BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking

Robert Brummayer Armin Biere Florian Lonsing

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

ABSTRACT

This is a proposal for a bit-precise word-level format, called BTOR. It is easy to parse and has precise semantics. In its basic form it allows to model SMT problems over the quantifier-free theory of bit-vectors in combination with one-dimensional arrays. Our main contribution is a sequential extension that can be used to capture model checking problems on the word-level. We present two case studies where BTOR is used as sequential format. Finally, we report on experimental results for the model checking extension of our SMT solver Boolector.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Verification; D.2.4 [**Software Engineering**]: Software/Program Verification; F.4.3 [**Mathematical Logic and Formal Languages**]: Decision Problems

Keywords

Word-Level Format, Bit-Vectors, Arrays, Model Checking

1. INTRODUCTION

To improve the state-of-the-art in automatic reasoning, standardized benchmarks are required. As a first step in producing such benchmarks one has to agree on an input format or at least on semantics. Not only for the developers, but also for the users of automatic reasoning tools, a simple standard format with precise semantics is of great value.

Prominent examples of such formats are, first of all, the DIMACS format [15] for SAT solvers, the TPTP format [22] for automatic theorem provers, the BAT format [18] for machine descriptions and LTL specifications, the Spear Format (SF) [3] for bit-vectors, and the CVC [1] and SMT-LIB format [20] for SMT solvers. More recently, the second author proposed a standard format for hardware model checking, called AIGER [5], which is used in the hardware model checking competition (HWMCC).

In the domain of model checking, with the exception of AIGER, every model checker seems to come with its own format. Differences in syntax and particularly in semantics make it very difficult to compare model checkers. For instance, in hardware model checking, common formats include SMV format [19], BLIF resp. BLIF-MV [8], and of course hardware description languages (HDLs) such as Verilog and VHDL. Similar arguments apply to software model checking formats like Promela [16], and real programming languages like Java [2, 23] and C [4, 13].

Even just for the SMV format, multiple interpretations exist on the precise semantics of the model, and in particular on how properties are interpreted. For example, consider an SMV model with an at least partial relational encoding of the transition relation in a TRANS section. This is typical for monitor constructions that model environment constraints, e.g. the Intel benchmarks in the HWMCC.

Consider the case where we want to check a simple safety property AGp. A counter example for this property consists of a path from an initial state to a bad state in which p does not hold. However, according to the original SMV semantics only those bad states are real bad states from which it is still possible to continue execution afterwards forever ($EG1$ holds in addition to $\neg p$). In particular, a state after which immediately, or after a while, no further transition is possible is not considered as a real bad state.

Various model checkers that accept the SMV input format treat this problem differently: one tool concludes that the property holds, the other not. It is also not always feasible to enforce the original SMV semantics, as the tool has to be able to check $EG1$, which for instance bounded model checkers can not. Various similar glitches exist among model checkers accepting the SMV input language. These problems become even more severe as soon as models have to be translated from one "rich" input language into the input language of another model checker.

In the AIGER format, the solution is to restrict the semantics to the least common denominator accepted by most model checkers. This solves issues with different interpretation of initial states and bad states. Some of the design decisions were to always enforce every latch resp. register to be zero initialized, outputs to be just bad state detectors, only one operation (AND) being supported, and last but not least, everything being represented on the bit-level.

Already the first time the AIGER format was discussed at the Alpine Verification Meeting (AVM'06), there was a hot debate, whether this last design decision, i.e. keeping everything on the bit-level, should not be taken back. We still think that a plain bit-level format has many advantages. The semantics are simpler than those of any word-level format. The developers can concentrate on improving the core algorithms. Furthermore, it is easier to convert benchmarks.

However, there are various techniques that can benefit from additional information which the word-level structure provides. This is one of the main arguments for applying SMT techniques to bit-precise problems. Our proposal for a new bit-precise format applies the same argument to word-level model checking.

Our proposed format BTOR is closer to an HDL language than the bit-vector format of SMT-LIB, for which BTOR can actually be seen as an alternative. However, the SMT-LIB format can not capture model checking problems. Finally, other alternative input formats, such as HDLs, word-level input languages of model checkers, or even programming languages, all have complex semantics. Translation, and even just parsing, is complex and error prone.

In order to foster research in model checking, it is important to have a simple and precise standardized format. A simple format places the burden of encoding a rich format and proper handling of complex semantics on the person resp. tool that generates the benchmark. However, it encourages the development of model checkers and thus will speed-up improving the state-of-the-art in model checking.

2. OVERVIEW

BTOR is a quantifier-free word-level format for formulas over bit-vectors in combination with one-dimensional arrays. In principle, it is a world-level generalization of the AIGER [5] format. It is strongly typed, easy to parse, multi-rooted, and has precise semantics. Unlike SF, which only supports bit-vectors up to 64 bits, bit-vectors in BTOR can have an arbitrary bit-width.

BTOR supports bit-vector variables, constants, and one-dimensional bit-vector arrays. Most bit-vector operations can be used in a signed or unsigned context. In the signed context, bit-vectors are interpreted as being represented in two's complement.

Similar to SF, the result of every operation is assigned to an intermediate variable, as the following example shows:

```
1 var 32          3 constd 32 127         5 eq 1 3 4
2 var 32          4 and 32 1 -2           6 root 1 5
```

In lines 1 and 2 we declare two 32-bit variables. In BTOR, a non-negative integer in the first column is used as unique identifier. Typically, we use the line number as identifier. In line 3 we declare the decimal 32-bit constant 127. Line 4 shows how computation is expressed in BTOR. We use the bit-wise operator *and* and apply it to the operands with the identifiers 1 and 2, which are the variables we have declared before. The result of this operation has the identifier 4. The minus before 2 expresses logical negation [1], i.e. we flip all bits of the variable 2 before we combine it with the variable 1. The result has 32 bits, which is indicated by the bit-width after the operator. In line 5 we compare the decimal constant with the subformula 4 for equality. The length of this result is 1, as relational operators are boolean. Finally, we declare the subformula 5 as boolean root.

This simple example already shows some interesting design decisions. The format should be easy to parse. The first column is used for the identifier, the second for the operator, and the third for the bit-width. This restriction makes it easy to write a parser for BTOR. The additional type information in the third column makes it easy for the parser to check type consistency on the fly.

Being able to add back annotation in form of an explicit symbol table simplifies many applications. In BTOR there is no separate symbol table as in AIGER. However, variable declarations can have an additional symbol part, separated by white space after the bit width. Comments start with '`;`' and stretch until the end of the line.

BTOR does not allow forward references, i.e. all operands have to be declared before they are used. This restriction makes parsing BTOR instances trivial, as it can be done in one pass. For example, our visualizer of the BTOR format consists of an AWK script of less than 80 lines, and produces a graph description in DOT format. A utility that prints a histogram of the number of operator occurrences can be implemented as the following simple AWK script:

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

As in Verilog, boolean variables are treated just as bit-vectors with bit-width one. This simplifies the format as we do not have to convert from boolean to bit-vector and vice versa. We can simply treat the boolean case as bit-vector case with bit-width one.

BTOR is multi-rooted, which allows to model multiple outputs. Multiple roots can be used to model hardware systems, which typically have multiple bit-vector outputs.

3. BIT-VECTORS

BTOR supports the following bit-vector constructors: *var* for bit-vector variables, *constd*, *consth* and *const* for decimal, hexadecimal and binary constants, and *one*, *ones* and *zero* for the constants 1, -1 and 0. All constructors take the bit-width as first argument. The constructor *var* takes an optional symbol string as second argument. The constructors *constd*, *consth* and *const* take the constant value as second argument.

The set of bit-vector operators is shown in tables 1, 2, 3 and 4. The columns w_1 to w_3 represent the bit-width of the operands. The column w_r represents the bit-width of the result. The semantics of most operators are defined by the semantics of the corresponding operators in the quantifier-free theory of fix-sized bit-vectors `QF_BV` in the SMT-LIB standard [20]. The only exceptions are as follows.

The SMT-LIB standard does not specify the result of dividing by zero. In BTOR the result of dividing by zero is the largest unsigned integer that can be represented with the bit-width of the operands. This corresponds to real divider circuits in hardware systems. Nevertheless, the underspecified variant of the SMT-LIB format can always be modelled by treating divison by zero as uninterpreted function.

The bit-width of shift operands are restricted in the following way. The bit-width of the first operand has to be a power of two. The bit-width of the second argument has to be log_2 of the bit-width of the first operand. If the bit-width of the second shift operand is log_2, then it is impossible to shift more than the bit-width, which for example is undefined in the programming language C [10, 17].

Additionally, BTOR supports the Verilog reduction operators *redand*, *redor* and *redxor*, the VHDL rotate operators

[1]Note that BTOR also supports the unary but redundant operator *not*.

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

Table 1: The unary bit-vector operators *not* and *neg* apply one's resp. two's complement. The operators *redand*, *redor* and *redxor* are reduction operators from Verilog. Finally, *inc* and *dec* are used to increment resp. decrement a bit-vector by one.

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, smod	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
concat	**concat**	n_1	n_2	$n_1 + n_2$

Table 2: Binary bit-vector operators. Some operators can be used in a signed or unsigned context, e.g. *sdiv* represents signed bit-vector division with two's complement semantics. For every arithmetic operator there is a corresponding overflow detection operator. The only exception is *udiv* as unsigned division can never overflow. Signed division overflows if we divide the smallest negative integer by -1.

rol and *ror*, and a new set of overflow detection operators. For example, the overflow detection operator *umulo* returns 1 if unsigned multiplication overflows. Consider the following example:

```
1 var 32             4 redand 1 2          7 and 1 6 -5
2 var 32             5 umulo 1 1 2         8 root 1 7
3 redand 1 1         6 and 1 3 4
```

The reduction operator *redand* returns 1, if all bits of the operand are set to one. In this case unsigned multiplication overflows. We assert the opposite in line 7, which makes this instance *unsatisfiable*.

Whether we want to detect arithmetic overflows or not, depends on the application scenario. Typically, in software verification we are interested in detecting overflows, as the results are often undefined and depend on compliler semantics [10]. However, in hardware verification overflow detection is in most cases unnecessary. In our C expression checker C32SAT [10] overflow detection is always automatically applied, which makes verification instances unnecessarily hard if we do not care about overflows. Thus, we separated overflow detection from arithmetic, and introduced an additional set of overflow detection operators in BTOR. We get only harder verification instances if we are interested in overflow detection.

Finally, it is of course possible to express some of our operators in terms of others. For instance our SMT solver Boolector [11, 12] supports the full set of operators, both

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

Table 3: The only ternary bit-vector operator *cond* represents a functional if-then-else. If the condition is 1, it returns the second argument, and the third argument otherwise.

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

Table 4: Miscellaneous bit-vector operators. The first operand identifies the variable to which *slice* is applied, *upper* and *lower* are immediates. The *slice* operator is the only operator that uses immediates.

in the application interface (API) and of course also in the parser for the BTOR format. Internally, however, we only use a sub-set of *base* operators, which are shown underlined and bold. Our selection can be considered to be arbitrary. It is motivated by the kind of word-level rewriting implemented in Boolector, which could be different for other solvers.

We can imagine various word-level techniques that can benefit from non-base operators. However, in contrast to the bit-level format AIGER, it is much harder to extract non-base word-level operators after they have been represented with base operators. Therefore, we argue that in a general format for bit-precise word-level modelling it should be possible to express and retain non-base operators. Nevertheless, it is always possible to rewrite benchmarks that contain non-base operators into the base format.

4. ARRAYS

Arrays can be constructed with the operator *array*. The first argument is the bit-width of the elements, and the second the bit-width of the addresses. BTOR supports bit-vector arrays in combination with the array operations *read*, *write*, *acond* and *eq*. Equality can be applied to arrays and array elements, i.e BTOR supports the extensional theory of arrays. Consider the following example:

```
1 array 32 4                  8 write 32 4 7 4 5
2 array 32 4                  9 read 32 8 4
3 array 32 4                  10 eq 1 5 9
4 var 4                       11 eq 1 3 8
5 var 32                      12 and 1 10 11
6 var 1                       13 root 1 12
7 acond 32 4 6 1 2
```

In the first three lines we declare three arrays with element bit-width 32, and size $2^4 = 16$. In line 7 we use an if-then-else on the arrays 1 and 2. If condition 6 holds, we return 1, and 2 otherwise. The result is an array with bit-width 32 and size 16. Line 8 writes on 7 the element 5 at index 4. The result is an array, where the element at position 4 is overwritten with 5. The elements on all other positions remain the same. In line 9 we read 32 bits on 8 at index 4, i.e. we read the value that has been written at this index before. In line 10 we compare the read value with 5. In line 11 we compare the arrays 3 and 8. Finally, we assert 10 and 11, and declare the result as root.

5. SEQUENTIAL EXTENSION

BTOR supports modelling sequential and synchronous circuits with registers and memories. Registers are modelled with the help of *var* and *next*, and memories with *array* and *anext*. Variables without *next*, and arrays without *anext*, are primary inputs, which are fresh in every clock cycle.

Registers are initialized to zero. Memories are uninitialized as this is the general case for main memory in software, and memory units in hardware. Consider the following:

```
1 var 32                3 xor 32 1 2
2 var 32                4 next 32 1 4
```

In line 5 we apply *next* to 1, which determines that 1 is a register. Variable 2 is an input as there is no *next* function applied to it. The *next* function applied to 1 is 4, i.e. in every cycle we apply *xor* to the current content of the register 1, and the primary input 2.

Additionally, BTOR supports modelling safety properties, which can be used for model checking. A safety property is represented by a root with bit-width one. Multiple roots with bit-width one are implicitly *disjuncted* as they represent different bad states. Another example with registers, one memory, and a safety property is shown in Fig. 1.

6. CASE STUDIES

We present two case studies where we show how BTOR, in particular its sequential extension, can be used to model hardware and software systems. In the first case study a FIFO with internal memory is modelled in BTOR. In the second case study we show how a program in a BASIC-like programming language can be *linearly* modelled in BTOR.

6.1 FIFOs

We took two different Verilog implementations of a typical hardware FIFO as shown in Fig. 2, and modelled both with BTOR. In this case we manually translated the Verilog models to BTOR, but in principle this could be automated. The first implementation organizes its internal memory as stack, while the second implementation uses a queue. Both implementations use head resp. tail registers which hold the address of the first resp. last element. The following Verilog code fragment shows how the queue-based implementation behaves if *dequeue* is active:

```
if (empty == 1'b0) begin
  data_out <= # 1 mem[head];
  head <= # 1 head + 3'b001;
end
if (head + 3'b001 == tail) begin
  empty <= # 1 1'b1;
end
full <= # 1 1'b0;
```

Consider the following BTOR fragment for a queue-based FIFO implementation. It shows how the *next*-operator can be used to model internal memory.

```
1 var 1 reset               7 var 1 full
2 var 1 enqeue              8 var 1 empty
3 var 1 deqeue              9 var 32 data_out
4 var 32 data_in            10 var 3 head
5 xor 1 2 3                 11 var 3 tail
6 array 32 3                ...
...                         ...
53 write 32 3 6 11 4        56 acond 32 3 5 55 6
54 acond 32 3 7 6 53        57 acond 32 3 1 56 6
55 acond 32 3 2 54 6        58 anext 32 3 6 57
```

```
1 array 8 32                ; next(flag)
2 var 32 start              21 next 1 5 8
3 var 32 end
4 var 32 p                  ; next(memory)
5 var 1 flag                22 write 8 32 1 4 6
                            23 eq 1 5 8
6 zero 8                    24 ult 1 4 3
7 one 32                    25 and 1 23 24
8 one 1                     26 acond 8 32 25 22 1
                            27 anext 8 32 1 26
; next(start)
9 var 32 start_in           ; check property
10 ne 1 5 8                 28 var 32 i
11 cond 32 10 9 2           29 read 8 1 28
12 next 32 2 11             30 ne 1 29 6
                            31 ult 1 2 3
; next(end)                 32 ult 1 28 2
13 var 32 end_in            33 ult 1 28 3
14 cond 32 10 13 3          34 and 1 -32 33
15 next 32 3 14             35 and 1 34 -10
                            36 and 1 31 35
; next(p)                   37 and 1 30 36
16 add 32 4 7               38 eq 1 4 3
17 ult 1 4 3                39 and 1 38 37
18 cond 32 17 16 4          40 root 1 39
19 cond 32 10 9 18
20 next 32 4 19
```

Figure 1: BTOR encoding of function `memclear` which sets a memory region from location *start* up to but not including *end* to zero. Memory is modelled as an 8-bit bit-vector array containing 2^{32} elements (line 1). In the first clock cycle indicated by register *flag* (lines 5 and 21), registers *start* and *end* are set to values *start_in* and *end_in* (lines 9-12 and 13-15), which represent user inputs, and remain constant afterwards. Value zero is written to memory (lines 22-27) at location *p*, which is first set to *start* and then incremented (lines 16-20) until *end*. Bad states (lines 28-40) represent that it is possible to read a non-zero value from a location between *start* and *end* after clearing memory has finished. The resulting instance is unsatisfiable.

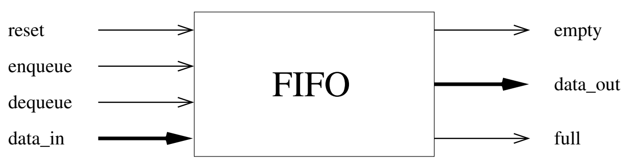

Figure 2: Hardware FIFO as black box. The clock signal is omitted.

Instruction	Semantics
`ADD arg`	$accu := accu + arg$
`NE arg`	$flag := (accu \neq arg)$
`JMP arg`	if $flag$ then $pc := arg$
`EXIT arg`	exit with code arg
`SAVE arg`	$arg := accu$

Table 5: Subset of the instruction set of a BASIC-like assembler for a hypothetical accumulator architecture. Instructions listed in the table are used in the example in the text. All instructions are unary. Address modes are register or immediate. There are 26 unsigned integer registers named *a* to *z*. A special register *flag* is used for conditional branches, which is set by a preceding relational instruction. Instructions `LOAD`, `SAVE`, `PEEK` and `POKE` read and write registers and memory locations, respectively. `EXIT` terminates the program with specified exit code.

In line 58 the *next*-operator is applied to the array, which models internal FIFO memory. The if-then-else chain can be read as follows. If *reset* is active (low), or *enqueue* and *dequeue* are equal, or *enqueue* is not active, or *enqueue* is active and the FIFO is full, memory remains in the same state. The if-then-else chain models the priority of the signals. However, if *enqueue* is active and the FIFO is not full, we write at the position the register *tail* points to the current input *data_in*. Analogously, we modelled the bit-vector registers *head*, *tail*, *empty*, *full* and *data_out*.

6.2 Checking assembly-level programs

As demonstrated in Fig. 1, model checking problems can be described in our BTOR format. We propose to model assembly-level programs in BTOR, where memory and registers are represented as bit-vector arrays. Program states can be related to each other by the *next*-operator on nested conditional expressions involving the program counter. Assertions can be regarded as safety properties. Violations can be detected by checking if bad states are reachable.

For demonstration purposes, we implemented a simulator and translator for a very simple, BASIC-like assembler for a hypothetical accumulator architecture as described in Tab. 5. An assembler program is translated into a BTOR model. The size of the model is linear in the size of the program. Failed assertions terminate the program with a non-zero exit code.

Consider the following trivial assembler program:

```
10 load 100          40 ne a           70 exit 1
20 add 1             50 jmp 70
30 save a            60 exit 0
```

After the accumulator has been assigned value 100 and incremented by one, its value is stored in register a (lines 10-30). If the accumulator value does not equal the value stored in a, then the program terminates with a non-zero exit code (lines 40, 50, 70). This behavior is regarded as an assertion failure. Fig. 3 shows the corresponding BTOR model.

In principle, our approach can be extended to subsets of more sophisticated assembly languages and architectures like x86.

```
1 var 32 pc
2 var 32 accu
3 var 1 flag
4 array 32 5
5 array 32 32
6 constd 32 0
7 constd 32 10
8 constd 32 20
9 constd 32 30
10 constd 32 40
11 constd 32 50
12 constd 32 60
13 constd 32 70
14 constd 5 0
15 write 32 4 14 2
16 read 32 4 14
17 eq 1 1 6
18 cond 32 17 7 1
19 constd 32 10
20 eq 1 1 7
21 cond 32 20 8 18
22 constd 32 100
23 cond 32 20 22 2
24 eq 1 1 8
25 cond 32 24 9 21
26 constd 32 1
27 add 32 23 26
28 cond 32 24 27 23
29 eq 1 1 9
30 cond 32 29 10 25
31 write 32 5 4 14 2
32 acond 32 5 29 31 4
33 eq 1 1 10
34 cond 32 33 11 30
35 ne 1 2 16
36 cond 1 33 35 3
37 eq 1 1 11
38 cond 32 37 12 34
39 and 1 37 3
40 cond 32 39 13 38
41 constd 32 70
42 eq 1 1 12
43 constd 32 0
44 redor 1 43
45 and 1 44 42
46 root 1 45
47 eq 1 1 13
48 constd 32 1
49 redor 1 48
50 and 1 49 47
51 root 1 50
52 next 32 1 40
53 next 32 2 28
54 next 1 3 36
55 anext 32 5 4 32
56 anext 32 32 5 5
```

Figure 3: BTOR model of the simple assembler program. The program counter *pc* can assume values for each line in the source code (lines 6-13), with zero as initial value. Its *next*-values are determined in nested conditional expressions (lines 52, 40, 38, 34, 30, 25, 21 and 18). If register *flag* is set at the `JMP` instruction (lines 37-40), then the new *pc* will equal 70, i.e. the jump to `EXIT 1` is executed. This kind of program termination corresponds to the failed assertion $REGS[a] = 101$ in line 40 of the program. If $pc = 30$, then the accumulator value is written to register a, which has index 0 (line 14) in the bit-vector array modelling the register set (lines 55, 32, 31, 29). All other registers remain constant. In this example, memory is not written and remains constant (line 56), but the idea for modelling memory writes is similar as for registers. The accumulator is assigned value 100 and incremented by one (lines 53, 28, 23). Exit code zero designates normal program termination.

	boolector				z3	
	inc		non-inc		non-inc	
k	adc	noadc	adc	noadc	adc	noadc
00	0.0	0.0	0.0	0.0	0.0	0.0
01	0.1	0.0	0.1	0.0	0.0	0.0
02	0.8	0.1	0.9	0.1	300.3	0.0
03	15.4	0.3	6.1	0.4	639.9	313.1
04	41.9	0.7	38.0	1.1	*to*	*to*
05	91.2	1.6	96.5	2.7	*to*	*to*
06	419.8	3.7	267.8	5.8	*to*	*to*
07	*to*	6.8	*to*	11.7	*to*	*to*
08	*to*	14.3	*to*	23.9	*to*	*to*
09	*to*	31.1	*to*	47.8	*to*	*to*
10	*to*	73.7	*to*	97.2	*to*	*to*

Table 6: Experiments for equivalence checking of the FIFOs in 6.1. We compare our SMT solver Boolector with Z3 1.2. We ran our benchmarks on our cluster of 3 GHz Pentium IV with 2 GB main memory, running Ubuntu Linux. Time limit is 900 seconds and memory limit is 1500 MB. Time out is indicated by *to*.

7. EXPERIMENTS

We extended our SMT solver Boolector [11, 12] for checking sequential BTOR, i.e. we implemented a bounded model checker within Boolector. Boolector uses a functional AIG encoding including two level AIG rewriting [9]. PicoSAT [6] is used as SAT solver. Boolector supports bounded model checking for witnesses [7], and k-induction [21] with and without All Different Constraints (ADCs). ADCs are used to represent simple path constraints. Note that model checking instances with memories generates ADCs with inequalities on arrays, which need extensionality [11, 12].

We compared Boolector's incremental model checking with non-incremental Boolector and Z3 1.2 [14]. The benchmarks are parametrized instances for checking behavioral equivalence of two FIFO implementations as presented in 6.1. Bitwidth and size of memory is 32. The results are shown in table 6. Column 1 shows the upper bound of the model checking instance. Column 2 shows the time of incremental model checking, which includes searching for witnesses and k-induction with ADCs. In column 3 ADCs are disabled.

The non-incremental results have been computed in the following way. First, we generated SMT instances that represent (*i*) search for witnesses, (*ii*) k-induction with ADCs, and (*iii*) k-induction without ADCs. In columns 4 and 6 we summed up the solving times for the results of (*i*) and (*ii*), from 0 to k. Analogously, we computed column 5 and 7 by summing up the results of (*i*) and (*iii*). Both versions of Boolector clearly outperform Z3. In the case where ADCs are disabled, the incremental model checking of Boolector is faster than the non-incremental.

8. CONCLUSION

We proposed a bit-precise word-level format BTOR. It is easy to parse, has precise semantics, and allows to model SMT problems over the theory of bit-vectors in combination with one-dimensional arrays. Our main contribution is a sequential extension of BTOR to model registers and memories. We discussed two sequential case studies. Finally, our experiments show that our incremental model checker can efficiently decide model checking problems in BTOR.

9. REFERENCES

[1] The CVC3 user's manual, 2007. `www.cs.nyu.edu/acsys/cvc3/doc/user_doc.html`.

[2] C. Artho, V. Schuppan, A. Biere, P. Eugster, and B. Zweimüller. JNuke: Efficient dynamic analysis for Java. In *Proc. CAV*, 2004.

[3] D. Babic. SPEAR modular arithmetic format specification - Version 1.0, December 2007. `www.cs.ubc.ca/~babic/doc/spear_modarith.pdf`.

[4] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. SPIN*, 2001.

[5] A. Biere. The AIGER And-Inverter Graph (AIG) format. Available at `fmv.jku.at/aiger`.

[6] A. Biere. PicoSAT essentials. *JSAT*, 2008.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, 1999.

[8] R. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. Kurshan, S. Malik, A. Sangiovanni-Vincentelli, E. Sentovich, T. Shiple, K. Singh, and H. Wang. BLIF-MV: An interchange format for design verification and synthesis. Technical Report M91/97, 1991.

[9] R. Brummayer and A. Biere. Local two-level And-Inverter Graph minimization without blowup. In *Proc. MEMICS*, 2006.

[10] R. Brummayer and A. Biere. C32SAT: Checking C expressions. In *Proc. CAV*, 2007.

[11] R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. In *Proc. SMT*, 2008.

[12] R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proc. TACAS'09*, 2009. To appear.

[13] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, 2004.

[14] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACACS*, 2008.

[15] DIMACS. Satisfiability suggested format, 1993.

[16] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[17] ISO/IEC. Programming languages - C (ISO/IEC 9899:1999(E)), 1999.

[18] P. Manolios. BAT documentation, 2008. www.cc.gatech.edu/~manolios/bat/doc.html.

[19] K. McMillan. *Symbolic Model Checking: An approach to the State Explosion Problem*. Kluwer, 1993.

[20] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, 2006. Available at `www.SMT-LIB.org`.

[21] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. FMCAD*, 2000.

[22] G. Sutcliffe. The TPTP problem library - TPTP v3.4.0, 2008.

[23] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. IEEE Intl. Conf. Automated Software Engineeering (ASE)*, 2000.