EE290H Project, Fall'93

# A User Friendly Environment for Property Specification

by

Rajeev K. Ranjan

Mentors: Tom Shiple, Vigyan Singhal and R. K. Brayton

## Abstract

*We present a user friendly environment for specifying properties. This has been built upon the Berkeley Hierarchical Sequential Interactive System (HSIS). A library of commonly used properties has been constructed. Several command level functions have been created so that the user can easily specify some properties of the system under design. For each property in the library an online help file provides the complete description. The library of properties is user extendible so that a user can easily build a property library pertaining to a particular design and can easily use the interface. Finally, when an incremental change in a design is made, it is important that the new design is verified against all the properties specified for the old design. A feature has been added which allows the user to do this in an easy manner.*

# 1 Motivation

Property specification and validation constitute an important step in formal verification. Current verification systems require that the property specifications must be included inside the system description (except HSIS, which requires a separate property description in a file). Unfortunately the issue of a user friendly property specification has not been addressed in any of the existing verification systems. In this work, a user friendly environment for property specification has been built on the Berkeley verification system, HSIS. This consists of an interface through which a user can interactively specify and verify properties. The interface maintains a database which can be created, modified and maintained by using prompt-level commands. The library of properties on which the database is built is user extendible. The advantage of this environment is two fold. First, novice users can make use of help files provided in the environment to specify properties on the system without worrying about the exact syntax of model checking or language containment (the two paradigms used by HSIS for verification). Second, it gives an experienced user, an easy and fast way for property specifications on a particular system, since the library of properties is user extendible.

In section 2 some introductory material is presented. In section 3 existing techniques for property specification are discussed in detail. The properties which constitute the library are described in section 4. The user interface is discussed in section 5, and finally conclusions are given in section 6.

# 2 Introduction

With the increasing use of computers and digital systems in every walk of life, their correctness has become an important concern. Traditionally, simulation has been used to verify the correctness of systems. This involves simulating the system over combinations and sequences of inputs. However, the increased complexity of digital hardware has made exhaustive simulation computationally infeasible. Typically only a subset of input patterns is applied to the hardware system and the correctness of the system is deduced from the results. This does not guarantee the complete correctness of the system, and often bugs are found late in the design cycle. It is important to detect the errors at an early stage of design. Bugs detected at the late stage lead to a delay in marketing the product. In some cases the bugs are detected after the product has been marketed leading to product recall. This is a highly costly proposition, and hence the verification of the system should be complete, and preferably done during the design stage.

Recently, formal verification has provided an alternative to simulation for determining hardware correctness. It involves the use of analytical techniques to prove that the implementation of a system conforms to a set of properties. A typical verification problem consists of formally establishing a relationship between an implementation and a specification [2]. *Implementation* refers to the hardware design to be verified and the term *specification* refers to the set of properties with respect to which correctness is to be determined. Since the reasoning has to be formal, it is required that all three entities - implementation, specification, and the relationship between them must be expressed

formally.
**Implementation:** An implementation consists of a description of the actual hardware design and its environment that is to be verified. Since the current work has been built on HSIS, it is assumed that the implementation is described by a network of interacting finite state machines.
**Specification:** A specification is a description of the desired behavior of a hardware design.
Since we are mainly concerned with the specification in this work, we discuss it in more detail. The formalisms used to describe specifications can be broadly categorized in two groups:

1. Logic: This consists of first-order predicate logic, propositional logic, higher order logic, temporal logic etc. In this framework typically *model checking* is used to verify the relationship between implementation and specification.
2. Automata, language theory: In this approach, specifications are represented as automata, languages, machines, trace structures, etc. Correspondingly, the relationship between implementation and specification is formalized using machine equivalence, language containment and trace conformation.

For any design, the complete specification of properties which the design should satisfy is of great importance. If the property list is not complete and the design is not verified for all the properties, it might result in undesirable behavior of the system.
Broadly speaking, properties can be divided into three categories [2]:

1. Functional correctness properties: This pertains to the behavior of the particular design. Typically the designer knows what the system is supposed to do if implemented correctly and he/she can specify those desirable behaviors in terms of properties.
2. Implementation properties: This includes generic properties, e.g., safety and liveness.
3. Timing properties: Some event must take place within a specified time after some other event has taken place.

In this work an attempt is made to cover as many generic properties as possible. Since many functional properties are also common to most designs, we have tried to cover some of those as well. At this point, the verification system HSIS does not support verification with timing constraints, and hence properties of the third kind have not been implemented in this work.
It has been observed that some properties can be formally described only by an automaton but not by a CTL formula. Yet, there exist instances where the reverse holds. In some cases, one formalism may be more suitable for specifying properties than an other. HSIS supports property specification in terms of CTL-formulae as well as automata. In this work wherever possible, a property is described by a CTL-formula as well as an automaton.

# 3 Existing Techniques for Property Specification

The verification systems which are available in public domain include - Mur$\phi$ and SMV. A well known industrial verification system is *COSPAN*.

*Mur$\phi$* supports only invariance type property specifications. *SMV* uses CTL formulae to specify properties; verification is done using model checking. Since there are some properties which cannot be expressed by CTL formulae, *SMV* lacks completeness in expressivity. Moreover since the specifications are part of the system description, an incremental change in the specification requires that the whole verification process should be repeated. Moreover since *SMV* does not work in an interactive mode, it is not possible to interactively specify and verify properties. *COSPAN* which is based on the selection resolution(S/R) model of the system, supports property specification in terms of automata. *COSPAN* also supports a library consisting of a set of generic properties. But it is not possible to interactively add or remove or modify a set of properties in *COSPAN* since property specification is a part of the system description and any change in the property requires that the file containing the system description be changed and recompiled.
These verification systems are limited in their application in the sense that they support specification in terms of either CTL-formulae or automata. HSIS [1], which is based on combinational-sequential(C/S) model supports property verification both in terms of language containment as well as model checking. It supports both of these methods allowing in addition fairly general fairness constraints to be specified for the implementation.

Next we discuss the current way of specifying and verifying properties in HSIS.

### 3.1 Current Property Specification in HSIS

After the system description has been read in, the user needs to specify the fairness constraints on the system. These fairness constraints are needed every time a new property needs to be verified and is read in. The property can be specified by either using an automaton or a CTL formula. In the first case the automaton is created by writing a Verilog description of the automaton. This is then translated into BLIF-MV using *"vl2mv"*. A "pif" file then describes the automaton and includes the generated "mv" file in it. It also contains the acceptance conditions of the automaton. This "pif" file is read in using the "read_pif" command and then language containment is performed. Sometimes a simple syntactic mistake at the level of property description in Verilog only gets detected while doing language containment and this then involves correcting the Verilog code, generating the BLIF-MV description, possibly changing the pif file, re-reading the pif file and then doing the language containment again. In the case when *model checking* is used, a CTL formula is specified in a separate file which is used during verification.
In order to make property specification more user friendly, an environment has been built on HSIS. This environment consists of an interface through which a user can interactively specify and verify properties. This interface maintains a database of properties and supports several prompt-level commands to create, maintain, and modify the database. In particular the salient features of this environment are following:

1. A library containing a set of frequently used properties.

2. The ability to describe a standard property at the prompt level.

3. The ability to extend the library. Therefore an experienced user can describe specific properties for his/her design system and can refer to them easily.

4. Complete help files for all the properties in the library. Therefore a novice user can find out about a particular property in detail and can use that information to specify properties on system.

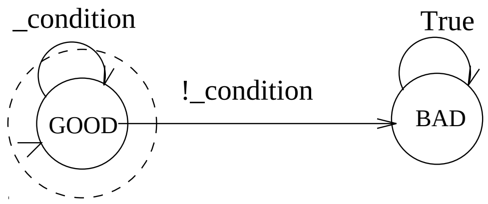
## 4 The library of properties

Below we describe the set of properties which constitute the library and are supported by this package. The references for some of the properties are given beside the property name. Cycle sets for a property automaton are shown by enclosing the associated state(s) by dotted circle(s). The recur edges are shown by zigzag lines.

1. INVARIANT: `invariant( _condition )`
   This property checks that `_condition` is always true. This can be used to specify a condition which should be true in all states, i.e., *"Nothing bad ever happens"*.
   For model checking, this property gets mapped to the formula:

   $$AG(\_condition)$$

   For language containment, the automaton is given as:

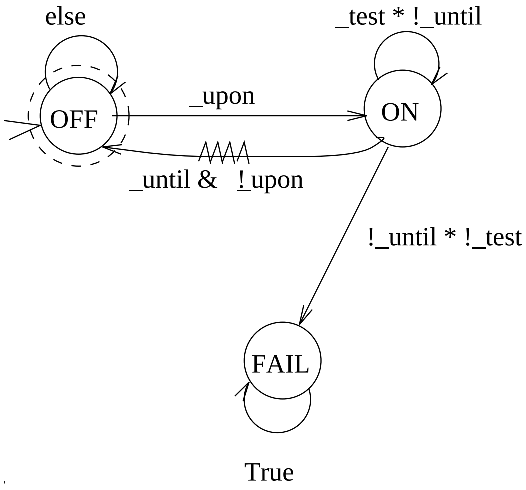



   This can be used for partial correctness (no wrong answers are produced), mutual exclusion (no two processors are in a critical section simultaneously), deadlock freedom (no deadlock state is reached), global invariants (no violation of the invariants takes place).

2. STOP_UPON [3]: `stop_upon(_upon ,_test, _until)`
   This property specifies that in any state of the system if `_upon` is true, then `_test` must be true until `_until` is true. The CTL-formula is given as:

   $$AG(\_upon \Rightarrow A(\_test \;\; U \;\; \_until))$$

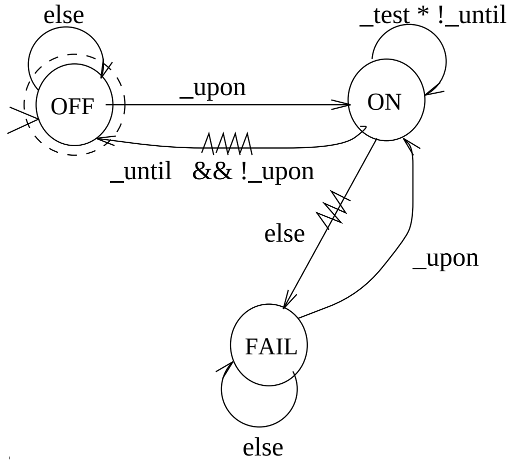
The corresponding automaton is given below:




3. EV_TESTUPON [3]: `ev_testupon(_upon, _test, _until)`
   This property specifies that infinitely often whenever a state is reached where `_upon` is true, the `_test` will remain true until `_until` becomes true. The CTL-formula is given as:

$$AF(AG(\_upon \Rightarrow A(\_test \;\; U \;\; \_until)))$$

   The corresponding automaton is shown below:

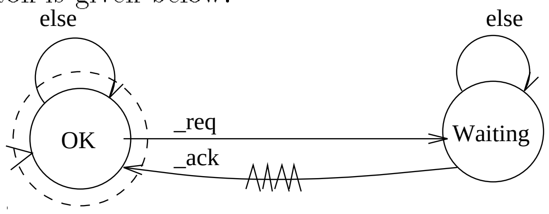



   This property could be used to specify the fact that there should be only a finite number of failures in the system, i.e., eventually must reach a state where the system behaves correctly.

4. LIVENESS: `liveness(_req, _ack)`
   This property describes that a *request* is eventually followed by an *acknowledgement*, e.g., it asserts that *eventually something good happens*. The CTL-formula is given as:

$$AG(\_req \Rightarrow AF(\_ack))$$

The corresponding automaton is given below:

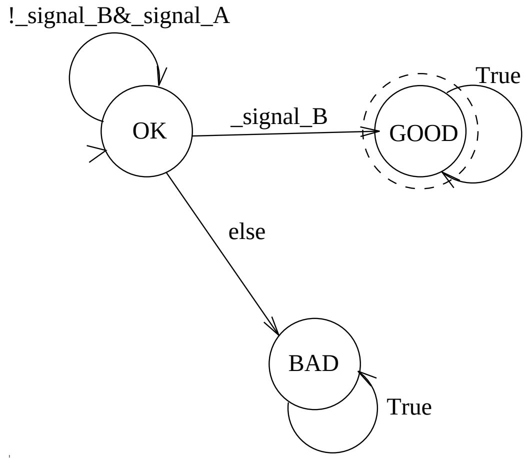



This can be used to express total correctness (termination eventually occurs with correct answers), accessibility (eventually a requesting process will enter its critical section), starvation freedom (eventually service will be granted to a waiting process).

5. PRECEDENCE: `precedence(_signal_A, _signal_B)`
   This describes the precedence order of events, i.e., `_signal_A` will hold until `_signal_B`. The CTL-formula is given as:

$$A(\_signal\_A \;\; U \;\; \_signal\_B)$$

   The corresponding automaton is given as:

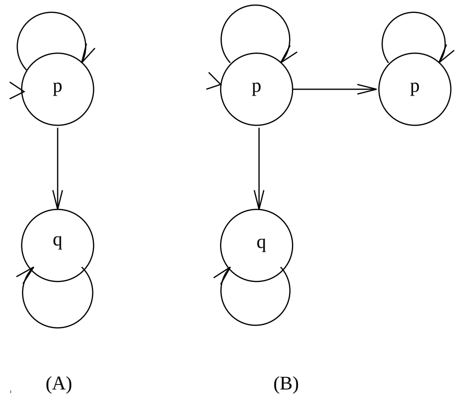



   This property can be used to describe safe liveness (nothing bad happens until something good happens).

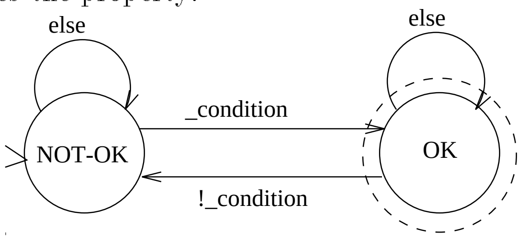
6. ALWAYS_POSSIBLE: `always_possible(_condition)`
   This property detects absence of deadlocks, implying that from each state in the system we can reach another state which satisfies the condition. The CTL-formula is given as:

$$AGEF(\_condition)$$

   This property cannot be expressed by an automaton, since this CTL-formula distinguishes structure as opposed to an automaton which distinguishes language. For example, following two structures have the same language ($p^{\omega} + p^{+}q^{\omega}$), but structure (A) satisfies the CTL-formula $AGEF(q)$ while structure (B) does not [4].



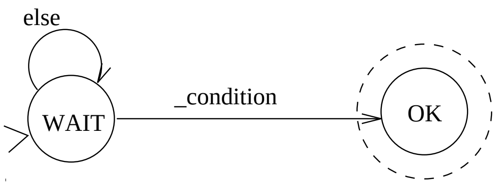

7. ALMOST_ALWAYS: `almost_always(_condition)`
   This property expresses that `_condition` should hold everywhere after a finite number of transitions in the system. In other words, we are allowing only a finite number of failures (`!_condition`) in the system. This is a case where a property cannot be described by a CTL-formula ($FG(p)$ is not a CTL-formula), but can be specified by an automaton. The following automaton describes the property:




8. FINALLY: finally`(_condition)`
   This specifies that eventually `_condition` is true. The CTL-formula is:

$$AF(\_condition)$$

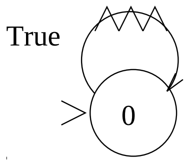
   The automaton is given as:




9. TRAVERSE: `traverse`
   Forces a complete traversal of the system state transition graph. The CTL-formula is:

$$AG(true)$$

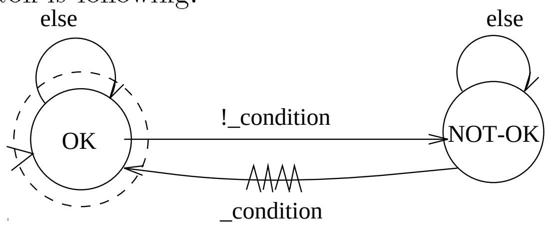
   and the automaton is:



10. INFINITELY_OFTEN: `infinitely_often(_condition)`
    This property expresses that from any state it should be possible to reach a state where `_condition` is true. The CTL-formula is:

    $$AGAF(\_condition)$$

    The corresponding automaton is following:

    This property can be used to ensure that from any state in any state in the system it should be possible to reach the "reset" state.

11. POSSIBLE: `possible (_condition)`
    This property specifies that starting from the initial state it is possible to reach a state where _condition is true. The CTL-formula is:

    $$EF(\_condition)$$

    This property cannot be expressed in terms of an automaton where we can perform standard language containment.

# 5 User Interface

Detailed descriptions of the user interface commands are given below. [1]

1. Specifying a property of the system: This can be done easily if the user is aware of the library property name to be verified and the arguments needed. The format is as follows:

```
hsis> prop_create [-l/m] -i<instance-name> -n<property-name>(<arg1>, <arg2>, ...)
```

   This creates a property of the specified type and stores it in the database. The user has the option of using either language containment or model checking for property verification. This can be specified by "-l" and "-m" respectively (by default it does model checking). Next the user gives the instance name of the property which can be used later to refer to this particular property. For instance if the model checking option is used, then a file called "instance-name.ctl" is created. If the "-l" option is used, then it creates a "pif" file called "instance-name.pif". The "property-name" refers to one of the properties stored in the library.

[1] All the commands described below have been prepended with an underscore in the implementation. For example, prop_list has been changed to _prop_list etc.

The arguments list is given as the proper sequence of signals of the system to be referenced by the property. The "help" mode described later can help the user find out the proper sequence of arguments for a particular property. Note that the property name and the associated argument list come at the end of the "prop_create" command. Hence anything following the property name will be treated as one of the arguments. The arguments should be comma separated.
It is an error to specify two property instances with the same instance name. An error message will be printed if this is tried.

2. Once some properties are specified and stored in the property data base, the user can verify any of these properties in the following way:
`hsis> prop_verify <instance_name_1> [<instance_name_2> .....]`

3. This package also supports a succinct way to verify some property on the system. The format is the following:
`hsis> prop_lc[prop_mc] [-i <instance_name>] -n <property_name> <arg_list>`

   If the "-i" option is used, "`prop_lc`" command is equivalent to "`prop_create`" followed by "`prop_verify instance_name`". Similarly in case of "`prop_mc`", model checking is carried out immediately. Note that if the "-i" option is not used the property instance is not created and hence cannot be referred to later on. In other words without "-i" option "`prop_lc`" or "`prop_mc`" performs the language containment or model checking but leaves the property database unchanged.

4. The user can specify any number of properties. For each property the corresponding "pif" or "ctl" file is created, depending upon the option. Later, the user can verify any of these properties in any sequence by referring to them by the instance name. The user can also get the list of the properties specified so far by typing:
`hsis> prop_list [-l]`
Without the "-l" option a short list of property instance names along with the property names get printed. Also printed is the option chosen for that particular instance (language containment or model checking). When the "-l" option is used, the list of the arguments attached to each property instance is also printed.

5. In the early design stage, often one or more properties fail when verified. In that case the user tries to modify the design so as to make the system compatible with the properties. In order to make sure that the modified design satisfies each property (including the ones satisfied by the earlier design) the user needs to re-check the new design against all the properties again. This package provides an easy way to perform this task. While performing verification on the first design if the properties were instantiated using either of "`prop_verify`" or "`prop_lc`" or "`prop_mc`" (with "-i" option) command then these instances are available for further use. The user just needs to read in the new design and type "`prop_all`". This will try to verify the new design against all the properties instantiated for the old design.

6. Sometimes the user needs to remove an instantiated property from the data base. This can arise because of a change in design, a change in signal names, wrong property specification etc. This can simply be done by using the command "`'prop_delete`" followed by a list of property instance names to be removed, as following:

`hsis> prop_delete [-f] prop_instance_1 prop_instance_2 .... prop_instance_n`
To remove the files associated with these property instances `-f` option should be used.
In case the user wants to get rid of all the property instances then "`prop_end`" (described later) can be used.

7. An important feature of the library is that it is user extendible. The user can create new property templates which pertain to a particular design and can use those templates to describe properties on different signals. This saves the user from writing different files for the same property pertaining to different signals. To make a new property "new_property" the user has to do the following:

   (a) If the property can be described by an automaton then the following steps need to be carried out:

      i. Write the Verilog description of the automaton. This needs to be translated into "blif-mv" and the file should be named "new_property.prop.mv".

      ii. Create a template file called "new_property.pif_template". This template file is used to generate the "pif" file corresponding to the instantiation of the property. The template file for the "invariant" property is shown below for illustration.

```
.properties
.automaton $0
# Instance Name = $0
# Property Name = $1
# include blif-mv file obtained from verilog description
.blifmv $1.prop.mv
# make connections to the process
.connections (_condition = $2)
# fairness constraints on automaton
.state st1 = state:GOOD
.posfair
.subsets {st1}
.endfair
.endautomaton
.endproperties
```

      The tokens in the template which need to be replaced by the real names of signals, are represented by character $ followed by a number. Hence there should not be any unnecessary presence of the character $. Note that there are three tokens present in the above template file. "$0" and "$1" are exclusively used for the instance name and property name respectively. Token of type '$2" onwards are used for specifying signals. Also the blif-mv file which is the translation of the Verilog description of the property automaton is included in the template file. Both the template file as well as the blif-mv file corresponding to the property should be present in the current directory or in one of the directories specified by "open_path" in ".hsisrc".

   (b) If the property can be specified by a CTL formula then create a template file, "new_property.ctl_template" which will be used to generate the file "instance-name.ctl".

The template file for the "invariant" property is shown below:

```
#This is the instance $0 of $1 property
AG($2)
```

8. To help the user make full use of this easy way of property specification, a complete set of help files are provided. To find out the type of properties available in the library, the user can type "`prop_help`". It gives the names of the properties available in the library as well as the type of the property (model checking or language containment). To find out in detail about a particular property, the user needs to type
`hsis> prop_help -n <property_name>`.

9. The user can destroy the property database by using the command "`prop_end`". In order to remove the files associted with the property instances "`-f`" option should be used.

| | |
|---|---|
| prop_create | Instantiates a property of the specified type |
| prop_verify | Verifies an array of properties already created |
| prop_lc[mc] | Specifies and verifies some property, using language containment [model checking] |
| prop_list | Lists all the properties instantiated in the database. |
| prop_all | Verifies design against all the properties in the database. |
| prop_delete | Deletes an array of properties from the database. |
| prop_help | Provides help for properties. |
| prop_read | Reads in the property instances from the current directory. |
| prop_end | Destroys the property data base. |

Table 1: List of Commands

These prompt level commands are summarized in Table [1].

## 6 Conclusion

Property specification plays an important role in formal verification. A user friendly interface for property specification has been built. This interface allows the user to interactively specify and verify properties on a system. To aid the user in specifying properties, help files are provided. The property library is user extendible. Hence specifying properties specific to a particular design becomes less tedious.

# 7 Possible Extensions:

There are a few extensions to the current work which can enhance the capability of property interface.

1. Currently invocation of "read_pif" replaces the properties read by previous "read_pif". This leads to some repetition of work in case a property is verified more than once in laguage containment paradigm while a "read_pif" takes place in the middle. A feature allowing the user to read in several "pif" files one after another without overwriting the existing properties can be quite useful.
2. Often we need to change the fairness constraints on the system before verifying some properties. It should be possible to override the old system fairness constraints with a new one without touching the property automata already specified.
3. Right now a task automaton is treated same as a system automaton. Hence if a task needs to evaluate the state of a system, it(the state) must be made an output of the system. The first extension relates to allowing the task automaton to look inside the system automaton.
4. Many a times the task automaton needs to evaluate a boolean expression involving some variables of the system automaton. In order that the property automaton is independent of the type of the system variables, the boolean expression must be evaluated and then be passed on to the automaton. The second extension relates to this aspect of property verification. Note that this extension would allow an arbitrary boolean expression to be passed as an argument to the property automaton. For uniformity sake, the semantics of boolean expression would be same as that in a CTL-formula.
5. Currently, the boolean operators allowed in the CTL-formula are restricted to "equality", "and", "or", "not" and "exor". Often we need to express boolean expressions involving operators like "geq", "leq" etc. The current model checking environment can be extended to include these operators.
6. In model checking if more than one formulas are to be verified, then, verifying them separately takes more time than it takes to verify them together. This is because in the latter case the reachability analysis needs to be done just once. A similar approach could possibly be adopted in the property verification using language containment. The idea is to make use of the information available while computing the reachable set of states for a particular property automaton. This information can be helpful in computing the reachable set of states for other property automaton.
7. Sometimes designers insert dummy signals in the system description which express a boolean relation amongst some variables in the system. The purpose of inserting these variables is to be able to watch the value of the boolean expression by observing the dummy variable. Since these dummy variables are temporary variables, they are smoothed out while forming the transition relation of the system. And hence they cannot be observed at the time of verification. One way to solve this problem is to make a dummy sub-circuit call using the dummy variable as output. This makes sure that dummy variable is not smoothed out.

8. The library of properties needs to be extended to cover a wider range of properties.

## 8 Acknowledgement

I would like to thank the EE290h instructors team for doing such a good job in the course, which motivated me to do this project. I would also like to thank Tom Shiple, Vigyan Singhal and Prof. R.K. Brayton for their helpful suggestions and for reviewing the draft.

I also gratefully acknowledge the support provided by Motorola under UPR agreement.

## References

[1] A. Aziz et. al. HSIS : A BDD-based Environment for Formal Verification. In *Proc. Design Automat. Conf.*, June 1994. submitted.

[2] Aarti Gupta. Formal Hardware Verification Methods: A Survey. In *Formal Methods in System Design*, pages 151–238. Kluwer Academeic Publishers, New York, 1992.

[3] Z. Har'El and R. P. Kurshan. COSPAN User's Guide. 1987.

[4] Serdar Tasiran. Private communication, 1993.