An Open Source Code Base for Digital Circuit Analysis, Simulation, and Modification

Spencer Millican
Auburn University
millican@auburn.edu

Abstract – Many research projects in the digital EDA community perform common circuit manipulation and simulation tasks. To perform these tasks, current projects must either rely on industry tools, which are either expensive or cannot be modified to fulfill project needs, or they must create new tools from scratch, which is an arduous process which takes away efforts from more productive tasks. Members of the EDA community may also lack a high-performance programming background, which limits the quality of tools and results produced. This article introduces an ongoing project to create open-source circuit analysis, manipulation, and simulation programs, whose end goal is to produce mature, easy-to-modify source code for the EDA community's benefit.

I. Introduction

The digital electronic design automation (EDA) community has many algorithms which require the analysis, manipulation, and simulation of circuits. These algorithms find, extract, or calculate values from a circuit, manipulate the circuit based off of these values, and then observe if the circuit manipulation produced the desired effect. Examples of such programs include test point insertion (TPI), logic synthesis, and redundancy removal. The list of such algorithms continues to grow as complex circuits introduce new design constrains, e.g. power constraints.

Although industrial EDA tools are sometimes available to the research community, these tools are not always viable for research purposes. Licensing requirements of industrial EDA tools may be unobtainable to particular research institutions given financial and non-disclosure constraints. Managing these licenses also imposes an administrative burden on researchers which takes resources away from their projects. Given the purpose of research is to improve upon previous methods, the use of industrial tools is not viable unless the source code for such tools is available, which is unlikely given the proprietary nature of such tools.

Without modifiable industrial-grade tools, original programs must be created for each digital EDA project, but this presents several difficulties to researchers. First, performing high-quality research requires specialization in the topic to be researched, and simultaneously specializing in high-performance programing may not be feasible for many researchers. Second, creating new programs requires considerable time and effort from researchers which can be better utilized elsewhere. Third, the quality of produced programs must be comparable against available tools in order to achieve meaningful comparisons and making such quality programs requires significant effort and resources.

This article presents an ongoing effort to create a circuit analysis, simulation, and manipulation code base. The first goal of this project is to create stable, reliable, and easy-to-use code such that EDA researchers can incorporate their needs through minimal code modifications. The second goal of this project is to implement high-performance implementation of several common circuit analysis methods which are comparable to techniques used in industry. The third goal of this project is to solicit feedback from the EDA community on how this code can be written to best benefit their research needs.

II. Project History

Initial versions of the code base was developed for a study on obfuscated and simulatable digital circuits [1]. This study required performing several functions which are common for binary circuits but which have yet to be done for non-binary discrete circuits. This included non-binary simulation, satisfiability (SAT) through a PODEM-like algorithm [2], the transformation of a circuit from binary to non-binary, and the attempted decryption of obfuscated circuits. This code was initially implemented in a scripting language (i.e., Perl) due to its ease-of-use, but initial results found this to be

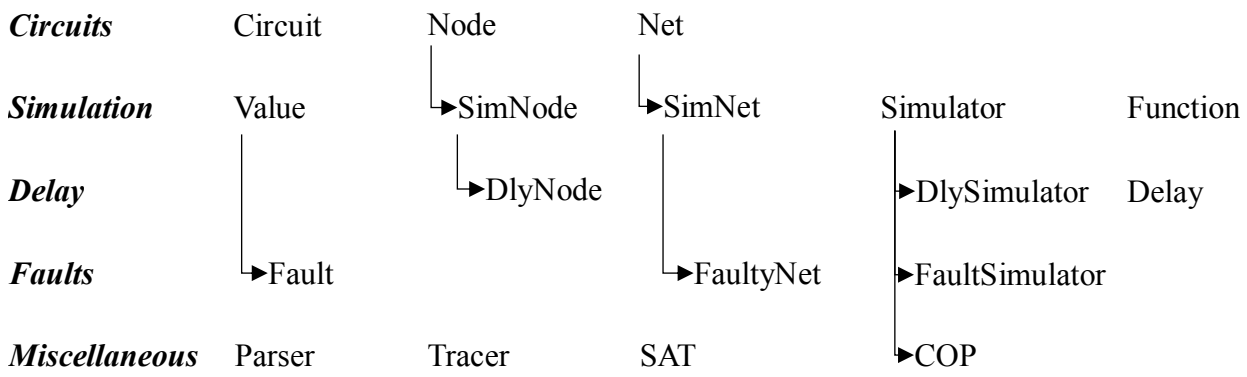

Figure 1: Each package (left, ***Bold Italics***) contains several code objects (*Italics*). Some package objects directly inherit from objects in other packages.

unacceptable due to infeasibly long execution times. The code was then re-written in C++, which provided significant performance improvements.

When later improvements were proposed to the original circuit obfuscated method, implementing these changes was found to be difficult due to the structure of the code. First, many standard C++ practices were ignored, and standard libraries were not utilized. This made expanding the code prone to errors, and debugging errors was found to be time consuming. Second, the structure of the code was nonintuitive, which made modifications to the code difficult to comprehend and hindered development time.

Deficiencies in the code led to several rewrites, and the code is now being used for projects beyond its original intention. As objects in the code were made more generic, implementing new methods based on existing objects became streamlined. For example, implementing 3-value (0,1, and X) fault simulation required minimal changes to binary simulation after the concept of a "value" was made generic, and likewise, such "values" could be made faulty. Modifications like this allowed the code to be applied to several projects involving design-for-test practices, such as using artificial neural networks (ANNs) to evaluate circuit testability [3].

III. Code Implementation

The code is implemented in the 2011 standard for C++ (i.e., C++11). This programming language was chosen for its frequent use in practice, its high-performance compiler support, and for its numerable object-oriented programming features. C++ was also chosen because many high-performance computing standards, e.g. OpenMP and CUDA, are implemented using C++ extensions. An older version of the standard (i.e., besides C++14 or C++17) was chosen for its availability at the time the original code was written and its robustness.

IV. Code Structure

Currently, code is divided into several "packages" where every package provides capabilities to implement one or more features. This allows code complexity to be reduced by excluding packages from a project which do not contribute required features. New packages can be created from existing ones through inheritance, which allows for faster and simpler code development.

The current packages and the features they provide are as follows. Packages are represented in ***Bold Italics*** while specific classes of the current code are represented in *Italics*. The relationship between package objects (i.e., inheritance) is illustrated in Figure 1.

A. Circuits

This is the "base package" which contains objects to represent a directed graph. A circuit is represented using single-output *Nodes* and single-input *Nets* in a *Circuit* container. This package by itself does not provide the ability to simulate values on a *Circuit*, but this can be done using the ***Simulation*** package. However, this package can be useful with modeling a directed graph, which can be useful for many EDA projects.

B. Simulation

This package adds features to *Nets* and *Nodes* to allow for binary, X-value, and non-binary simulation. This is accomplished by allowing *Nets* to hold *Values* and by giving each *Node* a *Function*. A *Simulator* can apply a vector of input *Values* to a *Circuit* and obtain the corresponding vector of output *Values* from the *Circuit*. Currently, a standard set of Boolean *Function* objects is provided, but the *Function* class can be expanded to implement arbitrary truth tables or to use non-binary *Values*.

C. Delay

This package adds the capability to model signal propagation delay in digital circuits. This is done by adding a *Delay* object to every *Node*, which creates a new *DlyNode* object. The Simulator object is updated to create a new *DlySimulator* object, which can calculate the signal arrival time on circuit outputs when a given input vector of *Values* is provided.

Currently, this package implements the controlling input delay model [4], but the delay model can be changed by replacing the appropriate functions of the *Delay* object.

D. Faults

This package adds fault simulation capabilities. This is accomplished by adding a *Fault* object, which can activate a stuck-at-0 or stuck-at-1 fault to a *Net*, which in turn creates a new object, the *FaultyNet*. A new *FaultSimulator* object performs simulation much like a normal *Simulator*, except for every vector of input *Values* applied it will iteratively activate every fault to observe if any output of the circuit is changed. Doing so will mark off detected faults and give a final *Circuit* fault coverage.

E. Miscellaneous

This package contains objects which thus far do not warrant their own package.

1. *Parser*: This object converts a text file netlist into a simulatable *Circuit*. If delay information is provided, *DlyNodes* will be created.
2. *Tracer*: This object can trace forwards and backwards in a *Circuit* starting at a given *Net* or *Node*.
3. *SAT*: This object performs satisfiability (SAT) [5] on a given set of *SimNets* by trying to set them to a given set of *Values* by setting *Values* on *Circuit* inputs.
4. *COP*: This object calculates controllability and observability measures on *SimNets* using the COP algorithm [6].

V. Future Directions

The code base presented here has the potential to allow for faster and more efficient development of research projects which require the analysis, simulation, and modification of digital circuits. However, some issues must be addressed to assure quality results, and feedback is needed from the EDA community to fulfill their needs.

A. New Packages

Many researchers will require algorithms which are not currently implemented in the code base. Although users are free to modify the current code to implement such methods, the value of the code will be greatly increased if such methods are already implemented in a standardized way. Such improvements may include power simulation, test point insertion, sub-circuit support, and standardized input/output waveform support.

B. High-performance implementation

To best support researchers, the performance of implemented algorithms must be comparable to those in industry. This is necessary for a fair comparison against industry tools. This will also give researchers confidence in the performance of their results.

Although the current tools are implemented in a modern programming language and compiled with maximum effort, the algorithms used must be updated to best match methods in literature. For instance, SAT is currently implemented in a manner resembling the PODEM ATPG algorithm [2], but other methods exist which can undoubtedly perform better. Also, parallel computing resources are not currently leveraged.

C. Code robustness

For any code to be used by a wider community, it must be tested thoroughly and conform to the standards and practices of the times. Although the current code has been used across many projects, it must be updated to conform to these practices. These practices include using standardized unit tests and using standardized documentation formats.

VI. Conclusions

When future plans are implemented, this code base will provide researchers with alternatives to industrial tools and allow researchers to focus their efforts on higher-quality projects. With time, researchers can contribute their methods to the code base, allowing for faster development of further projects to further propel their results.

The author openly seeks feedback from the EDA community on their needs and desires from such a code base. With such feedback, the author hopes this code base will be most useful to the EDA community for years to come.

VII. References

[1] S. Millican, P. Ramanathan, and K. Saluja, "CryptIP: An Approach for Encrypting Intellectual Property Cores with Simulation Capabilities," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, 2014, pp. 92–97.

[2] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. C–30, no. 3, pp. 215–222, Mar. 1981.

[3] S. Millican, "VLSI Test Lab." [Online]. Available: http://www.eng.auburn.edu/ece/faculty/skm0049/vlsi/index.html.

[4] P. Ramanathan and K. K. Saluja, "Crypt-Delay: Encrypting IP Cores with Capabilities for Gate-level Logic and Delay Simulations," in *2016 IEEE 25th Asian Test Symposium (ATS)*, 2016, pp. 7–12.

[5] R. Impagliazzo and R. Paturi, "On the Complexity of k-SAT," *Journal of Computer and System Sciences*, vol. 62, no. 2, pp. 367–375, Mar. 2001.

[6] F. Brglez, "On testability analysis of combinational networks," *Proceedings - IEEE International Symposium on Circuits and Systems*, vol. 1, Jan. 1984.