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

信学技報
TECHNICAL REPORT OF IEICE.
SS2000-49 (2001-01)

レガシーソフトウェアを対象とするクローンコードの定量的分析

中江 大海[1] 神谷 年洋[2] 門田 暁人[1] 加藤 裕史[3] 佐藤 慎一[1,3] 井上 克郎[1,2]
[1]奈良先端科学技術大学院大学 情報科学研究科
〒630-0101 奈良県生駒市高山町 8916-5
TEL:(+81)743-72-5312 FAX:(+81)743-72-5319
[2]大阪大学 大学院基礎工学研究科
[3]株式会社 NTT データ
E-mail: daikai-n@is.aist-nara.ac.jp , kamiya@ics.es.osaka-u.ac.jp , akito-m@is.aist-nara.ac.jp , katouhrb@nttdata.co.jp , s-sato@rd.nttdata.co.jp , inoue@ics.es.osaka-u.ac.jp

あらまし 本研究では, ある大規模なレガシーソフトウェアを題材として, クローンコードが持つ性質について, 定量的な面と定性的な面の両方から分析を試みた. 定量的な面からは, クローンコードに関するメトリクスを定義し, モジュール年齢, バグ数等との関係を分析した. 定性的な面からは, ソフトウェア中のクローンコードを実際に見て分析した. 分析の結果, 新しいモジュールほどクローンコードを多く含むこと, クローンコードの半数以上は年齢差が 30 日以内のモジュール間で発生していること, クローンコードを含むモジュールはクローンコードを含まないモジュールに比べて1行当たりのバグ数が少ないこと等がわかった.

キーワード クローンコードメトリクス, コピー&ペースト, レガシーソフトウェア, ソフトウェアの保守

Quantitative Analysis of Cloned Code on Legacy Software

Daikai Nakae[1] Toshihiro Kamiya[2] Akito Monden[1] Hiroshi Kato[3] Shin-ichi Sato[1,3] Katsuro Inoue[1,2]
[1]Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0101, Japan
TEL:(+81)743-72-5312 FAX:(+81)743-72-5319
[2]Graduate School of Engineering Science, Osaka University
[3]NTT DATA Corporation
E-mail: daikai-n@is.aist-nara.ac.jp , kamiya@ics.es.osaka-u.ac.jp , akito-m@is.aist-nara.ac.jp , katouhrb@nttdata.co.jp , s-sato@rd.nttdata.co.jp , inoue@ics.es.osaka-u.ac.jp

Abstract This paper shows the results of both quantitative analysis and qualitative analysis about the nature of cloned code on large legacy software. We defined some cloned code metrics, and then quantitatively analyzed the relation among the metrics and module's age, the number of bugs, etc. We also looked over the source code in order to qualitatively analyze the nature of cloned code. As the results, we found that new modules have more cloned code than old modules, more than half of cloned code were found between modules whose age are within 30 days difference, and the modules that have cloned code have fewer bugs (per line) than the modules that do not have cloned code, etc.

Keywords cloned code metrics, copy and paste, legacy software, software maintenance

1 はじめに

今日，多くの企業においてレガシーソフトウェア(Legacy Software:遺産的ソフトウェア)の保守コストの増大が問題となっている[1][7][9]. レガシーソフトウェアとは，10～20年以上も前に開発・リリースされ，今なお古いシステム上で動作し続けているソフトウェアのことである. ソフトウェアを長年にわたり使いつづけていると，度重なる機能変更，機能追加，修正のためにソフトウェアの規模や複雑さが増大し，保守性や信頼性が低下する.

レガシーソフトウェアの保守性・信頼性が低下する具体的な要因の一つとして，本研究ではクローンコード(Cloned Code: Code Clone, Duplicated Code とも呼ばれる)に注目する. クローンコードとは，コピー&ペースト等により生じる重複したコード対のことである. クローンコードの存在は，ソフトウェア保守作業を困難にする原因となるため，除去することが望ましいとされている[2][6]. 多数のクローンコードを含むソフトウェアでは，クローン部分のコードの変更が必要な場合に，変更すべき場所の特定や変更の波及個所の特定にコストがかかり，変更し忘れによるバグが混入することがある. ただし，保守作業者が意図的にクローンコードを生じさせる場合もあり，一概に全てのクローンコードを除去すればよいとは言えない. 例えば，著しく複雑な部分のコードを安全に変更するためには，変更したい部分をコピーしてクローンを作り，元のコードからクローン部分へと処理を分岐させて変更を加えることがある.

本研究では，ある大規模なレガシーソフトウェアを題材として，クローンコードの存在がソフトウェアの信頼性や保守性にどのように影響するか，および，そもそもクローンコードにはどのような種類があり，どのような性質を持つかについて，定量的，定性的な両面からの分析を試みる. 仮にある種のクローンコードが保守性に対して特に悪影響を与えることが明らかとなれば，そのようなクローンコードを除去したり，できるだけ生じさせないように保守現場へフィードバックすることで，保守性の向上に役立つと期待される.

関連研究としては，近年，Baker[1]が大規模 C 言語プログラム中のクローンコードを現実的な時間内で検出するアルゴリズムを提案したことがきっかけとなり，クローンコード検出に関する研究が数多く行われるようになった[2][3][9][10]. オブジェクト指向プログラムのみを対象としたものであるが，クローンコードを除去するための系統的な方法も提案されている[7]. これらの研究は，クローンコードを検出・除去する方法の開発が主目的であるのに対して，本研究では，検出したクローンコードの分析を主目的とする.

以降，2 章においてクローンコード検出方法について述べる. 3 章ではクローンコードに関する尺度について述べる. 4 章ではクローンコードの分析内容と結果を示し，最後に5章において全体のまとめと今後の課題について述べる.

2 クローンコードの検出

近年，クローンコードを検出する方法，および，検出ツールがいくつか提案されている[1][2][3][9][10]. これらのツールの多くは，ソースコードの一部分を他の部分と比較することによってクローンコードか否かの判定を行っており，完全に同一のコード対だけでなく，類似するコード対もクローンとみなして検出する. ソフトウェア開発・保守現場では，コピー&ペーストを行った後に内容の一部を変更することが実際にはよく行われるので，類似するコード対もクローンコードとして検出したほうが有用なためである.

本研究では，神谷らのツール(CCFinder)[9]を用いてクローンコードの検出を行った. 以降では，神谷らのツールを含め，代表的なクローンコード検出ツールを紹介する.

2.1 Baker の検出ツール

Baker のツール[1][2]は，クローンコード検出ツールの先駆けであり，コードを行単位で比較することによってクローンコードを検出している. また，このツールは検出の際に Parameterized Match という手法を用いている.

Parameterized Match 手法とは，変数や定数を共通のパラメータに置き換えた上で，一致するコード対をクローンコードと見なして検出を行うという手法である(図 1). この手法によって検出されたクローンコードは，共通のパラメータ部分を引数とする関数や手続きとして置き換えることが可能となる. そのような置き換えをすれば，コード量を抑制することができる上に，修正する場合も一箇所だけで済むため，ソフトウェアの保守性や信頼性が改善されると考えられている. Parameterized Match 手法は，その後に作成される他のツールにおける「類似コード対」の検出手法に少なからず影響を与えている.

2.2 Baxter らの検出ツール

Baxterらのツール[3]は，コードの構文解析を行うことによって，より正確な検出を行っている. しかし，構文解析を行うため，ツールの言語依存度が大きく，検出することができるのはC言語のクローンコードに限定されている.

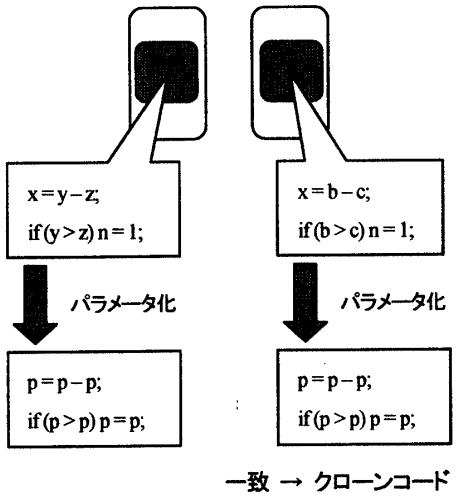

図1 Parameterized Match の例

2.3 神谷らの検出ツール

神谷らのツール(CCFinder)[9]は，トークン単位での比較による検出を行うことにより，行単位での比較では検出することができないようなクローンコードの検出を可能にしている(図 2)．クローンコードの検出においては，ユーザー定義名を共通トークンに置き換えた上で，一致するコード対を「類似コード対」として検出を行っている．この置き換え法はParameterized Match 手法と同じ類のものである．CCFinder では，ツール中のトークン解析部分だけを置き換えることで， C, C++, Java, COBOL 等多くの言語のクローンコードを検出することができる．

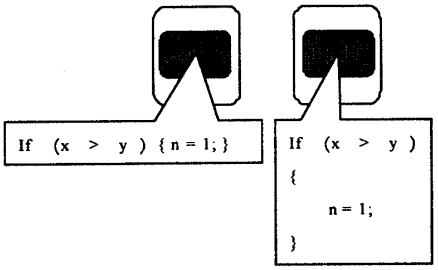

図2 行単位の比較では検出不可能だがトークン単位の比較では検出可能なクローンコードの例

3 クローンコードメトリクス

クローンコードを定量的に分析するためには，クローンコードに関する定量的な尺度(クローンコードメトリクス)が必要となる．ここでは，ソフトウェアのモジュール単位のクローンコードメトリクスとして，次の 2 つを定義する．

- クローン含有率
- クローン点在数

これらの定義を説明するために図 3 を用いる．図 3 にはA, B, C, Dの4つのモジュールが示されている．各モジュールのLOC(Lines Of Code:行数)はそれぞれ 100, 100, 50, 100 とする．各モジュール内に模様がついた四角形がいくつか存在するが，同じ模様の四角形が共通のクローンコードを表し，四角形内の数字はクローンコードの行数を表しているものとする．

クローン含有率は，LOC に対するクローンコード行数の割合と定義する．例えばモジュールAの場合，LOCは 100 であり，その中にクローンコードが全部で 45 行含まれているので，モジュール A のクローン含有率は 0.45 となる．

次に，クローン点在数を説明するために，「共通クローン含有群」という概念をまず定義する．共通クローン含有群は，あるモジュールXを基準とした場合の「モジュールXと共通のクローンコードを含むモジュールの集合」である．図3のモジュールBを例に取ると，この中には3種類のクローンコードが含まれている．一番上に位置するクローンコードはモジュール A, B, C に存在し， 真ん中のものはモジュール A, B に存在し，一番下のものはモジュールB, D に存在している．従って，モジュールBを基準とした共通クローン含有群は{A, B, C, D}となる．また，モジュールDを例に取ると，この中には1種類だけクローンコードが含まれており，それがモジュール B, D に存在しているので，モジュール D を基準とした共通クローン含有群は{B, D}となる．

クローン点在数は，あるモジュールを基準とした「共通クローン含有群の個数」と定義する．例えば，モジュールBを基準にとると，共通クローン含有群は{A, B, C, D}であるので，クローン点在数は 4 となる．また，モジュール D を基準にとると，共通クローン含有群は{B, D}であるので，クローン点在数は 2 となる．

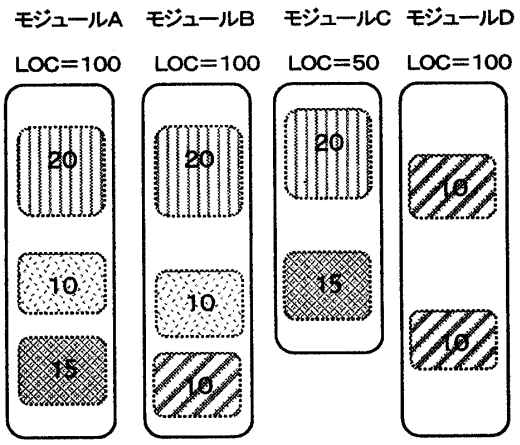

図3 クローンコードの分布例

4 クローンコードの分析

4.1 分析対象ソフトウェア

本研究で題材としたソフトウェアの特徴を次に示す.

- 構造化 COBOL(COBOL を拡張した言語)で記述されている.
- 今から 20 数年前に開発されて以来, 保守が続けられ, 現在においても実際に使われている.
- 1883 個のモジュールで構成されており, 全体のコード行数は 100 万行を越える.
- 各モジュールについて次のソフトウェアメトリクス値(ソフトウェアに関する定量的属性値)が既に計測されている.
  - 年齢(AGE: 日数で表す)
  - 行数(LOC)
  - 改版数(REV)
  - 過去 6 年間にバグを発生させた個数(BUGSUM)

4.2 分析手順

以下の手順で分析を行った.

1. クローンコードの検出: 分析対象となるソフトウェアから, CCFinder[9]を用いてクローンコードを検出した. CCFinder ではクローンコード最低行数を設定することができる. 予備検討の結果, この値を 30 に設定した. すなわち, 30 行に満たないクローンコードは検出しないようにした.
2. クローンコードメトリクスの計測: 1.の検出結果をもとに, 3 章で述べたクローンコードメトリクスを計測した.
3. クローンコードの定量的分析: 2.で計測したクローンコードメトリクスとソフトウェアメトリクス(AGE, LOC, REV, BUGSUM)との関係を分析した.
4. クローンコードの定性的分析: クローンコードメトリクスからだけでは計ることが困難と思われるようなクローンコードの性質を調べるために, ソフトウェア中のクローンコードを実際に見ることによって分析を行った.

4.3 クローンコードの定量的分析

(分析 1-1)

まず, 3 章において定義したクローンコードメトリクス(クローン含有率, クローン点在数)と AGE, LOC, REV, BUGSUM との各相関値を求めた(表 1, 2). その結果, これらの間には, いずれも顕著な相関は見られなかった.

表 1 クローン含有率との相関値

AGE	LOC	REV	BUGSUM
-0.127	0.203	0.0772	0.0775

表 2 クローン点在数との相関値

AGE	LOC	REV	BUGSUM
0.0523	0.312	0.146	0.011

(分析 1-2)

次に, より詳細な分析を行うために, クローンコードメトリクスと AGE, LOC, REV, BUGSUM との関係を表す散布図を求めた. その結果, LOC, REV, BUGSUM からは特徴的なことは見られなかったが, AGE についてはいくつか特徴的なことがわかった(図 4).

1. モジュールの年齢によって, 大きく 3 つの年代群に分けられる(閾値は AGE = 4500, 7000 とした). 以降では, これらの 3 つの年代群を便宜上, AGE の値が大きいものから順に Old, Middle, Young と呼ぶことにする.
2. 新しいモジュールほどクローン含有率が高く, 古いモジュールほどクローン含有率が低い. 各年代群のクローン含有率の平均値を表 3 に示す.
3. 各年代群(Old, Middle, Young)毎のクローン含有率が異なっているので, 年代をまたぐようなクローンコードは少ないと思われる(例えばもし Old と Middle との間でクローンコードが多い場合, 両群のクローン含有率がそろって上がる). 言い換えれば, クローンコードは各年代群内で閉じている傾向が強いようである.

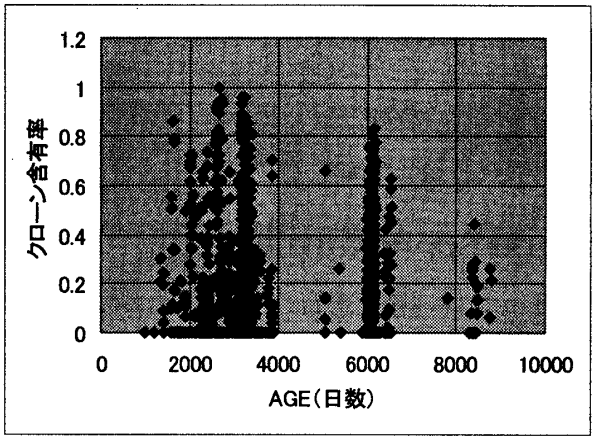

図 4 クローン含有率と AGE との関係(散布図)

表3 年代別クローン含有率平均値

Young	Middle	Old
0.175	0.128	0.107

(分析 1-3)

分析 1-2 の 3.を裏付けるため，各年代群内で閉じているクローンコードを含んでいるモジュールの数と，各年代群間にまたがるクローンコードを含んでいるモジュールの数を調べた．調査結果を表 4 に示す．表のかっこ内の数字はモジュール数である．例えば，Old モジュール間に存在するクローンコードを含むモジュールは全部で 11 個存在した(Old モジュールが全部で 29 個なので，割合は 37.9%)．また，Middle モジュールと Young モジュールにまたがるクローンコードを含むモジュールは全部で 77 個存在した(Middle モジュールと Young モジュールあわせて全部で 1854 個なので，割合は 4.1%)．表より，クローンコードは各年代内で閉じている傾向が強いことがうかがえる．

表4 年代別クローンコード保有モジュール割合

	Old (29)	Middle (802)	Young (1052)
Old	37.9% (11)	0.2% (2)	0.1% (2)
Middle	／	36.7% (295)	4.1% (77)
Young	／	／	44.7% (471)

(分析 1-4)

分析 1-3 の結果，クローンコードは各年代内で閉じている傾向が強いことがわかったが，例えば Young の年齢(日数)は約 1000 から約 4000 までと幅広いので，同年代のモジュール同士でクローンコードが存在していても，その年齢差が大きいか小さいかはわからない．そこで，同一クローンコードを含むモジュール間の年齢差を調べることにした．ただし，1つのモジュールには 2 種類以上のクローンコードを含む場合があり，また，1種類のクローンコードが 3 個以上のモジュールにまたがって存在する場合もあるので，ここでは，最大の年齢差についてのみ調査を行った．具体的には，「あるモジュールXを基準とした共通クローン含有群のうち，Xとの年齢差が最大のモジュールYを求める」ことにした．例えば，図 3 においてモジュール A, B, C, D の年齢をそれぞれ 1000, 2000, 3000, 4000 とすると，モジュール A を基準とする共通クローン含有群は{B, C}である．このうち，最大年齢差モジュールはＣとなり，AとＣの年齢差は 2000 となる．また，モジュール B を基準とした場合には，共通クローン含有群は{A, C, D}，最大年齢差モジュールはＤとなり，BとＤの年齢差は 2000 となる．このようにして各モジュールについて，最大年齢差モジュールとの年齢差を求めた結果を図 5 に示す．対象となったモジュールは全部で 935 個あったが，そのうち 1/3 以上が同年齢，1/2 以上が 30 日以内の年齢差のモジュール間でクローンコードが発生していることがわかった．従って，半分以上のクローンコードは年齢差の小さなモジュール間で発生していると言える．

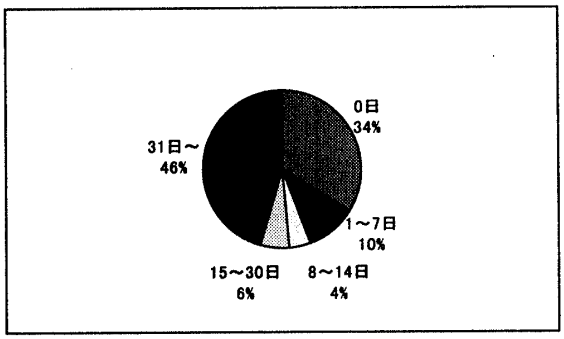

図5 クローンコード保有モジュール間最大年齢差

(分析 2-1)

分析 1-1 や 1-2 では，クローンコードを持っているモジュールと持っていないモジュールとを分けずに分析を行っていた．しかし，クローンコードの有無による差について詳細な分析を行うためには，これらを分類した方がよいと思われる．そこで，全モジュール 1883 個を次の 2 つに分類した．

- クローンコードを持っていないモジュール群(0群と呼ぶことにする)
- クローンコードを持っているモジュール群(1 群と呼ぶことにする)

0群に属するモジュール数は948であり，1群に属するモジュール数は 935 であった．

まず，各群における 1 モジュールあたりの AGE, LOC, REV, BUGSUM の値を求めた(表 5)．その結果，LOC, REV, BUGSUM の各値は，1 群の方が大きいことがわかった．特に 1 群の LOC 値は，0 群の LOC 値の約 2 倍の開きがあった(有意水準 5%で差があることが検定された)．

ところで，このように LOC 値に大きな開きがある場合，1モジュールあたりのバグ数(BUGSUM)の比較評価については注意が必要である．モジュールの規模(LOC)

が大きければ，それだけバグが入り込む余地が大きくなるため，LOC による要因を排除して評価を行うことが望ましい．そこで，BUGSUM／LOC（1 行あたりのバグ数）の値を用いて再評価を行うことにした（表 6）．すると，BUGSUM／LOC の値は，BUGSUM とは逆に 1 群の方が小さくなった（有意水準 5%で差があることが検定された）．従って，クローンコードを含んでいるモジュールの方が，クローンコードを含まないモジュールよりも 1 行あたりのバグ数が少ないといえる．

表 5 0 群と 1 群の AGE, LOC, REV, BUGSUM（平均値）

	AGEの平均	LOCの平均	REVの平均	BUGSUMの平均
0群	4479	420	41	0.161
1群	4310	845	58	0.244

表 6 0 群と 1 群の AGE, LOC, REV, BUGSUM／LOC（平均値）

	AGEの平均	LOCの平均	REVの平均	BUGSUM/LOCの平均
0群	4479	420	41	0.000432
1群	4310	845	58	0.000255

（分析 2-2）

分析 2-1 における 1 群については，クローン含有率の値によってさらに分類することができる．ここでは，1 群をさらに 5 つの群に分類した．この分類は，モジュールをクローン含有率の小さな順に整列した上で 5 等分することによって行った．便宜上，クローン含有率の小さな方から 1-a 群，1-b 群，・・・，1-e 群と呼ぶことにする．そして，各群における 1 モジュールあたりの AGE, LOC, REV, BUGSUM／LOC の値を求めた（表 7）．

表 7 から，1-e 群には AGE の平均が他の群に比べて小さいことが分かる．この結果は，分析 1-2 における「新しいモジュールほどクローン含有率が高く，古いモジュールほどクローン含有率が低い」という結果に合致している．また，1-a 群～1-e 群の間では BUGSUM／LOC の値にばらつきがあることが分かったが，その原因は現時点では不明である．

表 7 各群の AGE, LOC, REV, BUGSUM／LOC（平均値）

	AGEの平均	LOCの平均	REVの平均	BUGSUM/LOCの平均
0群	4479	420	41	0.000432
1-a群	4460	1074	73	0.000306
1-b群	4499	794	56	0.000243
1-c群	4507	801	52	0.000263
1-d群	4398	790	55	0.000168
1-e群	3687	764	52	0.000294

4.4 クローンコードの定性的分析

一口にクローンコードと言っても，様々な種類のものが存在すると考えられる．そこで，クローンコードにはどのような種類があり，それぞれどのような性質を持っているかを，実際にソースコードを参照することによって調べてみた．

そのために，ソースコードを参照する前にまず，CCFinder の検出結果データから特徴的なクローンコード数十個を選び出し，それらの特徴をまとめてみた．それらを以下に示す．

1. 行数が大きい．
2. 年代群にまたがって（Old-Middle, Old-Young, Middle-Young 間に）存在する．
3. 開始行番号が小さい．
4. モジュール内で閉じている．
5. 2 つのモジュール間にわたって存在する．
6. 3 つ以上のモジュール間にわたって存在する．
7. 2 つのモジュール間にわたって並行にいくつか存在する（図 6）．

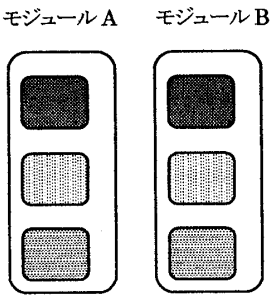

図 6 2 つのモジュール間にわたって並行にいくつか存在するクローンコードの例

次に，これらの特徴毎に，クローンコードのいくつかについてソースコードを実際に参照してみた．

（1.について）

行数が 500 を越える 4 組のクローンコードについて調査した．いずれも内容が同一（or 酷似）であり，コピー＆ペーストによるものと思われる．

（2.について）

年齢差の特に大きい 7 組のクローンコードについて調査した．1組のクローンコードはコピー＆ペーストによると思われるものであった．しかし，残りの 6 組はコピー＆ペーストによるものではないと思われた．例えば，クローン部分のコードは WRITE 文の連続であり，各行の変数名が全く異なっていた．

（3.について）

4 組について調査したが，いずれもプログラム先頭の初期設定部であり，多くのモジュールで同じような形の初期設定が行われているため，クローンコードとして検出されたようである．ただし，4 組のうち 1 組は，変数名がほぼ完全に一致しており，コピー＆ペーストによるものであると思われる．

(4.について)

3 組について調査したが，いずれもコピー＆ペーストによると思われるものであった．コード内容は次の通り．

- WHILE 文の内外で内容がまったく同じ．
- CASE 文中の各場合分け記述中において，変数名や代入値の 1 文字だけが異なる．
- IF 文内における LOOP 文の連続で，各 LOOP 文中における変数名のごく一部が異なる．

(5.について)

4 組について調査したが，2 組はコピー＆ペーストされたと思われるものであった．コード内容は次の通り．

- CASE 文内の各場合分け記述中において，代入文の各変数が異なる(構造をコピー＆ペーストしたと判断)．
- WRITE 文の連続であり，変数名と数字の一部が異なる．

残りの 2 組はコピー＆ペーストではなさそうであった．コード内容は次の通り．

- 手続呼出文の連続で，各手続名が異なる．
- 代入文の連続で，各変数名が異なる．

(6.について)

10 組について調査した．6 組はコピー＆ペーストされたと思われるものであった．コード内容は次の通り．

- 代入文＋IF 文(もしくは CASE 文)中の代入文の連続で，変数名と数字が一部異なる．

残りの 4 組はコピー＆ペーストではなさそうであった．コード内容は次の通り．

- 代入文の連続で，各変数名が異なる．

(7.について)

2 組について調査したが，いずれも内容が同一(or 酷似)であり，コピー＆ペーストによるものと思われる．

1.～7.の結果から，CCFinder が検出するクローンコードには，コピー＆ペーストによって生じたものと，そうではないものとの両方が含まれていることもわかった．1.の結果から，行数が大きなクローンコードは，一度に大量のコードをコピー＆ペーストしたようであり，内容が極めて類似している傾向が強いといえる．2.からは，年代が異なるモジュール間では，コピー＆ペーストはほとんど行われていないことがうかがえる．また，4.のようにモジュール内で閉じているクローンコードは，5.や 6.のようなモジュール間にまたがるものに比べて，コピー＆ペーストによる割合が多いことがうかがえる．また，7.のようなクローンコードは，コピー＆ペーストを行った後にコードの追加や修正が行われたのではないかと思われる．

全体的には，検出されるクローンコードが

- CASE 文や IF 文の各場合分け部
- WHILE 文や LOOP 文の内外
- WRITE 文や代入文の連続部

に集中していることもわかった．

5 まとめと今後の課題

本研究では，レガシーソフトウェアから検出されるクローンコードの性質を明らかにするために，定量的，定性的に分析を試みた．その結果わかったことを，以下にまとめる．

1. 新しいモジュールほどクローンコードを多く含む(分析 1-2)．
2. クローンコードは同年代に属するモジュール間で発生する傾向が強く，さらに半数以上のクローンは，年齢差が 30 日以内のモジュール間で発生している(分析 1-3，分析 1-4，4.4 節の 2.)．
3. クローンコードを含むモジュールは，クローンコードを含まないモジュールと比べて 1 行当たりのバグ数が少ない(分析 2-1.)．
4. 今回検出したクローンコードには，コピー＆ペーストによって生じたものと，そうではないものとの両方が含まれる(4.4 節の 1.～7.)．
5. 行数が大きいクローンコードは一度に大量のコードがコピー＆ペーストされている(4.4 節の 1.)．
6. モジュール間に比べてモジュール内ではコピー＆ペーストが発生しやすい(4.4 節の 4.～6.)．

今回の分析では，クローンコードを含むモジュールの方が含まないモジュールよりも1行当たりのバグ数が少ないという結果が得られたが，このことから，クローンコードの存在がソフトウェアの信頼性を向上させるとは断定できない．なぜならば，今回計測したモジュールごとのバグ数(BUGSUM)は最近 6 年間に検出されたものだけを含むためである．今回の分析では，新しいモジュールほどクローンコードを多く含むことが分かったが，一般に，新しいモジュールほどバグを多く発生する傾向にある(残存バグ数が多いため)．そのため，クローンコードの存在がバグの発生に影響しているのか，モジュール年齢がバグ

の発生に影響しているのか，あるいは，その両方であるのかは，現時点では明らかでない．

今後の課題としては，得られたクローンコードのうちコピー&ペーストによって生じたものだけを取り出して分析することが挙げられる．コピー&ペーストによるものを詳細に分析することによって，コピー元とコピー先との区別ができるようになれば，クローンコードの生成メカニズムが明らかにできると期待される．

参考文献

[1] B. S. Baker, "A program for identifying duplicated code," Proc. Computer Science and Statistics : 24th Symposium on the Interface, March, 1992.
[2] B. S. Baker, "On finding duplication and near-duplication in large software systems," Proc. Second IEEE Working Conference on Reverse Engineering, pp. 86-95 Jul.1995.
[3] I. D. Baxter, A.Yahin, L. Moura, M. Sant'Anna, L.Bier, "Clone detection using abstract syntax trees," Proc. of IEEE International Conference on Software Maintenance (ICSM) '98, pp. 368-377, Bethesda, Maryland, Nov.1998.
[4] K. Bennett, "Legacy systems: Coping with success," IEEE Software, 12, 1, pp. 12-23, 1995.
[5] E. Burd, M. Munro, C. Wezeman, "Analysing large COBOL programs: the extraction of reusable modules," International Conference on Software Maintenance (ICSM) '96 November 4-8, 1996 in Monterey, California
[6] S. Ducasse, M. Rieger, S. Demeyer, "A language independent approach for detecting duplicated code," Proc. of IEEE International Conference on Software Maintenance (ICSM) '96, pp. 244-253, Monterey California, Nov. 1996.
[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, "Refactoring : Improving the design of existing code," Addison-Wesley, 1999.
[8] C. Jones, "Large software system failures and successes," American Programmer, pp. 3-9, Apr. 1996.
[9] 神谷年洋，楠本真二，井上克郎，"コードクローン検出における新手法の提案および評価実験，" 電子情報通信学会技術報告，ソフトウェアサイエンス，Jan. 2001.
[10] J. Mayrand, C. Leblanc, E. M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," Proc. of IEEE International Conference on Software Maintenance (ICSM) '96, pp. 244-253, Monterey, California, Nov. 1996.
[11] N. F. Schneidewind and C. Ebert, "Preserve or redesign legacy systems," IEEE Software, 15, 4, pp. 14-17, July/Aug. 1998.