HardwareDescriptionLanguageを対象としたコードクローンの調査

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 のコード中にどの程度の量のコードクローンが存在し，影響を与えているか明らかにされていない．また，HDL を対象としたコードクローン検出ツールは我々の知る限り存在しないそこで，本研究では HDL におけるコードクローンを検出する手法を提案する．提案手法では Verilog を疑似的な C++コードに変換することで，既存のコードクローン検出ツールである CCFinder を用いてコードクローンを検出できるようにする．提案手法を用いて，OpenCores が提供するプロジェクトを対象にコードクローンの調査を行った．調査の結果，HDL のコード中にはコードクローンが多数存在することが確認された．

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

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，将来的に検証やバグ修正が回路開発のボトルネックになると言われている．そのため，バグを含むモジュールの推定など，検証作業の支援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．コードクローンの存在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 [1]．コードクローン研究においては，CCFinder [2] が広く利用されている．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．しかし，既存のコードクローン検出ツールは HDL を対象としていない．そこで，本研究では代表的な HDL である Verilog を対象にコードクローンの検出手法を提案し，Verilog におけるコードクローンの実態について調査を行った．提案手法は CCFinder が検出できるよう，Verilog を疑似的な C++コードに変換する．

調査にあたり，以下のリサーチクエスチョンを設定した．

[RQ1]　疑似 C++コードのトークン化によるコードクローン検出手法は有効か．

[RQ2]　Verilog においてコードクローンはどの程度存在するのか．

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．4，5 章で実験概要及び，実験結果と結果に対する考察を述べる．6 章で関連研究を紹介し，7 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クローンペアと呼ばれる．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 [3]．

[**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 [4]．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

リスト 1　example.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　example.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
```

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われるため，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．例えば，OpenCores [5] では，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ソースコードの上から下に順に実行される．しかし，HDL においては，順に実行されるとは限らない．Verilog のコードを例に挙げて説明する．リスト 1，リスト 2 の 5 行目から始まる 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 信号のどちらかの立ち上がりが入力された際に出力 d に b の値が設定されるような，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 [6]．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 [7]．eyesOn はレジスタ，ゲート，信号線の数や，LoC 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．

現在，HDL を対象としたコードクローン検出ツールは我々の知る限り存在しない．そのため，HDL にどの程度の量のコードクローンが存在する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．我々は前述した実行順序の違いに着目し，タイプ 1 からタイプ 3 のコードクローンを検出するツールを開発中であるが，未だ十分な機能を有していない [8]．そこで，本研究では既存のコードクローン検出ツールを用いて HDL におけるタイプ 1，2 のコードクローンを検出する手法を提案し，HDL 中に存在するコードクローンの実態について調査を行う．

3. コードクローンの検出

ここでは，本研究で提案する HDL のコードクローンの検出手法について説明する．まず，3.1 節にてソフトウェアを対象とした既存のコードクローンの検出手法について紹介する．続いて，3.2 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 [9]〜[11]．

Kamiya らの開発した CCFinder はトークンの並びに着目し，タイプ 1，2 のコードクローンを検出するツールである [2]．COBOL，C++，C #，Java，VisualBasic を対象に構文解析結果に基づきトークンを正規化し，正規化したトークンを基にコードクローンを検出する．構文が解釈できずトークンの正規化ができない箇所については，トークン化に失敗したとしてコメント文と同様に探索対象から除外する．また，上記のプログラミング言語以外のテキストについては，正規化をせずにクローンを検出することもできる．なお，構文解析されるのは限定的な範囲で，例えば C++では関数の階層までで，たとえ関数以下に文法上の誤りがあったとしてもコードクローンの検出は実施される．

3.2 提案手法

CCFinder は Verilog コードには対応していないため，プログラミング言語としては構文を解釈せず，トークンの並びとして認識されない．本研究では，Verilog を疑似的な C++コードに変換をすることで，CCFinder を用いて Verilog コード中のコードクローンを検出する手法を提案する．細かな構文解析を行わない階層に対応付けるよう Verilog のコードを変換するこ

リスト 3　convert_example.v

```
1 module edge (
2     input rst,
3     input clk,
4     input sig,
5     output rise,
6     output fall
7 );
8   reg [1:0] sig_reg;
9   always @(posedge clk or
        posedge rst)
10    if (rst == 1'b1)
11      sig_reg <= 2'b00;
12    else
13      sig_reg <= {sig_reg[0],
          sig};
14  assign rise = sig_reg[0] ==
        1'b1 && sig_reg[1] ==
        1'b0 ? 1'b1 : 1'b0;
15  assign fall = sig_reg[0] ==
        1'b0 && sig_reg[1] ==
        1'b1 ? 1'b1 : 1'b0;
16 endmodule
```

リスト 4　convert_example.cpp

```
1 class edge{edge(
2   input rst,
3   input clk,
4   input sig,
5   output rise,
6   output fall
7 );{
8   reg [1:0] sig_reg;
9   always @(posedge clk or
        posedge rst)
10    if (rst == 1'b1)
11      sig_reg <= 2'b00;
12    else
13      sig_reg <= {sig_reg[0],
          sig};
14  assign rise = sig_reg[0] ==
        1'b1 && sig_reg[1] ==
        1'b0 ? 1'b1 : 1'b0;
15  assign fall = sig_reg[0] ==
        1'b0 && sig_reg[1] ==
        1'b1 ? 1'b1 : 1'b0;
16 }};
```

とで，大幅な変更を加えることなく，Verilog のトークン化が可能になる．なお，C++を選択した理由は，言語を構成する構文要素や利用される記号が Verilog と類似しているためである．

Verilog は assign 文や always ブロック，if ブロックなどの構文要素をもち，これらは module ブロックの中に記述される．module ブロックの定義は，module 宣言の後ろに module 名，入出力信号のパラメータ列が続き，前述の構文要素が記述され，endmodule で閉じられる．一つの Verilog ファイルには module を複数記述することができる．このような Verilog における module の階層構造は，C++における class の階層構造と類似しているといえる．従って Verilog の module の定義は C++の class の定義と対応付けるのが適切であると考える．そして，assign 文などの構文要素は class 中の function 内の記述内容と対応付けることができる．したがって，module 定義を class 及び class の constructor として扱い，入出力信号のパラメーターを constructor の引数，module 内の構文要素を constructor 内の構文要素となるように Verilog を変換することで，疑似的な C++コードとして扱うことができる．リスト 3 とリスト 4 に変換前の Verilog コードと変換後の疑似 C++コードの例を挙げる．7 行目のコンストラクタの引数リストの直後に存在するセミコロンのように，本来の C++では文法上エラーとなるような構文であっても，CCFinder は構文解釈可能である．変換例に示すように，コードの行数には変化が無いため，変換前と変換後の比較を行うことも容易である．

4. 実験概要

Verilog のコードクローンの調査を行うにあたり，2 つのリサーチクエスチョンを設け，これらに答える形で実験を行う．

[RQ1]　疑似 C++コードのトークン化によるコードクローン検出手法は有効か．

[RQ2]　Verilog においてコードクローンはどの程度存在するのか．

RQ1 では変換規則の妥当性を確認するため，3.1 節で述べた CCFinder を用いて，疑似 C++コードがトークン化された行数と，構文が解釈できずトークン化に失敗した行数について分析を行う．加えて提案手法と，Verilog コードを変換せず plaintext としてトークン化しコードクローン検出を行う非変換手法の，2 つの手法で検出されるコードクローンの件数について比較し提案手法の有効性について検討を行う．

RQ2 では提案手法を用いてコードクローンの検出を行い，コード中に含まれるコードクローンの量について分析する．

実験対象は OpenCores で公開されている Verilog のプロジェクトからドメインが異なるように無作為に 3 件を選択した．対象プロジェクトの情報を表 1 にまとめる．

CCFinder の設定は Minimum Clone Length を 20 に変更し，その他の設定はデフォルト値のままとした．Minimum Clone Length を 20 とした理由は，事前実験において 30 トークンで構成されるコードクローンの件数が多く確認されたためである．

表 1　対象プロジェクト

プロジェクト名	ファイル数	LoC
1000base-x	13	4737
mor1kx	45	21326
sd_card_controller	17	3355

表 2　トークン化失敗率

プロジェクト名	LoC	コメント行数	失敗行数	割合
1000base-x	4737	1217	484	13.8
mor1kx	21326	3908	6345	36.4
sd_card_controller	3355	1056	469	20.4
total	29418	6181	7298	31.4

5. 実験結果と考察

5.1 RQ1

各対象プロジェクトにおける，コメントを除いたコード行数と，そのうちのトークン化に失敗した行数について表 2 にまとめる．トークン化に失敗している行数は全体の 3 割程度である．ファイル毎のトークン化失敗率の分布を図 1 に示す．プロジェクトごと大きく傾向が異なることが読み取れる．これは，利用されることの多い文法構造や，コーディング規約がプロジェクトごとに異なり，その一部が C++として解釈できないためと考えられる．より詳細な分析のためには，トークン化に失敗したコードおよび周辺の記述についてより詳細に観察する必要がある．

それぞれの手法で検出されたコードクローンの件数について表 3 にまとめる．非変換手法の方が，より多くのコードクローンを検出できているが，この原因としては 2 つの要因が考えられる．第 1 に提案手法ではコメント文がクローン検出対象ではないのに対し，非変換手法ではコメント文もコードクローンとして検出されるという点が挙げられる．実際に，提案手法で

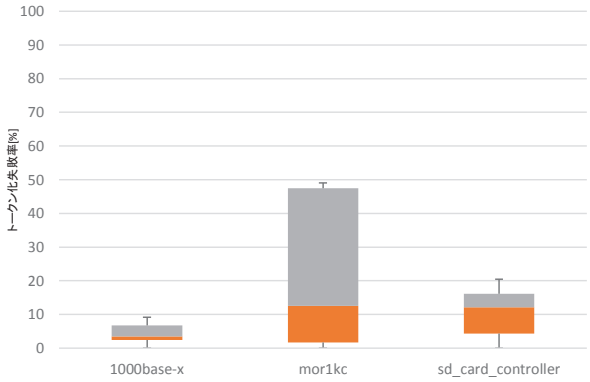

図 1　ファイル毎のトークン化失敗率の分布

表 3　ファイル毎のコードクローンの検出件数

手法	TotalNum	Ave Num	Max Num	Min Num
提案手法	847	11.29	41	0
非変換手法	2344	31.25	89	2

はクローン件数が 0 件だったファイルにおいて，非変換手法はコメント文がクローンとして検出されている例が確認できた．第 2 に，非変換手法では空白文字や記号毎にトークンが区切られるため，同じ長さの文を対象としても提案手法と比べトークン数が大きくなる．例えば a_b_c という文字列は 5 個のトークンで構成される．従って，CCFinder の設定である Minimum Token Length の閾値を超えやすいといったことが挙げられる．これらのコードクローンは本来検出する必要のないものであるため，提案手法が検出する件数が非変換手法よりも少ないことは大きな問題ではないと考えられる．

上述のように，トークン化に失敗する構文が存在する課題はあるが，一定量のクローンが検出できているという点に加え，非変換手法と比較し，不必要なクローンが検出されないという点から，提案手法は Veriilog におけるコードクローンの検出において有効な手法であると言える．

5.2　RQ2

提案手法で検出したコードクローンのファイル毎の件数の分布及びサイズの分布，各ファイルのうちコードクローンが占める割合の分布を図 2，図 3，図 4 に示す．

図 2 が示すように，プロジェクトごとに差はあるが，どのプロジェクトにもコードクローンが存在している．また，ファイルの大半をコードクローンが占めている事例が図 4 から確認できる．従って，コードクローンを管理できるようにすることは，プロジェクト管理上有用であると考えられる．

図 3 に示すように，一つのクローンあたりの規模については 30-70 個のトークンで構成されるものが多い．これにより，always ブロック等の構文上のひとまとまりが 30-70 個程度であることが多く，クローンもそのような単位で作成されていると推測される．

5.3　妥当性への脅威

RQ1 におけるトークン化失敗率についてはトークン数ではなく行数を評価基準としている．これは，トークン化に失敗している箇所についてはトークン数を定義することができないためである．しかし，ソースコード全体がどれだけトークン化できているかという評価をするにあたり，行数による評価は十分

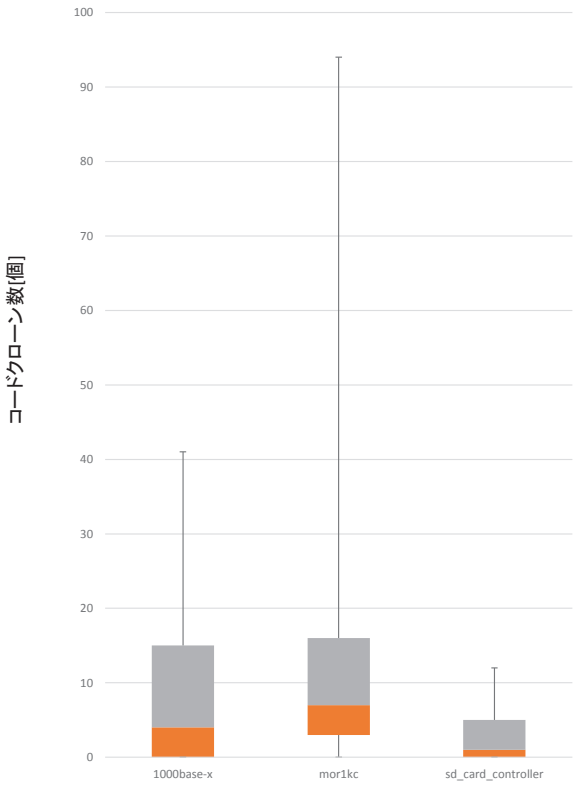

図 2　ファイル毎のクローン数の分布

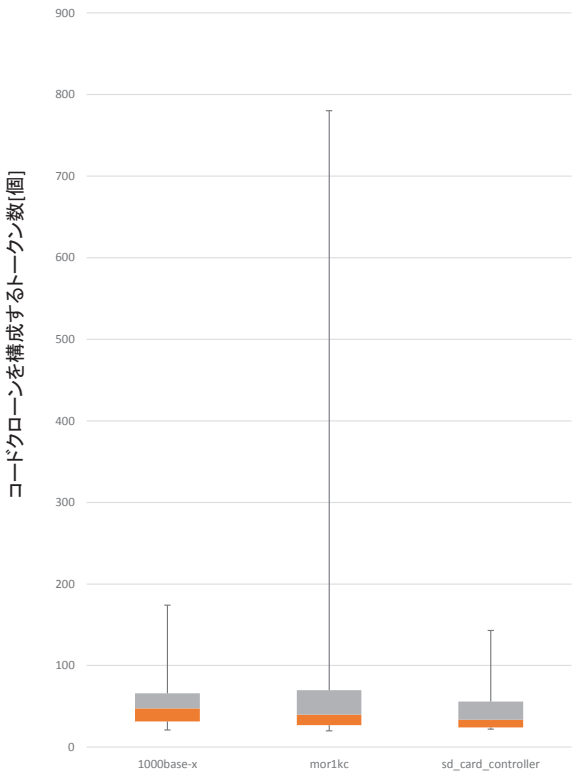

図 3　クローンのサイズの分布

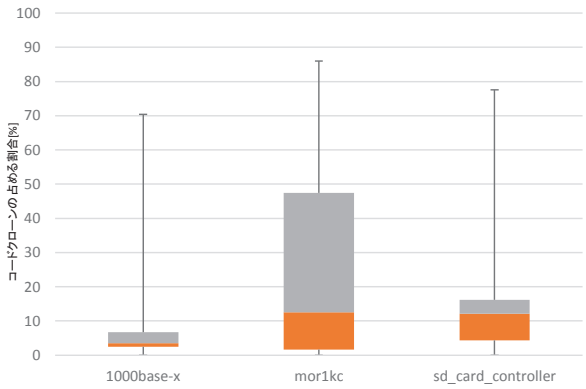

図 4　クローンが占める割合の分布

なものであると考える．

RQ2 の実験については提案手法を用いており，全体の Verilog コードの 3 割程度がトークン化に失敗しているため，コードクローンの占める割合が正確でない可能性がある．しかし，トークン化できていない箇所についても，トークン化できている箇所と同じ割合でコードクローンが存在すると考えられるため，影響は軽微であると考えられる．しかし，今後より正確な調査を行うためには，トークン化に失敗しているコードについて原因を調査し，使われている構文に応じて変換規則を変更するなど，疑似 C++コードへの変換規則を考案する必要がある．

提案手法はタイプ 1，2 のコードクローンを検出するものであり，タイプ 3 コードクローンは検出することができない．HDL は処理が記述の順の通りであると限らないという特徴上，文の入れ替わりが起きやすく，タイプ 3 のコードクローンが多く含まれている可能性があり，今後検証する必要がある．

今回調査対象としたプロジェクトは 3 つであり，調査対象として数が十分でない可能性がある．しかし，3 つのプロジェクトはそれぞれ異なるドメインを対象とした回路であり，偏りはなく一般的な傾向を示しているものと考える．

6. 関 連 研 究

我々は AST を基にしたコードクローンの検出ツールを開発し，297 個のプロジェクトを対象としたコードクローンの調査を行っている [8]．この調査では複数のプロジェクト間に横断的に存在するコードクローンにも着目し，作成されているコードクローンについてパターンの分類を行っている．しかし，このツールは開発途中のものであり，検出結果の妥当性については確認されていない．手法により検出されるコードクローンに差異があるのか，今後検証が必要である．

Verilog 中のコードクローンの存在が品質やメンテナンス性に与える影響については，今後調査する必要がある．ソフトウェアにおける品質やメンテナンス性へコードクローンの存在が与える影響については，多数の研究が行われている．中山らはコードクローンの内外のバグの混入率の違いについて着目した分析を行っている [12]．

7. ま と め

本研究では，Verilog コードを疑似 C++コードに変換することで，CCFinder を用いたトークンの並びにに基づくコードクローンの検出を行う手法を提案した．提案手法は，全体の 3 割が検出対象から漏れるが，残りのコードについては疑似 C++コードに変換しない手法と比較して，コメント文などの余分な文を検出することなくコードクローンを検出できることができた．また，3 つのプロジェクトを対象に，提案手法を用いて Verilog コード中に存在するコードクローンの実態について調査を行った．その結果，プロジェクトごとに傾向の差異はあるものの，Verilog 中にコードクローンが多数存在することが確認できた．

提案手法においてトークン化に失敗することで検出対象から漏れるコードが存在する課題については，詳細に観察し，疑似 C++コードへの変換規則を改善する必要があり，今後の研究課題とする．

文　　献

[1] 服部剛之，吉田則裕，早瀬康裕，肥後芳樹，松下　誠，楠本真二，井上克郎，“識別子の共起関係に基づく類似コード検索法の提案と欠陥検出への適用，” 電子情報通信学会技術研究報告. SS, ソフトウェアサイエンス，vol.107，no.392，pp.55–60，dec 2007. http://ci.nii.ac.jp/naid/110006549283/

[2] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: A multilinguistic token-based code clone detection system for large scale source code,” IEEE Transactions on Software Engineering, vol.28, no.7, pp.654–670, 2002.

[3] 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.

[4] 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.

[5] OpenCores, “opencores,” http://opencores.org.

[6] 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.

[7] 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.

[8] 上村恭平，藤原賢二，飯田　元，“Hardware description language におけるコードクローンのパターン分類，” IEICE-SS2015-5，pp.23–28，May 2015.

[9] 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.

[10] 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.

[11] 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.

[12] 中山直輝，吉田則裕，藤原賢二，飯田　元，“開発履歴分析を用いたコードクローン内外における欠陥発生率の調査，” 情報処理学会研究報告，第 2014-SE-186 巻，pp.1–8，Nov. 2014.