AIGER 1.9 And Beyond

Armin Biere[1], Keijo Heljanko[2], and Siert Wieringa[2]

[1] Johannes Kepler University, Austria
[2] Aalto University, Finland

Abstract. This is a short note on the differences between AIGER format version 20071012 and the new versions starting with version 1.9.

To ease the transition, the new 1.9 series of AIGER is intended to be syntactically *downward compatible* with the previous format, but contains already all the new features of the upcoming AIGER version 2.0 format. The future AIGER 2.0 version will not be syntactically downward compatible, because it uses a new binary encoding. However, at least initially, it will not have new features.

For the HWMCC'11 competition we will accept tools that work on the old format or with the new 1.9 series formats. However, for the new track with multiple properties and particularly for the liveness track only the new 1.9 series format is supported.

For the upcoming version 2.0 there will be a new format report. Until it is available, the format report for version 20071012 and this note serve as language definition for the 1.9 series of AIGER.

In essence there are five new semantic features:

- reset logic
- multiple properties
- invariant constraints
- justice properties
- fairness constraints

We discuss all of them in separate sections, including syntactic extensions to the old format. Then the API changes are considered followed by the new witness format.

1 Reset Logic

As AIGER may be used as intermediate language in synthesis, where uninitialized latches occur frequently and should be marked as such, we added support for reset logic. In the new format, a latch is either initialized to 0 (as in the old format, now the default), initialized to 1, or it is uninitialized.

Syntactically, the line in the AIGER format which defines the next state literal of a latch might now optionally contain an additional literal, which is either '0', '1', or the literal of the latch itself. The former are used for constant initialization and the latter to define an uninitialized latch.

Technical Report 11/2, July 2011, FMV Reports Series
Institute for Formal Models and Verification, Johannes Kepler University
Altenbergerstr. 69, 4040 Linz, Austria

Without syntactic changes this allows more general reset logic in the future. For the current benchmarks, we only support either constant initialized or uninitialized latches.

2 Multiple Properties

A common request was to allow using and checking multiple properties for the same model. In practice, a model rarely only has one property and in addition properties like invariants that hold on the model might help to prove other properties faster.

The extension is rather straight forward. At those places, where only one property was allowed, i.e. one output in the old format, multiple properties can now be listed. The major change is in the witness format, which is explained further down.

The other minor change is in the first line of the header. As in the previous format it has the form "`aig` M I L O A", but may now be extended with four more decimal numbers "B C J F", where B denotes the number of "bad state" properties, and C, J, F the number of invariant constraints, justice properties, respectively fairness constraints. Bad state properties are as before negations of simple safety properties resp. invariants. The latter three are explained below.

The idea is, that the new header is an extension of the old header. For the extension it is possible to drop a suffix that only contains zeros. As example consider the following 1-bit counter with one enable input (literal 2). The bad state property is the latch output (literal 4), which is initialized to 0 and has 10 as next state literal. It flips the latch if the input is 1. Otherwise it does not change it. This logic is realized by the XOR implemented with 3 AND gates.

```
aag 5 1 1 0 3 1
2
4 10 0
4
6 5 3
8 4 2
10 9 7
```

As in the old format a "bad state" property b is derived from the negation of a simple **AG** g safety property, with $b \equiv \neg g$ and g denotes "good" states. This benchmark is "satisfiable" if a bad state is reachable.

As described in the report for the old format we have an ASCII and binary syntax of the format. For the examples we use the ASCII version with header `aag`. The difference to the binary format with header `aig` is as before. In the binary format the input section (literal 2) and the first literal of the latch definitions (literal 4) can be dropped. AND gates are binary encoded as before.

In the old format there were no bad state sections, the last 1 in the header line above. Bad state properties had to be listed as outputs. Thus in the old format the example would only have a different first line:

```
aag 5 1 1 1 3
```

3 Invariant Constraints

In the example above we might want to check, whether it is possible to reach a bad state, in which the latch is 1, without ever enabling the input.

```
aag 5 1 1 0 3 1 1
2
4 10 0
4
3
6 5 3
8 4 2
10 9 7
```

This invariant constraint is supposed to hold from the first state until and including where the bad state is found. In linear temporal logic (LTL) a witness for such a bad state is a path satisfying the formula $c\ \mathbf{U}\ (c \wedge b)$, where c is the conjunction of the invariant constraints, and b is one bad state property.

Witnesses for bad state properties are essentially finite paths and can thus be found with safety property checking algorithms without further translations.

Positively, negating the bad state property, we actually try to prove that the LTL formula $\bar{c}\ \mathbf{R}\ (c \rightarrow g)$ holds on all initialized paths, where $g = \bar{b}$ (good) is the negation of the considered bad property. This property can be interpreted as follows: when c stops to hold it releases (at this very moment) g to hold in states where c holds.

The stronger $(\mathbf{G}\ c) \wedge \mathbf{F}\ b$, or positively $(\mathbf{G}\ c) \rightarrow \mathbf{G}\ g$, requires to check that after a bad state has been reached, without violating the invariant constraints, it still is possible to extend the path to an infinite path on which c holds all the time. This might be really complicated, if for instance the constraints restrict latch values. Then extending the finite path to an infinite path requires to solve another PSPACE hard problem.

Even though this stronger version is the standard semantics of combining `INVAR` sections with safety properties in SMV, we prefer the weaker version, which is easier to check and has simple (finite path) semantics.

Note that an infinite path might still be considered as a witness of a bad state property. Actually, only a finite prefix until and including the bad state is sufficient.

4 Liveness Properties and Fairness Constraints

We assume that the reader is familiar with translations of LTL into generalized Büchi automata, similar to what the LTL2SMV tool does. The result of such a translation is a set of fairness constraints, one set of (local) fairness constraints for each LTL property. We call such a set a "justice" property. A witness for a justice property is an infinite initialized path, on which each fairness constraint in the set is satisfied infinitely often.

In AIGER, such a set is represented as a list of literals. In particular, there might be a list of F fairness constraints, as last section, which is an example of one global justice property. The justice section contains J numbers, where the i^{th} number denotes the number of (local) fairness constraints of the i^{th} justice property. After the sizes of all the justice properties, the literals of the first justice property are listed, followed by the literals of the second justice property etc.

As discussed above, fairness constraints do not apply to bad state properties, even though both are listed for the same model. In particular, if there are only bad state properties, but no justice properties, fairness constraints are actually redundant.

More formally, assume we have B bad state literals $b_1, \ldots, b_{\text{B}}$, C environment constraints $c_1, \ldots, c_{\text{C}}$, F (global) fairness constraints $f_1, \ldots, f_{\text{F}}$ and J justice properties $J_1, \ldots, J_{\text{J}}$. The i^{th} justice property is a set of $s_i = |J_i|$ fairness constraints $J_i = \{j_1^i, \ldots, j_{s_i}^i\}$. A witness for the i^{th} bad state property b_i is an initialized path of the model satisfying the LTL formula

$$c \,\mathbf{U}\, (c \wedge b_i) \qquad \text{with} \quad c \equiv \bigwedge_{k=1}^{\text{C}} c_k$$

This path does not necessarily need to be infinite. A witness for the i^{th} justice property is an *infinite* path which satisfies the following LTL formula

$$(\mathbf{G}\, c) \;\wedge\; (\bigwedge_{k=1}^{\text{F}} \mathbf{GF}\, f_k) \;\wedge\; (\bigwedge_{k=1}^{\text{s}_\text{i}} \mathbf{GF}\, j_k^i)$$

Note how the environment constraints are shared among witnesses for all properties, and also required to hold. This applies both to bad state and justice properties. However, for bad state properties they do not need to hold forever. The global fairness constraints in the F section are shared among all justice properties.

5 API

The library API in 'aiger.h' is extended with functions to support new features but the already existing functions do not change their meaning. For the new features the following functions have been added:

```
void aiger_add_reset (aiger *, unsigned lit, unsigned reset);
void aiger_add_bad (aiger *, unsigned lit, const char *);
void aiger_add_constraint (aiger *, unsigned lit, const char *);
void aiger_add_justice (aiger *, unsigned size, unsigned *, const char *);
void aiger_add_fairness (aiger *, unsigned lit, const char *);
```

There is an additional '`reset`' field for latches, as well as separate '`bad`', '`constraints`', '`justice`' and '`fairness`' sections. Each section is a list of '`aiger_symbol`' entries:

```
struct aiger_symbol
{
  unsigned lit;                 /* unused for justice */
  unsigned next, reset;         /* used only for latches */
  unsigned size, * lits;        /* used only for justice */
  char *name;
};
```

6 Witness Format

The witness format has been adapted to the new features as follows. First the output file might contain an arbitrary number of witnesses.

A witness starts with either 0, 1, or 2, where 0 means that the property can not be satisfied, i.e. there is no reachable bad state, respectively an infinite path, satisfying the justice property and all the invariant constraints. The status 2 denotes unknown, while 1 means that a witness has been found.

The second line contains the properties satisfied by the witness. A bad state property is referred to with "`b`i" and a justice property by "`j`i", where i ranges over the bad state respectively justice property indices, which start at 0.

This is the same convention as in the symbol table, which in addition to the valid entries of the old format might now also contain '`b`', '`c`', '`j`' and '`f`' entries.

There might be uninitialized latches. Therefore it is required that the third line contains the initial state, represented as a list of '0' and '1' ASCII characters. The following lines contain input vectors as in the old format.

As all properties and constraints might refer to inputs, i.e. they are "Mealy" outputs, the number of input vectors is the same as the number of states. Thus a witness contains at least one input vector line.

To separate witnesses and catch early termination of model checkers, in the process of printing a full witness, we require that each witness is ended with a '.' character on a separate line. Thus the output format looks like follows

```
(
    { '0' | '1' | '2' }       <newline>          status
  ( { 'b' | 'c' } <index> ) + <newline>          properties
    { '0' | '1' | 'x' } *     <newline>          initial state
  ( { '0' | '1' | 'x' } *     <newline> ) +      input vector(s)
      '.'
) *
```

Of course, the initial state line and the input vectors should only occur for status 1 witnesses.

The 'x' characters denotes a "don't care" value. For the competition we require that grounding to arbitrary values will always produce a valid witness. Since this check is co-NP hard, we will actually only check that grounding them all to zero produces a valid witness.

Comments start with 'c' and extend until and including the end of the line. After filtering them out they are interpreted as new line separators.

For justice properties the state reached after the last input vector has to occur before. It is not mandatory to specify the loop start. It is calculated by the simulator.

A valid witness for the first example is as follows:

```
1
b0
0
1
1
.
```

7 Acknowledgements

Acknowledgements go to all the supporters of AIGER and the HWMCC. The new format report version 2.0 will contain a complete list.