ソフトウェア工学 100－8
（1994．9．8）

構文木の比較によるプログラム開発履歴分析ツールの試作

会沢 実† 練 林† 荻原 剛志‡ 井上 克郎† 鳥居 宏次†‡

†大阪大学基礎工学部情報工学科
‡奈良先端科学技術大学院大学情報科学研究科
†〒 560 大阪府豊中市待兼山町 1-3
‡〒 630-01 奈良県生駒市高山町 8916-5

ソフトウェアの変更履歴はプログラムの各バージョンとして管理されるが，その情報を活用して保守作業を支援するためのツールは今まであまり提案されていない．

本研究では，ソフトウェアの複数のバージョンを比較することによって得られる差分情報を基に，開発履歴を分析するツールを提案し，その試作を行なった．各バージョン間の差分は構文木の比較によって求められ，プログラムの構文情報に則した対応結果を得る．また，構文木の比較を用いることにより，複数のバージョンを通しての関数，ブロックの対応をとることも容易である．ここで得られる履歴情報を利用して，保守作業を支援することが考えられる．

A Tool Which Analyzes Software Development Histories by Comparing Syntax Trees

Minoru AIZAWA† Lin LIAN† Takeshi OGIHARA‡ Katsuro INOUE† Koji TORII†‡

†Department of Information and Computer Sciences,
Faculty of Engineering Science, Osaka University
‡Nara Institute of Science and Technology
†1-3 Machikaneyama, Toyonaka, Osaka 560, Japan
‡8916-5 Takayama, Ikoma, Nara 630-01, Japan

A history of update or modification of software is saved as a series of versions. Few tools have been proposed to support maintenance using information included in histories of software.

We propose a tool which gets differential information between versions of software and analyzes its history. We are also developing a prototype of the tool. As differential information is obtained by comparing syntax trees of versions, it is suitable for program structure. Therefore, it is also easy to get correspondence of functions or blocks through many versions. This tool could support maintenance work providing history information of software.

1 はじめに

大規模なソフトウェアのソースプログラムは, 開発, 保守のそれぞれの過程を通じて頻繁に変更を加えられる. これら変更の履歴はプログラムの各バージョンとして管理され, デバグや機能の拡張など様々な目的に利用される. このような変更されたプログラムの履歴から, バグの作り込まれた時点の特定などの解析を行なう際, バージョン間の差分や対応部分に関する情報は非常に有用なものである.

各バージョンの間の対応部分を求めるためには, 2つのプログラムを比較するツールが必要である. このようなツールとしては, UNIX の `diff`[1] などが広く用いられている. `diff` では行単位のストリング比較を基にして差分を求めている [2]. しかし, C のプログラムのようにブロック構造を持つテキストの差分を求める場合, ネストの深さの変化や制御の流れの違いなども開発者にとっては重要な情報である. `diff` の差分の計算方法では, このようなプログラムの構造に関わる変化の有無をとらえることは困難である.

筆者らは, 2つのプログラムテキストの間の差分を求めるため, プログラムの構文木の比較を用いる方法を提案した [3]. ここでは, 木の比較アルゴリズムを利用して2つの構文木の各頂点の最大の対応を求めることにより, プログラムの比較を行なう. この方法を利用すると, プログラムの持つ構造を考慮に入れた上で, 開発者の直観により近いプログラム差分を求めることが可能である.

本研究では, この構文木の比較を複数のバージョンに適用して各バージョン間の差分を求め, この結果を利用してプログラムの開発履歴を分析するツールの試作を行なった. このツールを利用することにより, 開発者は各バージョンのプログラムテキスト間の対応を基にして, プログラム中のどの部分に変更が加えられたか, また, プログラム中のある特定の部分がどのバージョンで追加されたかを知ることができる.

本稿ではこれ以降, 2節で複数のバージョンの履歴の分析について述べ, 3節でソースプログラムを比較する方法, 4節で今回試作したツールの概要を, それぞれ説明する.

2 複数バージョン間の履歴の分析

ソフトウェアの開発において, ソースプログラムは頻繁に変更を加えられ, 保守の段階に入ってからも, 新たなバグの発見や機能の拡張の度に変更が繰り返される. このような変更の履歴はソースプログラムの各バージョンとして管理される. バージョン管理のためのシステムとしては RCS[4] や SCCS[5] が代表的な例として挙げられる. 開発, 保守に携わる作業者はこのようなバージョン管理システムを用いて過去の任意のバージョンのソースプログラムを復元, 参照できる.

一般に, 大規模なソフトウェアの開発, 運用では長期間にわたって多人数の作業者が関わることが普通である. 実際に設計仕様からコードを作成する作業者とテストをする作業者, 保守を行なう作業者がそれぞれ全く異なる場合も少なくない. 特に保守を行なう作業者にとっては, 保守すべきソースコードがどのような開発履歴を経てきたか分析を行なうことは重要であり, 分析のための適切な情報を効率良く提供することで作業者の負担を軽減することができる.

ここで, 過去の複数のバージョンから以下のような情報を取り出すことができれば, それは保守段階の作業にとって大変有用な情報となる.

- ある特定のブロックやステートメントは, いつどのバージョンで追加されたものか?
- ある特定の関数やブロックは, 各バージョンを通してどのように変更されてきたか?
- ある特定の関数やブロックに大規模な変更が行われたのはどのバージョンか?

保守の段階で新たにバグが発見された場合, そのバグを引き起こす原因となっていると推測されるステートメントがどのバージョンで付け加えられたものであるかを速やかに特定することで, 作業者はバグの修正をし易くなることが考えられる. また, 何度も変更されている不安定なコードを持つ関数が見つかれば, プログラムを再構成する際に特別に注意を払うこともできるであろう.

これらの情報は, それぞれのバージョンの間の差分情報を利用すれば, 容易に取り出すことができると考えられる.

```
 1: void sub(int M)
 2: {
 3:     int i;
 4:
 5:     i = 0;
 6:     while(i < M){
 7:         printf("\n");
 8:         printf("N =");
 9:         printf("%d",i);
10:         ++i;
11:     }
12: }
13:
14:
15: main()
16: {
17:     int MAX;
18:
19:     scanf("%d",&MAX);
20:     sub(MAX);
21:     printf("\n");
22: }
```

program-1

```
 1: void sub(int M)
 2: {
 3:     int i;
 4:
 5:     i = 0;
 6:     printf("\n");
 7:     printf("N =");
 8:     while(i < M){
 9:         if(i % 2){
10:             printf("%d",i);
11:             printf(", ");
12:         }
13:         ++i;
14:     }
15: }
16:
17:
18: main()
19: {
20:     int MAX;
21:
22:     scanf("%d",&MAX);
23:     sub(MAX);
24:     printf("\n");
25: }
```

program-2

図 1: プログラムの例

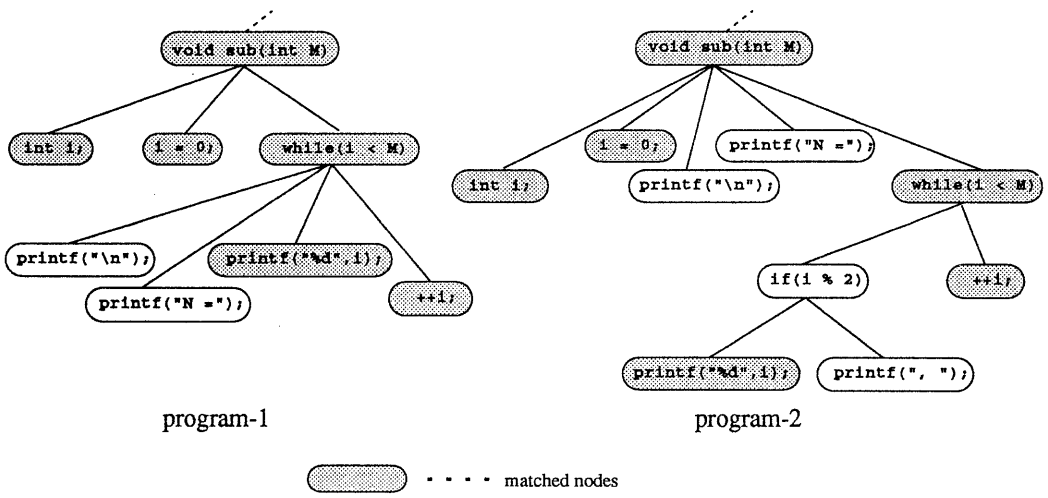

図 2: 構文木による比較

3 ソースプログラム間の比較

3.1 行単位の比較に基づくプログラムの差分

プログラムテキスト比較ツールとしては, UNIX の diff コマンドが広く使われており, 多くのバージョン管理システムでも, バージョン間の差分情報は diff によって得ている. diff では行単位のストリング比較に基づいた方法でファイル間の差分を求めている [2]. しかし, この方法では, C で記述されたプログラムのように構造を持つものを比較する際に, 構造的な違いの有無を反映しないということが考えられる.

例として図1に示すような2つの簡単なプログラムの比較を考える. これらのプログラムを diff を用いて比較した場合, 次のような結果が得られる.

```
6d5
<     while(i < M){
8a8,9
>     while(i < M){
>       if(i % 2){
9a11,12
>           printf(", ");
>       }
```

ここでは, while 文のループの内側から外側に2つのステートメントが移動したために, 本来変更のなかったはずの while 文の条件部が削除されたとみなされており, 実際の開発者の直観とはやや違った結果となっている.

diff コマンドは過去のバージョンの保存という観点からは簡便で優れているが, このまま複数バージョンの比較の目的に利用するには幾つかの問題がある.

まず, 始めに述べたように, diff は行単位のストリング比較に基づいて差分を求めているために, プログラムの内容に関係のないコメント文や改行位置などの変更によって各ステートメント間の正確な対応がとりにくくなってしまうことがある.

また, 上で挙げた例からも分かるように, 行単位での比較は人間の直観とはかけ離れた結果になりやすい. 人間がソフトウェアの比較を行なう場合は, まず, 関数や制御文などのブロック構造に注目して違いを見つけ出そうとする. 行単位での比較による差分情報を提示されても, 人間にはそのままでは理解できず, 保守作業を支援するためにはあまり役に立たない.

さらに, プログラムの持つ構造を考慮していないために, ブロック間の対応をとることは非常に困難である. 例えば, ある関数の内部に何らかの変更が加えられたかどうかといったことを知るのは容易ではない. 関数やブロックについて複数のバージョンにわたる追跡や分析を行うためには, プログラムを単なるテキストとして比較するのではなく, プログラムの構造に注目してその間の比較を行う必要がある.

3.2 構文木の比較に基づくプログラムの差分

プログラムの持つ構造を考慮した上でプログラムテキストの比較を行なうため, 筆者らはプログラムの構文木の比較に基づいてプログラムテキストの差分を得るツールを試作した [3].

ここでは, C プログラムを次のような構文木に変換して比較を行なう.

- プログラム中の各実行文 (代入文, if 文等の制御文), 変数宣言文, 関数定義部は, 構文木の頂点に 1 対 1 に対応づけられる.
- 構文木の根はプログラム全体を表現し, 全ての関数定義部は根の直下の頂点となる.
- 関数や制御文のブロック内の実行文, 変数宣言文の頂点は, その関数や制御文を表す頂点の子となる.
- 構文木に含まれる各頂点にはそれぞれラベルとして, 対応する実行文を表す文字列が付けられている. 木の根に当たる頂点にはラベルはない.
- 生成された木は, preorder でたどることによって元のプログラムテキストに復元することが可能である.

ここでは文とブロックの構造までしか構文木に反映させていないが, さらに変数や演算子などのレベルまでを木で表現し, 比較に利用することも可能である. なお, ファイル中に含まれるコメント文などは構文木に含まれていない.

筆者らはすでに, このような構文木間の差分を求めるためのアルゴリズムを提案している [3]. 2つの木の間の距離の定義は SSPM(Strongly Structure Preserving Mapping)[6, 7] による. SSPM の距離の計算法を利用すると, 木の頂点間の最大対応が求められる. 筆者らのアルゴリズムは SSPM に基づき, 2つの木の間の最大対応を直接求めるものである (アルゴリズムの詳細は [3] を参照).

このアルゴリズムを先述の構文木に適用して同じラベルを持つ頂点の組の最大集合を求める. 求められた集合に含まれない頂点は削除または挿入されたものとみなし, この結果をもとに2つのプログラムのステートメント毎の差分を求める.

例えば, 図 3に示すような 2 つのプログラムの構文木 T, T' の場合, SSPM に基づく頂点の最大の対応は {(1,1'), (2,2'), (3,4'), (5,5'), (6,6')} となる. (4,3'), (7,7') はラベルが等しいがこれらを対応の集合に入れると SSPM の条件を満たさなくなる. 従って差分としては, この場合 T から 4, 7 が削除され, T' へ 3', 7' が挿入されたと考える.

同様にして先ほどの図 1のプログラムの比較を行なうと, 図 2のような結果が得られる. この図は変更のあった関数の部分木について, 対応のとれている頂点を区別して示している. この場合には, 3.1節の `diff` の結果と異なり, while 文のループは互いに対応がとれ, 実際の変更に即した差分が求められている.

3.3 複数バージョン間の比較

上で示したように, 構文木に基づいてプログラムテキストを比較することで, 関数やブロックなどの構造を反映した比較が可能となる. このため, 人間が見ても直観的に分かり易い方法でバージョン間の差分情報を取り出すことができ, さらに, 複数バージョンの間の差分情報から, 保守作業に有用な情報を取り出すのも容易である.

構文木の比較に基づく, 複数バージョン間の各頂点 (ステートメント) の対応関係について, 図 4を使って簡単に説明する. この図では隣接するバージョン間で比較を行ない, その結果対応する頂点 (ステートメント) 同士を点線で結んで表している. この点線をたどることにより, 複数バージョンにわたるステートメントの対応を得ることができる.

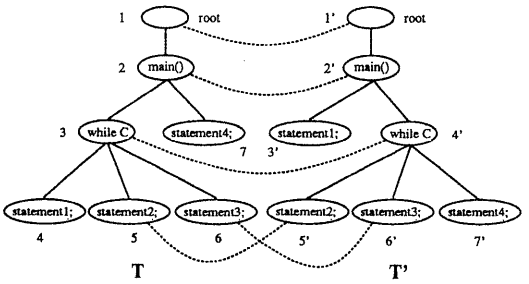

図 3: プログラムから得られる構文木の比較

また, 各内部頂点を結ぶ対応関係は, 関数やブロックについての対応を表している. これらステートメント毎の対応はテキストの行番号 (ファイル内の相対的な位置関係) に依存するものではなく, それぞれの属する構文木における頂点の相互の対応関係のみから得られる.

図 4に示したような各バージョン間の木構造の対応は, そのままプログラムの変更に関する情報として利用できる. 従って, 各バージョン間についてこのような情報を保持しておけば, 特定のステートメントやブロックに関して, 2節で述べたような複数バージョンにわたる解析を行うことは容易である.

4 ツールの試作

4.1 ツールの概要

構文に基づいたプログラムの差分情報を利用すれば, 複数バージョンの差分情報からプログラム開発履歴の分析を行うことができ, これによって保守作業の支援が可能になると期待される. 我々は, 構文に基づいたプログラムの差分情報が開発履歴の分析に実際に役立つことを示し, また, どのような支援が有効であるかを調べるためにツールの試作を行った.

今回試作したツールでは, 関数, ブロック, あるいはステートメントを指定し, それが追加されたバージョン, および追加された位置についての情報を得ることができる. 同時に, 制御文の挿入, 削除によるネストの変化などの構造上の変更に関す

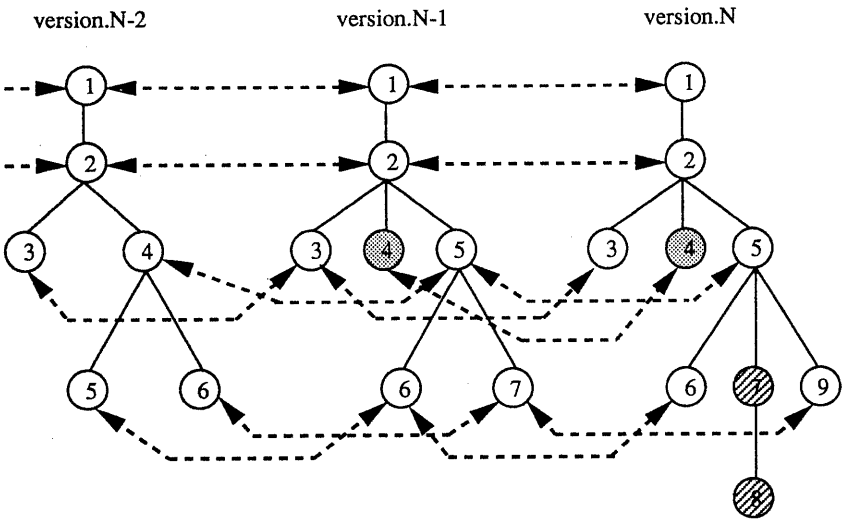

図 4: 複数バージョン間の対応の例

る情報も得ることができる.

ソースプログラムの各バージョンの比較は各関数毎に行なう. 従って, 作業者はある関数がどのバージョンで付加, あるいは変更されたかというような点について, 特定の関数についてのみ履歴を遡って分析をすることができる.

試作したツールでは各バージョンから履歴情報を取り出すために, 大きく分けて次の 3 つの処理を行う（概念図を図 5に示す）.

1. 各バージョンから構文木を生成し, 隣接する 2 つのバージョンの間で比較を行なう.
2. 1. で求まった結果から, 対応するステートメントの頂点番号の組を基に対応表を作成する.
3. 隣接する 2 つのバージョン間の対応表を合成してバージョン全体にわたるステートメントの対応を求める.

以下, 各部分の概要を述べる.

4.2 構文木の生成及び比較

今回のツールで各バージョンに対して生成される構文木は 3.2節で挙げたものと同様の構文木である.

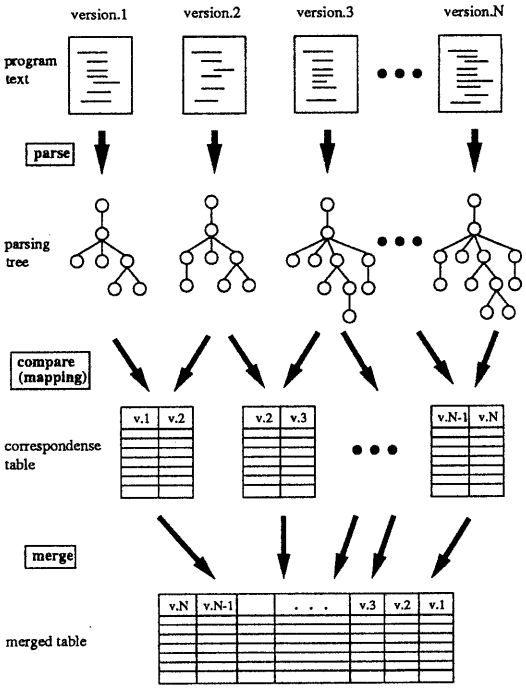

図 5: ツールの処理の概念図

ここでは，コンパイラが行なう構文解析のような，各文の詳細な解析に基づく構文木にはせず，実行文や宣言文を1つの頂点としている．このような，文や宣言毎の差分の方が，作業者が直観的に変更を把握し易いと考えられる．また，余分な頂点の対応を求める必要がなく，木の生成及び比較に要する計算時間を短縮できる．

また，ファイル中に含まれるコメント文などは構文解析時に読み飛ばされ結果に影響は与えない．

このようにして生成された構文木に対して比較を行なう．比較は関数単位で行なわれる．まず，構文木からそれぞれの関数にあたる部分木を取り出し，それらに対して3.2節で述べた木の比較アルゴリズムを適用する．構文木に含まれる各関数毎に部分木の比較を行なった結果，構文木全体の対応が求められる．

4.3 対応表の作成

各構文木の比較によって得られた対応関係の情報は，履歴情報の解析に利用するために保持しておかなければならない．しかし，構文木のまま対応を保持したのでは大量のスペースが必要となる．そこで，今回のツールでは対応表を作成し，対応する各ステートメントの頂点番号及びファイル内の行番号のみをテーブルに保持しておく方法をとった．この方法により，大規模なプログラムに対しても比較的少量のスペースで対応情報を保持できる．

対応表の例を図6に示す．この表では，対応するステートメントの番号を組にして示している．対応する番号のない場合は，ステートメントが追加，あるいは削除されたことを表している．

ver.N-2	ver.N-1
1	1
2	2
3	3
-	4
4	5
5	6
6	7

ver.N-1	ver.N
1	1
2	2
3	3
4	4
5	5
6	6
-	7
-	8
7	9

図 6: 隣接する2バージョン間の対応表の例

4.4 対応表の合成

隣接する2つのバージョン間の比較により得られたそれぞれの対応表を合成することにより，複数のバージョンにわたるステートメントの対応を求める．この合成は，それぞれの対応表の頂点番号をたどることにより，容易に行なうことができる．

作業者は最新バージョンに着目していると考えるのが最も適当であると考えられる．従って合成の際には，現在最新となっているバージョンを基準として頂点をたどっていく．新たにバージョンが追加された際にも，追加されたバージョンとそれまでの最新バージョンとの間の対応を求めるだけで容易に表の書換えができる．バージョン全体にわたり合成された対応表の例を図7に示す．

ver.N		ver.N-1	ver.N-2	···
1	R	1	1	···
2	F	2	2	···
3		3	3	···
4		4	-	···
5	C	5	4	···
6		6	5	···
7	C	-	-	···
8		-	-	···
9		7	6	···

図 7: 合成された対応表の例

図7の左端の列が最新バージョンの各頂点（ステートメント）であり，'R', 'F', 'C' の各印は，その頂点がそれぞれルート，関数定義部，制御文であることを示している．

4.5 ツールの利用

このツールでは，上で述べた方法によって各バージョンの間の対応関係を得た後，この情報を用いて開発履歴に関するさまざまな事柄を対話的に調べられる．

現在，最新バージョンにおいて指定した文やブロックがどのバージョンで追加，あるいは修正されたかを調べることができる．また，このツールの機能を拡張することによって，特定の関数の修正履歴を一覧表示させたり，頻繁に修正されている関数やブロックを検出するなどの機能も比較的容易に実現できると考えられる．

5 議論

前節でも述べたように, 今回試作したツールの目的は, 大規模なプログラムの保守を行なう作業者に対し, 開発過程における各バージョンの差分情報を履歴の分析に生かすことである. 過去のバージョンのソースコードは比較計算の段階で既に得られていることを前提として, 比較以降の処理に着目している. 従って, ここではRCSやSCCSのようなバージョン管理システムとは異なり, 差分の保存などバージョンそのものの管理方法については触れていない.

ツールから得られる情報を作業者に分かり易く提示するためにはインターフェース部における工夫が重要である. 現在は, 最新バージョンの文や関数を指定して, それが追加, 変更されたバージョンの情報を表示するだけであるが, 過去のバージョンの対応する箇所を復元して表示するなどの機能も必要となろう. このような過去のソースコードの保存及び復元を効率良く行なうことのできるシステムと組み合わせることによって, 今回試作したツールをより有効に利用できると考えられる.

また, 今回の試作では比較の単位を関数としたため, 複数のファイルから成るようなプログラムに関してはあまり注意を払っていない. 関数とファイルの関係や, さらに関数が局所的な定義かどうかなどの情報も利用できれば, プログラムの修正にとってより便利なツールとなるであろう.

従来, 過去のバージョンが持っている情報を積極的に活用するための方法やツールについては, あまり目立った提案がなされていなかった. 保守段階でのソフトウェアの修正や, 古いソフトウェアの書き直し (Re-Engineering) のためには, 過去のバージョンが持っている情報を活用することが重要であると考えられる.

今回のツールでは, バージョン間の構文の違いについてのみ注目をしているが, 関数や大域変数の参照関係の解析などの機能を組み込み, バージョン間の違いについてより幅広い検討が行えるようなシステムが必要であろう. このようなシステムが利用できれば, あるバージョンで変更が行われた理由を検討したり, ある時点から発生するようになったバグの原因を追求したりするのに大変有効であると思われる.

6 おわりに

複数のバージョンにわたるプログラム開発履歴を分析するツールの試作を行なった. このツールでは複数のバージョン間の差分情報から, 過去にどのような変更が行われたのかを調べることができる. 差分情報の抽出には木の比較に基づいて差分をとる方法を用いているため, 複数のバージョンの文やブロックの間の関係を容易にたどることができ, 関数やブロックの変更について調べることも可能である.

今後の課題としては, 5節で述べたような各バージョンの保存方法, インターフェース部の工夫とともに, 保守においてさらに役立つと考えられる情報を提供する機能の拡張などが挙げられる.

参考文献

[1] Kernighan, B.W. and Pike, R. : "The UNIX Programming Environment", *Prentice Hall*, 1985.

[2] 角田 博保 : "ファイル間の相違検査法", 情報処理, Vol.24, No.4, pp.514–520 (Apr. 1983).

[3] 練 林, 井上 克郎, 鳥居 宏次 : "構文木間の対応に基づくプログラムテキスト比較ツールの試作", 信学技報, SS93–18, pp.33–99 (Jul. 1993).

[4] Roekind, M.J. : "The source code control system", IEEE Trans. on Software Engineering, Vol.SE-1, No.4, pp.364–370 (Dec. 1975).

[5] Tichy, W.F. : "RCS – A System for Version Control", Software – Practice and Experience, Vol.15, No.7, pp.637–654 (Jul. 1985).

[6] 田中 栄一, 田中 圭子 : "木の間の距離とその計算法", 信学論 (D), Vol.J65-D, No.5, pp.511–518 (May 1982).

[7] 田中 栄一 : "強構造保存写像に基づく木の間の距離とその計算法", 信学論 (D), Vol.J67-D, No.6, pp.722–723 (June 1984).