Empirical Design Bugs Prediction for Verification

Qi Guo*†, Tianshi Chen*‡, Haihua Shen*‡, Yunji Chen*‡, Yue Wu*† and Weiwu Hu*‡
*Key Laboratory of Computer System and Architecture,
Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
†Graduate University of Chinese Academy of Sciences(GUCAS), Beijing 100049, China
‡Loongson Technologies Corporation Limited, Beijing 100190, China

***Abstract*—Coverage model is the main technique to evaluate the thoroughness of dynamic verification of a Design-under-Verification (DUV). However, rather than achieving a high coverage, the essential purpose of verification is to expose as many bugs as possible. In this paper, we propose a novel verification methodology that leverages the early bug prediction of a DUV to guide and assess related verification process. To be specific, this methodology utilizes predictive models built upon artificial neural networks (ANNs), which is capable of modeling the relationship between the high-level attributes of a design and its associated bug information. To evaluate the performance of constructed predictive model, we conduct experiments on some open source projects. Moreover, we demonstrate the usability and effectiveness of our proposed methodology via elaborating experiences from our industrial practices. Finally, discussions on the application of our methodology are presented.**

***Index Terms*—Verification; Complexity Metric; Bug Prediction; Empirical Study**

I. Introduction

As hardware designs continue to grow in complexity and time-to-market pressure intensifies, verification has become the key bottleneck of state-of-the-art design development cycle. Although dynamic simulation-based verification is still the main verification technique to guarantee the success of first tape-out of practical industrial designs, the main drawback is the difficulty of assessing the verification progress. More precisely, it is difficult to determine whether the verification has been sufficient to find out all potential bugs, owing to the complexity of the industrial design. As a compromise, the problems of evaluating the quality and determining the completeness of verification are addressed by employing coverage models that measure the state space of a DUV has been touched during simulation.

Generally, coverage models can be classified into two categories: *structural coverage* and *functional coverage*. Actually, both coverage models are widely accepted alternatives to provide feedback mechanism for evaluating the quality of verification. However, structural coverage is only an indicator that partially quantify the progress of verification, and it is hard to be used to guide the verification process. Another category of coverage models, functional coverage is somewhat casual without effective metrics to measure its quality. Meanwhile, the intrinsic goal of verification is to reveal all potential design bugs instead of meeting all coverage requirements, which implies that even all coverage goals are satisfied, one still cannot guarantee that the design is bug-free since the coverage space represents only a subset of testing space [5]. Hence, we seek for a complementary method of current coverage models to evaluate and guide the verification process more effectively and reliably.

This work is partially supported by the National S&T Major Project (under Grant No. 2009ZX01028-002-003, 2009ZX01029-001-003 and 2010ZX01036-001-002), the National 863 Program of China (under Grant No. 2005CB321600), the National 973 Program of China (under Grant No. 2008AA010901 and 2009AA01Z125), and the National Natural Science Foundation of China (under Grant No. 6073601260921002, 60803029 and 61003064).

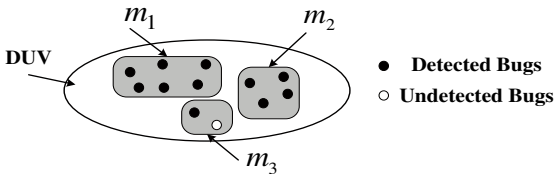

Fig. 1. A practical example to illustrate how to leverage early bug prediction to assist verification.

Since detecting all bugs is the essential purpose of verification, in this paper, we propose a novel verification methodology that incorporates *bug information*, which indicates the distribution and density of bugs in a design, to guide and assess the verification flow. The basic idea is that if we can precisely estimate the distribution and density of bugs for a DUV in the early stage, then the predicted bug information can be adopted to assist the verification process. An example about our methodology is illustrated in Fig. 1, where we consider a DUV consisting of 3 modules as m_1, m_2, and m_3, and distribution of bugs varies significantly among different modules. During the entire life cycle of verification, we describe the usability of early bug prediction as following:

1) During the arrangement of verification plan, the one with most bugs (*i.e.*, m_1) should obtain the largest amount of verification resources and time efforts.
2) As verification proceeds, once most of the bugs have been detected in m_1, verification resources should be steered to those modules with unexposed bugs (*i.e.*, m_3). In other words, the verification process could be controlled quantitatively and precisely according to predicted results.
3) Once there is no difference between the number of exposed bugs and the predicted value, we can conclude that the verification is complete, under the condition that the predictor is "perfect".

Hence, this paper is dedicated to introduce a novel verification methodology from bug-centric perspective and offers an alternative perspective to the traditional coverage-centric view.

Obviously, the decisive issue is how to construct a highly accurate bug predictor for a design at the early stage of verification. To construct such a predictor, we employ machine learning techniques to model the relationship between the static metrics of high-level designs and the corresponding bug information that are mined from bug repository. It stems from the assumption that the complexity of a design can be estimated by some crucial high-level attributes [13], [11], [1], while more complex a design is, more bug-prone it

978-3-9810801-7-9/DATE11/©2011 EDAA

should be. To validate this assumption, we perform preliminary experiments on several open source projects from different fields. And the experimental results show that complexity metrics indeed highly correlate with bug occurrence.

A potential concern about our approach is how to collect enough data to train highly accurate predictors. However, in industrial practices, such a concern can be addressed by utilizing the bug reports of previous releases (or early life cycle of the current release) of the design. Actually, most successful chips are series of productions, which manage to occupy the market for a long time. As significant examples, the Power series of IBM and the X86 series of Intel have even tens of releases in their decades of history. Thus, there are often plenty of *prior knowledge* to train accurate predictors. To provide empirical evidence for the above view, we carry out experiments on two releases of Godson-2 microprocessor [8]: We utilized the bug information gathered from Release 1 to construct predictors for Release 2, and the promising experimental results on Release 2 demonstrate that our methodology is practical to assist the verification of future designs via cross-releases prediction.

The main contributions of this paper are: (1) To our best knowledge, it is the first time that a quantitative methodology is proposed to employ bug prediction built from previous bug repositories to provide feedback to verification plan and execution; (2) We introduce our industrial experiences from employing the proposed methodology to assist the verification process on a series of microprocessors.

This paper proceeds as follows. Section II presents previous related work. Section III introduces the framework of our methodology. Section IV shows the research approach we followed. Section V studies the effectiveness of proposed approach on some open source projects. Section VI describes how to employ constructed predictors from previous release to assist the verification of next release in industrial practice. Section VII discusses some issues on bug prediction for verification. Finally, section VIII concludes this paper.

II. Related Work

A. Complexity Metrics and Bug Analysis

An important issue associated with our work is to measure the design complexity of high-level hardware designs. Unfortunately, there are few studies that systematically investigate the complexity metrics of hardware designs. At first, inspired by the structural and process similarities between VHDL description and software language, measures of syntactic complexity have been proposed in [13]. Then, to evaluate the design quality, Protheroe *et al.* [11] have provided some metrics on register transfer level (RTL) of a design. Besides, Bazeghi *et al.* have studied some metrics of HDL code and results provided by Design Compiler to estimate design effort of microprocessors [1]. And Bentley has introduced the sources for bugs from the verification practices of Intel processor [2]. Even Guo *et al.* have investigated the relationship between the complexity metrics and bug occurrence [7], none of these works have discussed how to use predicted bug information to facilitate verification.

B. Fault Prediction in Software

In the field of software engineering, many investigations have been dedicated to characterize the relationship between code metrics and fault-proneness of software to assess the design quality [6], [3]. Most of those studies have focused on quantitatively characterizing the impact of static metrics on bug occurrence. However, only a few of them have worked on the validation of constructed predictor to other releases [10], or other projects [15]. According to these literatures, due to complicated contributors to bug occurrence, constructing a general bug predictor for different softwares is still a challenge and open issue. Nevertheless, the successful application of fault prediction in software implies that there are some underlying relationships between the complexity and bug-proneness. Due to the similarities between HDL and software languages, it is intuitive to investigate the impact of complexity on bug occurrence.

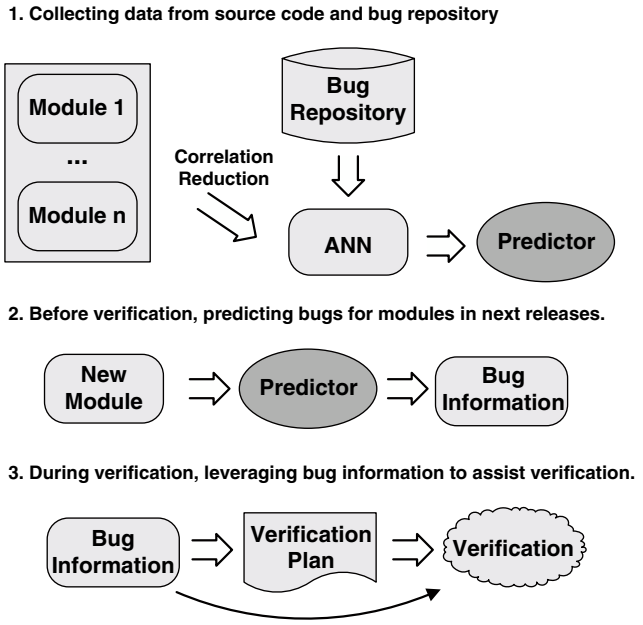

Fig. 2. Framework of our methodology that leverages early bug prediction to assist verification.

III. Framework

In this section, we elaborate the framework of our methodology that leverages early bug prediction to assist verification, which are illustrated in Fig. 2. From this figure, we can detail our methodology into three steps: 1) we collect training data from the source codes (where high-level attributes concerned with complexity are extracted), and the bug repository. Before sending the extracted candidate attributes into Artificial Neural Networks (ANNs), Principal Component Analysis (PCA) [12] is employed to reduce the correlations among the attributes, which is crucial to the accuracy of the resultant predictor. After that, we use ANNs to model the relationship between crucial attributes and corresponding bug-proneness. 2) Once we obtain the predictor on bugs, we can employ it to predict bugs for new modules in the next releases before verification. Technically, we employ the predictor to estimate the distribution and density of bugs for each new module in interested releases to obtain predicted bug information. 3) During the arrangement of verification plan, we can quantitatively allocate resources and time efforts according to predicted bug information on each new module. Moreover, the verification process can also be guided and assessed.

It is notable that the constructed predictor is employed to predict bug information before the arrangement of verification plan, which is referred to "early bug prediction". In other words, it is in contrast to predicting bug information *during* the verification, when the number of undetected bugs of a module obviously decreases as verification proceeds. In fact,

TABLE I
ATTRIBUTES IN MODULE LEVEL

Metrics	Definition
LOC	Lines of Code
NODP	No. of Decision Points
NOP	No. of Ports
NOIP	No. of Input Ports
NOOP	No. of Output Ports
NOTO	No. of Total Operators
NOUO	No. of Unary Operators
NOBO	No. of Binary Operators
NOSD	No. of Signal Definitions
NOIM	No. of Instantiated Modules
NODM	No. of unique Instantiated Modules
NOII	No. of Inputs to Instantiated Modules
NOOI	No. of Outputs from Instantiated Modules
TOC	Times of Clocks are used
NOC	No. of unique Clocks
NOS	No. of Statements
NOCS	No. of Control Statements
NOPB	No. of Process Blocks

to predict bug information during verification, we have to implement a dynamic bug prediction mechanism, which is unnecessary for assisting verification. The reason is that once we can obtain the number of potential bugs in a module before the verification, it is adequate to guide and assess the overall process of verification with possessed bug information, which will be elaborated later.

IV. MODEL CONSTRUCTION

In this section, we present the approach followed to construct the bug predictor. Firstly, we introduce high-level attributes of a design, which implies the design complexity. Then, we utilize PCA to reduce the correlation among the original attributes. Finally, two categories of ANNs models are shown to construct predictive models for bug distribution and density, respectively.

A. High-level Attributes of a Design

The key idea of our approach is to empirically explore the relationship between high-level attributes of a design and its bug-proneness. Therefore, we should specify related attributes at first. As stated, design complexity may be closely related to the bug occurrence of a design. Thus, attributes that determine the design complexity are listed in Table I, where interpretation of each metric is explained in detail. In order to make complexity attributes of a design independent of specific HDL language, we choose attributes from the following aspects: lines of codes, number of decision points (*i.e.*, branch conditions), port related, operator related, signal related, instantiated module related, clock related, and statement related.

B. Correlation Reduction of Attributes

The attributes specified in last section may contain "noisy data", and complicated dependencies among variables, *i.e.*, some attributes are highly correlated with each other. For instance, NOP is the sum of NOIP and NOOP. To identify and filter the correlation among attributes, statistical method (*e.g.*, PCA) is performed on the original attribute set. Thus, we can attain a reduced set best representing the original attributes at hand while with reduced correlation among attributes.

PCA captures the majority of the variations as best as possible via a few unrelated new variables which are linear combinations of the original attributes, called *principal components* (PCs). Intuitively, to form the reconstructed attribute set, it is possible to extract the same number of PCs as the number of the original variables. However, our goal is to extract as few PCs as possible while preserving most of the original information. The retained information of extracted PCs can be represented by cumulative variance. For instance, if the cumulative variance of q PCs contributes to 90% of the total variance, q PCs can be employed to construct a new data set with no more than 10% loss of information. Here we select the PCs only whose eigenvalue is larger than 1.0.

C. ANNs Model

In this section, we elaborate the application of ANNs in modeling the relationship between aforementioned PCs and bug information. First, we briefly introduce the technique of ANNs. Then, in order to estimate the bug information in different aspects, we construct *classification* and *regression* model via ANNs, respectively.

1) Overview of ANNs: Machine learning techniques have recently been utilized by a number of researchers in the field of design verification, and most of the investigations focused on CDG (Coverage Directed test Generation). For this problem, we pay our attention to a particular class of machine learning algorithms called Artificial Neural Networks (ANNs). An ANN consists of a collection of highly-interconnected processing elements to model the complicated relationship between input and output. The actual relationship is determined by the set of weights, which are dynamically updated by a training algorithm, such as back-propagation *etc.*, associated with the links connecting elements. Without any prior knowledge of target function, the representational power of ANNs is rich enough to express complex interactions among variables: any function can be approximated to arbitrary precision by three-layer ANNs [9]. Another advantage of ANNs is that the target function output can be discrete-valued, real-valued, or a vector of several discrete- or real- attributes, which is well suited to express the bug information of a design.

2) Classification Model: Identifying whether or not a module contains bugs is an important task of the bug prediction, which can be treated as a conventional *classification* problem according to the machine learning view. More specifically and formally, to construct such a classification model, first we should attain a set of training data with l samples as $D = \{(\boldsymbol{z_i}, y_i) | \boldsymbol{z_i} \in R^m, y_i \in \{+1, -1\}, i = 1, \ldots, l\}$, where $\boldsymbol{z_i}$ is a vector of m PCs as $\boldsymbol{z_i} = \{f_{i1}, \ldots, f_{im}\}$ (which is reconstructed from the original attributes extracted from the ith module as stated in Section IV-B), and y_i is a *label* indicating the ith module contains bug or not. According to the label y_i, the data set is divided into two categories: one category that contains modules with at least one bug is labeled as +1 and the other contains modules without bugs is labeled as -1. Once we obtain the classification model from training data with stated form, it can be employed to estimate the *label* of new given testing data.

3) Regression Model: In addition to the estimation of whether or not a module is buggy, we also want to predict the probable number of bugs in a module with the help of ANN model, which can be regarded as a *regression* problem. Unlike stated classification problem, the target variable of regression problem is real-valued instead of discrete-valued, that is, the form of training data with l samples can be expressed as $D = \{(\boldsymbol{z_i}, y_i) | \boldsymbol{z_i} \in R^m, y_i \in R, i = 1, \ldots, l\}$, where $\boldsymbol{z_i}$ is the same meaning as in classification model and y_i indicates the number of bugs of the ith module. And y_i is attained from the bug repositories of evaluated designs in the training data. Then, during prediction, y_i provides the probable number of bugs for a new module by trained regression model.

TABLE II
PROFILE OF EVALUATED DESIGNS

Design Name	# of Revisions	# of Modules
UART 16550	108	10
Memory Controller	30	15
AC 97 Controller	20	15

V. PRELIMINARY EVALUATION

A. Experimental Setup

To validate the assumption that there exists underlying relationship between design complexity and bug occurrence, we preform experiments on several open source projects downloaded from **www.opencores.org**, as listed in Table II.

To collect training data from these designs, we intensively inspect their source codes and corresponding change logs. According to [14], the revisions with change logs containing bug fixing information are considered as the bug fix revisions and the revisions prior to them are their respective buggy revisions. Thus, by comparing the buggy and the bug fix revisions, it is possible to obtain the number of bug fix patterns, *e.g.*, **Addition of if branch**, *etc.*, in these two revisions. Unfortunately, this scheme is not practical to count the actual number of bugs. The reason is that one bug may be related to more than one line or one "bug hunk", *i.e.*, code section in the buggy version that is modified in the bug fix version, which causes counting the number of actual bug is extraordinary ambiguous without a detail change log of bugs. Nevertheless, whether or not a fixed module evolved from a buggy version can be identified via "diff" each version. During the comparison of each version, we ignore those changes not related to the functional correctness, *e.g.*, modification of comments, includes, *etc*. The analyzed results of employed designs are listed in Appendix A. Accordingly, in this experiment, we only construct classification models for evaluated designs due to the lack of accurate training information. We anticipate that in the future more detail bug repositories (similar to information provided by Bugzilla [*] in software engineering) can be contributed by developers and maintainer accompanying with open source designs. Besides, we employ **Perl** to build an automatic static analysis tool for attaining attributes, which are the source to reconstruct PCs of training data, of a high-level design.

B. Results

Following the approach described in previous sections, we first demonstrate the effectiveness of PCA that reduces the correlation among attributes. The result on UART is presented in Table III. We only reserve 3 PCs since their eigenvalues are both greater than 1.0 and the cumulative variance is already more than 95%. We can see that PC_1 is almost related to all the attributes except NOII, which is highly correlated with PC_2. And PC_3 is more correlated with NOC and NOIP.

During the training of classification models, to avoid over-fitting of the training model, 10-fold cross-validation is utilized for learning and testing. In the 10-fold cross-validation method, we randomly divide the training set into 10 subsets. Then, we select the first fold as the test set, and the others as the training set. To measure the classification results, *Precision* (the proportion of true positive against all positive modules[†]) and *Accuracy* (the proportion of correctly classified modules to all modules) are employed. Actually, according to Appendix A, we can see that most modules in UART 16650 and Memory Controller are buggy. Conversely, only a few modules in AC 97 controller are buggy. From Table IV, we can see that learned bug models on both categories performs well, *i.e.*, only two modules are classified into opposite class in total.

TABLE III
PRINCIPAL COMPONENTS FOR UART 16650

Metrics	PC_1	PC_2	PC_3
LOC	0.2704	0.0073	0.0876
NODP	0.2677	0.1364	-0.014
NOP	0.2303	-0.2528	-0.2535
NOIP	0.0286	-0.4217	-0.5609
NOOP	0.2685	-0.0792	-0.0031
NOTO	0.268	0.0716	0.1545
NOUO	0.2649	0.1157	-0.0044
NOBO	0.2589	0.0474	0.2252
NOSD	0.276	-0.0289	0.0289
NOIM	0.1752	-0.4253	0.1694
NODM	0.2075	-0.35	0.1856
NOII	0.0733	-0.5455	0.0999
NOOI	0.2646	0.125	0.0566
TOC	0.2566	0.1265	-0.1765
NOC	0.1579	0.1795	-0.6442
NOS	0.2689	0.0678	0.0466
NOCS	0.2526	0.1629	0.0673
NOPB	0.2609	0.1257	-0.0929
Eigenvalues	12.97361	2.9684	1.25473
Cum. Var. %	72.076	88.567	95.537

TABLE IV
CLASSIFICATION RESULTS

Design Name	Precision	Accuracy
UART 16550	100%	90%
Memory Controller	90.91%	93.33%
AC 97 Controller	100%	100%

According to above experimental results, we are confident that proposed attributes are indeed related with bug occurrence. Furthermore, evaluated designs from different application fields show that our approach is independent of concrete designs and can be widely used in practice.

VI. INDUSTRIAL PRACTICES

A. Investigated Designs

In this section, we conduct a case study on a series of industrial general purpose microprocessors named Godson-2 [8], which is a 64-bit, 4-issue, out-of-order execution RISC processor that implements the 64-bit MIPS-like instruction set. Precisely, we adopt two releases as Godson-2A and Godson-2E to perform the experiments. Godson-2A is the first version of Godson family and consists of more than 340 modules. And improved Godson-2E is composed of about 530 modules and has been massively produced over several millions. For simplicity, we refer these two releases as Release 1 and Release 2, respectively. In each release, the whole system is logically divided into several subsystems, each is built of components. Each component, which is stored in a file, consists of a number of modules. And in comparison with Release 1, 16 new components with about 120 modules are added in Release 2 to enhance the functionality and performance.

Currently, we intend to verify Release 2 on the premise that we have already possessed the bug information of Release 1. In other words, we hope to assist verification process on Release 2 with the help of predictive models on bugs that are obtained via proposed approach from high-level design and bug repository of Release 1. To attain the set of training data as stated in Section IV-C, we collect the number of bugs detected and corrected in each module of Release 1 from its bug repository, which is used to report and track

[*] www.bugzilla.org

[†] positive module here means the module is buggy

TABLE V
CLASSIFICATION RESULTS ON RELEASE 2

		Predicted Buggy	
		yes	no
Actual Buggy	yes	$A = 18$	$B = 4$
	no	$C = 11$	$D = 87$

Positive Precision = 62.07% Negative Precision = 95.60%

$$\text{Accuracy} = \frac{A+D}{A+B+C+D} = 87.5\%$$

problems with, and potential enhancement to, the design. In Release 1, the number of *logic*, *algorithmic* and *timing design bugs* [4] that are detected during system-level verification, which is especially expensive and resource-intensive phase of development, is 118. And they scatter in 41 different modules of 3 main subsystems as out-of-order execution logic, memory system (includes I/DCache, I/DTLB and load/store queue) and functional units (includes integer and floating-point functional units). Considering the balance of the number of two category modules, we *randomly* select other 59 representative modules without bugs to constitute the training set with 100 modules.

B. Early Bug Prediction on Release 2

As stated in Section III, cross-releases bug prediction focuses on components that are newly added in Release 2, which consist of 120 modules. The results of classification and regression model on such modules are presented as follows.

1) Classification Results: Since we have built classification model on Release 1, it can be used to predict bug information of new modules in Release 2. The effectiveness of such model is also evaluated by two stated measures: *precision* and *accuracy* as shown in Table V. Moreover, positive precision, which is defined as the number of true positives divided by all positive modules, of classification model is 62.07%. Similarly, negative precision is 95.6%. And accuracy, *i.e.*, the proportion of correctly classified samples in the total population, of classification model is 87.5%.

Positive precision of classification results (62.07%) implies that a considerable proportion of modules that are classified as buggy are actually bug-free. Fortunately, such inaccurate result only incurs a small amount of waste of resources and time efforts. On the other hand, most prone-to-bug modules ($\frac{A}{A+B}$ = 81.82%) are captured by proposed predictive models, which is very beneficial to verification since those modules we will focus on indeed contain bugs.

2) Regression Results: Since it is impractical to list prediction results of all modules in this paper, we categorize the modules in Release 2 into different classes according to corresponding *number of system-level bugs*, which ranges from 0 to 7, as shown in Table VI. As we can see, most of the modules (81.67%) are bug-free and the mean absolute error of predicted value in this category is only 0.3419. Although for some modules (in the second row) the predicted results may be too inaccurate to determine the completion criterion of verification, we can still ascertain that such modules are with more bugs than modules with 1 bugs according to predicted results. Thus, unbalanced resources allocation can still be carried on under the guidance of bug prediction.

C. Utilization of Early Bug Prediction

Now that the effectiveness of proposed approach to predict bug information of Release 2 has been validated, we give a detailed explanation that employs bug prediction in the early stage to assist verification of Release 2 as described in Section I.

TABLE VI
SUMMARY OF REGRESSION RESULTS ON RELEASE 2

# of System-level Bugs	**# of Modules**	**Mean Absolute Error**
7	1	1.258
6	2	2.132
4	1	0.024
3	4	1.063
2	2	1.317
1	12	0.8326
0	98	0.3419

(1) When creating a verification plan, verification resources should be inclined to those prone-to-bug modules, *i.e.*, those modules are predicted containing bugs as identified by classification models. More precisely, for those buggy modules, regression model demonstrates the verification priorities of all modules according to their potential bugs. For instance, *dcache* module contains more bugs than *gr* module in our case, where more resources should be invested consequently, *e.g.*, during the full-chip verification, we can set constraints of constrained random generator so that the difference of probabilities of hitting the coverage space of these two modules is comparable with the gap of numbers of predicted bugs in these two modules.
(2) Bug information not only can facilitate the allocation of verification resources, it also can be employed to assess the completeness of the verification, even provide grounds to select modules for formal verification. Take *fmaf* component as an example, which is an additional functionality in Release 2 against Release 1 to replace the original *fadd* and *fmul* components so as to improve the floating-point performance. An important characteristic of such operation units is that their functional coverage models are not so straightforward to specify as control logic due to the large testing space. By employing proposed approach, the total number of probable bugs in *fmaf* is 8, while the detected number was only 2 without report of new bugs in later two months during system level verification. In this situation, we could not conclude that it was near the end of verification even constraints of other modules were met. Fortunately, since many verification resources were transferred from other modules where completion criteria were satisfied, two extra bugs were exposed. However, the number of detected bugs was still far less than predicted value. Therefore, formal method were considered to expose potential counterexamples in the design to ensure its quality. Eventually, final three bugs were detected by adopted formal method. In this case, although the prediction is not complete accurate, our method can yet employed to guide verification in a higher level compared with coverage model.
(3) As verification proceeds, most of the potential bugs are exposed for several modules. Then, if *bug detection rate*, the proportion of exposed bugs against the total potential ones of a design, and other constraints (functional coverage, structure coverage and bug drop rate, *etc.*) are met, we can conclude that verification on such modules approximates the end phase. Therefore, we can concentrate on those complicated modules which are still with many unexposed hard-to-verify bugs.

VII. DISCUSSIONS

A. Emphasis on bug repository

Obviously, performance of predictive model is decisive to the successful application of our methodology. To obtain an accurate enough model, the quality of training data is very crucial. Unfortunately, the bug repository is always maintained for specific purposes, such as project management and problem tracking *etc.*, instead of for bug prediction. Hence, the structure

and quality of such bug repository may be inappropriate to collect required metrics for prediction. Actually, in the case study of ourCPU, through inspecting the source codes of such modules, we notice that many bugs documented in the RTL source codes, which are represented by *comments*, do not comply with bugs reported in bug repository. Therefore, we should attach more importance to the maintenance of bug repository in the future, which may not only be beneficial to proposed approach, but also facilitate future design and verification to improves the design quality. According to our experience, each bug report submitted to the repository should at least contains bug ID, time of exposure, exposure tool, information of corresponding module (which can be tracked from version database), bug type, bug level(unit-level, system-level or instruction-level) *etc.*

B. Limitations and extensions

Now we highlight the limitations of our proposed verification-assisting methodology. First and most important, our work only investigates the correlation between complexity metrics of HDL codes and bug occurrence, which is still hard to delve all possible factors, such as, designers experiences, management ability *etc.*, that impact bugs. To eliminate the effect of these human-related factors, our approach can be applied on consecutive releases of one project, which implies that human intervention does not fluctuate a lot among releases. Second, our scheme tries to predict bug information of new modules, *modules without bug history*, in the next release. Whether or not predictors constructed by our approach can be employed to predict the bug occurrence of evolving modules should be validated by further experiments. To extend our approach, we may import *changing rate* of the code as a metric to predict bugs. The intuition is that module that changing a lot is of lower quality. In summary, the above stated limitations stem from the consideration of a small subset that affects the bug-proneness of a design. Thus in the future we try to investigate more factors, such as the qualification of designers, the time taken to write a module, the quality of specification and the competence of management, *etc.* Therefore, we can determine the weight of each component contributes to bug occurrence, which is instructive to provide insights for designers and verification engineers.

VIII. Conclusions

We propose a novel verification methodology, which is based on the predictive models on bugs to guide and assess verification process. It can be treated as a crucial complementary technique to existing coverage model so as to make verification more effective and efficient. Such a methodology is especially suitable for the verification of a series of designs, since the more prior knowledge, the more accurate the predictive model could be. When enough prior knowledge is accumulated, the novel verification methodology could even become the main evaluation technique of verification. Since there are so many design series in industry, our methodology may facilitate more industrial practices in the future. As stated, we hope that our work can open up a large unexplored area of hardware design that deals with mining all possible factors that are decisive to the occurrence of bugs.

References

[1] C. Bazeghi, F. J. Mesa-Martinez, and J. Renau. μComplexity: estimating processor design effort. In *Proc. MICRO*, pages 209–218, 2005.

[2] B. Bentley. Validating the intel pentium 4 microprocessor. In *Proc. DAC*, pages 244–248, 2001.

[3] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *J. Syst. Softw*, 51(3):245–273, 2000.

[4] K. Constantinides, O. Mutlu, and T. Austin. Online design bug detection: RTL analysis, flexible mechanisms, and evaluation. In *Proc. MICRO*, pages 282–293, 2008.

[5] A. Gluska. Practical methods in coverage-oriented verification of the merom microprocessor. In *Proc. DAC*, pages 332–337, 2006.

[6] J. Greenwald and A. Frank. Data mining static code attributes to learn defect predictor. *IEEE Trans. Softw. Eng.*, 33(1):2–13, 2007.

[7] Q. Guo, T. Chen, H. Shen, and Y. Chen. Estimating design qulaity of digital systems via machine learning. In *Proc. ICECS*, 2010.

[8] W. Hu, F. Zhang, and Z. Li. Microarchitecture of the godson-2 processor. *J. Comput. Sci. Technol.*, 20(2):243–249, 2005.

[9] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[10] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[11] D. Protheroe and F. Pessolano. An objective measure of digital system design quality. In *Proc. ISQED*, pages 227–233, 2000.

[12] R.A.Johnson and D.W.Wichern. *Applied Multivariate Statistical Analysis*. Prentice-Hall, 1998.

[13] N. S. Stollon and J. D. Provence. Measures of syntactic complexity for modeling behavioral vhdl. In *Proc. DAC*, pages 684–689, 1995.

[14] S. Sudakrishnan, J. Madhavan, E. J. Whitehead, Jr., and J. Renau. Understanding bug fix patterns in verilog. In *Proc. MSR*, pages 39–42, 2008.

[15] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proc. ESEC/SIGSOFT FSE*, pages 91–100, 2009.

Appendix

Module	Observed Buggy	Predicted Buggy
UART 16650		
uart_raminfr	false	false
uart_debug_if	true	true
uart_receiver	true	true
uart_regs	true	true
uart_rfifo	true	true
uart_sync_flops	false	false
uart_tfifo	**false**	**true**
uart_top	true	true
uart_transmitter	true	true
uart_wb	true	true
Memory Controller		
mc_adr_sel	true	true
mc_cs_rf	true	true
mc_cs_rf_dummy	true	true
mc_dp	true	true
mc_incn_r	false	false
mc_mem_if	true	true
mc_obct	true	true
mc_obct_dummy	false	false
mc_obct_top	false	false
mc_rd_fifo	**true**	**false**
mc_refresh	true	true
mc_rf	true	true
mc_timing	true	true
mc_top	true	true
mc_wb_if	true	true
AC 97 Controller		
ac97_cra	true	true
ac97_dma_if	false	false
ac97_dma_req	false	false
ac97_fifo_ctrl	false	false
ac97_int	false	false
ac97_in_fifo	true	true
ac97_out_fifo	true	true
ac97_prc	false	false
ac97_rf	false	false
ac97_rst	false	false
ac97_sin	false	false
ac97_soc	true	true
ac97_sout	false	false
ac97_top	true	true
ac97_wb_if	true	true