Pre-Silicon Bug Forecast

Qi Guo, Tianshi Chen, Yunji Chen, Rui Wang, Huanhuan Chen, Weiwu Hu, and Guoliang Chen

***Abstract*—The ever-intensifying time-to-market pressure imposes great challenges on the pre-silicon design phase of hardware. Before the tape-out, a pre-silicon design has to be thoroughly inspected by time-consuming functional verification and code review to exclude bugs. For functional verification and code review, a critical issue determining their efficiency is the allocation of resources (e.g., computational resources and manpower) to different modules of a design, which is conventionally guided by designers' experiences. Such practices, though simple and straightforward, may take high risks of wasting resources on bug-free modules or missing bugs in buggy modules, and thus could affect the success and timeline of the tape-out. In this paper, we propose a novel framework called pre-silicon bug forecast to predict the bug information of hardware designs. In this framework, bug models are built via machine learning techniques to characterize the relationship between design characteristics and the bug information, which can be leveraged to predict how bugs distribute in different modules of the current design. Such predicted bug information is adequate to regulate the resources among different modules to achieve efficient functional verification and code review. To evaluate the effectiveness of the proposed pre-silicon bug forecast framework, we conducted detailed experiments on several open-source hardware projects. Moreover, we also investigate the impacts of different learning techniques and different sets of characteristic on the performance of bug models. Experimental results show that with appropriate learning techniques and characteristics, about 90% modules could be correctly predicted as buggy or clean and the number of bugs of each module could also be accurately predicted.**

***Index Terms*—Bug forecast, code review, design characteristics, functional verification, machine learning.**

Manuscript received May 1, 2013; revised July 7, 2013 and August 24, 2013; accepted October 7, 2013. Date of current version February 14, 2014. This work was supported in part by the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant XDA06010401-02, in part by the National Natural Science Foundation of China under Grant 61100163, Grant 61003064, Grant 61222204, Grant 61050002, Grant 61173006, Grant 61133004, and Grant 61173001, in part by the National S&T Major Project of China under Grant 2009ZX01028-002-003, Grant 2009ZX01029-001-03, and Grant 2010ZX01036-001-002, and in part by the National 863 Program of China under Grant 2012AA012202. The work of H. Chen was supported by the One Thousand Young Talents Program. This paper was recommended by Associate Editor K. Chakrabarty *(Corresponding author: T. Chen)*.

Q. Guo, T. Chen, Y. Chen, and W. Hu are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, China (e-mail: joyguoqi@gmail.com; chentianshi@ict.ac.cn; cyj@ict.ac.cn; hww@ict.ac.cn).

R. Wang is with Anhui USTC iFLYTEK Company, Ltd., Heifei, 230088, China (e-mail: wrui1108@mail.ustc.edu.cn).

H. Chen and G. Chen are with the University of Science and Technology of China, Heifei, 230027, China (e-mail: hchen@ustc.edu.cn; glchen@ustc.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2013.2288688

I. Introduction

A. Motivation

HARDWARE designs continue to grow in complexity, and it has been a great challenge to guarantee the successful tape-out of a hardware design under the ever-intensifying time-to-market pressure. During pre-silicon design cycles, functional verification and code review (or code inspection) are two critical stages to detect functional bugs. When verifying STMicroelectronics' processor families, functional verification and code review exposed 74% and 23% bugs, respectively [23]. When verifying Intel's Pentium 4 processor, functional verification and code review exposed 80% and 20% bugs, respectively [4].

Functional verification and code review are so important that designers are willing to invest considerable resources (e.g., manpower or computational resources) on them. However, investing more resources does not necessarily lead to exposures of more bugs, especially when they are not adequately regulated among different modules of a large-scale hardware design. Conventionally, resource allocation largely depends on personal experiences of designers. For example, modules with complex logics or written by inexperienced engineers should be allocated more computational resources in functional verification, and may also be carefully reinspected in code review. Strongly relying on experiences of designers, such ad-hoc rules may take a high risk of wasting resources on bug-free modules, or missing bugs in buggy modules. Although coverage-centric verification (e.g., coverage-directed [11], [38] and coverage-oriented verification [13], [14]) may alleviate the above burden, even 100% coverage does not indicate a module to be bug-free, because coverage models (especially the widely used functional coverage model) still rely heavily on designer's experiences, and typically only cover a subset of functional components that are thought to be most vulnerable in a module.

B. Our Idea and Contributions

The above challenges can be significantly alleviated by exploring bug information of a pre-silicon design in a smart way. To be specific, if we are able to know how bugs distribute in different modules of the design, then we can adequately regulate resources spent on different modules to achieve a successful and efficient pre-silicon design phase. In line with the above thinking, we propose a bug forecast framework for pre-silicon design.

Before introducing the pre-silicon bug forecast framework, we first take an example of how it benefits functional verification and code review. Let us consider a pre-silicon design consisting of three modules, m_1, m_2, and m_3, as illustrated in

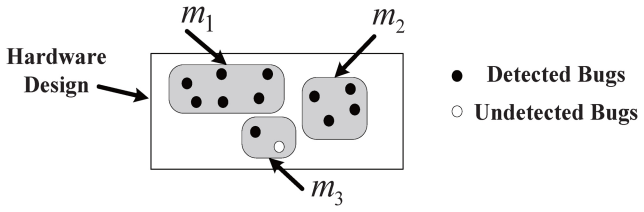


Fig. 1. How predicted bug information assists verification.

Fig. 1. According to the predicted bug information, m_1, m_2, and m_3 are suspected to have five, three, and two bugs, respectively. When arranging the initial verification plan, the module suspected to have the largest number of bugs (m_1 in Fig. 1) shall be allocated the largest amount of verification resources. As verification proceeds, once all suspected bugs in a module have been exposed, the verification resources allocated to the module shall be progressively transferred to modules with unexposed bugs. Once there is no difference between exposed and predicted numbers of bugs, we can conclude that the verification is complete if the bug predictor is perfect. For code review of a pre-silicon design, it is even more straightforward to leverage results of bug forecast. In short, one will pay special attention to modules that are predicted to be buggy. In fact, a similar methodology has already been utilized in code review of large-scale software systems of Microsoft [10] and Google [1], which significantly improves the maintenance efficiency and quality of their commercial softwares. The successful applications in software development clearly show potential benefits of bug forecast in hardware design.

Here, we briefly explain the pre-silicon bug forecast framework proposed in this paper. In a nutshell, pre-silicon bug forecast consists of two steps. At the first step, bug models are constructed via machine learning techniques to characterize the relationship between design characteristics and bug information, where design characteristics are grouped to three categories, including code characteristics about the code complexity, history characteristics about the changing history of the design, and organization characteristics about the organization of developers. The training process leverages historical data collected from previous revisions of the design to obtain bug models. After training, the bug models are ready to predict the bug information with respect to each module of the current design. To evaluate the effectiveness of proposed framework, we conduct detailed experiments on several open-source hardware projects. Experimental results show that with appropriate learning techniques and characteristics, about 90% modules could be correctly predicted as buggy or clean, and the number of bugs of each module can also be accurately predicted.

The main contributions of this paper can be summarized as follows. First, we propose a pre-silicon bug forecast methodology to accurately predict bug information of a design based on its code/history/organization characteristics. Second, we empirically reveal that choices of machine learning technique and design characteristics may have impacts on the accuracy of pre-silicon bug forecast, and suggest trying multiple learning techniques for building bug models in practice, and then use the model that performs the best on the validation set

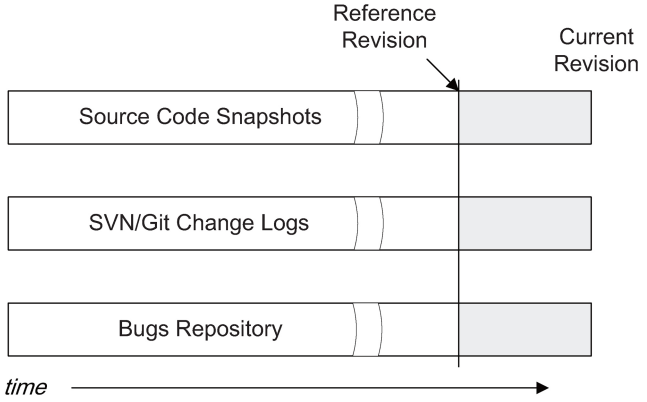


Fig. 2. Bug information of the current revision is predicted by the bug models built from the reference revision.

for further prediction. At last, we propose a bug-oriented methodology that adopts pre-silicon bug forecast to facilitate functional verification and code review.

The rest of this paper proceeds as follows. Section II presents the pre-silicon bug forecast framework. Section III provides experimental results on several open-source projects. Section IV empirically studies how different sets of design characteristics impact the accuracy of pre-silicon bug forecast. Section V presents the results of cross-revision prediction. Section VI introduces bug-oriented methodology for functional verification and code review. Section VII discuss threats and potential extension of our approach. Section VIII reviews related work, and Section IX concludes the whole paper.

II. Pre-Silicon Bug Forecast

A. Overview

Pre-silicon bug forecast consists of two steps. At the first step, machine learning techniques are utilized to train bug forecast models with historical data collected from a previous revision (e.g., a revision that is one year before the current revision) of the pre-silicon design, and this revision is called the reference revision. At the second step, the bug information with respect to each module of the current revision can be predicted by these trained bug forecast models.

Historical data utilized at the first step of bug forecast are collected from the reference revision, which falls into two parts. The first part of data contains design characteristics with respect to the reference revision, which are extracted from three main repositories. Fig. 2 depicts the three repositories. The first repository is the source code snapshots, which are usually stored in the version control systems (VCS), such as SVN[1] and Git.[2] The second repository is the changing log repository of VCS, and it always contains additional information (e.g., comments on revised modules) for each committed revision. The third repository is the bug repository that should contain the detailed information of each detected bug (e.g., bug ID, bug severity, bug description, etc.).

The second part of historical data contains bugs that are detected from the reference revision, which are also extracted

[1]Available at http://subversion.tigris.org.
[2]Available at http://git-scm.com.

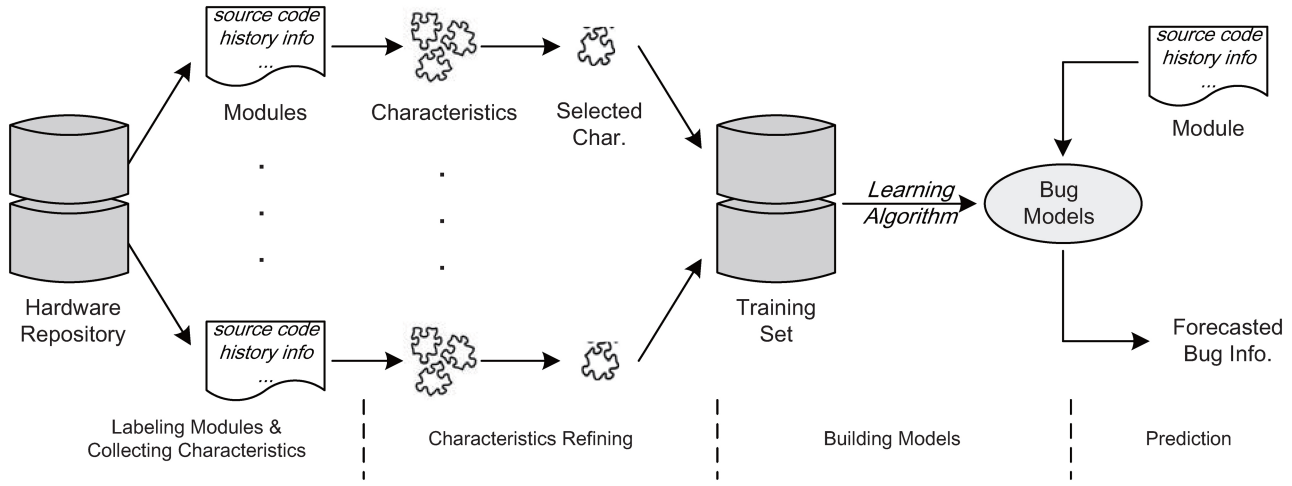


Fig. 3. Framework of pre-silicon bug forecast methodology.

from the bug repository. The extracted bug information, together with the stated design characteristics, constitutes the entire training set for constructing bug forecast models.

B. Framework

The overall framework of pre-silicon bug forecast is illustrated in Fig. 3, which consists of five stages, i.e., labeling historical modules, collecting module characteristics, characteristic refining, building bug models (learning), and prediction.

1) *Labeling Historical Modules:* The label of a module is the bug information (e.g., whether buggy or not, the number of bugs) of the module, and labeling modules refers to the process that collects bug information for modules of the reference revision.

2) *Collecting Module Characteristics:* The bug occurrence of a module relates to different factors, including design complexity, revision interval, experiences of engineers, and so on. In this paper, we specify several module characteristics to quantify such factors, and collect them from hardware repositories, i.e., source code snapshots, changing log repository, and bug repository.

3) *Characteristic Refining:* Module characteristics manually specified at the last step are based on experiences, and may be mixed up with redundant and/or noisy characteristics. To remove such characteristics, we use a genetic algorithm to select informative characteristics that are most closely related to bug occurrence. After refining module characteristics, we obtain the training set for building bug models.

4) *Building Bug Models (Learning):* We use various learning techniques, such as naive Bayesian [25] and support vector machines (SVMs) [37], to build classification or regression models based on the training set.

5) *Prediction:* With bug models, the bug information (e.g., whether buggy or not, the number of bugs) with respect to modules of the current revision can be predicted.

Details of each stage are presented in the rest of this section.

C. Labeling Historical Modules

The label of a historical module is the bug information of that module, which is collected from hardware repositories. Although there have been several automatic algorithms (e.g., SZZ algorithm [33]) to collect labels of functions for software designs, they are not applicable to hardware designs, because the bug tracking systems required by these algorithms are still not widely deployed in hardware designs. In the absence of a detailed bug tracking system, we follow the basic idea of the approach proposed in [35] to label modules, which only requires inspecting the VCS changing logs.

More specifically, the labeling process has the following steps.

1) For all intermediate revisions between the reference revision and the current revision, we scan their corresponding SVN changing logs via regular expression as bug|fix to determine the bug-fix revisions that contain committed changes fixing at least one bug.
2) For each bug-fix revision identified at Step 1, we execute the diff command to compare it with its preceding revision on the changed modules to locate the modified chunks. For each modified chunk, by executing blame command supported by SVN, we identify the bug-introducing revision that brings this buggy code chunk to the module.
3) For each bug-introducing revision identified at Step 2, if it is committed before the reference revision, we link the introduced bug to the module in the reference revision according to the SVN logs, and the number of bugs of this module is increased by one. Fig. 4 shows that modules in the reference revision are only linked with the bugs that are introduced before the reference revision and fixed after the reference revision.
4) Once all bug-introducing revisions have been processed, the number of bugs with respect to each module in the reference revision can be identified.

D. Collecting Module Characteristics

There are, in general, three categories of module characteristics that may be decisive to bug occurrence: code characteristics, history characteristics, and organization characteristics.

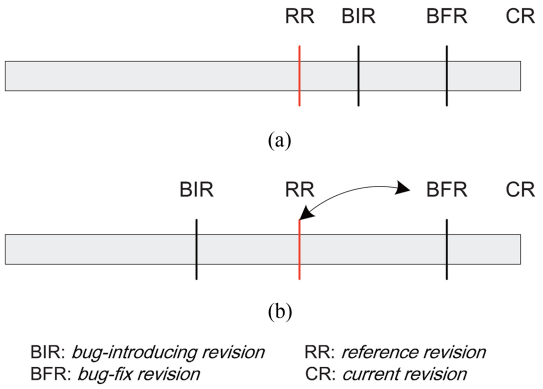


Fig. 4. How to link fixed bugs to modules in the reference revision. (a) If a fixed bug is introduced after the reference revision, it should be ignored. (b) If a bug is introduced before the reference revision and fixed after the reference revision, the number of bugs in the related module of the reference revision should be increased by one.

TABLE I
CODE CHARACTERISTICS OF MODULE

Characteristics	Definition
LOC	Lines of Code
DecPntNum	Number of Decision Points
PortNum	No. of Ports
InPortNum	No. of Input Ports
OutPortNum	No. of Output Ports
OpNum	No. of Operators
UnOpNum	No. of Unary Operators
BiOpNum	No. of Binary Operators
SigDefNum	No. of Signal Definitions
InModNum	No. of Instantiated Modules
ConnNum	No. of Connections to Inst. Modules
StatNum	No. of Statements
ProBlkNum	No. of Process Blocks

1) *Code Characteristics:* It is natural that a more complex module is more vulnerable to bugs. A straightforward way of measuring the design complexity of a hardware module is to use code characteristics. In our study, code characteristics are chosen independent of specific hardware description languages, which is listed in Table I.

2) *History Characteristics:* History information, such as past changes, fixes, bugs, and so on, may also have significant impacts on bug occurrence. In the domain of software engineering, history information has already been demonstrated to be helpful in predicting software defects [20]–[22], [26], [31], [42]. Hence, the proposed pre-silicon bug forecast framework also utilizes history information. Table II lists 11 history characteristics of hardware modules considered in our study. The first characteristic is the number of changes on all previous revisions, which is calculated with the diff command that compares the changed code blocks between two consecutive revisions. The second characteristic is the number of bug-fix

TABLE II
HISTORY CHARACTERISTICS IN MODULE LEVEL

Characteristics	Definition
Chgs	number of changes
FixChgs	number of bug-fix changes
PastBugs	number of fixed bugs
Period	period between first and lasted rev.
AvgInt	avg. interval between two rev.
MaxInt	max interval between two rev.
MinInt	min interval between two rev.
AvgLocAdd	avg. added LOC between two rev.
MaxLocAdd	max. added LOC between two rev.
AvgLocDel	avg. deleted LOC between two rev.
MaxLocDel	max. deleted LOC between two rev.

changes, which are introduced for fixing specific bugs in the bug-fix revisions. Such bug-fix revisions are identified from the initial revision to the reference revision. The third characteristic is the number of fixed bugs, and it is collected by manually inspecting the source code and the SVN changing logs, which is very similar to the labeling process introduced in Section II-C. The main difference is that the fixed bugs are identified from the initial revision to the reference revision, while the labeled bugs are identified from the reference revision to the current revision. The next four characteristics are about the time interval between two revisions, and these characteristics are extracted from the changing log repository. The last four characteristics are about the added and deleted lines of code between two consecutive revisions, which are extracted from both source code snapshots and changing log repository.

3) *Organization Characteristics:* Organization characteristics depict the organization profile of developers of a hardware project, which may also be crucial to bug occurrence. Actually, Hata *et al.* [20] proposed several relatively easy-to-collect organization characteristics for software projects. Following their methodology, we specify four organization characteristics as listed in Table III. The first characteristic is the number of developers that contribute to a module, and the other three characteristics are closely related to the ownership [5], [30]. The ownership of a developer for a specific module is defined as the ratio of the number of changes committed by the developer to the total number of committed changes for that module. If the ownership of a developer is less than a predefined threshold (e.g., 20%), the developer is considered a minor developer, otherwise, a major developer.

E. Characteristics Refining

The collected characteristics, in general, influence the bug occurrence of a module from different aspects, but may still be mixed up with redundant and/or noisy characteristics. To remove impacts of such characteristics on bug models, we consider using feature selection techniques to select the most informative characteristics. Compared with feature

TABLE III
ORGANIZATION CHARACTERISTICS IN MODULE LEVEL

Characteristics	Definition
DevTot	number of developers
DevMin	number of minor developers
DevMaj	number of major developers
MaxOwn	the max ownership

construction techniques, such as principal component analysis (PCA), which reduce the dimension of feature vector by mixing the original features, the feature selection techniques directly select subsets of features that are useful for building a good predictor.

One of the most critical steps in feature selection techniques is to define how to explore feature subset space. In this paper, we decide to employ genetic algorithms (GA) [15] to efficiently explore the space, since GA is a widely used search strategy that has prominent capability in solving global optimization problems.

GA simulates the process of natural evolution to solve various optimization problems. More specifically, GA starts from an initial population that typically contains randomly generated solutions (a solution is represented as a binary string). For each string in the population, the corresponding fitness is evaluated via fitness function. Based on the fitness value, some strings are selected to produce the next population. The new population is typically produced by two kinds of operators, that is, crossover that crosses over two strings with a specific probability to form a new string, and mutation that mutates each position in the string with a specific probability. Given the newly generated population, the stop criterion (e.g., overall fitness cannot be improved) will be tested to decide whether or not terminate the evolution process.

In our problem, all 28 characteristics are encoded as a binary string, where 1-bit represents that the corresponding characteristic is selected and 0-bit represents that the corresponding characteristic is not selected. During evolution, the correlation between individual characteristics and bug occurrence, along with the intercorrelation among characteristics, is taken into account by the fitness function [18]. Under the fitness function, high fitness scores will be assigned to solutions with some characteristics that are highly correlated with the class label, yet uncorrelated with each other. More specifically, the fitness function is formally defined as

$$fitness(s) = \frac{k\overline{r_{cf}}}{\sqrt{k + k(k-1)\overline{r_{ff}}}} \quad (1)$$

where s represents a feature subset consisting of k features, $\overline{r_{cf}}$ is the average feature-class correlation, and $\overline{r_{ff}}$ is the average feature-feature correlation. Once the fitness of all individuals in one generation is the same (i.e., the evolution process converges), or the total number of generations reaches the predefined threshold, the GA stops and the selected characteristics are output. Besides, once the final solutions (feature subsets) have the same fitness, we simply take one from such solutions. In practice, the importance of different characteristics may vary in different projects. Thereby, the characteristic refining process should be independently conducted for each project.

F. Learning and Prediction

Labels and characteristics of historical modules extracted from hardware repositories can be treated as the training data to build classification and regression bug models. Here, we introduce such bug models in the context of pre-silicon bug forecast.

1) *Classification-Based Bug Model:* It is an important task to identify whether a module contains bugs in pre-silicon bug forecast, and we formulate it as a traditional classification problem from the machine learning perspective. Specifically, we first obtain a set of training data D 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 selected module characteristics $\boldsymbol{z_i} = \{f_{i1}, \ldots, f_{im}\}$, and y_i is the label indicating whether the ith module contains bugs. Based on the value of the label y_i, the ith module could be classified into two categories: +1 indicates that the module contains at least one bug (e.g., buggy), and -1 indicates that the module does not contain any bugs (i.e., clean). Once the classification model is built from training data in the above form, it can be used to predict the label of a new module.

2) *Regression-Based Bug Model:* In addition to predicting whether a module is buggy, we also want to predict the number of potential bugs in a module, which can be formulated as a regression problem. In this regression problem, the form of training data with l samples can be formulated as $D = \{(\boldsymbol{z_i}, y_i)|\boldsymbol{z_i} \in R^m, y_i \in R, i = 1, \ldots, l\}$, where $\boldsymbol{z_i}$ has the same meaning as in classification model, and y_i indicates the number of bugs of the ith module. Then, in prediction, y_i corresponds to the predicted number of bugs for the ith module in the current revision by the trained regression model.

III. EMPIRICAL EVALUATION

In this section, we evaluate the effectiveness of the pre-silicon bug forecast framework over several open-source hardware designs.

A. Datasets

We employ five open-source hardware designs in our experiments, where four designs (ethmac, openMSP430, or1200, vga_lcd) are from OpenCores,[3] and one (non router) is an open-source project of Stanford University.[4] They cover different application fields, such as processors (or1200), communication controller (ethmac), video controller (vga_lcd) and so on. Table IV summarizes some information about the five designs, including the total number of revisions as well as the reference revision considered in our study. Taking ethmac as an example, the reference revision is 301, which means that the code/history/organization information before

[3]Available at http://www.opencores.org.

[4]Available at https://nocs.stanford.edu/cgi-bin/trac.cgi/wiki/Project/RepositoryAccess.

TABLE IV
SUMMARY OF THE STUDIED PROJECTS

Name	Initial Date	Last Date	# of Bugs	# of Revisions	# of Modules on Last Date	Ref. Revision
ethmac	2002-11-15	2012-02-15	52	368	26	301
noc router	2007-11-16	2013-02-27	76	5487	33	1579
openMSP430	2009-07-01	2013-02-26	52	183	21	80
or1200	2009-04-21	2013-01-26	20	856	105	600
vga_lcd	2001-08-21	2009-03-10	23	62	15	37

revision 301 is extracted from the repositories and organized as design characteristics to construct bug models, and the bug information after revision 301 is treated as labels for checking the effectiveness of the bug models.

B. Selected Characteristics

Before using machine learning techniques to build bug models for each projects, GA is applied to select the most informative module characteristics. Parameter settings of GA are as follows. The population size is set to 100, the crossover probability is set to 0.7, and the mutation probability is set to 0.02.

The selected module characteristics for each project are listed in Table V, from which we observe that module characteristics selected for different projects are quite different. However, there are still common characteristics shared by different projects. For example, characteristic PortNum appears in the characteristic sets of ethmac, noc router, and vga_lcd, and Chgs appears in the characteristic sets of ethmac and noc router, which suggests that such characteristics are critical indicators of bug occurrence. Moreover, all organization characteristics have been eliminated by the GA, which suggests that they are not closely related to the bug occurrence in the evaluated projects.

C. Evaluations of Bug Models

We built both classification-based and regression-based bug models for the five designs, and evaluate the prediction accuracies of the models.

1) *Evaluation Techniques:* Before introducing the experimental results, we present the accuracy metrics utilized in our study. When querying a classification-based bug model whether a module is buggy, there are four possible outcomes, which are shown in Table VI [32]: 1) the module is buggy and been correctly predicted to be buggy (true positive, TP); 2) the module is clean but been incorrectly predicted to be buggy (false positive, FP); 3) the module is clean and been correctly predicted to be clean (true negative, TN); and 4) the module is buggy but been incorrectly predicted to be clean (false negative, FN). The overall accuracy of a classification-based bug model can be formulated by

$$Accuracy = \frac{\#TP + \#TN}{\#TP + \#FP + \#TN + \#FN} \tag{2}$$

which can simply be interpreted as the percentage of correctly classified modules. Sometimes, the number of clean modules can be much larger than the number of buggy modules, and the overall accuracy of a bug model can easily be very high. In this case, it would be more desirable to know how well the bug model predicts buggy modules. Thereby, we also employ metrics called precision and recall in our evaluations

$$Precision(Buggy) = \frac{\#TP}{\#TP + \#FP}$$
$$Recall(Buggy) = \frac{\#TP}{\#TP + \#FN}$$

where the precision on buggy modules is the percentage of correctly predicted buggy modules in all predicted buggy modules, and the recall on buggy modules is the percentage of correctly predicted buggy modules in all actual buggy modules. Based on precision and recall, the F-measure can be defined as

$$F - measure(Buggy) = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3}$$

which could be interpreted as the weighted average of precision and recall. In a similar manner, the precision, recall, and F-measure on clean modules can also be evaluated.

For regression-based bug models, we utilize root mean squared error (RMSE) to quantify the prediction error

$$RMSE = \sqrt{\frac{\sum_{t=1}^{n}(y_t - \hat{y}_t)^2}{n}} \tag{4}$$

where y_t and $\hat{y}_t$ are the actual and predicted #bug for module t, respectively.

In our experiments, we use ten-fold cross validation [39] to reduce stochastic effects brought by training and testing data. To be specific, the entire training set is randomly divided into ten subsets. Then, one subset is selected as the test set and the other nine subsets are taken as the training set for building a bug model. This procedure is repeated ten times on each subset. Finally, performance of all bug models is averaged as the evaluated results.

2) *Classification Results:* In our study, the learning techniques used for constructing classification models are artificial neural networks (ANN) [25], naive Bayesian (NB) [25], decision tree (J48) [25], random forest (RF) [6], and SVMs [37]. For ANN, we set two hidden layers, the first layer contains 16 neurons and the second contains four neurons. Besides, the momentum is 0.5, the learning rate is 0.001, and the learning epoch is 30 000. For SVM, we adopt grid search as suggested by Chang *et al.* [7] to find promising parameters for

TABLE V
SELECTED CHARACTERISTICS FOR EVALUATED PROJECTS VIA GA

Projects	Characteristics
ethmac	{PortNum, UnOpNum, BiOpNum, StatNum, Chgs, MaxLoCAdd, MaxLoCDel}
noc router	{DecPntNum, PortNum, UnOpNum, BiOpNum, StatNum, ConnNum, Chgs}
openMSP430	{LoC, OpNum, BiOpNum, PastBugs}
or1200	{DecPntNum, ProBlkNum, Period, MLoCAdd, MLoCDel}
vga_lcd	{PortNum, InPortNum, OutPortNum}

TABLE VI
CLASSIFICATION OUTCOMES

		Predicted	Results
		Buggy	Clean
Actual	Buggy	TP	FN
Results	Clean	FP	TN

each evaluated project. Table VII compares different learning techniques over different projects, where the best result on each classification metric (e.g., accuracy, precision, and recall) is marked in bold. Regarding the overall accuracy, SVM performs the best among evaluated learning techniques for all evaluated projects. In addition to the overall accuracy, we also present the classification results on buggy and clean modules. For some projects, such as noc router and or1200, the bug models perform better on clean modules than on buggy modules. The reason is that the clean modules are much more than the buggy modules in these projects. It is possible to employ advanced learning techniques [36] to handle such imbalanced data set, which should be left as our future work.

3) *Regression Results:* We employ ANN, M5P [29], REPTree [25], and SVM to build regression-based bug models. For each learning technique, we evaluate the prediction performance via ten repeated experiments for each learning technique. After that, we estimate the mean and standard deviation of the ten predicted results. Fig. 5 compares the learning techniques over all evaluated projects, where the mean and standard deviations of RMSE of bugs models are presented. We can see that SVM performs the best among all evaluated learning techniques, e.g., the RMSE of ANN, M5P, REPTree, and SVM on the project openMSP430 are 0.542, 0.732, 0.914, and 0.488, respectively. Besides, the low standard deviation of the predicted results of SVM also demonstrates the built bug models are stable to forecast bug information. Nevertheless, experiences gained on the evaluated projects (i.e., SVM outperforms other learning techniques on evaluated projects) may not be directly generalized to other projects due to no-free-lunch theorem (a famous theory of machine learning and optimization, which states that no algorithm can universally outperform another on all problems) [40]. Thus, we highly recommend trying multiple learning techniques for building bug models, and choose the model that performs the best on the validation set for further prediction in practice.

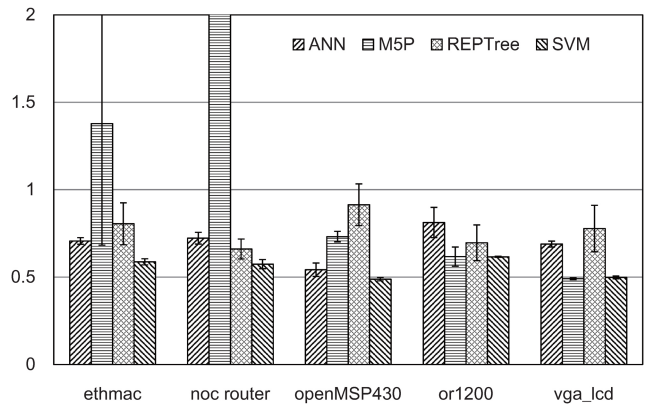


Fig. 5. Regression results (RMSE, the smaller the better) of bug models constructed by various learning algorithms.

In addition to reporting the average RMSE of each compared method, we further categorize the modules into different classes based on their number of bugs and list the corresponding prediction errors in Table VIII. As we can see, there is only one module containing four bugs, and the corresponding predicted result is 3.18. For those 16 clean modules, the mean prediction error is only 0.38.

IV. CHOICES OF MODULE CHARACTERISTICS

The choice of module characteristics may influence the accuracy of a bug model. In this section, we compare impacts of different characteristic groups on the accuracies of classification-based bug models, and empirically validate that the GA would be an appropriate way of refining characteristics.

Fig. 6 shows the overall accuracy of classification-based bug models built with code characteristics, history characteristics, organization characteristics, all characteristics (including all code/history/organization characteristics), principal components analysis (PCA)-constructed characteristics and GA-selected characteristics, respectively. The first observation is that no single type of characteristics[5] can achieve the best performance over all projects. For example, code characteristics could lead to best prediction accuracy on project noc router, openMSP430, and vga_lcd, while they lead to the worst prediction accuracy on project ethmac. The second observation

[5]We consider three types of module characteristics in this paper, which are code characteristics, history characteristics, and organization characteristics.

TABLE VII
CLASSIFICATION RESULTS OF EVALUATED PROJECTS

Projects	Classifiers	Accuracy	Buggy Precision	Buggy Recall	Buggy F-Measure	Clean Precision	Clean Recall	Clean F-Measure
ethmac	ANN	0.640	0.500	0.556	0.526	0.733	0.688	0.71
	NB	0.800	**1.000**	0.444	0.615	0.762	**1.000**	**0.865**
	J48	0.720	0.625	0.556	0.588	0.765	0.813	0.788
	RF	0.800	0.833	0.556	0.667	0.789	0.938	0.857
	SVM	**0.8**	0.75	**0.667**	**0.706**	**0.824**	0.875	0.848
noc router	ANN	0.826	0.600	0.429	0.500	0.864	0.927	0.895
	NB	0.797	0.500	0.500	0.500	0.873	0.873	0.873
	J48	0.870	0.692	**0.643**	0.667	**0.911**	0.927	0.919
	RF	0.884	0.800	0.571	0.667	0.898	0.964	0.930
	SVM	**0.899**	**1.000**	0.500	**0.667**	0.887	**1.000**	**0.940**
openMSP430	ANN	0.538	0.000	0.000	0.000	0.538	1.000	0.700
	NB	0.923	1.000	0.833	0.909	0.875	1.000	0.933
	J48	0.769	0.714	0.833	0.769	0.833	0.714	0.769
	RF	0.923	1.000	0.833	0.909	0.875	1.000	0.933
	SVM	**0.923**	**1.000**	**0.833**	**0.909**	**0.875**	**1.000**	**0.933**
or1200	ANN	0.922	0.500	0.625	0.556	0.968	0.947	0.957
	NB	0.922	0.500	0.750	0.600	0.978	0.937	0.957
	J48	0.942	0.600	**0.750**	**0.667**	**0.978**	0.958	0.968
	RF	0.942	0.625	0.625	0.625	0.968	0.968	0.968
	SVM	**0.951**	**1.000**	0.375	0.545	0.950	**1.000**	**0.974**
vga_lcd	ANN	0.625	0.000	0.000	0.000	0.625	1.000	0.769
	NB	0.938	1.000	0.833	0.909	0.909	1.000	0.952
	J48	0.813	0.714	0.833	0.769	0.889	0.800	0.842
	RF	0.813	0.714	0.833	0.769	0.889	0.800	0.842
	SVM	**0.938**	**1.000**	**0.833**	**0.909**	**0.909**	**1.000**	**0.952**

TABLE VIII
SUMMARY OF REGRESSION RESULTS OF SVM FOR PROJECT *ethmac*

# of Bugs	# of Modules	Mean Absolute Error
4	1	0.813
3	1	0.756
2	3	0.679
1	4	0.552
0	16	0.380

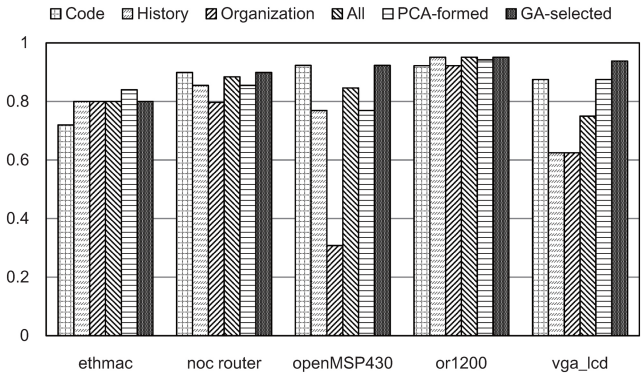


Fig. 6. Classification performances of bug models built from different characteristic types via SVM (with grid search to find the optimal parameters).

is that the bug models built with all characteristics often do not perform the best (the only exception here is the case of project or1200), which implies that all specified characteristics also may not be suitable for constructing bug models. The third observation is that for the evaluated projects, organization characteristics are not effective indicators of bug occurrence, as evidenced by the low prediction accuracy of corresponding bug models. A potential explanation is that extracted organization characteristics of different modules are very similar to each other because the entire project is contributed by only a few developers (e.g., one or two). In the future, we should evaluate the effectiveness of proposed organization characteristics on large-scale projects that are developed by many interactive developers. The fourth observation is that the bug models built with PCA-constructed characteristics could not always be better than those models constructed with all characteristics, which implies that PCA may not be a promising feature reduction technique for the evaluated projects. The fifth observation is that bug models constructed with GA-selected characteristics perform the best on four of five evaluated projects. Moreover, bug models constructed with GA-selected characteristics are better than those constructed with all characteristics. As a result, we could conclude that compared with the PCA, GA could select more appropriate characteristics for constructing accurate models.

V. CROSS-REVISION EVALUATION

A. Evaluation Results

In this section, we further conduct experiments to demonstrate that the bug models constructed at the reference revision could also be effective for predicting the bug information of the current revision, i.e., cross-revision bug prediction. Fig. 7 shows the workflow of the practical cross-revision bug prediction. In more detail, the module characteristics are extracted from the reference revision and the bugs, which are introduced before the reference revision but fixed within the time interval between the reference and the current revisions, are used to label each module at the reference revision. Then, bug predictive models could be built for predicting the bug

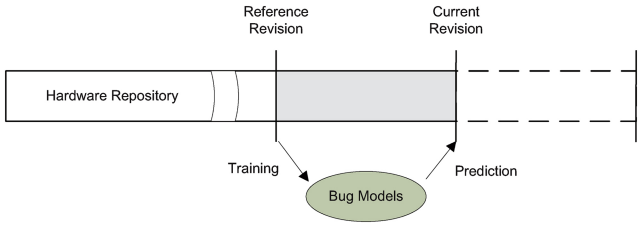


Fig. 7. Workflow of the practical cross-revision bug prediction.

TABLE IX
CURRENT REVISIONS FOR CROSS-REVISION VALIDATION

Name	Revision	Date	Time Interval
ethmac	325	2005-02-21	16 Months
noc router	1765	2010-02-27	5 Months
openMSP430	115	2011-05-30	7 Months
or1200	808	2012-05-27	10 Months
vga_lcd	55	2003-05-07	12 Months

information of the current revision. The current revisions used for validation in each project are listed in Table IX, where the time intervals between the reference revision and the current revisions are also given.

Fig. 8 compares the regression results of cross-revision bug prediction against the results obtained with intrarevision prediction, as shown in Fig. 5. The observation is that the proposed approach is, in general, effective for cross-revision prediction on the evaluated projects. Although on some projects, such as openMSP430 and vga_lcd, the cross-revision prediction is slightly less accurate than the intrarevision prediction, the cross-revision prediction can still achieve better performance on the project ethmac and or1200. Thus, the proposed approach could still be effective for cross-revision prediction for the evaluated projects.

As a project continues to evolve, the built bug models should be dynamically updated to reflect the status of the latest revision. Otherwise, the previously constructed models would perform badly since more bugs would be detected in recent revisions. In this case, such bug information should be used to relabel the modules in the reference revision to update the training data. Then, the data can be leveraged to build new bug models to accurately predict the bug information of the latest revision.

B. Illustrative Example

To elaborate the detailed process when applying the approach in practice, we give an example on building the classification model for the project ethmac.

At the reference revision, the original characteristics are extracted from the historical repositories (i.e., source code snapshots, changing logs, and bug repository) before this revision. Meanwhile, the bug information between the reference revision and the current revision is utilized to label whether each module is buggy and the number of bugs of each module in the reference revision. In this example, there are seven bugs

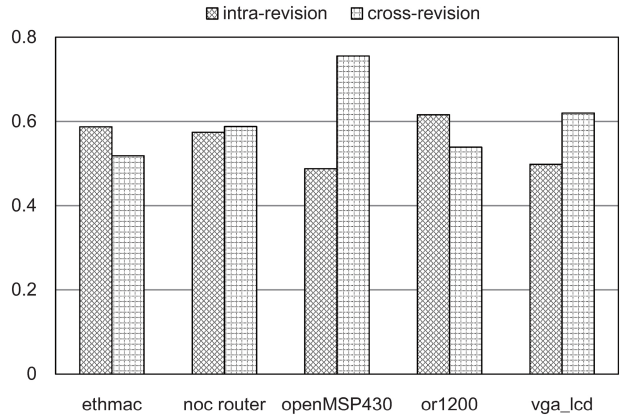


Fig. 8. Comparison of the regression results (RMSE, the smaller the better) of intrarevision and cross-revision bug prediction.

distributed in the module eth_spram_256x32, eth_rxethmac, eth_registers, eth_top, and eth_wishbone. In other words, there are five buggy modules and the rests are clean. Once the original training data are ready, we employ correlation-based GA to select the most informative characteristics from the original characteristics, and the selected characteristics are LOC, MaxLoCAdd, AvgLoCDel, and DevTot. Note that the characteristics selected with bug information from the reference revision to the current revision are different from the characteristics shown in Table V. Based on such characteristics, we then use grid search to find the appropriate parameters for training SVM. In this example, the near-optimal parameters are c=64.0, g=1.0. Then, we can train an SVM-based classification model based on these parameters.

At the current revision, the original characteristics are extracted from the historical repositories before this revision. Then, we only retain the characteristics that have already been determined by the GA-based feature selection techniques conducted at the reference revision, that is, LOC, MaxLoCAdd, AvgLoCDel, and DevTot. Such characteristics are directly leveraged by the previously constructed bug models to estimate the bug information of the current revision.

VI. BUG-ORIENTED METHODOLOGY FOR FUNCTIONAL VERIFICATION AND CODE REVIEW

In this section, we propose to leverage bug information predicted by the pre-silicon bug forecast framework to facilitate functional verification and code review. The basic idea is to use predicted bug information to efficiently allocate computational resources and manpower for different modules.

A. Bug-Oriented Verification

The pre-silicon bug forecast framework predicts bug information about each module of a design, which enables verifying hardware design from a bug-oriented perspective. To be specific, bug-oriented verification leverages predicted bug information to guide the resource allocation before the verification, monitor the verification process and define termination criterion near the end of verification.

1) During the creation of the verification plan, more verification resources should be invested to those prone-to-bug modules, i.e., those modules that are identified as buggy according to the classification models. Moreover, for those buggy modules, the regression model determines the verification priorities according to their potential bugs. For example, in the project ethmac, module eth_registers contains more bugs than module eth_crc based on predicted results (i.e., predicted bugs are 3.19 and 0.036 for eth_registers and eth_crc, respectively), and thus more resources should be invested consequently to eth_registers, e.g., we can constrain the random generator so that the difference of probabilities of hitting the coverage space of these two modules is proportional to the numbers of predicted bugs in these modules.
2) Bug information can also facilitate assessment of the completeness of the verification, and even helps select modules for formal verification. Taking or1200_alu component as an example, the coverage models of such functional units are hard to specify because of their large testing space. By employing the proposed approach, the total number of probable bugs in or1200_alu is 4.32 (rev. 600 on 2011-07-28), while the detected number of bugs was only 1 without report of new bugs in the later five months (rev. 672 on 2011-12-13). Apparently, we could not conclude that it approached the end of verification. However, in the next six months, the number of detected bugs was still far less than predicted value. In this case, formal method could be considered to expose potential counterexamples in the design to ensure its quality. Although the prediction is not completely accurate, our method can yet guide verification in a higher level compared with coverage model. Actually, a preliminary practice of our technique has already been employed in the verification of a commercial processor [17].
3) As verification proceeds, we use the bug detection rate, the proportion of exposed bugs against the total potential ones of a design, to evaluate whether all bugs of a module have been detected. After that, together with other constraints (e.g., functional coverage, structure coverage, and bug drop rate, etc.), we can determine whether verification on this module approximates the end phase. If so, we can concentrate on those complicated modules that still contain many unexposed hard-to-verify bugs.

B. Suspicious List for Code Review

The bug information predicted by the proposed pre-silicon bug forecast framework can also be used to improve the efficiency of code review, which is critical to guarantee the quality of a hardware design. A potential usage is to first conduct pre-silicon bug forecast on all modules, and then rank them based on the predicted numbers of bugs. Project managers can then decide to invest resources to the modules according to their ranks in the ranking list. For example, when the time-to-market budget is very limited, some modules ranked in the bottom of the list, i.e., the number of predicted bugs is less than 1, could be considered to escape from intensive verification. A problem of such ranking list is that it may vary significantly when time proceeds; thus, the bugs model should be rebuilt accordingly to offer accurate guidance for code review. Once the time budget and resources are abundant, we can simply use classification-based bug models to identify buggy modules and then conduct code review on all of them.

VII. Discussion and Extension

A. Threats to Validity

In practice, the industrial projects always contain much more bugs (e.g., more than 1000 bugs) than the evaluated projects. For example, according to the bug analysis of Intel Pentium 4 processor, which is one of the most representative industrial products, the number of detected bugs before releases is about 5809 [4]. Moreover, by analyzing its errata sheet, even after product release, there is still an average of 1.2 design bugs discovered per month [9]. However, codes of such industrial products are not available for us to conduct detailed evaluation. Although the evaluated projects are not as complex as such industrial processor designs, i.e., the numbers of bugs in evaluated projects are less than 100, we still believe that our approach may also be effective on large-scale projects, since the learning techniques considered in our study have been well developed, and would be more effective with large data sets [2], [12], [24].

B. Severity-Aware Bug Prediction

In our previous experiments, we did not consider the severity of each bug. In practice, major bugs always matter most for register transfer level (RTL) designers and verification engineers. Thus, it is very helpful to distinguish the bugs according to the severity of impacts. We may learn from the community of software engineering, where the severity of bugs has already been taken into account to build software defect models [8], [41]. The basic idea of these approaches is to use logistic regression or other machine learning techniques to characterize the relationship between the characteristics and ratings. Similarly, our approach can be naturally extended to achieve such severity-aware bug prediction. We elaborate the detailed process of severity-aware bug prediction as follows.

Following the specification of the most widely used bug tracking system as BugZilla, we can first roughly classify the bugs into six levels of severity, that is, blocker, critical, major, normal, minor, and trivial,[6] based on their severity. As a result, during the submission of a bug into the bug tracking system, the submitter labels the severity of this bug based on its impacts.

Then, during the model construction process, we should build multiple models for different severity of bugs. Taking classification model for blocker bugs as an example, the model can determine whether a module contains a blocker bug. To build such a model, each training module should be

[6]The detailed definition of each type could be found at http://bugzilla.mozilla.org/page.cgi?id=fields.html.

labeled to have a blocker bug or not. Thus, we can obtain six classification models for bugs at different severity levels. When predicting the bug severity levels for a new module, the characteristics of this module could be sent into all these bug models to identify at which severity level the bugs it contains. Similarly, it is also helpful to build a regression model for predicting bugs at each severity level. Once we can determine the number of bugs at different severity levels, the information could enable more flexible and effective resource allocation for functional verification and code review.

Nevertheless, for the evaluated designs, the information extracted from the SVN logs is very simple and relatively vague, which makes it hard to manually determine the severity level of each bug. In the future, we will investigate large-scale designs with well-maintained bug repositories, where the severity level of bugs is explicitly recorded.

C. Confidence on Predicted Number of Bugs

This paper mainly focuses on pre-silicon bug prediction of hardware designs. In practice, it is possible that the predicted number of bugs may not completely match the actual number of bugs, it would be very useful to offer confidence interval of each predicted result. Technically, confidence interval can be estimated from predicted results provided by multiple bug models. More specifically, multiple bug models (e.g., 10) can be built from different reference revisions. Then, the predicted results provided by these models can be used to estimate the confidence interval of the predicted number of bugs for each module in the current revision.

Once we can estimate the confidence interval of each predicted result, we could further improve the effectiveness of proposed bug-oriented verification methodology. For example, as verification proceeds, we can compare the number of detected bugs with the confidence interval of predicted number of bugs to assess the progress of verification. Only when the number of detected bugs exceeds the predicted number of bugs with a high confidence (e.g., 0.95), we could determine that the verification is comprehensive for this module.

VIII. Related Work

A. Complexity Characteristics and Bug Analysis

There are some studies focusing on the design complexity of high-level hardware designs, yet few of them systematically investigate how complicated characteristics of a hardware design influence the bug occurrence. Stollon *et al.* [34] proposed the measures of syntactic complexity, which was inspired by the structural and process similarities between VHDL description and software language. Protheroe *et al.* [28] provided some characteristics on RTL of a design to evaluate the design quality. Bazeghi *et al.* [3] studied characteristics of hardware description languages (HDL) code and results provided by design compiler to estimate design effort of microprocessors. There are also investigations on the bug analysis of hardware designs. Bentley [4] introduced the sources for bugs from the verification practices of Intel processor. Guo *et al.* [16], [17] studied the relationship between the complexity characteristics and bug occurrence. In these studies, many important characteristics (e.g., history characteristics) relating to bug occurrence have been overlooked, and it is still not clear how predicted bug information can facilitate not only functional verification but also code review.

B. Defect Prediction in Software Engineering

In the field of software engineering, many studies have been dedicated to characterize the relationship between the software characteristics and fault-proneness to assess the design quality [5], [19]–[22], [26], [27], [30], [31], [42], which mainly focused on selecting characteristics that have most impacts on the fault-proneness of software.

There are several key differences between the software studies and our work. First, our work is dedicated to hardware designs, which considers code characteristics that are only appropriate for HDL. Second, we propose to use correlation-based genetic algorithms [18] to select the most informative characteristics for model construction, which improves accuracies of bug models. Third, we propose the bug-oriented verification methodology that adopts pre-silicon bug forecast to facilitate functional verification, a critical step unique to designing hardware.

IX. Conclusion and Future Work

In this paper, we propose a pre-silicon bug forecast framework, which uses machine learning techniques to build bug models to characterize the relationship between the design characteristics (i.e., code, history, and organization characteristics) and bug-proneness of hardware designs, and we conduct detailed experiments on several open-source projects to demonstrate that such bug models could accurately predict the bug information of modules. Based on the predicted bug information, we further propose a bug-oriented methodology to facilitate the functional verification and code review process.

In our future work, there are still open problems to explore. For example, although we specify several characteristics that may correlate with bug occurrence, those characteristics still cannot cover all factors that are potentially related to bug occurrence. It is interesting to study whether adding other characteristics, e.g., code change complexity [19], can lead to more accurate pre-silicon bug forecast.

References

[1] C. Lewis and R. Ou, (2013, Apr. 1) [Online]. Available: http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html
[2] M. Banko and E. Brill, "Scaling to very very large corpora for natural language disambiguation," in *Proc. 39th ACL'01*, pp. 26–33.
[3] C. Bazeghi, F. J. Mesa-Martinez, and J. Renau, "Ucomplexity: Estimating processor design effort," in *Proc. 38th MICRO-38*, 2005, pp. 209–218.
[4] B. Bentley, "Validating the intel pentium 4 microprocessor," in *Proc. 38th DAC'01*, pp. 244–248.
[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code! Examining the effects of ownership on software quality," in *Proc. 8th Joint Meeting Eur. Software Eng. Conf. ACM SIGSOFT Symp. Found. Software Eng.*, 2011, pp. 4–14.
[6] L. Breiman, "Random forests," *Mach. Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[7] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, 2011.
[8] R. S. Chhillar and Nisha, "Empirical analysis of object-oriented design metrics for predicting high, medium and low severity faults using mallows Cp," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 6, pp. 1–9, Nov. 2011.
[9] K. Constantinides, O. Mutlu, and T. Austin, "Online design bug detection: RTL analysis, flexible mechanisms, and evaluation," in *Proc. MICRO'41*, 2008, pp. 282–293.
[10] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev, "Crane: Failure prediction, change analysis and test prioritization in practice: Experiences from windows," in *Proc. ICST'11*, pp. 357–366.
[11] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using Bayesian networks," in *Proc. 40th DAC'03*, pp. 286–291.
[12] A. Fuxman, A. Kannan, A. B. Goldberg, R. Agrawal, P. Tsaparas, and J. C. Shafer, "Improving classification accuracy using automatically extracted training data," in *Proc. KDD'09*, pp. 1145–1154.
[13] A. Gluska, "Coverage-oriented verification of banias," in *Proc. DAC'03*, pp. 280–285.
[14] A. Gluska, "Practical methods in coverage-oriented verification of the merom microprocessor," in *Proc. DAC'06*, pp. 332–337.
[15] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st ed. Reading, MA, USA: Addison-Wesley Longman, 1989.
[16] Q. Guo, T. Chen, H. Shen, and Y. Chen, "Estimating design quality of digital systems via machine learning," in *Proc. 17th ICECS'10*, pp. 623–626.
[17] Q. Guo, T. Chen, H. Shen, Y. Chen, Y. Wu, and W. Hu, "Empirical design bugs prediction for verification," in *Proc. DATE'11*, pp. 161–166.
[18] M. A. Hall, "Correlation-based feature selection for discrete and numeric class machine learning," in *Proc. 17th ICML'00*, pp. 359–366.
[19] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st ICSE'09*, pp. 78–88.
[20] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proc. 34th ICSE'12*, pp. 200–210.
[21] S. Kim, T. Zimmermann, E. J. Whitehead, Jr., and A. Zeller, "Predicting faults from cached history," in *Proc. 29th ICSE'07*, pp. 489–498.
[22] M. Kläs, F. Elberzhager, J. Münch, K. Hartjes, and O. von Graevemeyer, "Transparent combination of expert and measurement data for defect prediction: An industrial case study," in *Proc. 32nd ICSE'10*, pp. 119–128.
[23] D. Malandain, P. Palmen, M. Taylor, M. Aharoni, and Y. Arbetman, "An effective and flexible approach to functional verification of processor families," in *Proc. 7th HLDVT'02*, p. 93.
[24] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.
[25] T. M. Mitchell, *Machine Learning*, 1st ed. New York City, NY, USA: McGraw-Hill, 1997.
[26] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 30th ICSE'08*, pp. 181–190.
[27] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th ICSE'05*, pp. 284–292.
[28] D. Protheroe and F. Pessolano, "An objective measure of digital system design quality," in *Proc. 1st ISQED'00*, pp. 227–233.
[29] J. R. Quinlan, "Learning with continuous classes," in *Proc. Australian Joint Conf. Artif. Intell.*, 1992, pp. 343–348.
[30] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," in *Proc. 33rd ICSE'11*, pp. 491–500.
[31] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: An experimental approach," in *Proc. 30th ICSE'08*, pp. 341–350.
[32] F. Sebastiani, "Machine learning in automated text categorization," *ACM Comput. Survey*, vol. 34, no. 1, pp. 1–47, 2002.
[33] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. MSR'05*, pp. 1–5.
[34] N. S. Stollon and J. D. Provence, "Measures of syntactic complexity for modeling behavioral VHDL," in *Proc. 32nd DAC'95*, pp. 684–689.
[35] S. Sudakrishnan, J. Madhavan, E. J. Whitehead, Jr., and J. Renau, "Understanding bug fix patterns in verilog," in *Proc. MSR'08*, pp. 39–42.
[36] J. Van Hulse, T. M. Khoshgoftaar, and A. Napolitano, "Experimental perspectives on learning from imbalanced data," in *Proc. 24th ICML'07*, pp. 935–942.
[37] V. N. Vapnik, *The Nature of Statistical Learning Theory*. New York, NY, USA: Springer-Verlag, 1995.
[38] I. Wagner, V. Bertacco, and T. Austin, "Microprocessor verification via feedback-adjusted Markov models," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 26, no. 6, pp. 1126–1138, Jun. 2007.
[39] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. San Mateo, CA, USA: Morgan Kaufmann, 2005.
[40] D. H. Wolpert, "The lack of a priori distinctions between learning algorithms," *Neural Comput.*, vol. 8, no. 7, pp. 1341–1390, Oct. 1996.
[41] Y. Zhou and H. Leung, "Empirical analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Trans. Softw. Eng.*, vol. 32, no. 10, pp. 771–789, Oct. 2006.
[42] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data versus domain versus process," in *Proc. ESEC/FSE'09*, pp. 91–100.

Qi Guo received the B.S. degree in computer science from the Department of Computer Science and Technology, Tongji University, Shanghai, China, in 2007, and the Ph.D. degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2012.

His current research interests include computer architectures, very large scale integration design, and verification.

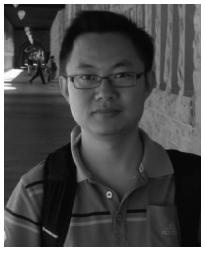
Tianshi Chen received the B.S. degree in mathematics from the Special Class for the Gifted Young, University of Science and Technology of China (USTC), Hefei, China, in 2005, and the Ph.D. degree in computer science from the Department of Computer Science and Technology, USTC, in 2010.

He is currently an Associate Professor with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His current research interests include computer architectures, parallel computing, and computational intelligence.

Dr. Chen was the recipient of the China Computer Federation Distinguished Doctoral Dissertation Award in 2011 and the Chinese Academy of Sciences Distinguished Doctoral Dissertation Award in 2011 for his Ph.D. work on computational complexity analysis of evolutionary algorithms.

Yunji Chen graduated from the Special Class for the Gifted Young, University of Science and Technology of China, Hefei, China, in 2002. He received the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences, Beijing, China, in 2007.

He is currently a Professor with ICT. His current research interests include parallel computing, microarchitectures, hardware verification, and computational intelligence. He has authored or co-authored one book and over 40 papers in these areas.



Rui Wang received the B.S. degree in computer science and technology from the Hefei University of Technology, Hefei, China, in 2008, and the Ph.D. degree in computer science from the Department of Computer Science and Technology, University of Science and Technology of China, Hefei, in 2013.

He is currently a Research Fellow with Anhui USTC iFLYTEK Company, Ltd., Heifei. His current research interests include machine learning and its applications in real-world problems.

Huanhuan Chen received the B.Sc. degree from the University of Science and Technology of China, Hefei, China, in 2004, and Ph.D. degree, sponsored by the Dorothy Hodgkin Postgraduate Award in computer science from the University of Birmingham, Birmingham, U.K., in 2008.

His current research interests include statistical machine learning, data mining, data fusion, and evolutionary computation.

Dr. Chen's Ph.D. thesis "Diversity and Regularization in Neural Network Ensembles" received the 2011 IEEE Computational Intelligence Society Outstanding Ph.D. Dissertation Award (the only winner) and the 2009 CPHC/British Computer Society Distinguished Dissertations Award (the runner up). His work "Probabilistic Classification Vector Machines" on Bayesian machine learning received the the IEEE TRANSACTIONS ON NEURAL NETWORKS Outstanding Paper Award (bestowed in 2012, and only one paper in 2009 received this award).

Weiwu Hu received the Ph.D. degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

He is currently a Professor of computer science with the Institute of Computing Technology, Chinese Academy of Sciences. His current research interests include high-performance computer architectures, parallel processing, and very large scale integration design.

Guoliang Chen received the B.S. degree from Xi'an Jiaotong University, Xi'an, China, in 1961.

Since 1973, he has been with the University of Science and Technology of China, Hefei, China, where he is currently a Professor with the School of Computer Science and Technology. From 1981 to 1983, he was a Visiting Scholar with Purdue University, West Lafayette, IN, USA. He is currently also the Director of the National High Performance Computing Center, Hefei, China. He has published nine books and more than 200 research papers. His current research interests include parallel algorithms, computer architectures, computer networks, and computational intelligence.

Prof. Chen is an Academician of the Chinese Academy of Sciences. He was the recipient of the National Excellent Teaching Award of China in 2003.