VISUALIZING HDL CODE EVOLUTION

Thiago S. F. Silva [1], *André S. F. Silva* [1], *José Augusto Nacif* [1,2]
Luiz F. M. Vieira [1], *Antônio O. Fernandes* [1]
[1] Computer Science Department, Universidade Federal de Minas Gerais
[2] Universidade Federal de Viçosa
{thiagosf,andresfs,jnacif,lfvieira,otavio}@dcc.ufmg.br

ABSTRACT

Verifying industrial scale designs is a challenging task that consumes a lot of development efforts. In order to handle this increasing design complexity, improvements in current verification methodologies are needed. A great contribution to digital integrated circuits development is the possibility to visualize HDL code evolution. In software engineering, code evolution visualization is a well known practice. In that area, there are many researches that present visualization methodologies and illustrate source code metrics through a timeline. We propose visualization mechanisms to facilitate the HDL code evolution analysis. Collecting data from versioning systems and error reports makes it possible to extract design improvements information and print bug reports. This problem is challenging because we are dealing with large data volume, about all design stages. In order to support HDL code evolution, we present visualization methodologies that will contribute for a better productivity on hardware projects.

1. INTRODUCTION

The design and verification stages are complex tasks during the development process of digital integrated circuits. As new VLSI technologies arrive at market, more complex chips are developed. In order to handle this complexity growth, it is also needed to increase hardware engineering teams productivity.

The source code evolution analysis term comes from software engineering. Through versioning control systems (VCS) and documented error reports from bug tracking systems (BTS), data about all software design steps are stored. The great volume of data is relevant to many research topics. On Software Engineering area, data extraction mechanisms and visualization methodologies were presented in order to facilitate inferences about development processes. With the support of these tools, different software aspects can be analyzed.

The technology needed to track hardware evolution information, based on revision history is the same used for software evolution. Despite this, collecting hardware specific metrics needs modifications. On hardware engineering research, the term hardware evolution is not used yet. However the analysis of these data has shown capable to offer a great potential to be explored [1]. In order to support HDL code evolution, we present visualization methodologies that will contribute for a better productivity on hardware projects.

This paper is outlined as follows. Section 2 describes related work on tracking software evolution and its visualization mechanisms. In Section 3, we present aspects to be analyzed about hardware evolution. Section 4 shows evolutionary visualization mechanisms. Finally, Section 5 concludes the paper.

2. RELATED WORK

There is a well known research area on Software Engineering that studies evolution analysis of source code. Tools, metrics, and visualization methodologies were found on recent researches [2]. The major motivation of this study area are the improvements on development processes. It is expected that through historic analysis, it will be possible to estimate, predict, or just known specific details about a developed object.

Popular papers on software evolution are CVSAnalY [3] and RHDB [4]. The former presents different analysis that can be done only from VCS data. In order to build a release history database (RHDB), the latter joined bug collected from BTS with VCS data, establishing links between modification reports from VCS, with problem reports from BTS. To track code structures evolution, this structures were statically retrieved from source code by a parser.

Another software source code static analysis tool was combined with VCS mining on [5]. On their case study, every registered modification on analyzed softwares were retrieved, and error metrics were statically extracted for each one.

The most complete tools found to track software evolution are Alitheia [6] and Kenyon [7]. They provide connectors to most common VCS and BTS tools, software metrics, and perform all data extraction. The collected data are stored on a database for future analysis.

Only to present evolution data, there is a specific research area. There are many papers that deals with the most different aspects that can be observed, since the modifica-

tions occurring at each code segment or structure, until specific statistic data, collected during the timeline.

RelVis [8] is an approach that presents Kiviat diagram, also known as radar chart, as a good mechanism to show multiple metrics. They also used multiple charts linked by edges representing couple among software modules.

Voinea presents a different approach, showing evolution inside code structures. CVSgrab and CVSscan [9] were built to extract data and present code change history.

3. ASPECTS TO BE ANALYZED

In general, any development aspect could be analyzed, resulting in qualitative information, based on quantitative metrics. As quantitative metrics, we present data extracted from static analysis of HDL code (3.1), versioning control systems (3.2), and error reports (3.3).

3.1. HDL code static analysis

The static analysis from HDL code differs from dynamic analysis because there is no simulation, only code structures are observed. Complexity data extracted from HDL statically are metrics like: lines of code (LoC), number of I/O signals, operands, operators, logic gates, and flip-flop/latch registers. There are also metrics used in software engineering like ciclomatic complexity [10] and number of acyclic paths [11].

The visualization of complexity metrics changing in time is useful for developers that want to follow their development processes. Metrics extracted in a timeline can be correlated with error report data. The analysis of values varying may point quality aspects and help on inference learning.

3.2. Versioning control tools

Versioning control systems make possible to work on all HDL code evolution perspectives. Usually, the data are organized by time, according with commit sequence. Each commit represents a new source version because one or more files were modified by a developer. Data about which files were modified, who did the modifications, and commit messages are stored.

When there is a commit sequence about the same modification context, it is possible to group them and label as the same transaction. So, it is possible to change the visualization perspective from commit by commit to commit per modification motivation. The commit data can also be grouped by developer.

3.3. Error reports

Versioning control systems can be used with error documentation tools, known as bug tracking systems. They store important statistics about modifications and errors found, that can be correlated with HDL versions. Data about which error classifications, when they were resolved, and who were the involved developers are obtained from these tools.

Since it is common to development teams spend great part of their efforts on verification step, only error information is interesting to be observed.

4. RESULTS

The visualization of HDL code evolving in time is done by simple charts as bar, line and pie. We have also found in software engineering research, works presenting interconnected geometric forms like graphs, representing connections among code structures [12]. Another interesting chart is the heatmap that represent proportion and intensity.

The metric evolution chart, illustrated by Figure 1 presents multiple metric values changing as new HDL versions are being committed. On Figure 2 the radar chart also presents multiple metric perspective, but with a fragmented timeline. The latter is good to show the variation of multiple metrics at specific time intervals.

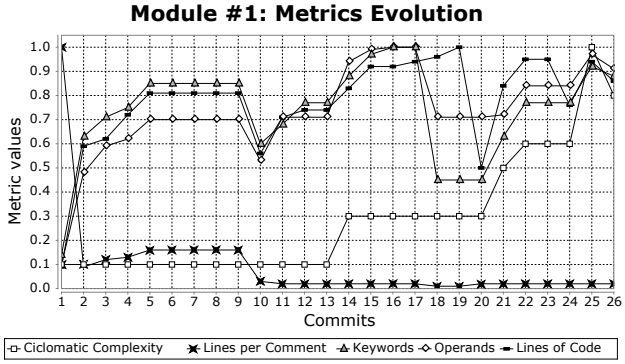

Figure 1. Metric evolution chart

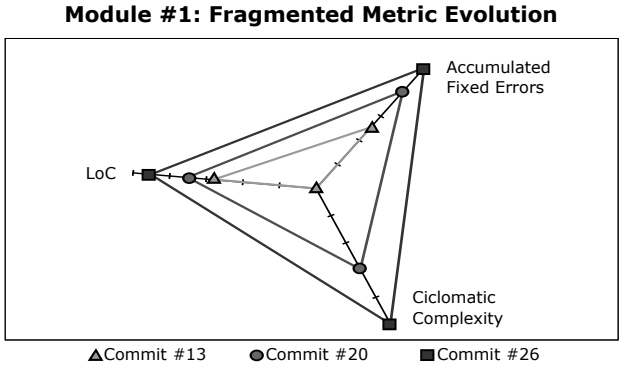

Figure 2. Radar chart

Collected errors from BTS tools are illustrated by the bar chart on Figure 3. The errors were grouped bi-weekly.

The heatmap chart, illustrated by Figure 4 depicts an intellectual property core with 17 modules. The rectangle area is used to represent measures that can be presented proportionally. The colors are used to distinguish aspects

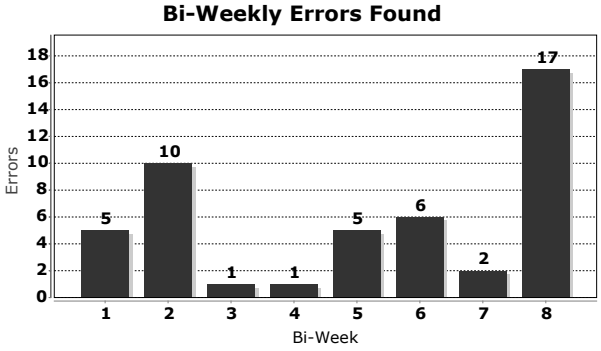

Figure 3. Bugs in time

about each object. On this example, we represent LoC size and use dark gray for the three most frequently fixed modules.

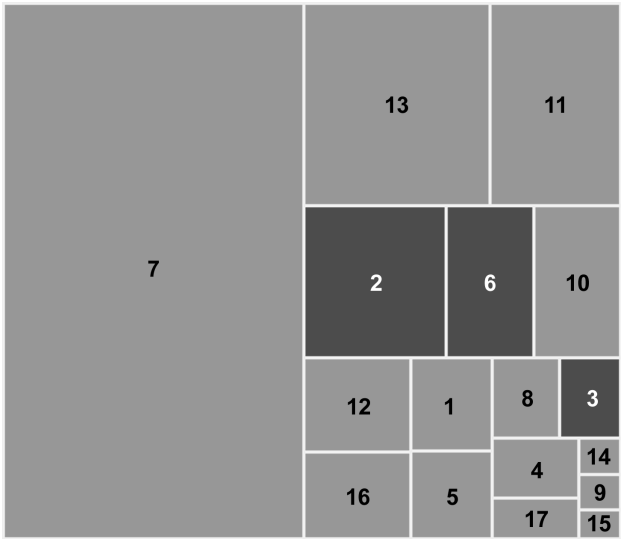

Figure 4. Heatmap chart

5. CONCLUSION AND FUTURE WORK

The visualization of hardware evolution statistics can widely collaborate on HDL design and verification stages. We presented the relevance of using VCS and BTS tools, that store historical data about all design process. We have also shown charts illustrating metrics evolving on time, errors reported on BTS and a heatmap that could point desired aspects about a designed IP core.

As future work we intend to design and implement a tool capable of automatically tracking VCS and BTS data. The collected data should be stored on a database. It is also interesting to facilitate chart generation.

6. REFERENCES

[1] J. Nacif, T. Silva, A. Tavares, A. Fernandes, and C. Coelho, "Efficient allocation of verification resources using revision history information," in *Proceedings of the 2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pp. 1–5, IEEE Computer Society, 2008.

[2] G. Gousios, *Tools and Methods for Large Scale Software Engineering Research.* PhD thesis, Athens University of Economics and Business, jul 2009.

[3] G. Robles, S. Koch, and J. M. Gonzalez-Barahona, "Remote analysis and measurement of libre software systems by means of the CVSAnalY tool," in *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, (Edinburg, Scotland, UK), 2004.

[4] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, (Washington, DC, USA), p. 90, IEEE Computer Society, 2003.

[5] C. Williams and J. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, 2005.

[6] G. Gousios and D. Spinellis, "Alitheia core: An extensible software quality monitoring platform," *Software Engineering, International Conference on*, vol. 0, pp. 579–582, 2009.

[7] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating software evolution research with kenyon," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, (New York, NY, USA), pp. 177–186, ACM, 2005.

[8] M. Pinzger, H. Gall, M. Fischer, and M. Lanza, "Visualizing multiple evolution metrics," in *Proceedings of the 2005 ACM symposium on Software visualization*, pp. 67–75, ACM, 2005.

[9] L. Voinea, A. Telea, and J. van Wijk, "CVSscan: visualization of code evolution," in *Proceedings of the 2005 ACM symposium on Software visualization*, pp. 47–56, ACM, 2005.

[10] T. J. McCabe, "A complexity measure," in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, (Los Alamitos, CA, USA), p. 407, IEEE Computer Society Press, 1976.

[11] B. A. Nejmeh, "Npath: a measure of execution path complexity and its applications," *Commun. ACM*, vol. 31, no. 2, pp. 188–200, 1988.

[12] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proceedings of the 2003 ACM symposium on Software visualization*, ACM, 2003.