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
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, verification 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 ...
(2) Model checking temporal properties without executing code (static)
- Considering all possible behaviors makes small programs have large state-space
- Tools: Ultimate LTLAutomizer, T2, ...
- Difficult to verify programs in large scale
- Poor in accuracy with lots of false positives
(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
Our approach:

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

Executing both the program and property
Background & Motivation

Approach Overview

**MSVL Program**

1. if (integer = 4) then
   2. if (integer [i] > 30 + i) then
   3. if (integer [i] > 30 + i) then
   4. if (integer [i] > 30 + i) then

**MSVL Compiler**

**KLEE**

**Verification cases**

**M_and_M’.exe**

**Verification Results**

**MSVL Program**

1. if (integer = i) then
   2. if (integer [i] > 30 + i) then
   3. if (integer [i] > 30 + i) then
   4. if (integer [i] > 30 + i) then

**MSVL Compiler**

**PPTL formula**

\[ p \land \Box^2 \diamond \left( (\Box \complement \diamond \text{more} \land \pi \diamond \text{true} \land \pi \land \text{more}) \right) \]

**MSVL Program**

Node = 1 and empty;
while (more)
{
  switch (node)
  {
    case 1: [i] <= i,
    case 2: [i] <= i,
  }
}
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
  \[ e ::= n \mid x \mid \bigcirc x \mid \bigotimes x \mid e_0 \ op \ e_1 (op ::= + \mid - \mid \ast \mid / \mid \text{mod}) \mid f(e_1, \ldots, e_n) \]

- Boolean expression
  \[ b ::= \text{true} \mid \text{false} \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \land b_1 \]
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
- **State Assignment:** $x <= = e$
- **Assignment:** $x := e$
- **State Frame:** lbf(x)
- **Interval Frame:** frame(x)
- **Conjunction:** p and q
- **Selection:** p or q
- **Next:** next p
- **Always:** always p
- **Conditional:** if b then p else q
- **Local variable:** local x: p
- **Projection:** $(p_1, ..., p_m) \text{ prj } q$
- **Sequence:** p; q
- **While:** while b do p
- **Parallel:** p || q
- **Await:** await(b)

```
def ε = ε
def (x = e) ∧ p_x = (x = e) ∧ p_x
def O(x = e ∧ p_x) ∧ Oε = O(x = e ∧ p_x) ∧ Oε
¬p_x → ∃b : (∀x = b ∧ x = b)
def □(more → □lbf(x)) = □(more → □lbf(x))
def p ∧ q = p ∧ q
:def p ∨ q = p ∨ q
:def Op = Op
:def □p = □p
:def (b → p) ∧ (¬b → q) = (b → p) ∧ (¬b → q)
def ∃x : p = ∃x : p
:def (p_1, ..., p_m) prj q = (p_1, ..., p_m) prj q
:def p ; q = p ; q
:def (p ∧ b)^* ∧ □(ε → ¬b) = (p ∧ b)^* ∧ □(ε → ¬b)
def (p ∧ (q ; true)) ∨ (q ∧ (p ; true)) ∨ (p ∧ q) = (p ∧ (q ; true)) ∨ (q ∧ (p ; true)) ∨ (p ∧ q)
def (frame(x_1) ∧ ... ∧ frame(x_h)) ∧ □(ε ↔ b) = (frame(x_1) ∧ ... ∧ frame(x_h)) ∧ □(ε ↔ b)
```
MSVL

Normal form of MSVL programs

\[ \phi \equiv (\bigvee_{i=1}^{k} \phi_{ei} \land \varepsilon) \lor (\bigvee_{j=1}^{h} \phi_{cj} \land \bigcirc \phi_{fj}) \]

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
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 (PPTL)

- Syntax

\[ P ::= p \mid \bigcirc P \mid \neg P \mid P \lor Q \mid (P_1, \ldots, P_m) \text{ proj } P \]

- Semantics

An interval \( \sigma \) is a non-empty sequence of states, which can be finite or infinite.
An interpretation is a triple $I = (\sigma, i, j)$, where $\sigma$ is an interval, $i$ is an integer, and $j$ an integer or $\omega$.

The satisfaction relation is inductively defined as follows:

- $I |= p$ iff $si[p] = true$, and $p \in Prop$ is an atomic proposition

- $I |= \circ P$ iff $i < j$ and $(\sigma, i+1, j) |= P$

- $I |= \neg P$ iff $I \not|= P$

- $I |= P \lor Q$ iff $I |= P$ or $I |= Q$
\( I \models (P_1, P_2, \ldots, P_m) \text{ prj } P \), 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 \(\sigma'\):

(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 \(0 \leq h \leq m\)

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

\((P_1, P_2, P_3) \text{ prj } P\)
A PPTL formula $P$ is in normal form if,

$$
P \equiv \bigvee_{i=1}^{l} P_{e_i} \land \varepsilon \lor \bigvee_{j=1}^{t} P_{c_j} \land \bigcirc P_{f_j}
$$

- $P_{f_j}$ is a PPTL formula without disjunct being the main operator
- $P_{e_i}$ and $P_{c_j}$ are true or state formulas of the form:

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

$\hat{p}_k$ 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 graphs (LNFG) are constructed based on normal form of PPTL formulas

- LNFG of a PPTL formula is a 4-tuple

\[
LNFG = (CL, EL, V_0, LL = \{L_1, \ldots, L_k\})
\]

- \(CL\): non-empty finite set of nodes
- \(EL\): set of directed edges among \(CL\)
- \(V_0\): set of initial (root) nodes
- \(L_i\): \(L_i \subseteq CL, 1 \leq i \leq k\), set of nodes with \(l_i\) being the label.

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

A path is acceptable if it is finite, or infinite and all the nodes in \(Inf(\pi)\) do not share a same label.
Labeled Normal Form Graph

Example

LNFG of $\Box(\bigcirc q) \land \Box((p; q) \lor q)$

$n_0$: $\Box(\bigcirc q) \land \Box((p; q) \lor q)$

$n_1$: $q \land \bigcirc(\bigcirc q) \land \Box((p; q) \lor q)$

$n_2$: $q \land \bigcirc(\bigcirc q) \land (true; q) \land \Box((p; q) \lor q)$

NF($n_0$) =
$q \land \bigcirc(q \land \Box(\bigcirc q) \land \Box((p; q) \lor q)) \lor$
$p \land \bigcirc(q \land \Box(\bigcirc q) \land (true; q) \land \Box((p; q) \lor q))$
Example

LNFG of $\Box(\Diamond q) \land \Box((p;q) \lor q)$

$n_0$: $\Box(\Diamond q) \land \Box((p;q) \lor q)$

$n_1$: $q \land \Box(\Diamond q) \land \Box((p;q) \lor q)$

$n_2$: $q \land \Box(\Diamond q) \land (true;q) \land \Box((p;q) \lor q)$

NF($n_1$) =
$q \land \Diamond(q \land \Box(\Diamond q) \land \Box((p;q) \lor q)) \lor p \land q \land \Diamond(q \land \Box(\Diamond q) \land (true;q) \land \Box((p;q) \lor q))$
Labeled Normal Form Graph

Example

LNFG of $\Box (\Diamond q) \land \Box ((p; q) \lor q)$

Rewrite $n_2$ with fin label

$n_0$: $\Box (\Diamond q) \land \Box ((p; q) \lor q)$

$n_1$: $q \land \Box (\Diamond q) \land \Box ((p; q) \lor q)$

$n_2$: $q \land \Box (\Diamond q) \land (\text{fin}(l_1); q) \land \Box ((p; q) \lor q)$
Example

Labeled Normal Form Graph

LNFG of $\Box (\Diamond q) \land \Box ((p \land q) \lor q)$

Rewrite $n_2$ with fin label

$\text{NF}(n_2) =$

$q \land \Diamond (q \land \Box (\Diamond q) \land \Box ((p \land q) \lor q)) \lor$

$q \land \Diamond (q \land \Box (\Diamond q) \land (\text{fin}(l_1); q) \land \Box ((p \land q) \lor q))$

$n_0: \Box (\Diamond q) \land \Box ((p \land q) \lor q)$

$n_1: q \land \Box (\Diamond q) \land \Box ((p \land q) \lor q)$

$n_2: q \land \Box (\Diamond q) \land (\text{fin}(l_1); q) \land \Box ((p \land q) \lor q)$
Example

- LNFG of $\Box (\Diamond q) \land \Box ((p \land q) \lor q)$
  - $CL = \{n_0, n_1, n_2\}$
  - $EL = \{<n_0, q, n_1>, <n_0, p, n_2>, <n_1, q, n_1>, <n_1, p \land q, n_2>, <n_2, q, n_1>, <n_2, q, n_2>\}$
  - $V_0 = \{n_0\}$
  - $L = \{L_1\}$ and $L_1 = \{n_2\}$
Example

LNFG of $\blacksquare (\lozenge q) \land \blacksquare ((p \land q) \lor q)$

- $CL = \{n_0, n_1, n_2\}$
- $EL = \{<n_0, q, n_1>, <n_0, p, n_2>,$
  $<n_1, q, n_1>, <n_1, p \land q, n_2>,$
  $<n_2, q, n_1>, <n_2, q, n_2>\}$
- $V_0 = \{n_0\}$
- $\mathbb{L} = \{\mathbb{L}_1\}$ and $\mathbb{L}_1 = \{n_2\}$

Path $\pi =< n_0, q, n_1, p \land q, (n_2, q)\omega>$

- Nodes that occur infinitely often have the same label $l_1$
- Unacceptable
Example

- LNFG of $\square(\bigcirc q) \land \square((p;q) \lor q)$
  - $CL = \{n_0, n_1, n_2\}$
  - $EL = \{<n_0, q, n_1>, <n_0, p, n_2>, <n_1, q, n_1>, <n_1, p \land q, n_2>, <n_2, q, n_1>, <n_2, q, n_2>\}$
  - $V_0 = \{n_0\}$
  - $L = \{L_1\}$ and $L_1 = \{n_2\}$

- Path $\pi =< n_0, q, n_1, p \land q, (n_2, q)^\omega>$
  - Nodes that occur infinitely often have the same label $l_1$
  - Unacceptable

- Path $\pi =< n_0, p, (n_2, q, n_1, p \land q)^\omega>$
  - Nodes that occur infinitely often do not have a same label
  - Acceptable
Use a unique integer to represent each of nodes in the LNFG
Global Variable **CuNode**: presenting the node explored at the current state. The first node to be explored is a root node (**CuNode** \(<=1\))
For each node $i$, the following program pattern is created: \( \text{if}(\text{CuNode}=i) \text{then}\{ \text{M} \} \)

\text{M} \text{ is another program pattern w.r.t all the edges starting from } i
For each node $i$, the following program pattern is created: \( \text{if}(\text{CuNode}=i)\text{then}\{ M \} \)

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

---

\[
\begin{align*}
\text{CuNode} &\leq 1 \quad \text{and} \\
\text{while}(\text{true}) &\{ \\
\quad \text{if}(\text{CuNode}=1) &\text{then}\{ \\
\quad \} \\
\quad \text{else} &\{ \\
\quad \quad \text{if}(\text{CuNode}=2) &\text{then}\{ \\
\quad \quad \} \\
\quad \} \\
\} 
\end{align*}
\]
For each node $i$, the following program pattern is created: \[ \text{if}(\text{CuNode}=i) \text{then}\{ \text{M} \} \]

\text{M} is another program pattern w.r.t all the edges starting from $i$. 

\[ \text{CuNode} <= 1 \text{ and while(true)}\{ 
\text{if}(\text{CuNode}=1) \text{then}\{ 
\}
\text{else}\{ 
\text{if}(\text{CuNode}=2) \text{then}\{ 
\}
\text{else}\{ 
\text{if}(\text{CuNode}=3) \text{then}\{ 
\}
\text{else}\{\text{false}\}
\}
\}} \]
For each node $i$, the following program pattern is created: $\text{if}(\text{CuNode}=i)\text{then}\{ M \}$

$M$ is another program pattern w.r.t. all the edges starting from $i$.
For each edge from i to j with the label being p, program
\[ \text{if}(p) \text{ then}\{ \text{CuNode} := j\} \text{ else } \{ \text{false}\} \] is produced
For each edge from i to j with the label being p, program

\[
\text{if}(p) \text{ then}\{\text{CuNode}:=j\}\text{ else}\{\text{false}\}
\]

is produced.
For each edge from $i$ to $j$ with the label being $p$, program

$\text{if}(p) \text{ then}\{\text{CuNode}:=j\} \text{ else } \{\text{false}\}$

is produced.
Verification as Dynamic Program Execution

Example: Traffic Light

```c
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**

```plaintext
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 $M$

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

Desired Property $P$

$$\Box(\neg((EW='g')\land(SN='g'))\land\Box(\neg((EW='y')\land(SN='y')))$$

MSVL Program

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

and

CuNode<==1 and
while(more)
$$\{ \text{ if(CuNode=1)then}\{ \text{ CuNode:=2 or if((EW=='g’ and SN=='g’) or (EW=='y’ and SN=='y’)) then{CuNode:=3} else{false} or CuNode:=4 } \}$$
$$\text{ else }$$
$$\{ \text{ if(CuNode=2)then}\{ \text{ L={l1} and ( CuNode:=2 or if(EW=='g’ and SN=='g’) then{Node:=3} else{false})} \}$$
$$\text{ else }$$
$$\{ \text{ if(CuNode=3)then}\{ \text{ CuNode:=3} \}$$
$$\text{ else }$$
$$\{ \text{ if(CuNode=4) then}\{ \text{ L={l2} and ( CuNode:=4 or if(EW=='y’ and SN=='y’) then {CuNode:=3} else{false})} \}$$
$$\text{ else{false}}})))$$
$$\}$$

if(CuNode=1) then{…}
else {if(CuNode=2)then{…} else{ if(CuNode=3)then{…} else{ if(CuNode=4)then{…} else{false}}}…else{false}}}
An **execution** of an MSVL program is sequences of states

\[ \sigma = \langle s_0, s_1, \ldots \rangle \]

A finite execution in **M**

A finite path in LNFG of \( \neg P \)

A finite execution in **M** and \( \neg P \)

Finite: 1,1 2,2 3,2 4,2 5,5

**EW**

‘g’ ‘y’ ‘r’ ‘r’

**SN**

‘r’ ‘r’ ‘g’ ‘y’

All executions in **M and \( \neg P \)**
An execution of an MSVL program is sequences of states

\[ \sigma = \langle s_0, s_1, \ldots \rangle \]

Verification as Dynamic Program Execution

An infinite execution in \( M \) and \( \neg P \)

Infinite:

\[
\begin{array}{c}
1,1 & 2,2 & 3,3 & 4,3 & 1,3 & 2,3 \\
EW & 'g' & 'y' & 'r' & 'r' & 'g' \\
SN & 'r' & 'r' & 'g' & 'y' & 'r'
\end{array}
\]
An execution of an MSVL program is sequences of states
\[ \sigma = \langle s_0, s_1, \ldots \rangle \]

Verification as Dynamic Program Execution

An infinite path in LNFG of \( \neg P \)

Infinite:

All executions in \( M \text{ and } \neg P \)
Verification as Dynamic Program Execution

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

Finite: \( 1,1 \rightarrow 2,2 \rightarrow 3,2 \rightarrow 4,2 \rightarrow 5,5 \)

| \( EW \)  | \( g \) | \( y \) | \( r \) | \( r \) |
| \( SN \)  | \( r \) | \( r \) | \( g \) | \( y \) |

| check | true | true | true | false |

Infeasible
Verification as Dynamic Program Execution

Feasible Execution: An execution $\sigma = < s_0, s_1, ... >$ 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: $1,1 \rightarrow 2,2 \rightarrow 3,3 \rightarrow 4,3 \rightarrow 1,3 \rightarrow 2,3$
**Verification as Dynamic Program Execution**

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

**Feasibility checking (infinite)**

<table>
<thead>
<tr>
<th>Infinite:</th>
<th>( 1,1 )</th>
<th>( 2,2 )</th>
<th>( 3,2 )</th>
<th>( 4,2 )</th>
<th>( 1,2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>'g'</td>
<td>'y'</td>
<td>'r'</td>
<td>'r'</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EW</td>
<td>'g'</td>
<td>'y'</td>
<td>'r'</td>
<td>'r'</td>
<td></td>
</tr>
<tr>
<td>SN</td>
<td>'r'</td>
<td>'r'</td>
<td>'g'</td>
<td>'y'</td>
<td></td>
</tr>
<tr>
<td>check</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Feasible
Verification as Dynamic Program Execution

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

Feasibility checking (infinite)

---

Infinite: $1,1 \rightarrow 2,2 \rightarrow 3,2 \rightarrow 4,2 \rightarrow 1,2$

<table>
<thead>
<tr>
<th>EW</th>
<th>SN</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>'r'</td>
<td>true</td>
</tr>
<tr>
<td>'y'</td>
<td>'r'</td>
<td>true</td>
</tr>
<tr>
<td>'r'</td>
<td>'g'</td>
<td>true</td>
</tr>
<tr>
<td>'r'</td>
<td>'y'</td>
<td>true</td>
</tr>
</tbody>
</table>
A feasible execution $\sigma = \langle s_0, s_1, \ldots \rangle$ is acceptable if

1. $\sigma$ is finite; or
2. $\sigma$ is infinite and no labels are shared by all the states in Inf ($\sigma$).

Whether a feasible path is acceptable?

Infinite: $l_1$ is shared by all the states in Inf ($\sigma$)

Unacceptable

<table>
<thead>
<tr>
<th>EW</th>
<th>SN</th>
<th>label</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>'r'</td>
<td>$\phi$</td>
<td>true</td>
</tr>
<tr>
<td>'y'</td>
<td>'r'</td>
<td>${l_1}$</td>
<td>true</td>
</tr>
<tr>
<td>'r'</td>
<td>'g'</td>
<td>${l_1}$</td>
<td>true</td>
</tr>
<tr>
<td>'g'</td>
<td>'y'</td>
<td>${l_1}$</td>
<td>true</td>
</tr>
<tr>
<td>'r'</td>
<td>'r'</td>
<td>${l_1}$</td>
<td>true</td>
</tr>
</tbody>
</table>

$\{l_1\} \cap \{l_1\} \cap \{l_1\} \cap \{l_1\} = \{l_1\}$
Verification as Dynamic Program Execution

Suppose the model is modified as follows:

\[
\begin{array}{c|c|c|c|c|c|c}
EW & 1 & 2 & 3 & 4 & 5 \\
\hline
'g' & 'g' & 'r' & 'r' & 'r' \\
\hline
SN & 1 & 2 & 3 & 4 & 5 \\
\hline
'r' & 'g' & 'g' & 'g' & 'y' \\
\hline
\end{array}
\]

A counterexample is found!

Infinite:

\[
\begin{array}{c|c|c|c|c|c|c}
EW & 1 & 2 & 3 & 4 & 1 & 2 \\
\hline
'g' & 'g' & 'r' & 'r' & 'g' & 'g' \\
\hline
SN & 1 & 2 & 3 & 4 & 1 & 3 \\
\hline
'r' & 'g' & 'g' & 'g' & 'y' & 'r' \\
\hline
\end{array}
\]

\[
\begin{array}{c|c}
check & true \\
\hline
label & \phi \\
\hline
\end{array}
\]

Acceptable

no label is shared by all the states in \( \text{Inf}(\sigma) \)
Implementation

UMC4MSVL

M_IR

MSVL Compiler

M and M'

LNFG2MSVL

Construct LNFG

LNFG

PPTL formula $\neg P$

M and M'

feasible?

yes

no

acceptable?

yes

Error

No error found

All cases are verified

KLEE

M_and_M'.exe

Verification cases

M_IR

MSVL Compiler

M and M'

LNFG2MSVL

Construct LNFG

LNFG

PPTL formula $\neg P$

M and M'

feasible?

yes

no

acceptable?

yes

Error

No error found

All cases are verified
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.
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

MSVL program (IR)

KLEE

execute

record path constraint

solve path constraints (SMT solver Z3)

negate a branch condition

successful?

no

end

yes

generate new inputs

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
Verification of (small) programs of Benchmark1

### Table: Verification Results

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

All the four tools can successfully output the verification results. However, UMC4MSVL is more efficient than other three tools.
## Verification of larger programs in Benchmark2

Our method

<table>
<thead>
<tr>
<th>Programs</th>
<th>LOC</th>
<th>Prop</th>
<th>LTLAutomizer</th>
<th>T2</th>
<th>RiTHM</th>
<th>UMC4MSVL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>×</td>
<td>★</td>
<td>Avg. Time (s)</td>
</tr>
<tr>
<td>RERS P14</td>
<td>514</td>
<td>50</td>
<td>20</td>
<td>2</td>
<td>28</td>
<td>37.24</td>
</tr>
<tr>
<td>RERS P15</td>
<td>1353</td>
<td>50</td>
<td>26</td>
<td>0</td>
<td>24</td>
<td>60.27</td>
</tr>
<tr>
<td>RERS P16</td>
<td>1304</td>
<td>50</td>
<td>19</td>
<td>0</td>
<td>31</td>
<td>70.85</td>
</tr>
<tr>
<td>RERS P17</td>
<td>2100</td>
<td>50</td>
<td>28</td>
<td>0</td>
<td>22</td>
<td>124.28</td>
</tr>
<tr>
<td>RERS P18</td>
<td>3306</td>
<td>50</td>
<td>24</td>
<td>0</td>
<td>26</td>
<td>296.59</td>
</tr>
<tr>
<td>RERS P19</td>
<td>8079</td>
<td>50</td>
<td>12</td>
<td>0</td>
<td>38</td>
<td>263.05</td>
</tr>
<tr>
<td>Total</td>
<td>16656</td>
<td>300</td>
<td>131</td>
<td>44%</td>
<td>169</td>
<td>133.49</td>
</tr>
</tbody>
</table>

### Success rate

- Our method: 100%
- LTLAutomizer: 44%
- T2: 19%
- RiTHM: 100%

### Avg Time(s)

- Our method: 14.29
- LTLAutomizer: 133.49
- T2: 49.41
- RiTHM: 134.49
Verifying Programs

Verification of real-world programs

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Property</th>
<th>Time(s)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTCS-3</td>
<td>1572</td>
<td>□(p → □q)</td>
<td>3.99</td>
<td>✓</td>
</tr>
<tr>
<td>CTCS-3</td>
<td>1572</td>
<td>◊(p₁; (p₂; p₃)*; p₄)</td>
<td>6.02</td>
<td>✓</td>
</tr>
<tr>
<td>CPU</td>
<td>1154</td>
<td>□(p₁ → ◊( ∨ pᵢ)</td>
<td>0.56</td>
<td>✓</td>
</tr>
<tr>
<td>CPU</td>
<td>1154</td>
<td>◊((p₁; p₂; p₃; p₄; p₅)*</td>
<td>28.56</td>
<td>✓</td>
</tr>
<tr>
<td>LTL2BA</td>
<td>8940</td>
<td>□(p)</td>
<td>6.33</td>
<td>✓</td>
</tr>
<tr>
<td>LTL2BA</td>
<td>8940</td>
<td>◊((p; q)*</td>
<td>6.82</td>
<td>✓</td>
</tr>
<tr>
<td>care</td>
<td>4179</td>
<td>□(p → ◊q)</td>
<td>7.58</td>
<td>✗</td>
</tr>
<tr>
<td>bzip2</td>
<td>4931</td>
<td>□(p → ◊q)</td>
<td>3.27</td>
<td>✗</td>
</tr>
<tr>
<td>bzip2</td>
<td>4931</td>
<td>◊((p₁; p₂; p₃; p₄)*</td>
<td>7.69</td>
<td>✓</td>
</tr>
<tr>
<td>mcf</td>
<td>2176</td>
<td>□(p → ◊q)</td>
<td>122.69</td>
<td>✓</td>
</tr>
<tr>
<td>mcf</td>
<td>2176</td>
<td>◊((p₁; (p₂; p₃; p₄)*</td>
<td>11.27</td>
<td>✓</td>
</tr>
<tr>
<td>art</td>
<td>1521</td>
<td>□(p → ◊q)</td>
<td>3.35</td>
<td>✓</td>
</tr>
<tr>
<td>gzip</td>
<td>6187</td>
<td>□(p → ◊q)</td>
<td>90.06</td>
<td>✓</td>
</tr>
<tr>
<td>twolf</td>
<td>32853</td>
<td>□(p → ◊q)</td>
<td>44.22</td>
<td>✗</td>
</tr>
<tr>
<td>gap</td>
<td>81087</td>
<td>□(p → ◊q)</td>
<td>60.79</td>
<td>✓</td>
</tr>
<tr>
<td>Total</td>
<td>163373</td>
<td>15</td>
<td>403.2</td>
<td>15 (100%)</td>
</tr>
</tbody>
</table>

All the programs and properties are successfully verified by UMC4MSVL in 403.2 seconds. Other three tools fail on these programs.
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 Compiler
- Bug-fixing guided by counterexamples
Thanks!
&
Questions?