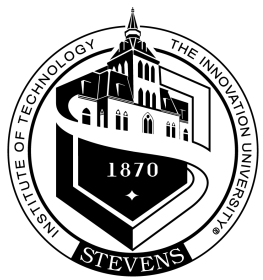
Temporal Verification of Non-linear Programs

Candidate
Yuandong Cyrus Liu

ADVISORY COMMITTEE
Dr. Eric Koskinen, Chairman
Dr. David Naumann, CS, Stevens
Dr. Jun Xu, CS, Stevens
Dr. Daniel Dietsch, SE, Freiburg
Dr. Hang Liu, ECE, Stevens

Stevens Institute of Technology
Department of Computer Science

17th December 2021



Outline

Part 1

Verification Background

Automated software verification: $P \models \varphi$

Challenges:

- Types of program P.
- Assertion logic of φ.

Verification Background

Automated software verification: $P \models \varphi$

Challenges:

- Types of program P.
- Assertion logic of φ.

Our work

Non-linear programs and LTL (including Reachability, Termination).

What is LTL?

Program properties (ϕ):

- **Reachability:** program never reach a bad state (err).
- **Termination:** program exits.
- **Linear Temporal Logic (LTL):** a generalization of program behaviors change over time.

What is LTL?

Program properties (ϕ):

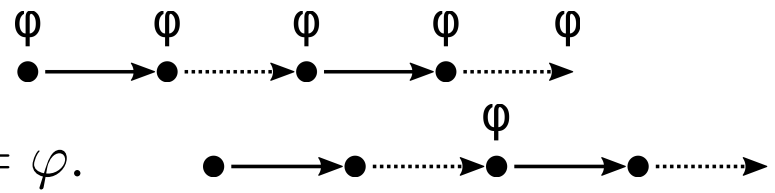
- **Reachability:** program never reach a bad state (err).
- **Termination:** program exits.
- **Linear Temporal Logic (LTL):** a generalization of program behaviors change over time.

LTL Formulae: $\neg\varphi$, $\Box\varphi$, $\Diamond\varphi$...

π is a program trace.

$\pi \models \Box\varphi \iff \forall j \geq 0, \pi_j \models \varphi.$

$\pi \models \Diamond\varphi \iff \exists j \geq 0, such\ that\ \pi_j \models \varphi.$



What is LTL?

Program properties (ϕ):

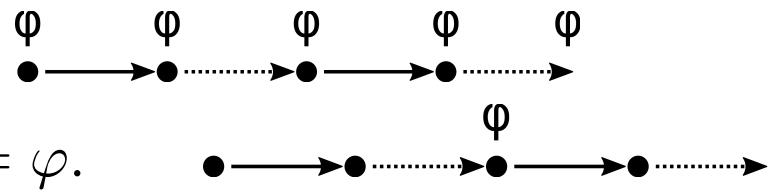
- **Reachability:** program never reach a bad state (err).
- **Termination:** program exits.
- **Linear Temporal Logic (LTL):** a generalization of program behaviors change over time.

LTL Formulae: $\neg\varphi$, $\Box\varphi$, $\Diamond\varphi$...

π is a program trace.

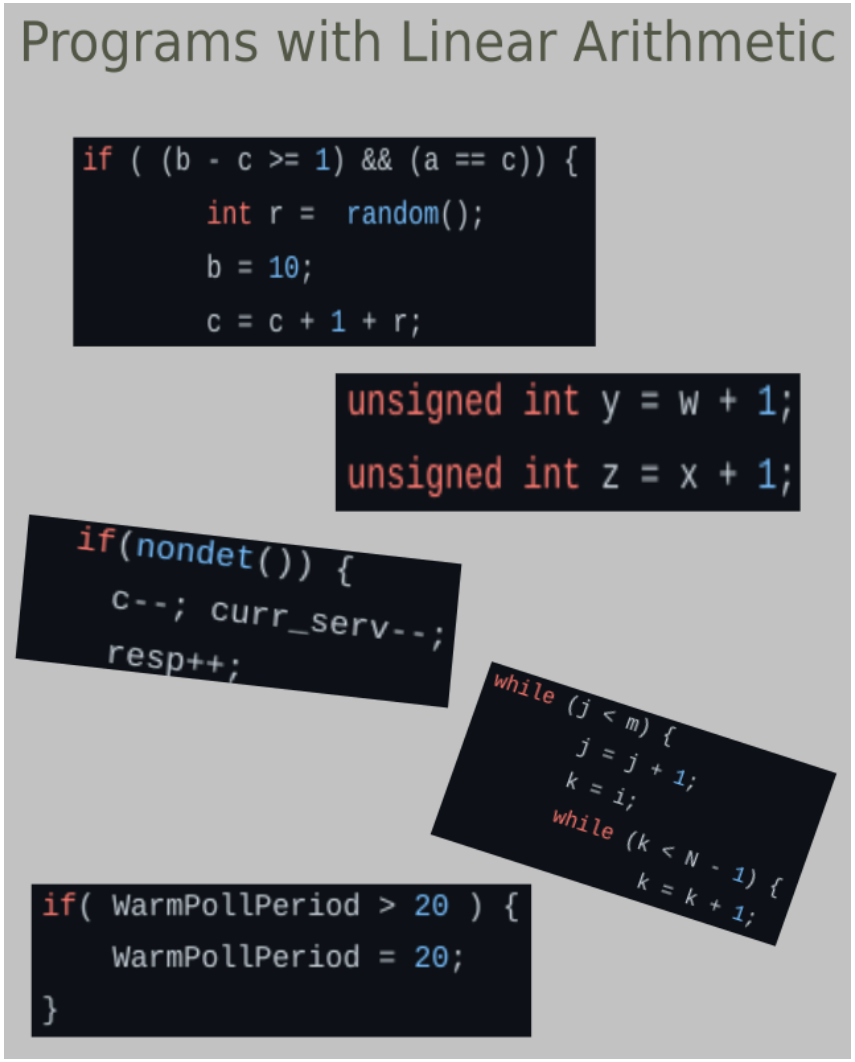
$\pi \models \Box\varphi \iff \forall j \geq 0, \pi_j \models \varphi.$

$\pi \models \Diamond\varphi \iff \exists j \geq 0, such\ that\ \pi_j \models \varphi.$



LTL encompasses reachability ($\Box\neg err$) and termination ($\Diamond$exit).

Benchmarks Repositories (SVCOMP [1] and ULTIMATE [2])



[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

[2] https://github.com/ultimate-pa/ultimate/tree/dev/trunk/examples/LTL

Benchmarks Repositories (SVCOMP [1] and ULTIMATE [2])

Programs with Linear Arithmetic

```
if ( (b - c >= 1) && (a == c)) {
        int r =  random();
        b = 10;
        c = c + 1 + r;
```

```
unsigned int y = w + 1;
unsigned int z = x + 1;
```

```
if(nondet()) {
    c--; curr_serv--;
    resp++;
```

```
while (j < m) {
    j = j + 1;
    k = i;
    while (k < N - 1) {
        k = k + 1;
```

```
if( WarmPollPeriod > 20 ) {
    WarmPollPeriod = 20;
}
```

Programs with Non-linear Arithmetic (NLA)

Bitvectors

```
for (v >>= 1; v; v >>= 1) {
    r <<= 1;
    r |= v & 1;
    s--;
  }
r <<= s; // shift when v's highest bits are zero
```

De-compiled Binaries

```
if (((((((tmp__18 == 0u)&1)) && ( tmp__15 != 0u) )&1))) {
tmp__14__PHI_TEMPORARY = tmp__15;   /* for PHI node */
goto block_401135;
```

Polynomials

```
while (1) {
    __VERIFIER_assert(4*x - y*y*y*y - 2*y*y*y - y*y == 0);
    if (!(c < k))
        break;
    c = c + 1;
    y = y + 1;
    x = y * y * y + x;    }
__VERIFIER_assert(k*y - (y*y) == 0);
__VERIFIER_assert(4*x - y*y*y*y - 2*y*y*y - y*y == 0);
```

Others (logarithm, square root etc.)

[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

[2] https://github.com/ultimate-pa/ultimate/tree/dev/trunk/examples/LTL

Problems in LTL Verification of NLA Programs

Why can't we currently verify these NLA programs?

Problems in LTL Verification of NLA Programs

Why can't we currently verify these NLA programs?

Static verification tools are limited for NLA programs:

- Complexity in bitvector SMT solving.
- Unable to find invariants and rank functions (needed for termination and LTL) for NLA programs.
- Limited benchmarks for temporal verification tasks of bitvector/polynomial programs.
- Some bitvector reachability/termination techniques, but not for LTL.

Problems in LTL Verification of NLA Programs

Why can't we currently verify these NLA programs?

Static verification tools are limited for NLA programs:

- Complexity in bitvector SMT solving.
- Unable to find invariants and rank functions (needed for termination and LTL) for NLA programs.
- Limited benchmarks for temporal verification tasks of bitvector/polynomial programs.
- Some bitvector reachability/termination techniques, but not for LTL.

Dynamic analysis in verification:

- Simply run the program; make inferences from observed states.
- How to learn invariants, e.g. what kinds of templates.
- Results are correct with respect to executing traces, soundness issue.
- Validation: SMT solving as a sub-process for correctness checking, inefficient in bitvector programs.
- Some dynamic analyses for reachability/termination, but not LTL.

Goal: LTL Verification of NLA Porgrams.

Solutions

LTL of Bitvector Programs via Bitwise Branching.[a]

[a] Yuandong Cyrus Liu, Chengbin Pang, Daniel Dietsch, Eric Koskinen, Ton-Chanh Le, Georgios Portokalidis, and Jun Xu. Proving LTL Properties of Bitvector Programs and Decompiled Binaries, APLAS 2021.

LTL of De-compiled Binaries via Translation and DarkSea.[b]

[b] Binary verification tool chain DarkSea(plan to submit a tool paper for it).

LTL of Polynomial Programs via Dynamic Analysis.[c,d]

[c] Yuandong Cyrus Liu, Poster Session, FMCAD-SF 2021.
[d] Plan to submit it to CAV'22 or OOPSLA'22.

Part 2

LTL of Bitvectors

LTL of Bitvectors

Bitvector Applications

- System/low level programs.
- Precise integer reasoning.
- Memory safety.
- Binary verification.

Today's Verification Tools

SMT solvers like MathSAT, CVC4, Z3, SMTInterpol support various theories:[a]

- Bools, Ints, Floats.
- FixedSizeBitVectors.

[a] http://smtlib.cs.uiowa.edu/theories.shtml

Challenges

Challenges in verification of bitvector programs

- Bit-blasting in SMT practical applications[a], leads to exponential growth ($\mathcal{O}(2^n)$).
- Verification tools (e.g. CPAchecker, Ultimate) have limited support for liveness verification over the bitvector domain.
- LTL verification tasks are absent from SV-COMP.
- Very limited bitvector benchmarks in SV-COMP.

[a] Kovásznai et al. - 2016 - Complexity of Fixed-Size Bit-Vector Logics

Example 1: Reachability

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

- CPAchecker, Ultimate etc., tools can verify it via bitvector theory for safety only.

Example 2: Termination

```
1 a = *;
2 assume(a>0);
3 while(x>0){
4   a--;
5   x = x & a;
6 }
```

- Fewer tools can handle termination of bitvector programs.
- For example, ULTIMATE does not support Bitvector logics for termination, reports Unknown results with Overapproximation.

Example 3: LTL ($\varphi = \Box(\Diamond(n < 0))$)

```
1     while(1) {
2       n = *; x = *; y = x-1;
3       while(x>0 && n>0) {
4         n++;
5         y = x | n;
6         x = x - y;
7       }
8       n = -1;
9     }
```

- No techniques can prove LTL of bitvector programs. The closest possible verifier is ULTIMATE.

Linear Approximation in SMT Solving

How can we address all of these examples?

Build on ideas from SMT solving: linear approximation.

Support bitvector through linear constraints:

$$in_w(x) =_{def} (0 \leq x < 2^w)$$

Key Idea

Approximate bitvector reasoning with integer reasoning through source translation.

- Various state-of-art verifiers support the integer domain.
- Overapproximate bit-vector operations with linear constraints.
- Sound transformation.
- Open path to binary verification.

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3    s = x >> 31;
4    x--;
5    r = x + (s&(1-s));
6    if (r<0) error();
7 }
```

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

1. Consider program expression `x >> 31`:
   - Under **condition** `x >= 0`, this expression is just "0". So replace it with `x >= 0 ? 0 : (x >> 31)`

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

1. Consider program expression `x >> 31`:
   - Under **condition** `x >= 0`, this expression is just "0". So replace it with `x >= 0 ? 0 : (x >> 31)`
   - Also, under **condition** `x < 0`, this expression is just "1", so further replace `x<0 ? 1 : ( x>=0?0:x>>31 )`

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

1. Consider program expression `x >> 31`:
   - Under **condition** `x >= 0`, this expression is just "0". So replace it with `x >= 0 ? 0 : (x >> 31)`
   - Also, under **condition** `x < 0`, this expression is just "1", so further replace `x<0 ? 1 : ( x>=0?0:x>>31 )`

After this step, **Bitwise Branching** translates above program to:

```
1 int r, s, x;
2 while(x>0){
3   s = x>=0 ? 0 : ( x<0 ? 1 : x >> 31);
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

1. Consider program expression `x >> 31`:

- Under **condition** `x >= 0`, this expression is just "0". So replace it with `x >= 0 ? 0 : (x >> 31)`
- Also, under **condition** `x < 0`, this expression is just "1", so further replace `x<0 ? 1 : ( x>=0?0:x>>31 )`

After this step, **Bitwise Branching** translates above program to:

```
1 int r, s, x;
2 while(x>0){
3   s = x>=0 ? 0 : ( x<0 ? 1 : x >> 31);
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

General Rule: $e_1 \geq 0 \wedge e_2 = \texttt{CHAR\_BIT} * \texttt{sizeof}(e_1) - 1 \vdash_E e_1 \texttt{>>} e_2 \rightsquigarrow 0$

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3    s = x >> 31;
4    x--;
5    r = x + (s&(1-s));
6    if (r<0) error();
7 }
```

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

2. Now consider expression `s&(1-s)`:

- Under **condition** `s>=0`$\wedge$`(1-s)=1`, this expression is equal to `s&1`, which is `s%2`, so replace it with `(s>=0&&(1-s)==1?s%2 :(s&(1-s)))`

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

2. Now consider expression `s&(1-s)`:

- Under **condition** s>=0∧(1-s)=1, this expression is equal to s&1, which is s%2, so replace it with `(s>=0&&(1-s)==1?s%2 :(s&(1-s)))`

After this 2nd step, **Bitwise Branching** translates this program to:

```
1 int r, s, x;
2 while(x>0){
3   s = x>=0 ? 0 : ( x<0 ? 1 : x >> 31);
4   x--;
5   r = x + (s>=0 && (1-s)==1 ? s%2 : (s&(1-s)));
6   if (r<0) error();
7 }
```

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

2. Now consider expression `s&(1-s)`:

- Under **condition** s>=0∧(1-s)=1, this expression is equal to s&1, which is s%2, so replace it with `(s>=0&&(1-s)==1?s%2 :(s&(1-s)))`

After this 2nd step, **Bitwise Branching** translates this program to:

```
1 int r, s, x;
2 while(x>0){
3   s = x>=0 ? 0 : ( x<0 ? 1 : x >> 31);
4   x--;
5   r = x + (s>=0 && (1-s)==1 ? s%2 : (s&(1-s)));
6   if (r<0) error();
7 }
```

General rule: $e_1 \geq 0 \wedge e_2 = 1 \vdash_E e_1 \& e_2 \rightsquigarrow e_1 \% 2$

Rewriting Rules

Table: Rewriting rules for arithmetic expressions.

Linear Condition		BV Expr.	Linear Apx.	
$e_1 = 0$	$\vdash_E$	$e_1 \& e_2$	$\rightsquigarrow 0$	[R-And-0]
$(e_1 = 0 \vee e_1 = 1) \wedge e_2 = 1$	$\vdash_E$	$e_1 \& e_2$	$\rightsquigarrow e_1$	[R-And-1]
$(e_1 = 0 \vee e_1 = 1) \wedge (e_2 = 0 \vee e_2 = 1)$	$\vdash_E$	$e_1 \& e_2$	$\rightsquigarrow e_1 \&\& e_2$	[R-And-LOG]
$e_1 \geq 0 \wedge e_2 = 1$	$\vdash_E$	$e_1 \& e_2$	$\rightsquigarrow e_1 \% 2$	[R-And-LBS]
$e_2 = 0$	$\vdash_E$	$e_1 \mid e_2$	$\rightsquigarrow e_1$	[R-Or-0]
$(e_1 = 0 \vee e_1 = 1) \wedge e_2 = 1$	$\vdash_E$	$e_1 \mid e_2$	$\rightsquigarrow 1$	[R-Or-1]
$e_2 = 0$	$\vdash_E$	$e_1 \hat{} e_2$	$\rightsquigarrow e_1$	[R-Xor-0]
$e_1 = e_2 = 0 \vee e_1 = e_2 = 1$	$\vdash_E$	$e_1 \hat{} e_2$	$\rightsquigarrow 0$	[R-Xor-Eq]
$(e_1 = 1 \wedge e_2 = 0) \vee (e_1 = 0 \wedge e_2 = 1)$	$\vdash_E$	$e_1 \hat{} e_2$	$\rightsquigarrow 1$	[R-Xor-Neq]
$e_1 \geq 0 \wedge e_2 = \texttt{CHAR\_BIT} * \texttt{sizeof}(e_1) - 1$	$\vdash_E$	$e_1 >> e_2$	$\rightsquigarrow 0$	[R-RightShift-Pos]
$e_1 < 0 \wedge e_2 = \texttt{CHAR\_BIT} * \texttt{sizeof}(e_1) - 1$	$\vdash_E$	$e_1 >> e_2$	$\rightsquigarrow -1$	[R-RightShift-Neg]

- Judgment $\mathcal{C} \vdash_E e_{bv} \rightsquigarrow e_{int}$ means under **condition** $\mathcal{C}$ bitvector expression e_{bv} can be approximated with linear expression e_{int}.
- Then, apply a substitution δ, and replace e_{bv} with `if-then-else` expression $\mathcal{C}\delta \ ? \ e_{int}\delta : e_{bv}$

Weakening Rules

Table: Weakening rules for **Relational Expressions** & **Assignment Statements**.

Linear Condition		Statement	Linear Approximation	
$e_1 \geq 0 \wedge e_2 \geq 0$	$\vdash_S$	r op$_{le}$ e_1&e_2	$\rightsquigarrow$ r<=e_1 && r<=e_2	[W-And-Pos]
$e_1 < 0 \wedge e_2 < 0$	$\vdash_S$	r op$_{le}$ e_1&e_2	$\rightsquigarrow$ r<=e_1 && r<=e_2 && r<0	[W-And-Neg]
$e_1 \geq 0 \wedge e_2 < 0$	$\vdash_S$	r op$_{eq}$ e_1&e_2	$\rightsquigarrow$ 0<=r && r<=e_1	[W-And-Mix]
$(e_1 = 0 \vee e_1 = 1) \wedge (e_2 = 0 \vee e_2 = 1)$	$\vdash_S$	$(e_1 \mid e_2)$==0	$\rightsquigarrow$ e_1==0 && e_2==0	[R-Or-LOG]
$e_1 \geq 0 \wedge$ is_const(e_2)	$\vdash_S$	r op$_{ge}$ e_1|e_2	$\rightsquigarrow$ r>=e_2	[W-Or-Const]
$e_1 \geq 0 \wedge e_2 \geq 0$	$\vdash_S$	r op$_{ge}$ e_1|e_2	$\rightsquigarrow$ r>=e_1 && r>=e_2	[W-Or-Pos]
$e_1 < 0 \wedge e_2 < 0$	$\vdash_S$	r op$_{eq}$ e_1|e_2	$\rightsquigarrow$ r>=e_1 && r>=e_2 && r<0	[W-Or-Neg]
$e_1 \geq 0 \wedge e_2 < 0$	$\vdash_S$	r op$_{eq}$ e_1|e_2	$\rightsquigarrow$ e_2<=r && r<0	[W-Or-Mix]
$e_1 \geq 0 \wedge e_2 \geq 0$	$\vdash_S$	r op$_{ge}$ e_1^e_2	$\rightsquigarrow$ r>=0	[W-XOr-Pos]
$e_1 < 0 \wedge e_2 < 0$	$\vdash_S$	r op$_{ge}$ e_1^e_2	$\rightsquigarrow$ r>=0	[W-XOr-Neg]
$e_1 \geq 0 \wedge e_2 < 0$	$\vdash_S$	r op$_{le}$ e_1^e_2	$\rightsquigarrow$ r<0	[W-XOr-Mix]
$e_1 \geq 0$	$\vdash_S$	r op$_{le}$ ~e_1	$\rightsquigarrow$ r<0	[W-Cpl-Pos]
$e_1 < 0$	$\vdash_S$	r op$_{ge}$ ~e_1	$\rightsquigarrow$ r>=0	[W-Cpl-Neg]

op$_{le}$ $\in$ {<,<=,==, := }, op$_{ge}$ $\in$ {>,>=,==, := }, and op$_{eq}$ $\in$ {==, := }

- Judgment $\mathcal{C} \vdash_S s_{bv} \rightsquigarrow s_{int}$ means under **condition** $\mathcal{C}$ bitvector statement s_{bv} can be approximated with linear statement s_{int}.
- Then, apply a substitution δ, and replace s_{bv} with `if-then-else` statement `if` $\mathcal{C}\delta$ `then assume(`$s_{int}\delta$`) else` s_{bv}

Weakening Rules and Termination

```
1 a = *;
2 assume(a>0);
3 while(x>0){
4   a--;
5   x = x & a;
6 }
```

Weakening Rules and Termination

$$e_1 \geq 0 \wedge e_2 \geq 0 \quad \vdash_S \quad r \ \mathtt{op}_{le} \ e_1 \& e_2 \quad \rightsquigarrow r <= e_1 \ \&\& \ r <= e_2 \quad [\text{W-And-Pos}]$$

```
1  a = *;
2  assume(a>0);
3  while(x>0){
4     a--;
5     x = x & a;
6  }
```

- $\mathcal{I} : x > 0 \wedge a > 0$
- $T : x{>}0 \wedge a'{=}a{-}1 \wedge x'{=}x\&a'$
- Tools fail to show:
  $\mathcal{I} \wedge T \wedge x'{>}0 \implies \mathcal{I}'$

Weakening Rules and Termination

$$e_1 \geq 0 \wedge e_2 \geq 0 \quad \vdash_S \quad r \ \mathrm{op}_{le} \ e_1 \& e_2 \quad \rightsquigarrow r <= e_1 \ \&\& \ r <= e_2 \quad [\text{W-And-Pos}]$$

```
1  a = *;
2  assume(a>0);
3  while(x>0){
4     a--;
5     x = x & a;
6  }
```

- $\mathcal{I} : x > 0 \wedge a > 0$
- $T : x{>}0 \wedge a'{=}a{-}1 \wedge x'{=}x\&a'$
- Tools fail to show:
  $\mathcal{I} \wedge T \wedge x'{>}0 \implies \mathcal{I}'$

```
1  a = *; assume(a > 0);
2  while (x > 0) {
3     { x > 0 ∧ a > 0 }
4     a--;
5     if (x >= 0 && a >= 0)
6     then { x = *; assume(x <= a); }
7     else { x = x & a; }
8  }
```

Weakening Rules and Termination

$$e_1 \geq 0 \wedge e_2 \geq 0 \quad \vdash_S \quad r \ \mathrm{op}_{le} \ e_1 \& e_2 \quad \rightsquigarrow r <= e_1 \ \&\& \ r <= e_2 \quad \text{[W-And-Pos]}$$

```
1  a = *;
2  assume(a>0);
3  while(x>0){
4     a--;
5     x = x & a;
6  }
```

- $\mathcal{I} : x > 0 \wedge a > 0$
- $T : x{>}0 \wedge a'{=}a{-}1 \wedge x'{=}x\&a'$
- Tools fail to show:
  $\mathcal{I} \wedge T \wedge x'{>}0 \implies \mathcal{I}'$

```
1  a = *; assume(a > 0);
2  while (x > 0) {
3     { x > 0 ∧ a > 0 }
4     a--;
5     if (x >= 0 && a >= 0)
6     then { x = *; assume(x <= a); }
7     else { x = x & a; }
8  }
```

- $T' = x{>}0 \wedge a' = a{-}1 \wedge ((x{\geq}0 \wedge a'{\geq}0 \wedge x'{\leq}a') \vee$ ~~$(\neg(x \geq 0 \wedge a' \geq 0) \wedge x' = x\&a'))$~~
- Tools can prove that $\mathcal{I} \wedge T' \wedge x'{>}0 \implies \mathcal{I}'$, ranking function $\mathcal{R}(x, a) = a$

Our experiments show bitwise branching also works well for LTL (we will see in the next slides).

Implementation and Benchmarks

Bitwise branching implementation

- A fork of ULTIMATE repository[a]
- Recursive AST transformation during ULTIMATE's translation from C to Boogie

[a] https://github.com/ultimate-pa/ultimate

Implementation and Benchmarks

Bitwise branching implementation

- A fork of ULTIMATE repository[a]
- Recursive AST transformation during ULTIMATE's translation from C to Boogie

[a] https://github.com/ultimate-pa/ultimate

Contributed Benchmarks

1. **ReachBitBench, TermBitBench, LTLBitBench** for reachability, termination, LTL verification, respectively.
2. **BitHacks**, online code optimization[a] adapted to termination, LTL verification.

[a] https://graphics.stanford.edu/~seander/bithacks.html

Experiments: Reachability with various solvers

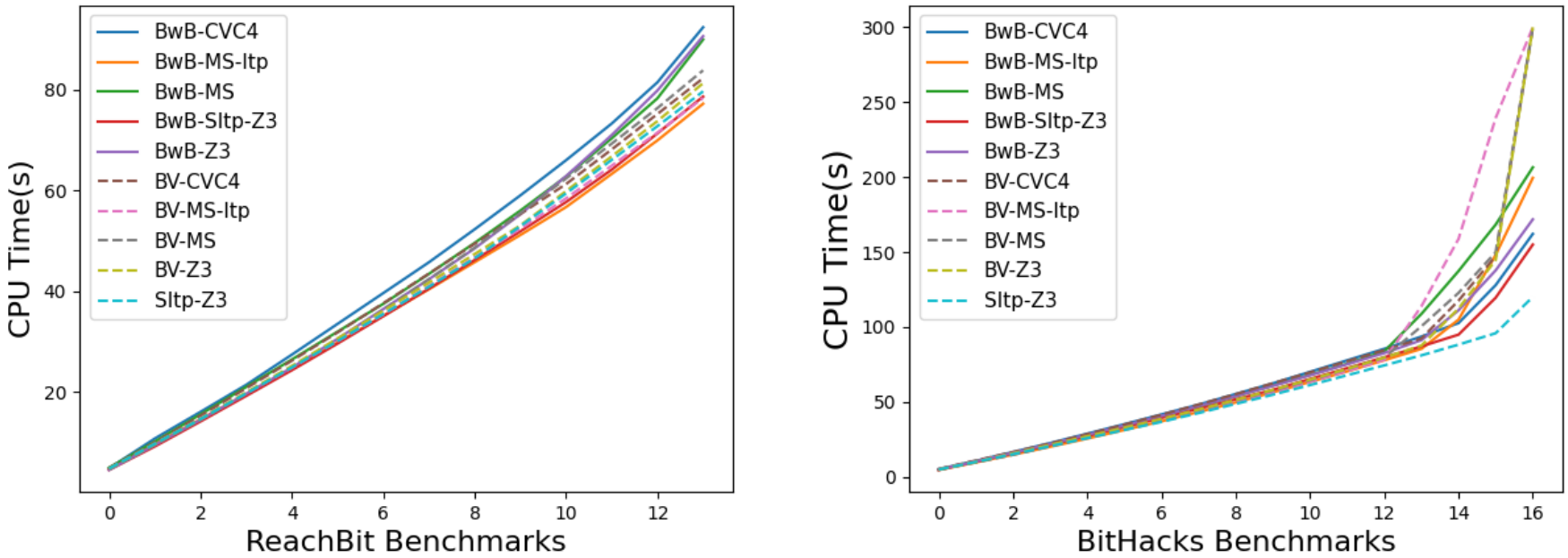


Figure: Performance of UltimateBwB with bitwise branching "BwB" in *integer mode* (solid lines) versus Ultimate (dashed lines, "BV" indicating *bitvector mode*) on bitvector programs, using various SMT solvers.
Default Ultimate (integer mode SItp-Z3) returns *Unknown* for 10/12 "ReachBit", 16/16 "BitHacks".

Experiments: Termination and LTL

State-of-the-art verification tools

Tool	BitVec.	Term.	LTL
Ultimate	Limited	Yes	Yes
AProVE	Yes	Yes	No
KITTeL	Yes	Yes	No
CPAchecker	Limited	Yes	No
2LS	Yes	Yes	No
UltimateBwB	Yes	Yes	Yes

Experiments: LTL

LTL Results Overview

	(iv) Bithacks		(iii) LTLBit Bench	
	ULTIMATE	w. BwB	ULTIMATE	w. BwB
✔ (Satisfied)	3	10	-	21
✗ (Unsatisfied)	-	7	-	20
?(Unknown)	21	5	42	-
T (Time Out)	1	1	-	1
M (Out of Memory)	1	3	-	-

- BitHacks, 18 satisfied, 8 violated.
- LTLBitBench, 22 programs satisfying LTL property, 20 programs are violated.

Part 3

LTL of De-compiled Binaries

Why binary verification & challenges

- Why: Executable is the one runs on machine, compiler errors, optimizations, proprietary software, malware etc..
- Challenges: Disassembly (function boundaries, symbol table, stack frame), control flow recovery etc..

Decompiled code needs bitwise branching

`if(x <= 1)`[a] in de-compiled code:

```
(((tmp_42!=0u)&1)&((((tmp_44 == 0u)&
1^(((((tmp_44^tmp_45)+tmp_45)==2u)&1)))&1)))&1
```

Decompilation is an approach for binary verification.

[a] http://github.com/ultimate-pa/ultimate/blob/dev/trunk/examples/LTL/simple/PotentialMinimizeSEVPABug.c

Challenges Through An Example De-compiled Binary

1. Emulated environment

```
store i64 %0, i64* getelementptr inbounds (%struct.State, %struct.State*
    @__mcsema_reg_state, i64 0, i32 6, i32 11, i32 0, i32 0), align 8
```

2. Emulation state in procedure calls

```
%10 = tail call %struct.Memory* @sub_401111_main(%struct.State* nonnull
```

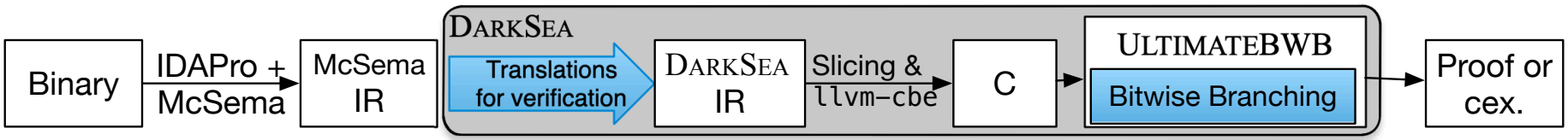
3. Emulation with nested structs, concrete addressing

```
tmp__4 = (&tmp__1->field6.field1.field0);
tmp__5 = (&tmp__4->field0);
```

4. Heavy use of bitwise operations

```
tmp__30 = llvm_lshr_u32(tmp__29, 31);
tmp__31 = (((((tmp__30 == 0u)&1)) & (((~((((tmp__29 == 0u)&1))))&1)))&1);
```

DarkSea Overview



DarkSea translations to re-target lifting for verification

- *Run-time environment.*
- *Passing emulation state through procedures.*
- *Nested structures.*
- *Property-directed slicing.*

DarkSea: `https://github.com/cyruliu/darksea`

LTL Experiments on Lifted Binaries

Table: Ultimate vs. DarkSea on decompiled programs with LTL properties.

Benchmark	Property	Exp.	Ultimate Time	Ultimate Result	DarkSea Time	DarkSea Result
01-exsec2.s.c	$\Diamond(\Box x = 1)$	✔	4.45s	☢	11.23s	✔
01-exsec2.s.f.c.c	$\Diamond(\Box x \neq 1)$	✗	6.31s	☢	10.36s	✗
SEVPA_gccO0.s.c	$\Box(x > 0 \Rightarrow \Diamond y = 0)$	✔	6.31s	☢	22.92s	✔
SEVPA_gccO0.s.f.c	$\Box(x > 0 \Rightarrow \Diamond y = 2)$	✗	5.16s	?	14.92s	✗
acqrel.simplify.s.c	$\Box(x = 0 \Rightarrow \Diamond y = 0)$	✔	5.17s	☢	9.00s	✔
acqrel.simplify.s.f.c.c	$\Box(x = 0 \Rightarrow \Diamond y = 1)$	✗	6.06s	☢	17.60s	✗
exsec2.simplify.s.c	$\Box\Diamond x = 1$	✔	4.92s	☢	5.60s	✔
exsec2.simplify.s.f.c.c	$\Box\Diamond x \neq 1$	✗	4.55s	☢	6.28s	✗

Part 4

Polynomial Program Example

LTL Property:

$\Box\Diamond(y == 1))$

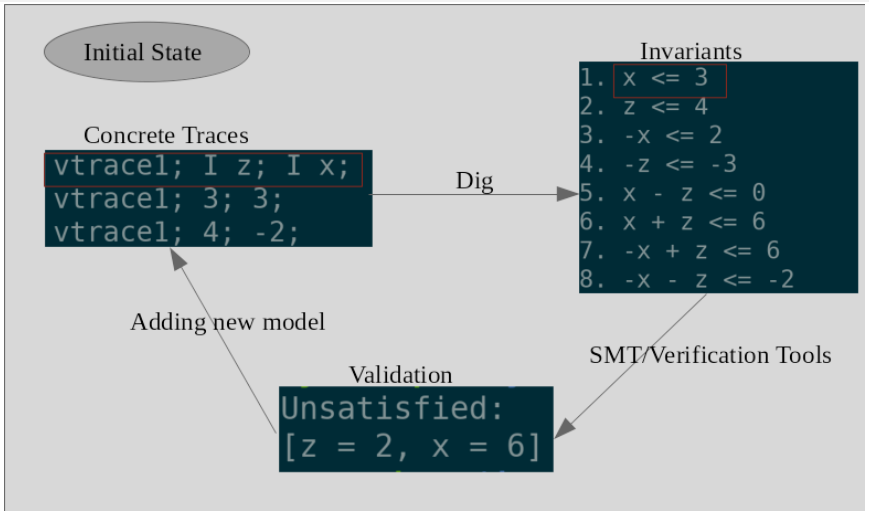
```
1 int x, y, z;
2 while(1){
3    z = nondet();
4    x = -z*z+2*z+6;
5    y = 0;
6   if(x<=7){
7        y=1;
8   }
9 }
```

Polynomial Program Example

LTL Property:

$\Box\Diamond(y == 1))$

```
1 int x, y, z;
2 while(1){
3     z = nondet();
4     x = -z*z+2*z+6;
5     y = 0;
6   if(x<=7){
7         y=1;
8   }
9 }
```

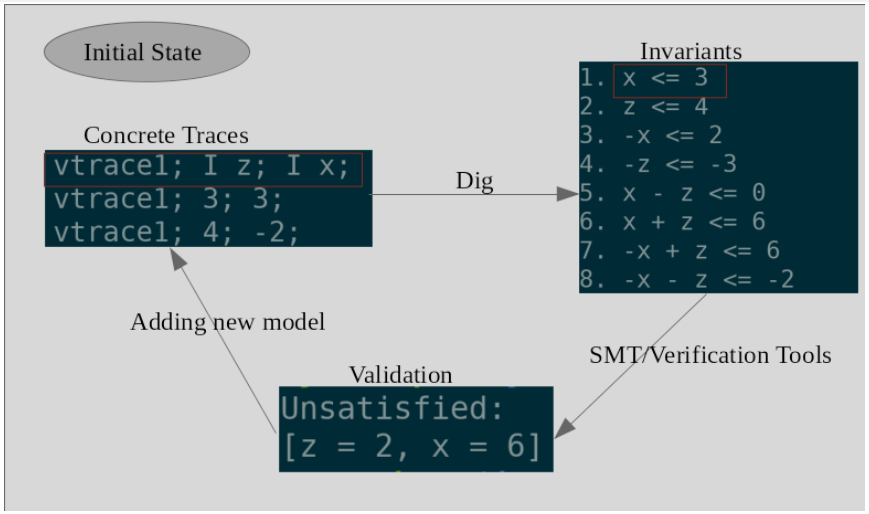


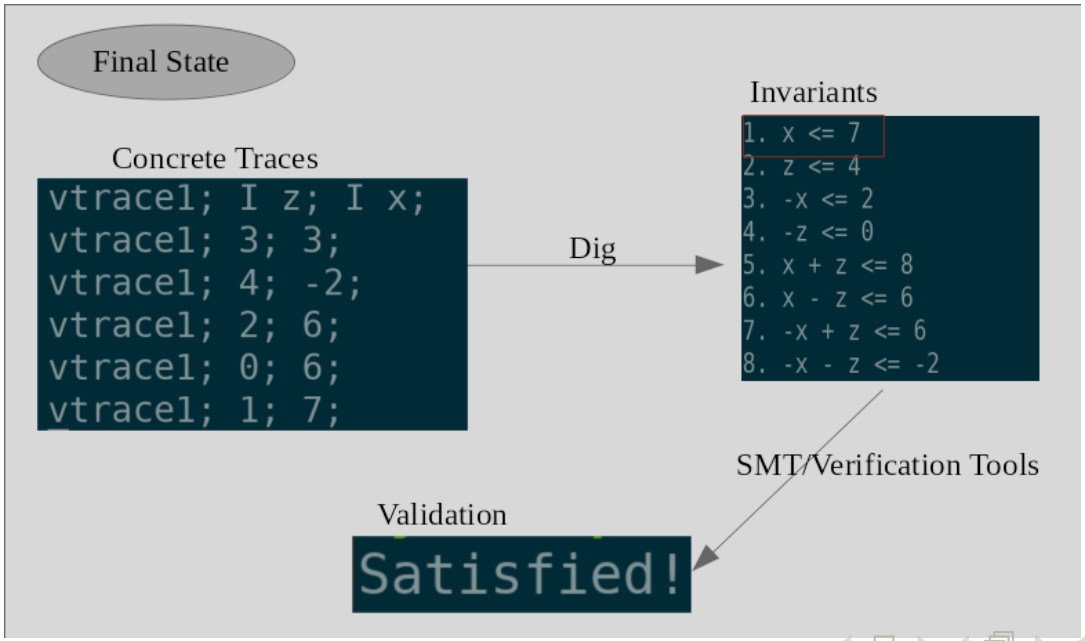
Polynomial Program Example

LTL Property:

$\Box\Diamond(y == 1))$

```
1 int x, y, z;
2 while(1){
3     z = nondet();
4     x = -z*z+2*z+6;
5     y = 0;
6   if(x<=7){
7         y=1;
8   }
9 }
```





Summary of the Proposed Approach

Static analysis and transformation:

- Locate and identify nonlinear expressions, instrument program with trace functions and fresh variables to record the concrete data transition.
- Further instrument the program with selected invariants from dynamic step, run static analysis tools.

Summary of the Proposed Approach

Static analysis and transformation:

- Locate and identify nonlinear expressions, instrument program with trace functions and fresh variables to record the concrete data transition.
- Further instrument the program with selected invariants from dynamic step, run static analysis tools.

Dynamic inferring:

- Random sampling concrete traces.
- Use dynamic tools to run and infer program invariants at trace locations of interest.
- Find the effective linear invariant for nonlinear expressions.

Part 5

Contributions and Findings

Part 2:

(**My role: theory, implementation, benchmarks, experiments**.)

- A novel theory of source level bitwise branching.
- An implementation of bitwise branching within ULTIMATE framework.
- Evaluations show that bitwise branching incurs negligible overhead.
- New benchmarks suites for reachability, termination [3], and LTL.

[3] `gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/main/c/termination-bwb`

Contributions and Findings

Part 2:
(**My role: theory, implementation, benchmarks, experiments**.)

- A novel theory of source level bitwise branching.
- An implementation of bitwise branching within ULTIMATE framework.
- Evaluations show that bitwise branching incurs negligible overhead.
- New benchmarks suites for reachability, termination [3], and LTL.

Part 3:
(**My role: benchmarks, partial implementation, experiments**.)

- Translations that re-target lifted binaries to verification.
- DARKSEA tool chain prepares decompiled programs for verification.
- A new benchmark suite for LTL of binary executables.
- Experimental results showing that DARKSEA is the first tool to verify LTL properties of binary executables.

[3] `gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/main/c/termination-bwb`

Proposed Research

Part 4: (Expected)
(**My role: implementaion, benchmarks, experiments, contributed to algorithm design**.)

- A novel strategy, combining dynamic and static analysis in order to verify programs with bitvector and other non-linear expressions.
- (Proposed) An algorithm.
- (Proposed) An implementation.
- (Proposed) An evaluation.

Expected contributions evaluation

- A submitted peer review paper.
- A working artifact with sets of benchmarks.

Q & A

Thank You!

Basic Notations

Standard notions of programs and semantics.

- $\Sigma : Var \to Val$, a state space mapping variables to values.
- $[\![exp]\!] : \Sigma \to Val$, expressions semantics.
- $[\![stmt]\!] : \Sigma \to \mathcal{P}(\Sigma)$, statements semantics.
- $[\![P]\!]$: traces of program P.

Define rewriting rules for expressions and statements.

- $T_E : exp \to exp$, rewriting rules application for expressions.
- $T_S : stmt \to stmt$, weakening rules application for statements.

Soundness Proof

Lemma (Rule correctness)

For every rule $\mathcal{C} \vdash_E e \rightsquigarrow e'$, $\forall \sigma.\ \mathcal{C}(\sigma) \Rightarrow e' = \sigma(e), [\![e]\!] = [\![e']\!]$.
For every rule $\mathcal{C} \vdash_S s \rightsquigarrow s'$, $\forall \sigma.\ \mathcal{C}(\sigma) \Rightarrow s' = \sigma(s), [\![s]\!] \subseteq [\![s']\!]$.

Theorem (Soundness)

For every P, T_E, T_S, $[\![P]\!] \subseteq [\![T_S\{T_E\{P\}\}]\!]$.

Prove by induction on traces, T_E preserves traces equivalence, T_S preserves trace inclusion.

Experiments: Termination

Termination Results Overview

	(ii) TermBitBench						(i) AproveBench					
	AProVE	CPAchecker	KITTeL	2LS	Ultimate	UltimateBwB	AProVE	CPAchecker	KITTeL	2LS	Ultimate	UltimateBwB
✔ (Terminating)	5	1	7	8	2	18	1	3	3	14	2	2
↯✔ (FN)	1	-	-	-	-	-	-	-	-	-	-	-
✗ (Nonterminating)	6	10	-	8	-	13	-	-	-	-	-	-
↯✗ (FP)	2	7	-	3	-	-	-	10	-	-	2	6
?(Unknown)	14	13	-	-	29	-	10	3	-	1	14	8
T (Time Out)	3	-	19	12	-	-	7	-	10	2	-	1
M (Out of Memory)	-	-	-	-	-	-	-	-	-	1	-	1
☢ (Crash)	-	-	5	-	-	-	-	2	5	-	-	-

- TermBitBench, 18 terminating, 13 non-terminating.
- AproveBench, 18/118 or 15% are bitvector programs.

Termination Experiments on Lifted Binaries

Table: Termination of Lifted Binaries, with and without DarkSea translations.

	Raw McSema						**DarkSea transl.**					
	AProVE	CPAchecker	KITTeL	2LS	Ultimate	UltimateBwB	AProVE	CPAchecker	KITTeL	2LS	Ultimate	UltimateBwB
✔	-	-	-	-	-	-	-	-	-	-	18	18
☢	-	18	-	-	3	-	-	-	-	-	-	-
M	-	-	-	-	-	3	-	-	-	-	-	-
T	-	-	18	-	15	15	-	18	18	-	-	-
?	18	-	-	18	-	-	18	-	-	18	-	-

Bitvector Example

LTL Property: $G((x > 0) \implies F(y == 0))$

```
1 while(c<50){
2   c++;
3   d = nondet();
4   if(d<0) d = d*(-1);
5   x = nondet();
6   y = 1;
7   while(x>0){
8     d--;
9     x = (x&d) -1;
10    if (x<=1){
11      y=0;
12    }
13  }
```

Bitvector Example

LTL Property: $G((x > 0) \implies F(y == 0))$

```
1  while(c<50){
2    c++;
3    d = nondet();
4    if(d<0) d = d*(-1);
5    x = nondet();
6    y = 1;
7    while(x>0){
8      d--;
9      x = (x&d) -1;
10     if (x<=1){
11       y=0;
12     }
13   }
```

Concrete Traces

```
vtrace25; I x; I y; I c; I d; I pre_x
vtrace26; I x; I y; I c; I d; I pre_x
vtrace28; I x; I y; I c; I d; I pre_x
vtrace30; I x; I y; I c; I d; I pre_x
vtrace25; 886; 1; 1; 4383; 1113016132
vtrace28; 277; 1; 1; 4382; 886
vtrace28; 276; 1; 1; 4381; 277
vtrace28; 275; 1; 1; 4380; 276
vtrace28; 274; 1; 1; 4379; 275
vtrace28; 273; 1; 1; 4378; 274
vtrace28; 272; 1; 1; 4377; 273
vtrace28; 271; 1; 1; 4376; 272
vtrace28; 262; 1; 1; 4375; 271
vtrace28; 261; 1; 1; 4374; 262
```

Run with Dig

Invariants

```
vtrace25 (7 invs):
1. -y <= 0
2. -x + y <= 1
3. -d - x <= -1
4. -pre_x <= -1
5. -d + y <= -1
6. -c + y <= -1
7. -c - x <= -1
vtrace28 (9 invs):
1. -x <= 1
2. -y <= 0
3. -c <= -1
4. -d - x <= -1
5. -d + x <= -1
6. -d + y <= -1
7. -c - x <= -1
8. -pre_x + x <= -1
9. -pre_x + y <= -1
```

Bitvector Example

LTL Property: $G((x > 0) \implies F(y == 0))$

```
1  while(c<50){
2    c++;
3    d = nondet();
4    if(d<0) d = d*(-1);
5    x = nondet();
6    y = 1;
7    while(x>0){
8      d--;
9      x = (x&d) -1;
10     if (x<=1){
11       y=0;
12     }
13   }
```

Concrete Traces

```
vtrace25; I x; I y; I c; I d; I pre_x
vtrace26; I x; I y; I c; I d; I pre_x
vtrace28; I x; I y; I c; I d; I pre_x
vtrace30; I x; I y; I c; I d; I pre_x
vtrace25; 886; 1; 1; 4383; 1113016132
vtrace28; 277; 1; 1; 4382; 886
vtrace28; 276; 1; 1; 4381; 277
vtrace28; 275; 1; 1; 4380; 276
vtrace28; 274; 1; 1; 4379; 275
vtrace28; 273; 1; 1; 4378; 274
vtrace28; 272; 1; 1; 4377; 273
vtrace28; 271; 1; 1; 4376; 272
vtrace28; 262; 1; 1; 4375; 271
vtrace28; 261; 1; 1; 4374; 262
```

Run with Dig

Invariants

```
vtrace25 (7 invs):
1. -y <= 0
2. -x + y <= 1
3. -d - x <= -1
4. -pre_x <= -1
5. -d + y <= -1
6. -c + y <= -1
7. -c - x <= -1
vtrace28 (9 invs):
1. -x <= 1
2. -y <= 0
3. -c <= -1
4. -d - x <= -1
5. -d + x <= -1
6. -d + y <= -1
7. -c - x <= -1
8. -pre_x + x <= -1
9. -pre_x + y <= -1
```

Invariant $-pre\_x + x \leq -1$ shows x is decreasing at bitwise location 9.

Implementation Algorithm

$T_E : exp \rightarrow exp$ to translate expressions.

```
type rule_exp = (exp -> exp -> exp) * (exp -> exp -> exp)
let rec T_E (e:exp) : exp =
  match e with
   | BinOp(⊗,e1,e2) ->
    let e1' = T_E e1 in
    let e2' = T_E e2 in
    let rules = Rules.find_exp(⊗) in
    fold_left (fun acc (cond,repl) ->
      ITE(cond e1' e2',repl e1' e2',acc)
    )  (BinOp(⊗,e1, e2)) rules
   | _ -> e
```

Implementation Algorithm

$T_S : stmt \rightarrow stmt$ to translate assignment statements.

```
type rule_stmt = (exp -> exp -> exp) * (lhs -> exp -> exp -> stmt)
let T_s (s:stmt) : stmt =
  match s with
   | Assign(lhs,BinOp(⊗,e1,e2)) ->
    let e1' = T_E e1 in
    let e2' = T_E e2 in
    let rules = Rules.find_stmt(⊗)in
    fold_left (fun acc (cond,repl) ->
      IfElseStmt(cond e1' e2',repl lhs e1' e2', acc)
    ) (Assign(l, BinOp(⊗, e1, e2) rules
   | _ -> s
```