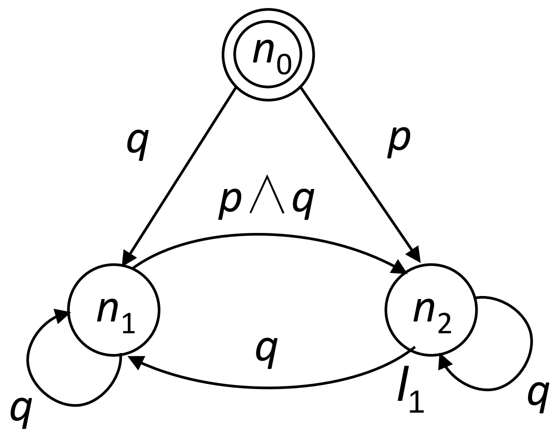
n_1: $q \wedge \Box(\bigcirc q) \wedge \Box((p;q) \vee q)$

n_2: $q \wedge \Box(\bigcirc q) \wedge (\mathrm{fin}(l_1);q) \wedge \Box((p;q) \vee q)$

Labeled Normal Form Graph

- Example

LNFG of $\square(\bigcirc q)\wedge\square((p;q)\vee q)$



Rewrite n_2 with fin label

$NF(n_2)$ =

$q\wedge\bigcirc(q\wedge\square(\bigcirc q)\wedge\square((p;q)\vee q))\quad\vee$

$q\wedge\bigcirc(q\wedge\square(\bigcirc q)\wedge(\mathrm{fin}(l_1);q)\wedge\square((p;q)\vee q))$

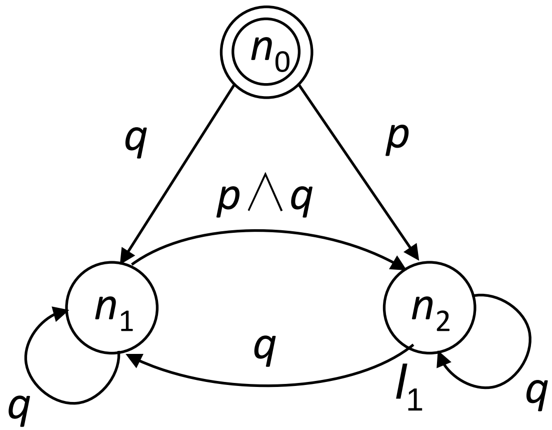
n_0: $\square(\bigcirc q)\wedge\square((p;q)\vee q)$

n_1: $q\wedge\square(\bigcirc q)\wedge\square((p;q)\vee q)$

n_2: $q\wedge\square(\bigcirc q)\wedge(\mathrm{fin}(l_1);q)\wedge\square((p;q)\vee q)$

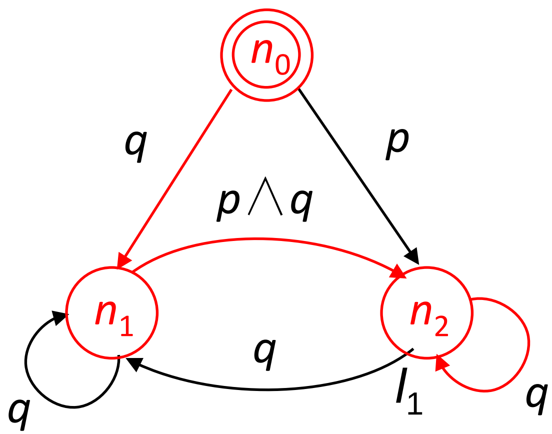
Labeled Normal Form Graph

- Example
  - LNFG of $\square(\bigcirc q) \wedge \square((p;q) \vee q)$
    - $\boldsymbol{CL} = \{n_0, n_1, n_2\}$
    - $\boldsymbol{EL} = \{ \langle n_0, q, n_1\rangle, \ \langle n_0, p, n_2\rangle, \ \langle n_1, q, n_1\rangle, \ \langle n_1, p \wedge q, n_2\rangle, \ \langle n_2, q, n_1\rangle, \ \langle n_2, q, n_2\rangle \}$
    - $\boldsymbol{V}_0 = \{n_0\}$
    - $\mathbb{L} = \{\mathbb{L}_1\}$ and $\mathbb{L}_1 = \{n_2\}$



Labeled Normal Form Graph

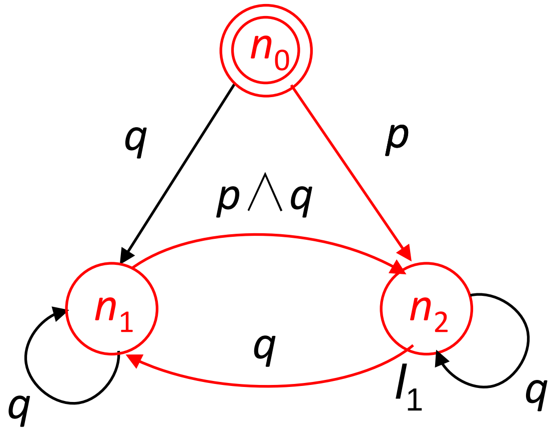
- Example
  - LNFG of $\square(\bigcirc q) \wedge \square((p;q) \vee q)$
    - $\boldsymbol{CL} = \{n_0, n_1, n_2\}$
    - $\boldsymbol{EL} = \{ \langle n_0, q, n_1 \rangle, \langle n_0, p, n_2 \rangle, \langle n_1, q, n_1 \rangle, \langle n_1, p \wedge q, n_2 \rangle, \langle n_2, q, n_1 \rangle, \langle n_2, q, n_2 \rangle \}$
    - $\boldsymbol{V}_0 = \{n_0\}$
    - $\mathbb{L} = \{\mathbb{L}_1\}$ and $\mathbb{L}_1 = \{n_2\}$



  - Path $\boldsymbol{\pi} = \langle n_0, q, n_1, p \wedge q, (n_2, q)^{\omega} \rangle$
    - Nodes that occur infinitely often have the same label l_1
    - **Unacceptable**

Labeled Normal Form Graph

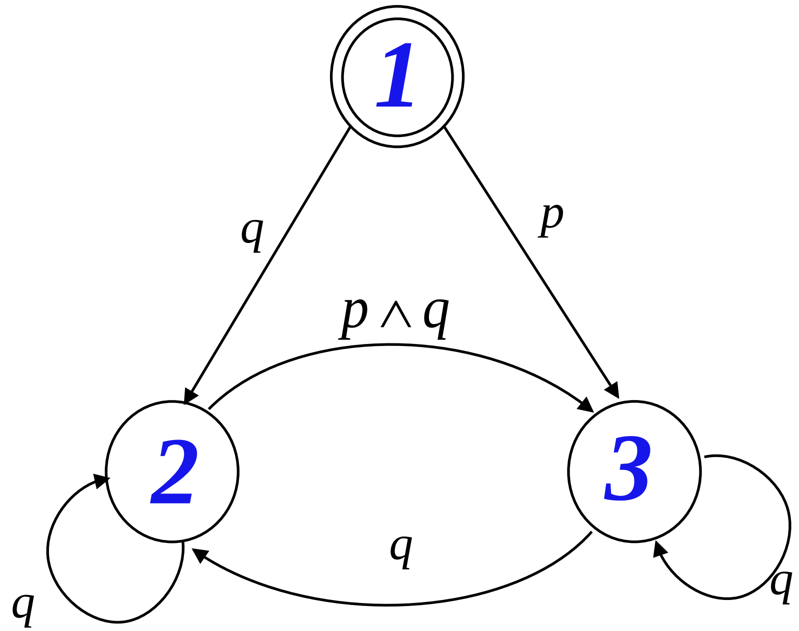
- Example
  - LNFG of $\Box(\bigcirc q) \wedge \Box((p;q) \vee q)$
    - $\boldsymbol{CL} = \{n_0, n_1, n_2\}$
    - $\boldsymbol{EL} = \{ \langle n_0, q, n_1\rangle, \ \langle n_0, p, n_2\rangle,$
      $\langle n_1, q, n_1\rangle, \ \langle n_1, p \wedge q, n_2\rangle,$
      $\langle n_2, q, n_1\rangle, \ \langle n_2, q, n_2\rangle \}$
    - $\boldsymbol{V}_0 = \{n_0\}$
    - $\mathbb{L} = \{\mathbb{L}_1\}$ and $\mathbb{L}_1 = \{n_2\}$



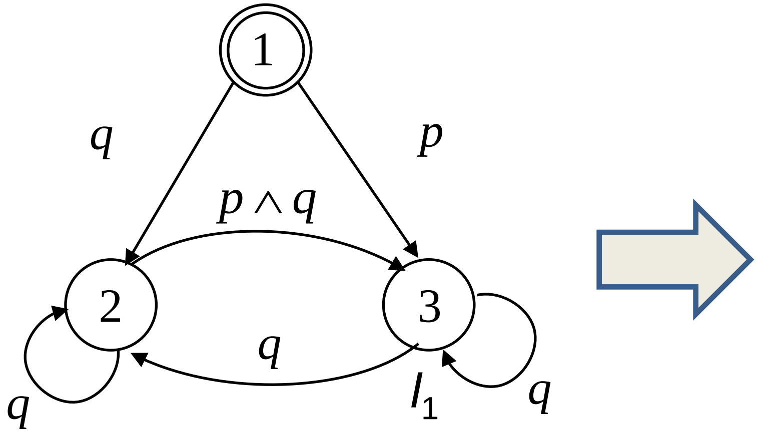
  - Path $\boldsymbol{\pi} = \langle n_0, q, n_1, p \wedge q, (n_2, q)^{\omega}\rangle$
    - Nodes that occur infinitely often have the same label l_1
    - **Unacceptable**
  - Path $\boldsymbol{\pi} = \langle n_0, p, (n_2, q, n_1, p \wedge q)^{\omega}\rangle$
    - Nodes that occur infinitely often do not have a same label
    - **Acceptable**

From PPTL to MSVL

Use a unique integer to represent each of nodes in the LNFG



From PPTL to MSVL

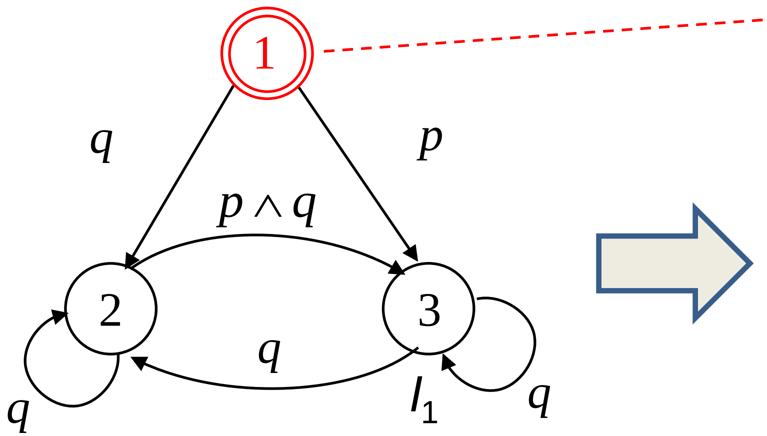


Program pattern

```
CuNode<==1 and
while(true){

}
```

Global Variable ***CuNode***: presenting the node explored at the current state.
The first node to be explored is a root node (***CuNode*** <==1)

From PPTL to MSVL



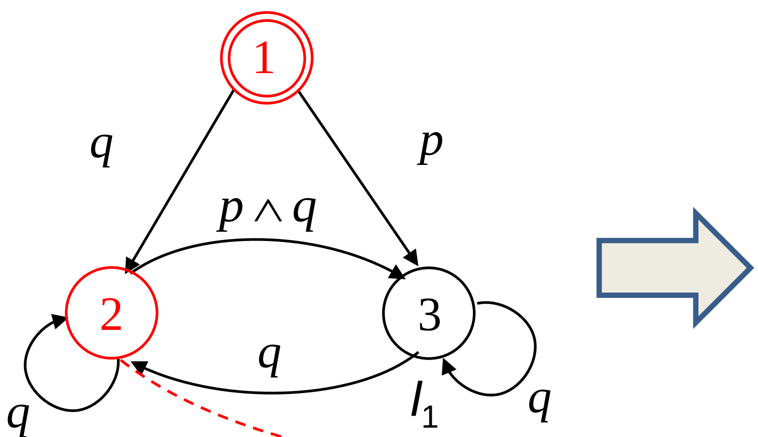
Program pattern

```
CuNode<==1 and
while(true){
  if(CuNode=1)then{


  }
}
```

For each node i, the following program pattern is created: **if(*CuNode=i*)then{ M }**

M is another program pattern w.r.t all the edges starting from i

From PPTL to MSVL



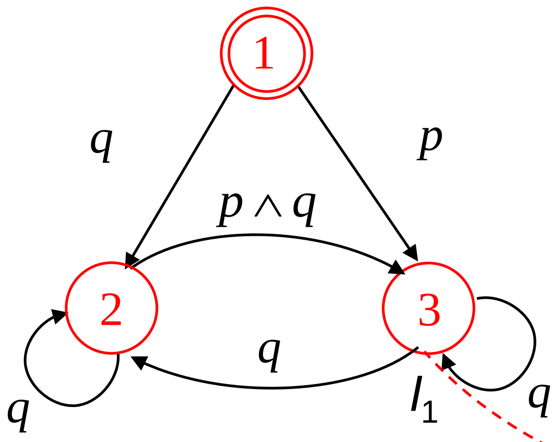
Program pattern

```
CuNode<==1 and
while(true){
  if(CuNode=1)then{


  }
  else{
    if(CuNode=2)then{


    }


  }
}
```

For each node i, the following program pattern is created: **if(*CuNode=i*)then{ M }**

M is another program pattern w.r.t all the edges starting from i

From PPTL to MSVL



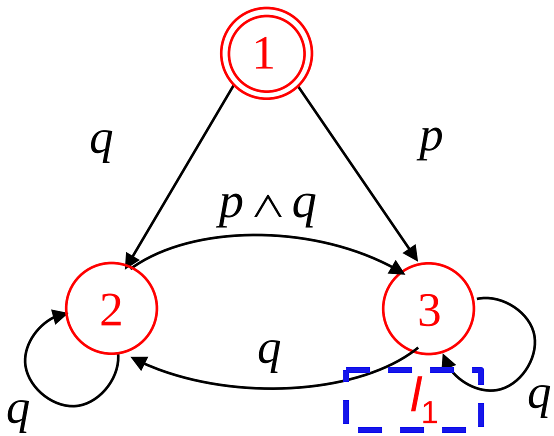
Program pattern

```
CuNode<==1 and
while(true){
   if(CuNode=1)then{


   }
   else{
      if(CuNode=2)then{


      }
       else{
          if(CuNode=3)then{


          }else{false}
       }
   }
}
```

For each node i, the following program pattern is created: **if(*CuNode=i*)then{ M }**

M is another program pattern w.r.t all the edges starting from i

From PPTL to MSVL



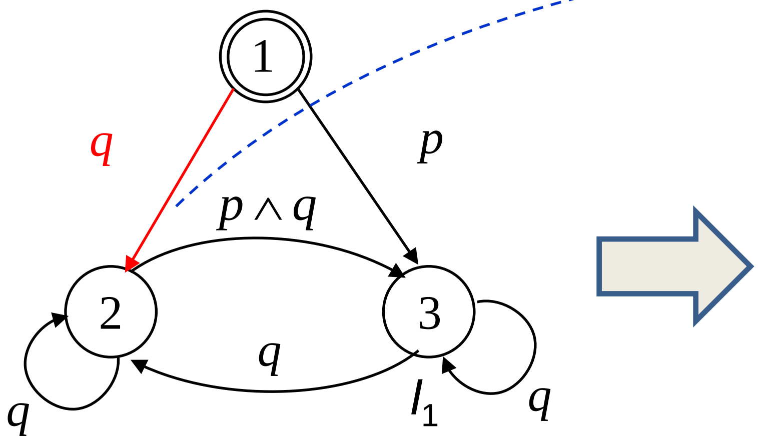
Program pattern

```
CuNode<==1 and
while(true){
   if(CuNode=1)then{


   }
   else{
      if(CuNode=2)then{


      }
       else{
          if(CuNode=3)then{ L={l_1}


          }else{false}
       }
   }
}
```

For each node i, the following program pattern is created: **if(*CuNode=i*)then{ M }**

M is another program pattern w.r.t all the edges starting from i

From PPTL to MSVL



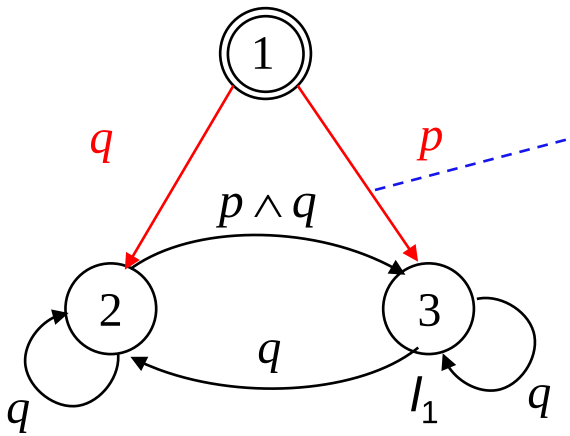
Program pattern

```
CuNode<==1 and
while(true){
  if(CuNode=1)then{
    if(q) then{CuNode:=2}else{false}

  }
  else{
    if(CuNode=2)then{


    }
     else{
      if(CuNode=3)then{ L={l1}


       }else{false}
     }
  }
}
```

For each edge from i to j with the label being p, program

if(p) then{$CuNode$:=j}else{false} is produced

From PPTL to MSVL



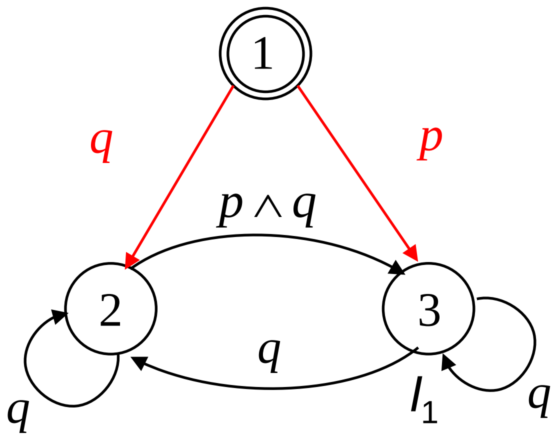
Program pattern

```
CuNode<==1 and
while(true){
   if(CuNode=1)then{
      if(q) then{CuNode:=2}else{false} or
      if(p) then{CuNode:=3}else{false}
   }
   else{
      if(CuNode=2)then{


      }
       else{
         if(CuNode=3)then{ L={l1}



         }else{false}
       }
   }
}
```

For each edge from i to j with the label being p, program

if(*p*) then{*CuNode*:=*j*}else{false} is produced

From PPTL to MSVL



Program pattern

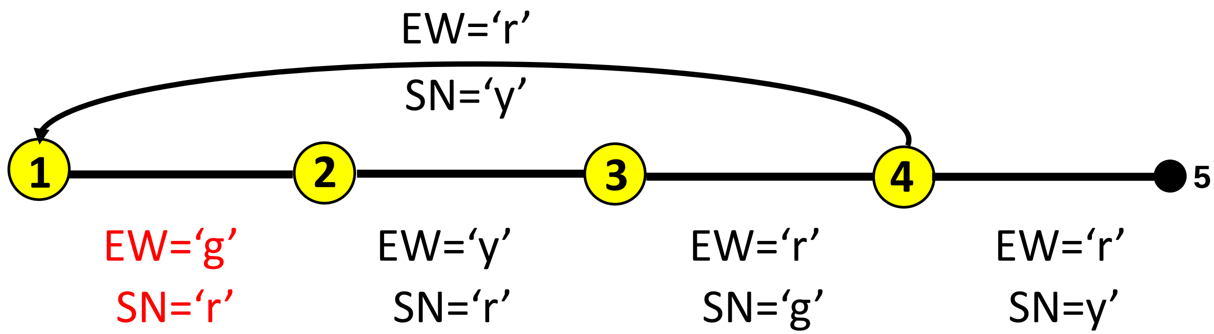
```
CuNode<==1 and
while(true){
   if(CuNode=1)then{
       if(q) then{CuNode:=2}else{false} or
       if(p) then{CuNode:=3}else{false}
   }
   else{
      if(CuNode=2)then{
         if(q) then{CuNode:=2}else{false}  or
         if(p and q) then{CuNode:=3}else{false}
      }
       else{
          if(CuNode=3)then{ L={l1} and (
              if(q) then{CuNode:=2}else{false} or
              if(q) then{CuNode:=3}else{false} )
          }else{false}
        }
   }
}
```

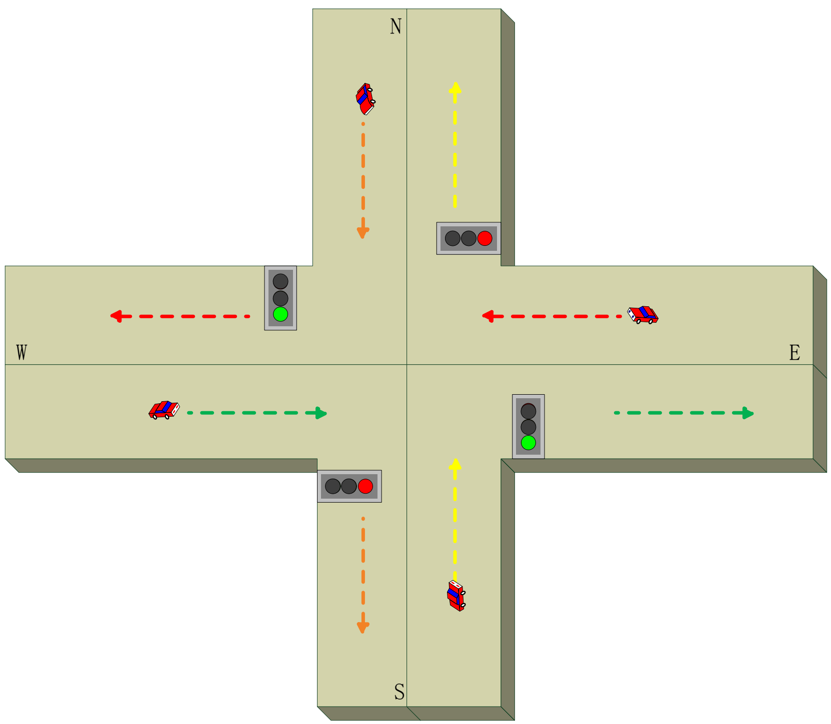
For each edge from i to j with the label being p, program

if(*p*) then{*CuNode*:=*j*}else{false} is produced

Verification as Dynamic Program Execution

Example: Traffic Light

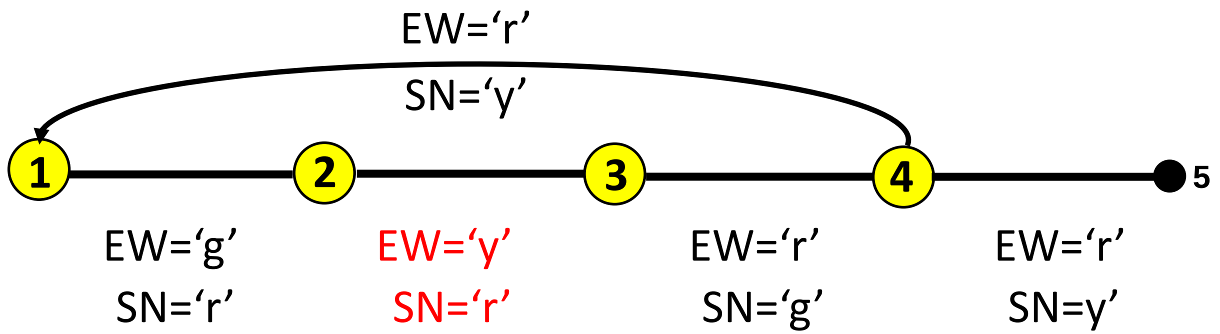


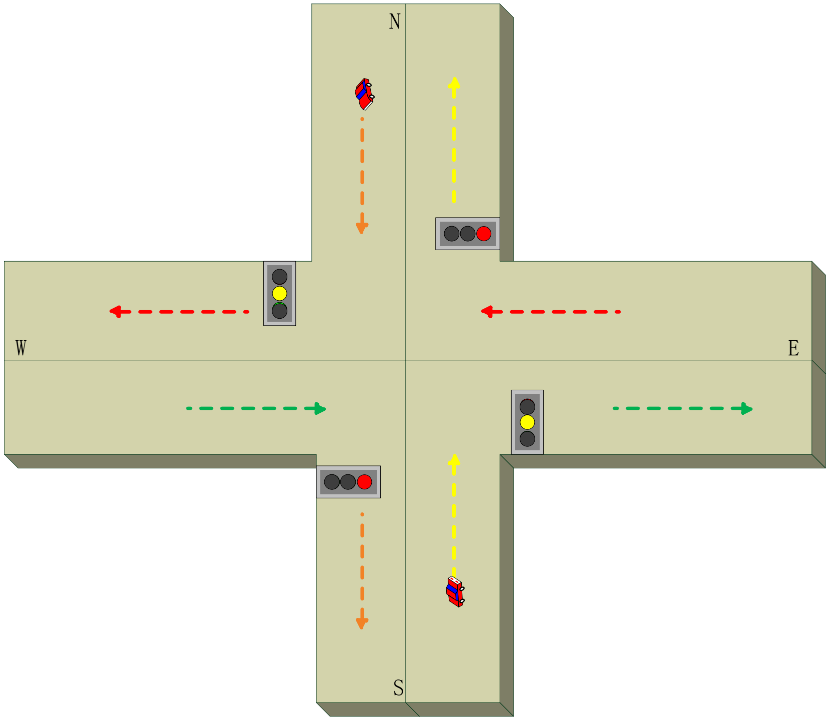



```
frame(EW,SN,f) and
( char EW and char SN and int f<==0
 while(!f)
  {
     EW<=='g' and SN<=='r' and skip;
     EW<=='y' and SN<=='r' and skip;
     EW<=='r' and SN<=='g' and skip;
     EW<=='r' and SN<=='y' and
     (f<==0 and skip  or f<==1 and empty)
 } )
```

Verification as Dynamic Program Execution

Example: Traffic Light

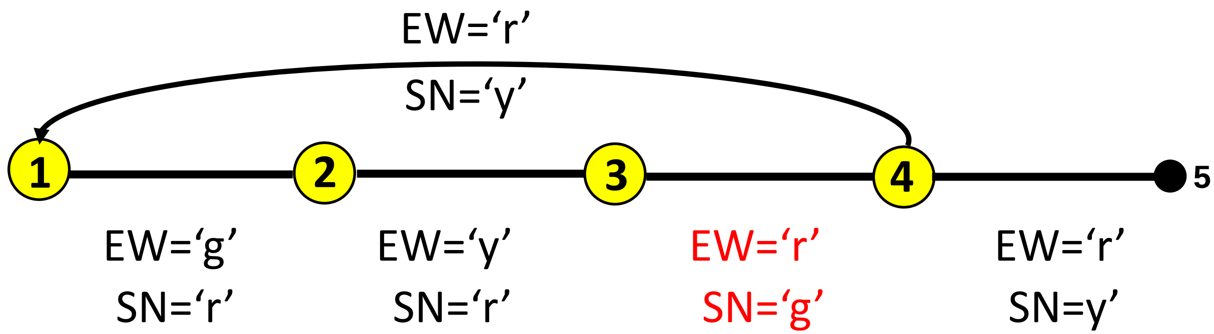


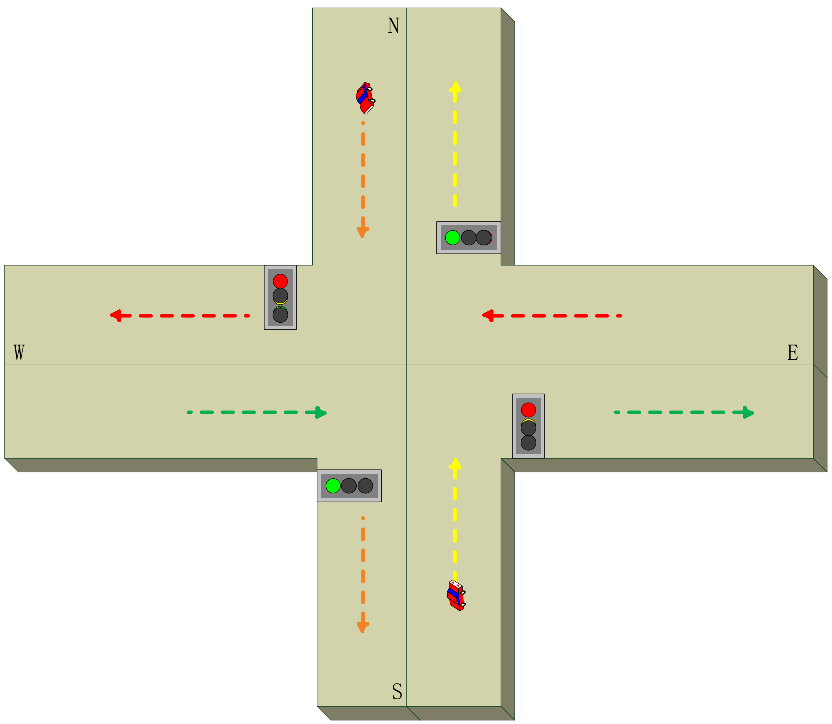



```
frame(EW,SN,f) and
( char EW and char SN and int f<==0
  while(!f)
   {
      EW<=='g' and SN<=='r' and skip;
      EW<=='y' and SN<=='r' and skip;
      EW<=='r' and SN<=='g' and skip;
      EW<=='r' and SN<=='y' and
      (f<==0 and skip  or f<==1 and empty)
  } )
```

Verification as Dynamic Program Execution

Example: Traffic Light

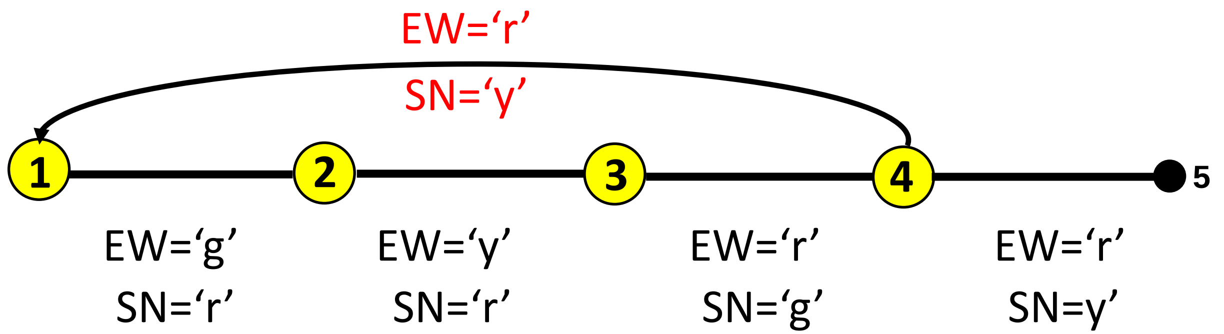


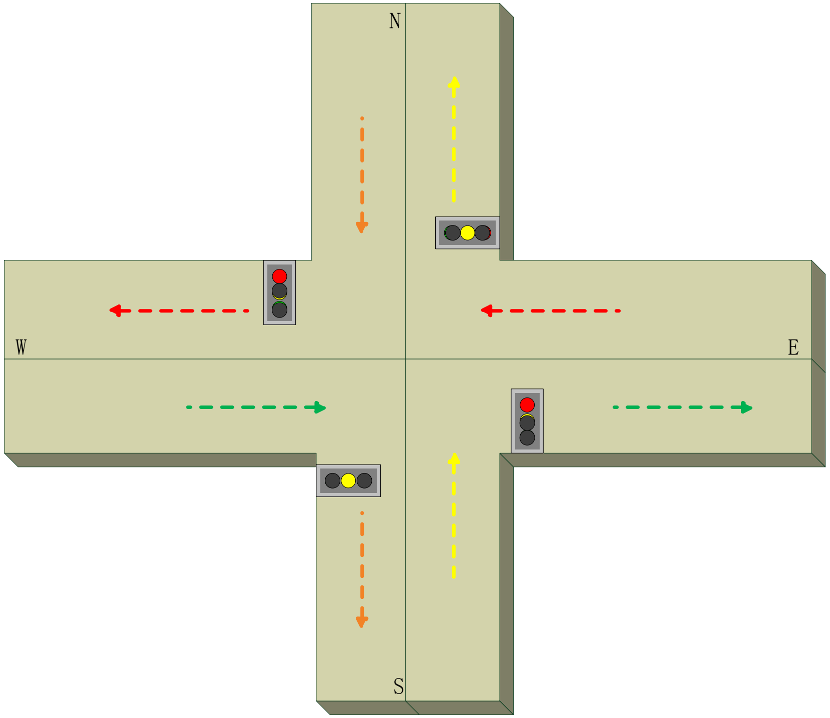



```
frame(EW,SN,f) and
( char EW and char SN and int f<==0
  while(!f)
  {
     EW<=='g' and SN<=='r' and skip;
     EW<=='y' and SN<=='r' and skip;
     EW<=='r' and SN<=='g' and skip;
     EW<=='r' and SN<=='y' and
     (f<==0 and skip  or f<==1 and empty)
  } )
```

Verification as Dynamic Program Execution

Example: Traffic Light






```
frame(EW,SN,f) and
( char EW and char SN and int f<==0
 while(!f)
  {
     EW<=='g' and SN<=='r' and skip;
     EW<=='y' and SN<=='r' and skip;
     EW<=='r' and SN<=='g' and skip;
     EW<=='r' and SN<=='y' and
     (f<==0 and skip  or f<==1 and empty)
 } )
```

Verification as Dynamic Program Execution

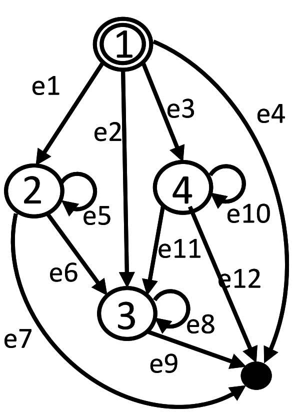
Whether M violates P?

Program *M*

```
frame(EW,SN,f) and
( char EW and char SN and int f<==0 and
  while(!f)
  {
    EW<==‘g’ and SN<==‘r’ and skip;
    EW<==‘y’ and SN<==‘r’ and skip;
    EW<==‘r’ and SN<==‘g’ and skip;
    EW<==‘r’ and SN<==‘y’ and
    (f<==0  and skip or f<==1 and empty)
  })
```

Desired Property *P*

$$\Box(\neg((EW=\text{‘g’})\wedge(SN=\text{‘g’})))\wedge\Box(\neg((EW=\text{‘y’})\wedge(SN=\text{‘y’})))$$



e1: true
e2: (EW=‘g’∧SN=‘g’)∨(EW=‘y’∧SN=‘y’)
e3: true
e4: (EW=‘g’∧SN=‘g’)∨(EW=‘y’∧SN=‘y’)
e5: true
e6: EW=‘g’∧SN=‘g’
e7: EW=‘g’∧SN=‘g’
e8: true
e9: true
e10: true
e11: EW=‘y’∧SN=‘y’
e12: EW=‘y’∧SN=‘y’

MSVL Program

M

```
frame(EW,SN,f) and
( char EW and char SN and int f<==0 and
  while(!f)
  {
    EW<==‘g’ and SN<==‘r’ and skip;
    EW<==‘y’ and SN<==‘r’ and skip;
    EW<==‘r’ and SN<==‘g’ and skip;
    EW<==‘r’ and SN<==‘y’ and
    (f<==0  and skip or f<==1 and empty)
  })
```

and

$\neg P$

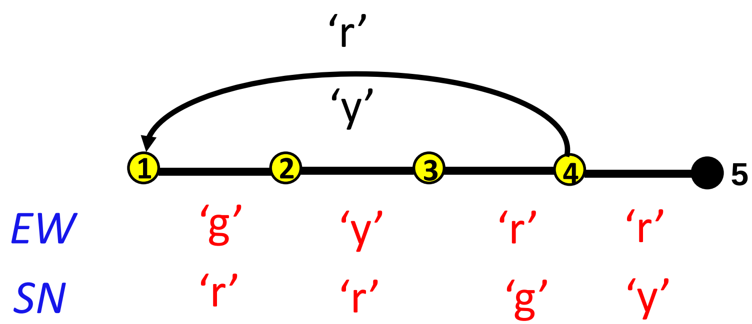
```
CuNode<==1 and
while(more)
{  if(CuNode=1)then{
      CuNode:=2 or
      if((EW=‘g’ and SN=‘g’) or (EW=‘y’ and SN=‘y’))
      then{CuNode:=3}else{false}  or  CuNode:=4 }
   else
   {  if(CuNode=2)then{  L={l1} and ( CuNode:=2 or
         if(EW=‘g’ and SN=‘g’) then{Node:=3}else {false})}
      else{
         if(CuNode=3)then{CuNode:=3}
         else{
            if(CuNode=4) then{  L={l2} and ( CuNode:=4 or
               if(EW=‘y’ and SN=‘y’) then {CuNode:=3}else{false})
            }else{false}}}}
};
if(CuNode=1) then{…}
else {  if(CuNode=2)then{…}
        else{ if(CuNode=3)then{…}
           else{ if(CuNode=4)then{…}else{false}}}
}
```

Verification as Dynamic Program Execution

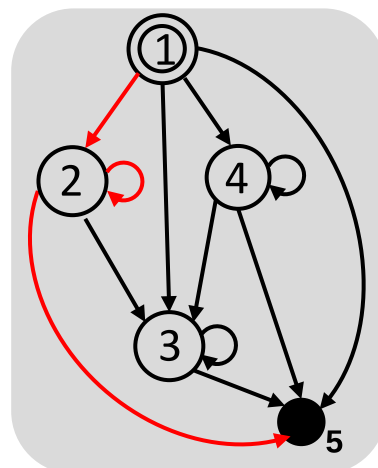
An **execution** of an MSVL program is sequences of states

$$\sigma = < s_0, s_1, \dots >$$

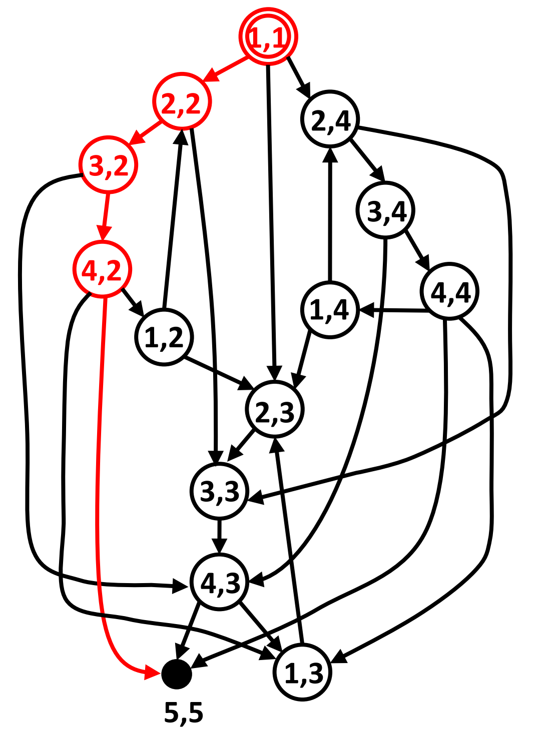
A finite execution in ***M***



A finite path in LNFG of ¬***P***

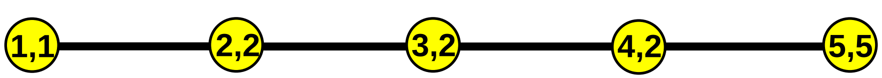


All executions in ***M and*** ¬***P***



A finite execution in ***M and*** ¬***P***

Finite:



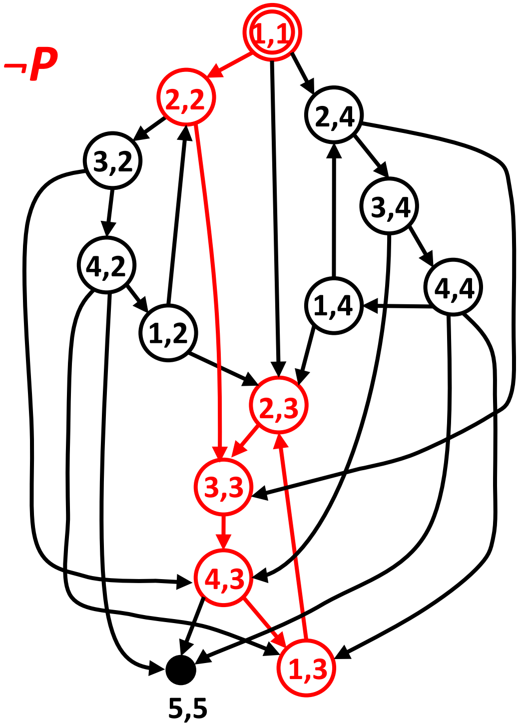
EW	'g'	'y'	'r'	'r'
SN	'r'	'r'	'g'	'y'

Verification as Dynamic Program Execution

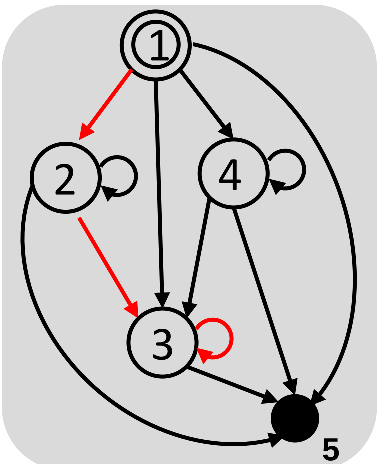
An **execution** of an MSVL program is sequences of states

$$\sigma = \langle \boldsymbol{s}_0, \boldsymbol{s}_1, \ldots \rangle$$

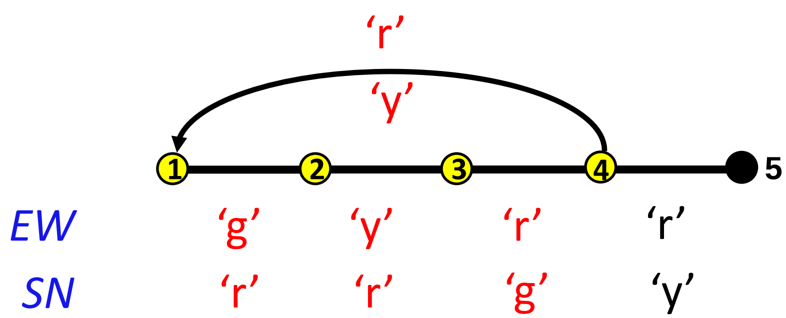
All executions in ***M and ¬P***



An infinite path in LNFG of ¬***P***

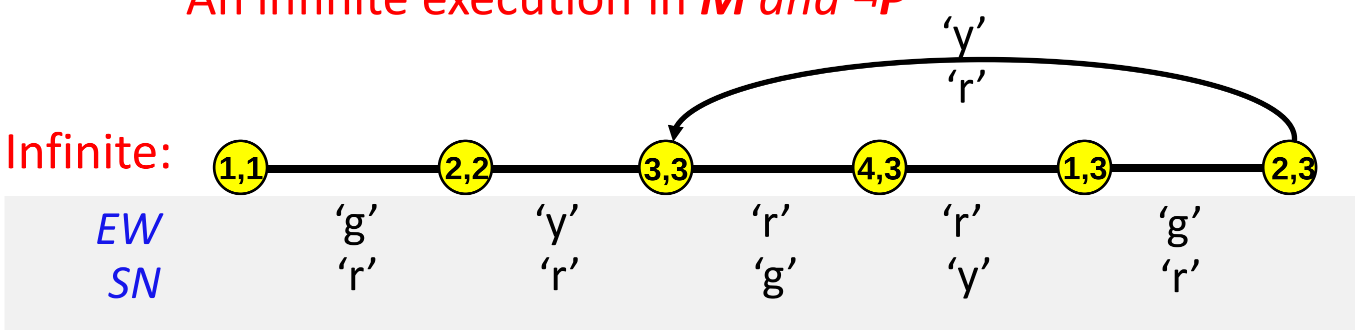


An infinite execution in ***M***



An infinite execution in ***M and ¬P***

Infinite:

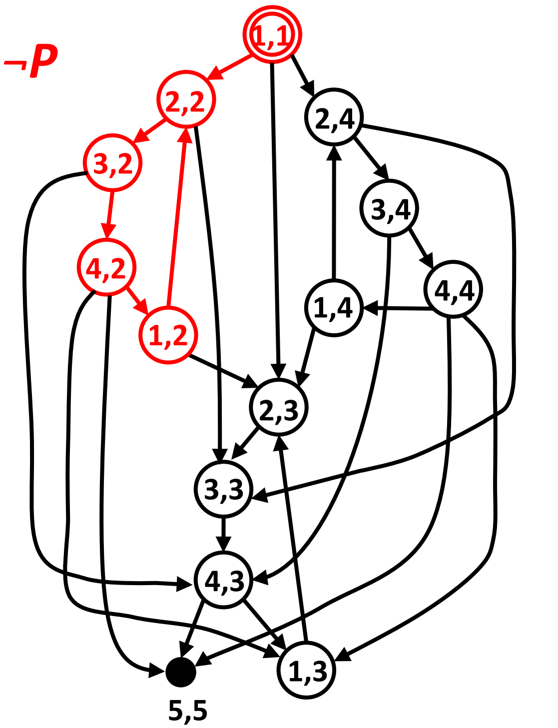


Verification as Dynamic Program Execution

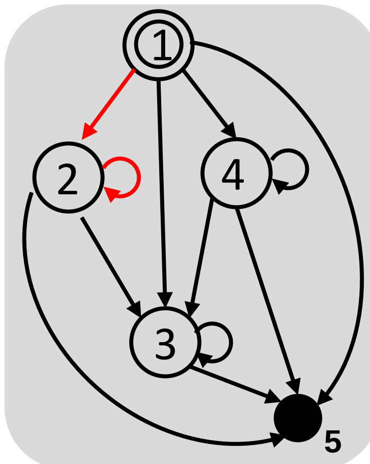
An **execution** of an MSVL program is sequences of states

$$\sigma = < s_0, s_1, \ldots >$$

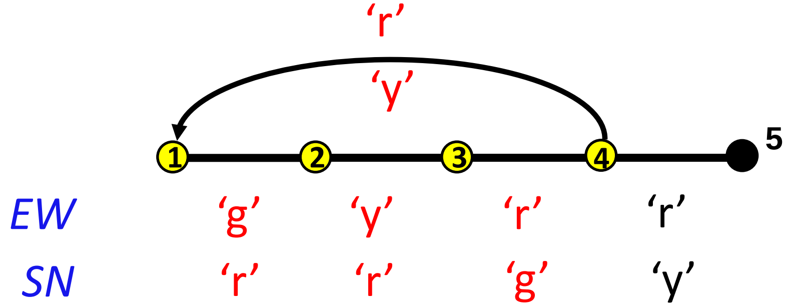
All executions in ***M and ¬P***



An infinite path in LNFG of ¬***P***

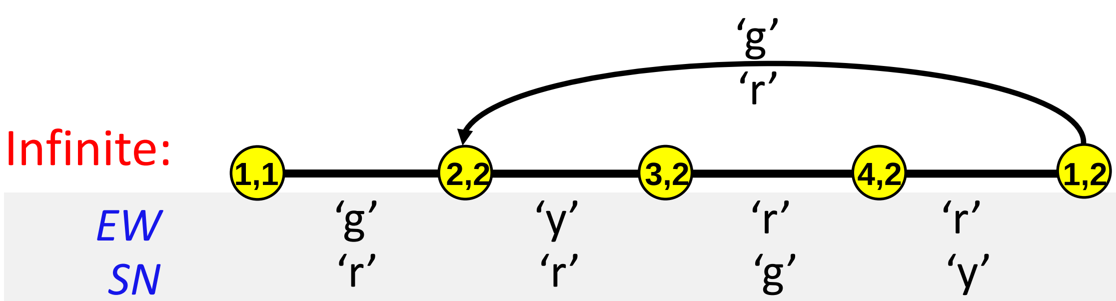


An infinite execution in ***M***



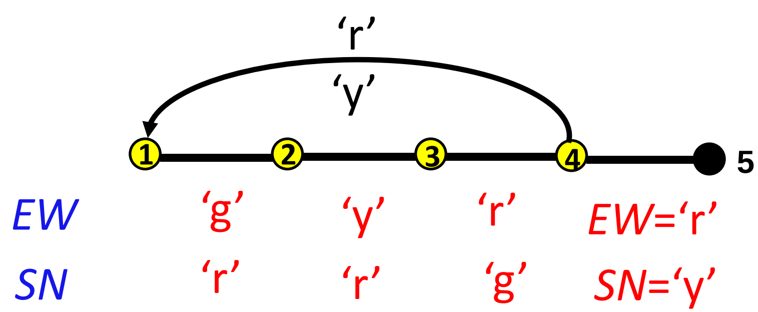
An infinite execution in ***M and ¬P***

Infinite:

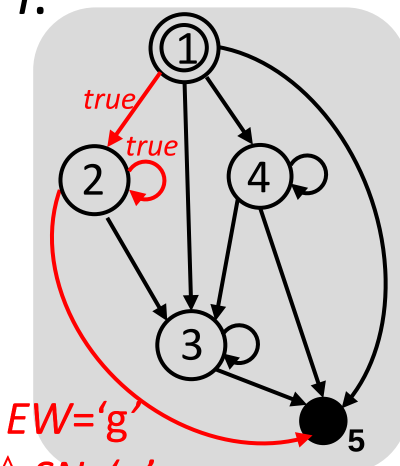


Verification as Dynamic Program Execution

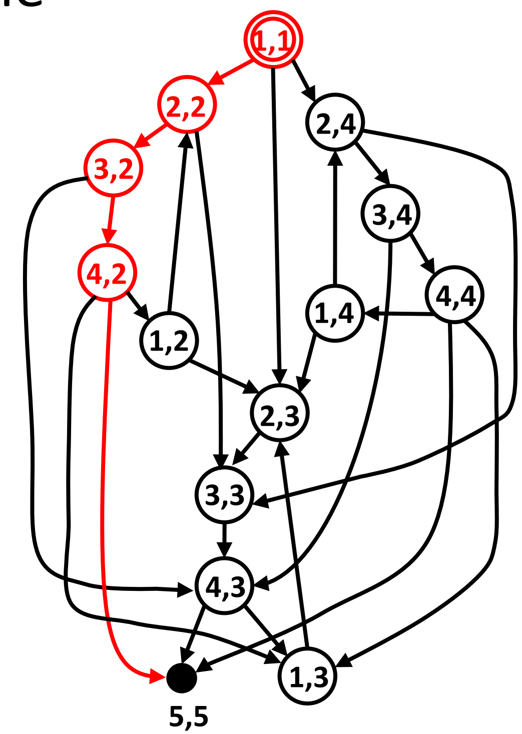
Feasible Execution: An execution $\sigma =< s_0, s_1, \ldots >$ is feasible if for all i, $check^i \equiv true$, where $check^i$ is a boolean variable representing whether a program state satisfies the desired state formula at state *i*.



$\boxed{EW=\text{'r'} \wedge SN=\text{'y'}} \wedge \boxed{EW=\text{'g'} \wedge SN=\text{'g'}} \equiv \text{false}$

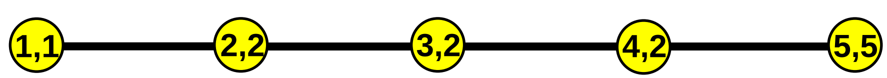


$EW=\text{'g'} \wedge SN=\text{'g'}$



Feasibility checking(finite)

Finite:

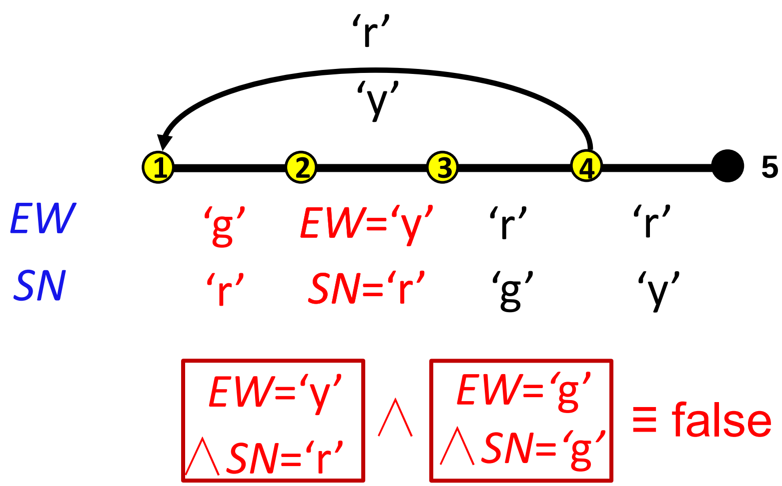


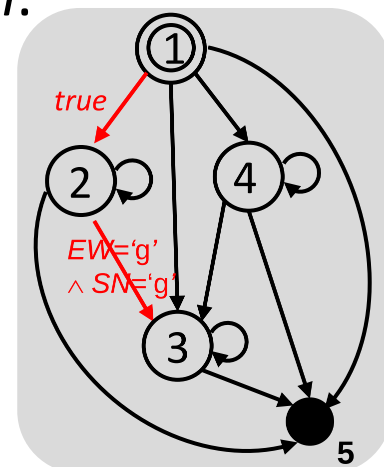
EW	'g'	'y'	'r'	'r'
SN	'r'	'r'	'g'	'y'
check	true	true	true	false

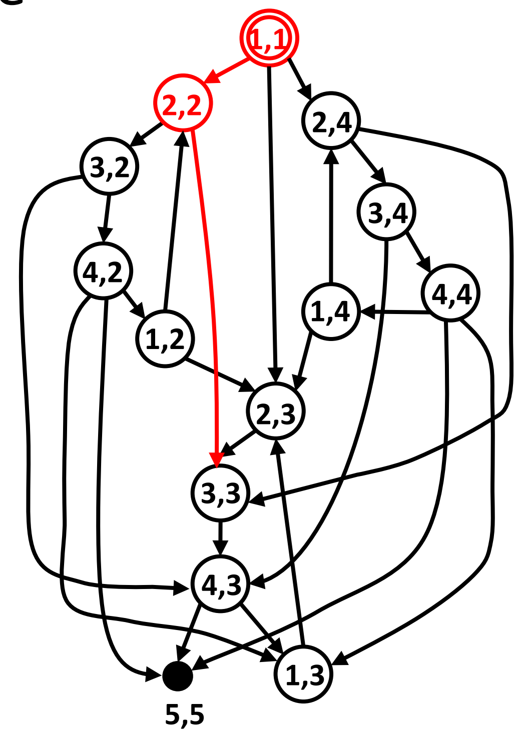
Infeasible

Verification as Dynamic Program Execution

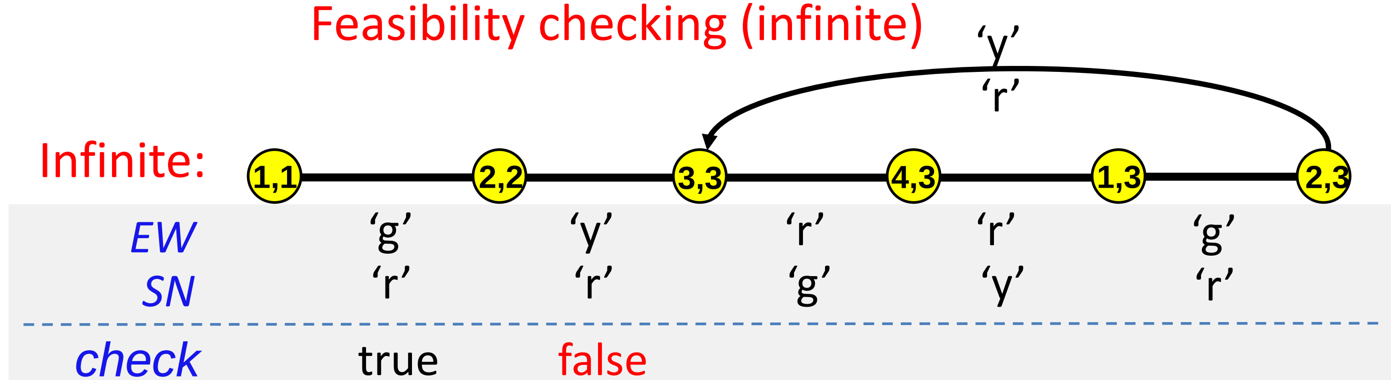
Feasible Execution: An execution $\sigma =< s_0, s_1, \ldots >$ is feasible if for all i, $check^i \equiv true$, where $check^i$ is a boolean variable representing whether a program state satisfies the desired state formula at state *i*.







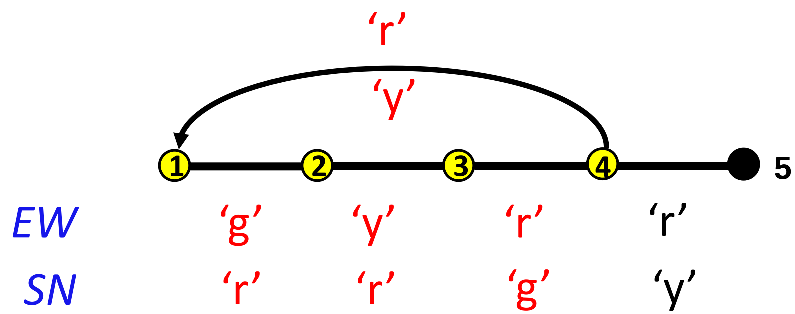
Feasibility checking (infinite)

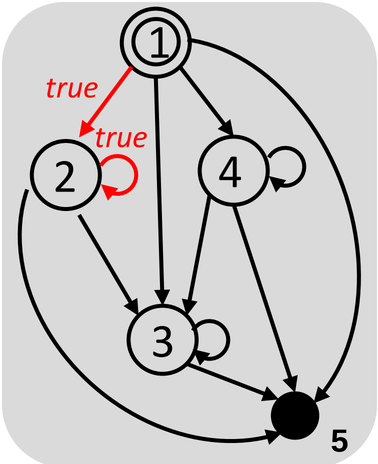


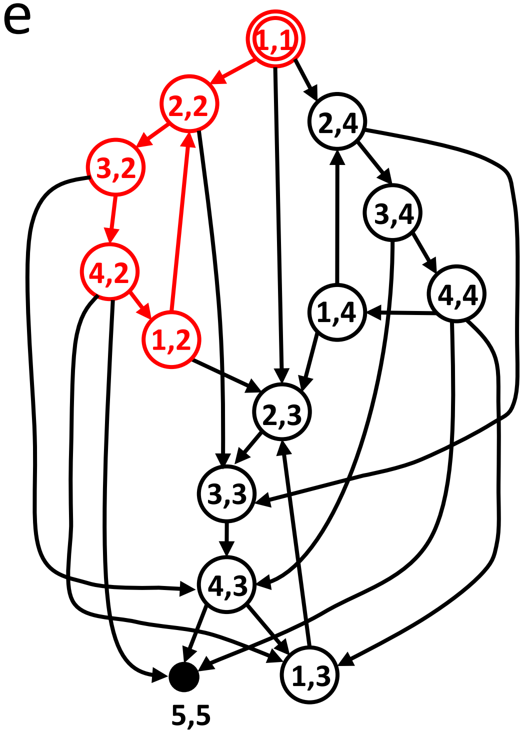
Infeasible

Verification as Dynamic Program Execution

Feasible Execution: An execution $\sigma =< s_0, s_1, \ldots >$ is feasible if for all i, $check^i \equiv true$, where $check^i$ is a boolean variable representing whether a program state satisfies the desired state formula at state i.

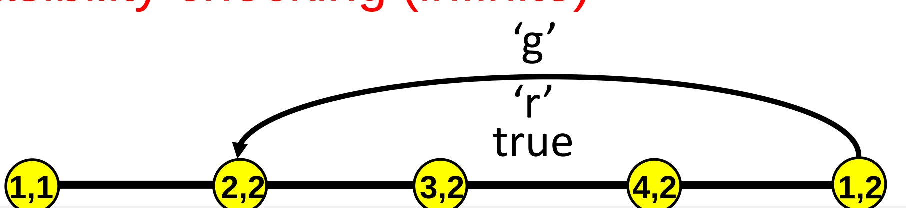






Feasibility checking (infinite)

Infinite:

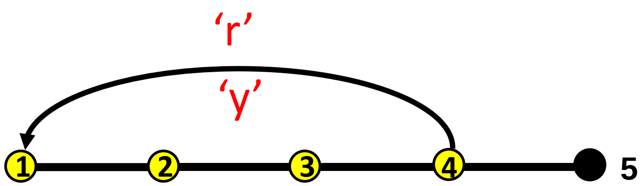


EW	'g'	'y'	'r'	'r'
SN	'r'	'r'	'g'	'y'
check	true	true	true	true

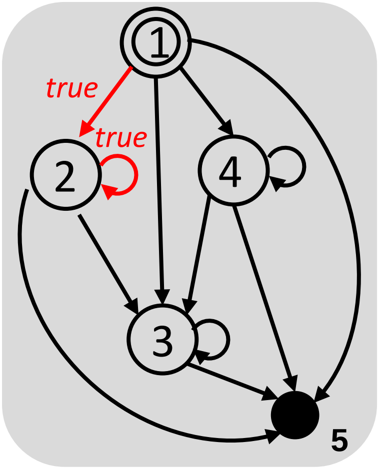
Feasible

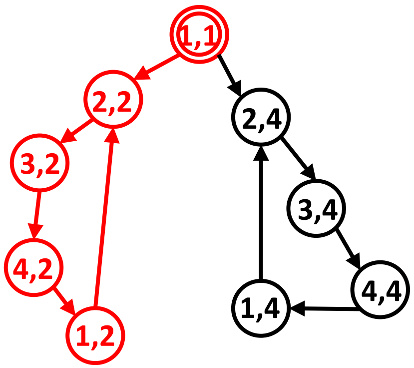
Verification as Dynamic Program Execution

Feasible Execution: An execution $\sigma = < s_0, s_1, \ldots >$ is feasible if for all i, $check^i \equiv true$, where $check^i$ is a boolean variable representing whether a program state satisfies the desired state formula at state *i*.



EW	'g'	'y'	'r'	'r'
SN	'r'	'r'	'g'	'y'

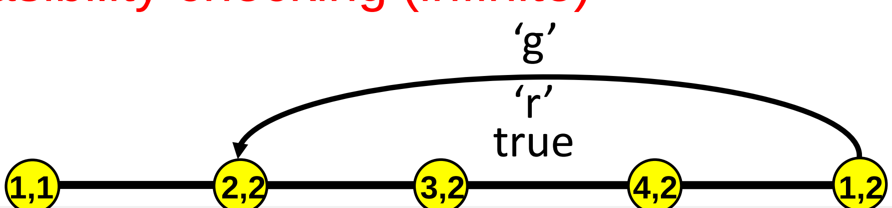




All feasible executions in ***M and ¬P***

Feasibility checking (infinite)

Infinite:



EW	'g'	'y'	'r'	'r'
SN	'r'	'r'	'g'	'y'
check	true	true	true	true

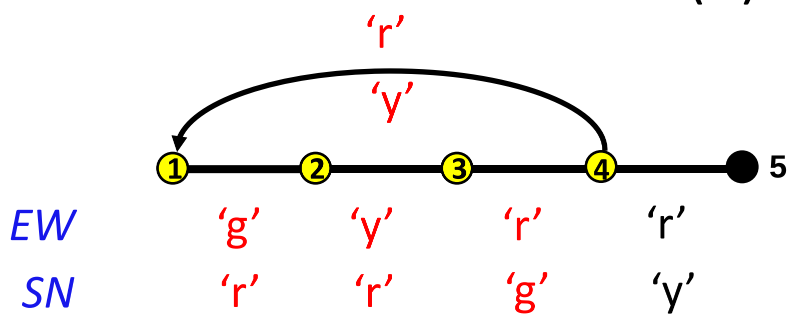
Feasible

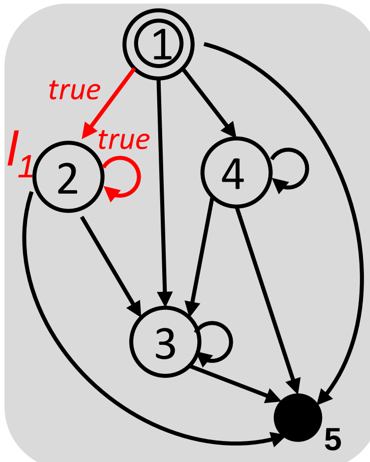
Verification as Dynamic Program Execution

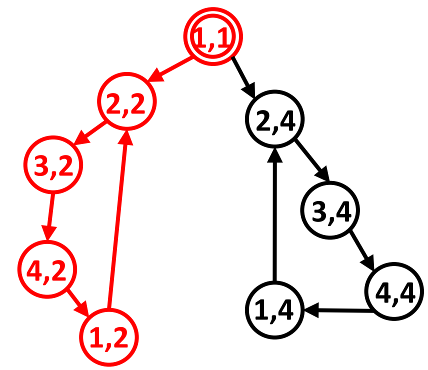
A feasible execution $\sigma = \langle s_0, s_1, \ldots \rangle$ is **acceptable** if

(1) σ is finite; or

(2) σ is infinite and no lables are shared by all the states in Inf (σ)

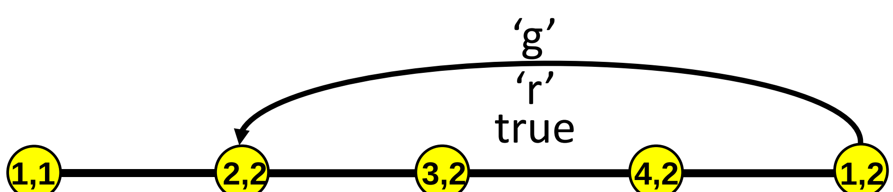






Whether a feasible path is acceptable?

Infinite:



EW		'g'	'y'	'r'	'r'
SN		'r'	'r'	'g'	'y'
check		true	true	true	true

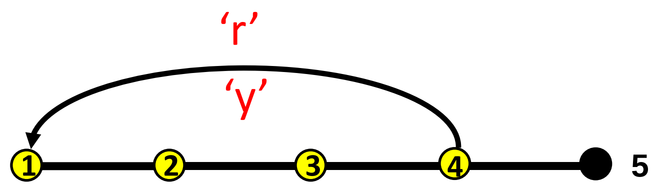
label ϕ $\{l_1\} \cap \{l_1\} \cap \{l_1\} \cap \{l_1\} = \{l_1\}$

l_1 is shared by all the states in Inf (σ)

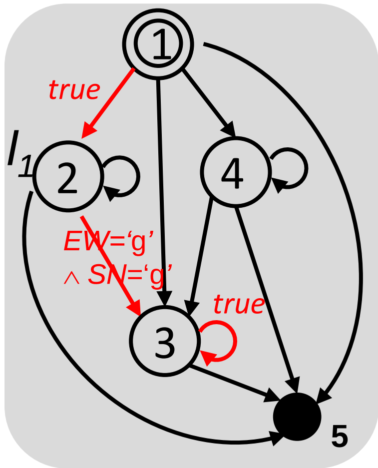
Unacceptable

Verification as Dynamic Program Execution

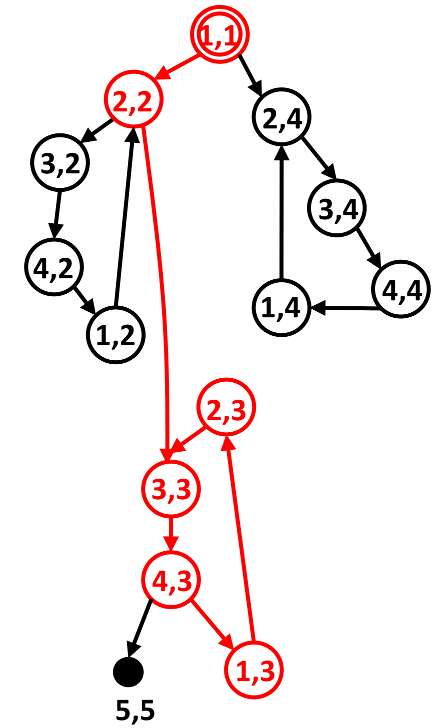
Suppose the model is modified as follows:



EW	'g'	'g'	'r'	'r'
SN	'r'	'g'	'g'	'y'

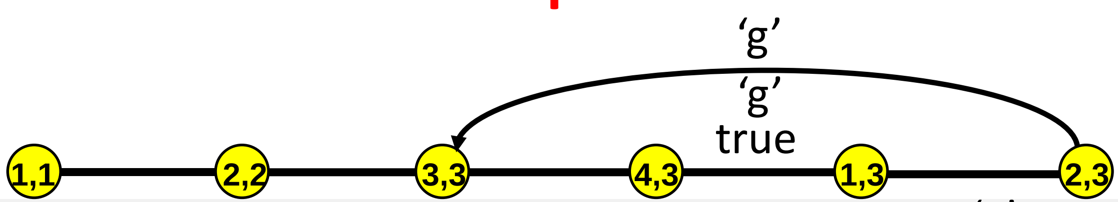


All feasible executions in ***M** and ¬**P***



A counterexample is found!

Infinite:



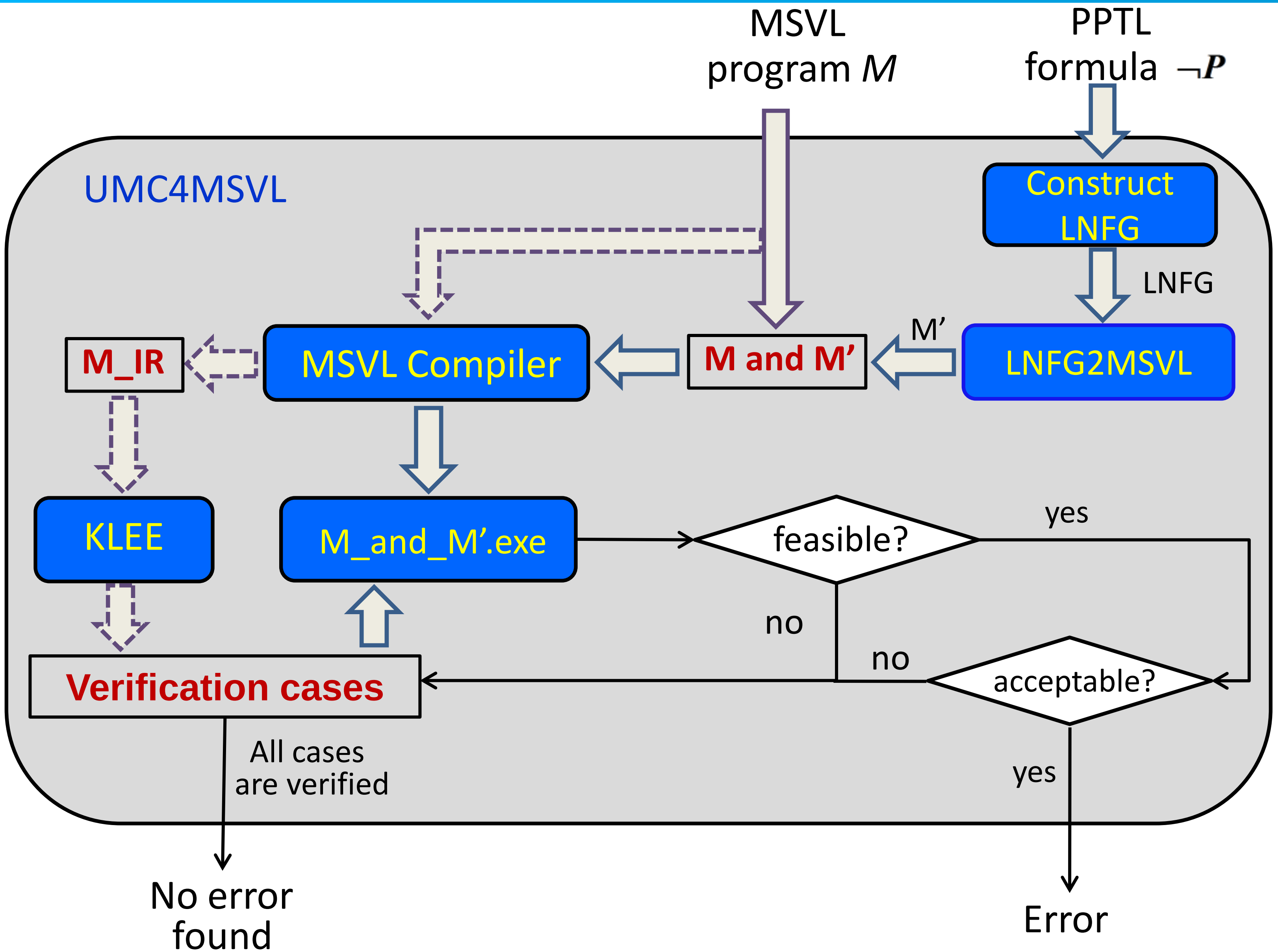
EW	'g'	'g'	'r'	'r'	'g'
SN	'r'	'g'	'g'	'y'	'r'
check	true	true	true	true	true

label ϕ l_1 $\phi \cap \phi \cap \phi \cap \phi = \phi$

no label is shared by all the states in Inf (σ)

Acceptable

Implementation



Verifying Programs

Case Studies

[Dining philosophers problem]

liveness property: every philosopher can eat.

[LTL2BA]

A software for translating LTL formula to Büchi automata

a Büchi automaton is generated with at most $n \times 2^n$ states

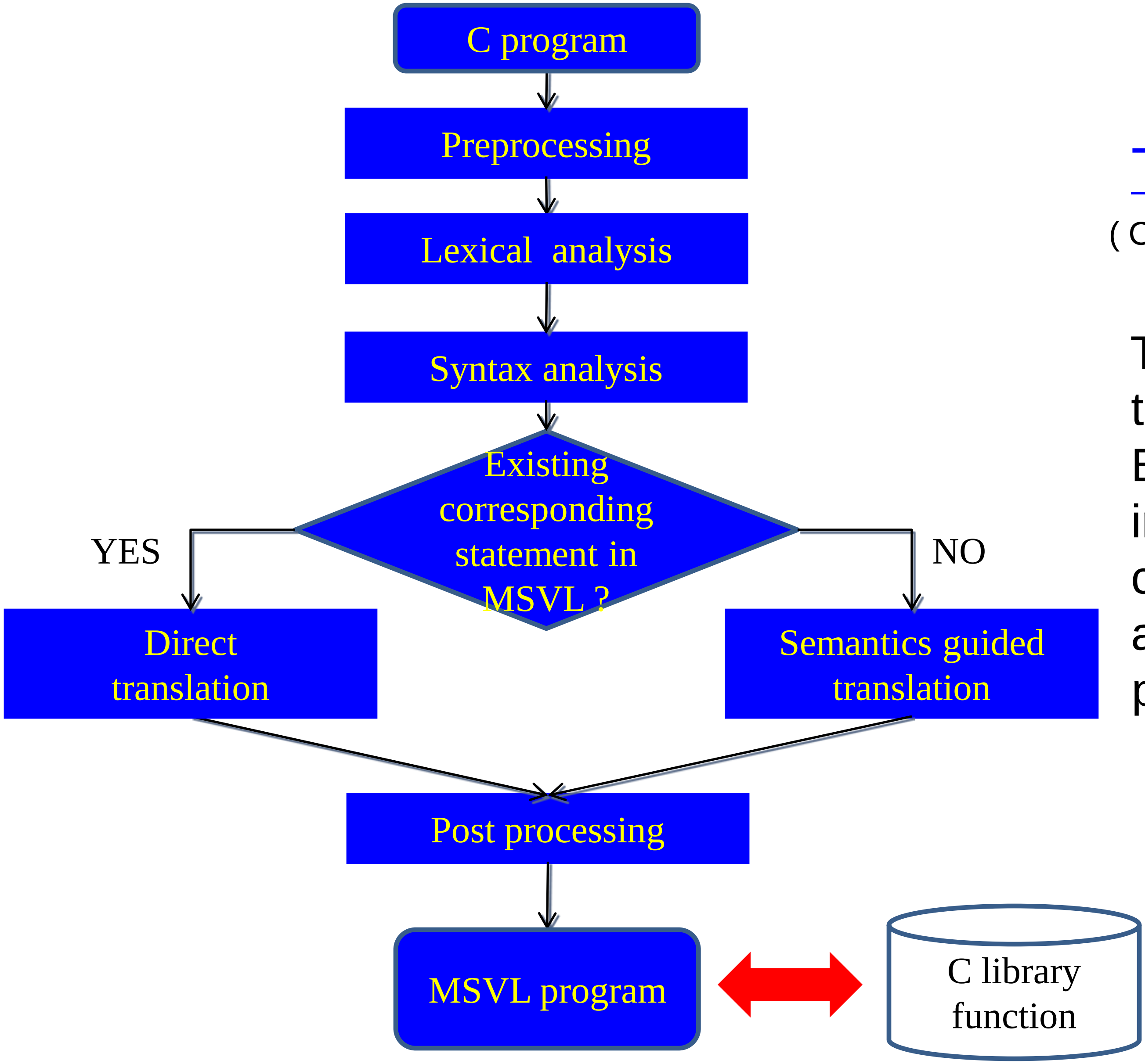
(n is the number of fairness conditions)

[Simple CPU]

An adder including dereference, decode and execution

If the address signal is true, the address is program counter address

Translating from C to MSVL

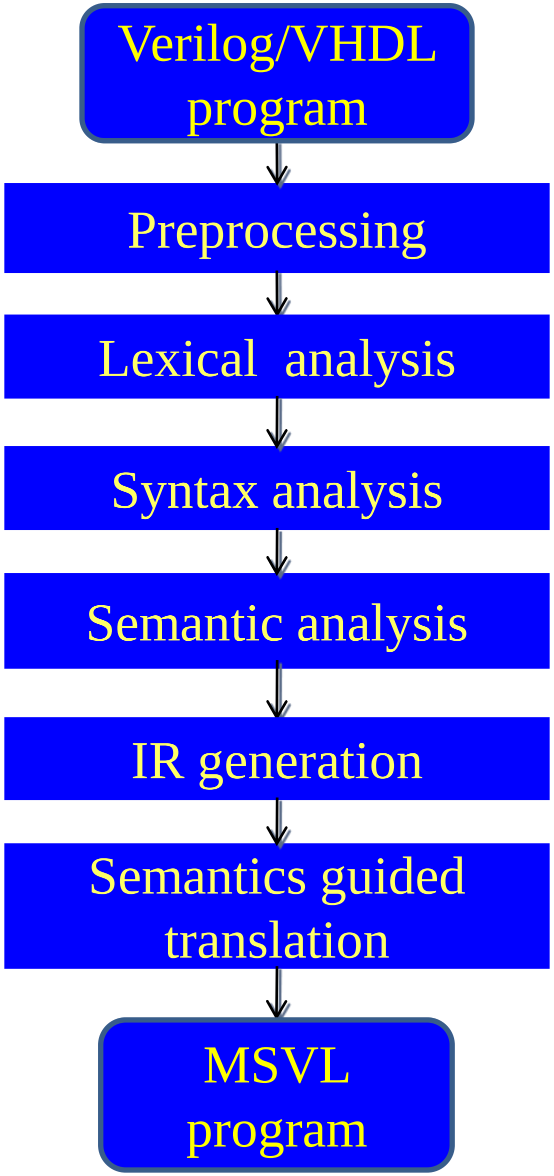


Twolf (C Program)

(C:15,912LOC MSVL:32,843LOC)

Twolf is selected from the SPEC CPU 2000 Benchmark. It is used in the process of creating the lithography artwork needed for the production of microchips.

Translating from Verilog/VHDL to MSVL



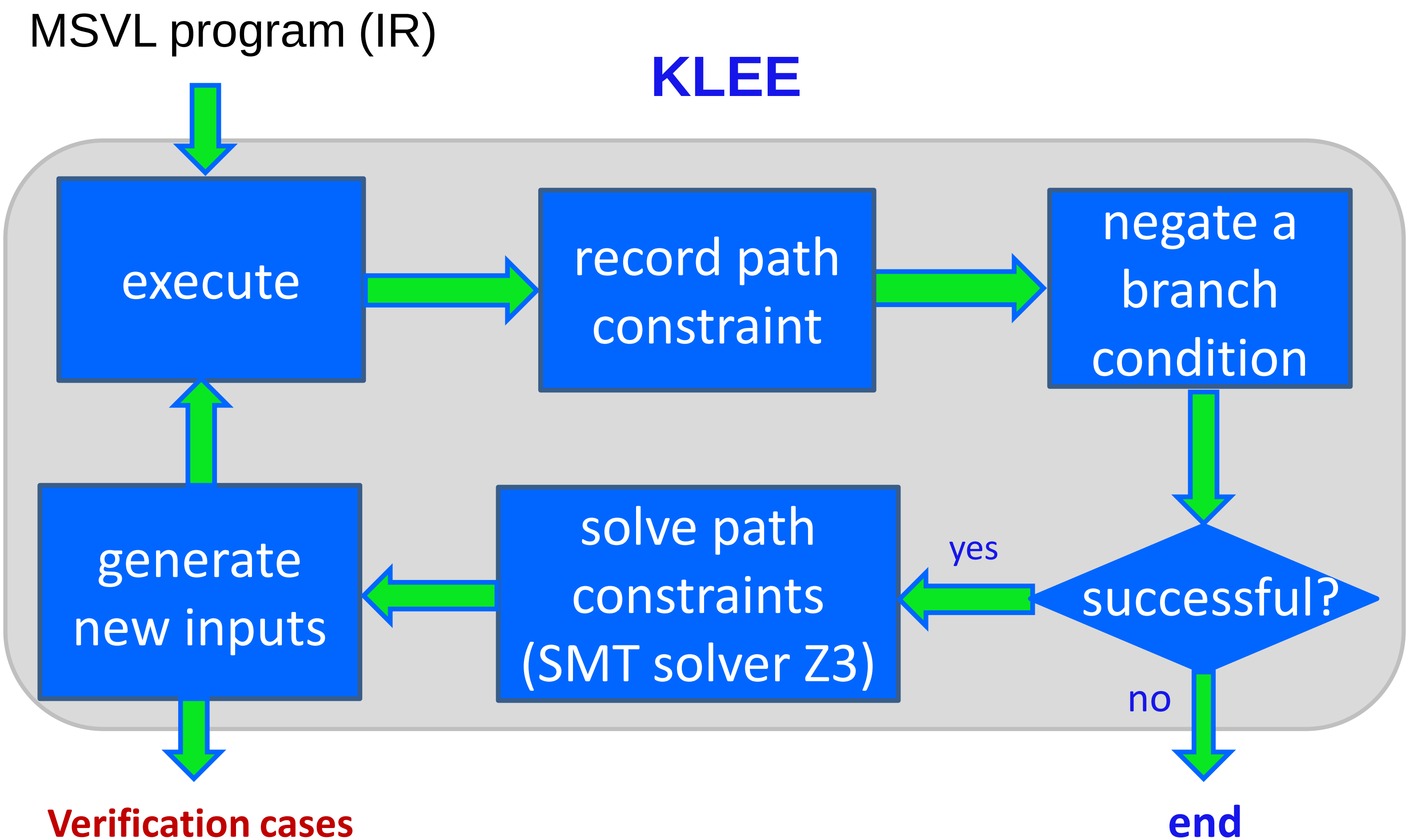
SHA (Verilog Program)

(Verilog:20,397LOC MSVL:44,583LOC)

SHA (Secure Hash Algorithm) is a cryptographic hash functions published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS).

Generating Verification Cases

Dynamic Symbolic Execution is used to generate verification cases



Generating Verification Cases

Case Studies

RERS P15 (RERS Benchmark)

A reactive system, where an engine calculates an output depending on the input and current state, and finally writes the output to the standard output

Totally, 16807 verification cases are generated with KLEE

Line Coverage: 41.81%
Branch Coverage: 50.70%

Property: 24 will never be output later than 22

Verifying Programs

Verification of (small) programs of Benchmark1

Our method

Program	LOC	Property	LTLAutomizer		T2		RiTHM		UMC4MSVL	
			Time(s)	Result	Time(s)	Result	Time(s)	Result	Time(s)	Result
Ex.Sec.2	5	$\Diamond\Box p$	0.37	√	0.38	√	0.75	√	0.24	√
Ex.Fig.8	34	$\Box(p \to \Diamond q)$	0.59	√	1.13	√	0.73	√	0.23	√
Toy acquire/release	14	$\Box(p \to \Diamond q)$	0.64	√	3.02	√	0.69	√	0.24	√
Toy linear arith. 1	13	$p \to \Diamond q$	0.80	×	3.24	×	0.61	×	0.22	×
Toy linear arith. 2	13	$p \to \Diamond q$	0.72	√	0.61	√	0.80	√	0.21	√
PostgreSQL strmsrv	259	$\Box(p \to \Diamond q)$	0.85	√	0.62	√	0.76	√	0.23	√
PostgreSQL strmsrv+bug	259	$\Box(p \to \Diamond q)$	1.62	×	1.99	×	0.78	×	0.25	×
PostgreSQL pgarch	61	$\Diamond\Box p$	1.10	×	1.14	×	0.75	×	0.24	×
PostgreSQL dropbuf	152	$\Box p$	1.39	×	0.58	×	0.77	×	0.28	×
PostgreSQL dropbuf	152	$\Box(p \to \Diamond q)$	0.89	√	1.27	√	0.78	√	0.28	√
Apache accept()	314	$\Box p \to \Box\Diamond q$	9.38	√	1.87	√	0.77	√	0.31	√
Apache progress	314	$\Box(p \to (\Diamond q_1 \vee \Diamond q_2))$	0.52	√	4.24	√	0.76	√	0.33	√
Windows OS 1	180	$\Box(p \to \Diamond q)$	0.72	√	0.58	√	0.77	√	0.28	√
Windows OS 2	158	$\Diamond\Box p$	0.59	√	0.53	√	0.77	√	0.26	√
Windows OS 2 + bug	158	$\Diamond\Box p$	0.77	×	0.95	×	0.65	×	0.28	×
Windows OS 3	14	$\Diamond\Box p$	0.42	√	0.57	√	0.73	√	0.24	√
Windows OS 4	327	$\Box(p \to \Diamond q)$	2.18	√	47.16	√	0.74	√	0.32	√
Windows OS 4	327	$(\Diamond p) \vee (\Diamond q)$	0.95	√	2.48	√	0.72	√	0.29	√
Windows OS 5	648	$\Box(p \to \Diamond q)$	0.56	√	0.48	√	0.74	√	0.33	√
Windows OS 6	13	$\Diamond\Box p$	1.05	√	1.06	√	0.74	√	0.22	√
Windows OS 6 + bug	13	$\Diamond\Box p$	0.51	×	0.60	×	0.81	×	0.23	×
Windows OS 7	13	$\Box\Diamond p$	0.66	√	1.57	√	0.73	√	0.23	√
Windows OS 8	181	$\Diamond\Box p$	0.46	√	0.39	√	0.70	√	0.26	√
Total	3622	23	27.74	23 (100%)	76.46	23 (100%)	17.05	23 (100%)	6.00	23 (100%)

All the four tools can successfully output the verification results. However, UMC4MSVL is more efficient than other three tools.

Verifying Programs

Verification of larger programs in Benchmark2

Our method

Programs	LOC	Prop	LTLAutomizer				T2				RiTHM				UMC4MSVL			
			√	×	★	Avg. Time (s)	√	×	★	Avg. Time (s)	√	×	★	Avg. Time (s)	√	×	★	Avg. Time (s)
RERS P14	514	50	20	2	28	37.24	0	11	39	0.96	20	30	0	22.60	20	30	0	0.48
RERS P15	1353	50	26	0	24	60.27	0	11	39	8.61	26	24	0	70.59	26	24	0	2.42
RERS P16	1304	50	19	0	31	70.85	0	9	41	5.45	20	30	0	45.65	20	30	0	2.41
RERS P17	2100	50	28	0	22	124.28	0	10	40	25.91	28	22	0	111.03	28	22	0	4.68
RERS P18	3306	50	24	0	26	296.59	0	8	42	24.19	26	24	0	99.69	26	24	0	11.93
RERS P19	8079	50	12	0	38	263.05	0	8	42	276.21	26	24	0	457.38	26	24	0	63.84
Total	16656	300	131(44%)		169	133.49	57(19%)		243	49.41	300(100%)		0	134.49	300(100%)		0	14.29

Success rate	44%	19%	100%	100%
Avg Time(s)	133.49	49.41	134.49	14.29

Verifying Programs

Verification of real-world programs

Program	LOC	Property	Time(s)	Result
CTCS-3	1572	$\Box(p \rightarrow \Box q)$	3.99	$\surd$
CTCS-3	1572	$\Diamond(p_1; (p_2; p_3)^*; p_4)$	6.02	$\surd$
CPU	1154	$\Box(p_1 \rightarrow \Diamond(\bigvee_{i=2}^{6} p_i)$	0.56	$\surd$
CPU	1154	$\Diamond((p_1; p_2; p_3; p_4; p_5)^*)$	28.56	$\surd$
LTL2BA	8940	$\Box(p)$	6.33	$\surd$
LTL2BA	8940	$\Diamond((p; q)^*)$	6.82	$\surd$
carc	4179	$\Box(p \rightarrow \Diamond q)$	7.58	$\times$
bzip2	4931	$\Box(p \rightarrow \Diamond q)$	3.27	$\times$
bzip2	4931	$\Diamond((p_1; p_2; p_3; p_4)^*)$	7.69	$\surd$
mcf	2176	$\Box(p \rightarrow \Diamond q)$	122.69	$\surd$
mcf	2176	$\Diamond(p_1; (p_2; p_3; p_4)^*)$	11.27	$\surd$
art	1521	$\Box(p \rightarrow \Diamond q)$	3.35	$\surd$
gzip	6187	$\Box(p \rightarrow \Diamond q)$	90.06	$\surd$
twolf	32853	$\Box(p \rightarrow \Diamond q)$	44.22	$\times$
gap	81087	$\Box(p \rightarrow \Diamond q)$	60.79	$\surd$
Total	163373	15	403.2	15 (100%)

All the programs and properties are successfully verified by UMC4MSVL in 403.2 seconds.
Other three tools fail on these programs.

Conclusion and Future Research

- We proposed a run-time unified model checking approach by executing both programs and properties at the same time.
- We use dynamic symbolic execution technique to generate verification cases for achieving higher path coverage.
- However, the proposed approach is incomplete.

In the future:

- Investigate more strategies for generating better verification cases
- Planning with MSVL Complier
- Bug-fixing guided by counterexamples

Thanks!

&

Questions?