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

コードクローンに着目した ソフトウェア保守支援ツールの設計と実装

山科 隆伸† 上野 秀剛† 伏田 享平† 亀井 靖高† 名倉 正剛† 川口 真司† 飯田 元†

†奈良先端科学技術大学院大学 情報科学研究科 〒630-0192 奈良県 生駒市 高山町 8916-5
E-mail: †{takanobu-y, hideta-u, kyohei-f, yasuta-k, nag, kawaguti}@is.naist.jp iida@itc.naist.jp

あらまし 多くの企業の開発現場でソフトウェアの保守作業の品質低下やコスト増大が問題になっている．これらの具体的な要因の一つにコードクローンの存在が指摘されている．我々の過去の調査では，企業の保守作業を行う開発者の多くは，コードクローンの存在をあまり意識せずに保守作業を行っていることがわかっている．そこで，この調査結果を基に，保守作業の際に開発者へコードクローンの存在を意識させるために必要な要件を抽出し，それらの要件を満たし保守作業を支援するツール SHINOBI を設計，実装した．そして，このツールが企業の大規模なソフトウェアに適用可能かどうかを検証した．

キーワード コードクローン，リアルタイム，レガシーソフトウェア，ソフトウェア保守

Design and Implementation of the Software Maintenance Tool based on Code Clone Detection

Takanobu Yamashina‡ Hidetake Uwano‡ Kyohei Fushida‡ Yasutaka Kamei‡
Masataka Nagura‡ Shinji Kawaguchi‡ Hajimu Iida‡

†Nara Institute of Science and Technology 8916-5, Takayama, Ikoma Nara, Japan
E-mail: †{takanobu-y, hideta-u, kyohei-f, yasuta-k, nag, kawaguti}@is.naist.jp iida@itc.naist.jp

Abstract In many software development companies, increasing maintenance cost and decreasing reliability of the product become serious problem. It is pointed out that code clone is one major causes of such problem. In our past study, most of maintenance developers for the enterprise software are not aware of code clones when they maintain the software. Then we retrieved requirements to remind programmers of code clones. In this work, we designed and implemented a software maintenance tool SHINOBI to satisfy these requirements for developers. We investigated whether this tool was usable for large enterprise software.

Keyword Code Clone, Real-Time, Legacy Software, Software Maintenance

1. はじめに

一般的に，大規模なレガシーソフトウェアは，長年にわたって機能の追加・変更や，フォールト修正が行われている．そして，コードクローンを含むモジュールの割合が大きいことが知られている[1]．コードクローンを含むモジュールに変更を加える時には，同様のコードクローンを含む他の全てのモジュールにも同様の変更を行わなければならないことが多く，保守コストが増大する原因となる．この際に，変更し忘れや見逃しが生じた場合には，フォールトが混入し，信頼性が低下する原因にもなる[2]．そのため，レガシーソフトウェア保守開発プロセスでは，コードクローンを認識して保守を行うことが特に重要である．しかし，コードクローンに関する多くの研究は，コードクローンの検出手法や検出率の向上に関して行われている[3]．我々の知る限りでは，検出されたコードクローンが保守プロセスでどのように役立っているのか，利用されているのかなど，実際の保守現場における開発者の行動を分析し評価した研究はほとんどない．そこで我々は，先行研究[4]において，開発者とコードクローンに関する問題点を整理し，開発者がコードクローンを意識して保守作業を行うための要件を定義した．

本研究では，これらの要件を満たす保守支援ツール SHINOBI を設計，実装した．SHINOBI は，開発環境に統合され，開発者がコードクローンを意識しなくても，リアルタイムにコードクローンを自動検出する．検出した結果をランキングし，開発者に表示する．開発者は，この表示を常に確認しながら作業を進めることができるため，作業効率の向上や品質の向上が可能になる．そして，実装したツールの性能評価を行い，企業の大規模なソフトウェアにも適用可能なことを確認した．

2. ソフトウェア保守時のコードクローン検出に関する問題点

我々の先行研究[4]では，ある企業においてコードクローンを含むソフトウェアに関する保守活動を観察した．その結果，保守開発者に対してコードクローンの存在する箇所を適切に修正させるためには，次のような問題があることがわかった．

・ あるコードの修正時に，関連するコードクローンを検出して修正するという意識は高くない．
・ コードクローンを的確に検出するためには，開発経験を要する．
・ 変数名が変更された場合にコードクローンを検出することは非常に難しい．
・ 仮にコードクローンを検出できたとしても，開発者は修正すべきかどうかを判断できないことがある．
・ プログラムを修正する前にコードクローンを検出しなければ，整合性のある修正をすることができない．

CCFinderX[5] や ICCA[6] などの代表的なコードクローン検出ツールによってコードクローン検出を行うためには，ソースコード変更のための編集作業とは別に，それらのツールを実行する手順が必要になる．ユーザの操作に特別な手順を必要とするため，積極的に利用されない．さらに，これらのツールは，いずれも検出されたコードクローンのコードを表示するのみであり，この情報のみでは，開発者は修正すべきかどうかを判断できないことが多い．

3. SHINOBI: コードクローンに着目したソフトウェア保守支援ツール

本研究では，コードクローンの存在を意識して保守開発を行うことを支援するツール SHINOBI を設計し，実装した．SHINOBI は，ソフトウェア保守を行う開発者の操作を監視する．そして，開発者がコードクローンを含むモジュールを変更する際に，同様のコードクローンを含む他の全てのモジュールを自動的に検出しそのリストを表示する．開発者は提示されたリストを参考に，必要に応じてそれらのモジュールに同様の変更を行うことができる．

本節では，まず 2 節で挙げた問題点解決のためにツールが満たすべき要件を定義する．そしてその後で，我々が実装したツール SHINOBI について述べる．

3.1. ツールが満たすべき要件

2 節で挙げた問題点を解決し，ソフトウェア保守時にコードクローン検出を支援するために必要な要件として，我々は先行研究において以下を挙げた．

・ 要件 1 (コードクローン検出の動機に関して)
開発プロセスに対して特別な手順の追加を必要とせず，コードクローンを自動的に検出できる必要がある．
・ 要件 2 (コードクローン検出方法に関して)
指定したパラメータの相違に関わらず，誰でも同じ結果が得られるコードクローン検出方法が必要である．また，変数名が変更された場合でも，容易に検出できる方法が必要である．
・ 要件 3 (検出したコードクローンの修正に関して)
検出結果表示の際には，検出したコードクローンのコードだけではなく，修正すべきか判断できるだけの情報を付加する必要がある．また，開発者がプログラムを修正する前に，コードクローンを提示する必要がある．

3.2. ツールの概要

図 1 は，我々が実装したツール SHINOBI を利用して開発を行っている画面である． SHINOBI は，統合開発環境の一部として動作する．そして，保守開発者がエディタを操作して行う編集作業を監視し，その際のエディタ上でのカーソル移動を検出する．保守開発者がカーソルを移動させた場合には，SHINOBI はカーソル移動先の近傍のコード列に対応するコードクローンが存在しないかどうかを，開発プロジェクト内の全ソースコードから検索する．コードクローンが存在する場合には， **Code Clone View** にコードクローンを含むモジュールのリストを表示する．モジュールリストは，カーソルの存在する部分のコード列との関連の強さを基にランキングされ，その順番で表示される．そして個々のモジュールに対してどのような経緯でコードクローンが発生したかの情報を，ソースコードリポジトリの分析により取得し併せて表示する．このように SHINOBI は，コードクローンが存在するモジュールのリストと，それらを変更すべきかどうか判断させるための情報を開発者に提示する．

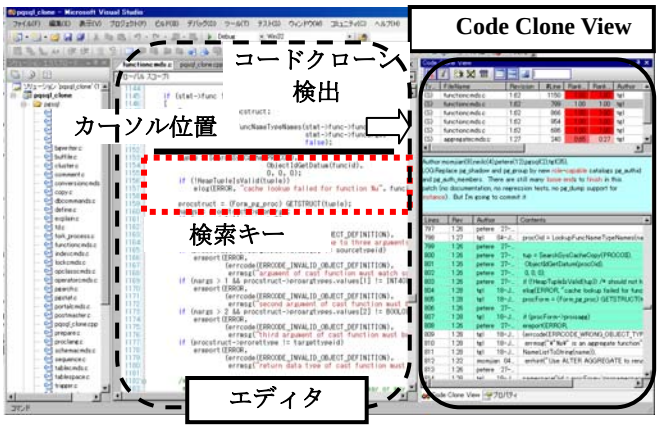

図.1 SHINOBI の起動画面

3.3. アーキテクチャ

SHINOBI のアーキテクチャを，図 2 に示す．
この図では, CVS によってソースコードの構成が管理されているプロジェクトにおいて, 開発者が保守開発を行っている．SHINOBI は, クライアントとサーバの 2 つの部分から構成されている． クライアントは，保守開発者の操作を監視し，開発者のカーソル移動に応じてサーバからコードクローンの存在するコードを検索する．サーバでは，CVS のソースコードリポジトリをあらかじめ分析してあり，クライアントからの問い合わせに応じてソースコードを検索し，対応するコードクローンが存在するモジュールのリストを応答として返す．そしてクライアントは，応答として受け取ったモジュールのリストを，Code Clone View に表示する．

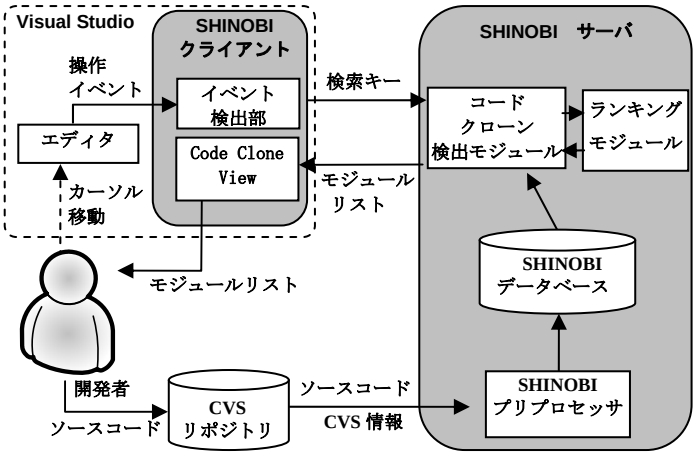

図 2. SHINOBI のアーキテクチャ

3.3.1. サーバ

図 2 に示すように，サーバは SHINOBI プリプロセッサ，SHINOBI データベース，コードクローン検出モジュール，ランキングモジュールの 4 つの部分から構成される．

SHINOBI プリプロセッサは，CVS のソースコードリポジトリをあらかじめ分析するためのモジュールである．対象プロジェクトに含まれる全ソースコードとコミット情報を取得し，検索を行いやすいようにインデックス化して SHINOBI データベースに保存する．コードクローン検出モジュールはクライアントからの検索要求を受け取ると，要求として送信されたコード列を利用して SHINOBI データベースを検索する．対応するコードクローンを含むモジュールが存在する場合には，ランキングモジュールを利用してランキングを行う．コードクローン検出モジュールは，ランキングを行ったモジュールリストを，クライアントへ検索結果として返す．

本節の残りでは，SHINOBI プリプロセッサによるソースコードリポジトリ分析方法，コードクローン検出モジュールによるコードクローン検出方法，ランキングモジュールによるランキングの方法について述べる．

ソースコードリポジトリ分析

SHINOBI プリプロセッサは，CVS リポジトリからソースコードと CVS コミット情報を取得し分析を行った結果を，SHINOBI データベースに登録する．

まず，ソースコードについては CCFinderX のプリプロセッサを利用してトークン解析を行い，ソースコード中に含まれるトークンをインデックス化する．次にモジュール間の関係の強さを測定するため，CVS コミット情報をトランザクション解析により解析する．トランザクション解析とは，ある一定時間内に開発者が同じコメントでコミットしているファイル群を，一つのトランザクションとみなし，トランザクション単位でグループ化を行う処理のことである[7]．今回の実装では，5 分以内にコミットしたものを一つのトランザクションとみなしている．そして，インデックス化したトークンと，トランザクション解析の結果抽出されたグループを，SHINOBI データベースに登録する．

このようにあらかじめトークン解析やトランザクション解析を行った結果を保持しておくことで，リアルタイムにコードクローン検出を行うことを可能にしている．またそれらの解析を行った後でリポジトリが変更された場合に対応するために，SHINOBI プリプロセッサは，常にリポジトリの監視を行う．そして，開発者が CVS リポジトリに新たにコミットを行った場合には，更新されたソースコードと CVS コミット情報を取り出して解析を行い，データベースの更新を行う．

コードクローン検出

コードクローン検出モジュールは，トークンベースでコードクローンの検出を行う．この方法は，プログラミング言語の構文規則に基づき，ソースコードをトークン別に変換するものである．コード列中の空白やコメント，インデントが異なったり，あるいは変数名等が書き換えられたりした場合でも，コードクローンを抽出することができる．

コードクローン検出処理では，検索キーをトークン解析しトークン列を求める．そして求めたトークン列と同じトークン列をもつ箇所を SHINOBI データベースから検索する．その際のデータ構造には，Suffix Array[8] を利用する．Suffix Array は，文書データベースから一致文字列を検索するためのデータ構造である． これは，文字列の全ての接尾辞のポインタを辞書順に格納した配列で，木構造よりも必要とするメモリが小さい．SHINOBI では，大規模なレガシーソフトウェアを対象に，コードクローン検出を高速に行うために，この Suffix Array を用いた．

コードクローンランキング

ランキングモジュールは，コードクローンを含むモジュール群を，編集中のコード列と関係の強い順にランキングする．本論文では，この関係の強さを表す度合いを論理的結合度と呼ぶ．論理的結合度の算出方法として，一般的に Apriori アルゴリズム[9] を使ったアソシエーションルールが利用されている．しかしこのアルゴリズムは算出に要する計算量が大きい．SHINOBI ではリアルタイム性を確保するため，このアルゴリズムを単純化したシンプルで高速な算出方法を利用して，論理的結合度を算出する．

いま，ファイル A の編集中に，ファイル B からコードクローンが検出された場合の，両ファイル間における論理的結合度の算出方法を述べていく．まず，２つのファイルが同時にコミットされた回数 $Count_{A \cap B}$ をファイル A のコミットされた回数 $Count_A$ およびファイル B のコミットされた回数 $Count_B$ で除算した割合 $P(B|A)$,$P(A|B)$を求める．

$$P(B \mid A) = \frac{Count_{A \cap B}}{Count_A} \quad , \quad P(A \mid B) = \frac{Count_{A \cap B}}{Count_B}$$

$P(B|A)$は，A が修正されたときに，B も同時に修正されている割合を示す．すなわち，割合が大きいほど同時に修正されており，論理的結合度が高いと考える．

そしてこの２つの値のうち大きい方を，ファイル A,B の論理的結合度 $LP(A, B)$とする．

$$LP(A,B) = \mathrm{MAX}(P(B \mid A), P(A \mid B))$$
$$= \mathrm{MAX}(\frac{Count_{A \cap B}}{Count_A}, \frac{Count_{A \cap B}}{Count_B})$$

ランキングモジュールは，論理的結合度 LP を，編集中のファイルと，コードクローンを含むすべてのモジュール群との間で算出する．そして，大きい順にランキングを行う．

3.3.2. クライアント

図 2 に示すように，クライアントは イベント検出部と Code Clone View から構成される．

イベント検出部は，エディタ上での保守開発者の編集操作を監視し，カーソル移動を検出する．カーソル移動を検出すると，その近傍のコード列に対応するコードクローンを含むモジュールをサーバから検索する．そしてサーバからの検索結果を得ると，その結果を Code Clone View に表示する．

SHINOBI クライアントは保守開発者の編集操作を監視できればよく，開発環境の種類に依存しないが，現段階では図 2 に示すように，Microsoft 社の Visual Studio 2005 の Add-In として動作するように実装している．そのため，イベント検出部は Visual Studio 2005 のエディタにおける編集作業を監視する．そして検出結果を表示する Code Clone View は，Visual Studio 2005 のペインとして実装している．

本節の残りでは，Code Clone View の詳細について述べる．

Code Clone View

図 3 は，Code Clone View の表示例である．Code Clone View は，検出したコードクローンのコードと，CVS リポジトリを

解析したことによって得られる情報を表示する．

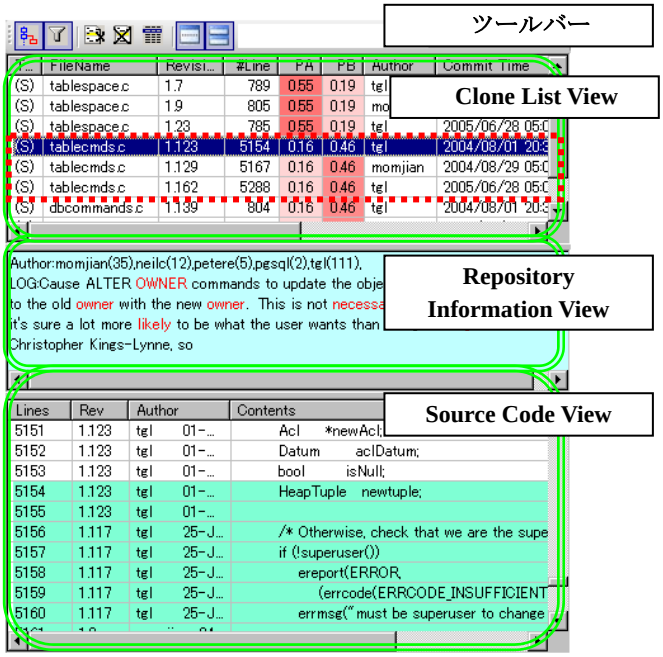

図 3 Code Clone View の詳細画面

Code Clone View は，ツールバー，Clone List View，Repository Information View，Source Code View から構成されている．ツールバーには，検出したコードクローンの表示の制御を行うためのコマンドが定義されている．これにより，表示対象外にするコードクローンやファイルを指定できる．Clone List View は，検出したコードクローンが存在するモジュールをランキング順に表示する．フィールドの先頭の記述(S)は，CVS リポジトリに含まれるソースコードからコードクローンを検出したものであることを意味する．これは，リポジトリに登録されて，他の開発者と共有されているソースコードにコードクローンが検出されたことを意味する．この欄が(L)と記述される場合は，保守開発者が編集を行っているクライアント端末で編集中のソースコードからコードクローンを検出したことを意味する．Clone List View にはこれ以外に，各モジュールに関する情報として，ファイル名，リビジョン番号，クローンの検出された行数，3.3.1 節で述べた論理的結合度の値 *P(B|A)* および *P(A|B)*，そのリビジョンを編集した開発者，編集した日付が表示されている．SHINOBI は CVS リポジトリの情報から，コードクローンが存在するモジュールが，リビジョン毎にどのように変遷しているかを分析する．そして，コードクローンが存在する全てのリビジョンを，Clone List View に表示している．例えば，図 3 の点線枠で囲まれている部分には，tablecmds.c の複数のバージョンが表示されている．これは，コードクローンが発生したリビジョンと，そのモジュールに変更が加えられたリビジョンである．この例では，tablecmds.c の Rev 1.123 でコードクローンが発生していて，Rev 1.129，および Rev 1.162 でモジュールが修正されていることを示している．

Repository Information View は，Clone List View で選択されているモジュールに関する CVS の履歴情報を表示する．表示する情報は，編集を行った開発者，開発者別の修正回数，コメントである．

Source Code View は，Clone List View で選択されているモジュールのソースコードを表示する．この中で，コードクローン片は色づけされて表示され，開発者がその内容を確認することができる．また，編集されたバージョンと，編集を行った開発者を行毎に表示している． このため，誰がコードクローンのどの部分をいつ編集したかを，容易に把握できる．

3.4. 特長・効果

本節では SHINOBI の特長と効果を，3.1 節に挙げた要件と対応させて，列挙する．

- 効果 1 (コードクローン検出の動機に関する改善)
  SHINOBI は，統合開発環境上での開発者の編集操作に応じて,コードクローンをリアルタイムに自動検出する．従って，開発者にコードクローン検出のために特別な手順の追加を必要としない．その上，今までと開発環境が変わらないため，学習コストも低い．
  また，開発者はコード編集前に他のコードクローンを確認できる．このため，全てのコードクローンに対して，一貫した方針でコードを設計できるようになる．
- 効果 2 (コードクローン検出方法に関する改善)
  SHINOBI は，カーソル位置の近傍のコード列に関連したコードクローンを自動検出する．従って，開発者の経験に関わらず，誰でも同じコードクローンが検出できる．
- 効果 3 (検出したコードクローンの修正に関する改善)
  SHINOBI は，検出したコードクローンを，修正中のソースコードと関係の強さ順にランキングを行い表示する．また，検出したコードクローンのコードだけではなく，リポジトリから取得した情報（いつ，だれが，どのような理由でコミットしたか）も表示する．従って開発者は，そのコードクローンを修正すべきかどうかを効率よく判断できる．

4. 評価実験

SHINOBI が大規模なソースコードに対して適用可能かを調査するために，動作時間に関して評価実験を行った．具体的には，(1) プリプロセッサが，CVS リポジトリの情報を読み取り SHINOBI データベースを構築するまでの時間，(2) コードクローン検出モジュールが，SHINOBI クライアントからの検索要求を受け取ってからコードクローンを検索する時間，以上 2 点について評価した．SHINOBI では，保守作業中のカーソル移動のイベントにあわせてリアルタイムにコードクローン検出を行う必要があるため，特に(2)コードクローン検出モジュールの検索は非常に高速に動作する必要がある．

対象としたプロジェクトは，PostgreSQL と 12 個のサブシステムからなる商用アプリケーションである．商用アプリケーションに関しては対象サブシステム数を 1 つ，4 つ，12 つとした場合のそれぞれについて計測し，規模に対する検出速度の推移を調査した．各ソフトウェアは主に C 言語，C++言語のソースコードから構成されている．また，ソースコードの差分情報は含めず，最新のソースコードのみを対象とした．実験環境としてはクライアント・サーバを同一マシンで動作させた．マシンのスペックは Pentium IV 3.6 GHz，メモリ 1GB，OS は Windows XP である．

4.1. SHINOBI データベース構築時間

本節では，プリプロセッサが SHINOBI データベース作成するのに必要な時間の測定結果について述べる．SHINOBI データベース作成では，まず指定した CVS リポジトリからソースコードをチェックアウトした後，CCFinderX のプリプロ

セッサを利用してトークン解析が行い，トークン解析の結果から Suffix Array を作成する．これらの一連の実行時間をプロジェクト毎に測定した．

測定結果を表1,2に示す. データベース構築時間としては，ソースコードが 100 万ステップで 2 分程度，400 万ステップを越えるような大規模な対象であっても 20 分強で完了している．また，トークン数と Suffix Array の作成時間は，ほぼ比例関係になっている．これは，Suffix Array 作成のソートアルゴリズムにマルチキークイックソート[10]を採用しているためである．

なお，PostgreSQL のソースコードのチェックアウト時間がトークン数に比べて時間がかかっているが，これはソースコード以外のファイルがリポジトリに多数含まれているのに対し，商用アプリケーションのリポジトリはほぼソースコードのみで構成されているからである．

表 1. PostgreSQL のデータベース構築時間

ファイル数		1,165
トークン数(M)		1.7
ファイルサイズ(MByte)		38.2
コード行数(MLOC)		0.6
Suffix Array インデックスサイズ (Mbyte)		8.4
実行時間	ソースコードチェックアウト(sec)	50
	CCFinderX プリプロセッサ時間(sec)	86
	Suffix Array 作成時間(sec)	2
	合計(sec)	138

表 2. 商用アプリケーションのデータベース構築時間

サブシステム数		1	4	12
ファイル数		4,377	9,260	13,844
トークン数(M)		8.4	16.7	21.9
ファイルサイズ(Mbyte)		186	366	401
コード行数(MLOC)		1.7	3.2	4.5
Suffix Array インデックスサイズ (Mbyte)		42	84	110
実行時間	ソースコードチェックアウト(sec)	35	93	148
	CCFinderX プリプロセッサ(sec)	258	935	1124
	Suffix Array 作成(sec)	19	43	60
	合計(sec)	312	1071	1332

4.2. コードクローン検出速度

次にコードクローン検出モジュールがコードクローンの検出に要する時間について述べる．本実験では SHINOBI クライアントが検索キーを要求してからコードクローン検出リストが返ってくるまでの時間を 10 回計測し，その平均値を求めた．

コードクローン検出の実行速度時間を，表 3,4 に示す．結果をみると，表 4 が示す通り，400 万ステップを超えるレガシープログラムでも平均 500ms 以内に検索できている．

次に，トークン数と処理時間の関係を示す．図 4 に示したグラフでは横軸にトークン数，縦軸に処理実行時間をプロットしている．図 4 からわかるように，Suffix Array のインデックスを読み込む処理は，トークン数の増加に従いリニアに増加している．コードクローン検出に関しては，トークン数の増加に対し指数的に時間が増えることはない. これは，Suffix Array の検索は，バイナリサーチにて行われるため，速度が理論的には $LOG_2(N)$に比例することによる．

4.3. 実験結果についての考察

以上の結果から，SHINOBI は，大規模なシステムに関してのセットアップに関して十分に実用に耐えうると言える．データベースの構築については, 最初に 1 度必要なだけであり，その後は随時更新分のみを処理すればよい．また処理速度も O(n) であり大規模な対象であっても現実的な時間で終えることができる．コードクローン検出については，標準的なマシン構成で平均 500ms 以内に検索を完了しており，利用者は SHINOBI クライアントを組み込んだ開発環境をストレスなく利用できるといえる．

なお，今後の改善として，インデックスをサーバ上で常時メモリ内に読み込んでおくことで，メモリロード時間の短縮を図ることを考えている. 検索キートークン解析については，先読みにてあらかじめ解析しておくことで，カーソル移動の度にトークン解析が不要となり，さらなる速度改善が図れる．

表 3. PostgreSQL での検出実行時間 (10 回の平均)

Suffix Array インデックス読み込み(ms)	45
検索キートークン解析(ms)	140
コードクローン検出(ms)	1
合計(ms)	186

表 4. 商用アプリケーションでの検出実行時間 (10 回の平均)

サブシステム数	1	4	12
Suffix Array インデックス読み込み(ms)	156	187	250
検索キートークン解析(ms)	156	172	140
コードクローン検出(ms)	1	6	14
合計(ms)	313	365	404

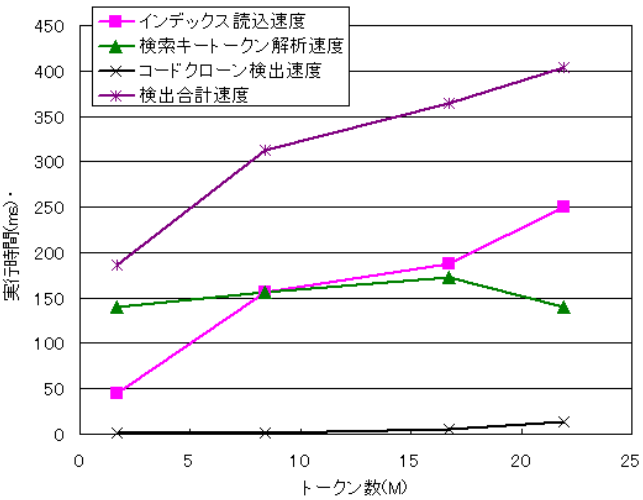

図 4 コードクローン検出実行時間

5. 関連研究

コードクローンを対象とした統合開発環境上で動作するツールとしては CloneDR[11], SimScan[12] が挙げられる，CloneDR は，Eclipse のプラグインとして動作し，コードクローン検出結果をビジュアルに表現している．SimScan も，

Eclipse のプラグインとして動作するコードクローン検出ツールである．これらのツールは，SHINOBI 同様，統合開発環境から呼び出せ非常に利便性は高い．ただしこれらのツールは対象システムに存在するクローンを俯瞰的に調査することが目的であり SHINOBI のようなコードクローンの自動提示を目的としていない．また，これらのツールは CVS リポジトリの履歴情報を分析対象としていない．

Suffix Array を用いたコードクローン検出ツールとして，RTF[13]は，コードクローン検出のアルゴリズムにトークンベースの Suffix Array を用いた検出を行っている．このアルゴリズムにより，本研究と同様に速度とスケーラビリティを確保している．このツールは，全ソースコード中からコードクローンを検出するコードクローン検出ツールであり，本研究のようにあるファイルに対するコードクローンをリアルタイムに開発者に知らせるようなツールではない．

また，SHINOBI 同様，開発環境上で保守作業に有用な情報を提示しソフトウェア保守の支援を行う手法として，様々な提案がなされている．CodeBroker[14] は，開発者がソースコードに記述した JavaDoc を自動的に取得し，LSA を用いて類似したコメントを持つメソッドを提示する．Selene[15]は，編集中のプログラムに関係の深いコードを自動的に検索し提示する． ROSE[7]は，Eclipse のプラグインとして動作し，CVS リポジトリに同時コミットされたファイル群を分析・提示するツールである．

そのほか，リポジトリを利用したコードクローン検出として，版管理システムを用いたクローン履歴分析手法[16]にてクローンの危険度を求める研究が行われている．

6. 結論および今後の研究

本論文では，ソフトウェア保守時に，コードクローン検出に関しての問題点を解決するために，リアルタイムコードクローンランキング提示ツール SHINOBI を設計し，実装した．SHINOBI は，統合開発環境のアドインとして動作し，開発者に必要なコードクローン情報を自動的に常に表示し気付きを促す．これにより，開発者は，プログラムの修正方針を早期に検討し，保守対象ファイルの修正漏れをなくし，リファクタリングを効率良く行うことが可能となる．また，本ツールは，市販の大規模レガシーシステムにも十分な高速性をもって動作すること示した．

今後の研究としては，本ツールをレガシーシステムの保守作業に長期に渡り適用し，導入前と導入後でコードクローン修正漏れの割合やコードクローン増加率の推移を調べ，ツールの有効性を確認する予定である．また，プロジェクト経験が短い開発者の修正プロセスがどのように変化があったのかを観測する．コードクローン検出時に，クローンパターンにあったトークン長の動的計算，ギャップクローンへの対応を行い検出精度の向上を図る．さらに，ランキングの評価を行い，必要があればランキング計算アルゴリズムの改善を図りたい．

謝辞

本研究の一部は，文部科学省「次世代 IT 基盤構築のための研究開発」の成果に基づく．

文　献

[1] A. Monden, D. Nakae, T. Kamiya, S. Sato and K. Matsumoto, “Software Quality Analysis by Code Clones in Industrial Legacy Software, ” IEEE Symposium on Software Metrics, pp. 87-94, 2002.

[2] B. Lague, D. Proulx, J. Mayrand, E. Merlo and J. Hudepohl, “ Assessing the Benefits of Incorporating Function Clone Detection in a Development Process, ” Proc. IEEE Int'l Conf. on Software Maintenance, pp. 314-321, 1997.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, “Comparison and Evaluation of Clone Detection Tools, ” IEEE Trans. on Software Engineering, vol. 33, no. 9, pp. 577-591, 2007.

[4] 山科隆伸，上野秀剛，伏田享平，亀井靖高，名倉正剛，川口真司，飯田元, “レガシーソフトウェア保守プロセスにおける開発者によるコードクローン認識についての観察, ” 1ZK-1，第 70 回全国大会講演論文集, 2008.

[5] http://www.ccfinder.net/ccfinderx.html

[6] http://sel.ist.osaka-u.ac.jp/icca/index-e.html

[7] T. Zimmermann, P. Weissgerber, S. Diehl and A. Zeller. “Mining Version Histories to Guide Software Changes, ” Proc. of IEEE Int'l Conf. on Software Engineering, pp.563–572, 2004.

[8] U. Manber and G. Myers, “Suffix Arrays: A New Method for On-Line String Searches, ” SIAM J. Comput. 22(5), pp. 935-948, 1993.

[9] R. Agrawal and R. Srikant. “Fast Algorithms for Mining Association Rules, ” Proc. of the Twentieth International Conference on Very Large Data Bases, pp.487-499, 1994.

[10] J. Bentley and R. Sedgewick, “Fast Algorithms for Sorting and Searching Strings, ” Proc. of the 8th Annual ACM SIAM Symposium on Descrete Algorithms, pp.360-369, 1997.

[11] R. Tairas, J. Gray and I. Baxter, “Visualization of clone detection results, ” Proc. of the 2006 OOPSLA Workshop on Eclipse Technology eXchange, pp.50-54, 2006.

[12] http://www.blue-edge.bg/simscan/simscan_help_r1.htm

[13] H. Basit, S. Pugliesi, W. Smyth, A. Turpin and S.Jarzabek,. “Efficient Token Based Clone Detection with Flexible Tokenization, ” Proc. of 6th Joint Meeting on ESEC/FSE, pp. 513-515, 2007.

[14] Y. Ye and G. Fischer, “Supporting reuse by delivering task-relevant and personalized information, ” Proc. of 24th Int. Conf. on Software Engineering (ICSE 2002), pp. 513–523, 2002.

[15] 渡邉卓也，増原英彦,”類似プログラムの提示ツール Selene.,” 日本ソフトウェア科学会第 24 回大会論文集, 3B-2, 2007.

[16] S. Kawaguchi, M. Matsushita and K. Inoue, “Code Clone Origin Analysis Using Version Control System, ” Technical report of IEICE. SS, pp.43-48, 2005.