MANCHESTER
1824
The University of Manchester

Contractors in Software Verification

By Mohannad Aldughaim

Supervised by Lucas Cordeiro and Alexandru Stancu

Problem Statement

- State space explosion problem
  - Too many states to explore
  - Number of concurrent states
  - Number of variables

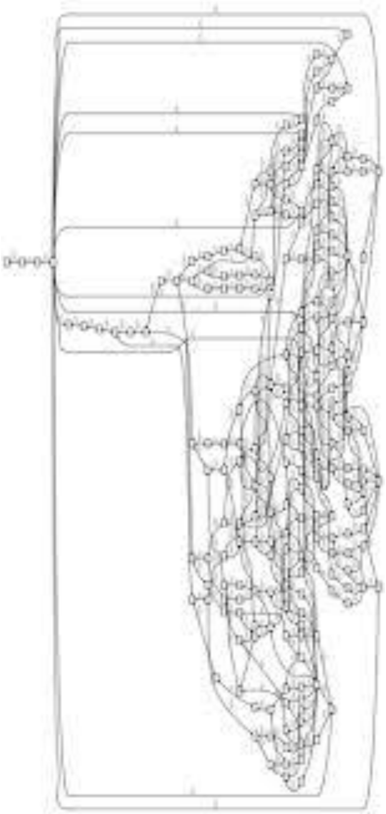
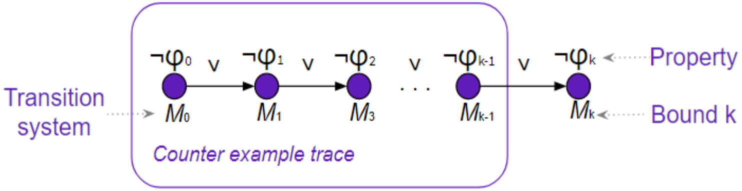


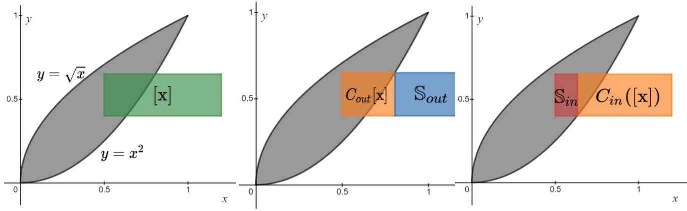
Illustration of state space

Preliminaries

- Bounded Model Checking
  - Check negation of given property up to given depth
- Interval Analysis and Methods
  - Constraint Satisfaction Problem (CSP)
  - Contractors (Outer and Inner)


$\neg\varphi_0$ ∨ $\neg\varphi_1$ ∨ $\neg\varphi_2$ ∨ . . . $\neg\varphi_{k-1}$ ∨ $\neg\varphi_k$
Property
Transition system
M_0
M_1
M_3
M_{k-1}
M_k
Bound k
Counter example trace

Bounded Model Checking



Contractors with: initial domain in green, contracted area in blue (outside solution), red (inside the solution), Orange is the boundary area

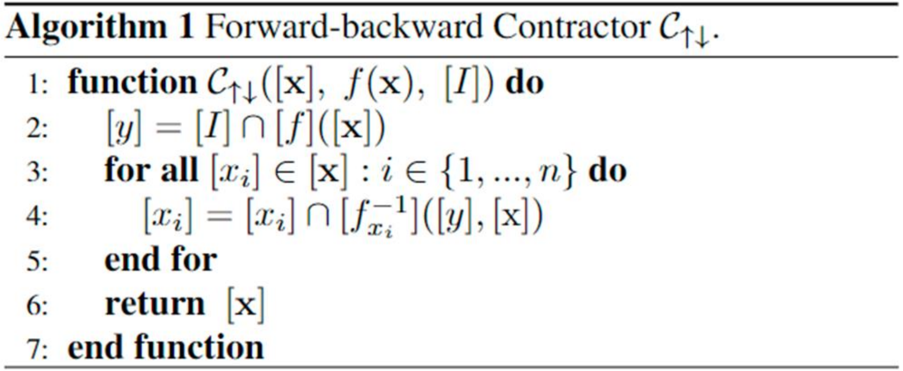
Forward-Backward Contractors

Constraint Satisfaction Problem.

One single constraint.

It contracts in two steps:

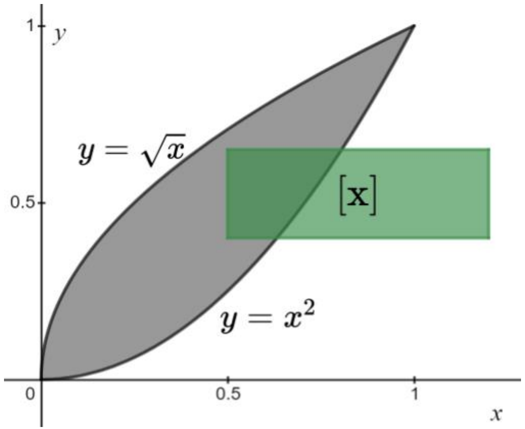
- forward evaluation.
- backward propagation.



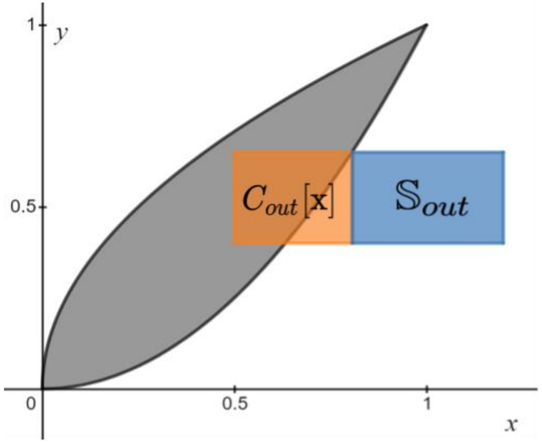
Algorithm 1 Forward-backward Contractor $\mathcal{C}_{\uparrow\downarrow}$.

```
function $\mathcal{C}_{\uparrow\downarrow}([\mathrm{x}],\ f(\mathrm{x}),\ [I])$ do
    $[y] = [I] \cap [f]([\mathrm{x}])$
    for all $[x_i] \in [\mathrm{x}] : i \in \{1, ..., n\}$ do
        $[x_i] = [x_i] \cap [f_{x_i}^{-1}]([y], [\mathrm{x}])$
    end for
    return $[\mathrm{x}]$
end function
```

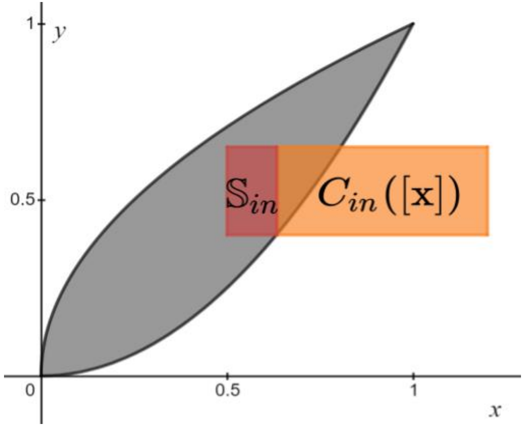
Contractors



Domain



Outer Contractor



Inner Contractor

Why Contractors?

Improved the uncertainty in robotics localization and mapping to provide more guaranteed solutions.

Improved complexity (polynomial) for branch and bound algorithms such as SIVIA.

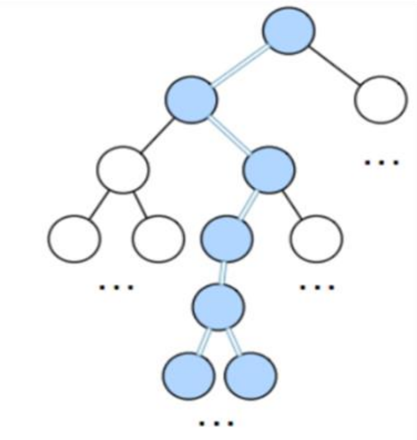
Method 1

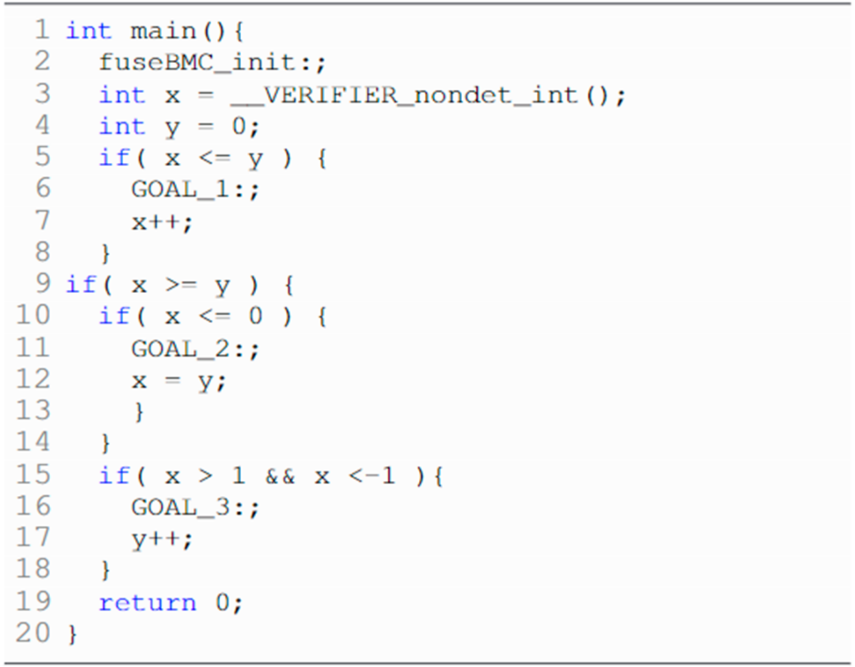
Contractors in *FuSeBMC*

FuSeBMC **v4.0**

Injects goals to test for reachability.

Deeper goals.

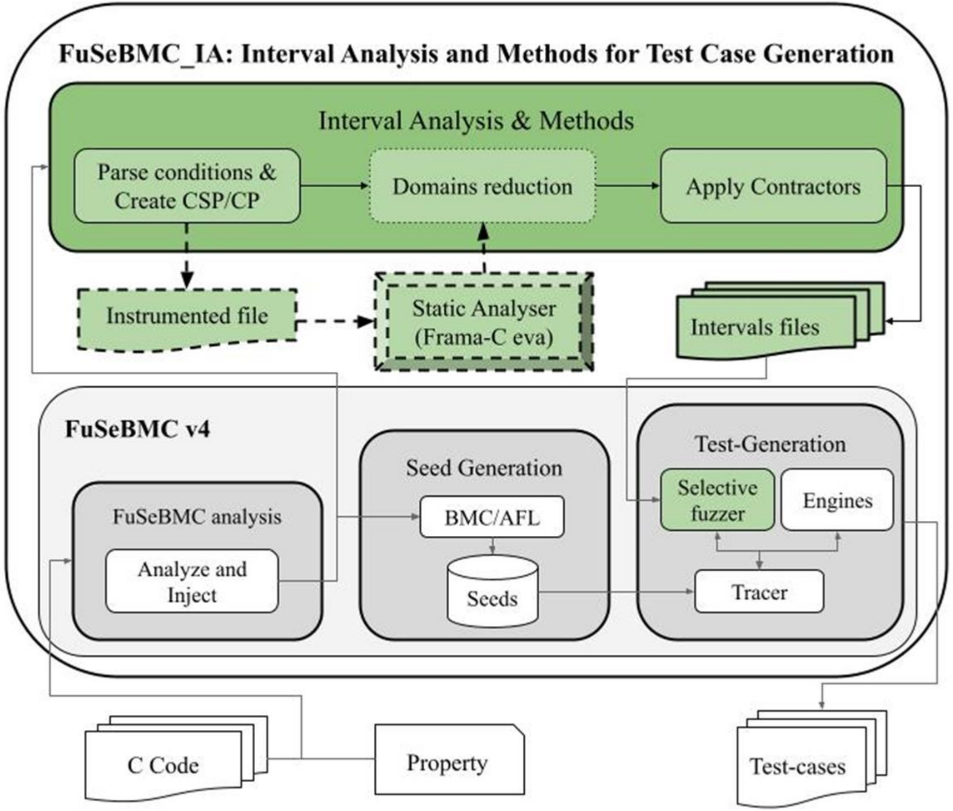




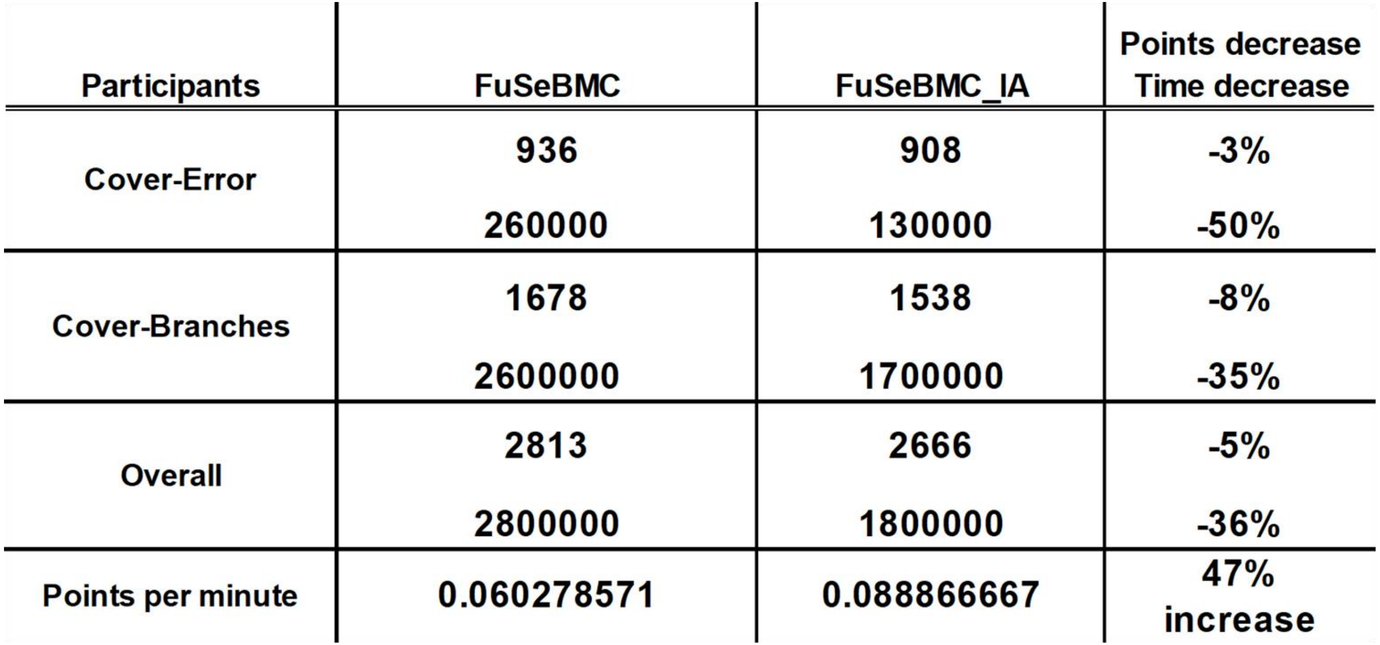
```
int main(){
  fuseBMC_init:;
  int x = __VERIFIER_nondet_int();
  int y = 0;
  if( x <= y ) {
    GOAL_1:;
    x++;
  }
if( x >= y ) {
  if( x <= 0 ) {
    GOAL_2:;
    x = y;
    }
  }
  if( x > 1 && x <-1 ){
    GOAL_3:;
    y++;
  }
  return 0;
}
```

FuSeBMC_IA

- Starting with FuSeBMC analysis. (Goals Instrumented)
- create CSP for each goal
- Produce an instrumented file to Frama-C and run it.
- Apply Contractors for each goal.
- Produce a file with each goal and variables intervals
- we fuzz the PUT with given intervals.
- If a goal is unreachable, we lower the priority for fuzzing it.



Results



Participants	FuSeBMC	FuSeBMC_IA	Points decrease Time decrease
Cover-Error	936 260000	908 130000	-3% -50%
Cover-Branches	1678 2600000	1538 1700000	-8% -35%
Overall	2813 2800000	2666 1800000	-5% -36%
Points per minute	0.060278571	0.088866667	47% increase

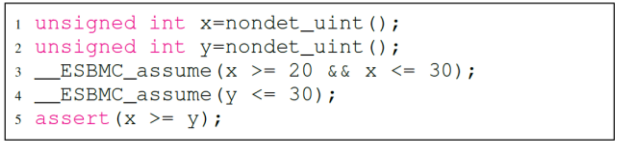
Third place in Test-comp 2023

Method 2

ESBMC `--goto-contractor`

Methodology

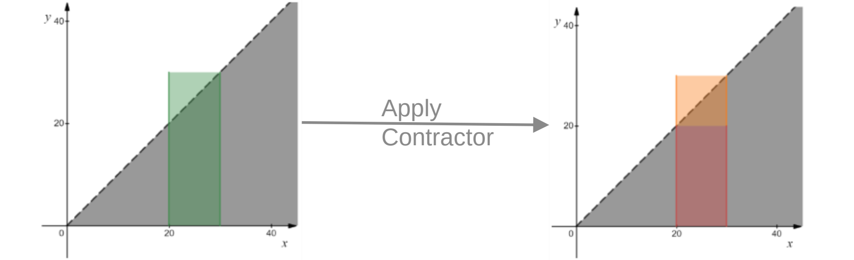
1) Analyze intervals and properties
   - Static Analysis
2) Convert the problem into a CSP
   - Variables, Domains and Constraints
3) Apply contractor to CSP
   - Forward-Backward Contractor
4) Apply reduced intervals back to program.
   - Via `assume()` directive



```
unsigned int x=nondet_uint();
unsigned int y=nondet_uint();
__ESBMC_assume(x >= 20 && x <= 30);
__ESBMC_assume(y <= 30);
assert(x >= y);
```

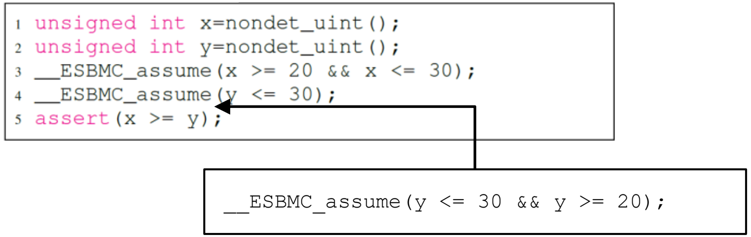
Domain: $[x] = [20, 30]$ and $[y] = [0, 30]$ Constraint: $y - x \leq 0$

$$\begin{array}{lll} f(x) > 0 & I = [0, \infty) & \\ f(x) = y - x & [f(x)_1] = I \cap [y_0] - [x_0] & \text{Forward-step} \\ x = y - f(x) & [x_1] = [x_0] \cap [y_0] - [f(x)_1] & \text{Backward-step} \\ y = f(x) + x & [y_1] = [y_0] \cap [f(x)_1] + [x_1] & \text{Backward-step} \end{array}$$



$[x] = [20, 30]$ and $[y] = [0, 30]$

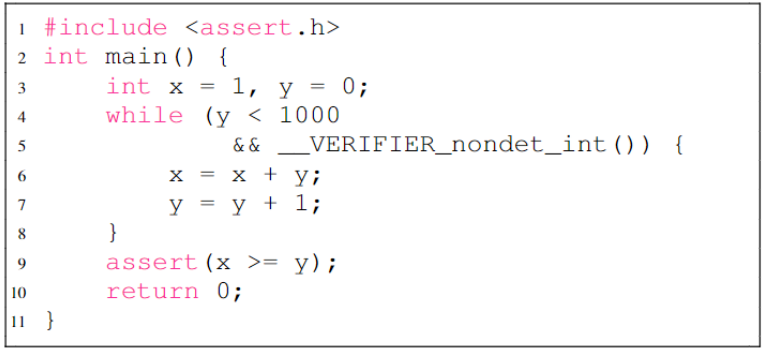
$[x] = [20, 30]$ and $[y] = [20, 30]$



```
unsigned int x=nondet_uint();
unsigned int y=nondet_uint();
__ESBMC_assume(x >= 20 && x <= 30);
__ESBMC_assume(y <= 30);
assert(x >= y);
```

```
__ESBMC_assume(y <= 30 && y >= 20);
```

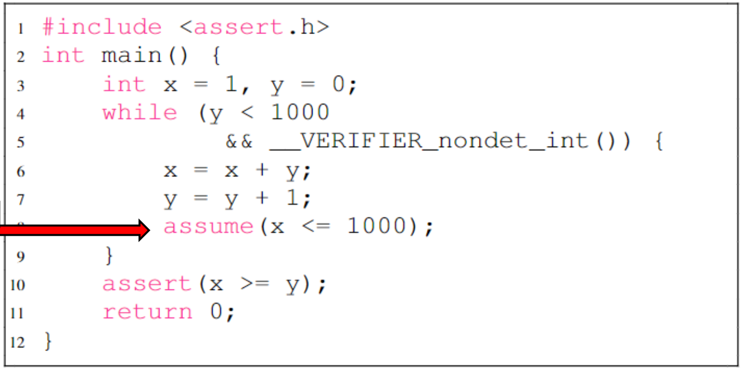
Modify Original Program (Instrument)



```c
#include <assert.h>
int main() {
    int x = 1, y = 0;
    while (y < 1000
            && __VERIFIER_nondet_int()) {
        x = x + y;
        y = y + 1;
    }
    assert(x >= y);
    return 0;
}
```

$$x = [1, Max{-}int] \qquad y = [0, 1000]$$

Instrument assume based on new intervals



```c
#include <assert.h>
int main() {
    int x = 1, y = 0;
    while (y < 1000
            && __VERIFIER_nondet_int()) {
        x = x + y;
        y = y + 1;
        assume(x <= 1000);
    }
    assert(x >= y);
    return 0;
}
```

$$x = [1, 1000] \text{ and } y = [1, 1000]$$

All benchmarks results were done using benchexec

	ESBMC$_{v6}$ contractor inc		**ESBMC$_{v6}$ incremental**		**ESBMC$_{v6}$ k-induction**	
Sub category	**score**	**time**	**score**	**time**	**score**	**time**
loops	**29**	**16582**	25	18525	1	13261
loop-acceleration	**31**	14890	29	15795	28	13734
loop-crafted	3	3608	3	3608	**8**	1723
loop-invgen	11	17584	11	17577	**15**	15683
loop-lit	**15**	12644	15	12645	13	11945
loop-new	10	5656	4	8354	**12**	6285
loop-industry-pattern	16	9060	16	9061	**28**	2923
loops-crafted-1	-26	40065	-26	40105	**-17**	19040
loop-invariants	3	7118	3	7121	**3**	6950
loop-simple	13	921	**13**	921	11	903
loop-zilu	**20**	**10927**	14	13655	16	12650
verifythis	2	3630	2	3630	**2**	3618
nla-digbench	7	23951	7	23941	**8**	23617
nla-digbench-scaling	**540**	**119542**	538	119669	481	141561
total	**674**	286178	654	294608	609	273893

Method 3

Contractors in `--interval-analysis`

Contractors in `--interval-analysis`

Part of `--interval-analysis` option.

```
--interval-analysis-ibex-contractor
```

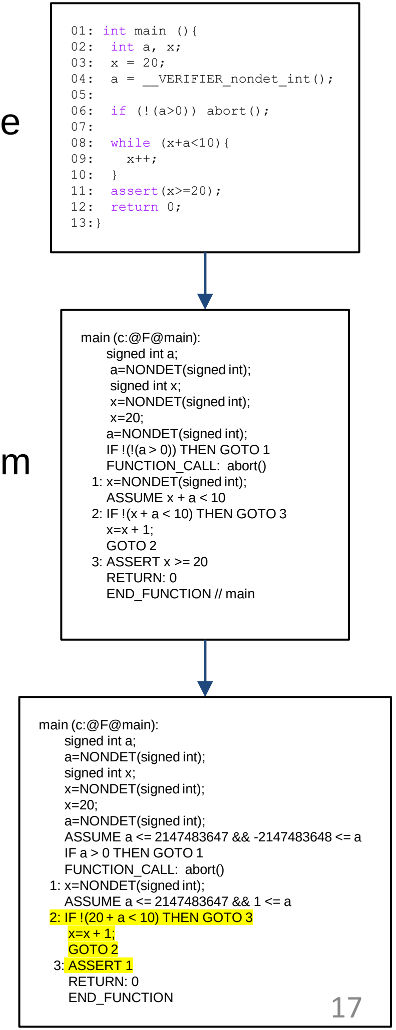
Improve intervals based on conditions.

- *Some improvement in verification time is expected.*
- *More improvement in performance when more than one variable is present in a single constraint.*

Methodology

- On goto-program level
- Parse conditions and apply contractors
- Feed back into interval analysis (where instrumentation happens)

Every goto with condition.



c code

```
int main (){
 int a, x;
 x = 20;
 a = __VERIFIER_nondet_int();

 if (!(a>0)) abort();

 while (x+a<10){
   x++;
 }
 assert(x>=20);
 return 0;
}
```

Goto program

```
main (c:@F@main):
     signed int a;
      a=NONDET(signed int);
      signed int x;
      x=NONDET(signed int);
      x=20;
     a=NONDET(signed int);
     IF !(!(a > 0)) THEN GOTO 1
     FUNCTION_CALL:  abort()
  1: x=NONDET(signed int);
     ASSUME x + a < 10
  2: IF !(x + a < 10) THEN GOTO 3
     x=x + 1;
     GOTO 2
  3: ASSERT x >= 20
     RETURN: 0
     END_FUNCTION // main
```

before : (<20, 20> ; [1, inf])

after : empty vector

```
main (c:@F@main):
     signed int a;
     a=NONDET(signed int);
     signed int x;
     x=NONDET(signed int);
     x=20;
     a=NONDET(signed int);
     ASSUME a <= 2147483647 && -2147483648 <= a
     IF a > 0 THEN GOTO 1
     FUNCTION_CALL:  abort()
  1: x=NONDET(signed int);
     ASSUME a <= 2147483647 && 1 <= a
  2: IF !(20 + a < 10) THEN GOTO 3
      x=x + 1;
      GOTO 2
   3: ASSERT 1
      RETURN: 0
      END_FUNCTION
```

Results (incomplete)

Results are mixed and are still under evaluations.

- *Improvement in some benchmarks*
- *More overhead in most benchmarks.*

Most sv-comp benchmarks contain single variable in a single constraint.

- *Which is evaluated easily without contractors.*

MANCHESTER
1824
The University of Manchester

Future work

Implement contractors natively on ESBMC.

Support operators and functions exclusive to programming languages. (bitwise operators)

Thank you