

Interval Analysis for BMC

Rafael Sá Menezes

Supervisor: Dr. Lucas Cordeiro

Co-Supervisor: Dr. Giles Reger

Agenda



Overview of Interval
Analysis



Interval Analysis in
ESBMC



Extensions for
ESBMC's Interval
Analysis



Preliminary
Experiments



Remarks

Overview



Introduction to Interval Analysis

- The interval analysis consists of computing all values the variables *might* assume at each program statement.
- The analysis can be used to infer properties regarding the program states and flow.

Line	Interval for "a"	Restriction
4	$(-\infty, +\infty)$	None
6	$(-\infty, 100]$	$a \leq 100$
7	$(100, +\infty)$	$a > 100$

```
1 int main()
2 {
3     int a = *;
4
5     while(a <= 100)
6         a++;
7     assert(a>10);
8     return 0;
9 }
```

Soundness and Completeness

- **Soundness:** The program analyzer analysis is sound with respect to property P whenever, for any program p , $\text{analysis}(p) = \text{true}$ implies that p satisfies property P .
- **Completeness:** The program analyzer analysis is complete with respect to property P whenever, for every program p , such that p satisfies P , $\text{analysis}(p) = \text{true}$.
- The interval analysis aims to be sound by **overapproximating** the intervals. In ESBMC, we will still rely on the symbolic execution to guarantee completeness.

Application of Interval Analysis

- This analysis can be used to decrease the number of states that need to be verified as statements might introduce new verification tasks:
 - **Memory safety:** unreachable dereferences, unreachable mallocs, or free.
 - **Reachability:** unreachable assertions, or assertions that always hold.
 - **Concurrency:** new interleavings that need to be verified (exponential!).

Interval Analysis in Bounded Model Checking

- Interval analysis can help BMC by removing unneeded verification states:

```
int main() 0 ref
{
    int a; 2 refs
    if(a < 0 && a > 0)
        while(1) assert(0);
    return 0;
}
```

Contradiction

Unreachable

Interval Analysis in *k*-Induction

- *k*-Induction algorithm “hijacks” loop conditions to nondeterministic values, thus computing intervals become essential

```
1 int main()
2 {
3     unsigned int a = 10;
4     unsigned int b = 1;
5
6     while(a < 50 && *)
7     {
8         a++;
9         b = a*2;
10    }
11
12    assert(b >= a);
13 }
14
```



```
1 #include <assert.h>
2 int main()
3 {
4     unsigned int a = 10;
5     unsigned int b = 1;
6
7     a = *; b = *;
8     assume(a < 50);
9     while(a < 50 && nondet_uint())
10    {
11        assume(a <= 49);
12        a++;
13        b = a*2;
14    }
15
16    assert(b >= a);
17 }
18
```


Summary



Interval Analysis is a technique to obtain the values for all variables in each program statement.



To be sound, the concrete intervals must be inside the abstraction.



The technique can be used to remove assertions that will definitely hold in a program.

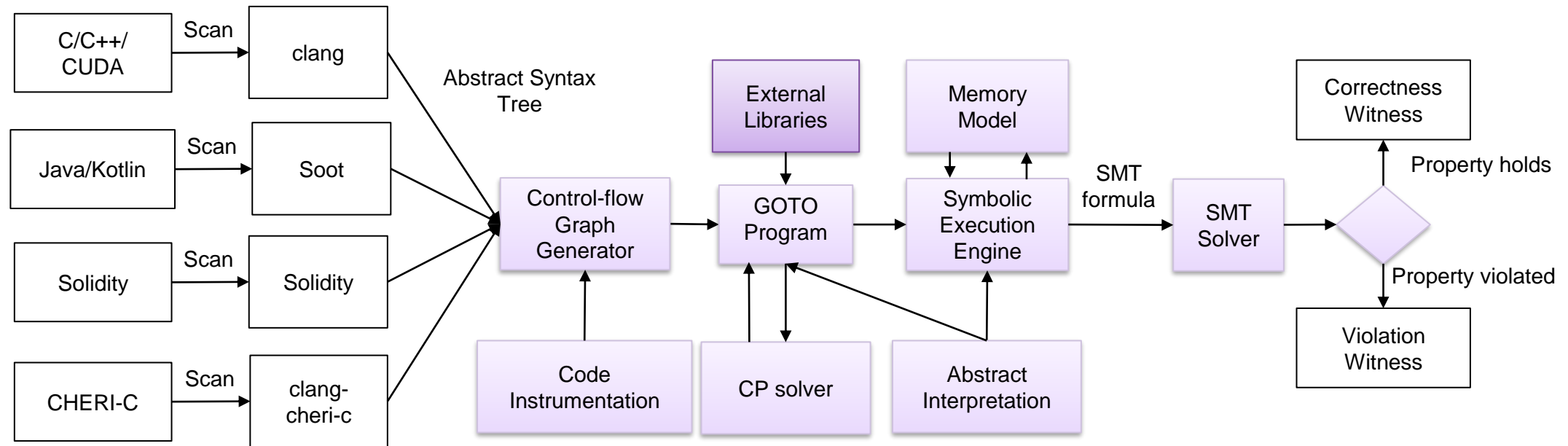


Computing intervals is a requirement for applying the *k-induction* proof.

Interval Analysis in ESBMC



ESBMC architecture



Informal Definition of Intervals

- In ESBMC, the interval has:
 - Lower: represents the lower bound of the interval (or infinity);
 - Upper: represents the upper bound of the interval (or infinity);
- With the following characteristics:
 - Lower is always less or equal than upper.

Computing Intervals

- Restrictions are computed through intersection:

$$(-\infty, \infty) \cap (-\infty, 50) = (-\infty, 50)$$

$$(-\infty, \infty) \cap [50, \infty) = [50, \infty)$$

- Merging is computed with the Hull operation:

$$[3,3] \sqcup [5,5] = [3,5]$$

```
1 int main()  
2 {  
3     int a;  
4     if(a < 50) {  
5         // ...  
6         a = 3;  
7     }  
8     else {  
9         // ...  
10        a = 5;  
11    }  
12    // ...  
13 }
```

Computing Intervals

- For non-loop sequences:
 1. Initialize variable interval to $[-\infty, \infty]$;
 2. Use conditionals to restrict the interval;
 3. Merge intervals after conditionals;

```
1 int main()  
2 {  
3     int a;  
4     if(a < 50) {  
5         // ...  
6         a = 3;  
7     }  
8     else {  
9         // ...  
10        a = 5;  
11    }  
12    // ...  
13 }  
14
```

Computing Intervals

- For non-loop sequences:
 1. Initialize variable interval to $[-\infty, \infty)$.
 2. Use conditionals to restrict the interval.
 3. Merge intervals after conditionals.

Line	Interval for "a"
4	$(-\infty, +\infty)$
5	$(-\infty, 50)$
7	$[3, 3]$
9	$[50, +\infty)$
11	$[5, 5]$
12	$[3, 5]$

```
1 int main()  
2 {  
3     int a;  
4     if(a < 50) {  
5         // ...  
6         a = 3;  
7     }  
8     else {  
9         // ...  
10        a = 5;  
11    }  
12    // ...  
13 }
```

Computing Intervals


```
1 int main()  
2 {  
3     int a;  
4     if(a < 50) {  
5         // ...  
6         a = 3;  
7     }  
8     else {  
9         // ...  
10        a = 5;  
11    }  
12    // ...  
13 }
```

```
1 int main()  
2 {  
3     int a;  
4     if(a < 50) {  
5         __ESBMC_assume(a < 50);  
6         // ...  
7         a = 3;  
8     }  
9     else {  
10        __ESBMC_assume(a >= 50);  
11        // ...  
12        a = 5;  
13    }  
14    __ESBMC_assume(a >= 3 && a <= 5);  
15    // ...  
16 }
```


Limitations

- Computing the intersection of abstract states can result in an underapproximation.


```
1  int main()
2  {
3      int a;
4      if(a > 10 && a < 30) {
5          // ...
6      }
7  }
8
```


$$(10, \infty) \cap [-\infty, 30) = (10, 30)$$

Limitations

- Computing the intersection of abstract states can result in an underapproximation.

```
1 int main()  
2 {  
3     int a = * ? 3 : 11; // [3,11]  
4     int b = * ? 2 : 9;  // [2,9]  
5     if(a < b) {  
6         // ...  
7     }  
8 }
```


$$[3,11] \cap [2,9] = [3,9]$$

Limitations

- Computing the intersection of abstract states can result in an underapproximation.

```
1 int main()  
2 {  
3     int a = * ? 1 : 11; // [1,11]  
4     int b = * ? 2 : 9;  // [2,9]  
5     if(a < b) {  
6         // ...  
7     }  
8 }
```

$[1,11] \cap [2,9] = [2,9]$

Limitations

- Too many over approximations: no arithmetic

```
1  int main()  
2  {  
3      int a = 0;  
4      if((a+1) <= 1) {  
5          // a : ?  
6      }  
7  }
```

Infinity

No restrictions over "a"

Summary



ESBMC tracks Integer intervals in a cartesian domain (min, max).



At each branch, it injects the intervals as ASSUMES.



It is very imprecise, so that it has no problem arriving at a fixpoint.

Extensions for ESBMC



Contracting Intervals

- Back to the comparison between two symbolic intervals

```
1 int main()
2 {
3     int a = * ? 1 : 11; // [1,11]
4     int b = * ? 2 : 9;  // [2,9]
5     if(a < b) {
6         // ...
7     }
8 }
```

We can apply contractor algorithms to contract "a" in terms of "b":

Forward: $y = a - b \rightarrow [y] = ([a] - [b]) \cap (-infinity, 0]$

Backwards:

$$[a] = [a] \cap ([b] + [y])$$

$$[b] = [b] \cap ([a] - [y])$$

Resulting in:

$$[a] = [1,9]$$

Interval Arithmetic

- By adding support for interval arithmetic: +, -, *, /. We can get more precise intervals

```
1 int main()
2 {
3     unsigned int a = * ? 0 : 6; // [0,6]
4     if((a + 1) < 5) {
5         // a: [0,3]
6     }
7 }
8
```


Modular Arithmetic

- At each assignment, we can do a modulus operation to restrict the values that a variable can assume

$$\begin{aligned} & \text{int } a = x \\ & a \triangleq f(x) \cap [-2^{31} - 1, 2^{31}] \end{aligned}$$

Modular Arithmetic

- At each assignment, we can do a modulus operation to restrict the values that a variable can assume

```
1 int main()
2 {
3     unsigned char a; // [0, 256]
4     if(a > 1000) {
5         // Bottom
6     }
7 }
```

Modular Arithmetic

- At each assignment, we can do a modulus operation to restrict the values that a variable can assume

```
1  int main()
2  {
3      unsigned char a; // [0, 256]
4      while(a < 1000) {
5          a++;
6      }
7  }
8
```

Loop Widening

- By getting more precise intervals, we will have trouble reaching to a fixpoint. We can accelerate the convergence by applying widening techniques.

```
1  ✓ int foo() {  
2      int a = 0;  
3      while (a <= 1000)  
4          a++;  
5      return a;  
6  }
```

Loop Widening

```
1 ✓ int foo() {  
2     int a = 0;  
3     while (a <= 1000)  
4     {  
5         a++;  
6     }  
7 }
```

$$F(x) \stackrel{\text{def}}{=} x \sqcup (x \sqcap (-\infty, 1000] + [1, 1))$$

$$G(x) \stackrel{\text{def}}{=} x \sqcap [1001, \infty)$$

Loop Widening

```
1 ✓ int foo() {  
2     int a = 0;  
3     while (a <= 1000)  
4         a++;  
5     return a;  
6 }
```

$$F(x) \stackrel{\text{def}}{=} x \sqcup (x \sqcap (-\infty, 1000] + [1, 1])$$

$$x_0 = [0, 0]$$

$$x_1 = F(x_0) = [0, 1]$$

...

$$x_{1001} = F(x_{1000}) = [0, 1001]$$

$$x_{1002} = F(x_{1001}) = [0, 1001]$$

Fixpoint!

$$G(x) \stackrel{\text{def}}{=} x \sqcap [1001, \infty)$$

$$G(x_{1002}) = [1001, 1001]$$

Loop Widening – Extrapolation (∇)

```
1 ✓ int foo() {  
2     int a = 0;  
3     while (a <= 1000)  
4         a++;  
5     return a;  
6 }
```

$$[L_1, U_1] \nabla [L_2, U_2] = [L_2 > L_1 ? -\infty : L_1, U_2 > U_1 ? \infty : U_1]$$

Where:

- $[L_1, U_1]$ is the lower and upper interval at x_n
- $[L_2, U_2]$ is the lower and upper interval at x_{n+1}
- The widening is represented through $x_n \nabla x_{n+1}$

Loop Widening – Extrapolation (∇)

```
1 ✓ int foo() {  
2     int a = 0;  
3     while (a <= 1000)  
4     {  
5         a++;  
6     }  
7 }
```

$$F(x) \stackrel{\text{def}}{=} x \sqcup (x \sqcap (-\infty, 1000] + [1, 1])$$

$$x_0 = [0, 0]$$

$$x_1 = x_0 \nabla F(x_0) = [0, \infty]$$

$$x_2 = x_1 \nabla F(x_1) = [0, \infty] \text{ Fixpoint!}$$

$$G(x) \stackrel{\text{def}}{=} x \sqcap [1001, \infty)$$

$$G(x_2) = [1001, \infty]$$

Loop Widening – Narrowing (Δ)

```
1 ✓ int foo() {  
2     int a = 0;  
3     while (a <= 1000)  
4         a++;  
5     return a;  
6 }
```

$$[L_1, U_1] \Delta [L_2, U_2] = [L_1 = -\infty ? L_2 : L_1, U_1 = \infty ? U_2 : U_1]$$

Where:

- $[L_1, U_1]$ is the lower and upper interval at x_n
- $[L_2, U_2]$ is the lower and upper interval at x_{n+1}
- The narrowing is computed through $x_n \Delta x_{n+1}$

Loop Widening – Narrowing (Δ)

```
1 ✓ int foo() {  
2     int a = 0;  
3     while (a <= 1000)  
4     {  
5         a++;  
6     }  
7 }
```

$$F(x) \stackrel{\text{def}}{=} x \sqcup (x \sqcap (-\infty, 1000] + [1, 1])$$

$$x_0 = [0, \infty]$$

$$x_1 = x_0 \Delta F(x_0) = [0, 1001]$$

$$x_2 = x_1 \Delta F(x_1) = [0, 1001] \text{ Fixpoint!}$$

$$G(x) \stackrel{\text{def}}{=} x \sqcap [1001, \infty)$$

$$G(x_2) = [1001, 1001]$$

Summary



ESBMC interval analysis lacks precision and does not apply some classic optimizations on it.



We can get more precise intervals by adding more extensions.

Preliminary Experiments



Goals

- Test the current implementation (soundness, correctness);
- Verify whether the extra preprocessing cost (time and memory) leads to more verification tasks;

Benchmarks

- For the evaluation we will rely on the ReachSafety category of SV-COMP 23 with benchexec
- Three configurations are going to be used:
 - A: Contractors: 150s, 6GB of RAM;
 - B: Contractors + Modular + Arithmetic + Extrapolation + Narrowing: 30s, 6GB of RAM;
 - C: Contractors + Modular + Arithmetic : 30s, 6GB of RAM;

Results – A (November 2022)

Status	Baseline	Contractor
Correct True (+2)	2819	2857
Correct False (+1)	1295	1614
Incorrect True (-32)	20	17
Incorrect False (-16)	10	11
Score	6133	6608

Results – A (November 2022)

Status	Baseline	Contractor
Correct True (+2)	2819	2857 (+)
Correct False (+1)	1295	1614 (+)
Incorrect True (-32)		
Incorrect False (-16)	10	11 (+)
Score	6133	6608

ESBMC was able to solve more benchmarks than before!

Results – A (November 2022)

Status	Baseline	Contractor
Correct True (+2)	2819	2857 (+)
Correct False (+1)	28	28
Incorrect True (-32)	20	17 (-)
Incorrect False (-16)	10	11 (+)
Score	6133	6608

Although the number of Incorrect benchmarks reduced drastically, it caused new unique benchmarks to fail.

Results – B/C

Status	Baseline	IA+Modular	IA+Modular+Widening
Correct True (+2)	2948	2599	2461
Unique True	-	36	5
Correct False (+1)	1831	1571	1459
Unique False	-	1	2
Incorrect True (-32)	49	34	32
Incorrect False (-16)	15	11	9
Score	5919	5505	5213

Results – B/C

Status	Baseline	IA+Modular	IA+Modular+Widening
Correct True (+2)	2948	2599	2461
Unique True	-	36	5
Correct False (+1)	1831	1571	1459
Unique False	-	1	2
Incorrect True (-32)	49	34	32
Incorrect False (-16)	15	11	9
Score	5919	5505	5213

Most of these the new unique were in ECA and Hardware safety programs.

Results – B/C

Status	Baseline	IA+Modular	IA+Modular+Identifying
Correct True (+2)	2948	2599	2461
Unique True	-	36	5
Correct False (+1)	1831	1571	1459
Unique False	-	1	2
Incorrect True (-32)	49	34	32
Incorrect False (-16)	15	11	9
Score	5919	5505	5213

Many timeouts trying to find a fixpoint. Goto-unwind might be causing issues!

Discussion

- The contractor is a sound way to contract abstract intervals.
- Extrapolation and Narrowing might take too long to reach a fixpoint. This can be even worse when combined with the static loop unrolling.
- Interval Arithmetic might give better results in ECA and Hardware categories if we add support for bitwise operations.
- The current experiments are too short in time.

Summary

- The preliminary experiments were planned to test out the soundness of the implementation so far.
- The tests have shown that overall:
 - Contraction algorithm can be used to merge symbolic states;
 - The extensions are sound, as it didn't add incorrect results;
 - The precision improvement didn't affect the scores by much (30s).
- Next experiments will aim to:
 - Verify longer runs (i.e., 90s, 150s, 300s)
 - Try out different configurations (unwind, extension combinations)
 - Implement more arithmetic operators

Next Steps



Heuristics for Interval Analysis Strategy

- Although the experiments have shown a decrease in the overall score, we still verified unique tasks;
- We could aim to find heuristics for the Interval Analysis to select precision through modular variables, interval arithmetic, contraction level, extrapolation, and narrowing;
- The heuristics could also consider other strategy parameters: max k, incremental, *k-induction*, etc.

Concurrency Support

- We could use the IA to also track intervals for a variable used in multi-threaded context. This would enable:
 - Use of *k-induction* proof for concurrency.
 - Optimization and removal of impossible interleavings.

Combination with Points-To

- Combining this analysis with a points-to analysis so that we can also track intervals through pointers.

This would:

- Improve the method soundness as pointer dereference is not updated.
- Extend the method to also work with memory safety properties.

```
1 int main()
2 {
3     int a = 2, b = 5;
4     int *ptr = * ? &a : &b;
5     if(*ptr < 4) {
6         // Definitely pointing to "a"
7     }
8     else {
9         // Definitely pointing to "b"
10    }
11
12    *ptr = 10;
13    // a: [2,10], b: [5,10] ???
14 }
```

Conclusions

- I have presented my on-going research about applying Interval Analysis over BMC;
- The preliminary results shows that:
 - Contraction algorithms can improve the analysis time
 - The extra preprocessing time for the extensions might not help the SMT solvers as much

Thanks for watching!

