SMT-based Bounded Model Checking for Multi-threaded Software in Embedded Systems

Lucas Cordeiro

lucascordeiro@ufam.edu.br

Embedded systems are ubiquitous but their verification becomes more difficult.

- functionality demanded increased significantly
  - peer reviewing and testing
- multi-core processors with scalable shared memory
  - but software model checkers focus on single-threaded or multi-threaded with message passing

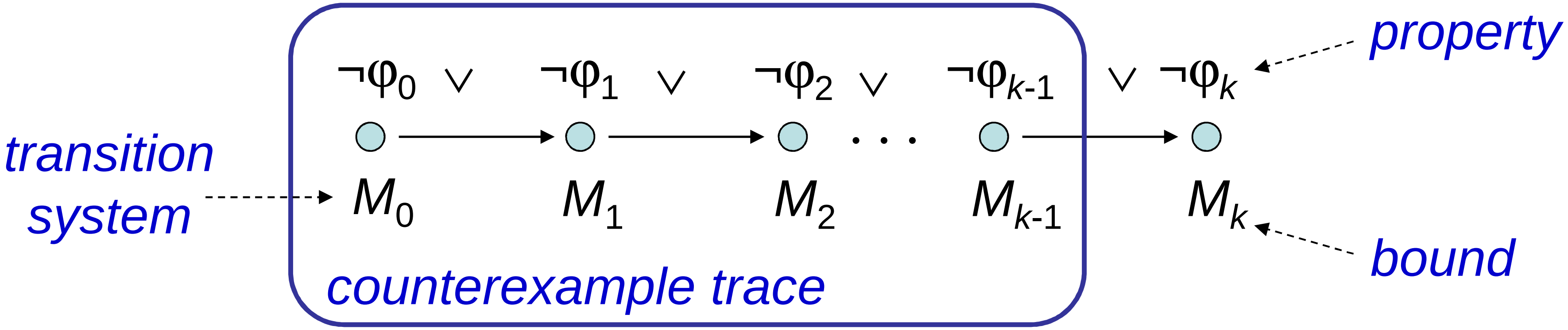
```
void *threadA(void *arg) {
  lock(&mutex);
  x++;
  if (x == 1) lock(&lock);
  unlock(&mutex); (CS1)
  lock(&mutex);   (CS3)
  x--;
  if (x == 0) unlock(&lock);
  unlock(&mutex);
}
```

Deadlock

```
void *threadB(void *arg) {
  lock(&mutex);
  y++;
  if (y == 1) lock(&lock); (CS2)
  unlock(&mutex);
  lock(&mutex);
  y--;
  if (y == 0) unlock(&lock);
  unlock(&mutex);
}
```

Bounded Model Checking (BMC)

Basic Idea: check negation of given property up to given depth



- transition system *M* unrolled *k* times
  - for programs: unroll loops, unfold arrays, …
- translated into verification condition ψ such that

ψ satisfiable iff φ has counterexample of max. depth *k*

- has been applied successfully to verify (sequential) software

BMC of Multi-threaded Software

- concurrency bugs are tricky to **reproduce/debug** because they usually occur under specific thread interleavings
  - most common errors: *67% related to atomicity and order violations, 30% related to deadlock* [Lu et al.'08]
- problem: the number of interleavings grows exponentially with the number of threads (n) and program statements (s)
  - *number of executions: $O(n^s)$*
  - *context switches among threads increase the number of possible executions*
- two important observations help us:
  - concurrency bugs are shallow [Qadeer&Rehof'05]
  - SAT/SMT solvers produce unsatisfiable cores that allow us to remove possible undesired models of the system

Objective of this work

Exploit SMT to extend BMC of embedded software

- exploit SMT solvers to:
  - encode full ANSI-C into the different background theories
  - prune the *property and data dependent* search space
  - remove interleavings that are not relevant by analyzing the proof of unsatisfiability
- propose three approaches to SMT-based BMC:
  - *lazy exploration* of the interleavings
  - *schedule guards* to encode all interleavings
  - *underapproximation and widening (UW) [Grumberg et al.'05]*
- evaluate our approaches implemented in ESBMC over embedded software applications

Agenda

- SMT-based BMC for Embedded ANSI-C Software

- Verifying Multi-threaded Software

- Implementation of ESBMC

- Integrating ESBMC into Software Engineering Practice

- Conclusions and Future Work

Satisfiability Modulo Theories (1)

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories** (⇒ building-in operators).

Theory	Example
Equality	$x_1=x_2 \wedge \neg (x_2=x_3) \Rightarrow \neg(x_1=x_3)$
Bit-vectors	$(b >> i) \& 1 = 1$
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \vee (y_2 - 3y_3 \leq 3)$
Arrays	$(j = k \wedge a[k]=2) \Rightarrow a[j]=2$
Combined theories	$(j \leq k \wedge a[j]=2) \Rightarrow a[i] < 3$

Satisfiability Modulo Theories (2)

- Given
  - a decidable Σ-theory T
  - a quantifier-free formula φ

  φ is *T*-satisfiable iff $T \cup \{\varphi\}$ is satisfiable, i.e., there exists a *structure* that *satisfies* both *formula* and *sentences* of T
- Given
  - a set $\Gamma \cup \{\varphi\}$ of first-order formulae over T

  φ is a *T*-consequence of Γ ($\Gamma \models_T \varphi$) iff *every model of* $T \cup \Gamma$ is also a *model of* φ
- Checking $\Gamma \models_T \varphi$ can be reduced in the usual way to checking the T-satisfiability of $\Gamma \cup \{\neg\varphi\}$

Satisfiability Modulo Theories (3)

- let **a** be an array, **b**, **c** and **d** be signed bit-vectors of width 16, 32 and 32 respectively, and let **g** be an unary function

$$g\left(select\left(store\left(a,c,12\right)\right), SignExt\left(b,16\right)+3\right)$$
$$\neq g\left(SignExt\left(b,16\right)-c+4\right)\wedge SignExt\left(b,16\right)=c-3\wedge c+1=d-4$$

⇩ **b'** extends **b** to the signed equivalent bit-vector of size 32

$$step\,1: g\left(select\left(store\left(a,c,12\right),b'+3\right)\right)\neq g\left(b'-c+4\right)\wedge b'=c-3\wedge c+1=d-4$$

⇩ replace b' by c−3 in the inequality

$$step\,2: g\left(select\left(store\left(a,c,12\right),c-3+3\right)\right)\neq g\left(c-3-c+4\right)\wedge c-3=c-3\wedge c+1=d-4$$

⇩ using facts about bit-vector arithmetic

$$step\,3: g\left(select\left(store\left(a,c,12\right),c\right)\right)\neq g\left(1\right)\wedge c-3=c-3\wedge c+1=d-4$$

Satisfiability Modulo Theories (4)

$$step\ 3: g(select(store(a,c,12),c)) \neq g(1) \wedge c-3=c-3 \wedge c+1=d-4$$

⇩ applying the theory of arrays

$$step\ 4: g(12) \neq g(1) \wedge c-3=c-3 \wedge c+1=d-4$$

⇩ The function g implies that for all x and y,
if x = y, then g (x) = g (y) (*congruence rule*).

$$step\ 5: SAT\ (\mathrm{c}=5,\ \mathrm{d}=10)$$

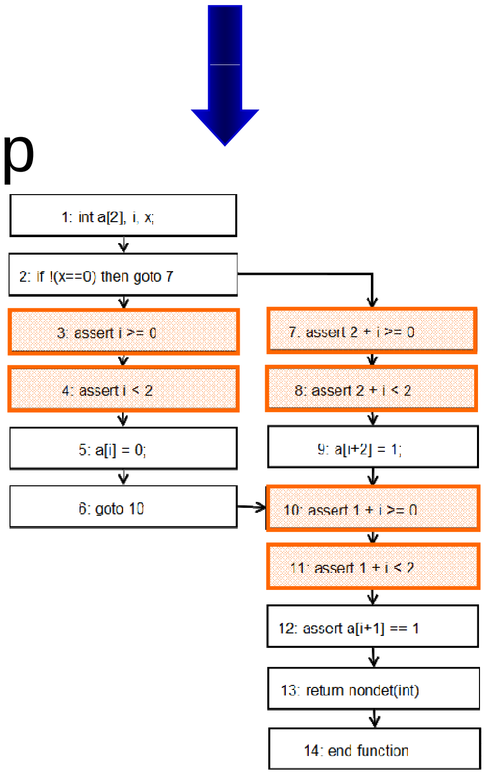
- SMT solvers also apply:
  - standard algebraic reduction rules $\boxed{r \wedge false \mapsto false}$
  - contextual simplification $\boxed{a=7 \wedge p(a) \mapsto a=7 \wedge p(7)}$

Software BMC using ESBMC

- program modelled as state transition system
  - *state*: program counter and program variables
  - derived from control-flow graph
  - checked safety properties give extra nodes
- program unfolded up to given bounds
  - loop iterations
  - context switches
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions

  } crucial

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```



Software BMC using ESBMC

- program modelled as state transition system
  - *state*: program counter and program variables
  - derived from control-flow graph
  - checked safety properties give extra nodes
- program unfolded up to given bounds
  - loop iterations
  - context switches
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions

  } crucial
- front-end converts unrolled and optimized program into SSA

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```

$$
\begin{aligned}
&g_1 = x_1 == 0\\
&a_1 = a_0 \text{ WITH } [i_0{:=}0]\\
&a_2 = a_0\\
&a_3 = a_2 \text{ WITH } [2+i_0{:=}1]\\
&a_4 = g_1 \;?\; a_1 : a_3\\
&t_1 = a_4\,[1+i_0] == 1
\end{aligned}
$$

Software BMC using ESBMC

- program modelled as state transition system
  - *state*: program counter and program variables
  - derived from control-flow graph
  - checked safety properties give extra nodes
- program unfolded up to given bounds
  - loop iterations
  - context switches
- unfolded program optimized to reduce blow-up
  - constant propagation
  - forward substitutions

  } crucial
- front-end converts unrolled and optimized program into SSA
- extraction of *constraints C* and *properties P*
  - specific to selected SMT solver, uses theories
- satisfiability check of *C* ∧ ¬*P*

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```

$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{bmatrix}$$

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{bmatrix}$$

Encoding of Numeric Types

- SMT solvers typically provide different encodings for numbers:
  - abstract domains ($\mathbf{Z}$, $\mathbf{R}$)
  - fixed-width bit vectors (`unsigned int`, …)
    - ▹ “internalized bit-blasting”
- verification results can depend on encodings

$$(a > 0) \wedge (b > 0) \Rightarrow (a + b > 0)$$

valid in abstract domains such as $\mathbf{Z}$ or $\mathbf{R}$

doesn’t hold for bitvectors, due to possible overflows

  - majority of VCs solved faster if numeric types are modelled by abstract domains but possible loss of precision
  - ESBMC supports both types of encoding and also combines them to improve scalability and precision

Encoding Numeric Types as Bitvectors

Bitvector encodings need to handle

- type casts and implicit conversions
  - arithmetic conversions implemented using word-level functions (part of the bitvector theory: Extract, SignExt, …)
    - ▹ different conversions for every pair of types
    - ▹ uses type information provided by front-end
  - conversion to / from bool via if-then-else operator

    $t = ite(v \neq k, true, false)$ *//conversion to bool*

    $v = ite(t, 1, 0)$ *//conversion from bool*
- arithmetic over- / underflow
  - standard requires modulo-arithmetic for unsigned integer

    $unsigned\_overflow \Leftrightarrow (r - (r \bmod 2^w)) < 2^w$
  - define error literals to detect over- / underflow for other types

    $res\_op \Leftrightarrow \neg\, overflow(x, y) \wedge \neg\, underflow(x, y)$
    - ▹ similar to conversions

Floating-Point Numbers

- over-approximate floating-point by fixed-point numbers
  - encode the integral (*i*) and fractional (*f*) parts
- **binary encoding:** get a new bit-vector *b* = *i* @ *f* with the same bitwidth before and after the radix point of *a*

$$i = \begin{cases} \mathit{Extract}(b, n_b + m_a - 1, n_b) & : \quad m_a \le m_b \\ \mathit{SignExt}(b, m_a - m_b) & : \quad \mathit{otherwise} \end{cases}$$

// m = number of bits of i

$$f = \begin{cases} \mathit{Extract}(b, n_b - 1, n_b - n_a) & : \quad n_a \le n_b \\ \mathit{ZeroExt}(b, n_a - n_b) & : \quad \mathit{otherwise} \end{cases}$$

// n = number of bits of f

- **rational encoding:** convert a to *a* rational number

$$a = \begin{cases} \dfrac{\left(i * p + \left(\dfrac{f * p}{2^n}\right)\right)}{p} & : \quad f \neq 0 \\ i \quad : \quad \mathit{otherwise} \end{cases}$$

// i = parte inteita
// f =parte fracionária
// n = número de bits da parte fracionária
// p = number of decimal places

Encoding of Pointers

- arrays and records / tuples typically handled directly by SMT-solver
- pointers modelled as tuples
  - p.o ≙ representation of underlying object
  - p.i ≙ index (if pointer used as array base)

```
int main() {
  int a[2], i, x, *p;
  p=a;
  if (x==0)
    a[i]=0;
  else
    a[i+1]=1;
  assert(*(p+2)==1);
}
```

⇒

$$C := \left(\begin{array}{l} p_1 := \text{store}(p_0,\ 0,\ \&a[0]) \\ \wedge\ p_2 := \text{store}(p_1,\ 1,\ 0) \\ \wedge\ g_1 := (x_1 == 0) \\ \wedge\ a_1 := \text{store}(a_0,\ i_0 \\ \wedge\ a_3 := \text{store}(a_2,\ 1+\ i_0,\ 1) \\ \wedge\ a_4 := \text{ite}(g_1,\ a_1,\ a_3) \\ \wedge\ p_3 := \text{store}(p_2,\ 1,\ \text{select}(p_2\ ,\ 1)+2) \end{array} \right)$$

Store object at position 0

Store index at position 1

Update index

Encoding of Pointers

- arrays and records / tuples typically handled directly by SMT-solver
- pointers modelled as tuples
  - p.o ≙ representation of underlying object
  - p.i ≙ index (if pointer used as array base)

```
int main() {
  int a[2], i, x, *p;
  p=a;
  if (x==0)
    a[i]=0;
  else
    a[i+1]=1;
  assert(*(p+2)==1);
}
```

$$P := \left(\begin{array}{l} i_0 \geq 0 \land i_0 < 2 \\ \land\ 1 + i_0 \geq 0 \land 1 + \quad 2 \\ \land\ \mathrm{select}(p_3, 0) == \&a[0] \\ \land\ \mathrm{select}(\mathrm{select}(p_3, 0), \\ \qquad\qquad \mathrm{select}(p_3, 1)) == 1 \end{array} \right)$$

*negation satisfiable (*a[2] *unconstrained)*
⇒ assert fails

Encoding of Memory Allocation

- model memory just as an array of bytes (*array theories*)
  - read and write operations to the memory array on the logic level
- each dynamic object d_o consists of
  - $m \triangleq$ memory array
  - $s \triangleq$ size in bytes of m
  - $\rho \triangleq$ unique identifier
  - $\upsilon \triangleq$ indicate whether the object is still alive
  - $l \triangleq$ the location in the execution where m is allocated
- to detect invalid reads/writes, we check whether
  - d_o is a dynamic object
  - i is within the bounds of the memory array

$$l_{is\_dynamic\_object} \Leftrightarrow \left(\bigvee_{j=1}^{k} d_o.\rho = j \right) \wedge (0 \leq i < n)$$

Encoding of Memory Allocation

- to check for invalid objects, we
  - set υ to *true* when the function *malloc* is called (d_o is alive)
  - set υ to *false* when the function *free* is called (d_o is not longer alive)

$$l_{valid\_object} \Leftrightarrow (l_{is\_dynamic\_object} \Rightarrow d_o.\upsilon)$$

- to detect forgotten memory, at the end of the (unrolled) program we check
  - whether the d_o has been deallocated by the function *free*

$$l_{deallocated\_object} \Leftrightarrow (l_{is\_dynamic\_object} \Rightarrow \neg\, d_o.\upsilon)$$

Example of Memory Allocation

```
#include <stdlil
void main() {
  char *p = ma
  char *q =  allo
  p=q;
  free(p)
  p = malloc(5);          // ρ = 3
  free(p)
}
```

memory leak: pointer reassignment makes $d_{o1}.\upsilon$ to become an orphan

Example of Memory Allocation

```
#include <stdlib.h>
void main() {
  char *p = malloc(5);  // ρ = 1
  char *q = malloc(5);  // ρ = 2
  p=q;
  free(p)
  p = malloc(5);        // ρ = 3
  free(p)
}
```

$$P := \left(\neg d_{o1}.\upsilon \wedge \neg d_{o2}.\upsilon \; \neg d_{o3}.\upsilon \right)$$

$$C := \left(\begin{array}{l} d_{o1}.\rho=1 \wedge d_{o1}.s=5 \wedge d_{o1}.\upsilon=\mathit{true} \wedge p=d_{o1} \\ \wedge\ d_{o2}.\rho=2 \wedge d_{o2}.s=5 \wedge d_{o2}.\upsilon=\mathit{true} \wedge q=d_{o2} \\ \wedge\ p=d_{o2} \wedge d_{o2}.\upsilon=\mathit{false} \\ \wedge\ d_{o3}.\rho=3 \wedge d_{o3}.s=5 \wedge d_{o3}.\upsilon=\mathit{true} \wedge p=d_{o3} \\ \wedge\ d_{o3}.\upsilon=\mathit{false} \end{array} \right)$$

Example of Memory Allocation

```
#include <stdlib.h>
void main() {
  char *p = malloc(5);  // ρ = 1
  char *q = malloc(5);  // ρ = 2
  p=q;
  free(p)
  p = malloc(5);        // ρ = 3
  free(p)
}
```

$$P := \left(\neg \mathbf{d_{o1}.\upsilon} \wedge \neg d_{o2}.\upsilon \; \neg d_{o3}.\upsilon \right)$$

$$C := \left(\begin{array}{l} d_{o1}.\rho=1 \wedge d_{o1}.s=5 \wedge \mathbf{d_{o1}.\upsilon=\mathit{true}} \wedge p=d_{o1} \\ \wedge\, d_{o2}.\rho=2 \wedge d_{o2}.s=5 \wedge d_{o2}.\upsilon=\mathit{true} \wedge q=d_{o2} \\ \wedge\, p=d_{o2} \wedge d_{o2}.\upsilon=\mathit{false} \\ \wedge\, d_{o3}.\rho=3 \wedge d_{o3}.s=5 \wedge d_{o3}.\upsilon=\mathit{true} \wedge p=d_{o3} \\ \wedge\, d_{o3}.\upsilon=\mathit{false} \end{array} \right)$$

Evaluation

Comparison of SMT solvers

- Goal: compare efficiency of different SMT-solvers
  - CVC3 (2.2)
  - Boolector (1.4)
  - Z3 (2.11)

- Set-up:
  - identical ESBMC front-end, individual back-ends
  - operations not supported by SMT-solvers are axiomatized
  - standard desktop PC, time-out 3600 seconds

Comparison of SMT solvers

Module	#L	#P	CVC3		Boolector		Z3	
			Time	Error	Time	Error	Time	Error
B[illegible]	43	17	[illegible]	[illegible]	**[illegible] (2)**	0	**2** (3)	0
[illegible]	43	17	[illegible]	[illegible]	[illegible] 1)	0	**265** (269)	0
SelectionSort (n=35)	34	17	18 (3)	0	**1 (1)**	0	**1 (1)**	0
(n=140)	34	17	M_b (209)	1	**161** (171)	0	165 (173)	0
InsertionSort (n=35)	86	17	4 (5)	0	**3 (3**[illegible]	[illegible]	[illegible]	0
(n=140)	86	17	**194** (283)	0	350 (219[illegible]	[illegible]	[illegible]	0
Prim	7[illegible]	[illegible]	5 (3)	0	**<1 (<1)**	0	**<1 (<1)**	0
StrCmp	[illegible]	[illegible]	[illegible]	0	195 (257)	0	35 (46)	0
MinMax	19	9	T_b (Mb)	1	42[illegible])	0	**6** (7)	0
lms	258	23	**225** (324)	0	303 [illegible]	[illegible]	[illegible]7)	0
Bitwise	18	1	**3** (6)	0	7 (8)	0	30 (26)	0
adpcm_encode	149	12	6 (26)	0	6 (6)	0	6 (6)	0
adpcm_decode	111	10	3 (27)	0	3 (3)	0	3 (3)	0

lines of code

number of properties checked

size of arrays

SMT-LIB interface

native API

Comparison of SMT solvers

Module	#L	#P	CVC3		Boolector		Z3	
			Time	Error	Time	Error	Time	Error
BubbleSort (n=35)	43	17	17 (5)	0	**2 (2)**	0	**2** (3)	0
(n=140)	43	17	M_b(M_b)	1	282 (311)	0	**265** (269)	0
SelectionSort (n=35)	34	17	18 (3)	0	**1 (1)**	0	**1 (1)**	0
(n=140)	34	17	M_b (209)	1	**161** (171)	0	165 (173)	0
InsertionSort (n=35)	8[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	**3 (3)**	0
(n=140)	8[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	212 (222)	0
Prim	7[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	**<1 (<1)**	0
StrCmp	14	6[illegible]	[illegible]	0	195 (257)	0	35 (46)	0
MinMax	19	[illegible]	T_b (Mb)	1	42 (7)	0	**6** (7)	0
lms	238	23	**225** (324)	0	303 (307)	0	306 (307)	0
Bitwise	**18**	**1**	**3 (6)**	**0**	**7 (8)**	**0**	**30 (26)**	**0**
adpcm_encode	**149**	**12**	**6 (26)**	**0**	**6 (6)**	**0**	**6 (6)**	**0**
adpcm_decode	**111**	**10**	**3 (27)**	**0**	**3 (3)**	**0**	**3 (3)**	**0**

All SMT-solvers can handle the VCs from the embedded applications

Comparison of SMT solvers

Module	#L	#P	CVC3		Boolector		Z3	
			Time	Error				rror
BubbleSort (n=35) (n=140)	43 43	17 17	17 (5) M_b(M_b)	0 1	 28			0 0
SelectionSort (n=35) (n=140)	34 34	17 17	18 (3) M_b (209)	0 1	 **161** (171)	 0	 165 (173)	0 0
InsertionSort (n=35) (n=140)	86 86	17 17	4 (5) **194** (283)	0 0	**3 (3)** 350 (219)	0 0	**3 (3)** 212 (222)	0 0
Prim	79	30	5 (2)	0	**<1 (<1)**	0	**<1 (<1)**	0
StrCmp	14	6	**11** (454)	0	195 (257)	0	35 (46)	0
MinMax	19	9	T_b (Mb)	1	42 (7)	0	**6** (7)	0
lms	258	23	**225** (324)	0	303 (307)	0	306 (307)	0
Bitwise	18	1	3 (6)	0	7 (8)	0	30 (26)	0
adpcm_encode	149	12	6 (26)	0	6 (6)	0	6 (6)	0
adpcm_decode	111	10	3 (27)	0	3 (3)	0	3 (3)	0

CVC3 doesn't scale that well and runs out of memory and time

Comparison of SMT solvers

Boolector and Z3 roughly comparable, with some advantages for Z3

Module					Boolector		Z3	
					Time	Error	Time	Error
BubbleSort (n=35) (n=140)	43 43	17 17	17 (3) $M_b(M_b)$	0 1	**2 (2)** **282 (311)**	0 0	**2 (3)** **265 (269)**	0 0
SelectionSort (n=35) (n=140)	34 34	17 17	18 (3) M_b (209)	0 1	**1 (1)** **161 (171)**	0 0	**1 (1)** **165 (173)**	0 0
InsertionSort (n=35) (n=140)	86 86	17 17	4 (5) **194** (283)	0 0	**3 (3)** **350 (219)**	0 0	**3 (3)** **212 (222)**	0 0
Prim	79	30	5 (2)	0	**<1 (<1)**	0	**<1 (<1)**	0
StrCmp	14	6	**11** (454)	0	**195 (257)**	0	**35 (46)**	0
MinMax	19	9	T_b (Mb)	1	**42 (7)**	0	**6 (7)**	0
lms	258	23	**225** (324)	0	**303 (307)**	0	**306 (307)**	0
Bitwise	18	1	**3** (6)	0	**7 (8)**	0	**30 (26)**	0
adpcm_encode	149	12	6 (26)	0	6 (6)	0	6 (6)	0
adpcm_decode	111	10	3 (27)	0	3 (3)	0	3 (3)	0

Comparison of SMT solvers

The native API is slightly faster than the SMT-LIB interface

Mo			C3		Boolector		Z3	
				rror	Time	Error	Time	Error
BubbleSort (n=35) (n=140)	43 43	17 17	17 (5) M_b(M_b)	0 1	**2 (2)** **282** (311)	0 0	**2** (3) **265** (269)	0 0
SelectionSort (n=35) (n=140)	34 34	17 17	18 (3) M_b (209)	0 1	**1 (1)** **161** (171)	0 0	**1 (1)** **165** (173)	0 0
InsertionSort (n=35) (n=140)	86 86	17 17	**4** (5) **194** (283)	0 0	**3 (3)** 350 (219)	0 0	**3 (3)** **212** (222)	0 0
Prim	79	30	5 (2)	0	**<1 (<1)**	0	**<1 (<1)**	0
StrCmp	14	6	**11** (454)	0	**195** (257)	0	**35** (46)	0
MinMax	19	9	T_b (Mb)	1	42 (7)	0	**6** (7)	0
lms	258	23	**225** (324)	0	**303** (307)	0	**306** (307)	0
Bitwise	18	1	**3** (6)	0	**7** (8)	0	30 (26)	0
adpcm_encode	149	12	**6** (26)	0	6 (6)	0	6 (6)	0
adpcm_decode	111	10	**3** (27)	0	3 (3)	0	3 (3)	0

Comparison of SMT solvers

The native API is slightly faster than the SMT-LIB interface, ***but not always***

Mo			VC3		Boolector		Z3	
				rror	Time	Error	Time	Error
BubbleSort (n=35)	43	17	**17** (5)	0	**2 (2)**	0	**2** (3)	0
(n=140)	43	17	M_b(M_b)	1	**282** (311)	0	**265** (269)	0
SelectionSort (n=35)	34	17	**18** (3)	0	**1 (1)**	0	**1 (1)**	0
(n=140)	34	17	M_b (209)	1	**161** (171)	0	**165** (173)	0
InsertionSort (n=35)	86	17	**4** (5)	0	**3 (3)**	0	**3 (3)**	0
(n=140)	86	17	**194** (283)	0	**350** (219)	0	**212** (222)	0
Prim	79	30	**5** (2)	0	**<1 (<1)**	0	**<1 (<1)**	0
StrCmp	14	6	**11** (454)	0	**195** (257)	0	**35** (46)	0
MinMax	19	9	T_b (Mb)	1	**42** (7)	0	**6** (7)	0
lms	258	23	**225** (324)	0	**303** (307)	0	**306** (307)	0
Bitwise	18	1	**3** (6)	0	**7** (8)	0	**30** (26)	0
adpcm_encode	149	12	**6** (26)	0	6 (6)	0	6 (6)	0
adpcm_decode	111	10	**3** (27)	0	3 (3)	0	3 (3)	0

Comparison to SMT-CBMC [A. Armando et al.]

- SMT-based BMC for C, built on top of CVC3 (hard-coded)
  - limited coverage of language
- Goal: compare efficiency of encodings

Module	ESBMC		SMT-CBMC
	Z3	CVC3	CVC3
BubbleSort (n=35) (n=140)	<1 (<1) 259 (265)	2 (2) M_b (M_b)	100 MO
SelectionSort (n=35) (n=140)	<1 (<1) 157 (162)	<1 (<1) 160 (193)	T T
BellmanFord	<1 (<1)	<1 (<1)	43
Prim	<1 (<1)	<1 (<1)	96
StrCmp	27 (38)	7 (261)	T
SumArray	25 (<1)	<1 (108)	98
MinMax	6 (6)	T_b (M_b)	65

Comparison to SMT-CBMC [A. Armando et al.]

- SMT-based BMC for C, built on top of CVC3 (hard-coded)
  - limited coverage of language
- Goal: compare efficiency of encodings

All benchmarks taken from SMT-CBMC suite

Module			
	Z3	CVC3	CVC3
BubbleSort **(n=35)** **(n=140)**	<1 (<1) 259 (265)	2 (2) M_b (M_b)	100 MO
SelectionSort **(n=35)** **(n=140)**	<1 (<1) 157 (162)	<1 (<1) 160 (193)	T T
BellmanFord	<1 (<1)	<1 (<1)	43
Prim	<1 (<1)	<1 (<1)	96
StrCmp	27 (38)	7 (261)	T
SumArray	25 (<1)	<1 (108)	98
MinMax	6 (6)	T_b (M_b)	65

Comparison to SMT-CBMC [A. Armando et al.]

- SMT-based BMC for C, built on top of CVC3 (hard-coded)
  - limited coverage of language
- Goal: compare efficiency of encodings

Module	ESBMC		SMT-CBMC
	Z3	CVC3	CVC3
BubbleSort (n=35) [illegible]	[illegible] (<1) [illegible]	**2** (2) M_b (M_b)	**100** MO
[illegible]	[illegible]	**<1** (<1) **160** (193)	**T** **T**
[illegible]	[illegible]	**<1** (<1)	**43**
Prim	<1 (<1)	**<1** (<1)	**96**
StrCmp	27 (38)	**7** (261)	**T**
SumArray	25 (<1)	**<1** (108)	**98**
MinMax	6 (6)	$\mathbf{T_b}$ (M_b)	**65**

ESBMC substantially faster, even with identical solvers
⇒ probably better encoding

Comparison to SMT-CBMC [A. Armando et al.]

- SMT-based BMC for C, built on top of CVC3 (hard-coded)
  - limited coverage of language
- Goal: compare efficiency of encodings

Module	ESBMC		SMT-CBMC
	Z3	CVC3	CVC3
BubbleSort (n[illegible]5) [illegible]40)	**<1 (<1)** **259 (265)**	**2 (2)** **M_b (M_b)**	100 MO
[illegible]) [illegible]0)	<1 (<1) **157 (162)**	<1 (<1) **160 (193)**	T T
BellmanFord	<1 (<1)	<1 (<1)	43
Prim	<1 (<1)	<1 (<1)	96
StrCmp	**27 (38)**	**7 (261)**	T
SumArray	**25 (<1)**	**<1 (108)**	98
MinMax	**6 (6)**	**T_b (M_b)**	65

Z3 uniformly better than CVC3