社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

Hardware Description Languageにおけるコードクローンのパターン分類

上村　恭平†　藤原　賢二†　飯田　元†

† 奈良先端科学技術大学院大学 情報科学研究科　〒630-0192 奈良県生駒市高山町 8916-5
E-mail: †{uemura.kyohei.ub9,kenji-f}@is.naist.jp, ††iida@itc.naist.jp

あらまし　近年，回路規模の増大と複雑化に伴い，回路の開発に Hardware Description Language (HDL) が広く用いられている．ソフトウェア開発においては，ソースコード中のコードクローンはソフトウェアの品質に悪影響を与えるとされ，コードクローンの検出手法や，その影響に関する調査などが盛んに研究されている．しかし，回路開発における HDL のコード中のコードクローンの影響は明らかにされていない．そこで，本研究では HDL におけるコードクローンに着目し，検出ツールを開発した．開発した検出ツールを用いて，OpenCores が提供するプロジェクトを対象にコードクローンの検出を行い，その結果を基に HDL におけるコードクローンを 4 つのパターンに分類した．
キーワード　コードクローン, コードクローン検出, Hardware Description Language, Verilog, 抽象構文木

Classification of Code Clones in Hardware Description Language

Kyohei UEMURA†, Kenji FUJIWARA†, and Hajimu IIDA†

† Graduate School of Information Science, Nara Institute of Science and Technology
Takayama-cho 8916-5, Ikoma-shi, Nara, 630-0192 Japan
E-mail: †{uemura.kyohei.ub9,kenji-f}@is.naist.jp, ††iida@itc.naist.jp

Abstract　With the growth of the electric circuit size and complexity, Hardware Description Language (HDL) is widely used in hardware development. Since code clones in the source code are said to have bad effect on software quality, developing code clone detection algorithms and investigating the effect of code clone on software quality are becoming research targets. However, code clones in HDL code have not been investigated enough yet. In this research, we developed a code clone detection tool for HDL and detected code clones in the HDL projects provided by Open Cores. As a result of our investigation, we classified code clones in HDL code into four patterns.
Key words　Code Clone, Clone Detection, Hardware Description Language, Verilog, Abstract Syntax Tree

1. はじめに

近年，電子回路の開発において Hardware Description Language (HDL) が広く用いられている．回路開発規模が大きくなり，複雑になるにつれ，検証やデバッグにかける時間が増大し，今後の回路開発のボトルネックになるのは検証やバグ修正だとも言われている．そのため，バグを含むモジュールの推定など，検証作業の補助による開発の効率化が求められている．ソフトウェア開発においては，ソースコードの解析技術が広く活用されており，バグ予測や，ソフトウェア品質を計測するためのメトリクスの提案，コードクローンの検出などの研究が盛んに行われている．コードクローンはソフトウェアの品質を低下させると言われており，コードクローンの修正支援手法などが提案されている．HDL のコードにもソフトウェアと同様にコードクローンが存在し，品質に影響を与えている可能性があるが，その実態は明らかにされていない．そこで，本研究では代表的な HDL である Verilog を対象としたコードクローン検出ツールを開発し，HDL におけるコードクローンについて調査を行った．

267 件のプロジェクトに対し開発したツールでコードクローンの検出を行ったところ，598 件のクローンセット候補が検出された．クローンセット候補から 100 件をランダム抽出し，目視によるコードクローンの判定を行った結果，20 件のコードクローンが確認された．それぞれのコードクローンを分析した結果，HDL におけるコードクローンのパターンを 4 種類に分類することができた．

以降，2 章ではコードクローンと HDL についての既存研究を踏まえて回路開発の問題点について述べる．3 章ではコードクローンの検出手法について説明する．開発したツールを用いたコードクローンの検出結果を 4 章で示し，5 章では発見したコードクローンをいくつかのパターンに分類し分析した結果と，コードクローンの検出精度についての考察および妥当性への脅

威についてまとめる．6 章では本稿のまとめについて述べる．

2. 背　　景

2.1 コードクローン

ソフトウェアのソースコード中に存在する，互いに一致する，あるいは類似するコードの断片をコードクローンという．コードクローンは，主に別のプログラムへの同一の処理や，一部挙動の異なる処理の追加を目的に，開発者がソースコードの断片をコピーし，貼り付けることで作成される．コピー元のソースコードの断片と，貼り付けられたソースコードの断片のペアはコードクローンと呼ばれる．Bellon らは，コードクローンを類似の度合いにより 3 つのタイプに分類している [1].

[**タイプ** 1]　空白やタブの有無，括弧の位置などのコーディングスタイルを除いて完全に一致するコードクローン．

[**タイプ** 2]　変数名や関数名などのユーザ定義名，変数の型などの一部の予約語のみが異なるコードクローン．

[**タイプ** 3]　タイプ 2 の差異に加え，文の挿入や削除，変更が行われたコードクローン．

ソースコード中のコードクローンの存在は，ソフトウェアの品質に悪影響を及ぼすといわれている [2]．例えば，コードクローンの元となるソースコードの断片にバグが含まれると，コピー先にも同様のバグを埋め込むことになる．また，ソースコードに変更を加える場合，対応するクローンペアの全てに同様の変更が必要かを判断し，適切な箇所に変更を加える必要がある．このような問題を解決するために，コードクローンの検出や，検出したコードクローンの変更支援，コードクローンを除去するためのリファクタリング支援などの研究が行われている．その他，コードクローンの含有量などをメトリクスとして活用し，品質や開発コストの予測，評価を行う研究が多数行われている．

2.2 Hardware Description Language

Hardware Description Language (HDL) は電子回路の状態や振る舞い，構造を定義するための言語である．代表的な HDL として Verilog や VHDL が広く使われている．HDL を用いて定義した電子回路は Application Specific Integrated Circuit (ASIC) として作成されるほか，Field-Programmable Gate Array (FPGA) を用いた回路開発に用いられる．HDL による電子回路の開発はプログラミング言語によるソフトウェア開発と同様にソースコードを記述することで行うため，類似した点が多い．ソフトウェア開発ではオープンソースソフトウェアとしてソースコードが公開されているものがあるのと同様に，電子回路開発においても，オープンソースの開発プロジェクトが存在する．OpenCores [3] 等の Web サイトで，一定の機能をまとめた回路である IP コア (Intellectual Property Core) のソースコードや開発履歴が公開されている．ソフトウェアのソースコードと電子回路のソースコードの大きな違いは，処理の実行順序にある．C 言語や Java 言語などの一般的なプログラミング言語においては，ある関数内の処理は記述した順に上から実行される．しかし，HDL において記述順は処理の順序に影響しない．例えば，リスト 1，リスト 2 にの 5 行目から始まる

リスト 1　sample1.v

```
1  module sample(clk, rst, a);
2    input clk, rst, a;
3    reg b, c;
4    output d;
5    always @(posedge clk or
         posedge rst)
6    begin
7      b <= a;
8      c <= b;
9    end
10   assign d = c;
11 endmodule
```

リスト 2　sample2.v

```
1  module sample(clk, rst, a);
2    input clk, rst, a;
3    reg b, c;
4    output d;
5    always @(posedge clk or
         posedge rst)
6    begin
7      c <= b;
8      b <= a;
9    end
10   assign d = c;
11 endmodule
```

always ブロックは，信号の状態を条件として非同期に処理されるブロックである．また，always ブロックの中で記述された 7, 8 行目の Nonblocking 代入文は 1 クロックの中で同時に処理される回路を生成する命令文である．そのため，リスト 1，リスト 2 に示すソースコードは 7 行目と 8 行目が入れ替わっているが，どちらのソースコードでも，clk，rst 信号のどちらか入力された際に出力 d に b の値が設定される同一の回路が定義される．

回路規模の増大に伴い，HDL のコードも複雑なものになり，検証やデバッグにかかる時間が増大している．そのため，効率よく検証やデバッグを行うための研究が多数行われている．Parizy らは，バグの含まれるモジュールを予測することで，検証にかかる時間を短縮できるという報告をしている [4]．Nacif らは，回路開発の品質管理を行うためのメトリクスを記録，可視化するフレームワークである eyesOn を開発し，ケーススタディを通して回路開発においてもソフトウェア開発と同様，メトリクスを用いた開発支援が有効であることを示している [5]．eyesOn は LoC や，レジスタやゲート，信号線の数をメトリクスとして採用しており，コードクローンは対象としていない．従来の研究においては，HDL にどの程度の量のコードクローンが存在するかどうかは明らかにされておらず，その影響についても不確かである．

3. コードクローンの検出

ここでは，本研究で開発した HDL を対象としたコードクローン検出ツールと，そのアルゴリズムについて説明する．まず，第 1 節にて既存のソフトウェアを対象としたコードクローンの検出手法について紹介する．続いて，第 2 節では HDL におけるコードクローンを検出するための提案手法について述べる．

3.1 既 存 手 法

ソフトウェアを対象としたコードクローンの検出手法は多数提案されている．ソースコードの構造に着目した検出手法としては，文字列や行の並びが一定以上の長さで一致しているものを検出する手法，トークン化した文字列の並びが一致するものを検出する手法，ソースコードを AST(抽象構文木) で表現し，類似する部分木を検出する手法などがある [6]〜[8]．この他，データの流れやモジュールの構造などの，意味的な違い

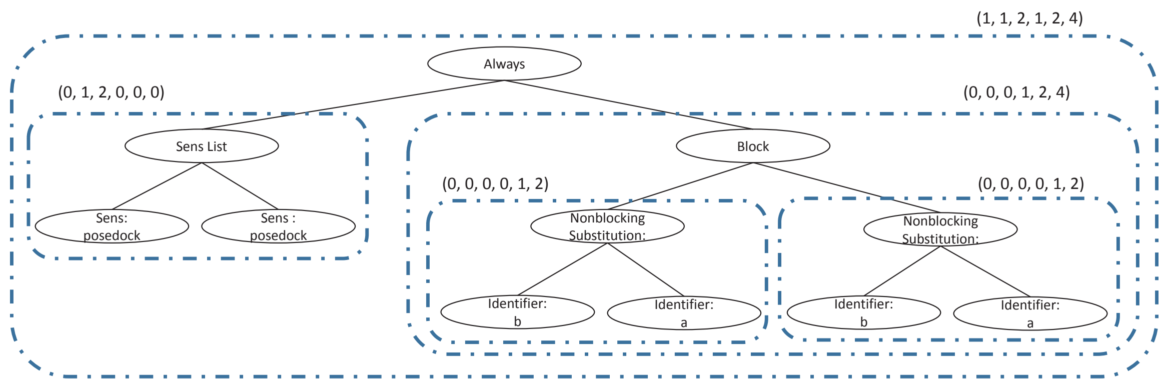

図 1　AST とベクトルの例 (図中のベクトルは構文ベクトルであり，前から順に Always, SenceList, Sens, Block, NonblockingSubstitution, Identifier の個数を意味する)

に着目したクローン検出手法も提案されている．Jiang らの開発したコードクローン検出ツール DECKARD では AST に含まれる構文要素の数から，部分木毎に構文ベクトルを作り，安定分布による Locality Sensitive Hashing (LSH) を用いてタイプ 1，タイプ 2，タイプ 3 のコードクローンの探索を行う [8]．LSH では類似するベクトルが高確率で同一のハッシュ値をとるため，同一のハッシュ値を持つ AST の部分木のペアはクローンペアの候補となる．このハッシュ値は，絶対値がベクトルの大きさに正比例する．

3.2　提 案 手 法

HDL は文法上，実際の振る舞いの順序と記述の順序が異なる特徴を持つことを考慮し，今回開発したツールでは DECKARD のアルゴリズムを参考にした，AST から作成される構文ベクトルから計算される LSH のハッシュ値を用いる手法を採用する．この手法では構文ベクトルが類似するものをクローンペアの候補とするため，記述の順序に影響を受けることがなく，タイプ 1，タイプ 2，タイプ 3 のコードクローンを検出することができる．提案手法ではまず，探索対象のファイルを AST 表現に変換する．続いて，葉の方向から根の方向へ順に，子ノードが持つ各構文要素を数えることで構文ベクトルを作成する．その後，ブロックを構成する構文要素をもつノードに対して，構文ベクトルから安定分布に基づく LSH によるハッシュ値を算出し，各ブロックのもつハッシュ値のリストを作成する．このリストから，同じハッシュ値を持つブロックをセットとして括りだし，クローンセット候補のリストを作成する．更に，LSH ではベクトルの大きさが小さい場合，類似していないベクトルであっても同じハッシュ値をとる可能性が大きくなるため，ハッシュ値の絶対値が小さいセットをクローンセット候補リストから除外する．最後に，クローンペア候補のリストから，コードクローンでないものをフィルタリングする．このようにブロック単位でクローンの判定を行うことで，フィルタリング対象を絞り，検出時間を短縮することができる．DECKARD ではクローンペア候補リストから，クローンセットリストを作成する際のフィルタリングにテキスト類似度による指標を利用している．提案手法では，まず対象のブロックがスペース，タブを除き一致する，タイプ 1 クローンであるかを確認する．次に，対象ブロックがラベル名を除き一致する，タイプ 2 クローンであるかを確認する．上記の二つに当てはまらない場合，タイプ 3 クローンの候補となる．タイプ 3 クローンの候補から，対象ブロックの構文要素が一致しないもの，含まれる構文要素が一致しないものを除外する．タイプ 1，タイプ 2，タイプ 3 クローンのいずれにも含まれないブロックをクローンセット候補のリストから削除する．

代表的な HDL である Verilog を対象に，提案手法をツールとして実装した．Verilog から AST への変換には高前田らの開発した Pyverilog [9] を利用する．Pyverilog は，Verilog ファイルを入力するとタブ文字で階層表現した AST ファイルを出力する．開発ツールは探索対象のファイルを Pyverilog に渡し，出力された AST ファイルを読み込み，構文ベクトルを作成する．構文ベクトルの作成には 37 種類の構文要素を利用した．そのうち ModuleDef，Block，Always，ForStatement，GenerateStatement，IfStataement，LocalParam，Parameter，ParameterList，PortList の 10 個の構文要素を，ブロックを構成する要素として扱った．図 1 に AST と構文ベクトルについて，簡素化した例を示す．続いて作成した構文ベクトルからブロックごとに LSH の計算を行い，ペアを構築しないもの，ハッシュ値の絶対値が 10 より小さいものを除外したうえでクローンセット候補リストを出力する．コードクローンでないセットのフィルタリングは目視にて行う．

4.　ケーススタディ

ケーススタディとして，開発ツールを用いて HDL におけるクローンの検出を行い，分析をした．第一節ではケーススタディの対象について紹介し，第二節で検出結果を記す．

4.1　対　　象

ケーススタディには，OpenCores に登録されている Verilog で記述されたプロジェクトを用いる．ケーススタディの概略を図 2 に示す．プロジェクト間をまたいだコードクローンが存在するかを調査するため，単一プロジェクトでなく，1 つ以上の Verilog ファイルを含む全 267 件のプロジェクトを対象とした．これらのプロジェクト中には 5200 個の Verilog ファイルが存在し，各プロジェクトは平均 19.5 個である．全ファイルの総

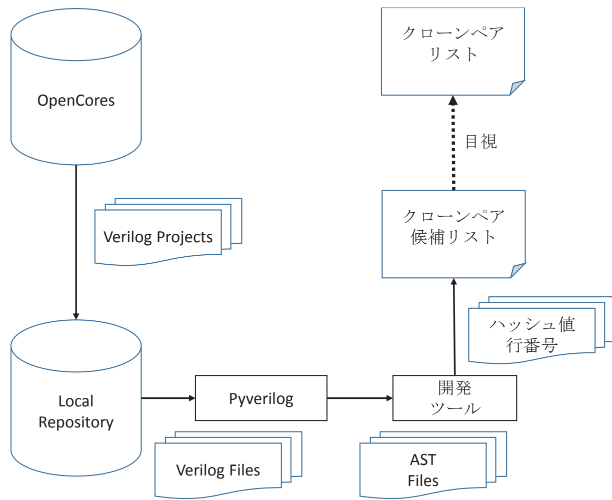

図 2　ケーススタディ概略

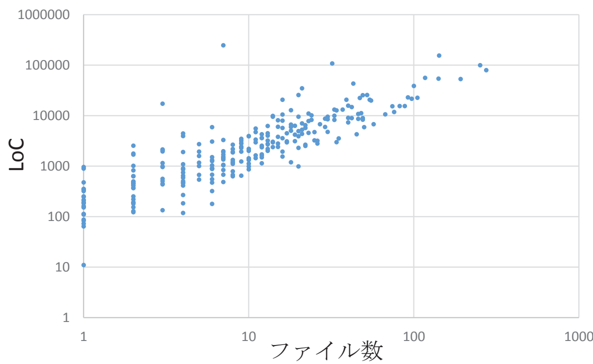

図 3　ファイル数と LoC の分布

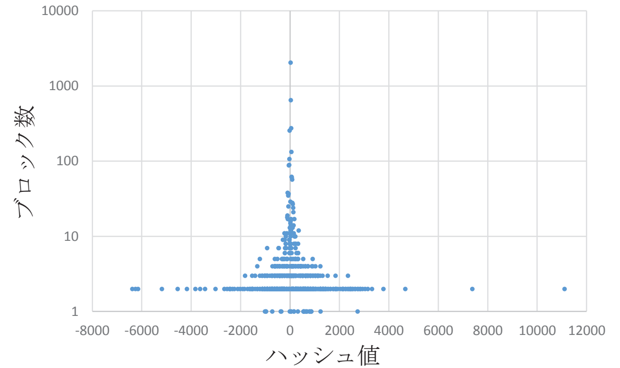

図 4　クローンセット候補のハッシュ値とセットに含まれるブロック数の分布

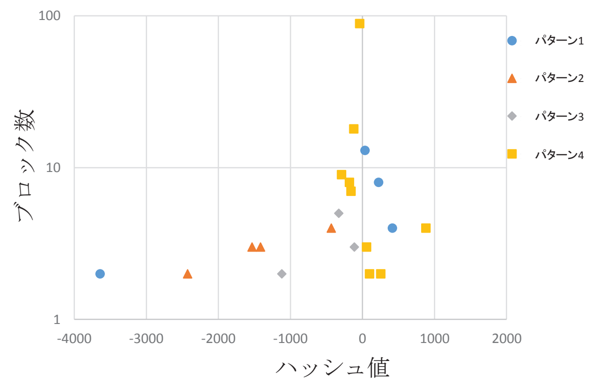

図 5　クローンセットのハッシュ値とセットに含まれるブロック数の分布

LOC は 2015377 行で，平均 387.6 行である．プロジェクトに含まれる Verilog ファイル数と，総 LoC の分布は図 3 のようになっている．このファイル群に対し，提案手法を実装したツールを用いてブロック毎のハッシュ値を計算し，クローンセット候補のリストを作成する．フィルタリングはクローンセット候補からランダムで 100 セット抽出し，目視にて行った．

4.2　結　　果

対象とした 267 件のプロジェクトから，164 件のプロジェクトにまたがる 952 個の AST ファイルが得られた．得られた AST ファイル中にクローン探索の対象となるブロックは 30228 個存在し，同じハッシュ値をもつブロックは 598 セットであった．尚，4248 個のファイルは構文エラーを含んでいるため，AST ファイルを出力できていない．クローンセット候補のハッシュ値とに含まれるブロック数の分布を図 4 に示す．この図より，ハッシュ値の絶対値が小さいほど，同じ値をとるブロックが多いことが確認できる．また，ハッシュ値は広い範囲に万遍なく分布することがわかる．クローンセット候補からランダム抽出した 100 件を目視判定でフィルタリングした結果，20 セットにコードクローンが含まれていることが確認できた．リスト 3 とリスト 4 は確認されたクローンセットの一例である．クローンセットのハッシュ値と，セットに含まれるブロックの数の分布を図 5 に示す．

5.　考　　察

この章では検出されたコードクローンを分析し，パターン分類した結果と，コードクローン検出精度向上に関する考察，及び妥当性の脅威について記述する．

5.1　コードクローンパターン分類

ケーススタディにてコードクローンと判定したものはいくつかのパターンに分類することができた．それぞれのパターンの件数を表 1 に示す．

[パターン 1]　複数のプロジェクトにまたがり，ファイル全体がタイプ 1 クローンになっている．このパターンは一般的な機能を持つ回路をライブラリの様に利用している．

[パターン 2]　単一のプロジェクト内でファイルの全体がタイプ 3 クローンになっている．プロジェクト中に複数の仕様の異なる回路に対応した HDL コードを持つ場合，このようなコードクローンが作成される．

[パターン 3]　単一のプロジェクト内でファイルの全体，あるいは module の全体がタイプ 1，タイプ 2 クローンになっている．加算回路などのような単純な機能はこのような形でコードクローンとして作られる．

[パターン 4]　プロジェクト中の同一ファイル，異なるファイルの両方に存在する，タイプ 2，タイプ 3 のコードクローン．

表 1 クローンパターンの個数

パターン	個数
パターン 1	4
パターン 2	4
パターン 3	3
パターン 4	9

always ブロックや if ブロックの単位でコードクローンになっている．このようなクローンは回路の初期化や値の変更などを目的とした処理のために作られる．

パターン 1 に属するコードクローンはファイル名も同一であり，コードクローン検出ツールの対象とせずとも見つけることができる．また，変更が必要な際の追従も容易であるため，このようなコードクローンの修正支援は必要ではないと考えられる．パターン 2 のコードクローンは，ファイル名が一部共通であることが多い．ファイル名に共通する部分があるため，検出は容易である．しかし，処理が一部異なるため，変更が必要な場合であってもすべてのクローンセットに対して同様の変更を加える必要があるか判断する必要がある．そのため，このようなコードクローンの変更に対する支援は，開発効率化において有用だと考えられる．パターン 3 における module 単位でのコードクローンは，module 名が共通であるとは限らない．変更の必要がある際には，同様の変更を加える必要があるため，クローン検出ツールの対象として検出し，同様に変更するなどの開発支援が有効である．パターン 4 については，コードクローン検出ツールを用いなければ発見が困難である．しかし，初期化する値や，初期化する対象が異なるものが多く，検出しても単純に変更支援の形で開発効率化に活用することはできないと考える．

5.2 検出精度

コードクローンではないと判断した 80 件のうちには，含まれる構文要素が大きく異なるセットも存在した．この場合，最上位の構成要素であるブロックも別の要素であることが多い．よって，クローンセット候補リストを作成する際に，同じハッシュ値をとるセットであって最上位の構文要素を比較し，異なる場合はフィルタリングすることで精度向上を図ることができると考える．

5.3 妥当性への脅威

今回のコードクローンの分析は，クローンセットの候補からランダム抽出したものを対象としたため，発見できていないコードクローンのパターンが存在する可能性がある．しかし，図 5 の分布図より，確認したコードクローンは広い範囲から抽出されているため，網羅性は十分であると考えられる．

6. まとめ

本研究では Verilog を対象としたコードクローンの検出ツールの開発を行い，ツールの検出結果を用いたケーススタディとして OpenCores のプロジェクト中に含まれるコードクローンを検出し，その分析を行った．ランダム抽出した 100 件のクローン候補を確認したところ，20 件のコードクローンが確認された．これらは，コピーするブロックの範囲や貼り付け先，クローンのタイプ，その目的により，4 パターンに分類できた．

リスト 3 clone-sample1

```
1 module hpdmc_oddr32 #(
2   parameter
        DDR_ALIGNMENT =
        "C0",
3   parameter INIT = 1'b0,
4   parameter SRTYPE = "
        ASYNC"
5 ) (
6   output [31:0]Q,
7   input C0,
8   input C1,
9   input CE,
10  input [31:0] D0,
11  input [31:0] D1,
12  input R,
13  input S
14 );
```

リスト 4 clone-sample2

```
1 module hpdmc_iddr32 #(
2   parameter
        DDR_ALIGNMENT =
        "C0",
3   parameter INIT_Q0 = 1'
        b0,
4   parameter INIT_Q1 = 1'
        b0,
5   parameter SRTYPE = "
        ASYNC"
6 ) (
7   output [31:0] Q0,
8   output [31:0] Q1,
9   input C0,
10  input C1,
11  input CE,
12  input [31:0] D,
13  inputR,
14  inputS
15 );
```

今後，コードクローンが回路開発にどのような影響を与えているのか調査する必要がある．また，今回作成したクローン検出ツールはブロック単位で構文情報を用いて LSH を計算した結果の出力に留まっており，クローンでないブロックのフィルタリングは目視により行う必要がある．今後の課題として，検出ツールへのコードクローンのフィルタリング機能の追加が必要となる．

文 献

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merio, "Comparison and Evaluation of Clone Detection Tools," IEEE Trans. Softw. Eng., vol.33, no.9, pp.577–591, 2007.

[2] L. Angela, W. Michel, and N. Bashar, "Evaluating the Harmfulness of Cloning: A Change Based Experiment," in Proceedings of 29th International Conference on Software Engineering Workshops, pp.18–21, 2007.

[3] OpenCores, "opencores," http://opencores.org.

[4] M. Parizy, K. Takayama, and Y. Kanazawa, "Software Defect Prediction for LSI Designs," in Proceedings of 30th IEEE International Conference on Software Maintenance and Evolution, pp.565–568, IEEE, 2014.

[5] J.A. Nacif, T.S.F. Silva, L.F.M. Vieira, A.B. Vieira, A.O. Fernandes, and C. Coelho, "Tracking hardware evolution," in Proceedings of 12th International Symposium on Quality Electronic Design, pp.1–6, 2011.

[6] E. Juergens, F. Deissenboeck, and B. Hummel, "CloneDetective - A workbench for clone detection research," in Proceedings of 31st International Conference Software Engineering, pp.603–606, 2009.

[7] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in Proceedings of 30th International Conference on Software Engineerin, pp.321–330, 2008.

[8] L. Jiang, G. Misherghi, and Z. Su, "DECKARD: Scalable and accurate tree-based detection of code clones," in Proceedings of 30th International Conference on Software Engineering, pp.96–105, 2007.

[9] S. Takamaeda-Yamazaki, "Pyverilog: A Python-based

Hardware Design Processing Toolkit for Verilog HDL," in Proceedings of 11th International Symposium on Applied Reconfigurable Computing, pp.451–460, 2015.