卒 業 研 究 報 告 書

題　目　　組み込みソフトウェアにおける
　　コードクローン出現に関する考察

指導教員　　水野　修　教授

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　15122058

氏　　名　　若林　奎人

平成31年2月13日提出

組み込みソフトウェアにおけるコードクローン出現に関する考察

平成 31 年 2 月 13 日　　15122058　若林　奎人

概　要

近年組み込みソフトウェアにおいて，個人でハードウェア部品を安価に入手できることや部品の多様化に伴い，ソフトウェア中にはコードクローンが発生することがある．

組み込み開発プラットフォームの 1 つである Arduino は，特にハードウェア部品が安価であり，また，ハードウェア，ソフトウェア共にオープンソースであるため，派生品や互換機も数多く存在し，環境下には多くのコードクローンが発生していると考えられる．コードクローンはソフトウェアの保守を困難にしている要因の 1 つであると指摘されている．

そこで，GitHub から watch 数の多い Arduino のプロジェクトを 170 個取得し，コードクローン検出ツールである CCFinder に入力することで，コードクローンの位置を把握し，他のプロジェクトにおけるコードクローンの数と比較することで Arduino プロジェクトにおけるコードクローンの発生率について考察する．さらに組み込みソフトウェア特有のコードクローンについて考察することで，組み込みソフトウェアにおけるコードクローンに関する知見を得る．

CCFinder を用いた結果，Arduino プロジェクトには他のプロジェクトよりもコードクローンが多く発生していることがわかった．また，発生率の高いコードクローンの種類や，組み込みソフトウェア特有と思われるコードクローンを発見した．

目　次

1. 緒言

ソフトウェアの果たす社会的な役割は大きく，ソフトウェアの品質を高く保つことはソフトウェア工学における重要な目標である．また，ソフトウェア品質を高く保つためには，ソフトウェアの保守性を高める必要がある．しかし，1950 年代にソフトウェアが生まれて以来，ソフトウェアの保守性を高めるのは困難な課題の 1 つである．

ソフトウェアの保守性を下げる要因の 1 つにコードクローンがある．コードクローンは主にソースコードの再利用によって生まれる，ソースコードの内容が同じ構造になっているものを指す．そのため，不具合を持つソースコードのクローンが他のソフトウェアに含まれる場合，その不具合全てを発見し，修正することは困難になる．特に，組み込みソフトウェアにおいては，個人でのソフトウェア開発者も多いことから，コードクローンが幅広く発生していると考えられる．中でも Arduino [1] プロジェクトはハードウェア，ソフトウェアともにオープンソースであり，一般に広く再利用されやすい構造であるため，組み込みソフトウェアの中でもコードクローンが発生しやすいと考えられる．

そこで，コードクローン検出ツールである CCFinder [2] を Arduino プロジェクトに適用することで，コードクローンの発生状況を調査する．

本研究では，Arduino プロジェクトと組み込みソフトウェア以外のソフトウェアにおけるコードクローンの発生率を比較し，Arduino プロジェクトの方がコードクローンの発生率が高いことを確認した．また， Arduino プロジェクトにおいて，発生率の高いコードクローンについて考察し，その中から組み込みソフトウェア特有と思われるものを発見した．

本報告書の構成を以下に示す．2 章では本研究を行う目的について述べる．3 章では本実験に必要な前提事項や対象となるプロジェクトについて述べる．4 章では実験の手順について説明する．5 章では実験結果を報告する．6 章では実験結果の考察を行う．7 章では本研究のまとめと今後の課題について述べる．

2. 研究の目的

本研究の目的は，CCFinder を Arduino プロジェクトに適用し，出力としてコードクローンのメトリクスを得た上で，Arduino 環境に存在するコードクローンについて考察することである．

この目的のため，以下の研究設問を設定し，実験結果について考察する．本研究における研究設問を以下に示す．

RQ1 Arduino 環境においてどの程度の割合でコードクローンが存在するか．

RQ2 ソースコードのどのような部分にコードクローンが存在するか．

RQ3 組み込みソフトウェア特有のコードクローンは存在するか．

3. 準備

3.1 Arduino

Arduino (注 1)とは，安価で入手できるオープンソースのプラットフォームである．Arduino の基板は，光センサーやスイッチ，音センサーから入力を受け付け，モーターや LED などへの出力を可能とする．Arduino 基板を動作させるには，C++言語を拡張した Arduino 言語と Arduino 用統合開発環境を用いる必要がある．

Arduino はイタリアのイベーラインタラクションデザイン大学で誕生した，エレクトロニクスやプログラミングの知識のない学生でも利用できるように開発されたツールである．幅広いコミュニティで利用されるようになると，シンプルな 8 ビット基板から，ウェアラブル機器，3D プリント，組み込み環境など，新しい形に変化していった．全ての Arduino 基板はオープンソースであり，個人のニーズに合わせて開発できる．ソフトウェアも同様にオープンソースであり，世界中のユーザーの知見によって拡大し続けている．

数年に渡り，Arduino は日用品から工業用まで様々なプロジェクトに応用されている．学生からプログラマー，アーティスト，専門家など，世界中の人々が Arduino を利用しており，知見を寄せ合っている．Arduino はそのシンプルさによって数多くのアプリケーションに利用されている．また，その統合開発環境である Arduino IDE は Mac，Windows，Linux 上で動作させることができる．

本研究では Arduino プロジェクトにおけるコードクローンの分析を行う．Arduino には派生プロジェクトが多く存在するため，コードクローンの影響が強いと予想される．

3.2 GitHub

GitHub (注 2) [3] とは，開発プラットフォームであり，オープンソースプロジェクトやビジネスユースまで，GitHub 上にソースコードをホスティングすることで他の開発者からレビューを得ることや，プロジェクトを管理しながらソフトウェアの開発

(注 1): https://www.arduino.cc
(注 2): https://github.com

を行うことができる．

バージョン管理は Git によって行われる．Git は分散型バージョン管理システムの一つであり，開発参加者 1 人 1 人がリポジトリのコピーを保有できる．

開発者は他者にレビューをリクエストでき，他者は開発者にプルリクエストをしてコードの変更を提案できる．GitHub にはフォーク機能があり，他者のリポジトリをコピーし変更を加えることができる．変更が加えられたら，コピー元の開発者はマージすることでその変更を自分のコードに導入できる．また，変更内容の差分を確認できるので，他者はレビューを迅速に行うことができる．

GitHub が誕生する以前は，オープンソースのプロジェクトに貢献するために，そのプロジェクトのソースコードを手作業でコピーし，変更をローカルに加え，パッチを元の開発者にメールで送る，という手順を踏む必要があった．しかし現在では，全て GitHub のプラットフォーム上で行うことができる．

また，Git がコマンドラインツールであるのに対し，GitHub はグラフィカルに扱うことができる．プロジェクトを管理するために，プロジェクト参加者のタスクの割り振りや，プロジェクトの進行度を把握する機能がある．また，プロジェクトについて議論できる機能もある．

GitHub には連携することで開発ワークフローを強化できるツールが存在し，コミュニケーションの円滑化，作業の自動化を促し，作業をスムーズにできる．また，GitHub GraphQL API を用いて独自にツールを開発し，さらにデータへのアクセス性を向上できる．開発者向けのツールも存在し，あらゆる規模での開発で効果が発揮される．

チーム管理機能も存在し，どのメンバーがどのリポジトリにアクセスできるかを管理できる．また，モデレーションツールによりイシューとプルリクエストをブロックし，メンバーをコードに集中させ続けることができる．

3.3 Google Big Query

Google Big Query (注 3) [4] とは，Google が提供するビッグデータ解析ツールである．データベースの操作には SQL 文を使用でき，GitHub のリポジトリのデータベース

(注 3): https://cloud.google.com/bigquery

が存在する．

3.4 コードクローン

コードクローンとは，複数のソースファイル中で類似したコード部分のことである [4]．コードクローンは，コピーアンドペーストなど，コードの再使用によって生まれる．また，改変が繰り返された生存時間の長いプロジェクトにもしばしばコードクローンが発生する．さらに，コードをコピーした後，僅かに変化を加えたものもコードクローンとなり得る．

コードクローンはソフトウェアの保守性を下げる．例えば，複製され，変化を加えられたコードクローンを複数持つソフトウェアが存在したとする．複数あるコードクローンの内，どれか 1 つに不具合が見つかったとき，エンジニアは全てのコードクローンの不具合を除去しなければならない．複数のエンジニアが開発に携わる，複雑で大規模なソフトウェアであった場合，不具合を除去するのはさらに困難になる．

もしコードクローンの存在を事前に知っており，メンテナンスをしていれば，不具合の除去は比較的簡単になるが，コードクローンの情報を保持し続けるのは困難でコストがかかる．

一般に，コードクローンは，文字数が最大になるように定義される．例えば，

a x y z b x y z c x y d

という文字列があった場合，コードクローンは，xyz，xy，yz の 3 種類があるが，このとき文字数が最大のものを選ぶので，コードクローンは xyz となる．

次に，コードクローンを含むソースコードの例を，ソースコード 3.1 に示す．クラス名が異なるだけで，同じ処理をしている．

ここで，MultiButtonUI と MultiColorChooserUI の 2 つのクラスは，クラス名が異なるだけで同じ処理をしているため，コードクローンであると言える．

ソースコード **3.1**　コードクローン例

```
1  public class MultiButtonUI extends ButtonUI {
2          public static ComponentUI createUI(JComponent a) {
3                  ComponentUI mui = new MultiButtonUI();
4                  return MultiLookAndFeel.createUIs(mui,((MultiButtonUI)
                     mui).uis,a);
5          }
6  }
7
8  public class MultiColorChooserUI extends ColorChooserUI {
9          public static ComponentUI createUI(JComponent a) {
10                 ComponentUI mui = new MultiColorChooserUI();
11                 return MultiLookAndFeel.createUIs(mui,((
                     MultiColorChooserUI) mui).uis,a);
12         }
13 }
```

3.5 CCFinder

3.5.1 概要

CCFinder とは，神谷らによって開発されたコードクローン検出ツールである．工業用の大規模なソフトウェアシステムのような，複数のエンジニアが数十年に渡りメンテナンスを続けているプログラムに含まれるコードクローンを効率的に検出するツールが存在しなかったことが CCFinder 開発の動機となっている．

CCFinder の特徴は下記のようになっている．

- メモリと計算能力があれば，100 万行を超える工業用ソフトウェアシステムのソースコードにも適用できる．
- 大規模なシステムで大量のコードクローンが検出された場合，役に立つ情報を持つコードクローンのみを選ぶ．例えば，変数の初期化などは役に立たないものとして選出されない．
- 完全に同じコードだけでなく，変数名が異なったものなど，僅かに異なったコードからもコードクローンを検出する．
- 数種類のプログラミング言語に対応している．

CCFinder は接尾辞木を用いて文字列マッチングを行っており，行マッチングよりも計算量が大きい．しかし，いくつかの最適化を行い，大規模なソフトウェアにも実践的に使えるように設計されている．また，数種類のメトリクス値が計算され，コードクローンに関する情報を得ることができる．

3.5.2 コードクローンの検出過程

ソースコードを入力し，コードクローンを出力するまでの過程を示す．CCFinder のコードクローン検出過程は 4 つのステップに分かれている．全体像を図 3.1 に示す．

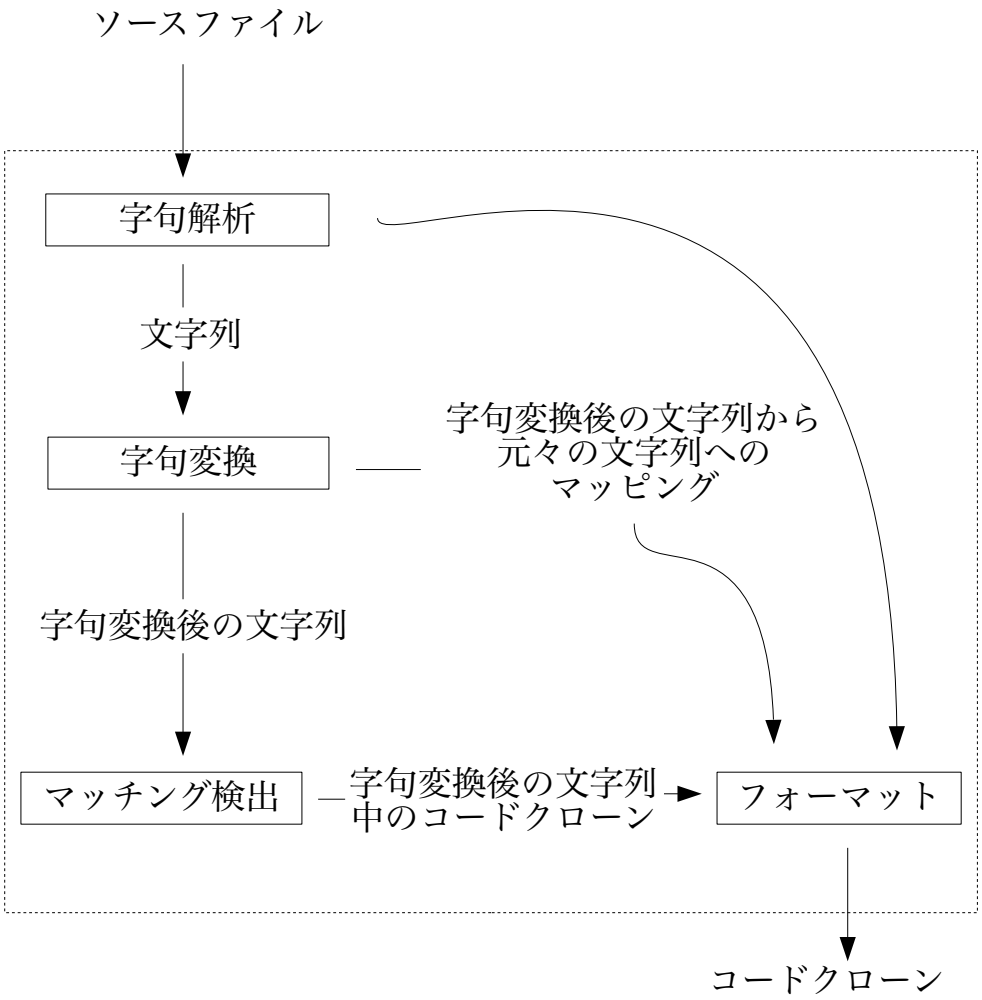

図 3.1　CCFinder のコードクローン検出過程

コードクローン検出過程の各ステップの説明を下記に示す.

1. **字句解析** 入力された各ソースファイル中のコードを全て一続きの文字列に変換する．この時，同時にコード中の空白，改行，コメントが削除される．これらの情報は最終ステップのフォーマッティングの際に利用するために保存される.
2. **字句変換** 各プログラミング言語毎に定まったルールに従い文字列の変換が行われる．また，型名や変数名などは全て一つの特殊な文字に変換することで，変数名が異なるだけで処理は同じプログラムをコードクローンとして検出できるようにしている．さらに，字句変換の前後のマッピング情報をフォーマッティングの際に利用するために保存する.
3. **マッチング検出** 接尾辞木を用いて文字列マッチングを行い，コードクローンを検出する.
4. **フォーマット** 各ステップで得た情報から，コードクローンの位置を元のソースファイルの行番号として出力する.

次に，C++言語で書かれたソースコードからコードクローンを検出するまでの過程をソースコード 3.2，ソースコード 3.3，ソースコード 3.4 に示す．ソースコード 3.2 は CCFinder に入力されるソースコードであり，ソースコード 3.3 はツールの定めたルールに従い，仮引数の宣言部やスコープ演算子を取り除いたものである．また，見やすさを優先して改行や空白は残しているが，本来は削除される．その後，変数名などを全て$p という文字列に置き換えたものがソースコード 3.4 である.

ソースコード 3.4 から，1 行目から 6 行目と，11 行目から 16 行目が同じ文字列になっており，コードクローンであることがわかる.

ソースコード 3.2　C++ファイルのコードクローン検出過程 1

```
1  Void print_lines(const set<string>& s){
2          int c = 0;
3          set<string>::const_iterator i
4                  = s.begin();
5          for(; i != s.end(); ++i){
6                  cout << c << “ , “
7                          << *i << endl;
8                  ++c;
9          }
10 }
11 Void print_table(const map<string, string>& m){
12         int c = 0;
13         set<string, string>::const_iterator i
14                 = m.begin();
15         for(; i != m.end(); ++i){
16                 cout << c << “ , “
17                         << i->first << “ “
18                         << i->second << endl;
19                 ++c;
20         }
21 }
```

ソースコード 3.3　C++ファイルのコードクローン検出過程 2

```
1 void print_lines(const set & s){
2 int c = 0;
3 Const_iterator i
4 = s.begin();
5 for(; i != s.end(); ++i){
6 cout << c << “ , “
7 << *I << endl;
8 ++c;
9 }
10 }
11 void print_table(const map & m){
12 int c = 0;
13 Const_iterator i
14 = m.begin();
15 for(; i != m.end(); ++i){
16 cout << c << “ , “
17 << i->first << “ “
18 << i->second << endl;
19 ++c;
20 }
21 }
```

ソースコード 3.4　C++ファイルのコードクローン検出過程 3

```
1 $p $p($p $p & $p){
2 $p $p = $p;
3 $p $p
4 = $p.$p();
5 for(; $p != $p.$p(); ++$p){
6 $p << $p << $p
7 << *$p << $p;
8 ++$p;
9 }
10 }
11 $p $p($p $p & $p){
12 $p $p = $p;
13 $p $p
14 = $p.$p();
15 for(; $p != $p.$p(); ++$p){
16 $p << $p << $p
17 << $p->$p << $p;
18 << $p->$p << $p;
19 ++$p;
20 }
21 }
```

3.5.3 メトリクス

CCFinder は検出したコードクローンがソフトウェアにどの程度影響を及ぼすのかを表すメトリクス値を算出する．各メトリクスの説明を表 5.3 に示す.

3.5.4 CCFinder に応用されている技術

CCFinder はコードクローンを検出したいソースファイルへのパスを入力として受け取り，コードクローンの位置情報を出力する．この時，コードクローンの最大文字数を m，ソースファイルの合計文字数を n とすると，CCFinder は接尾辞木を用いて計算量 $O(mn)$ で実行される．さらに，m が n に依存しない場合，計算量は $O(n)$ となる．この他に，大規模なソフトウェアに CCFinder を適用するために使われている最適化技術を下記に示す.

- **文字列のアラインメント** コードクローンを検出する際，関数やモジュールなどのブロック毎にマッチングすることで，接尾辞木のノード数を減らし効率を上げている．例えば，”{“や”}”は分岐文や繰り返し文の始まりと終わりを表しているので，これらの記号の間のコードを一塊としてマッチングを行う.
- **コードの繰り返しの除去** ソースコード 3.5 において，コードクローンであるペアの組み合わせは，${}_6C_2$ より，15 パターンある．このように，短いコードの繰り返しを全てコードクローンとして検出していると，数が多くなりすぎてしまうので，短文の繰り返されるソースコードは省かれる.
- **文字の連結** 空白や改行，コメントを取り除き，全ての文字列を一続きに繋げることで文字列の長さを削っている.
- **大規模ソースファイルの分割** ソースファイルの数が多すぎると接尾辞木の容量を超えてしまうので，複数の接尾辞木に分割し，分割統治法を用いる.

表 3.1　CCFinder が算出するメトリクス

メトリクス名	略称	内容
Number of File	FILE	入力するファイル数を表す.
Length	LEN	コードクローンの文字数を表す.
Line of Code	LOC	入力されたソースファイル群の合計行数を表す.
Source Line of Code	SLOC	LOC から空行とコメント行の数を引いたもの.
Clone Line of Code	CLOC	コードクローンの合計行数を表す.
Population of a Clone	POP	入力されたソースファイル中に，対象のコードクローンが出現する回数を表す.
Coverage of Clone % FILE	CVR %FILE	合計ファイル数のうちコードクローンを含むファイル数の割合を表す.
Coverage of Clone % LOC	CVR %LOC	合計行数のうちのコードクローンの行数の割合を表す.

ソースコード **3.5**　**switch** 文におけるコードクローン

```
1  switch (c) {
2   case '0' : value = 0; break;
3   case '1' : value = 1; break;
4   case '2' : value = 2; break;
5   case '3' : value = 3; break;
6   case '4' : value = 4; break;
7   }
8
9  case 'a':
10  flag = 2;
11  break;
```

4. 実験

本章では実験する対象と内容について説明する．本研究では，多くの Arduino モジュールのライブラリが GitHub のリポジトリとして存在することと，実験対象となるプロジェクト数が多いことから，Google Big Query を用いて GitHub から Arduino 用ライブラリとそのライブラリを使用しているプロジェクトを取得する．その後，取得した Arduino プロジェクトに CCFinder を適用し，出力されるコードクローンに関するメトリクスを観測することで研究設問に答える．

4.1 実験の準備

実験に使用するツールと実験対象データの取得方法について説明する．

まず，実験でコードクローンを検出するためのツールである CCFinder (注 4) を入手する．公式の最新バージョンである CCFinder は 2010 年で開発が止まっており，当時の実行環境 (Win32，Ubuntu9.1) を用意することが困難であるため，GitHub 上で第 3 者によってフォークされたものを使用した．次に，実験対象データである Arduino プロジェクトを Google Big Query を用いて GitHub から入手する．この時，スター数に関する情報は Google Big Query の GitHub データベースに存在しないため，ウォッチ数が 10 以上のプロジェクトを集めたところ，170 個のプロジェクトを入手した．このとき使用したクエリを 4.1 に示す．さらに，Java のビルドツールである Apache Ant (v1.10.5) を Apache ソフトウェア財団のホームページ (注 5) [5] からダウンロードする．

(注 4): https://github.com/radekg1000/ccfinderx
(注 5): http://www.apache.org

ソースコード **4.1**　**Arduino** プロジェクトを取得するクエリ

```
1  SELECT
2    repo_name,
3    watch_count
4  FROM
5  [bigquery-public-data:GitHub_repos.sample_repos]
6  WHERE
7  repo_name IN(
8    SELECT sample_repo_name
9    FROM [bigquery-public-data:GitHub_repos.sample_contents]
10   WHERE content LIKE '%#include "Arduino.h"%' OR sample_repo_name LIKE
         '%Arduino%' OR sample_path LIKE '%.ino%'
11 )
12 ORDER BY watch_count DESC
13 LIMIT 200;
```

4.2 実験内容

各研究設問に対する実験内容について説明する．

4.2.1 RQ1

設問内容:Arduino 環境においてどの程度の割合でコードクローンが存在するか．

GitHub から入手した 170 個の Arduino プロジェクトに CCFinder を適用し，CVR % LOC を取得し，関連研究 [2] の結果から，gcc，JDK，Linux の CVR % LOC の値と比較する．

次に，コンパイル機能を持つ ESP8266 (注 6) [7] の Arduino ライブラリと，Java のコンパイラである Apache Ant に対しそれぞれ CCFinder を適用することで，同じ機能を持つプロジェクト間におけるコードクローンの検出結果を比較する．

4.2.2 RQ2

設問内容:ソースコードのどのような部分にコードクローンが存在するか．

RQ1 で得た 170 個の Arduino プロジェクトに含まれるコードクローンから，ソフトウェアへの影響が大きいコードクローンを選ぶため，CLOC が 500 行以上で POP が 10 回以上のコードクローンを集め，ソースコードのどのような部分にコードクローンが発生しているかを調べる．

4.2.3 RQ3

設問内容:組み込みソフトウェア特有のコードクローンは存在するか．

RQ2 で得たコードクローンのソースファイルを観察することで，そのコードクローンを含むプロジェクトの特徴を調べ，組み込みソフトウェア開発経験者の手で分類を行い，組み込みソフトウェア特有のコードクローンの存在について調べる．

(注 6): https://github.com/esp8266/Arduino.git

5. 結果

各研究設問に対する実験結果を示す.

5.1 RQ1

設問内容:Arduino 環境においてどの程度の割合でコードクローンが存在するか.

170 個の Arduino プロジェクト全体，Wi-Fi モジュールである ESP8266 の Arduino ライブラリ，Java のビルドツールである Apache Ant に含まれるコードクローンに関する情報と，関連研究 [2] より，gcc，JDK，Linux における CVR % LOC を表 5.1 に示す．また，170 個の Arduino プロジェクトそれぞれに CCFinder を適用し算出した各プロジェクトの CVR % LOC を図 5.1 に示す.

図 5.1 より，ESP8266 Library と Apache Ant の CVR の値を比較すると，ESP8266 Library の方が大きいことがわかった.

表 5.1　各プロジェクトのコードクローン情報

	FILE	LOC	CVR % LOC
Arduino Projects	25,882	6,595,226	62.3
ESP8266 Library	1,464	316,462	37.8
Apache Ant	1,272	272,359	26.6
gcc	221	460,000	8.7
JDK	1,877	570,000	29.0
Linux	6,497	4,365,124	22.7

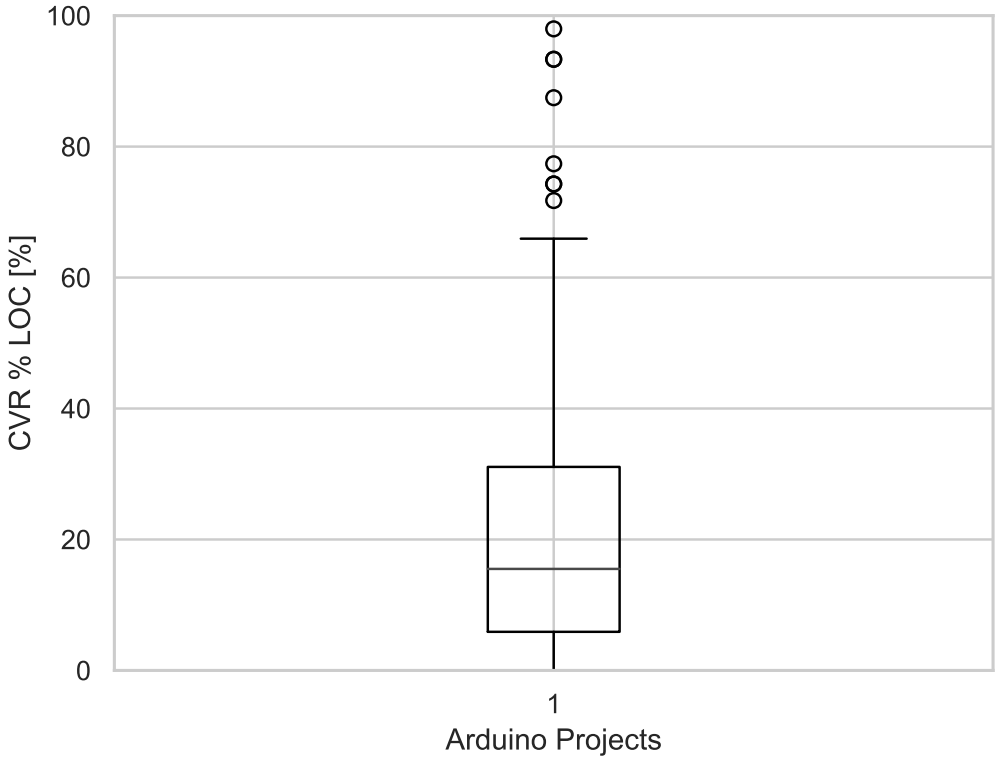

図 5.1　170 個の Arduino プロジェクトの CVR % LOC

5.2 RQ2

設問内容:ソースコードのどのような部分にコードクローンが存在するか.

4.2 節で説明した条件に基づきコードクローンを選出したところ，27 種類のコードクローンを得ることができた．これらのうち 5 種類を例に挙げ表 5.2 に示す．また，ソースコード 5.1 から 5.5 に記されている各コードクローンの例について説明する．

- **組み込み関数による SIMD 命令の羅列** ソースコード 5.1 は，ESP8266 のプロジェクトに含まれており，組み込み関数として使用されている SIMD 命令がコードクローンとして検出されている．各コードクローンの違いは，演算子や型の違いである．27 種類のコードクローンの合計行数のうち 22.9%を占めている．
- **レジスタ処理** ソースコード 5.2 は，ESP8266 のプロジェクトに含まれており，ネットワークのパラメータを得るための関数を再利用しているため，コードクローンとして検出されている．各コードクローンの違いは，対象となっているレジスタ名の違いである．27 種類のコードクローンの合計行数のうち 5.0%を占めている．
- **コンパイラによる構文解析** ソースコード 5.3 は，RTOS プロジェクト (注 7)[8] のビルドツールによる構文解析のための関数がコードクローンとして検出されている．各コードクローンの違いは，関数名の違いである．27 種類のコードクローンの合計行数のうち 2.7%を占めている．
- **ツールで自動生成されたコード** ソースコード 5.4 は，基板に非対応のモジュールを接続するための変換アダプタのプロジェクト (注 8)に含まれており，ソースコードのコメントから，自動生成されたものであることがわかった．各コードクローンの違いは，初期化するパラメータの違いである．27 種類のコードクローンの合計行数のうち 9.0% を占めている．
- **コアの種類によるクローン** ソースコード 5.5 は，SAM マイクロコントローラー [9] プロジェクトと AVR マイクロコントローラー [10] プロジェクト (注 9)に含まれており，異なるマイクロコントローラーに対し，同じライブラリを使ってい

(注 7): https://github.com/TrampolineRTOS/trampoline
(注 8): https://github.com/Microsoft/Windows-iotcore-samples
(注 9): https://github.com/adafruit/ESP8266-Arduino

る例である．全く同じライブラリを使用しているので，コードクローンの間に違いはない．27 種類のコードクローンの合計行数のうち 7.6%を占めている．

表 5.2　CVR の大きいコードクローン情報

クローンの種類	LEN(コードクローンの文字数)	POP(コードクローンの出現回数)
組み込み関数による SIMD 命令の羅列	2,626	46
レジスタ処理	1,511	20
コンパイラによる構文解析	1,081	11
ツールで自動生成されたコード	3,184	15
コアの種類によるクローン	4,037	10

ソースコード **5.1　SIMD** 命令の羅列によるコードクローン例

```
1  __attribute__( ( always_inline ) ) static __INLINE uint32_t __SADD8(
     uint32_t op1, uint32_t op2)
2  {
3    uint32_t result;
4
5    __ASM volatile ("sadd8␣%0,␣%1,␣%2" : "=r" (result) : "r" (op1), "r
       " (op2) );
6    return(result);
7  }
8
9  __attribute__( ( always_inline ) ) static __INLINE uint32_t __QADD8(
     uint32_t op1, uint32_t op2)
10 {
11   uint32_t result;
12
13   __ASM volatile ("qadd8␣%0,␣%1,␣%2" : "=r" (result) : "r" (op1), "r
       " (op2) );
14   return(result);
15 }
```

ソースコード **5.2**　レジスタ処理に関連するコードクローン例

```
1  static inline void emac_enable_transceiver_clock(Emac* p_emac,uint8_t
     uc_enable)
2  {
3    static inline void emac_enable_receive(Emac* p_emac, uint8_t
       uc_enable)
4    {
5      if (uc_enable) {
6        p_emac->EMAC_NCR |= EMAC_NCR_RE;
7      } else {
8        p_emac->EMAC_NCR &= ~EMAC_NCR_RE;
9      }
10   }
11
12   static inline void emac_enable_transmit(Emac* p_emac, uint8_t
       uc_enable)
13   {
14     if (uc_enable) {
15       p_emac->EMAC_NCR |= EMAC_NCR_TE;
16     } else {
17       p_emac->EMAC_NCR &= ~EMAC_NCR_TE;
18     }
19   }
20 }
```

ソースコード **5.3**　コンパイラの構文解析によるコードクローン例

```
1 GALGAS_arxmlMetaClasslist GALGAS_arxmlMetaClasslist::add_operation (
    const GALGAS_arxmlMetaClasslist & inOperand,C_Compiler
2 COMMA_UNUSED_LOCATION_ARGS) const
3 {
4   GALGAS_arxmlMetaClasslist result ;
5   if (isValid () && inOperand.isValid ()) {
6     result = *this ;
7     result.appendList (inOperand) ;
8   }
9   return result ;
10 }
11
12 GALGAS_locationList GALGAS_locationList::add_operation (const
    GALGAS_locationList & inOperand,C_Compiler
13 COMMA_UNUSED_LOCATION_ARGS) const
14 {
15   GALGAS_locationList result ;
16   if (isValid () && inOperand.isValid ()) {
17     result = *this ;
18     result.appendList (inOperand) ;
19   }
20   return result ;
21 }
```

ソースコード **5.4**　自動生成されたコード中のコードクローン例

```
1   ref class AdapterAttribute : BridgeRT::IAdapterAttribute
2   {
3   public:
4
5       virtual property BridgeRT::IAdapterValue^ Value
6       {
7           BridgeRT::IAdapterValue^ get() { return this->value; }
8       }
9
10      virtual property BridgeRT::IAnnotationMap^ Annotations
11      {
12          BridgeRT::IAnnotationMap^ get()
13          {
14              return ref new BridgeRT::AnnotationMap(this->annotations
                    );
15          }
16      }
17
18      virtual property BridgeRT::E_ACCESS_TYPE Access
19      {
20          BridgeRT::E_ACCESS_TYPE get()
21          {
22              return this->access;
23          }
24
25          void set(BridgeRT::E_ACCESS_TYPE accessType)
26          {
27              this->access = accessType;
28          }
29      }
30
31      virtual property BridgeRT::SignalBehavior COVBehavior
32      {
33          BridgeRT::SignalBehavior get() { return this->covBehavior; }
34
35          void set(BridgeRT::SignalBehavior behavior)
36          {
37              this->covBehavior = behavior;
38          }
39      }
40  }
```

ソースコード **5.5**　コアの種類によるコードクローン例

```
1  String::String(const char *cstr)
2  {
3    init();
4    if (cstr) copy(cstr, strlen(cstr));
5  }
6
7  String::String(const String & value)
8  {
9    init();
10   *this = value;
11 }
12
13 String::String(const __FlashStringHelper *pstr)
14 {
15   init();
16   *this = pstr;
17 }
18
19 #ifdef __GXX_EXPERIMENTAL_CXX0X__
20 String::String(String &&rval)
21 {
22   init();
23   move(rval);
24 }
25 String::String(StringSumHelper &&rval)
26 {
27   init();
28   move(rval);
29 }
```

5.3 RQ3

設問内容:組み込みソフトウェア特有のコードクローンは存在するか.

RQ2 への実験結果で得たコードクローンのうち，組み込みソフトウェア特有であるものの情報を表 5.3 に示す.

また，各コードクローンの例について説明する.

RQ2 で選出した 27 種類のコードクローンの内，組み込みソフトウェア特有であると思われるものは 13 種類あった．また，27 種類のコードクローンの合計行数の内，組み込みソフトウェア特有のコードクローンの割合は 40.0%であった.

- **コアの種類** 5.2 節で説明した通りである．コアモジュールの種類が異なるが，同じソースコードを用いているので，組み込みソフトウェア特有のクローンであると判断した.
- **ディスプレイの種類** ソースコード 5.6 は，ESP8266 のディスプレイ制御プロジェクト (注 10)に含まれており，異なる種類のディスプレイに対し同じ制御ライブラリを使用し，コードクローンとなっているので，組み込みソフトウェア特有のコードクローンであると判断した．例は色彩制御ライブラリであるが，フォント制御ライブラリなど，ディスプレイの制御ライブラリには多数のコードクローンが存在する．27 種類のコードクローンの合計行数のうち 5.4%を占めている.
- **ファームウェアの動作モードの種類** ソースコード 5.7 は，温度調節機のプロジェクト (注 11)に含まれており，調節機の動作モードによって異なる制御ライブラリが用意されているが，ソースコードの内容が等しいため組み込みソフトウェア特有のコードクローンと判断した．27 種類のコードクローンの合計行数のうち 5.9%を占めている.

(注 10): ttp://github.com/energia/Energia
(注 11): https://github.com/matsstaff/stc1000p

表 **5.3**　組み込みソフトウェア特有のコードクローン情報

	LEN	POP	コード例
コアの種類	4,037	10	ソースコード 5.5
ディスプレイの種類	2,205	13	ソースコード 5.6
ファームウェアの動作モードの種類	3,115	10	ソースコード 5.7

ソースコード **5.6**　ディスプレイの種類によるコードクローン例

```
1  void Adafruit_GFX::drawCircle(int16_t x0, int16_t y0, int16_t r,
2  uint16_t color) {
3    int16_t f = 1 - r;
4    int16_t ddF_x = 1;
5    int16_t ddF_y = -2 * r;
6    int16_t x = 0;
7    int16_t y = r;
8
9    drawPixel(x0, y0+r, color);
10   drawPixel(x0, y0-r, color);
11   drawPixel(x0+r, y0, color);
12   drawPixel(x0-r, y0, color);
13
14
15 void Adafruit_GFX::drawCircle(int16_t x0, int16_t y0, int16_t r,
16 uint16_t color) {
17   int16_t f = 1 - r;
18   int16_t ddF_x = 1;
19   int16_t ddF_y = -2 * r;
20   int16_t x = 0;
21   int16_t y = r;
22
23   drawPixel(x0, y0+r, color);
24   drawPixel(x0, y0-r, color);
25   drawPixel(x0+r, y0, color);
26   drawPixel(x0-r, y0, color);
27 }
```

ソースコード **5.7**　動作モードによるコードクローン例

```
1  unsigned char hex_nibble(unsigned char data) {
2    data = toupper(data);
3    return (data >= 'A' ? data - 'A' + 10 : data - '0') & 0xf;
4  }
5
6  unsigned char parse_hex() {
7    unsigned char data;
8    while (Serial.available() < 2)
9      ;
10   data = hex_nibble(Serial.read()) << 4;
11   data |= hex_nibble(Serial.read());
12
13   return data;
14 }
15
16
17 unsigned char hex_nibble(unsigned char data) {
18   data = toupper(data);
19   return (data >= 'A' ? data - 'A' + 10 : data - '0') & 0xf;
20 }
21
22 unsigned char parse_hex() {
23   unsigned char data;
24   while (Serial.available() < 2)
25     ;
26   data = hex_nibble(Serial.read()) << 4;
27   data |= hex_nibble(Serial.read());
28
29   return data;
30 }
```

6. 考察

6.1 RQ1

設問内容:Arduino 環境においてどの程度の割合でコードクローンが存在するか.

表 5.1 より，Arduino プロジェクト群の方が gcc，JDK，Linux に比べて CVR % LOC の値が大きく，より多くのコードクローンを含んでいることがわかる．同様に，ESP8266 Library と Apache Ant の CVR % LOC の値を比較すると，ESP8266 Library の方が大きく，より多くのコードクローンを含んでいることがわかる．

さらに，Miryung Kim ら [6] の実験結果から，多くの大規模プロジェクトにおける CVR % LOC はおおよそ 10%から 30%の間となっており，Arduino プロジェクトは全体で 62.3%であるので，他の大規模プロジェクトに対してコードクローンの発生率が高いと言える．

また，図 5.1 と R. Al Ekram ら [11] の実験結果を比較すると， 小規模な組み込みソフトウェアでないプロジェクトの CVR % LOC は平均 0.38%となっており，Arduino ソフトウェアそれぞれの CVR % LOC の平均値は 21.5%であるので，組み込みソフトウェアでないソフトウェアに比べてコードクローンの発生率が高いと言える．

以上のことから，Arduino プロジェクトには組み込み以外のソフトウェアより多くのコードクローンが発生していることがわかる．

6.2 RQ2

設問内容:ソースコードのどのような部分にコードクローンが存在するか.

組み込み関数による SIMD 命令の羅列はコードクローンとして検出された．しかしこれは，組み込み処理特有の最適化によるものであるため，避けることが困難な記法であると考えられる．

一方で，レジスタを直接参照する記述においてもコードクローンが多く発見されることがわかった．

同様に，5.2 節の結果と神谷ら [2] の実験結果から，ツールによって自動生成されたコードには多くのコードクローンが含まれることがわかる．原因として，自動生成ツールは同じ処理のコードを，使用する変数名のみを変更して生成するためであ

ると考えられる．

以上のことから，SIMD 命令の羅列や，レジスタの処理，自動生成によるコードなどにコードクローンが多く検出されるということがわかる．

6.3 RQ3

設問内容:組み込みソフトウェア特有のコードクローンは存在するか．

4.3 節で述べた方法に基づき，コアの種類，ディスプレイの種類，ファームウェアの動作モードの種類によって生まれるコードクローンを組み込みソフトウェア特有のコードクローンであると判定した．

また，Ekram ら [11] の研究から，RQ2 で検出した短い if 文のコードクローンと変数名のみが違うコードクローンは，テキストエディタやウィンドウマネージャーにも検出されたので，組み込みソフトウェア特有のものではないと考えられる．反対に，組み込みソフトウェア特有であると判断したコードクローンと同じ特徴をもつコードクローンは，Ekram ら [11] の研究おいて検出されなかった．

構文解析文におけるコードクローンは，Apache Ant において 3.1%，ESP8266 において 1.6%確認されたので，組み込みソフトウェア特有ものではないと判断した．

本研究で発見した組み込みソフトウェア特有のコードクローンが発生する原因として，伊藤ら [12] の研究から，あるハードウェアを改良して新たにハードウェアを開発する際，開発の効率を上げるためにソースコードを再利用するということや，別種のハードウェアであっても，機能が似ていれば同じ処理のソースコードが使われるということが考えられる．

Arduino 言語は C++言語ベースであるためモジュール化できず，関数化すると呼び出しの際にオーバーヘッドが生まれてしまうため，RQ2 で検出した組み込み関数によるインライン展開を利用するなど，対処する必要がある．Joseph Caldwell ら [13] の研究から，ソースコードの量を増やさずにインライン展開を行い，関数呼び出しの最適化をする手法が考案されている．この研究の結果から，この手法を C++言語プログラムに適用し 29%のオーバーヘッドを軽減できるので，C++言語ベースである Arduino ソフトウェアにも同等の結果が期待できる．

7. 課題

今後の課題として以下のようなことが挙げられる．

本研究ではオープンソースである Arduino のプロジェクトを実験対象としたが，他の組み込みソフトウェアではコードクローンはどのように発生しているのか調べる必要がある．

また，本研究で組み込みソフトウェアにコードクローンが発生するいくつかの原因を発見することができたが，発生した場合の対処方法と，コードクローンの発生を未然に防ぐ方法についてより深く考察する必要がある．

8. 結言

本研究では組み込みソフトウェアにおけるコードクローンの存在について考察する足がかりとして，GitHub から取得した 170 個の Arduino プロジェクトに対し，コードクローン検出ツールである CCFinder を適用した．実験の結果，Arduino プロジェクトには組み込み以外のソフトウェアより多くのコードクローンが発生していることがわかった．発見されたコードクローンのうち，SIMD 命令の羅列や，レジスタの処理，自動生成されたコードにコードクローンが多く見られた．また，組み込みソフトウェア特有と判断できるコードクローンも見られた．今後の課題として，Arduino 以外の組み込みソフトウェアにおけるコードクローンの発生状況を調べ，コードクローンが発生した場合の対処法を考える必要がある．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁重なご指導を頂きました，本大学情報工学人間科学系水野修教授に深く感謝いたします．本報告書執筆にあたり貴重な助言を多数頂きました，ソフトウェア工学研究室のみなさん，学生生活を通じて著者の支えとなった家族や友人に深く感謝いたします．

参考文献

[1] Arduino, （オンライン）, 入手先〈https://www.arduino.cc〉（参照 2019-1-26）.

[2] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: A multilinguistic token-based code clone detection system for large scale source code,” IEEE Transaction on Software Engineering, vol.28, no.7, pp.1–17, July 2002.

[3] Github, （オンライン）, 入手先〈https://github.com〉（参照 2019-1-26）.

[4] Google, Google BigQuery, （オンライン）, 入手先〈https://cloud.google.com/bigquery〉（参照 2019-1-26）.

[5] The Apache Software Foundation, （オンライン）, 入手先〈http://www.apache.org〉（参照 2019-1-26）.

[6] M. Kim, V. Sazawal, D. Notkin, and G.C. Murphy, “An empirical study of code clone genealogies,” ACM SIGSOFT Software Engineering Notes, vol.30, no.5, pp.187–196, Sept. 2005.

[7] ESPRESSIF, ESP8266（オンライン）, 入手先〈https://www.espressif.com/en/products/hardware/esp8266ex/overview〉（参照 2019-1-26）.

[8] trampoline, TrampolineRTOS, TrampolineRTOS/trampoline（オンライン）, 入手先〈https://github.com/TrampolineRTOS/trampoline〉（参照 2019-1-26）.

[9] MICROCHIP, Microchip/sam（オンライン）, 入手先〈https://www.microchip.com/design-centers/32-bit/sam-32-bit-mcus/sam-c-mcus〉（参照 2019-1-26）.

[10] MICROCHIP, Microchip/avr（オンライン）, 入手先〈https://www.microchip.com/design-centers/8-bit/avr-mcus〉（参照 2019-1-26）.

[11] R.A. Ekram，C. Kapser，R. Holt，M. Godfrey，“Cloning by accident: an empirical study of source code cloning across software systems,” International Symposium on Empirical Software Engineering, vol.10, no.7, pp.17–18, Dec. 2005.

[12] M. Ito, Y. Sakai, T. Sato, and M.S. madTakuya Matsumoto, “The method of extracting the reusable software assets for improving the quality in embedded software,” 2005.

[13] J. Caldwell and S. Chiba, “Reducing calling convention overhead in object-oriented programming on embedded arm thumb-2 platforms,” Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, vol.52, no.12, pp.146–156, Oct. 2017.