Extending the BTOR Language

Armin Biere and Florian Lonsing

extends joint work with Robert Brummayer published at BPR'08

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

SVARM'10
Edinburgh, UK

July 20, 2010

Overview

- overview of the current BTOR language: bit-vectors & arrays
- proposed extensions
  - tables
  - functions
  - quantifiers
  - commands
  - types
- design decisions, related work and conclusions

BTOR

- BTOR = native language of SMT solver Boolector
  - corresponds to QF_ABV of SMT-LIB — no quantifiers
  - but bit-vectors (BV),
  - arrays (A) and — actually an extensional theory of arrays
  - even a sequential extension for model checking — see BPR'08
- easy to parse, strongly typed, clean BV semantics
  - division by zero is fully defined — undefined in SMT-LIB
- all operators / constructors correspond to API calls of `libboolector.a`
- Boolector recently released under GPL — http://fmv.jku.at/boolector

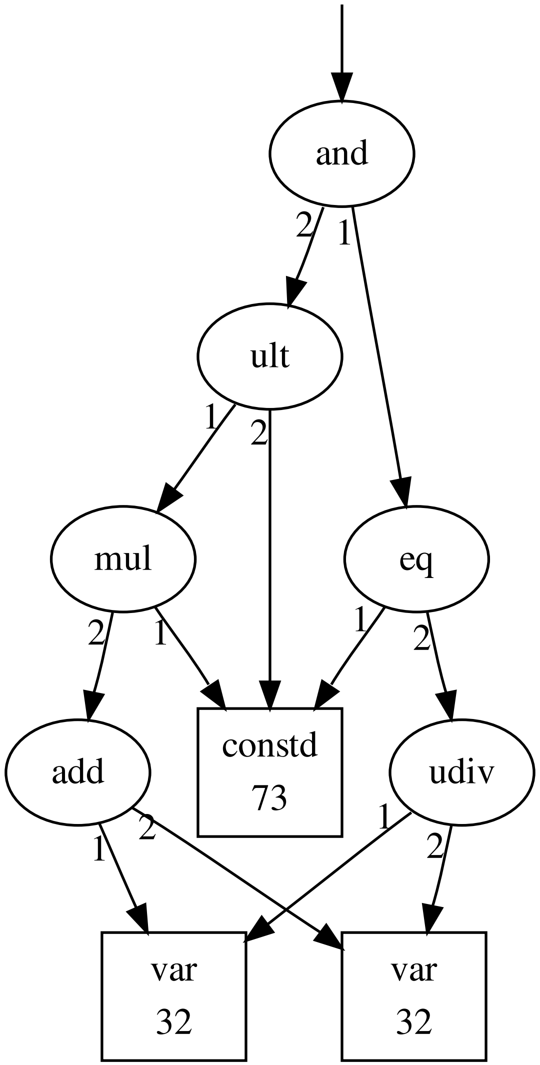
Bit-Vector Example (1/2)

```
1 var 32
2 var 32
3 constd 32 73
4 udiv 32 1 2
5 eq 1 3 4
6 add 32 1 2
7 mul 32 3 6
8 ult 1 7 3
9 and 1 5 8
10 root 1 9
```

```
id op bw id_or_num*
```

- first column: id
- second column: operator
- third column: bit-width of result
- other columns: id's of operands, or immediates

Bit-Vector Example (2/2)



Bit-Vector Constructors

- `var`

```
1 var 16 x
```

  - bit-width
  - optional string for back annotation

- `const` for binary constants

```
2 const 4 1101
```

- `constd` for decimal constants

```
3 constd 4 13
```

- `consth` for hexa-decimal constants

```
3 consth 4 d
```

Unary Bit-Vector Operators

class	operators	w_1	w_r
negation	**not**, neg	n	n
reduction	redand, redor, redxor	n	1
arithmetic	inc, dec	n	n

```
1 var 32
2 var 32
3 not 32 1
4 not 32 2
5 or 32 -3 -4
6 and 32 1 2
7 eq 1 5 -6
8 root 1 -7
```

- one's complement `not`
  - can also be expressed by a minus in front of an operand as in AIG's
- two's complement `neg`
- reduction operators from Verilog
- increment and decrement by one

Binary Bit-Vector Operators

class	operators	w_1	w_2	w_r
bitwise	**and**, or, xor, nand, nor, xnor	n	n	n
boolean	implies, iff	1	1	1
arithmetic	**add**, sub, **mul**, **urem**, srem **udiv**, sdiv, [us]mod,	n	n	n
relational	**eq**, ne, **ult**, slt, [us]lte, [us]gt, [us]gte	n	n	1
shift	**sll**, **srl**, sra, ror, rol	n	$log_2 n$	n
overflow	[us]addo, [us]subo, [us]mulo, sdivo	n	n	1
concatenation	**concat**	n_1	n_2	$n_1 + n_2$

- unsigned and signed context

- second operand of shift-operations has bit-width $log_2 n$

Ternary and Miscallenous Bit-Vector Operators

class	operators	w_1	w_2	w_3	w_r
conditional	**cond**	1	n	n	n

`cond` as the only ternary operator — if-then-else

class	operators	w_1	upper	lower	w_r
extract	**slice**	n	u	l	$u - l + 1$

`slice` extracts bits out of a bit-vector — operands are immediates

Arrays (1/2)

- BTOR supports one-dimensional bit-vector arrays
  - multi-dimensional arrays can be simulated by `concat` of operands
- constructor
  - `array e i`
  - elements have bit-width `e`
  - indices have bit-width `i`, i.e. size is $2^{\mathtt{i}}$

```
1 array 32 8
```

4 GB of memory

- array access
  - `read` can be used to model uninterpreted functions
  - `write` updates of arrays / functions

Arrays (2/2)

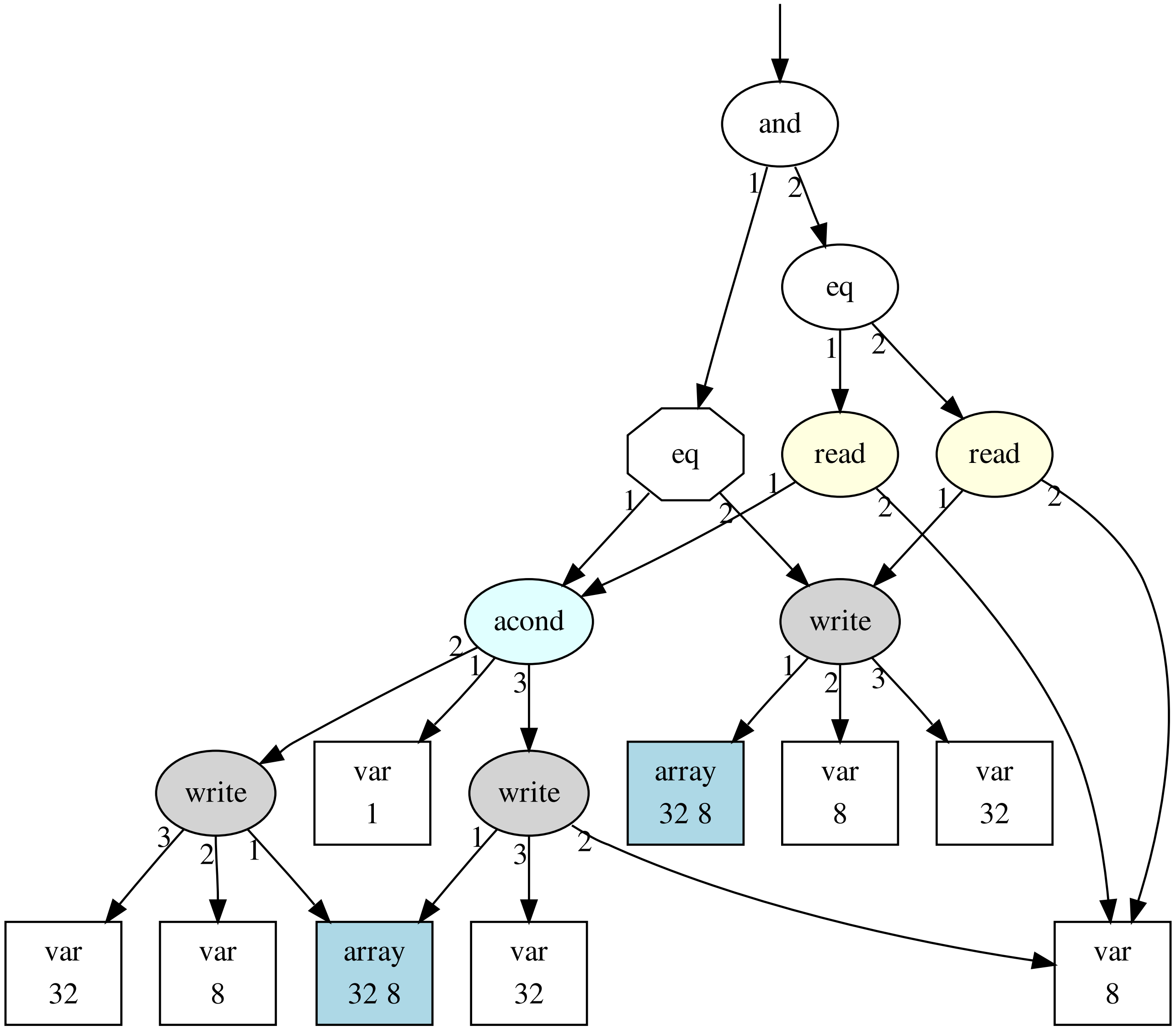
- if-then-else on arrays
  - `id acond ew iw cond then else`
- comparing arrays
  - arrays of the same type can be compared for equality with `eq`
  - two arrays are equal *iff* their elements are equal
- thus we have an extensional theory of arrays
  - can be used for comparing memory “before” and “after”
  - for instance equivalence checking of basic blocks in C with pointers

Array Example (1/2)

```
1 array 32 8
2 array 32 8
3 var 8
4 var 32
5 var 8
6 var 32
7 var 8
8 var 32
9 var 1
10 write 32 8 1 3 4
11 write 32 8 1 5 6
12 acond 32 8 9 10 11
13 write 32 8 2 7 8
14 eq 1 12 13
15 read 32 12 5
16 read 32 13 5
17 eq 1 15 16
18 and 1 14 17
19 root 1 18
```

- `write` and `acond` return an array of type `32 8`
- `read` returns a bit-vector of bit-width `32`

Array Example (2/2)



Design Decisions

- easy to parse
  - numerical ids, thus no symbol table   variable names can be added
  - simple single pass parser: read line by line
  - no yacc/lex, no recursive decent necessary
- also not hard to write / print, since there is no need for pretty printing
  - as in parsing: simple non-recursive implementation
- easy to script

```
awk '{a[$2]++}END{for(k in a)printf "%-7s%d\n", k, a[k]}' | sort -n -k 2
```

- strongly typed + fixed precise semantics

Tables

```
1 table 3 8
00000000
00000001
00000011
00000010
00000110
00000111
00000101
00000100
```

```
1 array 3 3
2 const 3 000
3 const 8 00000000
4 write 3 8 1 2 3
5 const 3 001
6 const 8 00000001
7 write 3 8 4 5 6
8 const 3 010
9 const 8 00000010
10 write 3 8 7 8 9
...
```

- initialization of constant memory   3-bit gray code in the example
  - used to model lookup-tables in programs
  - *will also be useful as internal operator*
- related zero initialized memory: `1 zarray 32 8`

Functions

- functions on bit-vectors are simply arrays without updates / `write`

- adding uninterpreted functions is a matter of syntatic sugar

```
1 fun 32 8              1 array 32 8
2 var 32                2 var 32
3 apply 8 1 2           3 read 8 1 2
```

- functions and arrays should be allowed to have multiple arguments

```
1 fun 32 1 8            1 array 32 1 8
2 var 32                2 var 32
3 var 1                 3 var 1
4 apply 8 1 2 3         4 read 8 1 2 3
```

- same applies to other associative operators
  - `concat`, `and`, ...

Quantifiers

- many verification (if not most) only need bit-vectors + arrays + quantifiers

  example: $\forall i, j [0 \leq i \leq j < n \rightarrow a[i] \leq a[j]]$

- first consider quantifiers over indices: $\forall x [\exists y [x = y]$ over 32-bit

```
1 var 32 x
2 var 32 y
3 eq 1 1 2
4 exists 1 3 2
3 forall 1 1 4
```

- methods for quantification
  - bit-blasting to QBF — needs (more) efficient QBF solvers
  - template based matching — yesterday's talk by Leonardo de Moura

Commands

- “ASCII API” to make Boolector “scriptable”

- add all current API functions to BTOR format

  - `assert`, `assume`, `sat`, `deref`, ...

- add new features to API and BTOR

  - `push`, `pop`, `failed`, `core`, `proof`, ...

- API is mostly the same as for plain SAT solvers such as PicoSAT

Types

- basic types
  - `bool`, `term`, `int`, `real`, ...
- constructors
  - `bv`, `array`, `fun`, ...
- replace bit-with argument at 3rd column by type id
- also merges “`acond`” and “`cond`” etc.

Related Work

- DIMACS, AIGER
  - based on the same similar principles as BTOR, e.g. only numeric id's
  - DIMACS = CNF, AIGER = AIG's
- Simplify, CVC, Z3, Spear native input formats
  - compromise between easy to read Simplify / CVC and
  - compact / easy to parse Z3 / Spear
- SMT-LIB, TPTP
  - extensible human-readable LISP/Prolog like syntax
  - SMT-LIB 2.0 is "scriptable", i.e. specifies "commands"

Conclusion

- BTOR is a clean and simple format for BV with arrays
- extensions needed in applications
  - without changing expressiveness: tables + functions + scripts
  - theory extensions: quantifiers + types
- could be a starting point for a compact SMT format
  - maybe even a binary format
- finally we need to extend Boolector to support all this