ハードウェア記述言語におけるコードクローンの調査

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

概要：本論文はハードウェア記述言語 (HDL) におけるコードクローンの特徴を明らかにすることを目的とする．類似あるいは一致するコードの断片であるコードクローンは，ソフトウェアにおいて品質や保守性を低下させる要因の一つであると言われている．ソフトウェアと同様に，HDL コードにおいてもコードクローンが存在し，品質や保守性に悪影響を与えている可能性があるが，HDL におけるコードクローンに着目した研究は報告されていない．本論文では代表的な HDL である Verilog HDL コードを擬似的な C++コードに変換することで，既存のツールを用いてコードクローンを検出する簡易な手法を提案する．加えて，オープンソースで開発されている回路を対象としたケーススタディによる調査結果を報告する．調査の結果，3 つのプロジェクトに含まれる 1 万 9 千行のコード中から 364 組のクローンセットが検出された．また，検出されたコードクローンを観察した結果，Verilog HDL におけるコードクローンは回路の性能と言語の仕様上の制約により作られており，また，除去することが困難であることが確認された．

キーワード：コードクローン，実証的ソフトウェア工学，回路開発 , ハードウェア記述言語

Investigation for Code Clone in Hardware Description Language

Abstract: The purpose of this study is to investigate the existence of code clones (i.e. code fragments that have similar or identical code fragments in the source code) and affectabillity to its quality for Hardware Description Language (HDL). There is a high possibility that code clones in software systems present a negative impact to the source code. Since HDL is similar to programming languages, code clones in HDL code may cause bugs. This paper proposes a technique that detects code clones from Verilog HDL, which is one of the most commonly used HDL. Basic idea of the technique is converting Verilog HDL code to Pseudo-C++ code so that CCFinderX can take and parse them. In addition, we conducted a case study to investigate the characteristics of the code clones in the three open source hardware projects. As a result, 364 clone sets are detected from 19 kLoC Verilog HDL code. The case study shows that merging these code clones is difficult, because of circuit performance and specification of Verilog HDL.

Keywords: Code Clone, Empirical Software Engineering, Hardware Development, Hardware Description Language

1. 背景

ハードウェア記述言語 (HDL) は LSI や，回路の再構築が可能なハードウェアである Field Programmable Gate Array (FPGA) の設計等に利用される，論理回路を記述するための言語である．論理回路の記述には，論理式に加え，条件分岐や繰り返し構造を用いることができ，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．代表的な HDL として Verilog HDL や VHDL が挙げられる．近年，FPGA の利用範囲の拡大などを背景に，回路開発の効率化が求められている．そうした中，回路開発の効率化を目的に，ソフトウェア開発で用いられる開発支援や効率化技術を応用する研究が行われている [1–3].

ソフトウェアにおいて，ソースコードの品質や保守性を下げる要因の一つとして，ソースコード中の類似あるいは

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology , Ikoma, Nara 630–0192, Japan
[2] 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)] fujiwara@toyota-ct.ac.jp
[c)] choi@is.naist.jp
[d)] iida@itc.naist.jp

©2016 Information Processing Society of Japan

同一のコードの断片であるコードクローンが研究されている．回路開発に用いられる HDL コード中にも同様にコードクローンが存在すると考えられる．例えば，回路中には，並列化を目的とした機能や，メモリへの入出力等の類似した構造を持つ箇所が複数存在する．これらの機能を HDL で記述する場合，同一，あるいは類似するコードを記述する必要がある．しかし，HDL におけるコードクローンに着目した研究は我々の知る限り存在しない．

本研究は，代表的な HDL である Veirlog HDL を対象に，コードクローンの特徴を明らかにすることを目的とする．まず，調査のために，Verilog HDL を対象としたコードクローン検出手法を提案する．従来のコードクローン検出ツールは，プログラミング言語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を目的に設計されており，対象とする言語の仕様に深く依存している．そのため，それらのツールを改変することなく，言語仕様の異なる Verilog HDL で記述されたコードを対象にコードクローンを検出することは容易ではない．提案手法では，新たにコードクローン検出ツールを開発することなく，Verilog HDL 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 [4] を利用してコードクローンを検出する．加えて，Verilog HDL で開発されている 3 つの回路プロジェクトを対象としたケーススタディを通し，Verilog HDL コード中のコードクローンの特徴を調査した．調査にあたり，3 つのリサーチクエスチョンを設けた．

RQ 1　Verilog HDL コード中にコードクローンは存在するのか．

RQ 2　なぜコードクローンが作られているのか．

RQ 3　一般的なプログラミング言語と比べて異なる特徴は存在するのか．

以降，2 章で本研究に関連する先行研究と関連技術について紹介する．3 章では本論文において提案する，Verilog HDL を対象としたコードクローン検出手法について説明する．4 章では提案手法を用いたケーススタディの概要と，その結果について述べる．5 章ではケーススタディの結果を踏まえた考察を述べ，6 章では本研究の妥当性について検討する．最後に 7 章で本論文についてまとめる．

2. 関連研究及び技術

本章では本研究に関連する既存研究と技術について説明する．2.1 節ではコードクローンについて，2.2 節では HDL について述べる．

2.1 コードクローン

本節では，コードクローンに関する定義を述べ，検出手法及び品質との関連性について調べた既存研究について紹介する．

2.1.1 コードクローンの定義

他のコードと一致，あるいは類似するコードの断片をクローン片，それらの集合をクローンセットと呼ぶ．また，互いに対応する 2 つのクローン片の組をクローンペアと呼ぶ．コードクローンの一般的な定義にはコード片の一致，または類似の程度についての基準は含まれていない．2.1.2 節で後述するコードクローン検出ツールは，独自に基準を用いてコードクローンの検出，フィルタリングを行う．

コードクローンは，主に類似機能の追加などを目的として，開発者がコードをコピーアンドペーストすることで作成されるといわれている．また，定型処理など，記述したコードが意図せずにコードクローンとなる場合もある．

Bellon らは一致の度合いによって，コードクローンを下記の三つのタイプに分類している [5]．

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による差異を除き，完全に一致するコードクローンを指す．

タイプ 2　タイプ 1 における差異に加え，変数名や関数名などのユーザー定義名，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を指す．

タイプ 3　タイプ 2 における差異に加え，文の挿入や削除，変更が行われたコードクローンを指す．

2.1.2 コードクローンの検出手法

一般的に，ソースコード中のコードクローンを手作業で見つけ出すのは困難である．そのため，同一プロジェクト，またはプロジェクトを跨ったソースコード中からコードクローンを自動的に見つけ出すためのコードクローン検出手法が提案されている．コードクローン検出手法には文字やトークンの並びが一致・類似するものを検出する手法や，抽象構文木が一致・類似するものを検出する手法，データの流れが一致・類似するものを検出する手法など多数のアルゴリズムが提案されている [6]．

2.2 ハードウェア記述言語（HDL）

本節では代表的な HDL である Verilog HDL の基本的な文法について述べる．加えて，HDL を用いた回路開発を対象とした開発の効率化のための研究について紹介する．

2.2.1 Velilog の基本文法

Verilog HDL において回路を構成する基本単位はモジュールである．開発者はモジュールの中にさまざまな機能や，他のモジュールとの接続関係を定義することで目的の回路を作る．

図 1 はモジュールの構成を簡易に示した図である．モジュールの定義は予約語 module とモジュール名の記述から始まり，end module で閉じられる．モジュールは大きく分けてポート宣言，その他の宣言，回路記述の 3 つの要素から構成される．ポート宣言部では，モジュールの外部

©2016 Information Processing Society of Japan

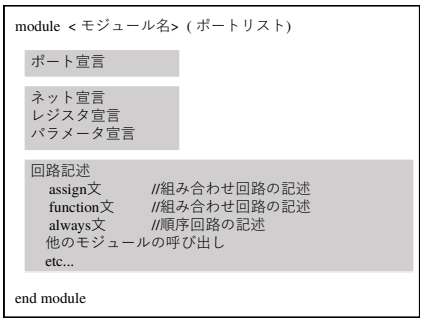


図 1: モジュール構成の概略

リスト 1: 文の入れ替えを示す例 1

```
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: 文の入れ替えを示す例 2

```
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
```

と接続される入出力端子であるポートが定義されている．その他の宣言部には回路を構成するポート以外の要素である，回路内の配線（ネット）やレジスタ，定数として用いられるパラメータが記述される．回路記述部にはモジュールを構成する組み合わせ回路と順序回路の振る舞いが記述されている．組み合わせ回路の記述には assign 文と function 文が用いられ，順序回路には always 文を用いて記述される．また，既に定義した別のモジュールを呼び出して利用することができる．

前述のように，Verilog HDL は C 言語などの一般的なプログラミング言語と類似した側面を持つ．しかし，処理の実行順序が必ずしも記述順序と一致しないという違いがある．C 言語や Java 言語などでは，ある関数内の処理は記述されている順に上から実行される．一方，Verilog HDL においては，文の位置関係は処理に影響がない．例えば，リスト 1,2 に示すコードは，それぞれ 7 行目と 8 行目の代入文が入れ替わっている．もし，処理が上から順に実行される場合，出力 d の値はリスト 1 では入力 a の値が設定され，2 ではレジスタ b の値が設定される．しかし実際には，この 2 つのコードはどちらも出力 d の値にレジスタ b の値が設定される，同一の回路を定義している．

2.2.2 回路開発の効率化に関する研究

回路開発の効率化を目的に，HDL コードに着目した研究が多数報告されている．Sudakrishnan らは Verilog HDL を用いた回路開発におけるバグ修正のパターンを調査している [1]．彼らは，4 つのオープンソースの回路開発プロジェクトのリポジトリを対象に，バグ修正時の編集内容を手作業で分類している．この研究では，コミットメッセージ中に *bug* や *fix* などのキーワードが含まれるコミットをバグ修正に関わる編集とみなしている．コミットメッセージからバグ修正のために行われた編集内容を調べる手法は，ソフトウェアを対象に Fischer らや，Čubranić と Murphy が提案しているもの [7,8] を参考にしている．

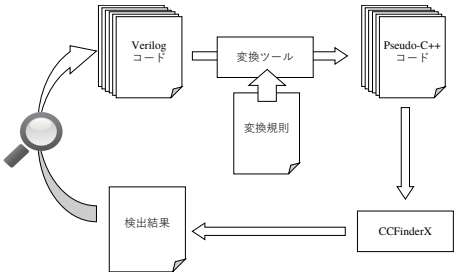


図 2: 提案手法の概要

Nacif らは回路開発におけるメトリクスの計測と可視化を行うためのツールとして EyesOn を提案している [2]．彼らはケーススタディを通し，ソフトウェア開発と同様に，回路開発においても，メトリクスを用いた進捗や品質の管理が有用であることを示している．

Parizy らは検証作業の優先順位付けにバグ予測の結果を用いることで，シミュレーション上では開発期間を 12%短縮することができたと報告している [3]．この研究もソフトウェア工学の分野で Hassan が提案する手法 [9] を参考にしている．

3. Verilog HDL コード中のコードクローン検出手法の提案

本章では本研究で提案する Verilog HDL を対象としたコードクローン検出手法について述べる．3.1 章では手法の概要について，3.2 章では手法で用いる Verilog HDL コードの変換規則について説明する．

3.1 概要

提案する Verilog HDL コード中のコードクローンの検出手法の概要を**図 2** に示す．提案手法では，既存のコードクローン検出ツールである CCFinderX [4,10] が C++コードとしてパースできるように Verilog HDL コードを変換する．そして変換されたコードを C++コードとして CCFinderX に入力することで，コードクローンを検出する．

CCFinderX を利用する理由として，CCFinderX が高速かつ大規模なソースコードにも適用可能であり，多数の研究及び開発現場で利用実績がある [11,12] ことが挙げられる．また，GUI フロントエンドである GemX によりコードクローンの閲覧及び分析が容易であることも理由の一つである．CCFinderX は plane text をスペースやアンダーバー，ハイフンで区切り，区切られた文字列をトークンとすることでコードクローンを検出する機能を持つ．しかし，トークンの区切りが実際のコードの区切りとは大きく異なるため，類似するコードであっても検出されない可能

©2016 Information Processing Society of Japan

リスト 3: Verilog HDL コードの例

```
1 module sample (ck, j, k, q, rb, sb );
2   input ck, j, k, rb, sb;
3   output q;
4   reg q;
5
6   always @( posedge ck or negedge rb or negedge sb ) begin
7     if ( rb == 1'b0 )
8       q <= 1'b0;
9     else if ( sb == 1'b0 )
10      q <= 1'b1;
11    else begin
12      case ( { j, k } )
13        2'b00 : q <= q;
14        2'b01 : q <= 1'b0;
15        2'b10 : q <= 1'b1;
16        2'b11 : q <= ~q;
17      endcase
18    end
19  end
20 endmodule
```

リスト 4: Pseudo-C++の例

```
1 class sample { sample (ck, j, k, q, rb, sb ) {
2   input ck, j, k, rb, sb;
3   output q;
4   reg q;
5
6   always @( posedge ck or negedge rb or negedge sb ) {
7     if ( rb == 1'b0 )
8       q <= 1'b0;
9     else if ( sb == 1'b0 )
10      q <= 1'b1;
11    else {
12      case ( { j, k } ) {
13        2'b00 : q <= q;
14        2'b01 : q <= 1'b0;
15        2'b10 : q <= 1'b1;
16        2'b11 : q <= ~q;
17      }
18    }
19  }
20 }};
```

性がある．例えば，変数名がそれぞれ異なり，含まれるハイフンの数が異なると，類似コードとみなされない．また，トークンの区切り方をスペースのみにしたとしても，トークンの正規化ができないため，タイプ 2 のような変数名が変更されたようなコードクローンを検出することはできない．従って，ソースコードとして CCFinderX が読み取れる形に変換し，検出を行う必要がある．

Verilog HDL コードを C++コードに変換するツールは既にいくつか提案されている [13, 14]．しかし，これらは今回の手法におけるコードの変換に利用するのに適していない．これらのツールは Verilog HDL コードが定義する回路の動作を，C++で記述されたプログラムでシミュレーションすることを目的に開発されている．そのため，変換に際して元のコードの構造を保つ必要がなく，異なる構造の C++コードが生成される．構造の異なるコード上でコードクローンが発見されたとしても，元の Verilog HDL コード上でコードクローンが存在するとは限らず，対応付けることはできない．そのため本論文では，Verilog HDL コードの構造を変化させず，行や文の対応関係がとれる形で変換する規則を提案する．本論文ではこの変換規則に則り変換されたコードを Pseudo-C++コードと呼ぶ．

3.2 変換規則

CCFinderX は入力されたコードを構文解析し，関数や if 文等のブロックを識別する．その一方で，CCFinderX は抽象構文木を用いるコードクローン検出ツールが行うような厳密な構文解析を行わないように実装されている．そのため厳密には C++の文法に則していないコードであっても，ブロックの対応さえ取ることができれば，CCFinderX を用いてコードクローンを検出することができる．以上のことを踏まえて，Verilog HDL を変換する規則を以下のように定めた．

変換規則 1 モジュールの定義をクラスのコンストラクタに置き換える．

変換規則 2 ブロックの開始と終了を表す予約語である “begin” や “end” などを開き中括弧と閉じ中括弧に置き換える．

Pseudo-C++コードでは，モジュールの入出力定義部をコンストラクタの引数となるよう書き換え，モジュール全体をクラスのコンストラクタとして扱う．また，Verilog HDL ではブロックの開始と終了を begin，end 句により示しているが，C++におけるブロックの開始と終了を示す中括弧に置き換えることで，CCFinderX に適用させる．

Pseudo-C++コードの例として，リスト 3 に対して変換規則 1，2 を適用した結果をリスト 4 に示す．このコードは C++としてコンパイルすることはできないが，CCFinderX により構文解析することはできる．リスト 4 の 1 行目は入出力ポートのリストがコンストラクタの引数に変換されている．引数のリストとして扱うには，変数の型に相当する記述が存在しないが，CCFinderX の構文解析器は厳密性を持たないため問題とならない．6 行目から始まる always 文と 1 2 行目から始まる case 文は関数定義として扱われ，構文解析されている．提案する変換規則は，元のコードの構造や行数に変更を加えない．そのため，Pseudo-C++を入力された CCFinderX が出力するコードクローンの位置情報や，クローンの規模などのメトリクスは，そのまま Verilog HDL コードのコードクローンの位置情報やメトリクスとして扱うことができる．

4. ケーススタディ

本章では，代表的な HDL である Verilog HDL における，コードクローンの存在性およびその特徴を調査するために行ったケーススタディの概要及び結果について述べる．

4.1 ケーススタディ概要

本節では，ケーススタディの概要について述べる．4.1.1 節でケーススタディの手順について，4.1.2 節で対象とするプロジェクトについて説明する．4.1.3 節では，ケーススタディにおける CCFinderX の設定について述べる．

4.1.1 手順

初めに，調査対象のプロジェクトに含まれる Verilog HDL コードを 3.2 章に示した変換規則に則り，Pseudo-C++コードに変換する．次に，変換した Pseudo-C++コードをプロジェクト毎に分け，CCFinderX に入力し検出結果を得る．

©2016 Information Processing Society of Japan

表 1: 対象プロジェクトの概要

プロジェクト名	概要	ファイル数	コード行数
mor1kx	OpenRISC プロジェクトに含まれる RISC プロセッサ	45	14747
1000base-X	Ethernet の機能を提供する回路	14	2407
sd_card_controller	SD カードの読み込み書き込みを行う回路	16	1915

表 2: コードクローンの検出結果

	mor1kx	1000base-X	sd_card_controller
クローンセットの数	293	49	22
クローン片の数	763	90	28

CCFinderX は複数のプロジェクトに跨るコードクローンの検出も可能であるが，本論文では単一プロジェクト内のコードクローンを調査対象とする．その後，CCFinderX の出力する検出結果をもとに定量的な値について分析を行う．加えて，検出されたコードクローンを観察し，定性的な分析を行う．

4.1.2 ケーススタディの対象

ケーススタディの対象として 3 つの回路開発プロジェクトを OpenCores から選択した [15]．OpenCores はオープンソースの回路開発を支援するコミュニティで，Subversion による開発リポジトリの他，フォーラムなどを提供している．本ケーススタディの対象を選出するにあたり，3 つのプロジェクトはそれぞれ，ファイル数，コード行数といったプロジェクトの規模が異なるようにした．各プロジェクトの概要を**表 1** に示す．

4.1.3 CCFinderX の設定

CCFinderX には検出するコードクローンをフィルタリングするための設定項目が複数存在する．本論文におけるケーススタディでは，検出するコードクローンの最小トークン長である，Minimum Clone Length の値のみを規定値の 50 から 20 に変更し，その他の項目は規定値を用いた．Minimum Clone Length の値を小さくした理由は，より多くのコードクローン検出例を分析するためである．

4.2 ケーススタディ結果

本節ではケーススタディの結果についてコードクローンの量に関する分析と，検出された事例に基づく定性的な分析を行う．

4.2.1 コードクローンの量

クローンセットの数とクローンセットに含まれるクローン片の数について計測した．提案手法により検出されたコードクローンの数を**表 2** に示す．3 つのプロジェクトで合計 364 組のクローンセットが検出され，そのクローンセットには 881 個のクローン片が含まれていた．

図 3 及び**図 4** はそれぞれ，各ファイルに含まれるクローン片の個数と各ファイルにおける，クローン片を含む行の割合をプロジェクト毎に示した箱ひげ図である．

図 3 より，プロジェクトの規模が大きい mor1kx は，規模の小さい 1000base-X と sd_card_controller と比較し，1 つのファイルに含まれるクローン片の量が多いことがわかる．一方，図 4 からは，プロジェクト規模に関わらず，全体の行の 30%程度をクローン片が占めているファイルも多く存在することも確認できる．

4.2.2 コードクローンの事例分析

検出されたコードクローンのうち，特徴的な 3 件のコードクローンを報告する．観察に際して，コードクローン検出の目的の一つであるコードクローンの集約と同時編集の必要性に観点を置いた．

1 件目は assign 文に関するコードクローンである（リスト 5）．このクローン片は 4 つのファイルに含まれている．この assign 文は代入先である左辺は異なるが，右辺はすべてのクローン片で一致している．Verilog HDL ではモジュール間で計算式の再利用を行う機能が提供されていないため，このようなコードクローンを集約することはできない．また，このような論理式は変更が生じた際に同時に編集する必要がある可能性が高い．

2 件目は 2 行目の if 文と，1 1 行目の else 文に続く if 文がそれぞれコードクローンになっている（リスト 6）．信号線の rst か flus_i が真値であれば初期化を行う回路が定義されている．rst が真値の場合と flus_i が真値の場合，どちらも初期化内容は同じである．このコードクローンは，ソフトウェア開発者の視点からすると，リスト 7 のように，if 文の条件を論理式で記述することで集約することができるように見える．しかし実際には，そのような集約は回路の性能にとって問題となりえる．例えば rst 信号線に対する論理演算は不安定な動作に繋がるなどの問題から推奨されていない．従って，このようなコードクローンの集約は困難であると言える．このコードクローンは条件や初期化内容に変更がある際，同時に編集する必要がある．

3 件目は generate 文に関するほぼ一致するコードクローンである．リスト 8 と類似する，一部の変数名のみが変更されたコード片が複数存在する．これらのクローン片を含むファイルは，他にも多くの共通部分を持ちファイル全体がタイプ 3 コードクローンになっている．このファイルは，仕様が若干異なる同様の目的を持った回路を定義するために作られている．言語の仕様上，このような場合は個別に module として定義する必要があるため，集約することができない．また，このようなコードクローンの例では，同

©2016 Information Processing Society of Japan

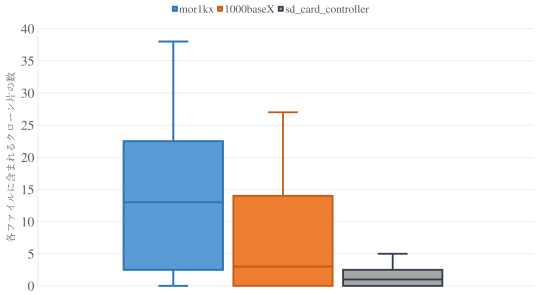


図 3: ファイルに含まれるクローン片の数の分布

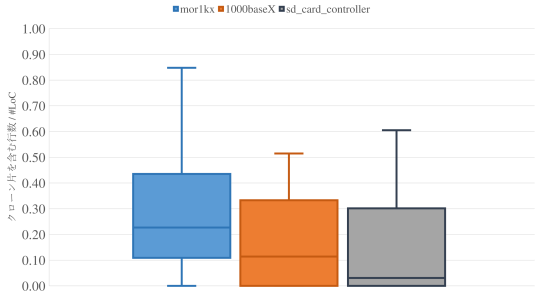


図 4: ファイルにおけるコードクローンを含む行の割合の分布

リスト 5: コードクローンの例 A

```
assign next_dbus_adr = (OPTION_DCACHE_BLOCK_WIDTH
      == 5) ?
  {dbus_adr[31:5], dbus_adr[4:0] + 5'd4} : // 32 byte
  {dbus_adr[31:4], dbus_adr[3:0] + 4'd4}; // 16 byte
```

リスト 6: コードクローンの例 B

```
always @(posedge clk or OR_ASYNC_RST) begin
  if (rst) begin
    // arithmetic results
    fpu_result_o <= 32'd0;
    fpu_arith_valid_o <= 1'b0;
    // comparison specials
    fpu_cmp_flag_o <= 1'b0;
    fpu_cmp_valid_o <= 1'b0;
    // exeptions
    fpcsr_o <= {`OR1K_FPCSR_WIDTH{1'b0}};
  end
  else if(flush_i) begin
    // arithmetic results
    fpu_result_o <= 32'd0;
    fpu_arith_valid_o <= 1'b0;
    // comparison specials
    fpu_cmp_flag_o <= 1'b0;
    fpu_cmp_valid_o <= 1'b0;
    // exeptions
    fpcsr_o <= {`OR1K_FPCSR_WIDTH{1'b0}};
  end

  ////////////////////
  //Partially omitted//
  ////////////////////

end // posedge clock
```

リスト 7: コードクローンの例 B の集約例

```
always @(posedge clk or OR_ASYNC_RST) begin
  if (rst | flush_i) begin
    // arithmetic results
    fpu_result_o <= 32'd0;
    fpu_arith_valid_o <= 1'b0;
    // comparison specials
    fpu_cmp_flag_o <= 1'b0;
    fpu_cmp_valid_o <= 1'b0;
    // exeptions
    fpcsr_o <= {`OR1K_FPCSR_WIDTH{1'b0}};
  end

  ////////////////////
  //Partially omitted//
  ////////////////////

end // posedge clock
```

時修正が必要であるとは限らない.

5. 考察

本章では，4 章でのケーススタディの結果に基づき，以下のリサーチクエスチョンに答える.

RQ 1 Verilog HDL コード中にコードクローンは存在するのか.

RQ 2 なぜコードクローンが作られているのか.

RQ 3 一般的なプログラミング言語と比べて異なる特徴は存在するのか.

リスト 8: コードクローンの例 C

```
generate
  for (i = 0; i < OPTION_DMMU_WAYS; i=i+1) begin : dtlb
    // DTLB match registers
    mor1kx_true_dpram_sclk
      #(
        .ADDR_WIDTH(OPTION_DMMU_SET_WIDTH),
        .DATA_WIDTH(OPTION_OPERAND_WIDTH)
        )
    dtlb_match_regs
      (
        // Outputs
        .dout_a (dtlb_match_dout[i]),
        .dout_b (dtlb_match_huge_dout[i]),
        // Inputs
        .clk (clk),
        .addr_a (dtlb_match_addr),
        .we_a (dtlb_match_we[i]),
        .din_a (dtlb_match_din),
        .addr_b (dtlb_match_huge_addr),
        .we_b (dtlb_match_huge_we),
        .din_b (dtlb_match_reload_din)
        );


    // DTLB translate registers
    mor1kx_true_dpram_sclk
      #(
        .ADDR_WIDTH(OPTION_DMMU_SET_WIDTH),
        .DATA_WIDTH(OPTION_OPERAND_WIDTH)
        )
    dtlb_translate_regs
      (
        // Outputs
        .dout_a (dtlb_trans_dout[i]),
        .dout_b (dtlb_trans_huge_dout[i]),
        // Inputs
        .clk (clk),
        .addr_a (dtlb_trans_addr),
        .we_a (dtlb_trans_we[i]),
        .din_a (dtlb_trans_din),
        .addr_b (dtlb_trans_huge_addr),
        .we_b (dtlb_trans_huge_we),
        .din_b (dtlb_trans_reload_din)
        );
  end
endgenerate
```

5.1 RQ1：Verilog HDL コード中にコードクローンは存在するのか

3 件のプロジェクトの 1 万 9 千行のコード中から 364 組のコードクローンが検出され，いずれのプロジェクトにおいてもコードクローンが存在することが確認された.

5.2 RQ 2：なぜコードクローンが作られているのか

4.2.2 節では Verilog HDL コード中に存在するコードクローンについて，3 つのクローンセットの実例を基に分析を行った．調査した 3 つの例はそれぞれコードクローンが作成される状況が異なる.

状況 1（リスト 5 に対応）

複数のモジュール間で同様の論理演算が行われている.

状況 2（リスト 6 に対応）

同一モジュール内において，異なる条件で動作する同様の機能をもつ回路が作られている.

状況 3（リスト 8 に対応）

単一プロジェクト内において，仕様が異なる複数の同

©2016 Information Processing Society of Japan

一の目的を持った回路が定義されている．

Verilog HDL の言語の仕様上，論理式を複数のモジュール間で使えるサブルーチン化するような機能が存在しない．しかし，回路の性質上，類似する論理演算が複数の箇所で何度も必要となるため，このようなコードクローンが作られる．このような論理式のクローンは assign 文による代入だけでなく，if 文の条件式など他の場面でも起こりうる．

状況 2 は，リスト 6 の例に見られるように，初期化を目的とした場面で起きやすいと考えられる．初期化条件ごとに同一の回路を複数用意した方が回路の性能上有利なため，このようなコードクローンが作られる．

状況 3 は，バス幅が異なる同一目的の回路など，利用する FPGA の種類に応じて回路を変更したい場合や，必要十分な機能のみを FPGA に実装できるようにしたい場合に起こる．このような仕様が若干異なる回路は，個別に module として定義する必要があるため，ファイル全体がコードクローンとなる．

以上のことから，回路の性能と安定性上，同一の回路が複数存在するほうが有利な場面が存在するが，Verilog HDL による回路開発おいては，単一のコードでそのような回路の定義を記述する仕組みが存在しないため，類似するコードを複数記述する必要があるためコードクローンが作られていると言える．

5.3 RQ 3：一般的なプログラミング言語と比べて異なる特徴は存在するのか

RQ 2 で述べたように，Verilog HDL コード中のコードクローンは回路の性能や言語の仕様上の都合で作られており，集約が困難である．また，論理式が複数のファイル間で多数再利用されている点も特徴として挙げられる．このような，式がコードクローンになっているものは，変更が生じた際に一貫して修正する必要がある可能性が高い．これらは一般的なプログラミング言語と異なる，Verilog HDL の特徴的といえる．そのため，回路開発に対するコードクローン支援技術としては，コードクローンの集約技術よりも，一貫した修正の支援が有効であると考えられる．

6. 妥当性の検討

本章では本研究の妥当性についての考察を行う．6.1 節では検出精度の検証について述べる．6.2 節ではケーススタディの対象プロジェクトによる結果への影響について述べる．6.3 節ではケーススタディにおいて用いた CCFinderX の設定について述べる．

6.1 コードクローン検出の精度

本論文では，提案する検出手法の精度について検証を行っていない．提案手法の精度は Pseudo-C++コードへの変換規則と CCFinderX に依存すると考えられる．

コードクローン検出の精度は Precision と Recall という二つの指標で評価することができる．Bellon らは複数のコードクローン検出ツールの比較評価において，各ツールの検出結果を統合し，手作業で実際にコードクローンであるかを独自の基準で判定し，正解集合を作ることで，Recall を測定している [5].

提案手法による検出結果を 1 件ずつコードクローンであるか判定することで，Precision の評価は手作業で行うことは可能である．そのためには，コードクローンの判定基準を設定する必要がある．また Recall の評価については，Bellon らの手法を用いて正解セットを構築することは可能であるが，これには多くの時間を要する．加えて，複数の検出手法を用意する必要があるため，提案手法の精度の検証は今後の課題である．

6.2 対象プロジェクト

本論文では 3 件のオープンソースで開発されている回路を対象に調査を行った．従って，分析内容が選出したプロジェクトの特性に依存している可能性は否定することができない．プロジェクト中のコードクローンの割合などはプロジェクトによる差異が大きいと考えられるため，より多くのプロジェクトを対象とした調査が必要である．

6.3 CCFinderX の設定

今回のケーススタディにおける CCFinderX の設定は，Minimum Clone Length の値を 20 とした他は規定値を利用した．コードクローンの検出結果は，この設定値に大きく依存する．Minimum Clone Length の値を大きくすることで，Precision の値は大きくなり，Recall は小さくなると考えられる．また，今回規定値を利用した Minimum TKS の設定は，特にトークン長の短いコードクローンに影響する Minimum TKS の値を大きくすると，検出されるコードクローンは少なくなり，小さくすると多くなる．今後，検出精度について調査する際にはこれらの設定項目の値について細かな検討が必要である．

7. まとめと今後の課題

本研究における貢献は以下の通りである．

貢献 1 Verilog HDL コードを対象としたコードクローン検出手法を提案した．

貢献 2 Verilog HDL コードにおいてもコードクローンが存在することを明らかにし，その特徴について分析した．

本論文では，Veilog を対象としたコードクローン検出手法を提案した．提案手法は Verilog HDL コードを Pseudo-C++コードに変換することで，既存のコードクローン検

©2016 Information Processing Society of Japan

出ツールである CCFinderX を利用する．この手法は，変換規則を構築することができれば，VHDL() のような他の HDL や，既存のコードクローン検出ツールが対応していないプログラミング言語にも適用可能である．

3 つのオープンソースの回路を対象としたケーススタディを通し，Verilog HDL におけるコードクローンの調査を行った．調査の結果，1 万 9 千行の Verilog HDL コードから 364 組のクローンセットが検出された．

検出されたクローンセットから 3 件を選び，分析した．コードクローンは，同様の論理式を複数のモジュール間で再利用する場面や，単一モジュール内で初期化を目的とした機能を持つ回路を複数定義する場面で作成されていた．加えて，仕様が若干異なる類似した複数の回路を定義するために作成されているコードクローンも確認された．対象プロジェクトを増やし，より詳細に分析することで今回紹介したもの以外の分類が見つかる可能性がある．またそれらの存在する割合に関する調査は今後の課題である．

Verilog HDL におけるコードクローンは，集約することが難しいことが明らかになった．集約が困難である理由には，言語の仕様上の理由と，回路の性能上の理由がある．コードクローンを含むソフトウェアでは，クローン片に対する一貫していない修正がバグの原因となるといわれている．ソフトウェアにおいては，コードクローンを集約することで，バグ混入のリスクを低減する研究が行われている [16, 17]．Verilog HDL のコードクローンは集約が難しいため，回路開発者はクローン片に含まれる箇所を編集する際には注意を払う必要がある．このような集約が難しいクローン片に対しては，ソフトウェアにおけるコードクローンに関する研究で提案されている，一貫した編集の支援 [18, 19] などの開発支援技術が有用であると考えられる．一貫した修正が必要であるコード片と，不要であるコード片の判断は，現在のところ開発者の判断に依存する．しかし，変数名やモジュールの親子関係などから判断できる可能性はあり，今後の研究課題といえる．

本論文では検出手法の妥当性について議論をしていない．妥当性の検証には正解集合の構築が必要であり，そのためには他の検出手法を用意する必要がある．他の検出手法を用いたツールの実装及び正解集合の構築，各手法の検出精度の評価は今後の課題とする．

また，Verilog HDL を用いた回路開発における，コードクローンとバグの関連性についての詳細な調査は今後の研究課題である．しかし，オープンソースの回路開発を対象に前述のソフトウェアにおける研究と同様の調査を行うにあたっては，編集履歴を記録する頻度が低いことや，記録されているバグ票の数が少ないといった課題がある．

参考文献

[1] Sudakrishnan, S., Madhavan, J. T., Whitehead Jr., E. J. and Renau, J.: Understanding Bug Fix Patterns in Verilog, *Proc. MSR* (2008).
[2] Nacif, J. A., Silva, T. S. F., Vieira, L. F. M., Vieira, A. B., Fernandes, A. O. and Coelho Jr., C.: Tracking Hardware Evolution, *Proc. ISQED*, pp. 1–6 (2011).
[3] Parizy, M., Takayama, K. and Kanazawa, Y.: Software Defect Prediction for LSI Designs, *Proc. ICSME*, pp. 565–568 (2014).
[4] Kamiya, T., Kusumoto, S. and Katsuro, I.: 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).
[5] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591 (2007).
[6] 神谷年洋，肥後芳樹，吉田則裕：コードクローン検出技術の展開，コンピュータソフトウェア，Vol. 28, No. 3, pp. 29–42 (2011).
[7] Fischer, M., Pinzger, M. and Gall, H.: Populating a Release History Database from Version Control and Bug Tracking Systems, *Proc. ICSM*, pp. 23–32 (2003).
[8] Čubranić, D. and Murphy, G. C.: Hipikat: Recommending Pertinent Software Development Artifacts, *Proc. ICSE*, pp. 408–418 (2003).
[9] Hassan, A. E.: Predicting Faults Using the Complexity of Code Changes, *Proc. ICSE*, pp. 78–88 (2009).
[10] the archive of CCFinder Official Site, `http://www.ccfinder.net/ccfinderxos.html`.
[11] German, D. M., Di Penta, M., Guéhéneuc, Y.-G. and Antoniol, G.: Code Siblings: Technical and Legal Implications of Copying Code Between Applications, *Proc. MSR*, pp. 81–90 (2009).
[12] Kononenko, O., Zhang, C. and Godfrey, M. W.: Compiling Clones: What Happens?, *Proc. ICSME*, pp. 481–485 (2014).
[13] Stoye, W., Greaves, D., Richards, N. and Green, J.: Using RTL-to-C++ Translation for Large SOC Concurrent Engineering: a Case Study, *IET Transactions on Electronics Systems and Software*, Vol. 1, No. 1, pp. 20–25 (2003).
[14] Verilog2C++ , `http://verilog2cpp.sourceforge.net/`.
[15] OpenCores, `http://opencores.org/`.
[16] Garg, R. and Tekchandani, R.: An Approach to Rank Code Clones for Efficient Clone Management, *Proc. ICAECC*, pp. 1–5 (2014).
[17] Krishnan, G. P. and Tsantalis, N.: Unification and Refactoring of Clones, *Proc. CSMR-WCRE*, pp. 104–113 (2014).
[18] Nguyen, H. A., Nguyen, T. T., Pham, N. H., Al-Kofahi, J. and Nguyen, T. N.: Clone Management for Evolving Software, *IEEE Transactions on Software Engineering*, Vol. 38, No. 5, pp. 1008–1026 (2012).
[19] Kawaguchi, S., Yamashina, T., Uwano, H., Fushida, K., Kamei, Y., Nagura, M. and Iida, H.: SHINOBI: A Tool for Automatic Code Clone Detection in the IDE, *Proc. WCRE*, pp. 313–314 (2009).

©2016 Information Processing Society of Japan