UNIVERSITY OF Southampton

School of Electronics and Computer Science

Encoding floating-point numbers using the SMT theory in ESBMC

Mikhail Ramalho, Lucas Cordeiro, Denis Nicole

mikhail.ramalho@gmail.com

UFAM

UNIVERSITY OF OXFORD

Agenda

- Motivation
- Model Checking vs Testing/Simulation
- ESBMC
- Floating-point SMT encoding
- Illustrative Example
- Experimental Evaluation
- Conclusions and Future Works

Why do we need to verify a program?

USS Yorktown

- Battleship built in 1946 and automated in 1996 (27 dual-core 200MHz processors and Windows NT).

Why do we need to verify a program?

USS Yorktown

- Battleship built in 1946 and automated in 1996 (27 dual-core 200MHz processors and Windows NT).
- **Failure due to a division by zero:** It had to be towed back to its naval base.

Why do we need to verify a program?

```
int main()
{
   float x;
   float y = x;
   assert(x==y);
   return 0;
}
```

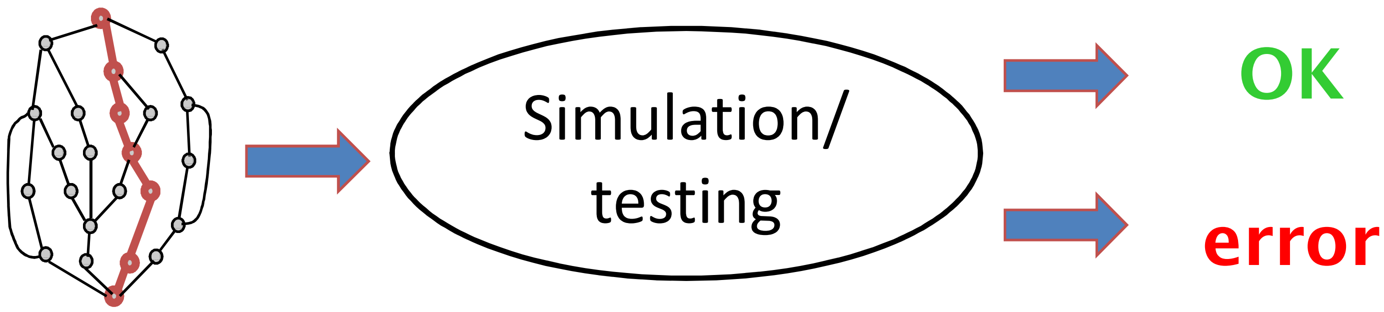
- Is this simple C program wrong?

Why do we need to verify a program?

```
int main()
{
  float x;
  float y = x;
  assert(x==y);
  return 0;
}
```

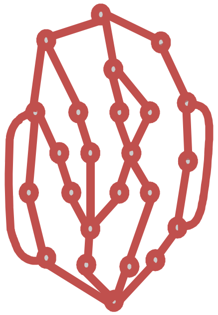
- Is this simple C program wrong?
- Yes! (x = NaN)

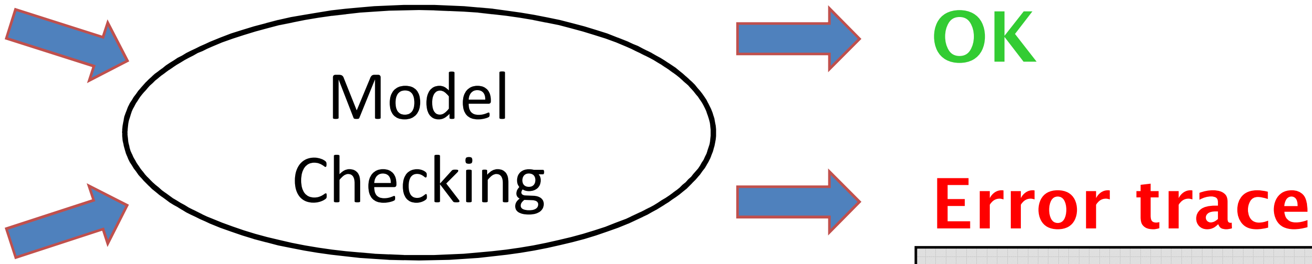
Model Checking vs Testing/Simulation



- Checks some of the system executions.
- May miss errors.
- Cheaper compared to model checking.

Model Checking vs Testing/Simulation



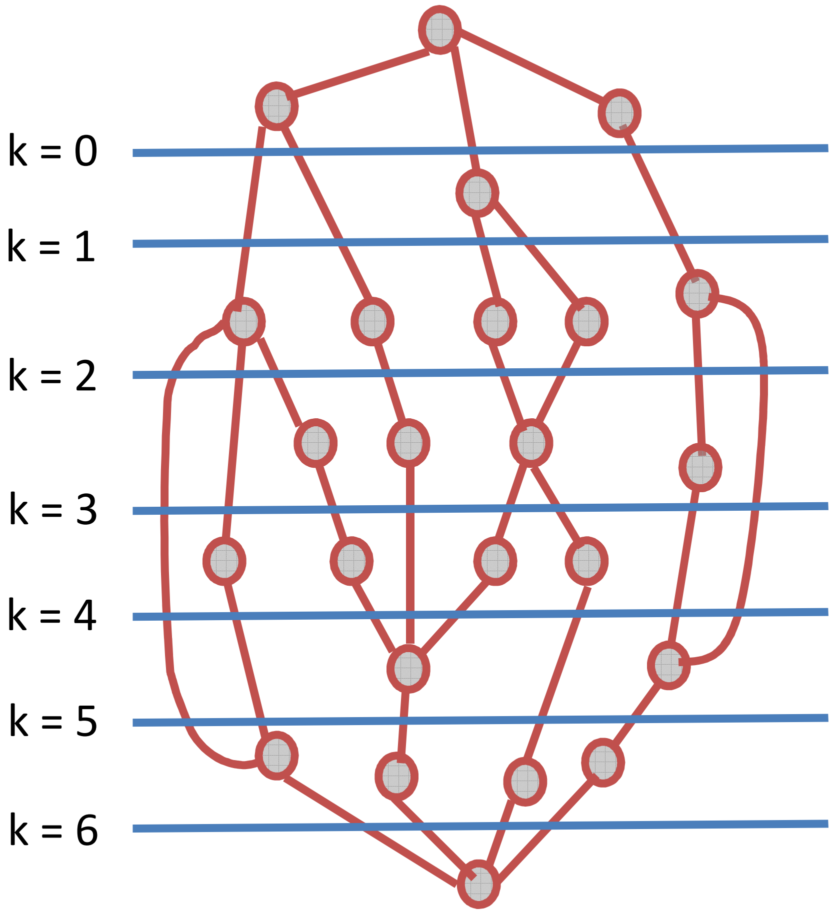



```
Line 5: …
Line 12: …
…
Line 41:…
```

Specification (e.g, LTL)

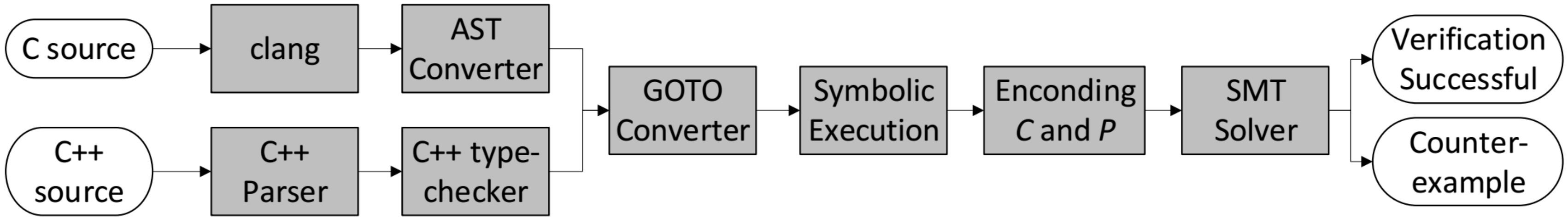
- Exhaustively explores all executions.
  - Can be bounded to limit number of iterations, context-switch, etc.
- Report errors as traces.
- Can be extremely resource-hungry.

Bounded Model checking



- Bounded model checker "slice" the state space
- It's aimed to find bugs and can only prove correctness if all states are reachable

ESBMC: BMC for C and C++



- Exploits SMT solvers and their background theories:
  - optimized encodings for pointers, bit operations, unions and arithmetic over- and underflow
  - Support for Boolector, Z3, MathSAT, CVC4 and Yices
- Supports verifying multi-threaded software that uses pthreads threading library

ESBMC: Verification Support

- built-in properties:
  - arithmetic under- and overflow, pointer safety, array bounds, division by zero, alignment check, memory leaks, atomicity and order violations, deadlock, data race

- user-specified assertions:
  - *(__ESBMC_assume, __ESBMC_assert)*

- built-in scheduling functions:
  - *(__ESBMC_atomic_begin, __ESBMC_atomic_end, __ESBMC_yield)*

Floating-point SMT Encoding

- The SMT floating-point theory is an addition to the SMT standard, proposed in 2010 and formalises:
  - floating-point arithmetic,
  - positive and negative infinities and zeroes,
  - NaNs,
  - comparison operators,
  - five rounding modes: round nearest with ties choosing the even value, round nearest with ties choosing away from zero, round towards zero, round towards positive infinity and round towards negative infinity

Floating-point SMT Encoding

- Missing from the standard:
  - Floating-point exceptions
  - Signaling NaNs
- Two solvers currently support the standard:
  - Z3: implements all operators
  - MathSAT: implements all but two operators (fp.rem and fp.fma)
- Both solvers offer non-standard functions:
  - fp_as_ieeebv: converts floating-point to bitvectors
  - fp_from_ieeebv: converts bitvectors to floating-point

How to encode programs?

- Most operations performed at program-level to encode floating-point numbers have a one-to-one conversion to SMT

- Special cases being casts to boolean types and the fp.eq operator.

Cast to/from booleans

```
int main()
{
  _Bool c;

  double b = 0.0f;
  b = c;
  assert(b != 0.0f);

  c = b;
  assert(c != 0);
}
```

- Usually, cast operations are encoded using extend/extract operations

Cast to/from booleans

```
int main()
{
  _Bool c;

  double b = 0.0f;
  b = c;
  assert(b != 0.0f);

  c = b;
  assert(c != 0);
}
```

- Usually, cast operations are encoded using extend/extract operations
- Extending floating-point numbers is non-trivial because of the format

Cast to/from booleans

- Simpler solutions:
  - Casting booleans to floating-point numbers can be done using an ite operator

```
(assert (= (ite |main::c|
            (fp #b0 #b01111111111 #x0000000000000)
            (fp #b0 #b00000000000 #x0000000000000))
    |main::b|))
```

Cast to/from booleans

- Simpler solutions:
  - Casting booleans to floating-point numbers can be done using an ite operator

If true, assign 1f to b

```
(assert (= (ite |main::c|
        (fp #b0 #b01111111111 #x0000000000000)
        (fp #b0 #b00000000000 #x0000000000000))
   |main::b|))
```

Cast to/from booleans

- Simpler solutions:
  - Casting booleans to floating-point numbers can be done using an ite operator

```
(assert (= (ite |main::c|
        (fp #b0 #b01111111111 #x0000000000000)
        (fp #b0 #b00000000000 #x0000000000000))
    |main::b|))
```

Otherwise, assign 0f to b

Cast to/from booleans

- Simpler solutions:
  - Casting floating-point numbers to booleans can be done using an equality and one not:

```
(assert (= (not (fp.eq |main::b|
         (fp #b0 #b00000000000 #x0000000000000)))
    |main::c|))
```

The fp.eq operator

```
:note
"(fp.eq x y) evaluates to true if x evaluates to -zero and y
to +zero, or vice versa. fp.eq and all the other comparison op-
erators evaluate to false if one of their arguments is NaN."
```

The fp.eq operator

```
:note
"(fp.eq x y) evaluates to true if x evaluates to -zero and y
to +zero, or vice versa. fp.eq and all the other comparison op-
erators evaluate to false if one of their arguments is NaN."
```

- In SMT, there is no difference between assignments and comparisons, except when it comes to floating-point numbers
- For floating-point numbers, the comparison operator is replaced by fp.eq

Unused operators

- fp.max: returns the larger of two floating-point numbers; equivalent to the fmax, fmaxf, fmaxl functions

Unused operators

```
double fmax(double x, double y) {
  // If both argument are NaN, NaN is returned
  if(isnan(x) && isnan(y)) return NAN;

  // If one arg is NaN, the other is returned
  if(isnan(x) || isnan(y)) {
    if(isnan(x))
      return y;
    return x;
  }

  return (x > y ? x : y);
}
```

Unused operators

```
double fmax(double x, double y) {
  // If both argument are NaN, NaN is returned
  if(isnan(x) && isnan(y)) return NAN;

  // If one arg is NaN, the other is returned
  if(isnan(x) || isnan(y)) {
    if(isnan(x))
      return y;
    return x;
  }

  return (x > y ? x : y);
}
```

Unused operators

```
double fmax(double x, double y) {
  // If both argument are NaN, NaN is returned
  if(isnan(x) && isnan(y)) return NAN;

  // If one arg is NaN, the other is returned
  if(isnan(x) || isnan(y)) {
    if(isnan(x))
      return y;
    return x;
  }

  return (x > y ? x : y);
}
```

Unused operators

```
double fmax(double x, double y) {
  // If both argument are NaN, NaN is returned
  if(isnan(x) && isnan(y)) return NAN;

  // If one arg is NaN, the other is returned
  if(isnan(x) || isnan(y)) {
    if(isnan(x))
      return y;
    return x;
  }

  return (x > y ? x : y);
}
```

Unused operators

- fp.min: returns the smaller of two floating-point numbers; equivalent to the fmin, fminf, fminl functions

Unused operators

```
double fmin(double x, double y) {
  // If both argument are NaN, NaN is returned
  if(isnan(x) && isnan(y)) return NAN;

  // If one arg is NaN, the other is returned
  if(isnan(x) || isnan(y)) {
    if(isnan(x))
      return y;
    return x;
  }

  return (x < y ? x : y);
}
```

Unused operators

```
double fmin(double x, double y) {
  // If both argument are NaN, NaN is returned
  if(isnan(x) && isnan(y)) return NAN;

  // If one arg is NaN, the other is returned
  if(isnan(x) || isnan(y)) {
    if(isnan(x))
      return y;
    return x;
  }

  return (x < y ? x : y);
}
```

Unused operators

- fp.rem: returns the floating-point remainder of the division operation x/y; equivalent to the fmod, fmodf, fmodl functions

Unused operators

```
double fmod(double x, double y) {
  // If either argument is NaN, NaN is returned
  if(isnan(x) || isnan(y)) return NAN;

  // If x is +inf/-inf and y is not NaN, NaN is returned
  if(isinf(x)) return NAN;

  // If y is +0.0/-0.0 and x is not NaN, NaN is returned
  if(y == 0.0) return NAN;

  // If x is +0.0/-0.0 and y is not zero, returns +0.0/-0.0
  if((x == 0.0) && (y != 0.0)) {
    if(signbit(x))
      return -0.0;
    return +0.0;
  }

  // If y is +inf/-inf and x is finite, x is returned.
  if(isinf(y) && isfinite(x)) return x;

  return x - (y * (int)(x/y));
}
```

Unused operators

```
double fmod(double x, double y) {
  // If either argument is NaN, NaN is returned
  if(isnan(x) || isnan(y)) return NAN;

  // If x is +inf/-inf and y is not NaN, NaN is returned
  if(isinf(x)) return NAN;

  // If y is +0.0/-0.0 and x is not NaN, NaN is returned
  if(y == 0.0) return NAN;

  // If x is +0.0/-0.0 and y is not zero, returns +0.0/-0.0
  if((x == 0.0) && (y != 0.0)) {
    if(signbit(x))
      return -0.0;
    return +0.0;
  }

  // If y is +inf/-inf and x is finite, x is returned.
  if(isinf(y) && isfinite(x)) return x;

  return x - (y * (int)(x/y));
}
```

Unused operators

```
double fmod(double x, double y) {
  // If either argument is NaN, NaN is returned
  if(isnan(x) || isnan(y)) return NAN;

  // If x is +inf/-inf and y is not NaN, NaN is returned
  if(isinf(x)) return NAN;

  // If y is +0.0/-0.0 and x is not NaN, NaN is returned
  if(y == 0.0) return NAN;

  // If x is +0.0/-0.0 and y is not zero, returns +0.0/-0.0
  if((x == 0.0) && (y != 0.0)) {
    if(signbit(x))
      return -0.0;
    return +0.0;
  }

  // If y is +inf/-inf and x is finite, x is returned.
  if(isinf(y) && isfinite(x)) return x;

  return x - (y * (int)(x/y));
}
```

Unused operators

```
double fmod(double x, double y) {
  // If either argument is NaN, NaN is returned
  if(isnan(x) || isnan(y)) return NAN;

  // If x is +inf/-inf and y is not NaN, NaN is returned
  if(isinf(x)) return NAN;

  // If y is +0.0/-0.0 and x is not NaN, NaN is returned
  if(y == 0.0) return NAN;

  // If x is +0.0/-0.0 and y is not zero, returns +0.0/-0.0
  if((x == 0.0) && (y != 0.0)) {
    if(signbit(x))
      return -0.0;
    return +0.0;
  }

  // If y is +inf/-inf and x is finite, x is returned.
  if(isinf(y) && isfinite(x)) return x;

  return x - (y * (int)(x/y));
}
```

Unused operators

```
double fmod(double x, double y) {
  // If either argument is NaN, NaN is returned
  if(isnan(x) || isnan(y)) return NAN;

  // If x is +inf/-inf and y is not NaN, NaN is returned
  if(isinf(x)) return NAN;

  // If y is +0.0/-0.0 and x is not NaN, NaN is returned
  if(y == 0.0) return NAN;

  // If x is +0.0/-0.0 and y is not zero, returns +0.0/-0.0
  if((x == 0.0) && (y != 0.0)) {
    if(signbit(x))
      return -0.0;
    return +0.0;
  }

  // If y is +inf/-inf and x is finite, x is returned.
  if(isinf(y) && isfinite(x)) return x;

  return x - (y * (int)(x/y));
}
```

Unused operators

```
double fmod(double x, double y) {
  // If either argument is NaN, NaN is returned
  if(isnan(x) || isnan(y)) return NAN;

  // If x is +inf/-inf and y is not NaN, NaN is returned
  if(isinf(x)) return NAN;

  // If y is +0.0/-0.0 and x is not NaN, NaN is returned
  if(y == 0.0) return NAN;

  // If x is +0.0/-0.0 and y is not zero, returns +0.0/-0.0
  if((x == 0.0) && (y != 0.0)) {
    if(signbit(x))
      return -0.0;
    return +0.0;
  }

  // If y is +inf/-inf and x is finite, x is returned.
  if(isinf(y) && isfinite(x)) return x;

  return x - (y * (int)(x/y));
}
```

Unused operators

```
double fmod(double x, double y) {
  // If either argument is NaN, NaN is returned
  if(isnan(x) || isnan(y)) return NAN;

  // If x is +inf/-inf and y is not NaN, NaN is returned
  if(isinf(x)) return NAN;

  // If y is +0.0/-0.0 and x is not NaN, NaN is returned
  if(y == 0.0) return NAN;

  // If x is +0.0/-0.0 and y is not zero, returns +0.0/-0.0
  if((x == 0.0) && (y != 0.0)) {
    if(signbit(x))
      return -0.0;
    return +0.0;
  }

  // If y is +inf/-inf and x is finite, x is returned.
  if(isinf(y) && isfinite(x)) return x;

  return x - (y * (int)(x/y));
}
```

Unused operators

- fp.isSubnormal: we could not find any user case for it when modelling C11 standard functions.

Illustrative Example

```
int main()
{
   float x;
   float y = x;
   assert(x==y);
   return 0;
}
```

Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeteministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                    (=> |execution_statet::\\guard_exec|
                        (fp.eq |main::x| |main::y|))))))
  (or a!1)))
```

Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeteministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                    (=> |execution_statet::\\guard_exec|
                        (fp.eq |main::x| |main::y|))))))
  (or a!1)))
```

Variable declarations

Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeteministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                    (=> |execution_statet::\\guard_exec|
                        (fp.eq |main::x| |main::y|))))))
  (or a!1)))
```

Nondeterministic symbol declaration (optional)

Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeteministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                    (=> |execution_statet::\\guard_exec|
                        (fp.eq |main::x| |main::y|))))))
  (or a!1)))
```

Guard used to check satisfiability

Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeteministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                    (=> |execution_statet::\\guard_exec|
                        (fp.eq |main::x| |main::y|))))))
  (or a!1)))
```

Assignment of nondeterministic value to x

Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeteministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                    (=> |execution_statet::\\guard_exec|
                        (fp.eq |main::x| |main::y|))))))
  (or a!1)))
```

Assignment x to y

Illustrative Example

```
; declaration of x and y
(declare-fun |main::x| () (_ FloatingPoint 8 24))
(declare-fun |main::y| () (_ FloatingPoint 8 24))

; symbol created to represent a nondeteministic number
(declare-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24))

; Global guard, used for checking properties
(declare-fun |execution_statet::\\guard_exec| () Bool)

; assign the nondeterministic symbol to x
(assert (= |nondet_symex::nondet0| |main::x|))

; assign x to y
(assert (= |main::x| |main::y|))

; assert x == y
(assert (let ((a!1 (not (=> true
                    (=> |execution_statet::\\guard_exec|
                        (fp.eq |main::x| |main::y|))))))
  (or a!1)))
```

Check if the comparison satisfies the guard

Illustrative Example

- Z3 produces:

```
sat
(model
  (define-fun |main::x| () (_ FloatingPoint 8 24)
    (_ NaN 8 24))
  (define-fun |main::y| () (_ FloatingPoint 8 24)
    (_ NaN 8 24))
  (define-fun |nondet_symex::nondet0| () (_ FloatingPoint 8 24)
    (_ NaN 8 24))
  (define-fun |execution_statet::\\\\guard_exec| () Bool
    true)
)
```

Illustrative Example

- MathSAT produces:

```
sat
( (|main::x| (_ NaN 8 24))
  (|main::y| (_ NaN 8 24))
  (|nondet_symex::nondet0| (_ NaN 8 24))
  (|execution_statet::\\guard_exec| true) )
```

Illustrative Example

```
Counterexample:

State 1 file main3.c line 3 function main thread 0
main
----------------------------------------------------
  main3::main::1::x=-NaN (11111111100000000000000000000001)

State 2 file main3.c line 4 function main thread 0
main
----------------------------------------------------
  main3::main::2::y=-NaN (11111111100000000000000000000001)

State 3 file main3.c line 5 function main thread 0
main
----------------------------------------------------
Violated property:
  file main3.c line 5 function main
  assertion
  (_Bool)(x == y)

VERIFICATION FAILED
```

Experimental Evaluation

- 172 benchmarks from SV-COMP'17
- Timeout: 900s
- Memory limit: 15GB
- MathSAT v5.3.14
- Z3 v4.5.0

Experimental Evaluation

	ESBMC (MathSAT v5.3.14)	ESBMC (Z3 v4.5.0)
Correct true	**139**	111
Correct false	**30**	16
Timeout	**3**	45
Total time (s)	**9977.40**	44992.76

Experimental Evaluation

	ESBMC (MathSAT v5.3.14)	ESBMC (Z3 v4.5.0)
Correct true	**139**	111
Correct false	**30**	16
Timeout	**3**	45
Total time (s)	**9977.40**	44992.76

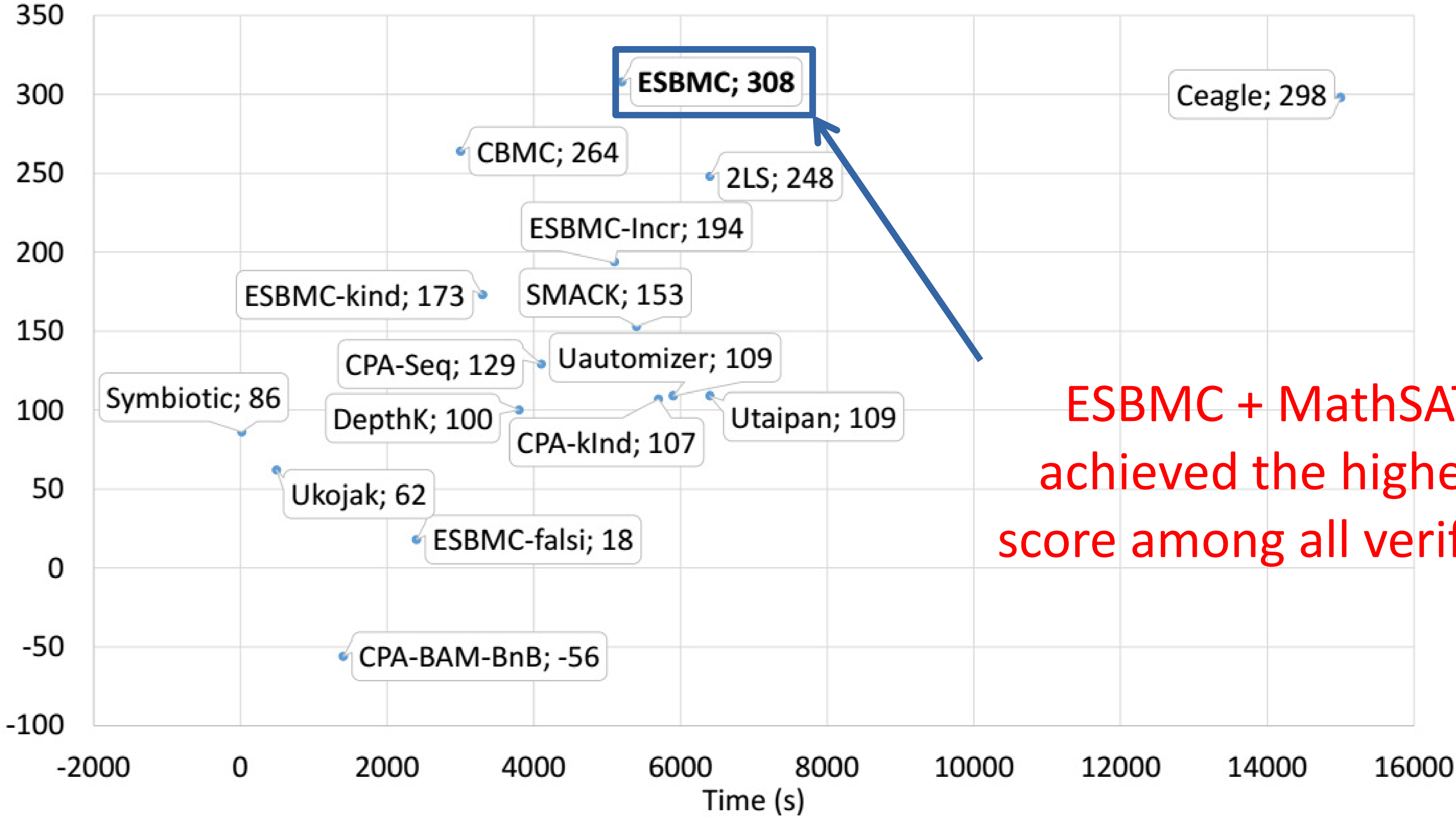
- 76 out of the 172 (44%) benchmarks are deterministic (no solver is invoked)

Experimental Evaluation

	ESBMC (MathSAT v5.3.14)	ESBMC (Z3 v4.5.0)
Correct true	**139**	111
Correct false	**30**	16
Timeout	**3**	45
Total time (s)	**9977.40**	44992.76

- 76 out of the 172 (44%) benchmarks are deterministic (no solver is invoked)
- MathSAT is 4.5x faster than Z3 when verifying the same set of benchmarks

Comparison to other Software Verifiers



ESBMC + MathSAT achieved the highest score among all verifiers

Conclusions

- We presented an approach to encode C programs, using the SMT floating-point theory
- We implemented our approach in ESBMC, using two different solvers, Z3 and MathSAT, and MathSAT proved to be much faster than Z3
- We evaluated our approach against other verifiers and ESBMC with MathSAT proved to be the state-of-art

Future Work

- Create a floating-point API to encode operations using bitvectors
  - It will enable verifying programs using other solvers (Boolector, CVC4 and Yices)
  - Public implementations available (CPROVER and Z3)

Thank you!

www.esbmc.org

https://github.com/esbmc/esbmc

mikhail.ramalho@gmail.com