情報処理学会論文誌 Vol.59 No.4 1225–1239 (Apr. 2018)

ハードウェア記述言語におけるコードクローンの定量的調査

上村 恭平[1,2,a)] 森 彰[2,b)] 藤原 賢二[3,c)] 崔 恩瀞[1,d)] 飯田 元[1,e)]

受付日 2017年8月3日, 採録日 2018年1月15日

概要：ハードウェア記述言語は，Field Programmable Gate Array（FPGA）開発などで回路の構造を定義するために用いられる言語である．近年の FPGA の利用拡大により，ハードウェア記述言語（HDL）を用いた回路開発の効率化が課題となっている．そこで，我々はソースコード中の重複あるいは類似したコード片であるコードクローンに着目した．ソフトウェアにおいて，コードクローンは開発効率を低下させる一因として研究されている．本論文では，代表的な HDL である Verilog HDL を対象としたコードクローン検出手法を提案し，コードクローンの特徴について調査した結果について述べる．提案するコードクローン検出手法は，Verilog HDL のソースコードに簡単な変換を適用することで，既存のツールを用いてコードクローンを検出する．評価の結果，提案手法は 90%以上の精度でコードクローンを検出できた．また，提案手法を用いてコードクローンの量と複雑さについて分析した結果，C や Java と同様にコードクローンが存在し，支援を要することが確認された．ソフトウェアと同様に，Verilog HDL のコードクローンに対しても同時編集支援やドキュメント化などの管理は有用である．一方で，Verilog HDL におけるコードクローンはリファクタリングによる集約を行う場合に回路性能とのトレードオフを考慮する必要がある．

キーワード：コードクローン，ハードウェア記述言語，実証的ソフトウェア工学

Quantitative Investigation of Code Clones in Hardware Description Language

Kyohei Uemura[1,2,a)] Akira Mori[2,b)] Kenji Fujiwara[3,c)] Eunjong Choi[1,d)] Hajimu Iida[1,e)]

Received: August 3, 2017, Accepted: January 15, 2018

Abstract: A hardware description language (HDL) is a computer language used to describe the structure and behavior of digital logic circuits including field programmable gate arrays (FPGAs). The rapid growth of FPGA usage requires us to make circuit development involving HDLs more efficient. In this paper, we focus on code clones in HDLs. Code clones are the similar segments of code that are typically created when the code is copied from one place to another. In software development, code clones are considered to decrease development efficiency. To study code clones in HDLs, we developed a code clone detection method for Verilog HDL, which is the most popular HDL. In this method, we apply simple conversion rules to the Verilog HDL code so that we can use existing code clone detection tools for traditional programming languages. The experiments showed that the accuracy of the proposed detection method was about 90%. We compared code clones in Verilog HDL with those in Java and C based on the metrics to identify the differences among languages. We found that the tool support for consistent modification over clone sets is also useful for Verilog HDL. However, aggregating clone sets in Verilog HDL must take into consideration the trade-offs between computational parallelism and circuit footprints.

Keywords: code clone, hardware description language, empirical software engineering

1. はじめに

Field Programmable Gate Array（FPGA）は構造を書

[1] 奈良先端科学技術大学院大学
Nara Institute of Sience and Technology, Ikoma, Nara 630–0192, Japan
[2] 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology, Ikeda, Osaka 563–8577, Japan
[3] 豊田工業高等専門学校
National Institute of Technology Toyota College, Toyota, Aichi 471–8525, Japan

a) uemura.kyohei.ub9@is.naist.jp
b) a-mori@aist.go.jp
c) fujiwara@toytoa-ct.ac.jp
d) choi@is.naist.jp
e) iida@itc.naist.jp

き換えることが可能な集積回路である．FPGA を用いたシステムは，計算内容によってはCPU のみを用いたシステムよりも高速，あるいは低消費電力に処理することができるといわれている．そのため，FPGA を用いたシステムが研究，開発の対象となっている [1], [2]．しかし，FPGA の性能向上により大規模かつ複雑化した回路の開発においては，デバッグやバグ修正などにかかる時間が問題になるといわれている．回路開発に用いられるハードウェア記述言語（HDL）はソフトウェア開発におけるプログラミング言語と類似した側面を持つ．そのため，ソフトウェア工学で提案されている技術や知見を応用することで回路開発を支援する研究が報告されている [3], [4], [5]．一方で，HDL とプログラミング言語では異なる面も多数存在する．HDL とプログラミング言語の最も大きな差異は，記述する対象が異なることによる，コードの構造の違いである．ソフトウェア開発では，プログラミング言語を用いて，データ構造や，どのような処理を行うかを抽象的に記述するため，コードの構造は処理の構造を直接示していない．これに対し HDL を用いた回路開発では，処理の内容ではなく，レジスタや配線などの回路の構造を記述するため，コードの構造は直接回路の構造を示す．そのため，ソフトウェア開発と回路開発では，記述されるコードの特徴にも差異が存在する．たとえば，複数のデータに対し並列的に同じ処理を行う場合，ソフトウェアでは行う処理に対応するコードを記述し，その処理を並列に実行するように呼び出すことで実現する．しかし，回路開発では処理を行うコードを並列に処理する数だけ，同様のコードを複数箇所に記述することになる．したがって，前述のソフトウェア工学における技術や知見を回路開発に対して応用する研究では，利用する手法において，どのようにしてこのような特徴の差異を埋めるかや，調査結果において特徴の差異がどのように影響を及ぼしているかを検討することが重要である．

ソフトウェア工学におけるさかんな研究分野の 1 つとしてコードクローンがあげられる．コードクローンとは，主に開発者が既存のコードの断片をコピーアンドペーストすることで作られる，類似あるいは重複するソースコードの断片である [6]．コピーアンドペーストされたコード片は，目的によって書き換えられることがある．そのため，コードクローンは変更の度合いによって，以下の 3 つのタイプに分類される [7]．

Type1 空白文字や改行およびコメント文の差異を除き一致するコードクローン

Type2 Type1 に加え，識別子名やリテラルの差異を除き一致するコードクローン

Type3 Type2 に加え，式・文の追加や削除を含むが，類似するコードクロ―ン

効果的なコピーアンドペーストは開発を効率的に進めることを可能にする一方で，コードクローンの存在は機能の変更やバグ修正の効率を低下させるともいわれている [8]．そのため，コードクローンはリファクタリングにより集約するか，集約できない場合には追跡管理することが望ましいといわれており，そのための手法が研究されている [9], [10]．

我々は，HDL コード中にもコードクローンが存在する場合，その特徴を明らかにすることで開発の効率化に貢献することができると考えた．前述のように，HDL コードは回路の構造を直接的に表現しているため，類似するコードを複数回記述する機会が多く，必然的にコードクローンが作られやすいと考えられる．我々は先行研究において，代表的な HDL である Verilog HDL を対象としたコードクローンの検出手法を提案し，10 件のプロジェクト中のコードクローンについて調査を行った [11]．しかし，先行研究では，検出手法の性能を十分に評価できていないことに加え，対象プロジェクトが少ないため定量的に議論できていないという課題がある．本論文の位置づけは，先行研究の発展として，新たに提案手法の検出精度を評価したうえで，対象プロジェクト数を増やし，統計的な分析を含めたコードクローンの定量的な調査を行うものである．調査の目的は Verilog HDL を用いた開発に対してコードクローンに関する支援が必要であるか，また必要な場合，どのような支援が有効であるかを明らかにすることである．本研究では，この目的を達成するため，以下の 3 つのリサーチクエスチョン（RQ）に沿って調査を行った．

RQ1 HDL のコードクローンは一般的に存在するか？

RQ2 HDL のコードクローンは単純なものではないか？

RQ3 HDL のコードクローンはどのような理由で作られるか？

HDL コード中にコードクローンがコード中の一部にしか存在しない，あるいは非常に単純なコードクローンである場合，コードクローンに関する開発支援の効果は低い．逆に，コードクローンが広く存在し，また，複雑であればコードクローンに関する支援の必要性は高い．そのため，RQ1 においてコードクローンが一般に存在するかを調査し，RQ2 においてコードクローンの複雑さを調査する．また，支援が必要である場合も，コードクローンの性質によって，有効な支援方法は異なると考えられる．そのため，RQ3 ではコードクローンが作られている理由に着目し，コードクローンの特徴を明らかにし，どのような支援が有効であるか検討する．

以降，2 章で関連研究について紹介する．3 章では提案するクローン検出手法の概要と，検出精度の評価結果について述べる．4 章で調査方法について説明し，5 章から 7 章で調査結果について述べる．8 章で調査結果について考察し，9 章で本論文をまとめる．

2. 関連研究

本章では，これまでに行われている，回路開発に対する開発者支援を目的とした関連研究および，コードクローンの検出手法とその評価手法，加えて，コードクローン管理手法に関する研究について紹介する．

2.1 回路開発の支援

Verilog HDL では一部の文法において，文の順序の違いが回路の構造の違いを意味しないため，diff などの一般的な差分検出ツールの結果には開発者にとって不要な情報が含まれる場合がある．そのため，Duley らは Verilog HDL の文法特徴を考慮した差分検出ツールを開発している [4]．Sudakrishnan らは，Verilog HDL で開発されたプロジェクトの，版管理システムに記録されている変更履歴とバグ管理システムの記録を調査している [3]．この研究では，Verilog HDL コードにおけるバグのパターンを分類し，if 文に関するバグが多いことなどを明らかにしている．また，Nacif らは回路固有のメトリクスを用いた開発管理のためのフレームワークを提案し，ケーススタディを通して，極端なゲート数やレジストリ数の増減が，バグの潜在性の予測に利用できることを確認している [5]．

一方で，高位合成と呼ばれる開発手法も研究されている [12], [13]．HDL を用いた開発では回路の構造を記述するが，高位合成による開発では，処理したい内容を C++ などのプログラミング言語で記述する．そして，記述したソースコードを HDL の記述に変換することで目的の回路を開発する．高位合成を用いるメリットとして，HDL による開発と異なり，回路構造を考慮する必要がなく，処理内容のみを考えればよいことあげられる．加えて，処理内容をソフトウェアとして検証，デバッグすることができる．反面，回路構造を考慮しないため，処理速度や消費電力，面積といった回路性能に関する最適化が困難であるといわれており，HDL を用いた開発に置き換わってはいない．

2.2 コードクローン検出

一般に，コードクローンの分析にはコードクローン検出ツールが用いられる．コードクローン検出ツールは，入力したソースコード中からコードクローンを検出し，その位置情報などを出力する．コードクローンを検出するためのアルゴリズムは多数提案されている．たとえば，Kamiya らの開発した CCFinderX は，ソースコードを識別子や数値を正規化したうえでトークン列として表現し，閾値を超えて連続するトークンの並びをコードクローンとして検出する [14]．Jiang らの提案する DECKARD は抽象構文木で表現したソースコードの，部分木ごとに求まる文法ベクトルに対し局所性鋭敏型ハッシュを計算し，同一ハッシュに属する部分木をコードクローンとして検出する [15]．これらの既存のコードクローン検出ツールは，いずれもソースコードをパースした結果を利用するため，C/C++や Java などの限られたプログラミング言語を対象としている．そのため，Verilog HDL を対象にコードクローンを検出，分析した研究は我々の知る限り報告されていない．

2.3 コードクローン検出ツールの評価手法

一般に，コードクローン検出ツールは，実際にはコードクローンでないコード片を誤検出する，あるいは本来存在するコードクローンを検出できないことがある．そのため，コードクローンの検出ツールの検出精度を評価する手法が研究されている．コードクローン検出ツールの検出精度は，検出されたコードクローンのうち正解の割合（Precision）と，すべての正解のうち検出できたコードクローンの割合（Recall）で評価することができる．Precision と Recall の計測には正解集合が必要だが，厳密なコードクローンの正解集合を求めるのは困難である．そのため Bellon らの研究では，多数の検出ツールの検出結果を合わせ，整理することで正解集合を構築している [16]．この手法は手作業での結果の整理をともなうため，大規模なプロジェクトに対して適用するのが困難である．また，複数の検出ツールが利用可能であることを前提としていることに加え，構築される正解集合は利用するツールの性能に依存する．

一方，Svajlenko らは自明なコードクローンを埋め込むことで正解集合を構築する手法を提案している [17]．この手法では，まずソースコード中から無作為に関数やブロック単位でコード片を複数選択する．次に，抽出したコード片に対して Type1，2，3 コードクローンに対応する 15 種類の変異をそれぞれ適用し，変異コード片を作成する．続いて，作成された各変異コード片をソースコード中の，文法上問題のないランダムな位置に挿入する．これにより，選択元のコードと，埋め込まれた変異コード片は自明なコードクローンとなる．Recall は，評価対象のコードクローン検出ツールが，埋め込んだ自明なコードクローンを検出できた割合で計算できる．また，自明なコードクローン，あるいはその一部を含む検出結果のうち，実際にコードクローンである検出結果の割合が Precision となる．したがって，この手法では，自明なコードクローンと関係しない検出結果は Recall，Precision の値に影響しない．この手法は Bellon らの手法と異なり，コードクローン検出ツールに依存しないことに加え，正解集合の構築に人の手が介在せず，半自動的に評価することができる．

2.4 コードクローン管理支援

コードクローンはソースコードの保守性に悪影響を及ぼすといわれている．たとえば，あるコード片を変更する際，そのコード片がコードクローンであると，対応するコード片は同時に変更されるべきである可能性が高い．そのよう

な場合に，どちらかのコード片のみを変更すると，バグの原因になる．そのため，コードクローンはリファクタリングにより集約することが望ましいといわれている．リファクタリングとは，ソースコードの可読性や変更性を高めるために，外部的な振舞いを変更せずソースコードを改善することをいう [18]．Baxter らは CloneDR というコードクローン検出ツールを開発している [6]．CloneDR は検出されたコードクローンの集約例を提示する機能を有している．

また，集約できないコードクローンに対しては，ソースコード中の位置などの情報の記録や，開発環境上で同時編集を補助するなどの開発支援が有効である．Duala-Ekoko らの提案する Clonetracker は，コードクローンを追跡し，コード片の編集時に同時に変更が必要な箇所を開発者に提示する [9]．また，山中らは開発者にコードクローンの存在を提示し，集約や同時修正，ドキュメントへの記録を促すシステムを開発している [10]．

3. コードクローンの検出

2.2 節で述べたように，既存のコードクローン検出ツールは特定の言語を対象としているため，Verilog HDL で記述されたソースコード中のコードクローンを検出することはできない．そこで我々は，Verilog HDL のソースコードを変換することで既存ツールである CCFinderX に対応させ，コードクローンを検出する手法を提案する．以降，提案する検出手法の概要について説明した後，提案手法の検出精度の評価について述べる．

3.1 検出手法概要

CCFinderX は C/C++，Java，C#，COBOL のコードを文法規則に則りトークン化し，閾値以上の長さで一致するトークンの列をコードクローンとして検出する．また，ソースコードではないテキスト文書（plaintext）中からも，空白文字に加え ‘-’ や ‘_’ などの文字で文字列を区切り，トークン化することで，一致する文字列を検出することができる．このトークン化には独自実装のパーサが用いられるが，厳密ではなく，部分的に文が検出対象の言語と類似していれば，その文をパース，トークン化してコードクローンを検出する．

Verilog HDL は文法において C/C++などの手続き型言語の影響を強く受けており，文法的特徴が類似している．一方で以下のような相違点も存在する．

- ブロックの開始や終了が ‘begin’，‘end’ などの文字列で表される．
- C/C++で式中に含まれない中括弧が，式中に含まれる．
- 整数の表現にバス幅（bit 数）と数値の組合せを利用することができ，‘$< bus\ width >'< value >$’ の形式で記述される．

提案手法では，CCFinderX がトークン化できるよう，これらの相違点を取り除いた疑似 C++コードに変換することで，Verilog HDL コード中のコードクローンを検出する．変換規則の一覧を**表 1** に示す．**図 1** に示す Verilog HDL のコードに，表 1 の変換規則を適用すると，**図 2** に示す疑似 C++コードに変換される．この例では，3，4，6，7 番目の変換規則が適用されている．

```
1    if(~RST_X)
2    begin
3        TXD       <= 1'b1;
4        READY     <= 1'b1;
5        cmd       <= 9'h1ff;
6        waitnum   <= 0;
7        cnt       <= 0;
8    end else if( READY )
9    begin
10       TXD       <= 1'b1;
11       waitnum   <= 0;
12       if( WE )
13           begin
14           READY <= 1'b0;
15           cmd   <= {DATA, 1'b0};
16           cnt   <= 10;
17       end
18   end
```

図 1 Verilog HDL コードの例
Fig. 1 Example of Verilog HDL code.

```
1    if(~RST_X)
2    {
3        TXD       <= 1;
4        READY     <= 1;
5        cmd       <= 511;
6        waitnum   <= 0;
7        cnt       <= 0;
8    } else if( READY )
9    {
10       TXD       <= 1;
11       waitnum   <= 0;
12       if( WE )
13       {
14           READY <= 0;
15           cmd   <= (DATA, 0);
16           cnt   <= 10;
17       }
18   }
```

図 2 疑似 C++コードの例
Fig. 2 Example of pseudo C++ code.

Verilog HDL コードを C++のコードに変換するツールはすでに存在する[*1]が，回路の動作をソフトウェアとしてシミュレートすることを目的としているため，このようなツールで変換されたソースコードは元の Verilog HDL コードとは構造が異なる．構造が異なるソースコード上でコードクローンを検出しても，元の Verilog HDL コード上にコードクローンが存在するとは限らず，対応付けることができない．そのため，変換には，ファイルや行，文の対応関係がとれるよう，Verilog HDL コードの構造を大きく変化させないことが求められる．我々の提案手法におけるコードの変換は単純な文字の置き換えのみであるため，コードの構造が大きく変化することはない．

我々は，Verilog HDL を疑似 C++に変換するツールを C#で実装した．変換ツールは，提案する変換規則に従い，正規表現を用いた文字列置換により変換する．

CCFinderX はソースコードをトークン化し，その並びが一致するコード片を検出するため，Type1，Type2 コードクローンを検出することができるが，Type3 コードクローンは検出することができない．提案手法は CCFinderX を利用するため，Type1，Type2 コードクローンのみを検出対象とする．

3.2 検出手法の評価

検出手法の評価には，2.3 節で紹介した Svajlenko らの提案する，変異コードを埋め込むことで正解集合を構築する手法を採用した．この手法は，Bellon らの手法と異なり，単一のコードクローン検出ツールで正解集合を用意できるため，本研究に容易に適用できる．しかし，Svajlenko ら

[*1] http://verilog2cpp.sourceforge.net

表 1 変換規則一覧
Table 1 Conversion rules.

変換規則	変換前	変換後
1	module < ModuleName > (< Port List >) begin < Description of Module > endmodule	class < Module Name > { < Module Name > (< Port List >) { < Description of Module > }};
2	always @(< Condition >) begin < Description of Always-Statement > end	always @(< Condition >) { < Description of Always-Statement > }
3	if (< Condition >) < Description of If-Statement > end	if (condition) { < Description of If-Statement > }
4	else < Description of If-Statement > end	else { < Description of If-Statement > }
5	case (< Condition >) < Description of Case-Statement > end	case (< Condition >) { < Description of Case-Statement > }
6	< “wire” or “register” > “<=” or “=” { < “wire” or “register” >, < “wire” or “register” > }	< “wire” or “register” > “<=” or “=” (< “wire” or “register” >, < “wire” or “register” >)
7	< bit width >’< Radix >< Constant Value >	< Constant Value in Decimal Notation >

の手法には一般的なプログラミング言語の特徴に依存した部分や，実装上不明瞭な点が存在する．そのため，本研究においては Svajlenko らの手法を一部変更して適用している．以降，変更を加えた評価手法と評価対象について説明した後，評価結果を述べる．

3.2.1 評価手順

Svajlenko らの論文では，抽出するコード片の粒度を関数およびブロック単位とすると述べている．しかし，抽出するブロックの粒度については述べられていないことに加え，Verilog HDL と一般的なプログラミング言語では関数やブロックを対応付けることができない．そのため本研究では抽出するコード片の単位として，Verilog HDL でよく用いられるブロックである **module**，**always**，**if**，**case** の 4 種類のブロックを対象とした．

本研究では，変異の種類について，Svajlenko らの提案するものから**表 2** のように変更を加えている[*2]．表中において，太字ゴシック体で表記されている 10 種類の変異が実際に適用した変異である．Verilog HDL においては文字列リテラルが存在しないため，文字列に対する変異は不採用とした．また，コメント文に関する変異は，複数行コメントの数が多くないため，1 行コメントと複数行コメントをまとめている．加えて，元の手法によるコメントに対する変更方法が明確でなかったため，空白文字に対する変異と同様に追加と削除を適用することとした．さらに，完全に同一のコードクローンも評価対象とするために，変更を加えない変異 **mN** を追加している．

評価対象には，Verilog HDL による RISC-V プロセッサ[*3]の実装である ridecore を選んだ．ridecore はファイル数が 59 個，総行数は 9,868 行で構成されており，Verilog HDL のプロジェクトとしては比較的規模が大きい．これらのコードを対象に，前述の 4 種類のブロックにつき最大

[*2] Svajlenko らの手法には Type3 コードクローンを想定した変異が 5 つ含まれるが，本論文で提案する検出手法は Type3 コードクローンを対象としていないため省略している．
[*3] https://riscv.org/

表 2 変異の種類
Table 2 Types of used mutation.

Clone Type	変異の種類	変異の内容	採否
1	**mN**	**変更なし**	**新規追加**
1	**mCW_A**	**空白文字の追加**	**採用**
1	**mCW_R**	**空白文字の削除**	**採用**
1	mCC_BT	1 行コメントに対する変異	不採用
1	mCC_EOL	複数行コメントに対する変異	不採用
1	**mCC_A**	**コメントの追加**	**新規追加**
1	**mCC_R**	**コメントの削除**	**新規追加**
1	**mCF_A**	**改行の追加**	**採用**
1	**mCF_R**	**改行の削除**	**採用**
2	**mSRI**	**変数名の変更**	**採用**
2	**mARI**	**変数名の入れ替え**	**採用**
2	**mRL_N**	**数値リテラルの変更**	**採用**
2	mRL_S	文字列リテラルの変更	不採用

表 3 埋め込んだ変異コードの数
Table 3 Number of injected mutation code fragment.

	module	always	if	case	合計
Type1	185	190	186	118	679
Type2	89	83	88	53	313
合計	274	273	274	171	992

30 個のコード片を無作為に選択した．**表 3** に作成された変異コード片の数を示す．ソースコード中に 30 個に満たないブロックが存在することに加え，特定の変異が適用できないコード片も存在するため，作成された変異コード片は 992 個となった．これらのコードをランダムかつ文法上適切な箇所に埋め込み，正解集合を構築した．

Precision は正解集合と位置が重なる検出結果のうち，誤検出でない検出結果の件数の割合である．誤検出であるかどうかは，本研究では著者による手作業で確認した．次に，Recall は正解集合の件数に対する，検出できた正解の件数の割合である．埋め込まれた正解と完全に一致する検出結果か，正解を内包する検出結果があれば，その正解は検出

表 4 Precision 測定結果
Table 4 Precision of evaluating clone detection.

Precision	手法 1（提案手法）					手法 2（無変換・plaintext）					手法 3（無変換・C/C++）				
(%)	module	always	if	case	合計	module	always	if	case	合計	module	always	if	case	合計
Type1	100	100	100	97	99	100	100	100	100	100	-	100	100	100	100
Type2	99	100	100	95	98	100	100	100	100	100	-	100	100	-	100
総計	99	100	100	96	99	100	100	100	100	100	-	100	100	100	100

表 5 Recall 測定結果
Table 5 Recall of evaluating clone detection.

Recall	手法 1（提案手法）					手法 2（無変換・plaintext）					手法 3（無変換・C/C++）				
(%)	module	always	if	case	合計	module	always	if	case	合計	module	always	if	case	合計
Type1	92	95	98	89	94	76	42	34	22	46	0	22	20	20	11
Type2	74	94	98	94	89	0	0	0	0	0	0	16	7	0	6
総計	86	95	98	91	93	51	29	24	16	32	0	21	17	0	10

できたものと判定する．

提案する変換の効果を確認するため，提案手法を含む 3 つの手法を評価し比較する．

手法 1（提案手法）
実装した変換ツールを用いて Verilog HDL コードを疑似的な C++コードに変換し，C/C++のソースコードとして CCFinderX に入力する．

手法 2（無変換・plaintext）
Verilog HDL コードを変換せず，plaintext として CCFinderX に入力する．

手法 3（無変換・C/C++）
Verilog HDL コードを変換せず，C/C++のソースコードとして CCFinderX に入力する．

Svajlenko らの手法では，検出ツールの設定は埋め込んだコード片それぞれに適した値を使うとしている．本研究では，**module**，**always** ブロックのコード片に対しては，CCFinderX の規定値のまま最小クローン長を 50，最少トークン種類数を 12 とした．また，**if**，**case** ブロックのコード片は **module** や **always** と比べて短いため，最小クローン長を 25，最少トークン種類数を 12 とした．

3.2.2 評価結果

それぞれの手法で検出した結果の Precision を**表 4** に，Recall を**表 5** に示す．表 4 において，母数が 0 で計算できない場合 '-' と記載している．いずれの表においても，総計は Type1 と Type2 を区別なく集計した結果を示す．

この評価手法においては，各手法の検出結果のうち，抽出元のコード片と埋め込んだコード片のいずれかを含む検出結果に対する，誤検出を除いた検出結果の割合が Precision となる．実際の測定手順としては，抽出元のコード片と埋め込まれたコード片のいずれかと，別のコード片をコードクローンとして検出した結果を，第 1 著者が実際にコードクローンであるか，誤りであるかを判定し，全体から誤りを除いた結果の割合を計算する．したがって，抽出元のコード片と埋め込まれたコード片をコードクローンとして検出した結果は，無条件で正解として扱われる．なお，この判定の基準として，識別子などがまったく異なり，同一の構文が並んでいるだけのようなコード片は検出誤りとした．また，識別子の一部，たとえば bit 位置を示す番号などのみが異なるような，互いに関係のあると判断できるものおよび，文字の並びが完全に一致するものを正解とした．Precision の分母となる，抽出元のコード片と変異コード片のいずれかと位置が重なるコードクローンの検出結果は，提案手法が 1,083 件，手法 2 が 651 件，手法 3 が 355 件であった．なお，手法 2 はトークン化を行わないため，Type2 コードクローンは分断された複数の Type1 コードクローンとして検出される場合がある．また，手法 3 の検出結果は，トークン化に失敗したコード片は検出対象とから除外されるため，本来ひとまとまりのクローン片が複数に分断し検出される場合がある．このため，手法 2 および手法 3 は提案手法と比較し，同じコードクローンを検出していたとしても，検出数が多くなる傾向にある．検出結果のうち，誤検出の可能性がある，正解集合と一致しない検出結果は提案手法では 70 件，手法 2 では 90 件，手法 3 は 0 件であった．手法 3 では CCFinderX がトークン化に失敗するコード片が多数存在するため，検出対象となるコード片が少なく，抽出元のコード片と埋め込まれたコード片のいずれかと，関係のないコード片をコードクローンとして検出される可能性が低い．そのため，この評価において手法 3 は誤検出を含む可能性が低い．今回の実験においては，抽出元のコード片と埋め込まれたコード片の組のみを検出したため，第 1 著者が正誤判断をする必要なく，Precision100%という結果になった．次に，誤検出の可能性がある，提案手法と手法 2 の正解集合と一致しない検出結果が実際にコードクローンか，ただ構文要素の並びが一致しているだけの誤検出か，第 1 著者が手作業で確認した．その結果，手法 2 の誤検出は 0 件で Precision は 100%となった．これは，

手法 2 はソースコードをトークン化せず，文字の並びが完全に一致するコード片のみをコードクローンとして検出するためである．一方で，提案手法は完全一致しない Type2 コードクローンも検出対象としているため，16 件の誤検出を含んでいた．しかし，提案手法は誤検出を含むものの，Precision は Type1，Type2 あわせて 99%と高い精度で正解している．

次に，検出できた正解の数は，提案手法が 919 件，手法 2 が 313 件，手法 3 は 102 件であった．提案手法の Recall は Type1，Type2 ともに他の手法と比べて高く，93%のコードクローンが検出できている．

4. 調査概要

本章では本研究における調査の目的，および対象データと調査手法について述べる．

4.1 調査目的

本調査は，Verilog HDL を用いた回路開発においてコードクローンを対象とした開発支援が必要であるかを，定量的な調査に基づき明らかにすることを目的とする．コードクローンの量が少ない，あるいは単純である場合，開発支援による効率化は高くない．加えて，開発支援が必要である場合，どのような支援が有用であるのかを検討する．

4.2 調査手法

調査手順の概要を**図 3** に示す．調査は以下のリサーチクエスチョンに沿って行った．

RQ1 HDL のコードクローンは一般的に存在するか？
RQ2 HDL のコードクローンは単純なものではないか？
RQ3 HDL のコードクローンはどのような理由で作られるか？

いずれの RQ においても，Verilog HDL のコードクローンについては，提案手法を用いて検出した結果に基づいて調査を行った．なお，3.2.2 項の結果から，提案手法の検出結果は信頼できるものとした．

RQ1，2 では，Verilog HDL と一般的なプログラミング言語のコードクローンに関するメトリクスの比較を行う．メトリクスを比較するため，C と Java についても提案手法で利用する CCFinderX を用いてコードクローンを検出する．CCFinderX の設定オプションは規定値を用いた．なお，CCFinderX がエラーになることがあるため，C のプロジェクトにおいては検出対象からヘッダファイルを除外する．メトリクスの差異は p 値 0.05 未満として Steel-Dwass の方法 [19] を用いて有意差検定を行った．Steel-Dwass の方法は 3 群以上のノンパラメトリックなデータに対して利用できる順位による多重比較である．

RQ3 では，Verilog HDL のプロジェクトの検出結果を観察することで，コードクローンがどのような理由で作られているのか，定性的な調査を行う．

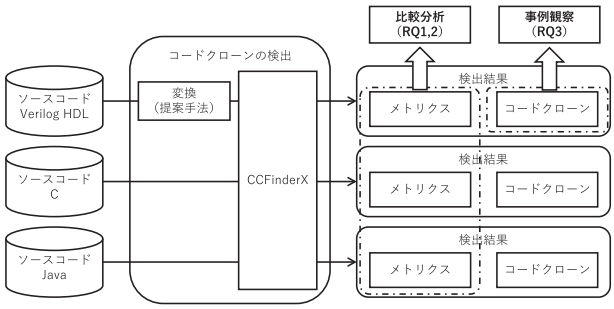


図 3 調査手順概要
Fig. 3 Overview of the investigation procedure.

表 6 分析対象概要
Table 6 Subjects overview.

		Verilog HDL	C	Java
ファイル数	総計	1,101	31,626	22,254
	最大値	316	24,296	6,623
	最小値	6	22	11
	中央値	28	175	453
LOC	総計	258,627	11,809,648	1,738,007
	最大値	91,397	9,357,449	478,285
	最小値	791	10,897	888
	中央値	6,083	87,105	29,106

4.3 調査対象

Verilog HDL，C，Java それぞれ 20 件のプロジェクトを調査，比較対象として選んだ．Verilog HDL については，まず，オープンソースの回路開発を支援するコミュニティである OpenCores[*4]で公認されているプロジェクトから開発状況が Stable のものを選択した．この条件を満たすものが 17 件のみであったため，それらに加えて，Github[*5]の検出結果から Star 数の多いものを 3 件選出した．OpenCores のプロジェクトを開発状況が Stable のものに限定したのは，仕様策定のみで，実装が進んでいないプロジェクトを除外するためである．C と Java は，Github の言語ごとの検索結果から，Star の数が多い上位 20 件のプロジェクトを選択した．対象としたプロジェクトの概要を**表 6** に示す．なお，各プロジェクトのデータは 2017 年 4 月 20 日の時点で取得した．

5. RQ1：HDL のコードクローンは一般的に存在するか？

この章では Verilog HDL においてコードクローンが広く存在するかをメトリクスに基づき調査する．

5.1 動機と調査方法

これまでに，Verilog HDL のソースコード中にコード

*4 http://opencores.org/
*5 https://github.com/

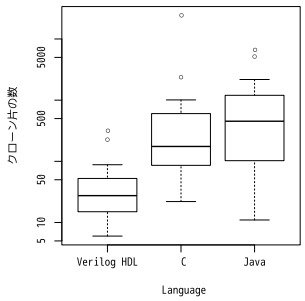


図 4 プロジェクトごとのクローンセットの数
Fig. 4 Number of clone set in each projects.

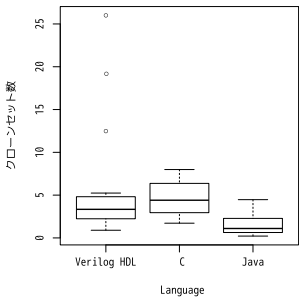


図 5 ファイルあたりのクローン片の数
Fig. 5 Number of clone fragment in files.

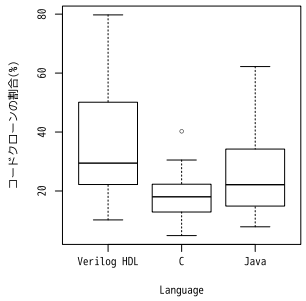


図 6 コード全体に対するコードクローンの割合
Fig. 6 Ratio of code clones in source code.

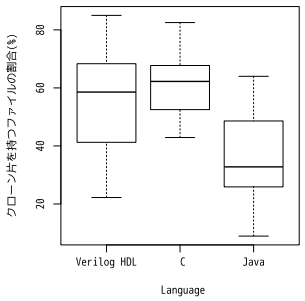


図 7 クローン片を 1 つ以上含むファイルの割合
Fig. 7 Ratio of files including clone fragment.

クローンが一般的に存在するかは明らかにされていない．コードクローンが存在していても，特定のプロジェクト，あるいは特定のファイルにしか存在しない場合，開発支援の対象としても効果は薄い．したがって RQ1 では，C および Java と比較する形で，プロジェクトごとのコードクローンの数，クローン率，クローン片を持つファイルの割合について調査を行う．

5.2 結果

測定されたメトリクスについて，項目ごとに結果を述べる．

5.2.1 コードクローンの量

コードクローンの量を示すメトリクスとして，プロジェクト中に存在するクローンセットの数と，ファイルごとのクローン片の数に着目した．クローンセットとは，互いに類似するコード片の集合のことを指す．また，クローン片とは，いずれかのクローンセットに含まれるコード片のことを指す．プロジェクトごとのクローンセット数を**図 4** に示す．また，**図 5** はプロジェクトごとの 1 ファイルあたりのクローン片の数の平均値を示す．Verilog HDL のクローンセットの数は C と Java に対して有意に少ない．これは，Verilog HDL は C や Java と比べてプロジェクトの規模自体が小さいためと考えられ，1 ファイルあたりのクローン片の数でみると Java より多く存在する．

5.2.2 コードクローンの割合

図 6 はコード全体に対するコードクローンの割合の平均値の分布を示す．Verilog HDL のプロジェクトでは，多いものだと 8 割がコードクローンの一部に含まれている．また Verilog HDL のプロジェクトは C 言語のプロジェクトに対し，有意にコードクローンの割合が高い．Verilog HDL と Java で有意差はなく，同程度のコードクローンの割合を持つことが確認できる．

5.2.3 クローン片を持つファイルの割合

図 7 はプロジェクトごとの，少なくとも 1 つ以上クローン片を含むファイルの割合の分布を示す．Verilog HDL と C は Java と比べて有意に高く，多くのプロジェクトにおいて 4–7 割のファイルがクローン片を含む．

5.3 RQ1 への解答

プロジェクト規模が小さいこともありコードクローンの絶対数は少ないが，ファイルあたりのクローン片の数は C や Java と比べ少なくない．Verilog HDL はファイル数の母数自体が少ないため，コードクローンが存在する可能性が同程度であれば，ファイルあたりのクローン片の数は必然的に高くなる．逆に，仮に Verilog HDL においてコードクローンが存在する可能性が低い場合，ファイル数にかかわらずファイルあたりのクローン片の数は少なくなる．このことから，Verilog HDL においてコードクローンが C や Java と比べて著しく少ないとはいえない．また，コード全体を占めるクローンの割合は少ないもので 10%，多いものでは 80%と，C に対して有意に高い傾向にあることが確認できた．コードクローンを持つファイルの割合も C とは差がなく，Java と比べて有意に多いことが確認できた．Java のコードクローンは特定のファイルに偏る傾向があることが分かっている [14]．そのため，プロジェクトごとの 1 ファイルあたりのクローン片の数の平均や，クローン片を含むファイルの割合が低くなると考えられる．一方で，Verilog HDL が有意に高いのは，多くのファイルがクローン片を含み，偏りなく存在するためと考えられる．

このことから，C や Java と同様に，Verilog HDL においてもコードクローンが一般的に存在するといえる．

6. RQ2：HDL のコードクローンは単純なものではないか？

この章ではコードクローンの複雑さについて，メトリクスに基づき調査する．

6.1 動機と調査方法

RQ1 では Verilog HDL においてもコードクローンが一般的に存在することを確認した．しかし，それらが単純な

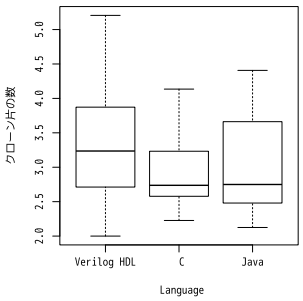


図 8 クローンセットに含まれるクローン片の数

Fig. 8 Number of clone fragments included clone sets.

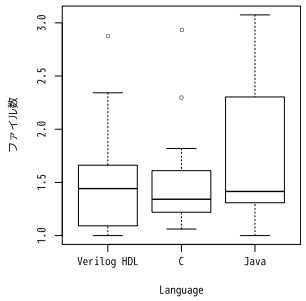


図 9 クローンセットに関係するファイルの個数

Fig. 9 Number of files related to clone set.

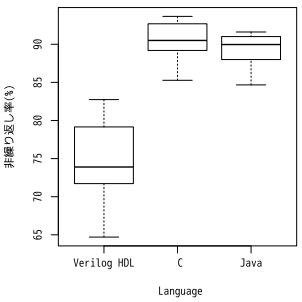


図 10 ファイルの非繰返し率

Fig. 10 Ratio of non repeated code in file.

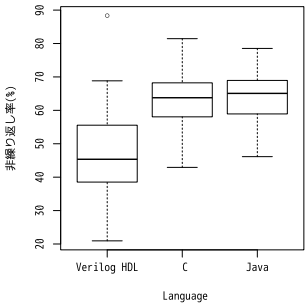


図 11 クローンセットの非繰返し率

Fig. 11 Ratio of non repeated code in clone fragment.

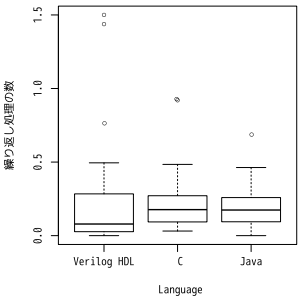


図 12 クローン片に含まれるループの数（LOOP）

Fig. 12 Number of loops in clone fragment.

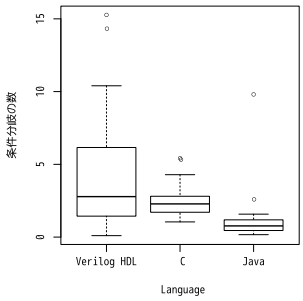


図 13 クローン片に含まれる条件分岐の数（COND）

Fig. 13 Number of conditional branches in clone fragment.

ものであるか，あるいは複雑なものであるかによって，コードクローンを対象とした開発支援の重要性は異なる．たとえば，単純な定義文が並ぶコードクローンなどは多く存在しても開発効率に大きな影響を与えず，支援の必要性が薄いとされる．逆に複雑なコードクローンや，複数のファイル間で分散しているコードクローンが多く存在する場合，開発効率の低下につながる．そのため RQ2 では，C，Java と比較する形で，クローンのファイル内外の散らばりや非繰返し率，コード複雑度に着目し，コードクローンの複雑さについて調査を行う．

6.2 結果

測定したメトリクスについて，項目ごとに結果を述べる．

6.2.1 クローンセットに含まれるクローン片の数

クローンセットに含まれるクローン片の数が多ければ，それだけ管理は煩雑になる．**図 8** はクローンセットに含まれる数の平均値のプロジェクトごとの分布を示す．有意差検定の結果，差は見られなかったが，多くのプロジェクトで平均的なクローンセットに含まれるクローン片の数が 3 つより多いことが確認できる．また，特に多いプロジェクトでは，クローンセットが平均 5 つ以上のクローン片で構成されている．

6.2.2 クローンセットと関係するファイル

図 9 は各クローンセットを構成するクローン片を含むファイルの数のプロジェクトごとの平均値を示す．この値は C，Java との間に有意差はなく，いずれの言語も平均 1.5 個程度のファイルにまたがるクローンセットを持つプロジェクトが多い．

6.2.3 非繰返し率

非繰返し率は CCFinderX が計算する，類似するコード片が連続している割合である．たとえば，変数宣言などが続くコードの場合，非繰返し率は低くなる．ファイルごとの非繰返し率の平均値の分布とクローンごとの非繰返し率の平均値の分布をそれぞれ**図 10**，**図 11** に示す．この結果から，C や Java と比べて Verilog HDL のコードは非繰返し率が著しく低いことが確認できる．この傾向はファイル全体で見てもコードクローンのみに着目しても同様である．

6.2.4 循環的複雑度

コードの複雑さを表すメトリクスとして，McCabe の循環的複雑度 [20] がよく利用される．循環的複雑度は，コード片に含まれる for 文や while 文といった繰返し文の数である LOOP と，if 文や case 文などの条件分岐の数である COND から計算される．CCFinderX ではクローンセットごとに LOOP と COND の値を計測する．LOOP と COND の平均値をプロットした図を**図 12**，**図 13** に示す．LOOP の値はいずれの言語でも大きな違いがない．一方，COND の値は Verilog HDL と C のプロジェクトは Java に対して有意に大きい．また Verilog HDL のプロジェクトでは平均 10 個以上の条件分岐を持つプロジェクトも存在する．

6.3 RQ2 への解答

まず，クローンセットに含まれるクローン片の数と，位置的な分散に着目した．クローン片の数が多い，あるいは複数のファイルにまたがるようなコードクローンは，変更漏れなどにつながりやすいと考えられる．調査の結果，クローンセットに含まれるクローン片の数，ファイルの分散

ともに C，Java と比べて大きな違いは見られなかった．

次に，コードの複雑度に着目する．コードクローン箇所の非繰返し率は C，Java と比較して Verilog HDL は著しく低いことが確認できる．コードの非繰返し率はクローン片に限らずソースコード全体でみても C や Java より低い．これは Verilog HDL がレジスタや配線とその接続関係から回路構造を定義する特徴上，代入文の連続が多くなることに起因していると考えられる．一般に，非繰返し率が低いコードクローンは単純な定義文の並びである可能性が高く，そのようなコードクローンは支援の重要性が低いといわれている [21]．しかし，ソースコード全体の非繰返し率とコードクローンの非繰返し率を比較すると Verilog HDL も C や Java 同様に低くなる傾向が確認できる．このことから，Verilog HDL におけるコードクローンがソースコード全体のうち特に単純な箇所でのみ作られているわけではないといえる．また，Verilog HDL のコードクローンは，C や Java と比べて有意に条件分岐の数が多いことが確認できた．条件分岐やループの数が多いコードは複雑でソースコードの理解や保守を困難にするといわれている．Verilog HDL は回路構造をそのまま表現するため，必然的にコード中の条件分岐は多くなると考えられるが，コードクローン中にも条件分岐を多数含む結果となった．

Verilog HDL のコードクローンは繰返しを多く含むが，これは Verilog HDL コード全体の特徴であり，ソースコード中の単純な箇所のみがコードクローンになっているわけではない．また，クローン片は多数の箇所に散らばっており，条件分岐を多数含む．これらから，Verilog HDL のコードクローンは開発支援を要さない単純なコードクローンばかりではないといえる．

7. RQ3：HDL のコードクローンはどのような理由で作られるか？

この章では Verilog HDL でどのような目的でコードクローンが作られているかを調査する．

7.1 動機と調査手法

どのようなコードクローンが作られているかによって，必要となる支援方法は異なる．Kapser らはコードクローンが作成された動機を基にパターンを分類し，パターンごとにメリットとデメリットを検討したうえで，求められる管理方法について議論している [22]．本研究では，Verilog HDL のソースコードにおいて，どのような目的でコードクローンが作られており，それらが一般的なプログラミング言語とどのように異なるかを調査する．そのため，20 件の Verilog HDL のプロジェクト中から検出されたコードクローン全 4,276 件を対象に，コードクローンの事例を観察し，その特徴を分析する．

```
1   mvx_H0_diff_a <= (MB_inter_size != `I8x8)? 0:mvx_CurrMb2[7:0];
2   mvx_H0_diff_b <= (MB_inter_size != `I8x8)? 0:mvx_CurrMb2[23:16];
3   mvx_H1_diff_a <= (MB_inter_size != `I8x8)? 0:mvx_CurrMb2[15:8];
4   mvx_H1_diff_b <= (MB_inter_size != `I8x8)? 0:mvx_CurrMb2[31:24];
5   mvx_H2_diff_a <= (MB_inter_size != `I8x8)? 0:mvx_CurrMb3[7:0];
6   mvx_H2_diff_b <= (MB_inter_size != `I8x8)? 0:mvx_CurrMb3[23:16];
7   mvx_H3_diff_a <= (MB_inter_size != `I8x8)? 0:mvx_CurrMb3[15:8];
8   mvx_H3_diff_b <= (MB_inter_size != `I8x8)? 0:mvx_CurrMb3[31:24];
9
10  mvy_H0_diff_a <= (MB_inter_size != `I8x8)? 0:mvy_CurrMb2[7:0];
11  mvy_H0_diff_b <= (MB_inter_size != `I8x8)? 0:mvy_CurrMb2[23:16];
12  mvy_H1_diff_a <= (MB_inter_size != `I8x8)? 0:mvy_CurrMb2[15:8];
13  mvy_H1_diff_b <= (MB_inter_size != `I8x8)? 0:mvy_CurrMb2[31:24];
14  mvy_H2_diff_a <= (MB_inter_size != `I8x8)? 0:mvy_CurrMb3[7:0];
15  mvy_H2_diff_b <= (MB_inter_size != `I8x8)? 0:mvy_CurrMb3[23:16];
16  mvy_H3_diff_a <= (MB_inter_size != `I8x8)? 0:mvy_CurrMb3[15:8];
17  mvy_H3_diff_b <= (MB_inter_size != `I8x8)? 0:mvy_CurrMb3[31:24];
```

図 14 代入文のコードクローンの例 1
Fig. 14 Example of code clone of assignment statement 1.

```
1   assign next_refill_adr = (OPTION_ICACHE_BLOCK_WIDTH == 5) ?
2                            {wradr_i[31:5], wradr_i[4:0] + 5'd4} : // 32 byte
3                            {wradr_i[31:4], wradr_i[3:0] + 4'd4};  // 16 byte
```

図 15 代入文のコードクローンの例 2
Fig. 15 Example of code clone of assignment statement 2.

7.2 コードクローンのパターン

コードクローンの検出事例を観察した結果，特徴的なコードクローンが存在し，文法上の粒度によってパターンとして分類することができた．以降，観察した 4 つのパターンについて述べる．

7.2.1 代入文のコードクローン

類似する目的を持ったレジスタに対する値の設定や配線の接続の定義が多数コードクローンとして記述されていた．**図 14** の例では，1 行目から 8 行目と，10 行目から 17 行目のレジスタに対するノンブロッキング代入文がコードクローンとして検出されている．

また**図 15** のような配線に対する継続的代入文は，同一のコード片がコードクローンとして異なるファイルで検出されている．一般的なプログラミング言語の場合，類似する変数の集合は配列やクラスなどで表すことで，まとめて記述することができる．しかし Verilog HDL の場合は回路の構造を定義する目的から，それぞれを個別に定義する必要があるためこのようなコードクローンが多数作られる．

このようなコード片に対して変更が必要なときは，対応するクローン片を同時に変更する必要がある場合がある．たとえば，図 14 の場合，識別子は各クローン片で個別に変更される場合もあるが，$mvx\_CurrMb2[7:0]$ の括弧内に記述されているビット位置が変更になる場合などは同様に変更する必要がある．

7.2.2 順序回路のコードクローン

always 文による順序回路を定義するコード片の単位でコードクローンになっている例も確認できた．一例として，**図 16** のコード片と類似するコード片が複数個所に記述されている．always 文は信号の立ち上がりなどを条件に動作する順序回路を定義するための文である．この例では clk 信号の立ち上がり時にレジスタの値を更新する回路が記述されている．類似するレジスタの集合がある場合，

```
1   always @(posedge clk)
2           begin
3                   text_out_stage9[127:120] <=  sa00_next_round10;
4                   text_out_stage9[095:088] <=  sa01_next_round10;
5                   text_out_stage9[063:056] <=  sa02_next_round10;
6                   text_out_stage9[031:024] <=  sa03_next_round10;
7                   text_out_stage9[119:112] <=  sa10_next_round10;
8                   text_out_stage9[087:080] <=  sa11_next_round10;
9                   text_out_stage9[055:048] <=  sa12_next_round10;
10                  text_out_stage9[023:016] <=  sa13_next_round10;
11                  text_out_stage9[111:104] <=  sa20_next_round10;
12                  text_out_stage9[079:072] <=  sa21_next_round10;
13                  text_out_stage9[047:040] <=  sa22_next_round10;
14                  text_out_stage9[015:008] <=  sa23_next_round10;
15                  text_out_stage9[103:096] <=  sa30_next_round10;
16                  text_out_stage9[071:064] <=  sa31_next_round10;
17                  text_out_stage9[039:032] <=  sa32_next_round10;
18                  text_out_stage9[007:000] <=  sa33_next_round10;
19
20          end
```

図 16　順序回路のコードクローンの例
Fig. 16　Example of code clone of always block.

```
1   if ((Auto_precharge[0] == 1'b1) && (Read_precharge[0] == 1'b1)) begin
2           if ((($time - RAS_chk0 >= tRAS) &&// Case 2
3                   ((Burst_length_1 == 1'b1 && Count_precharge[0] >= 1) ||// Case 1
4                    (Burst_length_2 == 1'b1 && Count_precharge[0] >= 2) ||
5                    (Burst_length_4 == 1'b1 && Count_precharge[0] >= 4) ||
6                    (Burst_length_8 == 1'b1 && Count_precharge[0] >= 8))) ||
7                   (RW_interrupt_read[0] == 1'b1)) begin// Case 3
8                           Pc_b0 = 1'b1;
9                           Act_b0 = 1'b0;
10                          RP_chk0 = $time;
11                          Auto_precharge[0] = 1'b0;
12                          Read_precharge[0] = 1'b0;
13                          RW_interrupt_read[0] = 1'b0;
14                          if (Debug) $display ("at time %t NOTE : Start Internal Auto
                            Precharge for Bank 0", $time);
15          end
16  end
```

図 17　条件分岐のコードクローンの例 1
Fig. 17　Example of code clone of conditional branch 1.

```
1   else if (word_select == 2'b01) begin
2           if ( (Cmd_Cnt>=40) && (Cmd_Cnt<72) )begin
3                   word_select_counter<= word_select_counter+1;
4                   Out_Buff[31-word_select_counter] = cmd_dat_i;
5           end
6
7   end
8   else if  (word_select == 2'b10) begin
9           if ( (Cmd_Cnt>=72) && (Cmd_Cnt<104) )begin
10                  word_select_counter<= word_select_counter+1;
11                  Out_Buff[31-word_select_counter] = cmd_dat_i;
12          end
13  end
14  else if  (word_select == 2'b11) begin
15          if ( (Cmd_Cnt>=104) && (Cmd_Cnt<128) )begin
16                  word_select_counter<= word_select_counter+1;
17                  Out_Buff[31-word_select_counter] = cmd_dat_i;
18          end
19  end
```

図 18　条件分岐のコードクローンの例 2
Fig. 18　Example of code clone of conditional branch 2.

always 文単位のコードクローンとして記述される．このようなコードクローンは前述の代入文のコードクローンと同様，まとめて記述するのは困難である．

7.2.3　条件分岐のコードクローン

前述の always 文単位のコードクローンより複雑な，信号やレジスタの状態によって動作が変わる順序回路の定義は，if 文などの単位でコードクローンとして記述されることが多い．**図 17** は 1 行目から 16 行目と類似するコード片が複数個所に存在し，条件によってレジスタに異なる値が代入されるコードクローンである．クラスなどを用いた抽象的な記述ができない Verilog HDL においては，このようなコードクローンの集約は困難である．

また，**図 18** の例では，1 行目から 7 行目，8 行目から 13 行目，14 行目から 19 行目がそれぞれクローン片であり，条件は異なるが同様の処理が記述されている．この例は，ソフトウェア開発者の視点から見ると，if 文の条件式を論理演算でまとめることで，コードクローンを集約することができるように思える．しかし，個別にコードクローンとして記述した場合と，集約した場合では生成される回路が異なる．前者ではそれぞれの信号の条件ごとに同様の処理を行う回路が複数生成され，後者の場合は 1 つの回路が生成される．回路の構造が異なるということは，回路面積や処理速度が異なることを意味する．このようなコードクローンは意図的に集約せず，分割して記述されていると考えられる．

7.2.4　ファイル/モジュールのコードクローン

ファイルや module ブロック単位でのコードクローンも多数存在する．たとえば並列化された回路はファイルや module 単位でコードクローンとして記述される．このような場合，具体的な計算を行う回路の定義は単一のコード片になる．しかし，その処理を行う回路を定義した module のインスタンス生成と接続の定義は，並列化する数だけ別のファイルや module として記述され，コードクローンになる．

また，回路開発では複数の仕様を同時に開発する場合がある．たとえばメモリコントローラなどは bit 数の異なる回路を同時に開発している．これは，その回路を利用した回路を作る場合に FPGA 上の面積の余裕と必要な性能に応じて適切な回路規模を選べるようにするためである．このような場合も，互いにコードクローンを複数持つ類似するファイルが仕様ごとに作られる．

このようなコードクローンは Verilog HDL の言語仕様上，集約することはできない．また，変更時には同時に変更する必要がある箇所も多く存在する．

7.3　RQ3 への解答

Verilog HDL では，クラスや関数を用いるような，抽象的な処理の記述ができないため，類似するレジスタへの代入などはコードクローンとして記述されている．また，同一の記述に分岐する if 文など，ソフトウェア開発者の視点からすると集約できるようなコード片もコードクローンとして記述されている．これは，Verilog HDL のソースコードがレジスタや配線およびその接続関係を定義するものであり，回路の構造そのものを表していることに起因する．したがって，コード片を集約すると，回路構造が変わり，回路面積や処理速度などに影響が出るため，開発者は意図的にこのようなコードクローンを記述していると考えられる．

また，FPGA の開発では目的に応じて回路の規模を変えられるように，性能の異なる複数の回路を定義することがある．性能の異なる回路の例として，たとえばメモリコントローラのビット数の違いや，CPU のパイプライン処理

の並列数の違いなどがあげられる．このような複数の回路の開発ではファイルやモジュール単位のコードクローンが多数作られる．

これらより，Verilog HDL におけるコードクローンは，回路構造を定義するという特徴上類似するコードを記述する必要があるほか，回路特有の開発形態の事情からコードクローンが作られているといえる．

8. 考察

本章では 8.1 節で検出手法，8.2 節でコードクローンの管理，開発支援の必要性について考察する．加えて 8.3 節で妥当性の脅威について検討する．

8.1 コードクローン検出手法

本研究では Verilog HDL を対象としたコードクローン検出手法を提案し，検出精度の評価を行った．提案手法は，簡易な変換を用いて Verilog HDL のコードを疑似的な C++ コードに変換することで，既存ツールである CCFinderX を用いてコードクローンを検出する．

変異コードを埋め込み，明示的な正解集合を作る手法を用いた検出精度の評価を行った結果，提案手法は Precision，Recall ともに 90%以上の精度を持つことが確認できた．提案手法を用いず，未変換の Verilog HDL コードを C/C++コードや，plaintext して入力して検出した場合は，Precision は 100%になるが，Recall はそれぞれ 10%，32%と著しく低い．このことから，提案手法におけるコードの変換が，CCFidnerX におけるコードのトークン化に貢献していることが分かる．

一方で，検出誤りや検出できなかったコードクローンも存在する．提案手法の検出した 16 件の誤りは，識別子はまったく異なるが，文法要素の並びが一致しているため検出されたものである．また，我々の先行研究においては，実際のコードクローン検出結果の Precision を手作業で測定しており，20%がこのような誤検出だった [11]．CCFinderX は，Type2 コードクローンを検出するために識別子や数値を正規化する．そのため，識別子が異なっていても，文法要素の並びが一致しているコード片は，コードクローンとして検出される．したがって，提案手法におけるコードクローンの検出誤りは，CCFinderX の限界であるといえる．

また，提案手法は埋め込んだ 992 個の正解集合のうち，73 件のコードクローンを検出できなかった．提案手法は CCFinderX がパースできるように Verilog HDL コードを変換している．しかし，すべての文法要素の対応がとれるように変換しているわけではないため，意図しない箇所でトークンが分割されることがある．本来コードクローンを構成するコード片の中央でトークンが余分に分割された場合，検出結果は前半分と後ろ半分に分かれて 2 つ検出される場合がある．本研究では，正解に対して完全一致するか，内包する検出結果が存在した場合に，その正解を検出できたとして扱っている．そのため，1 つの正解に対して分割されてコードクローンが検出されていると，その正解は検出できていないものとして Recall を計算する．

コードクローンの分析調査において，検出ツールの Precision と Recall がどの程度であれば十分とするか，絶対的な評価基準について定まった見解は存在しない．Cheung らの研究では，提案する Javascript 中のクローン検出ツールの精度を，埋め込んだ変異コード片を正解集合とする手法で評価し，Precision が 96%，Recall が 87%であるとしたうえで調査分析をしている [23]．

Svajlenko らは，変異コード片を埋め込む手法により，コードクローン検出ツールの比較評価を行っている [24]．彼らの調査では，CCFinderX の Recall は Type1 コードクローンについては C 言語，Java ともに 90%以上，Type2 コードクローンについては C 言語が 75%，Java が 60%とされている．提案手法はコードクローンの検出に CCFinderX を利用しているため，検出精度は CCFinderX に依存するが，我々の提案手法の評価結果は Svajlenko らの調査した結果と比べて高い．Svajlenko らの論文では検出できなかったコードクローンの詳細は記述されていないため，この結果の違いの原因について詳細に議論することはできない．本研究において Verilog HDL 向けに変更した，変異の種類や変異コード片の粒度が影響を及ぼしている可能性は否定できない．

提案手法における Verilog HDL コードの変換は，ブロックの開始や終了の記述を C++に合わせ，式中などに C++で使われない，あるいは C++の中でブロックの定義に用いられるような文字を含む場合にそれを除去するものである．このようなコードに単純な変換を適用することで既存ツールに適応させる手法は，Verilog HDL に限らず C++と類似するような言語であれば，たとえば Verilog HDL と並んでよく利用される HDL である VHDL や，他のプログラミング言語に対しても応用が可能である．このような手法は，簡易にコードクローンを検出し，分析あるいは開発を支援する手法として有用であると考える．

8.2 コードクローンの管理・開発支援の必要性

RQ1 および RQ2 において，コードクローンの量と複雑さの観点から一般的なプログラミング言語と比較を行った．その結果，絶対数は少ないものの，コードクローンの割合は C や Java と同程度であることが分かった．コードクローンになっているコード片は，非繰返し率が低いことが確認された．一般に，非繰返し率が低いコードクローンは単純な定義文の繰返しなどの，支援の必要性が低いコードであるといわれている．しかし，観察の結果，Verilog HDL においては図 14 や図 15 に示す例のような，類似する bit 列

への代入文の記述が非繰返し率を低下させる要因になっていた．この例は 7.2.1 項で述べたように，同時変更を必要とする場合などがあり，非繰返し率の低さは Verilog HDL において必ずしも支援の必要性の低さを示さないと考えられる．また，Verilog HDL では図 17 や図 18 に例を示すコードクローンのように，条件分岐が多く含まれることが分かった．このような条件分岐は Verilog HDL に特有の回路構造を直接記述するために作られている．一般に，条件分岐の多いソースコードは複雑で，理解を妨げるといわれている [20]．一方で，繰返し率が低く条件分岐の多いコードは，単純な条件分岐の連続であり，開発者にとって興味深くないコードクローンであるともいわれている [21]．しかし，図 18 は非繰返し率が低く繰返し率が高い例の 1 つであるが，回路の構造を考慮して，同一の処理を条件ごとに分割して記述しており，これらのコードは同時編集が必要である場合がある．このことから，Verilog HDL では非繰返し率と条件分岐の数からは一概に興味深いコードクローンでないとはいえないと考える．さらに，これらのコードクローンは複数のファイルや位置に分散している．このように，条件分岐が多く複雑で，かつそれらのコードクローンの位置が分散している場合，コードの修正を阻害する要因になりうる．このことから，Verilog HDL においても，コードクローンに関する管理や開発支援が開発効率の向上に貢献できると考える．

RQ3 ではコードクローンが作られる目的に着目し，検出事例を観察した．コードクローンは，回路構造を定義する目的上，必要性があり作られている．類似するレジスタが複数ある場合などは，ほぼ同一のコード片が複数記述される．また，動作条件が複数ある場合に，条件ごとに同一のコード片を複数記述している例も見られた．これらのコード片は，変更内容によっては同時に編集する必要がある．したがって，コードクローンのドキュメント化や同時編集の支援などは開発者支援に貢献できる．

一方で，リファクタリングによるコード片の集約は慎重に行う必要がある．これは Verilog HDL のコードが回路の動作を記述しているわけではなく，レジスタや配線，およびその接続関係を記述しているため，リファクタリングによる集約などのコードの編集が，回路の構造および性能に直接を及ぼすためである．リファクタリングによる集約を行うには，回路性能とのトレードオフを考慮する必要がある．

8.3 妥当性の検討

本研究では提案手法を用いた Verilog HDL のコードクローン検出結果と，CCFinderX を用いた C と Java のコードクローンの検出結果を対象に分析を行った．したがって分析したコードクローンは提案する変換手法と，CCFinderX の検出結果に依存している．CCFinderX 以外の検出結果を用いた場合，異なる分析結果になる可能性がある．しかし，提案手法の検出精度は Precision，Recall ともに 90%以上であることを確認している．このことから Type1，Type2 コードクローンの特徴においては検出手法の影響は大きくないと考える．一方で，CCFinderX は Type3 コードクローンを検出できないため，Type3 コードクローンを含めた分析を行うと異なる傾向がみられるかもしれない．

また，8.1 節で述べたように，提案手法は構文要素の並びが一致しているだけのコード片もコードクローンとして検出している．このような検出結果は，実際には関係のないコード片であるため，定量的なデータの測定に影響を及ぼしている可能性がある．しかし，これは Type2 コードクローンを検出するうえでの CCFinderX の仕様による限界であり，C 言語や Java におけるコードクローン検出にも同様に影響を及ぼしている．したがって，同条件で比較できており，分析結果に大きな影響はないと考える．また，定性的な分析においては，手作業で検出されたコードクローンの分類を行っているため，誤検出の影響はないと考える．

本研究では，コードクローンに関する量的調査において，C 言語についてはヘッダファイルを除外した検出結果で分析を行っている．これは，CCFinderX が，一部のヘッダファイルを読み込んだ際に異常終了する場合があるためである．ヘッダファイルを読み込んだ時点で動作を停止するため，どのヘッダファイルでエラーが発生するかは事前には判断できず，対象ファイルのみを除外するのは困難である．CCFinderX は C 言語のヘッダファイルに主に記述されている関数のプロトタイプ宣言や構造体の定義を検出対象から除外するが，ヘッダファイルに関数定義が記述されている場合は検出対象となる．そのため，ヘッダファイルを除外せず分析した場合，RQ1 および RQ2 におけるメトリクスの計測結果に違いが出る可能性がある．この影響を調べるには，エラーが発生しないプロジェクトのみを対象に分析する必要があり，今後の課題である．

本研究ではオープンソースの回路プロジェクトとソフトウェアプロジェクトを対象としている．対象プロジェクトは各言語 20 件ずつ，それぞれの目的や対象は幅広く選択されているため，プロジェクトの性質による調査結果への影響は小さいと考える．一方で，オープンソースではない産業プロジェクトを対象とすると異なる傾向がみられる可能性がある．

9. おわりに

本研究の貢献をまとめる．まず，Verilog HDL を対象としたコードクローン検出手法の提案し，その検出精度を評価した．提案手法は Precision，Recall ともに 90%を超える精度でコードクローンを検出することができた．

次に，C および Java と比較する形で，Verilog HDL におけるコードクローンの定量的な調査を行った．その結果，

Verilog HDL においても C や Java と同様にコードクローンが存在し，支援を要することを明らかにした．

検出された事例を観察した結果，Verilog HDL においては，リファクタリングによる集約は回路性能への影響を考慮したうえで行う必要性があることが分かった．一方，ドキュメントへの記録や，変更時の同時編集の支援などは一般的なプログラミング言語と同様に有効である．

今後の課題として，Verilog HDL を用いた開発における，具体的なコードクローンを対象とした開発支援手法を構築する必要がある．提案する検出手法を用いることで，クローン片の編集漏れなどを防ぐ支援は実現可能である．一方で，リファクタリングによるクローン片の集約には課題がともなう．Verilog HDL で記述されたソースコードは回路の動作と構造を記述したものであり，コードクローンを集約することによる性能への影響が一般的なプログラミング言語と比べて大きいと予想される．しかし，実際にどのような集約がどの程度性能に影響を及ぼすかは明らかにできていない．したがって，コードクローンの変更履歴や，回路のシミュレーションや合成の結果を合わせた分析を行い，集約によって面積や速度といった回路性能にどの程度影響があるかを調査する必要がある．また，ヘッダファイルを除外することによる影響の調査も今後の課題である．

参考文献

[1] Neshatpour, K., Malik, M., Ghodrat, M.A., et al.: Energy-Efficient Acceleration of Big Data Analytics Applications Using FPGAs, *Proc. BigData'15*, pp.115–123 (2015).

[2] Wang, C., Gong, L., Yu, Q., et al.: DLAU: A Scalable Deep Learning Accelerator Unit on FPGA, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.36, No.3, pp.513–517 (2017).

[3] Sudakrishnan, S., Madhavan, J., Whitehead, Jr., E.J., et al.: Understanding Bug Fix Patterns in Verilog, *Proc. MSR'08*, pp.39–42 (2008).

[4] Duley, A., Spandikow, C. and Kim, M.: Vdiff: A program differencing algorithm for Verilog hardware description language, *Automated Software Engineering*, Vol.19, No.4, pp.459–490 (2012).

[5] Nacif, J., Silva, T., Vieira, L., et al.: Tracking hardware evolution, *Proc. ISQED'11*, pp.1–6 (2011).

[6] Baxter, I.D., Yahin, A., Moura, L., et al.: Clone Detection Using Abstract Syntax Trees, *Proc. ICSM'98*, pp.368–377 (1998).

[7] Roy, C.K., Cordy, J.R. and Koschke, R.: Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach, *Science of Computer Programming*, Vol.74, No.7, pp.470–495 (2009).

[8] Zeller, A.: *Why Programs Fail*, Morgan Kaufmann Publishers (2005).

[9] Duala-Ekoko, E. and Robillard, M.P.: Clonetracker: Tool Support for Code Clone Management, *Proc. ICSE'08*, pp.843–846 (2008).

[10] 山中裕樹，崔　恩瀞，吉田則裕ほか：コードクローン変更管理システムの開発と実プロジェクトへの適用，情報処理学会論文誌，Vol.54, No.2, pp.883–893 (2013).

[11] Uemura, K., Mori, A., Fujiwara, K., et al.: Detecting and analyzing code clones in HDL, *Proc. IWSC'17*, pp.1–7 (2017).

[12] Cong, J., Liu, B., Neuendorffer, S., et al.: High-Level Synthesis for FPGAs: From Prototyping to Deployment, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.30, No.4, pp.473–491 (2011).

[13] Nane, R., Sima, V.M., Pilato, C., et al.: A Survey and Evaluation of FPGA High-Level Synthesis Tools, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.35, No.10, pp.1591–1604 (2016).

[14] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).

[15] Jiang, L., Misherghi, G., Su, Z., et al.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *Proc. ICSE'07*, pp.96–105 (2007).

[16] Bellon, S., Koschke, R., Antoniol, G., et al.: Comparison and Evaluation of Clone Detection Tools, *IEEE Trans. Softw. Eng.*, Vol.33, No.9, pp.577–591 (2007).

[17] Svajlenko, J., Roy, C.K. and Cordy, J.R.: A Mutation Analysis Based Benchmarking Framework for Clone Detectors, *Proc. IWSC'13*, pp.8–9 (2013).

[18] Fowler, M.: *Refactoring: Improving the design of existing code*, Addison Wesley (1999).

[19] 永田　靖，吉田道弘：統計的多重比較法の基礎，サイエンティスト社 (1997).

[20] McCabe, T.J.: A Complexity Measure, *IEEE Trans. Softw. Eng.*, Vol.SE-2, No.4, pp.308–320 (1976).

[21] 工藤良介，伊達浩典，石尾　隆ほか：コードクローンに含まれるメソッド呼び出しの変更度合の調査，情報処理学会研究報告，Vol.2013-SE-179, No.15, pp.1–8 (2013).

[22] Kapser, C.J. and Godfrey, M.W.: "Cloning considered harmful" considered harmful: Patterns of cloning in software, *Empirical Software Engineering*, Vol.13, No.6, pp.645–692 (2008).

[23] Cheung, W.T., Ryu, S. and Kim, S.: Development nature matters: An empirical study of code clones in JavaScript applications, *Empirical Software Engineering*, Vol.21, No.2, pp.517–564 (2016).

[24] Svajlenko, J. and Roy, C.K.: Evaluating Modern Clone Detection Tools, *Proc. ICSME'14*, pp.321–330 (2014).

上村 恭平 （学生会員）

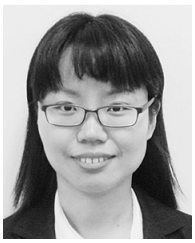
平成 26 年大阪府立大学工業高等専門学校専攻科総合工学システム専攻修了．平成 28 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．現在，同大学院博士後期課程に在学．修士（工学）．コードクローンを中心に，ソフトウェア工学分野の研究に従事．

森 彰

平成 7 年京都大学大学院工学研究科情報工学専攻博士課程修了（博士（工学））．英国オックスフォード大学訪問研究員，米国カリフォルニア大学サンディエゴ校ポスドク，北陸先端科学技術大学院大学助手を経て，平成 13 年産業技術総合研究所入所．現在同所研究グループ長．ソフトウェア工学およびコンピュータセキュリティに関わるプログラム解析技術の研究に従事．

藤原 賢二 （正会員）

平成 22 年大阪府立工業高等専門学校総合工学システム専攻修了．平成 27 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．同年同大学院博士研究員．平成 28 年豊田工業高等専門学校情報工学科助教．博士（工学）．リファクタリングの適用履歴分析，プログラミング教育支援に関する研究に従事．



崔 恩瀞 （正会員）

平成 24 年大阪大学大学院情報科学研究科博士前期課程修了．平成 26 年同大学院情報科学研究科博士後期課程修了．平成 27 年大阪大学大学院国際公共政策研究科助教を経て，平成 28 年より奈良先端科学技術大学院大学情報科学研究科助教．博士（情報科学）．コードクローン管理やリファクタリング支援手法に関する研究に従事．

飯田 元 （正会員）

昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 7 年奈良先端科学技術大学院大学情報科学センター助教授．平成 17 年同大学情報科学研究科教授．博士（工学）．