PIF: a Property Interchange Format for verification using HSIS

Berkeley CAD Group

1 July 1994

1 Introduction

This document describes the Property Interchange Format (PIF) for use in HSIS verification (for a detailed description of the HSIS system, refer to [5, 1]). This format is used to describe fairness constraints on the system specification (which has already been read in using the `read_blif_mv` command in HSIS), and automata properties to be proved of this system. For CTL properties we use a separate CTL file.

2 Definitions

In this section we describe a few terms which we will use in detailing the PIF syntax in the following sections.

Definition 1 *Each PIF file is related to a hierarchically defined specification in a BLIF-MV file. This hierarchically defined specification is referred to as the* **system**.

Definition 2 *A* **process** *refers to any instance of a model in the hierarchy of the current system or property automaton. Notice that if the current BLIF-MV file is derived from a Verilog file,* **process** *refers to an instantiation of a module in the hierarchy.*

For a detailed description of the BLIF-MV system hierarchy inside HSIS refer to [2, 4].

Definition 3 *A* **state** *of a process is a unique assignment of values to all of the latches*[1] *for the process. Similarly, a state of the system is a unique assignment of values to all of the latches for all the processes in the hierarchy of the system. A* **state-set** *of a process is any arbitrary set of states of the process. A* **state-cube** *of a process is a state-set of the process which corresponds to an assignment of values to a subset of the latches for the process (the unassigned latches can take on any value). An* **edge-cube** *is an ordered set of two state-cubes* (S_1, S_2) *and it denotes all transitions from any state in* S_1 *to any state in* S_2.

[1] If the description originates from a Verilog source, these latches correspond to the `reg` variables in Verilog.

For each process, the BLIF-MV specification implies a transition structure over the states of the process and the inputs/outputs to/from the process. This transition structure can either be deterministic or non-deterministic. For property automata, the transition structure is currently restricted to be deterministic. In addition, property automata are restricted to be non-hierarchical.

Definition 4 *For the system, a* **string** *is an infinite sequence of outputs; for a property automaton, a string is an infinite sequence of inputs. Given a string, a* **run** *is an infinite sequence of states over all processes such that this sequence follows the string over the transition structure. Since the transition structure may be non-deterministic, each string may correspond to more than one run also. For each run, its* **infinitary set** *is the set of states which are visited infinitely often.*

There is a large class of specifications (systems or properties) which cannot be expressed just by providing a transition structure and allowing all runs on the transition structures to be acceptable [8]. Specifying additional constraints (fairness constraints) on the transition structure alleviates this problem. When properties are expressed as automata, we need additional constraints on the transition structure to increase the expressiveness of the automata. For the system, even though the actual hardware does not have fairness constraints, allowing fairness constraints is helpful when system specifications are simplified using abstractions and scalings.

Definition 5 *A run is* **accepted** *(an accepting run) by the system or a property automaton if it satisfies the fairness constraints on the respective system/automaton. The set of fairness constraints on the processes of the system and property automata are specified using a PIF file. A string* **belongs** *to the system or the automaton if there exists an accepting run on that string. Since a non-deterministic transition structure may lead to multiple runs for the same string, a string that belongs to the system/automaton may also have non-accepting runs in addition to at least one accepting run. The* **language** *of the system/automaton is the set of all strings that belong to the system/automaton.*

3 PIF usage

The fairness constraints and property automata are described in a PIF file. During an HSIS invocation, this PIF file can be read in anytime after the command `vinit` has been executed. The command to read in a PIF file is

`hsis>` **read_pif <file_name>**

In the current release, we do not allow `<file_name>` to contain the character –. Reading in a PIF file using `read_pif` undoes the effect of any previously read in PIF files. However, if `-p` option is used (as below), the existing system fairness constraints are not undone. However, the existing set of property automata is lost.

`hsis>` **read_pif -p <file_name>**

4 PIF syntax

The PIF file is used for two purposes— to describe fairness constraints on the current system hierarchy, and to describe the automata properties one wants to prove of this system. Consequently, the PIF file consists of two parts— the first part to describe fairness constraints on processes in the system hierarchy and the second part to describe automata properties. If properties are to be specified only in CTL, then the PIF file may only contain fairness constraints on the system. CTL with fairness constraints is called "fair CTL", which is strictly more expressive than CTL, as described in the HSIS paper [1].

4.1 Comments

Whenever the PIF parser encounters a `%` character in the PIF file, it ignores it and all other characters from the `%` to the end of the line. Thus the `%` can be used for putting in a comment.

4.2 Fairness constraints on a process

The `.instance` command is used to specify fairness constraints on any instance of a model in the system hierarchy associated with the current BLIF-MV file. The fairness constraints for any instance must be encompassed by the following two statements:

```
.instance <instance_name1> <instance_name2> ...  <instance_namen>
   ⋮
.endinstance
```

In the above declaration, `<instance_name1>` through `<instance_namen>` specify the full-path hierarchical names of the processes, beginning with the root in the current hierarchy[2]. The fairness constraints described between the above two constructs are applied to all the n instance names. For example, the instance name `ping_pong.player_A` refers to child process `player_A` of the root process of the hierarchy called `ping_pong`.

Another way to apply fairness constraints on multiple instances it to use the `.model` construct. These fairness constraints are enclosed by the following two declarations:

```
.model <model_name>
   ⋮
.endmodel
```

[2] For convenience, we allow the root name of the topmost module in the hierarchy to be dropped. Thus the instance `root.mod1.int2` may be referred to as `mod1.int2`.

For this specification, all the fairness constraints described between these declarations are applied to *all* instances of <model_name> in the current system hierarchy. Notice that unlike <instance_name>, the name model_name is only a flat identifier (not a hierarchical identifier).

Fairness constraints for any instance can only be described once. If an instance matches multiple descriptions (either by .instance construct or by .model construct), then only the fairness constraints described by the last match are considered. The previous constraints are ignored.

Two types of fairness constraints— *negative* and *positive*— can be associated with any process [7]. Subsections 4.2.2 and 4.2.2 discuss these. Before that, in the next subsection, we define a way of specifying sets of states for these fairness constraints.

4.2.1 Defining a set of states

The .state construct is used to specify a set of states for the current process. This construct is used before specifying any fairness constraints. Currently this is the only way states can be specified inside the fairness constraints, i.e. any state set referred to in a fairness constraint must use a name declared by the .state construct. The syntax is:

```
.state <st_name> = <st_var_1>:<st_val_1> ...  <st_var_n>:<st_val_n>
```

Here <st_name> is the identifier for the state specified. This is the identifier that will be used to refer to this state in the specification of fairness constraints to follow. <st_var_1> through <st_var_n> are the names of the latches to be assigned values and <st_val_1> through <st_val_n> are the respective values. Not all latches need to be specified for a state-definition; a subset is enough. In this case <st_name> would denote a state-cube.

4.2.2 Fairness constraints

Fairness constraints are used to restrict the language of the system specified in the corresponding BLIF-MV file by making certain runs unacceptable. PIF allows two ways of specifying fairness constraints on the processes: negative and positive fairness constraints. A run on the system is accepted if it satisfies all the fairness constraints.

Negative fairness constraints

Intuitively negative fairness constraints are used to specify that certain runs are *excluded* from the behaviour of the system. Each accepted run must satisfy each of the negative fairness constraints.

```
.negfair
   ⋮
.endfair
```

There are four ways of specifying negative fairness constraints[3]:

1. Node. This identifies a state-cube. A run satisfies this constraint if it visits none of the states in the state-cube infinitely often. The syntax for this specification is:

   ```
   .nodes <st_name>
   ```

2. Edge. This identifies an edge-cube. A run satisfies this constraint if it traverses none of the transitions in the edge-cube infinitely often. The syntax for this specification is:

   ```
   .edges (<st_name_1>, <st_name_2>)
   ```

3. Set. This identifies a set of states as the union of state-cubes. A run satisfies this constraint if its infinitary set is not contained in the specified set. The syntax is:

   ```
   .subsets {<st_name_1>, <st_name_2>, ...  <st_name_n>}
   ```

4. Canonical Fairness Constraint (CFC). This is of the form $F^\infty(S) \wedge G^\infty(T)$, where S and T are sets of states. The condition $F^\infty(S)$ is true of a run if some state in S is visited infinitely often and $G^\infty(T)$ is true if the infinitary set is a subset of T. The mnemonics F^∞ and G^∞ are often used in the literature to denote *GF* and *FG* respectively. The letter F is used to denote *finally* and G is used to denote *globally*. Thus FG would mean *finally globally*, i.e. G^∞; conversely GF would mean *globally finally*, i.e. F^∞. Given a negative CFC, a run satisfies this constraint if the condition $F^\infty(S) \wedge G^\infty(T)$ is not true of the run. The syntax for the specification is

   ```
   .FG {<st_name_F1>, <st_name_F2>, ...  <st_name_Fn>}
   .GF {<st_name_G1>, <st_name_G2>, ...  <st_name_Gm>}
   ```

   In the above, *.FG* denotes G^∞ and the union of the set of state-cubes following this construct denotes T. Similarly *.GF* denotes F^∞ and the union of the set of state-cubes following this denotes S. For a Canonical Fairness Constraint both *.FG* and *.GF* must be specified and in this order.

The user is allowed to use the constants `true` or `false` in place of a state-set above (items 3 and 4), or wherever a set of states is expected in the PIF file; these two constants denote the universe set and the empty set respectively. Also note that multiple constraints of each type can be defined. For example, multiple `.subsets` constraints correspond to a set of *cysets* of Kurshan [6].

[3] For a beginning user, we suggest the use of only negative fairness constraints for the system. Also, many constraints can be specified without using CFC's (the 4th type of negative fairness constraint). However, CFC's are the most expressive kinds of constraints, and an experienced user will probably use just CFC's.

Positive fairness constraints

Positive fairness constraints give positive conditions which must be met by all fair runs. The positive fairness constraints are encompassed by the following two statements:

```
.posfair
   ⋮
.endfair
```

Described below are the four ways of specifying positive fairness constraints. Notice that there is a small difference from the specification for negative fairness constraints. (Here we have `.nodesets` instead of `.nodes`, `.edgesets` instead of `.edges` and the canonical fairness constraint has a different interpretation.)

1. Nodes. This identifies a set of states. A run satisfies this constraint if it visits at least one of the states in this set infinitely often. The syntax is:

   ```
   .nodesets {<st_name_1>, <st_name_2>, ...  <st_name_n>}
   ```

2. Edges. This identifies a set of edges. A run satisfies this constraint if it traverses at least one of the transitions in this set infinitely often. Syntax:

   ```
   .edgesets {(<st_name_1s>, <st_name_1d>), ...  (<st_name_ns>, <st_name_nd>)}
   ```

3. Set. This identifies a set of states. A run satisfies this constraint if its infinitary set is contained inside this set. The syntax is identical to that of the set specification in Section 4.2.2.

4. Canonical Fairness Constraint. This canonical fairness constraint is of the form $F^\infty(S)+G^\infty(T)$, where S and T are sets of states. A run satisfies this constraint if $F^\infty(S)+G^\infty(T)$ is true of the run. The syntax is identical to that for the negative canonical fairness constraint in Section 4.2.2. Note, however, that the interpretation of the specification is different.

Multiple positive canonical fairness constraints can be used to specify fairness constraints of the Streett type [8], which are known to be exponentially more compact than any other type of popular fairness constraint.

4.3 Property description in PIF

Besides specifying fairness constraints in the system, the PIF file is also used to specify the set of automata properties that one intends to prove of the system. HSIS supports two complementary ways of checking properties— language containment (properties are specified as

ω-automata) and model checking (properties are specified as CTL formulae[4]). The properties section of the PIF file is encompassed by the following two declarations:

```
.properties
   ⋮
.endproperties
```

4.3.1 Properties as edge-Rabin automata

In the language containment paradigm, properties are specified as ω-automata. ω-automata are merely transition structures over a finite alphabet along with some acceptance conditions or fairness constraints. These conditions are necessary because it is well-known that certain behaviours cannot be expressed by merely using transition structures (for example, the property that whenever a service is requested the resource is eventually granted).

These automata are used to specify some set of behaviours defining the property. The system satisfies the property if and only if all the behaviours of the system are contained in the set of behaviours of the property (hence, the term language containment). For the property automata these conditions are described as *acceptance conditions*. Recall that for the system, we use fairness constraints to describe these conditions. The main difference is that while an accepted run for the system has to satisfy *all* fairness constraints, an accepted run for the property automaton has to satisfy *at least one* acceptance condition.

In PIF, we allow the user a large set of options to describe the acceptance conditions— all of these can be handled with ease in the HSIS language containment environment [7]. The description of the automaton must be encompassed by the following two declarations:

```
.automaton <property_name>
   ⋮
.endautomaton
```

The <property_name> is the name used to refer to this property in the language containment routines of HSIS. For example, if we want to do language containment for this property, we type lc <property_name> on the hsis> command line.

The transition structure for a property automaton is described using a BLIF-MV file. This BLIF-MV file can be obtained from a Verilog source by using the `vl2mv` translator (see [3, 5]). For this release of HSIS, the hierarchy tree in this BLIF-MV file must be a single-node tree. (In the future, we plan to extend this to allow arbitrary hierarchy trees for properties.) Also note that the property hierarchy has inputs which must eventually be associated with some variables inside the system hierarchy. This is done by treating the property automaton as a subroutine of the system hierarchy using a call-by-name syntax (see below). The property hierarchy has no outputs. The syntax for specifying the BLIF-MV file for the property transition structure is

[4] In the current release, CTL formulae must be specified in a separate CTL file, and not in the PIF file.

```
.blifmv <file_name>
```

where <file_name> is a BLIF-MV file typically generated by compiling a Verilog file using the tool vl2mv.

For this release, the transition structure for the property automaton (specified in the file <file_name>) should be deterministic and complete. If it is not, HSIS will return with an error message.

As discussed above, this BLIF-MV specification has external inputs. The BLIF-MV file only has formal input names. The connection to actual hierarchical system names must be made inside the PIF file using the following syntax:

```
.connections (<formal_1> = <actual_1>, ...  , <formal_n> = <actual_n>)
```

The formal variables <formal_1> through <formal_n> are flat names (corresponding to the external input names declared as inputs inside the property hierarchy). The actual variables <actual_1> through <actual_n> are hierarchical names in the system hierarchy[5]. These can be any variable names in the system; they are not constrained to be inputs/outputs or latch values (states).

Acceptance conditions on a property automaton

The acceptance conditions for the property automaton are defined in a way very similar to how fairness constraints are described for the system[6] (as in Section 4.2). Acceptance conditions on an instance in the property hierarchy should be encompassed by the following two declarations:

```
.instance <instance_name>
   ⋮
.endinstance
```

However, since we expect most hierarchy trees for properties to consist of a single-node (instead of a tree with many nodes), the acceptance conditions for the property need not be encompassed by the above two declarations. If the above two declarations are omitted, it is implied that the acceptance conditions refer to the root instance in the property hierarchy.

State definitions A set of states is defined by the same construct for defining state-cubes, as in Section 4.2.1.

Positive acceptance conditions A run is accepted if it meets any one of the positive acceptance conditions. The positive acceptance conditions are enclosed in the following two

[5] For convenience, we allow the root name of the topmost module in the hierarchy to be dropped. Thus the instance root.mod1.int2 may be referred to as mod1.int2.

[6] Internally, property automata are complemented and become in essence part of the system for which language containment is checked. Thus the positive acceptance conditions for the automata are analogous to the negative fairness constraints on the system. Similarly for the negative acceptance conditions and the positive fairness constraints.

declarations:

```
.posfair
   ⋮
.endfair
```

There are four ways of specifying positive acceptance conditions[7]. The syntax is identical to that of specifying negative fairness constraints in Section 4.2.2. The semantics are given below.

1. Node. A run meets this condition if it visits any one state in the state-cube infinitely often.
2. Edge. A run meets this condition if it traverses any one transition in the edge-cube infinitely often.
3. Set. A run meets this condition if its infinitary set in contained in the set.
4. Canonical Fairness constraint. The sets S and T are specified in exactly the same way as for negative CFC's in Section 4.2.2. However, the interpretation here is different— a run meets this condition if $F^{\infty}(S) \wedge G^{\infty}(T)$ is true of the run.

Negative acceptance conditions A run is accepted if it satisfies any of the negative acceptance conditions (however, each condition is expressed negatively as seen in the semantics below). The negative acceptance conditions are enclosed in the following two declarations:

```
.negfair
   ⋮
.endfair
```

There are four ways of specifying negative acceptance conditions. The syntax is identical to that of specifying positive fairness constraints in Section 4.2.2. The semantics are given below.

1. Nodes. A run meets this condition if it visits none of the states in this set infinitely often.
2. Edges. A run meets this condition if it visits none of the transitions in this set infinitely often.
3. Set. A run meets this condition if its infinitary set is not contained in the set.

[7] For a beginning user, we suggest the use of only positive acceptance conditions for property automata. Also, many conditions can be specified without using CFC's (the 4th type of positive acceptance condition). However, CFC's are the most expressive kinds of constraints, and an experienced user will probably use just CFC's.

4. Canonical Fairness constraint. The sets S and T are specified in exactly the same as for negative CFCs in Section 4.2.2. However, the interpretation of these constraints is different— A run meets this condition if it does not satisfy $F^{\infty}(S) + G^{\infty}(T)$.

4.3.2 Properties as CTL formulae

Properties can also be described using CTL formulae. Though we intend to specify these CTL specified properties in the PIF format, in this release of HSIS, we do not support this. Currently, the way to invoke model checking on CTL formulae is through the HSIS command interface [5]:

.mc <CTL_formula_filename>

where <CTL_formula_filename> is a file containing possibly multiple CTL formulae.

4.4 Including PIF descriptions inside current PIF file

Since each reading of a new PIF file destroys the effects of the previously read in PIF files (Section 3), it might be convenient to store some fairness constraints on some instances or even some properties in a separate file. This file can be included in the current PIF file using the following construct:

.include <file_name>

This construct can be used to include fairness constraints on a set of instances using the `.instance` or `.model` constructs, or to include a set of properties using the `.properties` construct.

5 Example

For an example of a PIF file see the detailed example in [5], which uses a PIF format to describe fairness constraints on the system and property automata for the `ping_pong` example. One PIF file from that documents is file `prop2.pif` which in turn includes file `process-fairness.pif`. These files refer to file `prop2.mv` which has been generated from `prop2.v` using `vl2mv` (see [5]).

```
file prop2.pif
-----------------------------------
.include process-fairness.pif

.properties
# property 2
# Begin instantiation of automaton with name ioh.
```

```
.automaton ioh
#
# include blif-mv file obtained from Verilog description
.blifmv prop2.mv
#
# make connections to the process ping_pong
.connections (player_action = ping_pong.action_A)
#
# begin fairness constraints on this automaton.
# created above. Since this is an automaton, the fairness constraints
# describe the set of all acceptable (good) behaviors. Hence they
# are positive constraints.
.state st1 = state:OKAY
.state st2 = state:PENDING
.posfair
.subsets {st1}
.edges (st2, st1)
.endfair
.endautomaton
# This ends the description of the automaton

.endproperties
-----------------------
```

The above file describes the property automaton named `ioh`. Two positive acceptance conditions have been used to define the behaviour of the automaton. Observe that there may be many multiple ways of specifying the same acceptance conditions. For example, the statement
`.subsets {st1}`
is equivalent to
`.FG {st1}`
`.GF true`
Also, one can use the knowledge of the transition structure of the automaton to describe the same acceptance condition using other types of positive or negative acceptance conditions. The above PIF file includes another PIF sub-file `process-fairness.pif` which describes the fairness constraints on the system:

```
file process-fairness.pif
---------------
# we begin with any fairness constraints that we want to
```

```
# put on the process. Since this refers to a process, the constraints
# are negative since we want to describe behavior that should be excluded
# from the process.
.model player
.state st1 = state:BALL
.negfair
.subsets {st1}
.endfair
.endmodel
---------------
```

References

[1] A. Aziz, F. Balarin, S.-T. Cheng, R. Hojati, T. Kam, S. C. Krishnan, R. K. Ranjan, T. R. Shiple, V. Singhal, S. Tasiran, H.-Y. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. HSIS: A BDD-Based Environment for Formal Verification. In *Proc. of the Design Automation Conf.*, pages 454–459, June 1994.

[2] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. P. Kurshan, S. Malik, A. L. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J. Singh, and H.-Y. Wang. BLIF-MV: An Interchange Format for Design Verification and Synthesis. Technical Report UCB/ERL M91/97, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, November 1991.

[3] S.-T. Cheng, G. York, and R. K. Brayton. VL2MV: A Compiler from Verilog to BLIF-MV, October 1993. Supplied documentation.

[4] Berkeley CAD group. Revisiting BLIF-MV, An Intermediate Format for Verification and Synthesis of Hierarchical Networks of FSMs, 1993. Supplied documentation.

[5] Berkeley CAD group. HSIS, June 1994. Supplied documentation.

[6] Z. Har'El and R. P. Kurshan. Software for Analytical Development of Communication Protocols. *AT&T Technical Journal*, pages 45–59, January 1990.

[7] R. Hojati, V. Singhal, and R. K. Brayton. Edge-Streett/Edge-Rabin Automata Environment for Formal Verification Using Language Containment. Technical Report UCB/ERL M94/12, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, March 1994.

[8] W. Thomas. Automata on Infinite Objects. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 133–191. Elsevier Science, 1990.