The AIGER And-Inverter Graph (AIG) Format Version 20070427

Armin Biere, Johannes Kepler University, 2006-2007

17th May 2007

This report describes the AIG file format as used by the AIGER library. The purpose of this report is not only to motivate and document the format, but also to allow independent implementations of writers and readers by giving precise and unambiguous definitions.

Acknowledgements

The format went through various incarnations even before real code became available. In particular the following colleagues gave invaluable feedback on earlier drafts.

> *Robert Brummayer, Alessandro Cimatti, Koen Claessen, Niklas Een, Marc Herbstritt, Hyondeuk Kim, Toni Jussila, Ken McMillan, Alan Mishchenko, Fabio Somenzi, Niklas Soerenson, Allen Van Gelder.*

We also want to thank *Holger Hermanns* who started the discussion on having a model checking competition affiliated to CAV and also provided strong support after the idea became more concrete.

Availability

You can download up-to date versions of this report, the AIGER library, and utilities from `http://fmv.jku.at/aiger`.

1 Introduction

North face of the Eiger.

The name AIGER contains as one part the acronym AIG of And-Inverter Graphs and also if pronounced in German sounds like the name of the 'Eiger', a mountain in the Swiss alps. This choice should emphasize the origin of this format. It was first openly discussed at the Alpine Verification Meeting 2006 in Ascona as a way to provide a simple, compact file format for a model checking competition affiliated to CAV 2007.

The AIGER format has an ASCII and a binary version. The ASCII version is the format of choice if an AIG is to be saved by an application which does not want to use the AIGER library. It is simple to generate and has less restrictions. The easiest way to write an AIG in AIGER format is of course to use the AIGER library.

The binary format is more restricted and much more compact. It is easier to read and thus is the format of choice for benchmarks and competitions. The average user can either use the AIGER library to generate a binary file directly or if it is impossible to use the library due to LICENSE or language restrictions, generate a file in ASCII format and transform it into the binary format with the `aigtoaig` utility. The reverse is also possible, if for instance a human readable version of an AIG in binary AIGER format is required. Another usage of the `aigtoaig` tool is to validate AIGs in binary AIGER format produced by other tools not using the AIGER library.

Let us start with some simple examples in ASCII format which all can be found in the `examples` subdirectory. An AIGER file in ASCII format starts with the format identifier string `aag` for ASCII AIG and 5 non negative integers `M`, `I`, `L`, `O`, `A` separated by spaces.

The interpretation of the integers is as follows

```
M = maximum variable index
I = number of inputs
L = number of latches
O = number of outputs
A = number of AND gates
```

The empty circuit without inputs nor outputs consists of the single line

```
aag 0 0 0 0 0                header
```

The file that consists of the following two lines

```
aag 0 0 0 1 0                header
0                            output
```

represents the constant FALSE. Note that the comments to the right are not part of the file. The following file represents the constant TRUE:

```
aag 0 0 0 1 0
1
```

The single `1` in the header specifies that the number of outputs is one. In this case the header is followed by a line which contains the literal of the single output.

The following file is a buffer

```
aag 1 1 0 1 0                header
2                            input
2                            output
```

and the following an inverter

```
aag 1 1 0 1 0                header
2                            input
3                            output          !1
```

The maximal variable index is 1, the first number. The second number represents the number of inputs. The first line consists of the single input literal. A variable is transformed into a literal by multiplying it by two. In the first case the output specified on the last line leaves the input unchanged. In the second case the output inverts the input. The output literal `3` has its sign bit, the least significant bit, set to one accordingly.

An AND gate looks as follows

```
aag 3 2 0 1 1
2                           input 0
4                           input 1
6                           output 0
6 2 4                       AND gate 0      1 & 2
```

The literal representing the AND gate is `6` with variable index `3`, the first number in the header, which denotes the maximal variable index. The last number is the number of AND gates.

An OR gate can be formulated as

```
aag 3 2 0 1 1
2                           input 0
4                           input 1
7                           output 0        !(!1 & !2)
6 3 5                       AND gate 0      !1 & !2
```

Let us now turn to a more complete combinational circuit

```
aag 7 2 0 2 3               header line
2                           input 0         1st addend bit 'x'
4                           input 1         2nd addend bit 'y'
6                           output 0        sum bit        's'
12                          output 1        carry          'c'
6 13 15                     AND gate 0      x ^ y
12 2 4                      AND gate 1      x & y
14 3 5                      AND gate 2      !x & !y
i0 x                        symbol
i1 y                        symbol
o0 s                        symbol
o1 c                        symbol
c                           comment header
half adder                  comment
```

The symbol table is optional and does not need to be complete, but may only contain symbols for inputs, latches, or outputs.

Sequential circuits have latches as state elements. Here is a toggle flip flop, which has no input, one latch, and two outputs, its current state and its negation:

```
aag 1 0 1 2 0
2 3                         latch 0 with next state literal
2                           output 0
3                           output 1
```

Latches are always assumed to be initialized to zero. The same toggle flip flop with an enable and additional explicit active low reset input:

```
aag 7 2 1 2 4
2                           input 0         'enable'
4                           input 1         'reset'
6 8                         latch 0         Q next(Q)
6                           output 0        Q
7                           output 1        !Q
8 4 10                      AND gate 0      reset & (enable ^ Q)
10 13 15                    AND gate 1      enable ^ Q
12 2 6                      AND gate 2      enable & Q
14 3 7                      AND gate 3      !enable & !Q
```

The order of the literals and the definitions of the AND gates is irrelevant. The binary format described more formally below places more restrictions on the order and also does not allow unused literals.

2 Design Choices

- The format should allow to model combinational circuits.
- Structural SAT problems can be described.
- The format should allow to model sequential circuits.
- Model checking problems can be described.
- The operators are restricted to bit level.
- The set of operators needs to be as simple as possible.
- A compact standardized binary format should be available.
- The ASCII format should be as easy as possible to write by programs.
- The binary format should be as easy as possible to read by programs.
- A symbol table and comments can be included.
- Symbol table and comments can be ignored reading the file sequentially.
- Some simple form of extensibility should be possible.

3 Header

The AIGER format describes circuits by multi-rooted And-Inverter Graphs (AIGs). A file in AIGER format has to start with a format identifier string, which is either `aag` for ASCII format or `aig` for the binary format. After the format identifier string and one space character follow 5 non negative integers `M`, `I`, `L`, `O`, and `A` in ASCII encoding. The

maximal variable index `M` is the first number in the header. The circuit has `I` inputs, `L` latches, `O` outputs, and consists of `A` AND gates. If all variables are used and there are no unused AND gates then `M = I + L + A`.

An unsigned respectively non negative integer is either `0` or a strictly positive ASCII encoded digit followed by a sequence of arbitrary digits including `0`. The `a` of the format identifier string is the first character of the file. The format identifier string `aag` respectively `aig` and the numbers are all separated by exactly one space character. The header line ends with a new line character immediately after the last digit of the number of ands `A`.

4 Variables and Literals

Literals are constants or signed variables and are also represented by unsigned integers. The least significant bit of the unsigned word encoding a literal is the sign bit. The remaining bits represent the variable index.

In essence, to obtain the sign bit of a literal we take its unsigned integer representation modulo 2. A sign bit of one means negated, a sign bit of zero unnegated. To extract the variable from a literal we divide it by 2. To obtain a literal from a variable in the other direction we multiple the variable index by 2 and optionally add 1 if the variable should be negated. A literal can be negated by just toggling its least significant bit.

The constant FALSE is represented by the literal `0`, the constant TRUE by `1`. This implies, that FALSE is unnegated, while TRUE is negated.

AIGER only models cycle-accurate models. Latches are assumed to have a reset zero state, i.e., are initialized to zero, and no explicit reset nor clock signal is necessary. More general circuit models with non zero reset states or even non deterministic transition relations can be translated into the AIGER format by simple constructions. See for instance the techniques implemented in `smvtoaig` to translate arbitrary flat and boolean encoded SMV models into AIGER format.

5 ASCII Format

The ASCII format has `aag` as format identifier in the header, which is an acronym for ASCII AIG. The names of files in the ASCII format are supposed to have an `.aag` extension. The AIGER library also supports files compressed with GNU GZIP. In this case an additional `.gz` suffix is expected.

After the header the `I` inputs are listed, one per line, as unnegated literal, i.e., represented as even positive integers. Then the `L` latches are defined. Again one latch per line. Each line consists of exactly two positive integers separated by a space character. The first is even and denotes the current state of the latch. The second is the literal that defines the next state function of this latch.

Then the `O` output literals are listed, one per line. Here arbitrary literals are allowed.

This concludes the interface part and the definitions of the `A` AND gates follow.

The definition of an AND gate consists of three positive integers all written on one line and separated by exactly one space character. The first integer is even and represents the literal or left-hand side (LHS). The two other integers represent the literals of the right-hand side (RHS) of the AND gate.

In order to be well formed, the `I` inputs, the current state literal of the `L` latches, i.e., the pseudo-primary inputs, and the LHS of the `A` AND gates are all different and define exactly `I + L + A` literals. The other literals, except for the two constants `0` and `1`, are undefined and can not be used as output, as next state literal, nor on the RHS of an AND. Furthermore, the definitions of the ANDs have to be acyclic. To be more precise: The literal on the LHS of an AND is defined to depend on the literals on the RHS after stripping sign bits. The transitive non reflexive closure of this dependency relation has to be acyclic.

In the ASCII format both checking for undefined literals and checking for cyclic dependencies has to be done explicitly. This is the price one has to pay for a simple, easy to generate and less restricted format. The binary format on the other hand has a strict order requirement on the literals not only for inputs and latches, but also for the RHS literals with respect to the LHS literals. The binary format does not require such an explicit check. The AIGER library and all the tools including `aignm` and `aigtoaig`, which read AIGs in AIGER format with the AIGER library, can be used to validate AIGER files in ASCII or binary format.

Finally note, that in the ASCII format the number of defined literals does not have to match the maximal variable index `M`. In essence, some literals may just not be used in ASCII format.

6 Symbols

After the definitions of the AND gates an optional symbol table may follow. A symbol is an arbitrary ASCII string of printable characters excluding the new line character. Symbols can only be attached to inputs, latches and outputs and there is at most one symbol per input, latch or output allowed. A symbol entry makes up one line in the input and consists of a symbol type specifier which is either `i`, `l`, or `o` on the first character position, followed by a position, which is not separated by a space. After a space character the symbol name starts and continues until but not including the next new line character. Therefore a symbol table entry looks as follows:

```
[ilo]<pos> <string>
```

The position `pos` of the symbol denotes the position of the input, latch, or output, in the list of inputs, latches, and outputs respectively. It has to follow immediately the symbol type identifier without space.

The symbol table is put after all the definitions, such that an application reading an AIG can just stop reading the file after it has read the definitions. The same applies to

comments which come last.

7 Comments

After the symbol table an optional comment section may start. The comment section starts with a `c` character followed by a new line. The following lines are comments. Each comment starts at the first character of a line and extends until the next new line character. There can be no comment lines at all, but the last comment has to be terminated by a new line character, which then also has to be the last character of the file.

8 Binary Format Motivation

The binary format is more restricted than the ASCII format. Literals have to follow a specific order. The order restrictions and using a two-complement binary representation of numbers makes the binary format not only easier to read but also much more compact. Experiments show that these restrictions and an additional delta encoding result in smaller uncompressed files. The binary format is often smaller than the same model in GZIP compressed ASCII format. Without symbol tables the reduction is typically a factor of two compared to the GZIP compressed ASCII format. Compressing the binary format results in additional reduction.

As example consider the following SMV model `texas.parsesys^1.E.smv`, the largest benchmark from the TIP suite:

```
1204637 unmangled.smv
 485595 mangled.smv
 191045 unstripped.aag
 185098 stripped.aag
 114796 unmangled.smv.gz
  97455 mangled.smv.gz
  72486 unstripped.aag.gz
  70693 stripped.aag.gz
  44044 unstripped.aig
  38097 stripped.aig
  28061 unstripped.aig.gz
  26226 stripped.aig.gz
```

After flattening and binary encoding, the SMV model has a size of 1204637 bytes (unmangled.smv). Simply renaming the symbols, e.g. mangling them, results in a reduction in file size of almost a factor of 3. At least another factor of two can be obtained by translating the original unmangled SMV file into an AIG and writing it in the ASCII format (unstripped.aag) with our tool `smvtoaig`. This file still contains all the unmangled symbols. Stripping the symbol table reduces the size slightly (stripped.aag).

Compressing these four files with GZIP results in another reduction of at least 2. In particular note, that the uncompressed and unstripped binary format is two third the size of the compressed and stripped ASCII format. Finally using the binary format, gives almost an additional factor of two. Compressing the binary format gives further reduction.

For the whole TIP suite the sizes are as follows

```
4393611 unmangled.tar.gz
3099882 mangled.tar.gz
2676172 aigtosmv.tar.gz
2303631 aag.tar.gz
 347325 aig.tar.gz
```

Here we stripped the symbols in the AIG case. The tar file `aigtosmv.tar.gz` contains the result of translating the files from binary format (`tip/aig.tar.gz`) back into SMV format. It is also remarkable, that with the exception of the binary format, GZIP can not make use of the fact that most of the models in the TIP suite occur multiple times though with different properties.

Finally, as a combinational example consider the CNF benchmark `f10bidw` submitted by P. Manolios and S. K. Srinivasan to the SAT Race 2006 with 800k variables and roughly 2.4 million clauses. Since the original benchmark consists of only AND gates and one unit clause, we could easily extract a corresponding AIG with our tool `cnf2aig`.

```
55125988 f10bidw.cnf
17728435 f10bidw.aag
11743646 f10bidw.cnf.gz
 5827259 f10bidw.aag.gz
 2904681 f10bidw.aig
 1931015 f10bidw.aig.gz
```

These experiments show that using a binary format can considerably reduce space requirements for storing benchmarks. This reduction is even more important if the AIGER format is used for large combinational benchmarks, e.g. in a SAT context as in the last example.

9 Binary Format Definition

The binary format is semantically a subset of the ASCII format with a slightly different syntax. The binary format may need to reencode literals, but translating a file in binary format into ASCII format and then back in to binary format will result in the same file.

The main differences of the binary format to the ASCII format are as follows. After the header the list of input literals and all the current state literals of a latch can be

omitted. Furthermore the definitions of the AND gates are binary encoded. However, the symbol table and the comment section are as in the ASCII format.

The header of an AIGER file in binary format has `aig` as format identifier, but otherwise is identical to the ASCII header. The standard file extension for the binary format is therefore `.aig`.

A header for the binary format is still in ASCII encoding:

```
aig M I L O A
```

Constants, variables and literals are handled in the same way as in the ASCII format. The first simplifying restriction is on the variable indices of inputs and latches. The variable indices of inputs come first, followed by the pseudo-primary inputs of the latches and then the variable indices of all LHS of AND gates:

```
input variable indices        1,          2,  ... ,  I
latch variable indices      I+1,        I+2,  ... ,  (I+L)
AND variable indices      I+L+1,      I+L+2,  ... ,  (I+L+A) == M
```

The corresponding unsigned literals are

```
input literals                2,          4,  ... ,  2*I
latch literals            2*I+2,      2*I+4,  ... ,  2*(I+L)
AND literals          2*(I+L)+2,  2*(I+L)+4,  ... ,  2*(I+L+A) == 2*M
```

All literals have to be defined, and therefore `M = I + L + A`. With this restriction it becomes possible that the inputs and the current state literals of the latches do not have to be listed explicitly. Therefore, after the header only the list of `L` next state literals follows, one per latch on a single line, and then the `O` outputs, again one per line.

In the binary format we assume that the AND gates are ordered and respect the child parent relation. AND gates with smaller literals on the LHS come first. Therefore we can assume that the literals on the right-hand side of a definition of an AND gate are smaller than the LHS literal. Furthermore we can sort the literals on the RHS, such that the larger literal comes first. A definition thus consists of three literals

```
lhs rhs0 rhs1
```

with `lhs` even and `lhs > rhs0 >= rhs1`. Also the variable indices are pairwise different to avoid combinational self loops. Since the LHS indices of the definitions are all consecutive (as even integers), the binary format does not have to keep `lhs`. In addition, we can use the order restriction and only write the differences `delta0` and `delta1` instead of `rhs0` and `rhs1`, with

```
delta0 = lhs - rhs0,  delta1 = rhs0 - rhs1
```

The differences will all be strictly positive, and in practice often very small. We can take advantage of this fact by the simple little-endian encoding of unsigned integers of the next section. After the binary delta encoding of the RHSs of all AND gates, the optional symbol table and optional comment section start in the same format as in the ASCII case.

10 Binary Encoding of Deltas

Assume that `w0, ..., wi` are 7-bit words, `w1` to `wi` all non zero and the unsigned number `x` can be represented as

```
x = w0 + 2^7*w1 + 2^14*w2 + 2^(7*i)*wi
```

The binary encoding of x in AIGER is the sequence of i bytes b0, ... bi:

```
1w0, 1w1, 1w2, ..., 0wi
```

The MSB of a byte in this sequence signals whether this byte is the last byte in the sequence, or whether there are still more bytes to follow. Here are some examples:

```
unsigned integer   byte sequence of encoding (in hexadecimal)

               x   b0 b1 b2 b3

               0   00
               1   01
2^7-1    =   127   7f
2^7      =   128   80 01
2^8  + 2 =   258   82 02
2^14 - 1 = 16383   ff 7f
2^14 + 3 = 16387   83 80 01
2^28 - 1           ff ff ff 7f
2^28 + 7           87 80 80 80 01
```

This encoding can reduce the number of bytes by at most a factor of 4, which very often in practice is almost reached, in particular in our application where many small numbers are expected.

This binary encoding of arbitrary precision unsigned integers is platform-independent and thus 64-bit clean. Unfortunately, this is not true for the following code snippets in C. We also just ignore overflows and file errors, but otherwise this code shows that encoding and decoding is very simple.

```
unsigned char
getnoneofch (FILE * file)
{
  int ch = getc (file);
  if (ch != EOF)
    return ch;

  fprintf (stderr, "*** decode: unexpected EOF\n");
  exit (1);
}

unsigned
decode (FILE * file)
{
  unsigned x = 0, i = 0;
  unsigned char ch;

  while ((ch = getnoneofch (file)) & 0x80)
    x |= (ch & 0x7f) << (7 * i++);

  return x | (ch << (7 * i));
}

void
encode (FILE * file, unsigned x)
{
  unsigned char ch;

  while (x & ~0x7f)
    {
      ch = (x & 0x7f) | 0x80;
      putc (ch, file);
      x >>= 7;
    }

  ch = x;
  putc (ch, file);
}
```

Not checking that the next character is an EOF may result in an infinite loop. In the binary format of AIGER we always expect a complete sequence. Therefore if an EOF is read before the sequence is complete a parse error occurs. A simple solution to this problem is to check the return value of `getc`. If the value is EOF then abort decoding with a parse error.

11 Property Checking

The AIGER format can easily be used for various types of property checking. A combinational circuit with no latches, i.e., L = 0, and exactly one output can be assumed to be a circuit for which we want to force the output to be one. This is an encoding of SAT.

For sequential circuits model checking of simple safety properties is encoded in the same way. To check liveness we interpret each of the outputs as fairness constraint. An algorithm that finds a reachable fair cycle for these fairness constraints allows to model check LTL properties.

In addition we plan to support PSL properties in the following way. PSL properties are written in a separate file. As atomic properties only outputs of the circuit are allowed. The outputs are referenced through a mandatory symbolic name in the symbol table.

12 Vectors, Stimuli, Traces, Solutions and Witnesses

In this section we define the semantics and syntax of traces and solutions to property checking problems. More specifically we only consider structural SAT solving and bad state detection witnesses.

In essence, a valid solution is a list of input vectors. An input vector may contain beside `0` and `1` also `x` values. Such a list of input vectors is a valid witness iff for any instantiation of the `x` values by `0` or `1` two-valued simulation will produce at least one `1` at the output of the AIG, assuming that the AIG starts in its initial all `0` state. For structural SAT problems, the AIG does not contain latches. In this case one input vector is enough.

In principle one can use three valued logic for fast simulation of AIGs under partial two-valued input assignments. Beside two-valued logic constants `0` and `1`, we also use the logic constant `x`. Operations in this three-valued logic are as usual:

```
a | !a     & | 0 1 x
--+---     --+------
0 | 1      0 | 0 0 0
1 | 0      1 | 0 1 x
x | x      x | 0 x x
```

A vector of values or short vector of size `n` is a list of `n` three-valued constants. A vector of size `n` is represented by an ASCII string of length `n` containing only the characters `0`, `1`, and `x`.

In the following fix an AIG in AIGER format of type `M I L O A`. An input vector is a vector of size `I`, an output vector is a vector of size `O`, and a state vector is a vector of size `L`.

A stimulus for an AIG in AIGER format is a list of input vectors. The file format for

a stimulus consists of lines of ASCII strings that represent the input vectors separated and terminated by exactly one new line character.

A transition contains, in this order, a current state, input, output, and next state vector. A transition is consistent with the logic of the given AIG if the result of simulating the current state vector under the input vector in three-valued logic produces the given output and next state. In the ASCII representation of a transition, the strings representing the four vectors are separated by space characters, exactly one space character between two strings. This implies that a transition for a combinational circuit with `L = 0` starts and ends with one space character. A combinational circuit without any input would even have two leading space characters.

A trace consists of a list of transitions. A trace is consistent if all its transitions are consistent, and the current state vector of all transitions except the first has the same value as the next state vector of its previous transition. A trace is initialized if the current state vector of the first transition only contains `0`. Each line of a trace file consist of exactly one transition. Again we use exactly one new line character to separate and terminate the list of lines.

Note that we only consider 'cycle accurate simulation' here. Furthermore the definitions also make sense for combinational circuits with `L = 0`. In this case, the state vectors will just be vectors of length 0 and represented by empty strings.

A simulator thus takes as input an AIG and a stimulus matching the type of the AIG and produces an initialized and consistent trace. A randomized simulator will use randomized input vectors and does not need a stimulus.

The `x` value should not be interpreted as don't care. For instance, if the literal `l` is assigned to the value `x`, then `l & !l` produces `x` in three-valued simulation and not `0`. This standard semantic of three-valued logic allows linear time simulation of a circuit under a total three-valued assignment to the inputs. Simulating partial assignments in two-valued logic, more specifically checking whether an output is forced to a certain value, is an NP complete problem. On the other hand three-value simulation is purely syntactic. For instance, optimizing an AIG does not need to preserve three-valued simulation semantics.

Because of this syntactic nature of three-valued simulation we define the concept of a grounded stimulus, which does not contain `x` values. A stimulus is an instance of another stimulus if the former is identical to the latter, except where the latter has `x` values. Instead of `x` the former can have any of the three values (`0`, `1`, or `x`).

A witness that a bad state can be reached and also a solution to a structural SAT problem is in both cases just a not necessarily grounded stimulus for which any grounded instance produces through simulation a trace in which the output vector of at least one transition contains `1`. This simulation is of course over two-valued logic. The whole discussion on three-valued logic and simulation is only included here to explain that even though that `x` can be used in a stimulus, we do not actually use three-valued semantics in the definition of a witness. In the context of ATPG, or where reset states are unknown (which would differ from AIG semantics), these definitions using three-valued semantics are still useful.

In order to distinguish successful from failing model checking respectively SAT solver runs we define the concept of solutions. A solution file may contain a result line and an optional stimulus part. The result line either is made up of the single character `0` to denote that the solver proved that the output of the AIG can never become one or `1` which means that the solver produced a witness that at least one output can become one starting from the initial state. Invalid result lines or empty files are interpreted as unknown. If the result is `1` then the rest of the file should contain a witness as defined in the previous paragraph.

The definition of a witness for the existence of a fair cycle is not covered yet.

13 Proposal for Future Extensions

The most important extension of the format is to include word level operators, which would also allow to capture even hierarchical RTL models and more general constraints. One simple way to achieve this is to add new operators through an escape mechanism.

The current proposal simply allows more general gates. These definitions are placed where ANDs are usually defined. To start the definition of a more general operator we use the literal `0` as escape sequence in the ASCII format and the zero byte in the binary format. The escape sequence is followed by a non negative number which denotes the type of the operator and its arguments. The arguments are partitioned into three sequences, the parameters, inputs and outputs. Each list starts with the number of elements.

The parameters are simply unsigned numbers and should not contain literals. The inputs are signed literals, the outputs define new literals.

We suggest to have the following predefined types:

type	operator	arguments
0	quantifier	1 omega 2 lit body 1 out omega=1->universal, omega=0->existential
1	generic circuit	k p1..pk n i1..in m o1..om
2	adder	0 2*n+1 cin x1..xn n+1 y1..yn s1..sn cout
3	cmp	1 p 2*n x1..xn y1..yn 1 o p=0->op=le, p=1->op=lt, p=2->op=ge, p=3->op=gt
4	mux	1 n n+2*n*w (n-bit selector, 2^n w-bit words input) w (one w-bit output) selector word starts with LSB 1st word is selected by the all zero selector
5	cardinality constraint	2 p k n x1 ... xn p=0->op=le, p=1->op=lt, p=2->op=ge, p=3->op=gt x1+..+xn op k
6	all different constraint	1 w n*w (n w-bit words) 1 o

More arithmetic operators will be necessary and the actual type encoding numbers may change as well. Please note that these future extensions are just a proposal and currently not supported at all. However, comments are very much welcome.

14 Related Work

AIGs and similar data structures have been around for a while. AIGs are described in [4,5], RBCs in [1] and BEDs in [2]. Recently AIG based algorithms have been used very successfully as alternative to classical synthesis algorithms in [6] using ideas from [3]. The former paper also contains additional references not listed here.

[1] P. Abdulla, P. Bjesse, and N. Een. Symbolic reachability analysis based on SAT-solvers. In Proc. TACAS'00.

[2] H. Andersen and H. Hulgaard. Boolean expression diagrams. In Proc. LICS'97.

[3] P. Bjesse and A. Boralv. DAG-Aware Circuit Compression For Formal Verification. In Proc. ICCAD'04.

[4] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. IEEE Trans. on CAD, 21(12), 2002.

[5] A. Kuehlmann, M. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. In Proc. DAC'01.

[6] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-Aware AIG Rewriting - A Fresh Look at Combinational Logic Synthesis. In Proc. DAC'06.