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 = native language of SMT solver Boolector
  - corresponds to QF_ABV of SMT-LIB
  - 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

- first column: id
- second column: operator
- third column: bit-width of result
- other columns: id’s of operands, or immediates
Bit-Vector Constructors

- **var**
  - bit-width
  - optional string for back annotation

- **const** for binary constants

- **constd** for decimal constants

- **consth** for hexa-decimal constants
Unary Bit-Vector Operators

<table>
<thead>
<tr>
<th>class</th>
<th>operators</th>
<th>( w_1 )</th>
<th>( w_r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>not, neg</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>reduction</td>
<td>redand, redor, redxor</td>
<td>( n )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>arithmetic</td>
<td>inc, dec</td>
<td>( n )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

- one’s complement \textit{not}
  - can also be expressed by a minus in front of an operand as in AIG’s

- two’s complement \textit{neg}

- reduction operators from Verilog

- increment and decrement by one
Binary Bit-Vector Operators

<table>
<thead>
<tr>
<th>class</th>
<th>operators</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitwise</td>
<td>and, or, xor, nand, nor, xnor</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>boolean</td>
<td>implies, iff</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>arithmetic</td>
<td>add, sub, mul, urem, srem udiv, sdiv, [us]mod,</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>relational</td>
<td>eq, ne, ult, slt, [us]lte, [us]gt, [us]gte</td>
<td>$n$</td>
<td>$n$</td>
<td>1</td>
</tr>
<tr>
<td>shift</td>
<td>sll, srl, sra, ror, rol</td>
<td>$n$</td>
<td>$log_2n$</td>
<td>$n$</td>
</tr>
<tr>
<td>overflow</td>
<td>[us]addo, [us]subo, [us]mulo, sdivo</td>
<td>$n$</td>
<td>$n$</td>
<td>1</td>
</tr>
<tr>
<td>concatenation</td>
<td>concat</td>
<td>$n_1$</td>
<td>$n_2$</td>
<td>$n_1 + n_2$</td>
</tr>
</tbody>
</table>

- unsigned and signed context

- second operand of shift-operations has bit-width $log_2n$
### Ternary and Miscellaneous Bit-Vector Operators

<table>
<thead>
<tr>
<th>class</th>
<th>operators</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>$w_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>conditional</td>
<td>cond</td>
<td>1</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

`cond` as the only ternary operator

<table>
<thead>
<tr>
<th>class</th>
<th>operators</th>
<th>$w_1$</th>
<th>upper</th>
<th>lower</th>
<th>$w_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract</td>
<td>slice</td>
<td>$n$</td>
<td>$u$</td>
<td>$l$</td>
<td>$u - l + 1$</td>
</tr>
</tbody>
</table>

`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^i`

- array access
  - read
  - write
    - can be used to model uninterpreted functions
    - updates of arrays / functions
• 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 \textit{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
Design Decisions

- easy to parse
  - numerical ids, thus no symbol table
  - 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

<table>
<thead>
<tr>
<th>No.</th>
<th>Operation</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>array</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>const</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>const</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>write</td>
<td>3</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>write</td>
<td>3</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>write</td>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>const</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>const</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>const</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>const</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>write</td>
<td>3</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>write</td>
<td>3</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

- Initialization of constant memory
  - Used to model lookup-tables in programs
  - Will also be useful as internal operator

- Related zero initialized memory:

```
1 zarray 32 8
```
functions on bit-vectors are simply arrays without updates / write

adding uninterpreted functions is a matter of syntactic sugar

functions and arrays should be allowed to have multiple arguments

same applies to other associative operators

- `concat`, and, ...
• 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
• “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”
• 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