版管理とソフトウェア再利用の支援に関する研究

山本 哲男

平成14年2月

内容梗概

ハードウェアの価格の低下とその機能の向上にともない，ソフトウェアシステムは，年々大規模かつ複雑化してきており，その構築や保守のために非常に大きな労力を必要とするようになってきている．

保守作業における主な作業は，ソフトウェアの要求仕様や利用環境の変化に対する適合である．多くの環境に適合させるためには，環境毎に既存プログラムを適合させる必要があるため，プログラムの構成管理が重要となってくる．プログラムの構成管理の研究では，バージョン（改訂毎に作成されるプロダクトのこと）に基づいてプロダクトの作成や変更作業の逐次記録や追跡を行うことを目的とした様々な管理モデルが提案され，そのモデルに基づくバージョン管理システムが実装されている．従来のバージョン管理システムでは，バージョンの保存や取り出しなどを行う際には各ユーザがそれらの命令をシステムに指示する必要がある．また，バージョンの保存はユーザの恣意的な作業で行われるため，その作業を忘れてしまう，あるいは，保存間隔がユーザによって異なるといった問題点が指摘されている．

構築作業を効率よく行う一つの手法にプログラムの再利用がある．再利用では既存のソフトウェアの中から再利用可能な部品を検索し，さらに変更を加えて，部品を利用することで開発の効率化，品質保証を得る．既存の研究では，コンポーネント，ライブラリといった部品を対象にしている．しかしながら，より大規模なソフトウェアシステム単位での再利用が必要になってきている．多くの類似したソフトウェアを開発し，その派生が分からなくなると，新たなソフトウェアを開発する際，どのソフトウェアを利用し開発すればよいかを見極めるのが困難である．そのため，開発したソフトウェアがどのように派生してきたかを自動的に調べることができれば，再利用ソフトウェアを検索する作業の効率の向上が期待される．

本論文では，上述したソフトウェアの構築や保守の段階で，既存のシステムが抱える問題点を解決する方法について提案を行い，それらに基づくシステムの構築を行った．

まず，ソフトウェア開発者の作成するプロダクトの全てを自動的に記録する新たなバージョン管理ファイルシステム Moraine を提案する．Moraine を用いること

で，ソフトウェアの開発履歴を全て保存することが可能になり，過去のどの時点でのソフトウェアにも復元可能になる．また，保存作業は自動で行うため，開発者の作業に影響を及ぼさないという特徴を持つ．さらに，Moraine の実装を行い，アプリケーションのコンパイル作業を行う場合の処理時間や実際のソフトウェア開発において蓄積される保存データの量の点で評価した．その結果，十分に高速な動作が行え，バージョン管理機能を容易に利用できることが確認できた．

次に，Moraine を用いたソフトウェアメトリクス計測システム MAME(Moraine As a Metrics Environment) を提案する．MAME は，開発者の開発環境を変更することなく，メトリクスデータを収集可能である．MAME を導入して開発を行うと，開発作業後でもファイルの開発者，行数，バージョン数，日時などを指定し，ファイルに関するメトリクスデータを，細かい粒度で取得可能になる．また，MAME を学生実験に適用し，MAME から得られたメトリクスデータを用いて学生の開発動向の分析を行った．その結果，MAME を用いることで，ソフトウェアの開発履歴を詳細に分析できることが確認できた．分析結果からソフトウェアの修正，改善に反映させることで，ソフトウェア保守作業を円滑に行うことが可能になる．

最後に，ソフトウェアのバージョン間における相違の度合を調べることによって，システムの派生の様子や進化の度合を知るためのシステム SMMT（Similarity Metrics Measuring Tool）を提案する．まず，ソフトウェア間の類似の度合を表す値である類似度の一般的定義を行う．さらに，その値を計測するシステムの実装を行った．ソフトウェアのすべてのファイルの組み合わせに対して類似部分を計測するのではなく，似たファイルの組を探し出し，その組に対してより詳細な類似部分を計測することで，計測時間の向上を図っている．そして，SMMT を種々の UNIX 系 OS のソースコードに適用し，それらの類似度からクラスタ分析を行い樹状図を作成することで，OS の分類を正しく行えるか検証を行った．その結果，類似度は OS の変遷を知る有効な指標となり，得られた樹状図は OS の分類を派生通りに表していることが確認できた．要求仕様や環境の変化による修正が必要な場合，この樹状図を用いることで，どのバージョンを修正するかといった目安になり，ソフトウェア構築を効率よく行うことが可能になる．

以上，バージョン管理ファイルシステム Moraine，ソフトウェアメトリクス計測システム MAME，ソフトウェアシステム間の類似度を計測するシステム SMMT を提案し，実装した．これにより，近年のハードウェアの価格の低下を考慮したバージョン管理の手法，および類似度を用いたソフトウェア派生関係の導出に関する知見を得ることができた．

主要論文一覧

[1] 山本哲男, 松下誠, 井上克郎, “堆積型ファイルシステム Moraine とメトリクス環境 MAME への適用”, コンピュータソフトウェア, Vol.18, No.3, pp.8–18, 2001, (学術雑誌).

[2] 山本哲男, 松下誠, 神谷年洋, 井上克郎, “ソフトウェアシステムの類似度とその計測ツール SMMT”, 電子情報通信学会論文誌 D-I, [採録決定], (学術雑誌).

[3] Tetsuo Yamamoto, Makoto Matsushita, Katsuro Inoue, “Moraine: An Accumulative Software Development Environment for Software Evolution”, International Workshop on Process of Software Evolution ’99 (IWPSE’99), pp. 89–93, July 16-17, 1999, Fukuoka, Japan, (国際会議).

[4] Makoto Matsushita, Tetsuo Yamamoto, Katsuro Inoue, “An Adaptive Version-Controlled File System”, The Fourth International Symposium on Future Software Technology (ISFST-99), pp. 36–41, Oct 26–29, 1999, Nanjing, China, (国際会議).

[5] Tetsuo Yamamoto, Makoto Matsushita, Katsuro Inoue, “Accumulative Versioning File System Moraine and Its Application to Metrics Environment MAME”, The Eighth International Symoposium on Foundation of Software Engineering(FSE8), pp. 80-87, Nov 8–10, 2000, San Diego, USA.
Published as ACM SIGSOFT Software Engineering Notes, Vol.25, No.6, November 2000, (国際会議).

謝辞

本研究の全般に関し，常日頃より適切な御指導を賜わりました，大阪大学大学院基礎工学研究科情報数理系専攻 井上克郎教授に，心から深く感謝申し上げます.

本論文を執筆するにあたり，適切な御助言と御指導を頂きました，大阪大学大学院基礎工学研究科情報数理系専攻 菊野亨教授，増澤利光教授に，心から感謝致します.

大阪大学大学院基礎工学研究科情報数理系専攻在席中に，適切な御助言と御指導を頂きました，大阪大学大学院基礎工学研究科情報数理系専攻 柏原敏伸教授，藤原融教授，東野輝夫教授，今井正治教授，萩原兼一教授，村田正幸教授，谷口健一教授，宮原秀夫教授，橋本昭洋教授に感謝致します.

本論文を執筆するにあたり，直接具体的な御指導を頂きました，大阪大学大学院基礎工学研究科情報数理系専攻 楠本真二助教授，松下誠助手に心より感謝致します.

本研究を行うにあたり，御協力を頂いた，大阪大学大学院基礎工学研究科情報数理系専攻 神谷年洋（現 PRESTO）氏に感謝致します.

最後に, 井上研究室の皆様の御助言, 御協力に御礼申し上げます.

目 次

図 目 次

表 目 次

第1章　まえがき

1.1　版管理

近年のソフトウェアは大規模化しており，より複雑になってきている．また，開発形態も大人数での開発や分散化の傾向にある．このような環境での，ソフトウェア開発やソフトウェアの保守作業というのは非常に難しいものとなっている．そこで，ソフトウェアの品質や保守性を向上させるための研究が数多く行われている．しかしながら，現在のソフトウェアは新しい環境で実行する際には，その新しい環境に合わせてソフトウェアを変更しなければならない．このソフトウェアの保守にかかるコストは非常に大きなものとなっている．これに対する解決策は，新しい環境になったとしてもソフトウェア自身が柔軟に対応するようにソフトウェアを開発することである．しかしながら，我々はソフトウェア自身だけでなくソフトウェア開発環境自身も環境の進化に対応すべきであると考えている．

将来のソフトウェア開発環境は過去や現在に行った開発環境の拡張であるため，ソフトウェアの進化に対応するための一つの方法としてソフトウェア構成管理やバージョン管理を行う方法がある．ソフトウェア構成管理はソフトウェア開発過程で作成されるプロダクトの識別や制御，状態の把握等を解決する研究である [8]．このソフトウェア構成管理は開発過程の各段階で行われる．例えば，設計段階では構成要素の決定や確認，コーディング段階では生成物の変更状態の追跡，リリース段階ではリリースされる生成物の特定を，それぞれ効率良く行うことを目的として行われる．

特に，コーディング段階以降では，開発チームが作成したプロダクト (ソースコードや付随する文書) に対する様々な修正を正しく識別し，組織化し，管理することが中心となる．プロダクトは複数回の改訂が行われる．その際，各改訂毎に作成されるプロダクトはバージョンと呼ばれる．そのため，通常，プロダクトの管理はバージョン単位で行われる．このバージョンに基づいてプロダクトの作成，変更作業の逐次記録や追跡をうまく行うために，様々な管理手法のモデルが提案され，そのモデルに基づくバージョン管理システムが実装されている．

従来のバージョン管理システムでは，バージョンの保存や取り出しなどを行う際には各ユーザがそれらの命令をシステムに指示する必要がある．また，バージョ

ンの保存はユーザの恣意的な作業で行われるため，その作業を忘れてしまう，あるいは，保存間隔がユーザによって異なるといった問題点が指摘されている．

1.2　ソフトウェア再利用

ソフトウェアの構築作業を効率よく行う一つの手法にソフトウェアの再利用がある．ソフトウェア再利用は，ソフトウェアの開発にかかるコストを削減し，同時にソフトウェアの品質を向上させる．ソフトウェアの再利用を行うもっとも大きな利点は生産性の向上である．この生産性の向上は，ソフトウェアのソースコードの記述に関してではない．分析，設計，テストの各段階でもコストが削減し，生産性の向上につながる．

さらに，ソフトウェアの再利用による利点として以下の点が挙げられる．

- 信頼性，性能の向上
  ソフトウェア部品が日々多々利用されるシステム中で使われると，その部品は高い信頼性を持つように改良されていく．また，部品が広く利用されると，より高性能な部品になるように改良されていく．

- 相互利用性の支援
  異なる二つ以上のソフトウェアシステムのインタフェース部分を作成するのに同じソフトウェア部品を用いたとする．これらのソフトウェアシステムのインタフェース部分は矛盾のないように実装することができ，同様な操作で利用が可能になる．

- 標準化の支援
  再利用を行うことで，システムアーキテクチャはより統一され，開発期間が短縮される．そのため，新しいソフトウェアをより効率的に開発することが可能になる．

- ラピッドプロトタイピングの支援
  動作するソフトウェアをすばやく組み立てることができる．再利用可能部品のライブラリがソフトウェアのプロトタイプを作成するための効率的な土台を提供することになる．

そして，ソフトウェアの再利用を実際に行うためには，

1. 既存のソフトウェアの中から再利用可能な部品を検索する．
2. 検索した部品に変更を加えて，その部品を利用する．

という作業が必要になる．しかしながら，一般に既存のソフトウェアから再利用可能な部分を見つけることは困難である．また，見つかったとしても，通常そのままの形で再利用できることは少なく，何らかの変更が必要になる．他人の記述したプログラムを変更する作業は，何もないところからプログラムを記述するより大変な作業である．そこで，これらの作業を支援することが，ソフトウェアの再利用を行うに当たって必要である．

既存の研究では，ソフトウェアの部品としてコンポーネント，ライブラリといったものを対象にしている．しかし，より大規模なソフトウェアシステム単位での再利用が必要になってきている．多くの類似したソフトウェアを開発し，その派生が分からなくなると，新たなソフトウェアを開発する際，どのソフトウェアを利用し開発すればよいかを見極めるのが困難である．そのため，開発したソフトウェアがどのように派生してきたかを自動的に調べることができれば，再利用ソフトウェアを検索する作業の効率の向上が期待される．

1.3 本論文の概要

本研究では，ソフトウェアの保守段階における版管理，および構築作業におけるソフトウェア再利用を支援するための環境を確立し，その評価を行う．さらに，具体的に活用できる支援システムを開発することを目標とする．本論文では，開発した以下の三つの支援システムについて述べる．

(1) 堆積型ファイルシステム Moraine
ソフトウェア開発者の作成するプロダクト (ファイル) の全てを自動的に記録する新たなバージョン管理ファイルシステム Moraine を提案する．Moraine を，BSD UNIX 系のオペレーティングシステムである FreeBSD 3.0-RELEASE を対象にして実装し，バージョンの読み出し時間，書き込み時間，アプリケーションのコンパイル作業を行う場合の処理時間，ある一連のプログラム開発において蓄積されるバージョンデータの量の点で評価した．その結果，十分に高速な動作が行え，バージョン管理機能を容易に利用できることが確認できた．

(2) メトリクス計測システム MAME
Moraine を用いたソフトウェアメトリクス計測システムである MAME(Moraine As a Metrics Environment) を提案する．MAME はソフトウェア開発に用いることにより，開発者に負担をかけることなく，開発の途中あるいは終了後にさまざまなメトリクスデータを収集し分析することを可能にする．また，MAME を学生実験に適用し，MAME から得られたメトリクスデータを用いて学生の

活動の分析を行った．MAME を用いることでメトリクスデータの収集及び解析は容易なものとなる．そのため，ソフトウェアの開発履歴を詳細に分析でき，分析結果からソフトウェアの修正，改善に反映させることで，ソフトウェア保守作業を円滑に行うことが可能になる．

(3) 類似度計測システム SMMT

二つのソフトウェアシステムが与えられた時，そのシステムの間の違いはどれぐらいあるのか，客観的に知ることは重要である．しかしながら，二つのシステムからそれらの相違点を定量的な値として計測することは容易ではない．そこで，ソフトウェアシステム間の類似度メトリクスとそれを計測するシステム SMMT（Similarity Metrics Measuring Tool）を提案し，SMMT を種々の UNIX 系 OS のソースコードに適用した．また，それらの類似度からクラスタ分析を行い樹状図を作成し，OS の分類を正しく行えるか検証を行った．その結果，類似度は OS の変遷を知る有効な指標となり，得られた樹状図は OS の分類を派生通りに表していることを確認できた．要求仕様や環境の変化による修正が必要な場合，この樹状図を用いることで，どのバージョンを修正するかといった目安になり，ソフトウェア構築を効率よく行うことが可能になる．

以下，第 2 章では，ファイルシステム Moraine について述べる．第 3 章では，メトリクス計測システム MAME について述べる．第 4 章では，類似度計測システム SMMT について述べる．最後に第 5 章で本論文の研究についてまとめ，今後の研究の方針について述べる．

第2章　堆積型ファイルシステム Moraine

2.1　導入

現在，我々がソフトウェア開発を行う際，用いているファイル管理やバージョン管理の方法は，過去，ディスクの容量が限られていた時期に考えられ，開発されたものが主である．不要になったファイルはユーザの手によって消去され，その領域は再利用される．バージョン管理システムでは、ユーザが陽に保存の操作を行うことによって始めて保存されるバージョンが作成される．しかし，近年，ハードディスクの容量は飛躍的に大きくなり，その値段は急速に低下してきている．1995 年の平均的なハードディスク容量は約 540M であったが，2001 年には同価格のハードディスクの容量は 80GB になっている．容量を比較すると約 200 倍にもなることが分かる．このような状況では，ディスク容量が十分に大きく，不要なファイルを消す必要はない．また，バージョン管理においても，特定のファイルのみをバージョンとして登録するのではなく，一時的に作成されたものを含めて，変化する全てのファイルのバージョンを保存することができるのではないかと考える．

そこで，本章では，ソフトウェア開発者の作成するプロダクト (ファイル) の全てを自動的に採取する新たなバージョン管理ファイルシステム Moraine を提案する．また，その実装と評価を行ったのでそれについても記述する．実装の基本方針は，導入が容易であることから，バージョン管理機構をファイルシステムとして実装することである．また，既存のファイルシステムを用いた堆積型（stackable）ファイルシステムとして実装する [15]．堆積型ファイルシステムは既存のファイルシステム内部の変更は行わない．つまり，ファイルシステムの出力は HDD 等に直接行われず，下層となるファイルシステム上のファイルとして行われる．従って，ファイル構造が専用の形式に固定されるという欠点が解消される．更に，ファイルシステムが用いるバージョン管理モデルをファイルシステムから独立させることにより，複数の異なる管理ツールを使い分けられるようにする．

実装は，BSD UNIX 系のオペレーティングシステムである FreeBSD 3.0-RELEASE [16] を対象にして行った．実装したバージョン管理システム Moraine は，ファイルシステム部分とそれに付随する管理用コマンド群，バージョン管理サブシステム

によって構成される．堆積ファイルシステムの下層ファイルシステムとして UNIX ファイルシステムを利用した．システムは C 言語で記述されており，全体で 5000 行程度である．更に，Moraine をバージョンの読み出し時間，書き込み時間，アプリケーションのコンパイル作業を行う場合の処理時間，ある一連のプログラム開発において蓄積されるバージョンデータの量の点で評価した．その結果，十分に高速な動作が行え，バージョン管理機能を容易に利用でき，また，利用するバージョン管理機構を選択できるため，開発環境に応じた管理手法を適用することが可能であることが確認された．

以降，2.2 ではバージョン管理モデルで用いられている一般的概念，及び従来のバージョン管理システムとその問題点について述べ，2.3 では今回提案するバージョン管理システムの設計と実装について述べる．2.4 ではそのバージョン管理システムの評価と考察について述べ，2.5 では Moraine について考察を行い，最後に 2.6 でまとめと今後の課題について述べる．

2.2 バージョン管理システム

ソフトウェア開発過程において作成されるプロダクト (ソースコード，マニュアルや付随する文書等) の新規作成や，既に作成されたプロダクトの修正等，プロダクトに対する作業がどのように行われているかを識別し，組織化した上で管理するために，バージョン管理システムが広く用いられている．バージョン管理システムでは，プロダクトの修正作業を開発者間で調整することによって間違いのない修正を行うことを可能とする [3, 11].

本節では，バージョン管理システムにおいて用いられているバージョン管理モデルの一般的概念について説明する．また，従来の研究にて提案されている異なる実装をした代表的なバージョン管理システムである RCS と 3D Filesystem を取りあげ，バージョン管理システムの構成とその問題点についてそれぞれ説明する．

2.2.1 バージョンモデル

バージョンモデルは，管理対象となるプロダクトの各状態 (バージョン, version) を識別し，組織化した上で，プロダクトの任意のバージョンの生成，また，新規バージョンの生成方法について定義したものである．バージョン間の相互の関係はバージョン空間 (version space) として定義されるため，バージョンモデルはバージョン空間に関する定義について述べたものとなる．

バージョンと差分

バージョンモデルでは，管理対象となるプロダクト，プロダクトの持つ共通の属性，そしてプロダクトのバージョン間における差分 (delta) のそれぞれが定義される．ここでは，バージョンと差分の定義ついて述べる．

バージョン　バージョンとは，開発作業の進行と共に変化するプロダクトのある状態を指す．直感的には，ある変更による結果作成されたプロダクトそれぞれがバージョンとなる．バージョン管理されていないプロダクトは，ある特定のバージョンしか存在しないプロダクトと考えることができ，その変更は上書きによってのみ行われると考えることができる．

バージョン付けを行う場合,「何をもって同一とみなすか」は考慮されるべきである．つまり，ある 2 つのプロダクトが存在した場合，これが「全く異なる 2 つのプロダクト」か「同一のプロダクトであり，その内容が異なる」が判断されなければならない．このような同一性問題については，プロダクトに対して一意の識別子 (identifier) を与えることによって解決できる．同様に，同一プロダクトに対する各バージョンにも，そのバージョンを意味する識別子を与える．従って，あるバージョンはプロダクトの識別子とバージョンの識別子を持つ．バージョン識別子の定義は自動的に生成されるものと恣意的に与えられる物が存在する．

差分　同一のプロダクトに対するある任意の 2 つのバージョンの相異点を差分 (delta) とする．一般的に差分は (対応する英語である delta と名付けられているように) ごく小さな量でなければならない．バージョン v_1 と v_2 に対する差分は，主に 2 種類の方法によって定義される．

- 相称的差分 (symmetric delta)
  「v_1 にあり，v_2 に存在しない部分」と「v_2 にあり v_1 に存在しない部分」の和集合として定義するもの．

- 方向的差分 (directed delta)
  v_1 に対してある一連の原始的操作 $op_1, op_2, ..., op_m$ を適用した結果 v_2 が作成されたとみなし，差分を $op_1, op_2, ..., op_m$ のような操作系列として定義するもの．

実際には差分は小さくなる必要はなく，また，小さくなりえないことがある．例えば，あるソースコード中で定義された関数群のインターフェイスが常に同一であり続けることは考えにくく，また，主要な部分の変更を行う場合にはその差分は一般に大きくなると言える [10].

バージョン付け

バージョンとその差分の定義については前節で述べた．ここでは，複数のバージョンに対するバージョン識別子の定義や，バージョン自体の定義方法について述べる．

バージョン識別子　各バージョンに対して，個別に識別子を付けることにより各バージョンを識別することについては先に述べた通りであるが，その識別子の名前付けの方法には大きく分けて 2 種類の方法がある．これらの名前付けは互いに相反するものではなく，両方を同時に用いることも可能である．

- 数え上げ可能な識別子
  これは，バージョン識別子として，ある事前に定められた一連の識別子を用いる方法である．この定義では一般的に数字や数字の組み合せ (1.1 など) が識別子として用いられる．このような定義では，過去の任意の時点におけるバージョンを簡単に識別することが可能になる．

- 述語によって導出される識別子
  これは，ある述語 $c()$ を定義し，その述語を真にするような識別子を用いる方法である．この定義では一般的に文字列を識別子として用いられる．任意の識別子名を自由に定義し用いることが可能になる．

バージョンの定義　すでに，各バージョンはプロダクトに対するある状態を指すものとして定義されることを述べた．ここではバージョンの定義に必要となる「プロダクトの状態」の定義について述べる．

プロダクトの状態を考える場合には，大きく 2 種類の方法がある．

- プロダクトそれ自体で定義するもの
  例えばファイル等の場合，ファイルの中身すべてをそのまま状態として考えることができる．この定義の場合，状態の変更内容とプロダクトの持つ内容の変更は一致したものとなる．

- 変更内容の集合を用いて定義するもの
  プロダクトのある状態を「ある基準となる状態」と「基準となる状態からの変更内容」の集合として捉える方法がある．変更内容はある特定の識別子によって識別され，プロダクトの状態は変更内容を表わす識別子の集合として定義される．

2.2.2 従来のシステム

ここでは，従来提案されている代表的なバージョン管理システムである RCS と 3D Filesystem について，どのようなバージョン管理モデルが用いられ，またどのような実装が行われているかについて説明する．さらに，既存のシステムが抱える問題点について考察を行う．

RCS

RCS は UNIX 上の動作するツール群として開発されたバージョン管理システムであり，現在広く用いられている物である [44]．RCS ではプロダクトをそれぞれ UNIX 上のファイルとして扱い，1 つのファイルに対する記録は別の 1 つのファイルとして扱われる．

RCS におけるバージョンは，管理対象となるファイルの中身それ自身によって定義され，バージョン間の差分は UNIX における diff コマンドの出力 (編集コマンドの列として表わされる) として定義されている．バージョンに対する識別子は数字の組で表現され，数え上げ可能な識別子となっている．新規バージョンの登録や，任意のバージョンの取り出し等は，RCS の持つツールを利用することによって行う．

しかし，RCS はツールとして実装されているため，RCS を利用する場合には RCS の持つさまざまなツールの利用方法を学ばなければならない．また，新規バージョンの定義はツールを明示的に起動することによって行うため，バージョン間の差分が大きくなりすぎる可能性がある．また，誤った操作を行うと，記録されたバージョンが破壊されてしまう危険性がある．

3D Filesystem

3D Filesystem は UNIX SystemV Release 3 のカーネルを変更することによって，ファイルシステム上に実装されたバージョン管理システムである [20]．3D Filesystem では RCS と同様，ファイルシステム上のファイルをプロダクトとして扱っている．

3D Filesystem におけるバージョンモデルは RCS と良く似た物となっている．つまり，ファイルの中身自身を用いてプロダクトの状態を定義し，バージョン識別子は数字の組みで表される数え上げ可能な識別子となっている．ファイルシステムにバージョン管理システムを組み込むことにより，別途ツールを用いることなく任意のバージョンを取り出すことが可能になっている．新規バージョン等の登録については，3D Filesystem と連携して動作するツール群によって行う．

3D Filesystem は RCS の持つ欠点を補った物となっているが，ファイル構造が 3D Filesystem 固有の構造となっているため，保存された情報は 3D Filesystem で

しか用いることができない．また，バージョン管理モデルは 3D Filesystem が定義する物しか用いることができないため，利用する際には制約が大きいという問題がある．また，3D Filesystem は現在研究段階のシステムであり，実用性に欠けるといった点も上げられる．

2.3 Moraine

バージョン管理ファイルシステム Moraine はファイルシステムにバージョン管理機能を持たせたものである．バージョン管理モデルには checkin/checkout モデルを採用した [13]．また，ファイルシステムとしての機能とバージョン管理を行う機能を，それぞれカーネルの内部とユーザプログラムに分けることでより柔軟なバージョン管理を行うことができる．

2.3.1 設計方針

近年，ディスクの容量は大容量化の一途を辿っており，CPU の性能は急速に向上している．開発管理，プロダクトの品質の向上など，ソフトウェア開発におけるさまざまな観点を考えた場合，ソフトウェア開発環境上で行われた開発者全員の作業は記録するべきであると我々は考えている．また，記録した作業履歴は容易に取得可能でなければならない．

このような開発環境では，作成されるすべてのファイルのすべてのバージョンが記録される．また，開発者はバージョンの保存について何も行う必要がない．ファイルが必要無ければ消去することも通常の操作で行うことができ，消去されたファイルもタイムスタンプやユーザによって指定されたタグによって後から取得できる．

これらの考えを元に Moraine の設計方針を以下に示す．

- 容易な操作
  ユーザは Moraine の操作を事前に学ぶ必要がない．Moraine は通常のファイル操作を自動的に監視し，開発者の特別な操作を必要としない．

- オープンな構造
  Moraine はデータリポジトリやバージョンの管理に独自の構造を持たない．そのため，容易に多くのシステムに移植が可能である．

以上の設計方針に基づいて UNIX のファイルシステム上にバージョン管理機能を組み込んだシステムを設計した．この手法では，通常のファイルに対する読み出し (read システムコール)，書き込み (write システムコール) 動作を直接バージョ

ン管理作業に対応付ける．また，既存のファイルシステムを用いたスタッカブルファイルシステム (Stackable File System)[15] として実装しており，既存の環境に容易に導入可能である．

2.3.2 アーキテクチャ

Moraine はファイルシステム部分とそれに付随する管理用コマンド群，バージョン管理デーモン，バージョン管理サブシステムによって構成される．図 2.1 は各構成要素のつながりを表したものである．各構成要素は以下のような動作を行う．

- ファイルシステム (VFS)
  ファイルシステムとしての機能をもつ部分であり，カーネル内のプログラムである．このファイルシステムに対してユーザプロセスからファイルの読み出し，書き込み，ディレクトリの作成等といった一般のファイルシステムの持つ機能を行うだけで，バージョン管理機能が動作する．

- バージョン管理デーモン (VCD)
  カーネル内のファイルシステムとのやりとりを行うデーモンプログラムである．カーネルから依頼されるファイルのバージョン管理を引き受ける．ファイルに対する実際のバージョン管理は，バージョン管理サブシステムに依頼する．

- バージョン管理サブシステム
  実際にバージョンを記録する部分である．バージョン管理サブシステムは複数のバージョン管理ツールの集合体である．バージョンの記録を行う場合，いずれかが選択されてそのツールを実行する．

- 管理用コマンド群
  過去のバージョンの参照やバージョン間の差分情報の取り出しなどの通常のファイル操作だけではできない操作を補助するコマンド群である．

バージョン管理モデルには RCS などで用いられている checkin/checkout モデルを採用した．これは，バージョン管理を特定のツールに依存するのではなく checkin/checkout モデルに基づいたツールであればそれが使用可能な事を意味する．今回の実装においては，RCS と VCS という二種類の管理方法を提供する．

あるディレクトリ以下をバージョン管理ファイルシステムでマウントすると，マウント元にはバージョンを記録したファイルと最新バージョンが存在し，マウント先には最新バージョンのみが見えるようになる．

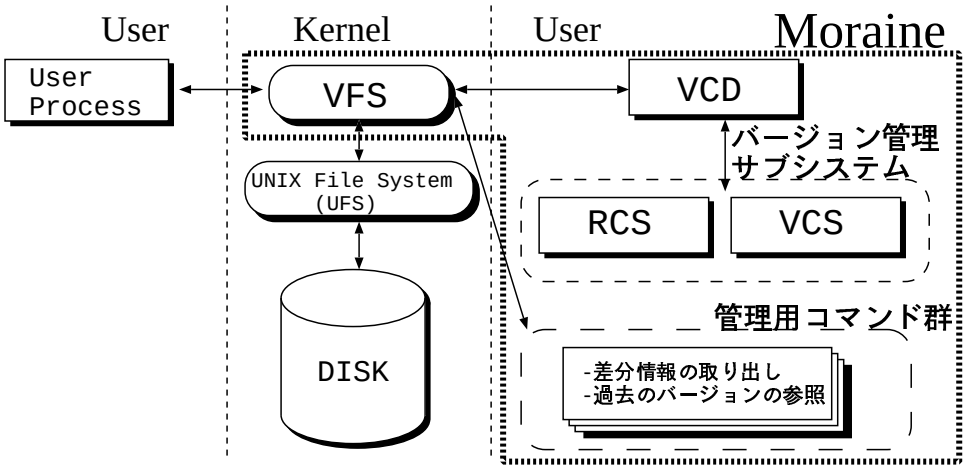

図 2.1: Moraine のアーキテクチャ

実際のファイル操作は，ファイルを操作するユーザプロセスにとっては一般のファイルシステム上のファイルを操作するのと同様の操作で可能である．

常に最新バージョンのファイルをファイルシステム上に checkout しておく．そのため，ファイルの読み書きは高速に動作する．ファイルを読み出し専用で開いた場合は，従来のファイルシステム上と同じ動作をする．それに対して，書き込みフラグを付けてファイルを開いた場合，読み出し，書き込みといった操作は従来と同じであるがファイルに対して閉じる動作 (close システムコール) を行ったとき，そのファイルに対して checkin 動作を行う．この checkin の動作はファイルシステムの中には定義しておらず，カーネルの外のバージョン管理デーモンとバージョン管理サブシステムが行う．

カーネル内に記述しない利点として，checkin/checkout の動作をカーネルを変更せずに切替ができる事があげられる．また，RCS 等のプログラムを使ってバージョン管理を行っていた場合，RCS で使われている機能をバージョン管理サブシステムのプログラムとして記述することで，今回のファイルシステムにおける実現においても以前のデータがそのまま使えるようになる．checkin/checkout 用のプログラムを，独自の形式で書くことも可能である．そのため，必要な情報を付加したバージョンを保存するようなプログラムを作成することで，RCS などの既存のプログラムでは取得できない情報を記録しておくことができる．

2.3.3 機能

Moraine では，通常のファイルに対する読み出し，書き込み動作を直接バージョン管理作業に対応付ける．既存のエディタ等によるファイルの読み書き作業だけでバージョン管理機能が利用できる．通常ファイルのみバージョンを取り，ディレクトリ，シンボリックリンク，特殊デバイスファイル，ソケット，名前付きパイプなどは取らない．バージョンを取るのは，新規にファイルを作成した場合や，既存のファイルを変更した場合であり，それらを新規バージョンの登録作業として扱う．ファイルを読み出す場合，最新のバージョンを自動的に取り出してそれを読める．

各ファイルは最新バージョンのファイルと過去のバージョンを記録したファイルからなる（図 2.2）．実ファイルシステムには最新のバージョンのファイルと過去のバージョンを記録したファイル共に存在するが，Moraine の VFS 上では最新のファイルのみが存在しているように見える．そのため，通常，Moraine では最新バージョンのファイルのみ操作可能であり，過去のバージョンは直接参照することが不可能である．例として，Moraine で */versiondb* 以下に存在するすべてのファイルやディレクトリを */proj* からも読み書きできるように接続し，*/proj/foo* というファイルを作成した場合，Moraine は *foo* (*/proj/foo* 自身) の最新バージョンのファイルとして *“/versiondb/foo,a”* を作成し，過去のバージョン記録ファイルとして *“/versiondb/foo,v”* を作成する．*/proj/foo* は */versiondb/foo,a* というファイルを指すことになる．また，*/versiondb/foo,v* は */proj* の下では参照できない．ユーザは */proj* の下のファイルのみを操作することになる．

Moraine は常に最新バージョンのファイルをファイルシステム上に用意しているため (上記の例では*/versiondb/foo,a*)，ファイルの読み出しはそのファイルを単に読み出すだけであるので高速に動作する．

通常，バージョン履歴ファイルはファイルシステムからは操作を行わないが，ファイル名変更 (rename システムコール)，削除 (remove システムコール) についてはバージョン履歴ファイルに対してもその操作を実行する．

2.3.4 堆積型ファイルシステム

ファイルシステムには直接物理装置を操作する UFS[30] や LFS[7] といったものや，ネットワークを用いた NFS といったものがある．Moraine で用いられるファイルシステムは，既存のファイルシステムの上に構築する堆積型ファイルシステム [15] として設計した．

堆積型ファイルシステムは vnode[17] と呼ばれるデータ構造を用いている．この

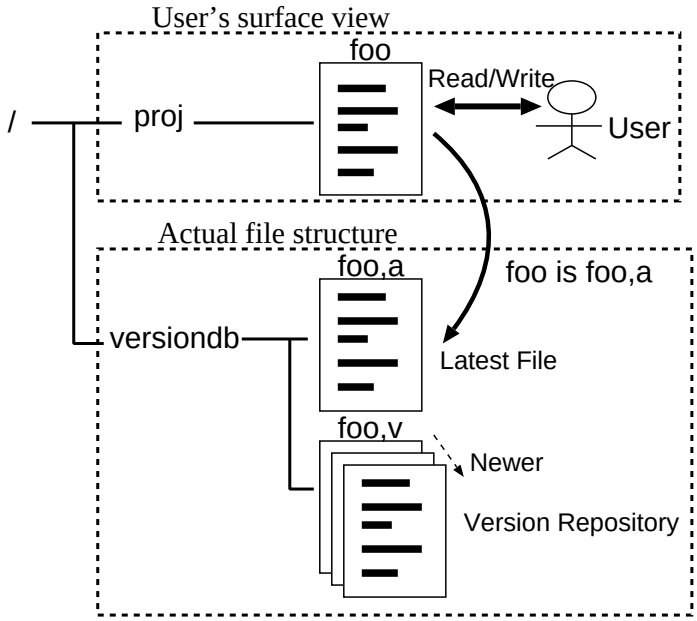

図 2.2: VFS と実ファイルシステムの関係

構造は近年の UNIX オペレーティングシステムでファイルやディレクトリなどを表現するのに使用されている．

堆積型ファイルシステムの特徴としてつぎのようなものが挙げられる．

- 堆積型ファイルシステムは既存のファイルシステム内部の変更は行わない．
- ファイルシステムの出力は HDD 等に直接行われず，下層となるファイルシステム上のファイルとして行われる．
- 記憶装置やネットワークを処理するコードを書く必要がなく，移植が容易である．

図 2.3 に堆積型ファイルシステムの構造を示す．ユーザプロセスからのファイルに対するシステムコールはすべて vnode 層の呼出しに変換される．そして，その呼び出しはファイルの存在するファイルシステムの関数を実行する．もし，ファイルが堆積型ファイルシステム上にあれば，堆積型ファイルシステムの各操作を呼び出す．堆積型ファイルシステム層の各操作は，下層のファイルシステムの対応する各操作を呼び出す．

以下に堆積型ファイルシステムの例を挙げる．

- NULLFS
  処理を直接下層に渡し，それ自身は何も行わないファイルシステムである．

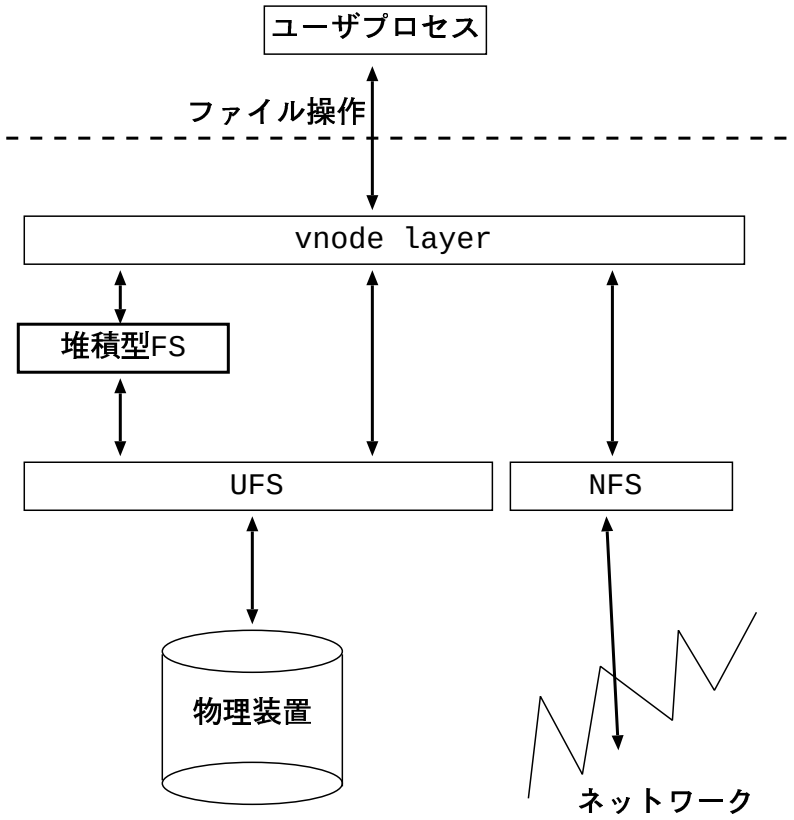

図 2.3: 堆積型ファイルシステム

マウントを行うと，マウント元のディレクトリ以下がマウント先のディレクトリ以下ですべての操作可能になる．NULLFS では下層へ渡すときに何も処理を行わないので，どちらのディレクトリも同一に見える．

- UMAPFS
  uid と gid のみを変更して処理を直接下層に渡すファイルシステムである．ファイルシステムのマウント時に uid と gid の対応ファイルを指定する．たとえば，下層ファイルシステムにおいて uid 1000 のファイルがあるとする．対応ファイルに 1000 から 2000 と記述する．そのファイルを指定してマウントすると，uid 1000 のファイルは、実際には uid 2000 が所有者であるかのように見える．

- UNIONFS[37]
  複数のディレクトリを重ねあわせ，同一のディレクトリとして見せることができるファイルシステムである．マウントを実行すると，マウント元のディレクトリ以下がマウント先のディレクトリ以下に重ね合わせる．つまり，あ

るファイルを指定した際にマウント先にファイルが存在すればそのファイルを，マウント先に存在せずにマウント元に存在すればそのファイルを返す処理を行う．

2.3.5 バージョン管理サブシステム

Moraine では，バージョン管理サブシステムとして，既存のバージョン管理システムを用いており，ファイルシステムを利用する際に，どのシステムを利用するかが選択できる．今回の実装では RCS と VCS の 2 種類のバージョン管理サブシステムが利用可能である．

RCS　RCS は各バージョンの差分のみを保存するために広く用いられているツールである．RCS ではバージョンの派生 (木構造) を許す．この派生された枝をブランチと呼ぶ．RCS のバージョンの木構造の例を図 2.4 に示す．

このサブシステムではバージョン記録ファイルに対する操作はすべて RCS のコマンドを内部で実行することでバージョンを記録する．

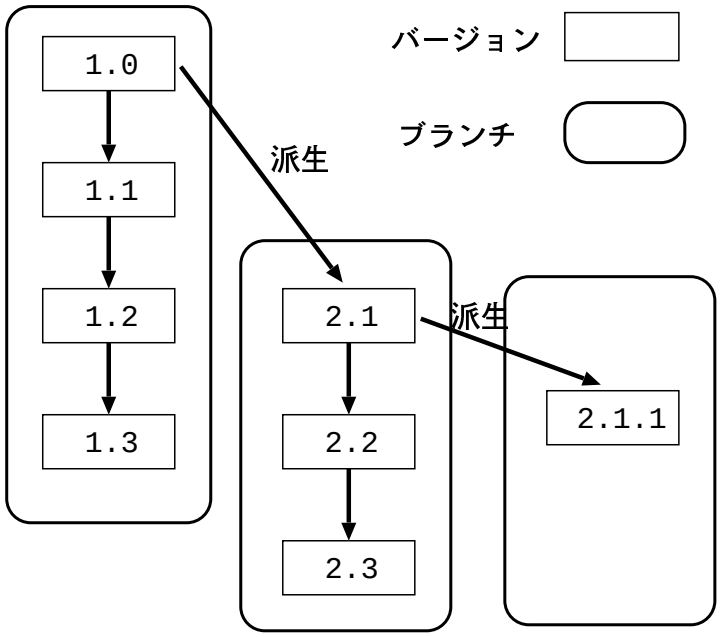

図 2.4: バージョンの木構造

VCS　今回作成した単純な機能を持つバージョン管理システム．各バージョンを圧縮しない形でバージョン間の差分もとらずにそのまま保存する．バージョンの

派生はなく，線型である．RCS と違いバージョン間の差分情報をとらないことで高速に動作させることを目的としている．

VCS のバージョン履歴ファイルのファイル形式を図 2.5 にファイルのヘッダを表 2.1 にバージョンヘッダを表 2.2 に示す．checkin 動作は登録するバージョンをバージョンヘッダと共にバージョン履歴ファイルに追加していく．ただし，ヘッダのバージョン番号と最新オフセットは更新しておく．過去のバージョンはヘッダの最新オフセットと各バージョンファイルのバージョンヘッダの前オフセットを辿ることで取得可能になる．

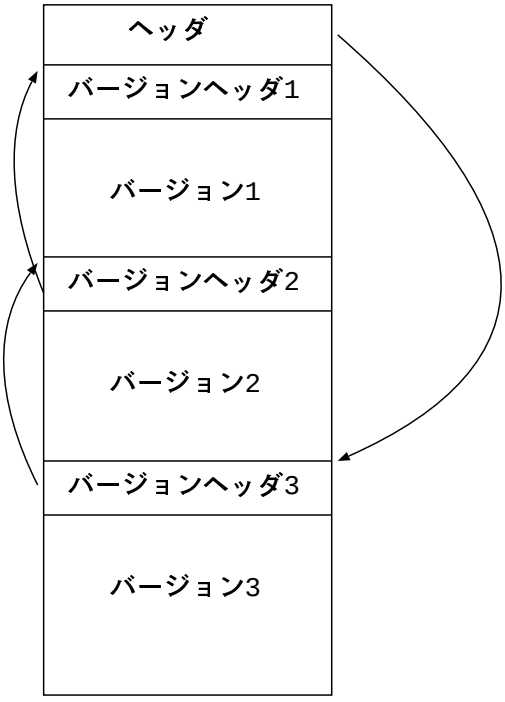

図 2.5: VCS のファイル形式

表 2.1: ヘッダ構造

名前	サイズ	内容
バージョン番号	4 バイト	最新バージョン番号を表す．初期バージョンを 1 として以降 2，3，…と続く．
最新オフセット	16 バイト	最新バージョンのヘッダのオフセットを表す．

表 2.2: バージョンヘッダ構造

名前	サイズ	内容
属性	vattr 構造体のサイズ	ファイルの属性を表す vattr 構造体.
前オフセット	16 バイト	前バージョンのヘッダのオフセットを表す.

2.3.6 管理コマンド群

通常のファイル操作では行えない操作をするためのコマンド群である.
コマンドとしては以下のものがある.

- 任意のバージョンファイルの取り出し
  リビジョン番号や日付などを指定し，過去のバージョンファイルを取り出すコマンドである.
- ブランチ作成
  RCS などバージョンの派生を許すモデルの場合に，派生を行うコマンドである．VCS など許さない場合はこのコマンドを実行しても何も起こらない.
- 差分の閲覧
  バージョン間の差分を取り出すコマンドである.

2.3.7 実装

BSD UNIX[22, 29] 系のオペレーティングシステムである FreeBSD 3.0-RELEASE [16] を対象にして実装を行った.

実装する機能 堆積型ファイルシステムの下層ファイルシステムとして UNIX ファイルシステム (UFS) を利用する.
以下の関数をファイルシステムとしての機能の実装をカーネルに追加した.

- vc_lookup
  ファイル名からカーネル内部で必要なデータ構造に変換する関数である．ファイル名の変換を行い下層ファイルシステムに処理を渡す.
- vc_access
  ファイルを操作する時に，操作可能かどうか判断する関数である．読み出し

専用でマウントされた時に，書き込みを行おうとした場合はエラーを返す．それ以外の場合は下層ファイルシステムに処理を渡す．

- vc_setattr
  ファイルに関する属性の設定を行う関数である．設定する値が無効であったり読み出し専用でマウントされた時にはエラーを返す．それ以外の場合は下層ファイルシステムに処理を渡す．

- vc_close
  close システムコールに対応する関数である．ファイルが書き込みフラグ付きで開かれていた場合 checkin 動作を行い，下層ファイルシステムに処理を渡す．

- vc_getattr
  ファイルに関する属性の取得を行う関数である．下層ファイルシステムに処理を渡した後，ファイルシステムの id をマウント先に変更する．

- vc_read
  read システムコールに対応し，ファイルからの読み出しを行う関数である．そのまま，下層ファイルシステムに処理を渡す．

- vc_write
  write システムコールに対応し，ファイルからの書き込みを行う関数である．そのまま，下層ファイルシステムに処理を渡す．

- vc_mkdir
  mkdir システムコールに対応し，ディレクトリを新しく作成する関数である．ファイル名の変換を行い下層ファイルシステムに処理を渡す．

- vc_rmdir
  rmdir システムコールに対応し，空のディレクトリを削除する関数である．そのまま，下層ファイルシステムに処理を渡す．

- vc_create
  create システムコールに対応し，新しくファイルを作成する関数である．ファイル名の変換を行い下層ファイルシステムに処理を渡す．

- vc_remove
  unlink システムコールに対応し，ファイルをディレクトリエントリから削除

する関数である．下層ファイルシステムに処理を渡した後，バージョン履歴ファイルも削除する．

- vc_rename
  rename システムコールに対応し，ファイル名の変更を行う関数である．ファイル名の変換を行い下層ファイルシステムに処理を渡す．その後，バージョン履歴ファイルのファイル名も変更する．

- vc_symlink
  symlink システムコールに対応し，シンボリックリンクファイルの作成を行う関数である．ファイル名の変換を行い下層ファイルシステムに処理を渡す．

- vc_readlink
  readlink システムコールに対応し，シンボリックリンクファイルに格納されているデータを読み出す関数である．ファイル名の変換を行い下層ファイルシステムに処理を渡す．

- vc_readdir
  getdents システムコールに対応し，ディレクトリからデータを読み出す関数である．下層ファイルシステムに処理を渡した後，読み出したデータのうちファイル名は変換を行い返す．

ファイルシステムとしての機能の他に以下の機能を実装した．

- システムコール
  バージョン管理デーモンとファイルシステムとのインタフェースとなる．バージョン管理デーモンはこのシステムコールを用い，新規バージョンとして登録するファイルがあるかを調べる．

- checkin 依頼機能
  ファイルを書き込みフラグを付けて open していた場合，そのファイルの close 時に呼び出される．ファイル名を調べ，そのファイルを新規バージョンとして登録するようにバージョン管理デーモンに依頼する．

- ファイル名変換機能
  マウント元のファイル名とマウント先のファイル名の相互変換を行う．

- vnode からパス名変換機能
  vnode からファイルのフルパス名の変換．

- 仮想デバイス
  通常のファイル操作だけではできない操作を行うためのものである．管理用コマンド群はこの仮想デバイスを利用する．

コードサイズ　実装はすべて C 言語で記述し，全体で 5000 行程度になった．それらの内訳はつぎのようになる．

- 堆積ファイルシステム (カーネルへの追加) 4500 行
- バージョン管理デーモン (VCD) (新規作成) 340 行
- その他 (バージョン管理サブシステムとのインタフェース等) 300 行

2.4　実験

本節では，システムの性能とファイルシステムに必要な容量の点から Moraine の評価を行う．システムを導入しても，システムの動作が遅く開発に支障を来してしまう場合には実際の開発環境に導入することは難しい．そこで，ファイルの読み書きに関してファイルシステムの性能評価を行った．評価の方法としては，ファイルシステムとして通常用いられる UNIX ファイルシステム (UFS) と堆積型ファイルシステムの一つである NULLFS と比較することで行った．

ファイルシステムの評価として 2 種類のテストを行った．

- 様々な大きさのファイルの読み出し，書き込みのようなファイルシステムの性能に関するもの．
- ファイルシステム上に蓄えられるファイルの大きさに関するもの．

以下で述べるテストはすべて Pentium 166MHz・48MB RAM の計算機で評価した．

2.4.1　ファイルの読み出しテスト

一般的なソフトウェア開発におけるファイルのサイズを考え，1KB から 16MB までの間の大きさのファイルの読み出しテストを行った．それぞれのファイルシステム上に各大きさのファイルを用意し，read システムコールを用いファイルをメモリ上に全て読み出すプログラムの実行時間の測定を行った．

測定結果を表 2.3 と図 2.6 に示す．Moraine の読み出しではバージョン管理サブシステムを使用しないので，RCS，VCS どちらのバージョン管理サブシステムを採用していても測定結果は同じ値になる．

表 2.3: ファイルの大きさの違いによる読み出し時間 (Sec)

size(byte)	1024	2048	4096	8192	16384	32768	65536	131072
Moraine	0.05	0.02	0.05	0.04	0.06	0.05	0.08	0.2
NULLFS	0.07	0.04	0.03	0.04	0.02	0.06	0.09	0.2
UFS	0.05	0.05	0.03	0.05	0.04	0.02	0.08	0.1
size(byte)	262144	524288	1048576	2097152	4194304	8388608	16777216	
Moraine	0.22	0.37	0.77	1.29	2.52	5.45	12.91	
NULLFS	0.34	0.6	0.68	1.38	3.31	5.22	11.23	
UFS	0.13	0.35	0.82	1.44	2.31	4.63	9.41	

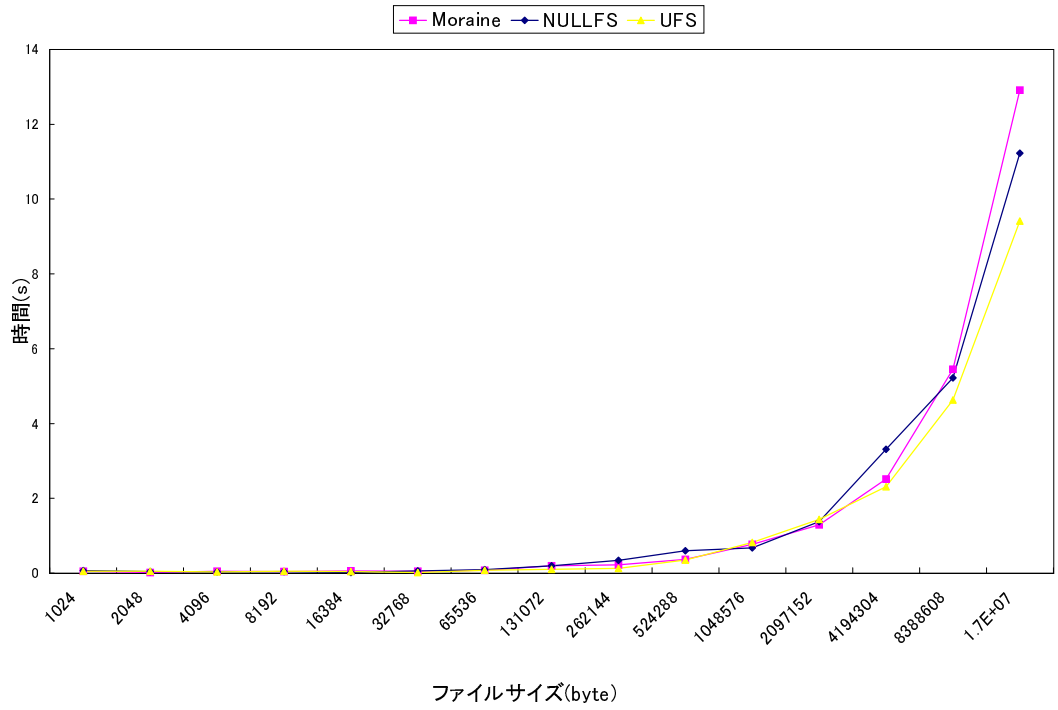

図 2.6: ファイルの大きさの違いによる読み出し時間

表 2.4: 1MB の連続読み出し時間 (Sec)

	1	2	3	4	5	6	7	8	9	10
Moraine	0.84	1.41	2.53	3.78	4.9	5.29	6.2	6.6	6.96	8.9
NULLFS	0.92	1.4	2.47	3.74	4.41	5.06	5.93	6.73	6.75	8.49
UFS	0.77	1.44	2.36	3.03	4.54	4.62	5.78	5.93	6.21	7.3

これらの結果から，いずれの大きさのファイルに対しても各ファイルシステムでほとんど差が見られないことが分かる．ファイルの大きさによって各ファイルシステムで読み出し時間の差はほとんど変化はなく，性能が著しく劣ることはない．最新バージョンのファイルは常にファイルシステム上に存在しており，ファイルの読み出しに関してはそのバージョンのファイルを読み出すだけなので，読み出しのみの場合はバージョン管理サブシステムは関与しないためである．

次に，ファイルの大きさが 1MB の異なるファイルを，単一プロセスで繰り返し連続して読み出す時の処理時間を計測した．ファイルの大きさが小さい場合，処理時間は小さくシステムの動作に支障を来す程度ではない．そのため，ファイルの大きさが小さくなく，一般的な大きさとして 1MB という大きさを選んだ．ここで，ファイルの内容は処理時間に影響を与えない．読み出し実験は回数を変化させて各回数毎に処理時間を測定した．複数回読み出しを行う時は，同一ファイルを指定された回数読み出すのではなく，異なるファイルを読み出す．そのため，ファイルの個数を読み出す回数分だけあらかじめ用意しておく．ここで，処理時間とは連続した読み出しを行うプロセスの実行時間である．つまり，プロセスの生成から消滅までの時間となる．測定した結果を表 2.4 と図 2.7 に示す．縦軸は実行時間を示し，横軸は読み出した回数を示す．1 回だけの読み出しから 10 回連続までの 10 個の値が各ファイルシステムについて存在する．Moraine(RCS) と Moraine(VCS) はそれぞれバージョン管理サブシステムとして RCS と VCS を用いたという事を意味する．以降，Moraine(RCS)，Moraine(VCS) は同様の意味で使う．

どの回数においても Moraine(RCS) と NULLFS はほとんど違いが見られない．これによりファイルの読み出しに関してはファイルシステム内のバージョン管理機能はほとんど時間がかからないことが分かる．Moraine(RCS) や NULLFS は UFS と比べると約 10%の時間を要するがこれは堆積型ファイルシステムとして実装したためであり，堆積型ファイルシステム部分の機能が 10%程度であるといえる．一般のファイルシステムと比べても十分に高速である．

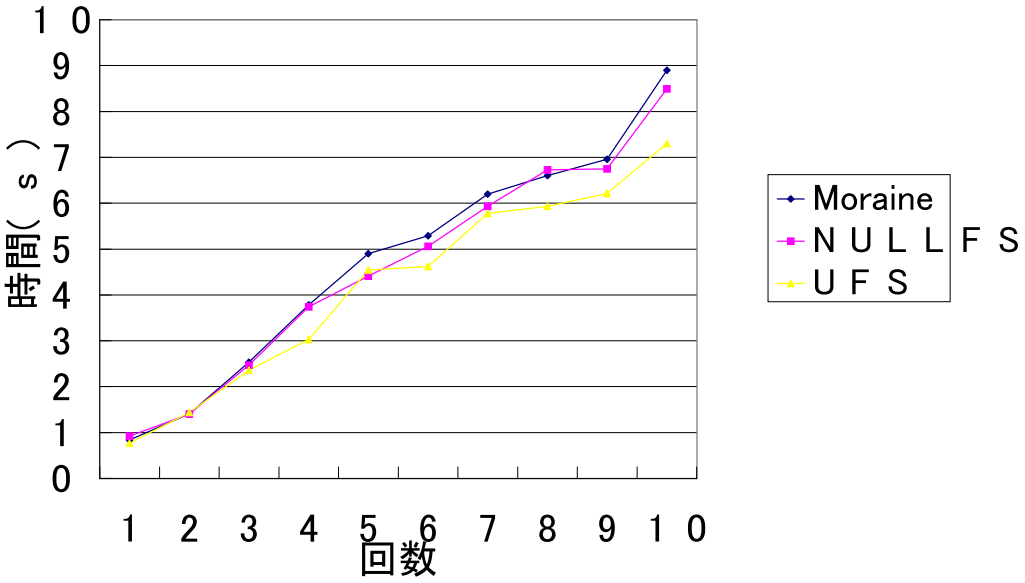

図 2.7: 1MB の連続読み出し時間

2.4.2 ファイルの書き込みテスト

同様のテストを，書き込みに関しても行った．ファイルの大きさを 1MB とし，異なるファイル名で単一プロセスで繰り返し連続して書き込む時の処理時間の計測を行った．ここで処理時間とは連続した書き込みを行うプロセスの実行時間である．書き込むファイルはファイルシステム上に存在していないものとする．

測定結果を表 2.5 と図 2.8 に示す．1 回だけの書き込みから 10 回連続までの 10 個の値が各ファイルシステムについて存在する．図 2.8 の上のグラフはすべての結果を表し，下のグラフは Moraine(RCS)，NULLFS，UFS だけの結果を表した図である．Moraine(VCS+) はバージョン管理サブシステムとして VCS を用い，完全にバージョンを記録するまでの時間を計測したものである．VCD とバージョン管理システムとの間で同期をとり，完全にバージョンの記録が完了するまでの時間を計測したものである．Moraine(VCS) の場合は VCS の終了を待たずに計測した時間を表している．Moraine(RCS+) についても同様に RCS の完全に記録が終了するまで計測した時間を表す．

UFS と比べ，NULLFS は 10%程度の時間が遅い．これは読み出しの際の UFS と NULLFS との違いと同じ割合である．何も行わない堆積型ファイルシステムでは

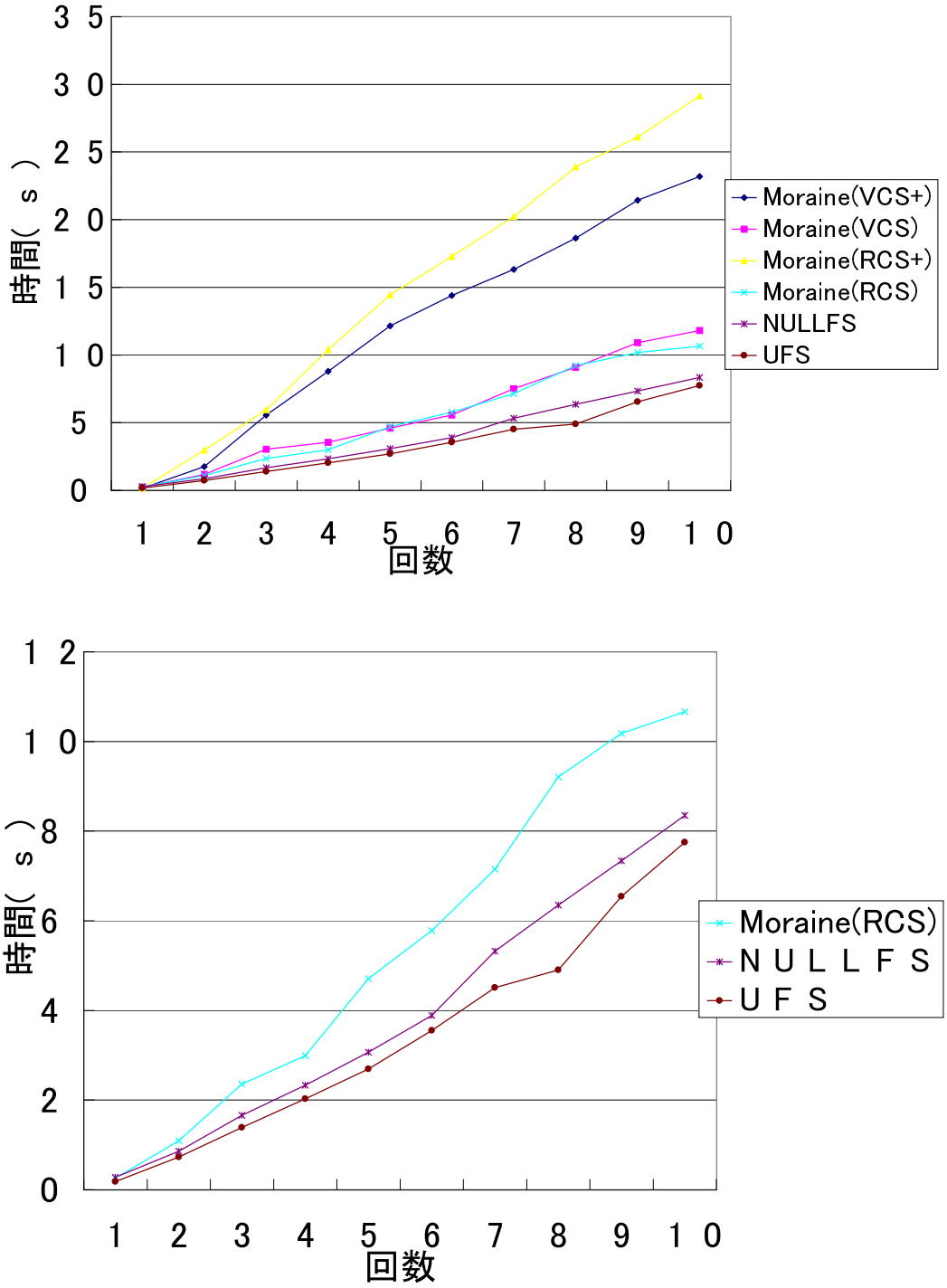

図 2.8: 1MB の連続書き込み時間

表 2.5: 1MB の連続書き込み時間 (Sec)

	1	2	3	4	5	6	7	8	9	10
Moraine(VCS+)	0.16	1.75	5.57	8.79	12.16	14.4	16.33	18.63	21.44	23.2
Moraine(RCS+)	0.15	2.98	5.95	10.39	14.45	17.29	20.24	23.9	26.11	29.13
Moraine(VCS)	0.24	1.18	3.04	3.55	4.59	5.57	7.5	9.1	10.91	11.8
Moraine(RCS)	0.26	1.09	2.36	2.99	4.71	5.78	7.15	9.21	10.18	10.66
NULLFS	0.28	0.86	1.66	2.33	3.07	3.89	5.32	6.35	7.34	8.35
UFS	0.18	0.73	1.39	2.03	2.69	3.55	4.51	4.9	6.54	7.75

表 2.6: 32KB の連続書き込み時間 (Sec)

	1	2	3	4	5	6	7	8	9	10
Moraine(VCS+)	0.02	0.13	0.23	0.32	0.46	0.52	0.61	0.73	0.79	0.88
Moraine(VCS)	0.02	0.06	0.15	0.16	0.22	0.29	0.36	0.38	0.48	0.57
Moraine(RCS+)	0.02	0.25	0.51	0.75	1	1.39	1.5	1.71	2.04	2.78
Moraine(RCS)	0.02	0.06	0.11	0.16	0.19	0.25	0.37	0.44	0.54	0.69
NULLFS	0.02	0.07	0.09	0.14	0.16	0.26	0.23	0.26	0.27	0.32
UFS	0.03	0.03	0.03	0.03	0.02	0.02	0.02	0.03	0.02	0.01

読み書き共に 10%程度の時間を余計に必要となり，読み書き共に十分高速である．Moraine(RCS) と UFS を比べると約 30%Moraine(RCS) のほうが遅い．読み込み時には 10%であったが，書き込みの際にはバージョン管理を行う分の時間を要するためである．しかし，バージョンの記録が完全に終了する時間は倍以上の時間を必要とする．今回の実装ではバージョンの記録はカーネル内では行わず，ユーザプロセスとして実行させるため，30%程度の時間の遅れで書き込みを行うプロセスは終了し，バックグランドでバージョン管理サブシステムが動作しバージョンの記録を行う．バージョンを記録する機能をカーネル内部から外したことで高速性をもつファイルシステムが実現できた．このため，ファイルを書き込む利用者に対する応答時間は，Moraine(VCS+) ではなく Moraine(VCS) の時間になる．Moraine(VCS) の時間は，利用者に対してファイルの保存に関する動作の支障をきたすことはないと考える．

さらに，ファイルの大きさが 32KB にして書き込みテストを行った．測定結果を表 2.6 と図 2.9 に示す．

基本的には上記の 1MB ファイルの書き込みと同じ傾向が見られるが，ファイルの大きさが小さいときには，バージョン管理サブシステムとして RCS より VCS を用いたほうが高速に動作する．

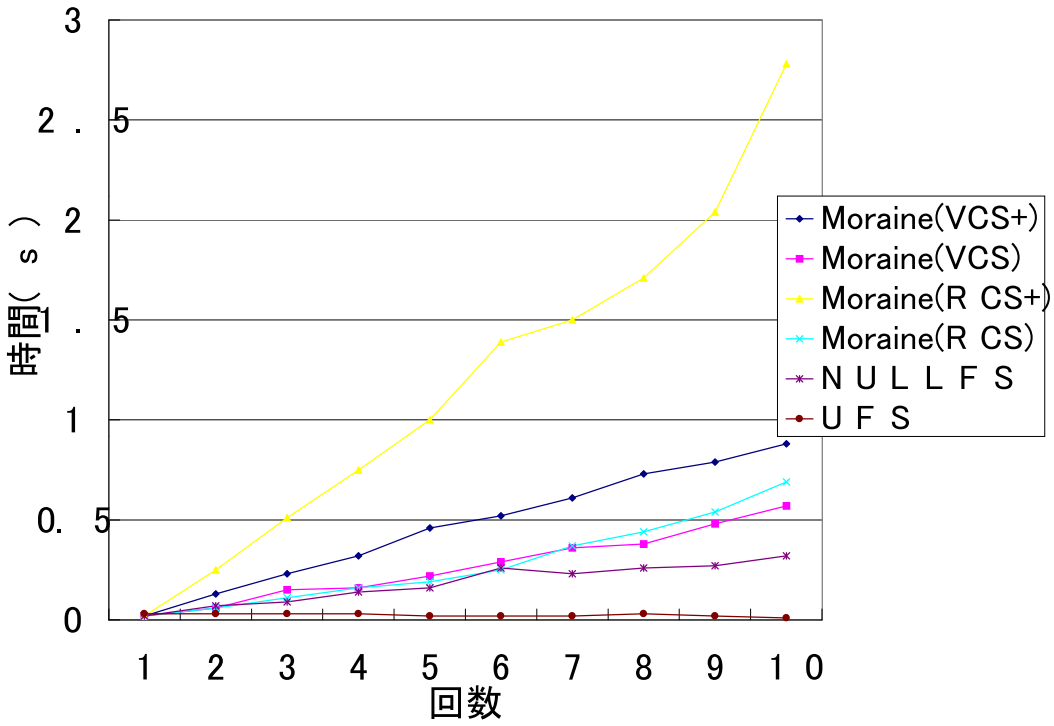

図 2.9: 32KB の連続書き込み時間

2.4.3 バージョン記録テスト

ファイルの更新時の時間について測定を行った．ファイルの大きさが 1MB のファイルを用意し，各ファイルは RCS で差分が大きくなるように値をそれぞれ 0 から 5 の値で埋めたものにしておく．更新の値が 0 というのは新規にファイルを作成したという事である．測定結果を表 2.7 と図 2.10 に示す．

表 2.7: バージョンの更新による時間 (Sec)

	0	1	2	3	4
Moraine(VCS+)	0.75	0.85	0.86	0.87	0.86
Moraine(VCS)	0.18	0.11	0.11	0.11	0.11
Moraine(RCS+)	1.91	5.88	7.55	8.6	10.21
Moraine(RCS)	0.2	0.12	0.13	0.11	0.11
NULLFS	0.33	0.21	0.21	0.21	0.21
UFS	0.24	0.12	0.12	0.12	0.12

バージョン管理サブシステムとして RCS を用いた場合を除いて更新したときに要する時間は新規にファイルを作成した時間と同じである．バージョン管理サブ

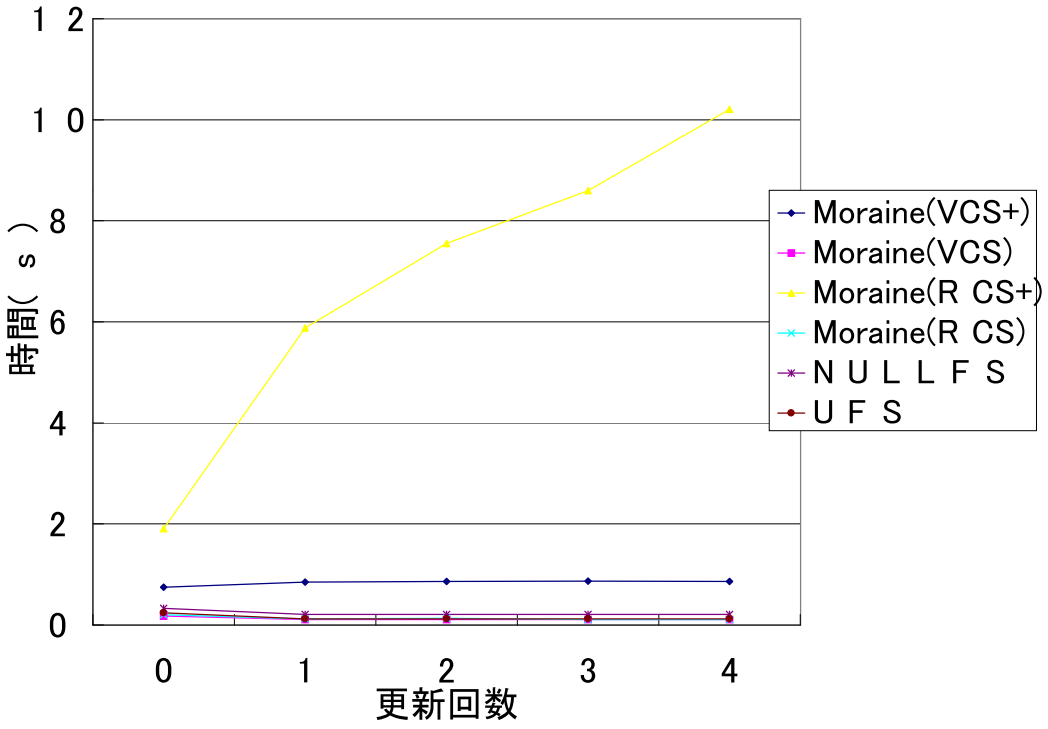

図 2.10: バージョンの更新による時間

システム VCS は以前のバージョンからの差分は取らずにバージョンの記録を行うため回数によらずファイルの大きさが同じならば，更新する時間はほぼ一定である．RCS の場合，前バージョンとの差分をとろうとするため回数が増えると，更新する時間も増加する．

以上の結果から，高速性が必要ならばバージョン管理サブシステムに VCS を用いるといい事が分かる．しかし，VCS にも RCS と比べるとバージョンの記録ファイルの大きさが大きくなるという欠点があるため，使い分ける事が重要である．ファイルの容量については次の節でのテストで確認できる．

2.4.4 コンパイルテスト

実際のアプリケーションのコンパイルに要する処理時間を測定した．測定対象となるアプリケーションとしては FreeBSD に付属の tar と dump とした．アプリケーションのソースファイルは各ファイルシステム上に用意しておき，make の開始から終了までを処理時間とした．各アプリケーションのファイルのサイズと行数を表 2.8 に示す．そして，測定結果を図 2.11 に示す．

make が行う作業は，異なるソースファイルのコンパイルの繰り返しである．コ

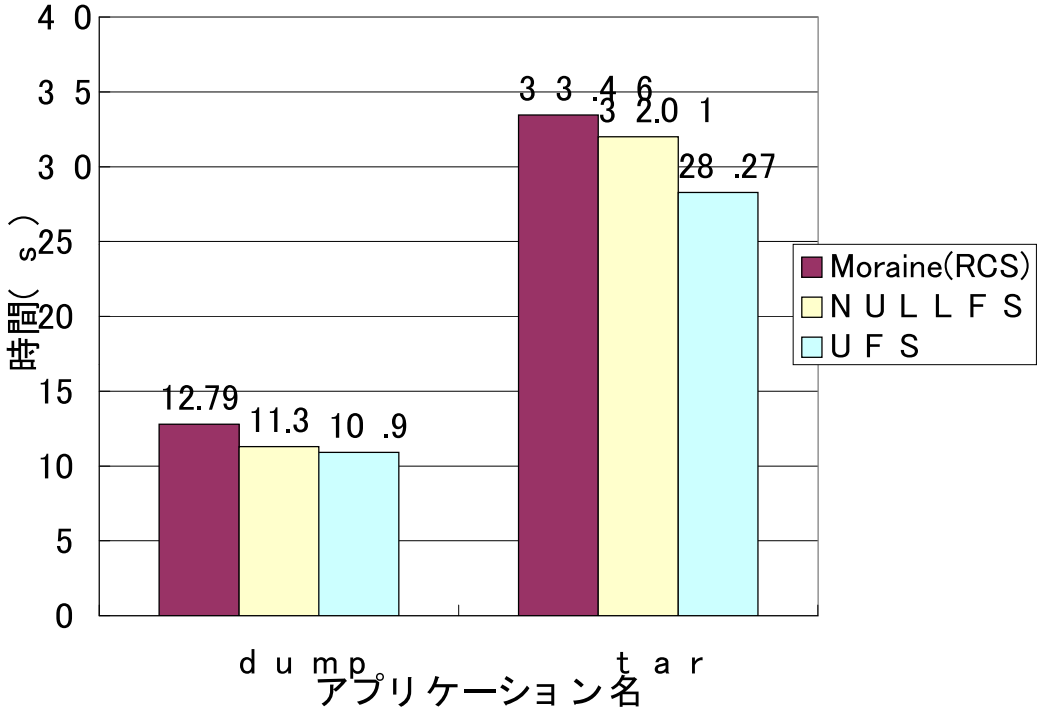

図 2.11: アプリケーション作成時間

ンパイルはソースファイルの読み出し，そのソースファイルから生成されたオブジェクトファイルの書き込みであり，ファイルシステムに対して読み書きを行う．Moraine は UFS 比べると 20%程度の時間を要する．これは，実用上問題ない程度の時間である．

2.4.5 容量テスト

大阪大学基礎工学部情報工学科で行われるコンパイラ作成演習で得られたプログラムの編集履歴データを Moraine に適用した．今回の実験データは小規模なデータであるが，実際の大規模な開発に対して実験を行う前に，Moraine の実用性を評価するために小規模な場合でも有効かどうか確認を行った．適用したデータを表 2.9，各学生の作成したファイルについて表 2.10 に示す．“全行数” は最終バージョンの C 言語のソースファイルの全行数を表す．“総ファイル数” は最終バージョン C 言語のソースファイルの総数を示す．“総バージョン数” は C 言語のソースファイル，オブジェクトファイルなどすべてのファイルのすべてのバージョンの総数を示す．括弧内の値は C 言語のソースファイルのみのバージョンの総数を示す．例えば，student 2 は全行数が 4067 行で，20 個のソースファイルを作成し，総バージョ

表 2.8: 各アプリケーションのファイルのサイズと行数

dump のファイル

ファイル名	サイズ	行数
Makefile	1097	37
dump.8	11151	400
dump.h	7358	216
dumprmt.c	8779	415
itime.c	6975	274
main.c	16558	635
optr.c	12236	533
pathnames.h	2099	42
tape.c	21825	875
traverse.c	15714	612
unctime.c	3217	103
合計	107009	4142

tar のファイル

ファイル名	サイズ	行数
COPYING	17982	339
ChangeLog	62125	1738
Makefile	667	17
Makefile.gnu	6480	185
README	1839	40
buffer.c	35197	1584
create.c	34371	1478
diffarch.c	16700	760
extract.c	25250	946
getdate.y	23224	977
getoldopt.c	2310	96
getopt.c	20131	712
getopt.h	4327	125
getopt1.c	3479	161
getpagesize.h	610	38
gnu.c	14028	677
list.c	20292	886
mangle.c	6635	270
msd_dir.h	1060	44
names.c	3302	149
open3.h	2639	67
pathmax.h	1691	53
port.c	25906	1256
port.h	5312	215
rmt.h	3321	98
rtapelib.c	13530	570
tar.1	13134	445
tar.c	38002	1570
tar.h	9491	296
update.c	13900	585
version.c	50	1
合計	42695	16378

表 2.9: 適用したデータ

	全行数	総ファイル数	総バージョン数 (ソースファイルのみ)
student1	9339	45	533 (311)
student2	4067	20	249 (147)
student3	2543	18	357 (247)

ン数は 249 個にであることを示す．そのうち，C 言語のソースファイルのバージョン数は 147 個であった．

各ファイルシステム上での最終的なソースコードの総容量を表 2.11 と図 2.12 に示す．

UFS 上の容量が元の大きさであり，この大きさと比べるとバージョン管理サブシステムでの容量は大きくなるが，RCS を用いた場合は数倍程度であり，VCS を用いた場合は数十倍まで増加するが，近年の HDD 容量では実用上問題ないと考える．

採用するバージョン管理サブシステムによってディスク使用容量が大きく変化し，ディスク容量を重要視するならば，VCS より RCS をバージョン管理サブシステムとして使用すればよい．

2.5 考察

Moraine は，現在の大容量なハードディスクがある環境下では有用である．作成したすべてのファイルのすべてのバージョンを開発者に負担をかけることなく自動的に記録するため，開発者はバージョン管理の機構やコマンドなどを学ぶ必要がない．Moraine を導入しても，開発者の環境を一切変更することなく保存されたファイルの履歴をすべて保存できる．

Moraine をバージョン管理システムとして見た場合，効率的に任意のバージョンを記録し，取得できるという機能を持っている．また，任意のバージョンに明示的なタグを用いることで指定のバージョンの特定を容易に行うこともできる．

現在の Moraine の実装では，タイムスタンプや指定したタグを用いることで過去のバージョンの取得を行う．これは CVS や RCS と同等の機能である．しかし，Moraine で記録したバージョンの粒度は CVS や RCS で記録したバージョンより細かい．そのため，過去のバージョンを検索し取得するための，より洗練された機構が必要である．

Moraine は利用者が保存したファイルを自動的にすべてのバージョンを記録す

表 2.10: 各学生の作成したファイル
student1

ファイル名	最終ファイルサイズ	バージョン数
block.c	5225	17
block.h	5161	5
checker.c	12857	1
code.c	738	8
code.h	738	4
data.c	2170	2
dyalloc.c	869	11
dyalloc.h	869	3
dynamic.h	1027	1
er.c	154	1
error.c	114	19
error.h	117	5
exp.c	15087	15
exp.h	15018	5
igen.c	2637	5
kai.c	3131	3
main.c	1395	29
master.c	6507	2
mmain.c	896	10
mmain1.c	2485	2
originmain.c	2017	2
rep-st.c	6810	1
rep1.c	3259	1
s.c	6081	1
sc.c	5652	1
sc1.c	6677	1
scan.c	7946	1
scanner.c	5918	24
scanner.h	5794	7
shiki.c	5263	5
spc.h	2809	22
sscan.c	7944	9
sss.c	7938	1
ssscan.c	7935	4
ssss.c	39	2
ssst.c	8865	1
sst.c	3131	1
st.c	8865	40
st.h	6076	4
symbol.c	3495	11
symbol.h	3495	5
symbol2.c	1397	1
type.c	4115	13
type.h	4112	4
typeexp.h	1390	1
合計	204218	311

studnet2

ファイル名	最終ファイルサイズ	バージョン数
code.c	10295	2
code2.c	9710	2
code3.c	9644	2
count.c	495	2
decl.c	8388	13
expcode.c	9286	2
expcode2.c	4352	2
expres.c	8350	18
prog.c	4853	19
prog2.c	5321	1
program.c	5207	1
scan.c	8587	19
scanner.c	4452	1
spcdef.h	3890	8
spclib.c	2842	9
spcmain.c	510	10
sscdef.h	1216	6
ssclib.c	629	5
sscmain.c	307	7
st.c	7498	18
合計	105832	147

studnet3

ファイル名	最終ファイルサイズ	バージョン数
block.c	9951	20
const.c	823	2
dallo.c	1066	19
error.c	162	25
exp.c	20321	19
gene.c	1095	23
label.c	141	7
machi.c	1136	5
main.c	1017	25
queue.c	686	10
sc1.c	923	3
sc8+.c	5113	1
sc8.c	3770	1
scan.c	7981	24
spc.h	2736	11
st.c	7185	19
symbol.c	3432	15
type.c	2341	19
合計	69879	247

表 2.11: ディスク使用容量 (KB)

	UFS	Moraine(RCS)	Moraine(VCS)
student1	225	657	1784
student2	117	290	899
studnet3	73	280	1020

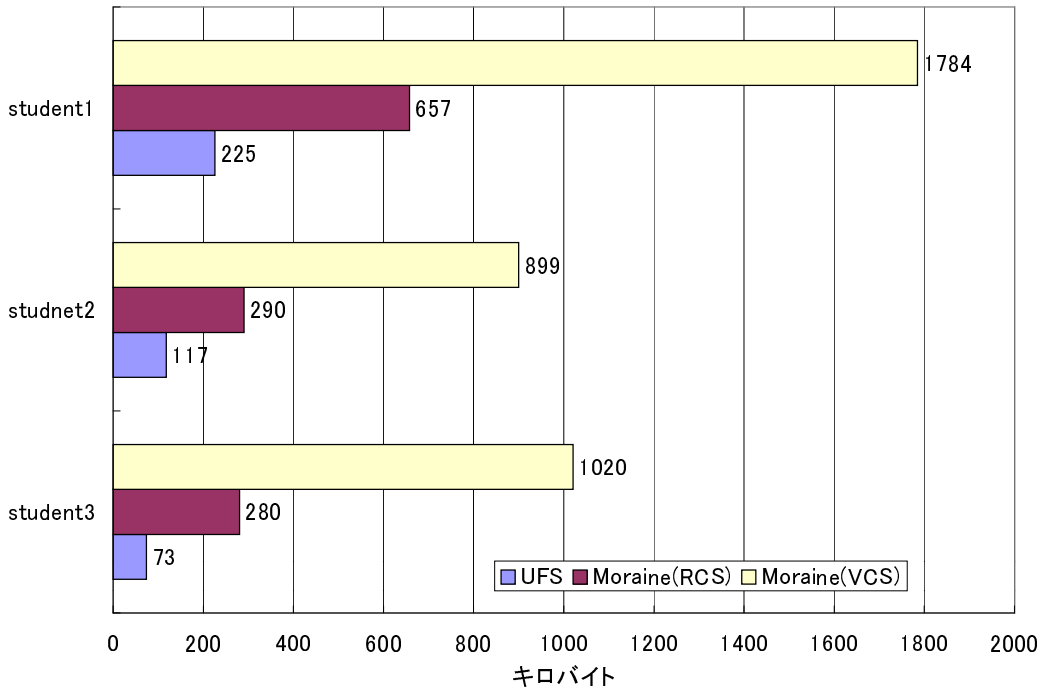

図 2.12: ディスク使用容量

る．バージョン管理システムとして関連したツールとして RCS[44]，SCCS[39]，3D Filesystem[20]，ClearCase[2]，PVCS[31]，Visual SourceSafe[32] などがある．RCS や SCCS は，開発者の作業対象となるファイルとは別に，各バージョンを記録するためのファイルが存在し，これを操作するツール群によって構成される．各バージョンは作業者がツールを用いて明示的に checkin, checkout の動作を指示することによって記録される．しかし，これらのツールを利用するにはツール等の利用方法を学ぶ必要があり，誤った操作を行った場合にはバージョンの記録が行えないといった欠点がある．また，その他のシステムではファイルシステムとバージョン管理機能が組みあわさっており，これと連携するツール群によって構成される．バージョンはファイル名で区別され，ツールによって明示的に指示することでバージョンが定義，記録される．また，これらのシステムは，check-in/check-out モデルを採用しており，分散開発や同時並行開発やビルド管理といった機能を実現している．さらに，グラフィカルユーザインターフェースを用いて操作を行うことができ，特定の開発環境と統合された場合にのみ使用できる．そして，これらのシステムはさまざまな粒度でオブジェクトのバージョンを記録する．

しかし，ファイル構造が専用の形式であり，バージョン管理モデルが固定であるため利用する際には制約が大きい．さらに，開発者はシステムの利用方法を知る

必要があり，check-in/check-out コマンドを利用しなければならない．また，特定の環境を想定しているため，汎用性に欠けるといった問題がある．一方，Moraine ではバージョンの記録は完全に自動化されており，ファイルシステムとして実装することで特定の環境に依存しない．しかしながら，Moraine はバージョンを取得する機能に中心がおかれており，ClearCase や PVCS などのツールに比べファイル間の関係の管理や複数人での開発を支援するなどの機能が現在の実装には備わっていない．また，一部の OS で用いられている自動バージョン番号付与ファイルシステム [28] は，ディスクの容量による制約から，保存できるバージョン数や廃棄期限が決められるのが通常である．また，これらのファイルシステムは OS と不可分な形で実現されており，可汎性も問題がある．一方，Moraine はファイルの全ての保存作業で生じたすべてのバージョンを保存している．

Moraine の性能は，実際のソフトウェア開発で十分実用的と言える．バージョンを記録する際の性能の低下はあまり存在しない．ファイルの差分を計算することによって，システムの負荷は上昇するが，この計算はバックグラウンドで実行されるため，実際には計算を終える前にファイルの操作は終了する．これにより，ユーザに対する負荷はほとんど存在しない．

Moraine は新しいソフトウェア開発環境モデルの基礎となるであろう．現在のソフトウェア開発環境はファイルなどに対して多くの管理操作が必要である．また，システムにすべての必要なファイルを保存しなけれならない．Moraine を用いることで，必要なくなったファイルを完全に消去することもできる．消去されたファイルも Moraine に保存されているため，将来またそのファイルが必要になった場合に，ファイルを復元することが容易に行える．

2.6 結論と課題

従来のバージョン管理システムとその問題点について述べた．そして，その問題を解決するバージョン管理機能を持つファイルシステムの設計と実装を行い，システムの評価を行った．本ファイルシステムは，

- 十分に高速な動作が行え，バージョン管理機能を容易に利用できる．
- 利用するバージョン管理機構を選択できるため，開発環境に応じた管理手法を適用することが可能である．

といった利点がある．

関連研究として分散型，複数人による分散ファイルシステム [14, 19] などの研究が行われている．そこで，今後の課題として分散環境に対応したバージョン管理

機構への対応が挙げられる．また，実環境への適用によるユーザビリティの評価が挙げられる．また，本ファイルシステムを発展させることによって，プロダクトに関する管理を扱いやすくすることが可能になっているが，これをソフトウェアプロセス中心型開発環境 [25, 26, 27] に対して適用することにより，プロセスとプロダクト双方の質を高めるためのソフトウェア開発環境の構築などが挙げられる．

第3章　メトリクス計測システム MAME

3.1　導入

現在まで、数多くのメトリクス計測システムが提案され，実装されている [5, 23, 40, 45]．しかし，これらは事前に収集するメトリクスを決め，そのためのツールを用意する必要がある．一般に，メトリクスを計測するためには，プロジェクトを開始する前にデータを収集する計画を立てる．例えば，GQM パラダイム [4] では目標に対して質問を決めることが必要である．それらを決めた後に収集するメトリクスを選ぶ．このようなトップダウン的な手法は，プロジェクトを開始する前にプロジェクトが詳細に決まっていれば成功する．そして，プロジェクトの開始時からデータの収集を行う事ができる．

しかし，この手法では、プロジェクト管理の方針の変更に対応することは困難である．プロジェクトの進行中や終了後に，計画したもの以外の新たなメトリクスデータを収集することはできない．失われたプロダクトや終了したプロセスからのデータの収集は現在のソフトウェア開発環境では実行不可能である．

そこで，ソフトウェア開発中にデータを収集しておき，そのデータを基にメトリクスが計測できる環境を構築すればよい．本章では，このデータ収集部分に 2 で述べた Moraine を用いたメトリクス計測システム MAME(Moraine As a Metrics Environment) の提案とその実現について述べる．Moraine はすべてのファイルのバージョンを細粒度で保存するため，MAME では，プロジェクトの開始後でさえもメトリクスを収集する方針を変更することが可能になる．

さらに，本研究では，MAME を実際の学生のコンパイラ作成演習に適用した．収集するメトリクスは演習の終了後に決定し，その後メトリクスの収集を行った．そして，これらのメトリクスの解釈から学生の演習活動の特徴が理解できた．

以降，3.2 ではメトリクス環境に必要な条件について考察し，3.3 では Moraine を用いたソフトウェアメトリクス計測システム MAME に付いて述べ，3.4 では MAME を学生実験に適用した結果を節で述べる．3.5 で MAME について考察を行い，最後に 3.6 節でまとめと今後の課題について述べる．

3.2 メトリクス環境の必要条件

本節では，定量的計測を行うメトリクス環境として必要な事項について考える．メトリクス環境とは，一般のソフトウェア開発の定量的プロセスやプロダクトの計測のための環境である．メトリクス環境では，開発者の活動から定性的ではなく定量的なメトリクスを収集する．また，データを保存し分析するための機能を提供する．メトリクス環境によって収集されたデータは，ソフトウェアプロセス評価の際にも使用できる．我々は，メトリクス環境には以下の 4 つが必要だと考える．

1. **開発者に余計な負担をかけない．**
   開発者は，“特別なツールを使うように” や “あらかじめ決められた方法で開発してくれ” といった，データを収集するために特別な操作を強要されるのを好まない．そのため，これらの余分な作業について考える必要がある．さらに，開発者にこれらの作業を課すことは収集するデータの品質に問題を引き起こすかもしれない．

2. **さまざまな種類のメトリクスデータを容易に収集できる．**
   あるメトリクスデータを収集するたびに単一のプログラムを利用するのではなく，汎用性のあるツールからなるツール群を構築する．これらのツール群は多くのメトリクスデータを収集するための一般的な環境を持つ．

3. **さまざまな粒度のメトリクスデータを容易に収集できる．**
   ソフトウェア開発活動には，開発者や管理者といった多くの側面からの活動がある．さまざまな側面に応じて，異なる粒度のデータが必要となる場合がある．さらに，あるメトリクスにおいてさまざまな粒度を考えることで，メトリクスの抽象化をより多くの角度から行うことができる．

4. **収集したメトリクスデータの構造は独自なものではない．**
   近年，オープンソースの考え方 [21] が広がりつつある．ソフトウェアアーキテクチャ，設計，実装などを隠すことなく，世の中へ公開する動きがある．このような状況では，収集するメトリクスデータに一般的に用いられる構造を採用することはとても重要である．メトリクス環境の質を高めることにもなる．

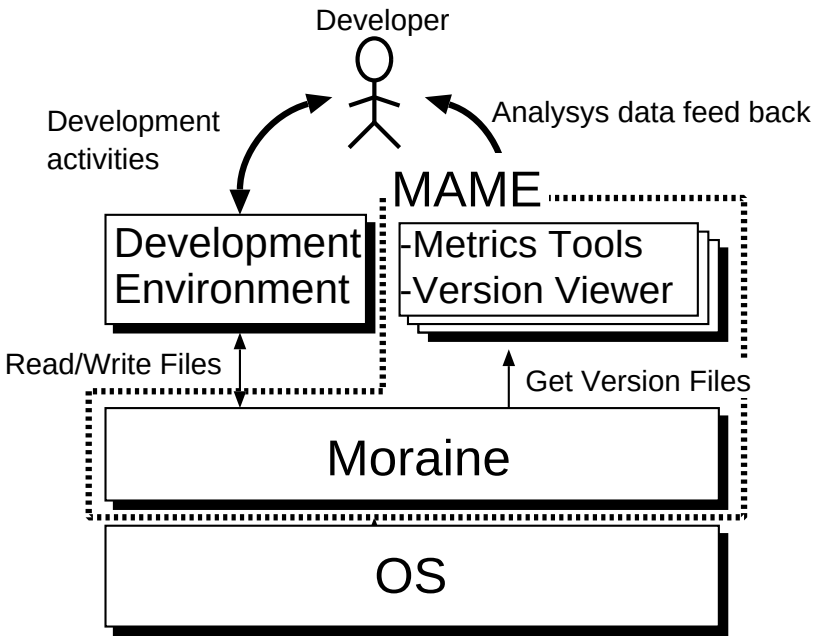

図 3.1: MAME のアーキテクチャ

3.3 MAME

3.3.1 アーキテクチャ

図 3.1 に MAME(Moraine As a Metrics Environment) のアーキテクチャを示す. MAME は Moraine を利用したメトリクスデータを容易に収集できるメトリクス計測システムである．Moraine が収集したファイルの更新履歴を用いて，さまざまなメトリクスデータを提供する．さらに，任意のファイルごとの詳細な更新履歴を，Web ブラウザを用いて参照できるツールも提供する．Moraine 上に開発環境を構築することにより，開発環境中のファイルに対する全ての操作は Moraine によって記録される．開発者はデータの分析ツールなどで，記録された履歴を Moraine から取得することができる．これらの情報を得るにあたって，開発環境自身を変更する必要はなく，単に Moraine 上でこれまでと同様の開発を行うだけである．開発者はメトリクスを MAME で取得可能かどうかを気にすること無く，開発作業を続けることができる．

MAME にはさまざまな種類のメトリクスツールが含まれている．現在の実装には，メトリクスツールが出力する情報はバージョンの数，ファイルの更新時間，ファイルの更新者，ファイルの行数，ファイルが C 言語で記述されている場合は関数の個数，ディレクトリ内のファイル一覧がある．また，これらの情報を組み合わ

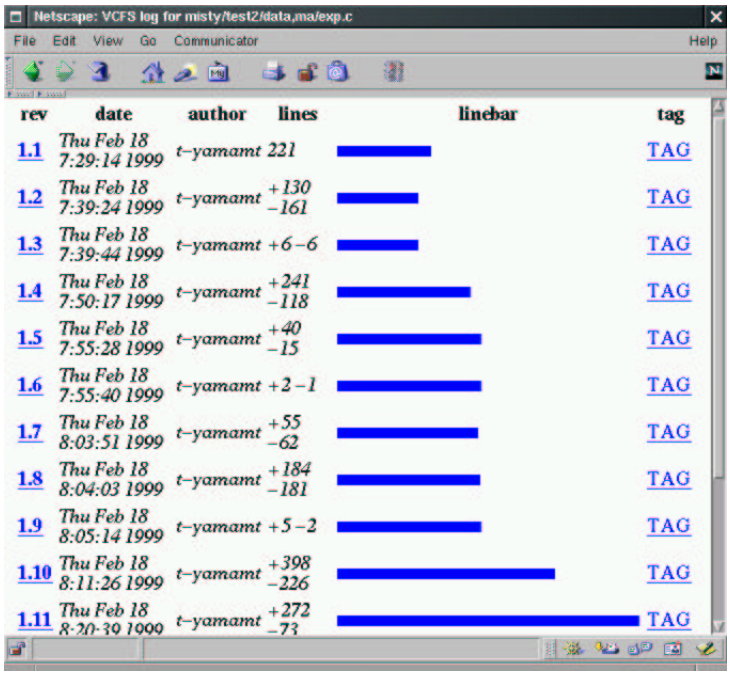

図 3.2: バージョンビューア

せた出力も可能である．これらのデータはテキスト形式で出力され，Perl などの言語を用いて容易に加工ができる．

図 3.2 はバージョンビューアの画面イメージである．バージョンビューアは Web ブラウザを用いている．画面の一番左の列はバージョン番号を表す．バージョンは最初のバージョンである 1.1 から始まり，最新のバージョンまですべてを表示する．各バージョンは，バージョン番号，更新した日付，更新した開発者，前バージョンからの追加もしくは削除された行数，行数を棒グラフ化したもの，もしタグがあればそのタグから構成される．このツールを用いることでバージョンの変更履歴が視覚的に表現できる．

3.3.2 機能

MAME は，3.2 に示したメトリクス環境に必要な機能を満たしている．以下に MAME におけるそれぞれの機能について説明する．

1. MAME では，メトリクスのデータの収集に Moraine を用い，従来の開発環境を Moraine の上にそのまま構築している．このため，開発者は開発者自身の環境を変更する必要がなく，MAME を導入する以前の開発形態でそのまま開発が続けられる．つまり，開発者に対してメトリクスを収集するための

特別な作業を指示する必要はなく，余計な負担をかけない．

2. MAME は，開発環境上のファイルに対する更新履歴を収集する．我々はソフトウェア開発作業の多くはファイルに対する操作だと考える．MAME はファイルの内容の変更や，ファイルの作成，削除といったファイルに対するすべての操作を記録する．ファイルの更新時間や更新者，ファイルの行数といったデータが容易に取得可能であり，これらのデータを加工することによってメトリクスデータを容易に収集できる．

3. MAME は，開発者が保存したファイルの更新履歴をすべて保存している．そのため，ファイルに関する細粒度のデータが取得できる．また，開発時に行った操作を別途に記録し，共に用いることで，より高い抽象度のデータを得ることも可能であろう．

4. MAME で取得可能なデータはファイルに関する情報のテキスト形式である．出力されたデータをそのまま使用することもでき，複数のデータを独自に加工するこも可能である．

3.4 実験

ソフトウェア開発作業後に，開発中に作成されたファイルに関するメトリクスデータが，MAME で実際に取得可能であることを示す．

3.4.1 概要

大阪大学基礎工学部情報工学科で行われるコンパイラ作成演習で得られたプログラムの編集履歴データを MAME に適用した．適用したデータを表 2.9 に示す．今回は，学生の開発動向を知るために，3 つのメトリクスを選んだ．それぞれ，ファイルの総数 (同時に存在する数)，ソースファイルの全行数，C 言語のソースファイル中の関数の総数である．

図 3.3，図 3.4，図 3.5 は student 2 の各メトリクス値を表している．これらのグラフの横軸は C 言語のソースファイル (147 個のバージョン) の累積したバージョン数を表している．横軸にはバージョン数ではなくファイルの更新時間を用いることも可能であるが，学生の演習の場合は実時間よりもバージョン数での推移の方が良いと考える．なぜならば，学生の場合は連続した開発は行わず，実時間には関係ない時間で開発するためである．

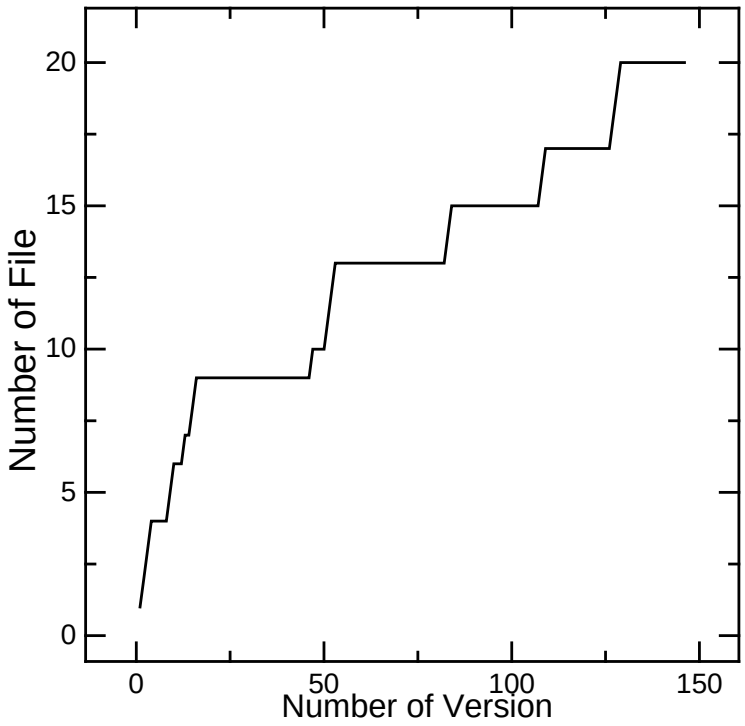

図 3.3: ファイル数の推移 (Student 2)

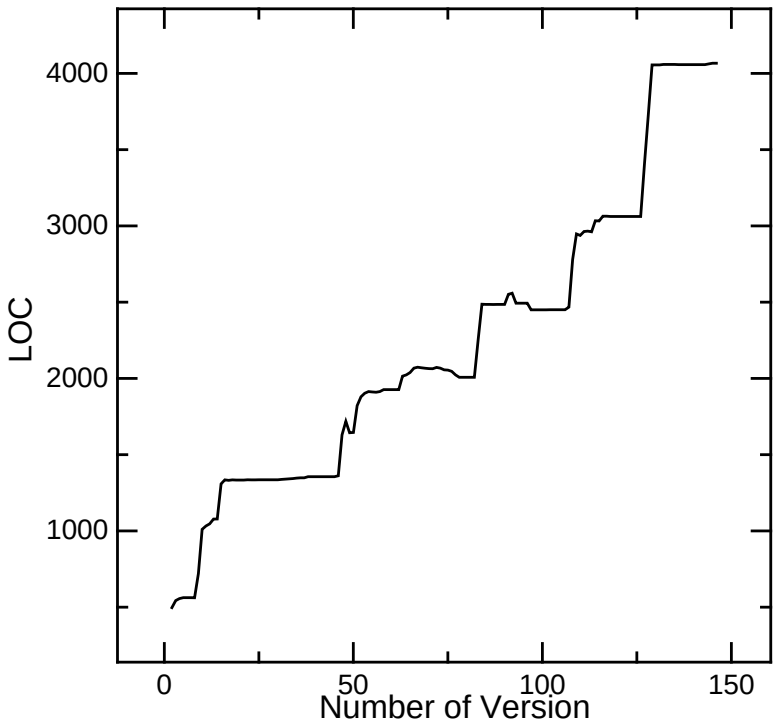

図 3.4: 行数の推移 (Student 2)

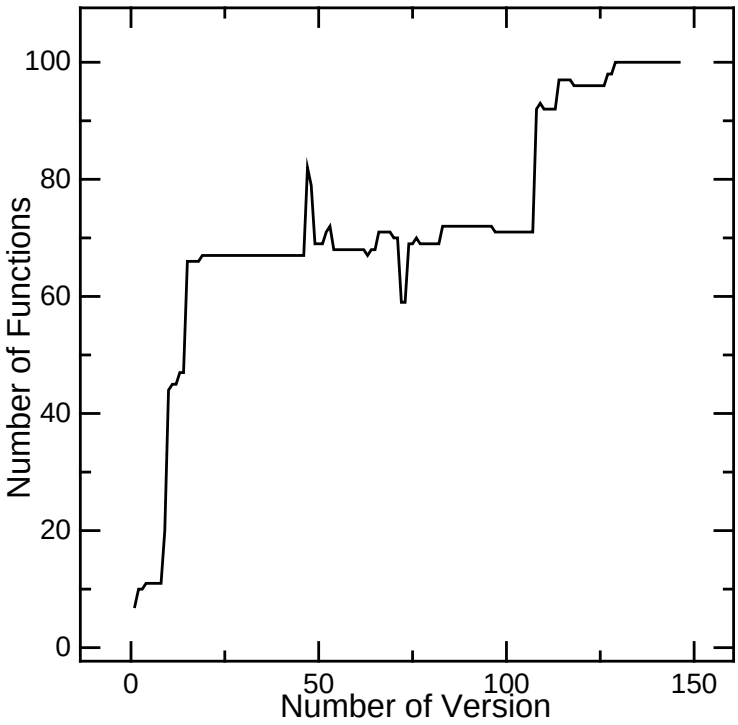

図 3.5: 関数の数の推移 (Student 2)

3.4.2 メトリクスデータの分析

図 3.3 と図 3.4 から，開発は特に問題なく順調に進んだと考えられる．両方のグラフ共に，ほとんど単調増加を示している．バージョン数が 20 から 50 の間では，すべてのグラフにおいてほとんど変化は見られない．これは，バージョン数は増えているがファイルの行数はほぼ一定であることを示している．これらのバージョンに対して作業を行っている間，学生はファイル内の細かな変更を行っていると考えられる．

図 3.5 は初期の段階 (バージョン数が約 20) で大きな増加を示している．そして，しばらくその状態が続いている．学生は初期の段階で必要な関数を作成し，後にその関数内を記述していることが分かる．また，最後の段階 (バージョン数が約 100) で多くの関数を追加している．

このように，学生が行った開発の動向は，MAME の出力から容易に得ることができる．

3.5 考察

MAME は Moraine を用いて実現されており，ソフトウェアメトリクス計測システムとして新しいアーキテクチャである．MAME は 3.2 節で示したメトリクス環境にとって必要な機能を全て持っている．このため，開発者に特別な強制を強いることなく，メトリクスデータを収集するための負荷もほとんどない．メトリクスデータを集める際，蓄積されたバージョンは後で容易に取り出すことができる．MAME を導入して開発を行った場合，開発作業後でもファイルの行数などのファイルに関するメトリクスデータを，細かい粒度で取得可能になる．

MAME の本質は過去のバージョンや消去されたファイルに対するメトリクスデータを取得できる所にある．そのため，プロジェクトの途中や終了後でさえもメトリクスの収集方針を設定したり変更したりすることができる．一般的なメトリクス環境ではあらかじめ決められた収集方針があるが，MAME では開発前にあらかじめ収集方針を決める必要がない．この特性はすべてのファイルのすべてのバージョンを記録するシステムによって成り立っている．また，3.4 節で示した実験データによって，MAME は十分実用的であることが示されている．ディスクの容量は通常のファイル使用容量よりも必要であるが，近年のディスクの大容量化により問題にはならない．

ファイルの更新をバージョンとして記録しているため，ファイルの更新履歴のみが取得できる．そのため，なぜファイルを更新したかを知るにはファイルの更新履歴から分析するしかない．たとえば，オブジェクトファイルが更新された場合，コンパイルを行ったであろうといった分析を行う．このような分析から開発者の作業過程の推測が可能であろうと考えている．

既存のメトリクス計測システムとして METKIT (Metrics Education Toolkit)[40] がある．METKIT は通常の開発でメトリクスデータを収集するために使われる．開発者は必要なデータを収集するためにモジュールを導入し作成する．つまり，METKIT を利用する際には事前に取得するメトリクスを定める必要がある．Crocodile[23] も既存のツールと組み合わせた環境である．しかし，これもメトリクスの方針は開発前に定める必要がある．TAME[5] はメトリクス環境を構築するためのプロジェクトである．TAME はメトリクスデータを収集するのに GQM パラダイム [4] を導入している．そのため，収集するメトリクスを決定するのにトップダウン的な手法が用いられている．Ginger2[45] もまたメトリクス環境を構築している．しかし，独自のツール群を用いているため，実際の開発環境に適用することは困難である．

3.6 結論と課題

本章では，容易にメトリクスデータを収集可能な環境である MAME について述べた．Moraine で収集した開発履歴データは容易に二次利用が可能であり，MAME はそれらのデータを利用する一つのアプリケーションである．そして，MAME のようなメトリクス計測システムが容易に構築できることを示した．また，MAME では Moraine を用いることによって，開発者に負担をかけることなくメトリクスデータの収集を行うシステムとして構築が可能になった．

MAME の本質は過去に作成されたファイルや既に消去されたファイルに対するメトリクスデータを取得できる所である．この本質により，ソフトウェアの開発プロジェクトの途中や終了後でさえもメトリクスの収集方針を設定したり変更したりすることが可能になる．そのため，MAME では開発前にあらかじめ収集方針を決める必要がなくなる．

さらに，MAME をある学生のプログラム演習に適用し，MAME から得られたメトリクスデータを用いて学生の活動の分析を行った．MAME を用いることでメトリクスデータの収集及び解析は容易なものとなることが分かった．解析したメトリクスデータを用いることで今後の開発の改善に役立つことが期待される．

今後の課題として，MAME を大規模なソフトウェア開発プロジェクトに適用することが挙げられる．

第4章　類似度計測システムSMMT

4.1　導入

二つのソフトウェアシステムが与えられた時，そのシステム間の違いはどれぐらいあるのか，定量的に知ることは重要である．いくつかあるシステムのバージョン間における相違の度合を調べることによって，システムの保守の様子や進化の度合を知ることができる．また，システムを改変する際の有益な指針にもなる．さらに，リリース版とその修正版といった少し異なるバージョンが多数存在する場合，その管理にも役立つ．例えば，あるソフトウェアで多くのバージョンを作成した際に，どのバージョンがどのバージョンから派生したか分からなくなる場合がある．このような時，バージョン間の相違の度合から派生の系統を調べ，系統樹を作成することで解決することができる．また，最新バージョンにバグがあり，ソフトウェアを作成し直す場合においても，バージョンの系統樹からどのバージョンから作成し直せば良いかの目安になる．

システムの改変時にドキュメントが作成されていれば，それを手がかりにして，システム間の相違点を知ることは可能であろう．しかし，システム間の違いを定量的な値として得ることは容易ではなかった．システムが小規模で，全体の構造を人間が容易に把握できる場合は，そのシステムの個々の構成要素について定量的な値を調べ，システム全体の値とすることができよう．しかし，構造が複雑になり，数百，数千にも及ぶファイルから構成されるシステムでは，何らかの機械的な処理により，自動的に値を求めることが必須となる．

システム間の類似度を求める研究としてさまざまな研究がある．Baxter らは抽象構文木（Abstract Syntax Tree）を利用したクローン検出手法を提案している [6]. しかしながら，類似度を求める定義はあるが，その値の有効性については述べられていない．また，実際にシステムに適用した結果はなく，定量的な評価を行っていない．[1, 38, 47] は，プログラムの類似度を自動的に計測するツールであるが，大規模システムに適用した結果はない．[35] では，提案した類似度を実際のソフトウェアに適用し，用途に応じてどのような類似度が考えられるかについて考察をしている．

そこで，ソフトウェアシステムの類似部分を見つけるだけではなく，計測した

類似度を用い，ソフトウェアシステムの各バージョンの派生も取得できるシステムを構築する．得られた派生図は，ソフトウェアシステムの改良，保守に役立つと考えられる．本章では，ソースプログラムとして与えられた二つのソフトウェアシステムの類似度を求めるためのメトリクスとそれを自動的に計測するシステム SMMT（Similarity Metrics Measuring Tool）の提案について述べる．具体的には，類似度メトリクスの形式的な定義を行い，次にそれを求めるための現実的な方法を示す．さらに，その方法を用いて実際にシステム SMMT を種々のバージョンの UNIX 系 OS に適用した結果を示す．そして，得られた類似度を基にして，クラスタ分析を行い OS のバージョンを分類し，バージョンの樹状図を作成した．その結果，ここで提案するメトリクスを用いた樹状図は，OS の開発者が示した系譜図にほぼ対応することが分かった．

以下，4.2 では，類似度の形式的な定義とそれを計測するメトリクスを提案する．4.3 では，メトリクスを現実的に求めるための方法と SMMT の実現について述べる．4.4 では，UNIX 系 OS への適用について述べ，適用結果に基づいたクラスタ分析を行い，提案する類似度の妥当性の検証を行う．4.5 でまとめと今後の課題について述べる．

4.2 類似度とそのメトリクス

本節では，ソースプログラムとして与えられた二つのソフトウェアシステムの類似度メトリクスを提案する．二つのソフトウェアシステムの類似度メトリクスには，多くの要素が考えられる．たとえば，ファイルの数や行数，ファイル名の違いなどである．以降，我々が提案する類似度メトリクスの形式的な定義を行い，それを求めるための具体的な方法を説明する．

4.2.1 類似度の定義

[6] では，二つの抽象構文木の類似度を定義している．定義している類似度は一致する節点数を全ての節点数で割った値である．また，[35] では，類似度を二つのプログラムの全行数の中で共通部分の行数の割合と定義している．我々も同様に，二つのソフトウェアシステムで等価な要素数を全要素数で割った値を類似度とする．以下に我々が提案する類似度メトリクスの形式的な定義を示す．

ソフトウェアシステム P はそれを構成する要素の集合と考え，P = $\{p_1, \cdots, p_m\}$ と書く．二つのソフトウェアシステム $P = \{p_1, \cdots, p_m\}$, $Q = \{q_1, \cdots, q_n\}$ に対し，等価な要素の対応 $R_s \subseteq P \times Q$ が得られるとする．P と Q の R_s に対する類似度

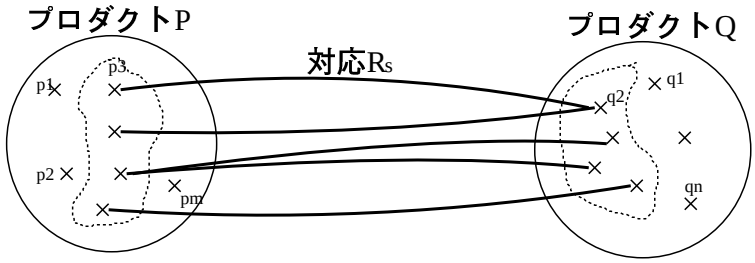

図 4.1: 要素の対応 R_s

$S(0 \leq S \leq 1)$ を次のように定義する．

$$S(P,Q) \equiv \frac{|\{p_i|(p_i,q_j) \in R_s\}| + |\{q_j|(p_i,q_j) \in R_s\}|}{|P| + |Q|}$$

これは，図 4.1 のように対応 R_s に含まれる P, Q の要素数を P と Q の総要素数で割ったものである．R_s に関係しない P, Q の要素が増えることによって S は下がる．$R_s = \phi$ では，$S = 0$ となる．また，P と Q が等価である時，$\forall i(p_i, q_i) \in R_s$ となり $S = 1$ となる．

4.2.2 類似度を求めるメトリクス

等価なファイル名を用いたメトリクス S_{fn}

ソフトウェアシステム P, Q に対し，p_i, q_j をそのソフトウェアシステムを構成するファイルとする．R_s を同じ名前を持つファイル同士の対応として類似度を計測するメトリクスを S_{fn} とする．この場合，容易にシステム間の類似度を計測できるが，ファイル名を変更した場合や名前は同じだがファイルの中身を変更した場合などは，類似度は直感的な値とは異なってしまう．また，ファイルの大きさにかかわらず均等な重みで類似度を求めるため，直感に合わない場合もある．例えば，小さな多数のファイルのみが対応にあり，少数の大きなファイルが対応していない場合，高い値になってしまう．

等価な行を用いたメトリクス S_{line}

ソフトウェアシステム P, Q に対し，p_i, q_j を P, Q それぞれの各ファイルの各行とする．直感的には P，Q の各ファイルを連結した一つのファイルをそれぞれ考え，その各行を構成要素とする．等価な行の対応を調べることによって対応 R_s が与えられる． この類似度メトリクスを S_{line} とする．これは，ファイル名やファイルの大きさに影響されず，直感的に近い値が得られることが期待される．

その他に，いろいろ類似度のメトリクスを考えることができようが，求める手間やその効果を考え，ここでは，S_{fn} と S_{line} について議論する．S_{fn} は容易に求めることができるので，以降では，S_{line} の求め方について詳しく述べる．

4.3 S_{line} の求め方と SMMT

本節では，前節で定義した類似度メトリクス S_{line} を求めるための対応 R_s の求め方の指針を述べる．さらに，与えられた二つのソフトウェアシステムから類似度を計算するツールについて述べる．

4.3.1 アプローチ

二つのソフトウェアシステム P，Q に対し S_{line} を求めるためには，R_s を得ること，すなわち P の各ファイルの各行が Q のどのファイルのどの行に対応するか，又は対応する行がないか，を知る必要がある．

このための簡単な方法としては，P の各ファイルを連結したファイル p_{all} と Q の各ファイルを連結したファイル q_{all} を作り，p_{all} と q_{all} の間の共通部分を求めて，そこに含まれる各行を R_s とすることが考えられる．共通部分の発見には，通常，最長共通部分列を発見するアルゴリズム LCS[33, 34, 46] を用いた diff[9] 等のツールが便利であるが，ファイル名が変わるなどしてファイルの連結順が変わった場合には，共通部分列として検出ができなくなる．

そこで，本研究ではコードの重複（クローンと呼ぶ）を求めるためのツール CCFinder[18] と diff とを組み合わせて R_s を求める．

CCFinder は，与えられたプログラムファイルの内に存在する同じプログラム文の系列を効率よく検出し，出力する．ただし，コメントや改行，空白の違い，また，変数や関数の名前（識別子）の違いがあっても同じものとして検出する．

まず CCFinder を用いて p_{all} と q_{all} の間に存在するクローンを検出する．クローン中の各行は R_s の要素とする．CCFinder は，後述のように保守作業にとって無意味なクローンを除去して出力する．しかし，除去されるものの中には類似度における対応としては有用なものもある．

そこで，検出されたクローンを持つファイル間に対し，共通部分列を diff によって求め，そこに含まれる各行も R_s の要素とする．二種類のツールを組み合わせることで，意味のある行の対応を効率よく求めることができる．

CCFinder では，プリプロセッサ命令（C 言語の#include 行など）は検出対象から除外される．また，クローンが完全に一致しないと検出されない．つまり，ある

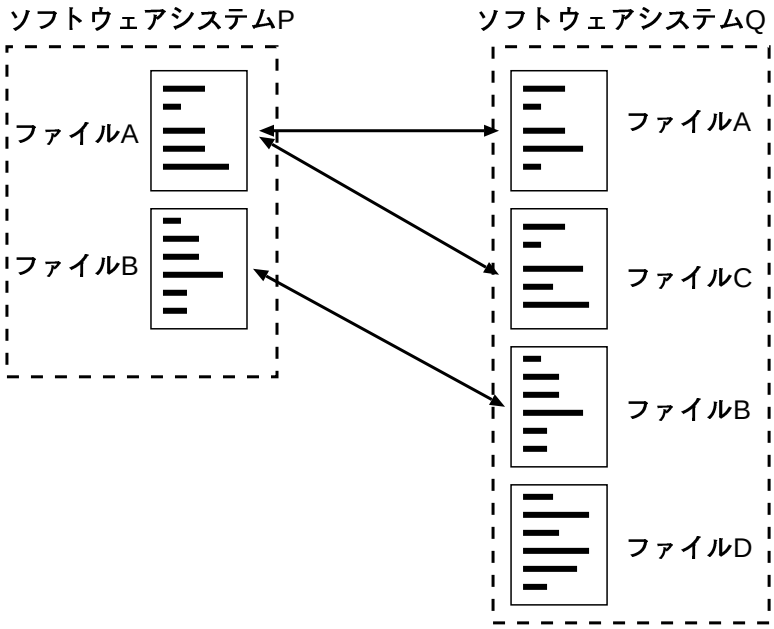

図 4.2: 対応の求め方

ファイルからソースコードをコピーし，一文でも追加，削除を行い他のファイルのペーストを行うと，それらはクローンとして検出されない場合がある．そのため，diff を用いた差分情報を加えることにより，同一行と判断できる行が増加し，より有効な行の対応が求められると考える．後節で述べる適用実験の結果，CCFinder だけを用いた対応の抽出方法より，diff と CCFinder を組み合わせた対応の抽出方法の方が対応をもつ行は一割程度増加し，diff と CCFinder を組み合わせた方法は有効な手法だと考えられる．

また，diff だけを用いた対応の抽出の場合，二つのソフトウェアシステムの間で同一の構造を持つ必要があり，ファイル名の変更やディレクトリ構造の変化に追従できない．同一の構造を持たない場合，P と Q のファイルの全ての組み合わせについて diff を実行しなければならず，多くの時間を要する．そのため，ここでは CCFinder と diff を組み合わせて対応を求める方法を採用する．

4.3.2 対応の求め方

CCFinder と diff を用いた対応の求め方の例を述べる．図 4.2 に示すようにソフトウェアシステム P, Q があり，P は 2 つのファイルから構成され, Q は 4 つのファイルから構成されているとする．また，Q は P の後継バージョンとする．Q を構成するファイルの中でファイル A とファイル B は，P のファイル A とファイル B

を基にしており，ファイル C はファイル A を基に新しく作成されたファイルである．さらに，ファイル D は新規に作成されたファイルとする．

この二つのソフトウェアシステム P，Q の R_s を求めるために，まず CCFinder を用いて P と Q の間に存在するクローンの検出を行う．検出されたクローンからクローンを含む行の間に対応を結び R_s の要素とする．さらに，クローンが一つでも存在するファイルの組を探す．図 4.2 の場合，矢印で結ばれた P のファイル A と Q のファイル A，P のファイル B と Q のファイル B，P のファイル A と Q のファイル C の間にクローンが一つ以上検出されたとする．

次に，これらの 3 つの組に対してのみ diff を用いてファイル間の差分を求める．差分の結果から共通行と判断された各行も R_s の要素とする．この方法では，まず全てのファイルを入力として CCFinder を実行し，その結果を使用して一部のファイル間の組み合わせについてのみ diff を実行する．CCFinder は $O(n)$ の手間で，高速に処理を行う．例えば，ファイル数が 1648 個で全行数が 50 万行のソフトウェアシステムの処理を行った場合の実行時間は約 3 分である [18]．また，C 言語の #include 行といった多くのファイルに存在するような行の対応もクローンが検出されたファイルの組に対してのみ付けられる．そのため，単純に共通行であるとしても対応はせず，意味のある行のみが対応する．

4.3.3 SMMT

S_{line} を求めるためのアプローチに基づき，S_{line} を計測するツールを作成した．以下に，ツールの入出力と処理の流れを述べる．図 4.3 に処理の流れを表す．

入力：二つのソフトウェアシステム P と Q
出力：P と Q の類似度 S_{line} $(0 \leq S_{line} \leq 1)$

Step1: 前処理

生成されるプログラムの機能に影響を与えない部分を取り除く．この処理は，用いられているプログラミング言語によって異なる．例えば，C 言語で記述されたファイルの場合，コメント部分，空行をすべて取り除く．これにより，diff を実行した時の類似度の精度を向上させる．

Step2: CCFinder の実行

与えられた二つのソフトウェアシステムを入力として CCFinder を実行させる．CCFinder の実行にあたって，最低一致トークン数を 20 とした．最低一致トークン数とは，一致するクローンを含むトークンの長さの最低値を表す．ツールのオプションとして，この値は変更可能である．

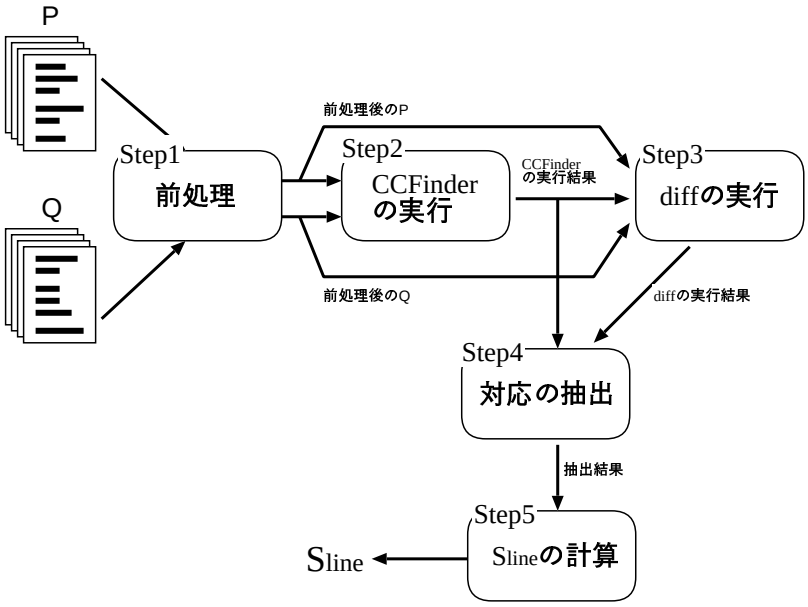

図 4.3: 処理の流れ

Step3: diff の実行

CCFinder の実行の結果，一つでもクローンが発見された P と Q のファイルの組の全てに対して diff を実行する．

Step4: 対応の抽出

CCFinder で検出されたクローンのトークン列から，一致している行同士を求める．さらに，diff で求まった差分情報から一致している行同士を計算する．CCFinder か diff のどちらかで一致している判断された行の組を R_s に入れる．

Step5: S_{line} の計算

S_{line} の定義より計算する．ただし，二つのソフトウェアシステムの全行数は前処理後の行数を用いる．

SMMT は Windows2000 上で稼働し，その対象言語は C，C++，Java，COBOL である．例えば，二つのソフトウェアシステムの総行数が 503884 行の S_{line} を計測する場合，PentiumIII 1GHz，メモリ 2GBytes のシステムで Step1: から Step5: までの処理に掛かる時間は 329 秒である（CCFdiner と diff の実行時間を含む）．

表 4.1: 各 OS のファイル数と総行数

FreeBSD

バージョン	2.0	2.0.5	2.1	2.2	3.0	4.0
ファイル数	891	1018	1062	1196	2142	2569
総行数	228868	275016	297208	369256	636005	878590

NetBSD

バージョン	1.0	1.1	1.2	1.3	1.4	1.5
ファイル数	2317	3091	4082	5386	7002	7394
総行数	453026	605790	822312	1029147	1378274	1518371

OpenBSD

バージョン	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8
ファイル数	4200	4987	5245	5314	5507	5815	6074	6298	6414
総行数	898942	1007525	1066355	1079163	1129371	1232858	1329293	1438496	1478035

4.4BSD

バージョン	Lite	Lite2
ファイル数	1676	1931
総行数	317594	411373

4.4 S_{line} の妥当性検証

4.4.1 UNIX 系 OS への適用

対象のソフトウェアシステムは，BSD UNIX である 4.4BSD Lite，4.4BSD Lite2[29] と，これらから派生した OS である FreeBSD[42]，NetBSD[43]，OpenBSD[36] を用いた．FreeBSD，NetBSD，OpenBSD は現在もオープンソースとして開発が進められている．本研究では，これら 3 つの OS のうち 4.4-BSD Lite のリリース以降に発表された主なバージョンを選んだ．その結果，FreeBSD は 6 バージョン，NetBSD は 6 バージョン，OpenBSD は 9 バージョンの 21 バージョンを選んだ．そして，4.4BSD Lite と 4.4BSD Lite2 を加え，総数 23 個の OS に対し，すべての組み合わせで類似度 S_{line} を計測した．計測対象は OS のカーネル部分のみとし，カーネルを生成するのに必要なファイルだけを取り出す．さらに，それらのファイルの中から拡張子が .c .h のファイルのみをツールへの入力とする．**3.3** の Step1: における言語依存部分では，入力されるファイルを C 言語とみなしコメントと空行は全て消去する処理を行う．

各 OS の総ファイル数と総行数を表 4.1 に示す．この表の値は，Step1: を行った後に測定した値である．計測した類似度の一部を表 4.2 に示す．

表 4.2 の FreeBSD だけに注目すると，各バージョンの中で，最も類似度が高い値を持つのはそのバージョンの前後のどちらかである．リリースした時期により前のバージョンか後のバージョンのどちらかになる．例えば，FreeBSD 2.2 と他のバージョンとの S_{line} を図 4.4 に示す．自分自身を除くと，最も類似度が高いバージョンは 0.706 の FreeBSD 2.1 である．次期バージョンである FreeBSD 3.0 との

表 4.2: 類似度一覧（一部）

	FreeBSD 2.0	FreeBSD 2.0.5	FreeBSD 2.1	FreeBSD 2.2	FreeBSD 3.0	FreeBSD 4.0
FreeBSD 2.0	1.000	0.833	0.794	0.550	0.315	0.212
FreeBSD 2.0.5	0.833	1.000	0.943	0.665	0.392	0.264
FreeBSD 2.1	0.794	0.943	1.000	0.706	0.421	0.286
FreeBSD 2.2	0.550	0.665	0.706	1.000	0.603	0.405
FreeBSD 3.0	0.315	0.392	0.421	0.603	1.000	0.639
FreeBSD 4.0	0.212	0.264	0.286	0.405	0.639	1.000
4.4BSD-Lite	0.419	0.377	0.362	0.226	0.138	0.101
4.4BSD-Lite2	0.290	0.266	0.258	0.179	0.133	0.100
NetBSD 1.0	0.440	0.429	0.411	0.291	0.220	0.140
NetBSD 1.1	0.334	0.348	0.336	0.254	0.193	0.152
NetBSD 1.2	0.255	0.269	0.265	0.225	0.190	0.158
NetBSD 1.3	0.205	0.227	0.225	0.201	0.208	0.179

	4.4BSD-Lite	4.4BSD-Lite2	NetBSD 1.0	NetBSD 1.1	NetBSD 1.2	NetBSD 1.3
FreeBSD 2.0	0.419	0.290	0.440	0.334	0.255	0.205
FreeBSD 2.0.5	0.377	0.266	0.429	0.348	0.269	0.227
FreeBSD 2.1	0.362	0.258	0.411	0.336	0.265	0.225
FreeBSD 2.2	0.226	0.179	0.291	0.254	0.225	0.201
FreeBSD 3.0	0.138	0.133	0.220	0.193	0.190	0.208
FreeBSD 4.0	0.101	0.100	0.140	0.152	0.158	0.179
4.4BSD-Lite	1.000	0.651	0.540	0.421	0.331	0.259
4.4BSD-Lite2	0.651	1.000	0.450	0.431	0.436	0.366
NetBSD 1.0	0.540	0.450	1.000	0.691	0.553	0.445
NetBSD 1.1	0.421	0.431	0.691	1.000	0.783	0.622
NetBSD 1.2	0.331	0.436	0.553	0.783	1.000	0.769
NetBSD 1.3	0.259	0.366	0.445	0.622	0.769	1.000

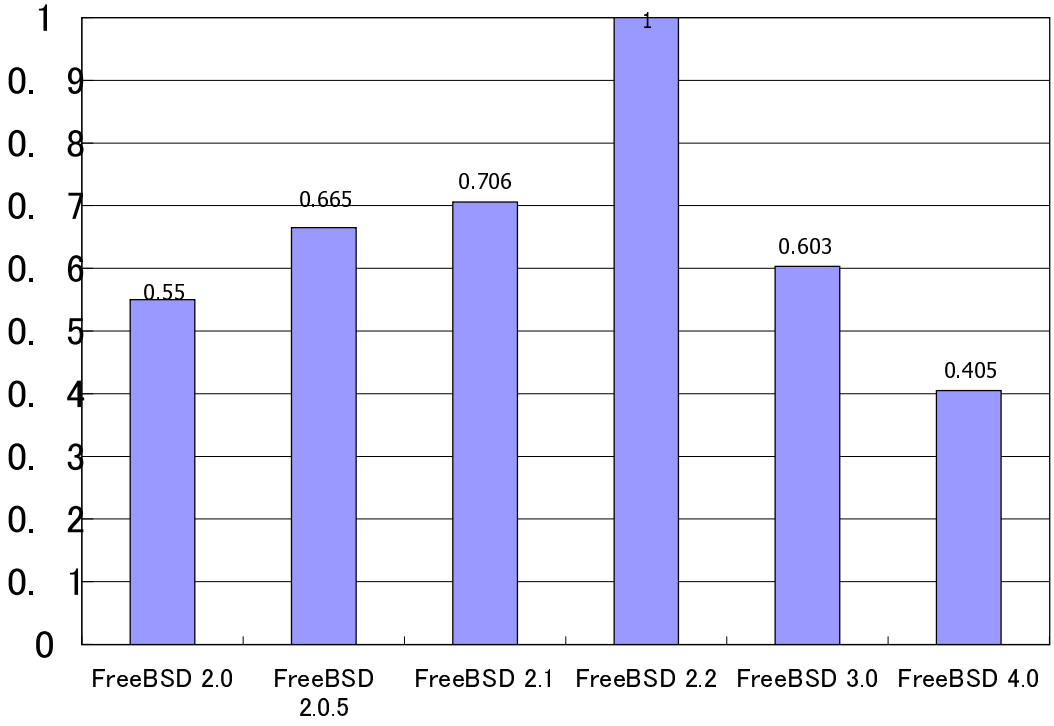

図 4.4: FreeBSD 2.2 に対する S_{line}

類似度は 0.603 であり，大幅な変更が加えられていることが読みとれる．表 4.1 のファイル数と行数の変化を見ると，ファイル数は約 1.8 倍増加し，行数は約 1.7 倍増加しており，それ以前の変更量とは異なることが分かる．

FreeBSD と NetBSD の間の S_{line} を図 4.5 に示す．FreeBSD 2.0 から FreeBSD 2.2 までは，NetBSD 1.0 と最も類似度が高く，NetBSD のバージョンが上がるにつれ類似度は減少していく．しかしながら，FreeBSD 3.0，FreeBSD 4.0 に関しては NetBSD 1.3 との類似度が他の NetBSD のバージョンと比べ高くなっている．FreeBSD と NetBSD という基本的に異なる開発環境（開発者や開発ポリシー）で行われている場合，類似度が上がる原因としては，他方の OS にある機能を追加するために，ソースコードをコピーした場合や，両方の OS に同一ファイル取り込んだ場合が考えられる．これを調べるために図 4.6 に示す BSD UNIX の派生図を調べた [41]. この図は，FreeBSD，NetBSD．OpenBSD といった OS がどのように派生し，いつリリースされたかを表している．FreeBSD 3.0，NetBSD 1.3 は，共に 4.4BSD Lite2 を取り込んだ最初のバージョンであることが分かる．つまり，FreeBSD，NetrBSD に取り込まれた 4.4BSD Lite2 のソースコードの行が対応し，類似度が増加したと考えられる．これらのことから，S_{line} は OS の変遷を知る有効な指標といえる．

さらに，例えば，4.4BSD Lite2 に致命的なバグが発見されたり，使用禁止になっ

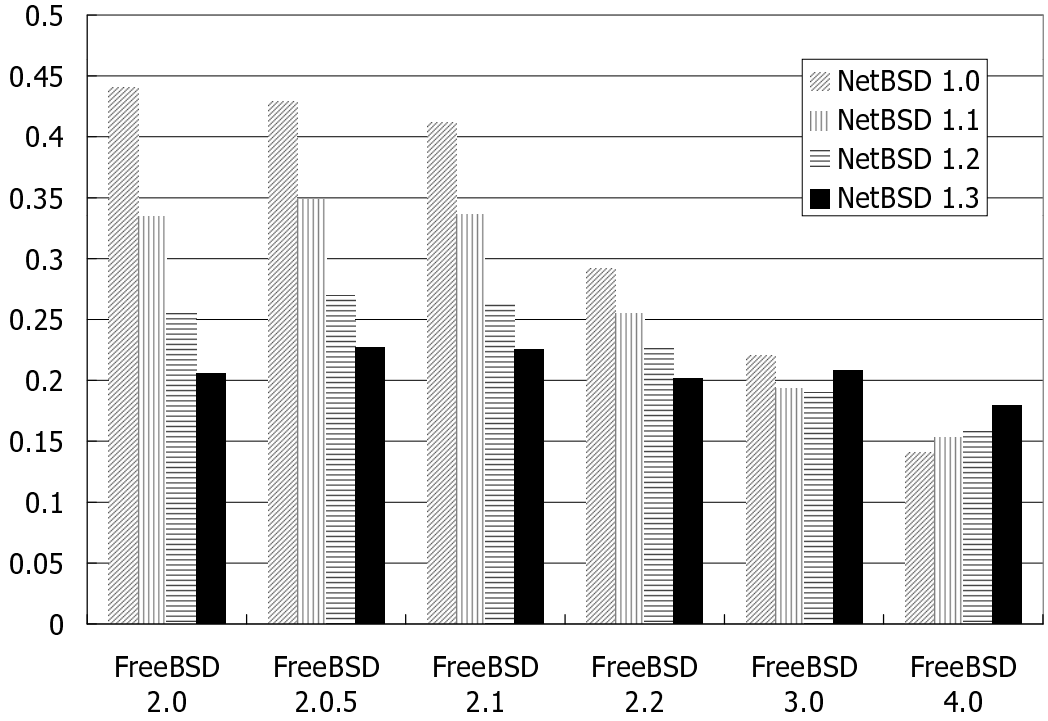

図 4.5: FreeBSD と NetBSD の S_{line}

た場合，4.4BSD Lite2 から取り込んだコードを取り除くか，取り込んだ以前のバージョンから最新バージョンを作成する必要がある．この際，取り込んだ以前のバージョンの特定は類似度を用い容易に検索可能であるため，類似度はソフトウェアを改変する際にも役立つ．

4.4.2 類似度メトリクスを用いたクラスタ分析

類似度メトリクス S_{line} を二つの OS の間の距離として，クラスタ分析 [12] を行う．クラスタ間の距離には平均距離を用いて計算する．クラスタ分析を行い，その結果から得られた樹状図（デンドログラム）を図 4.7 に示す．横軸は結合距離を表しており，左側で結合しているものほど類似度が高く，近い OS であることを示しているが，図 4.6 によってこの分析結果が正しいことが分かる．

図 4.7 では，最も大きな分類としてクラスタ I と II に分けられている．FreeBSD は全てクラスタ I に含まれ，OpenBSD は全てクラスタ II に含まれており，お互い類似度が高くない，遠い系統であることを示している．図 4.6 の派生図を見ると OpenBSD は NetBSD から分岐しており，FreeBSD とは違う系統として開発されてきたことが分かる．

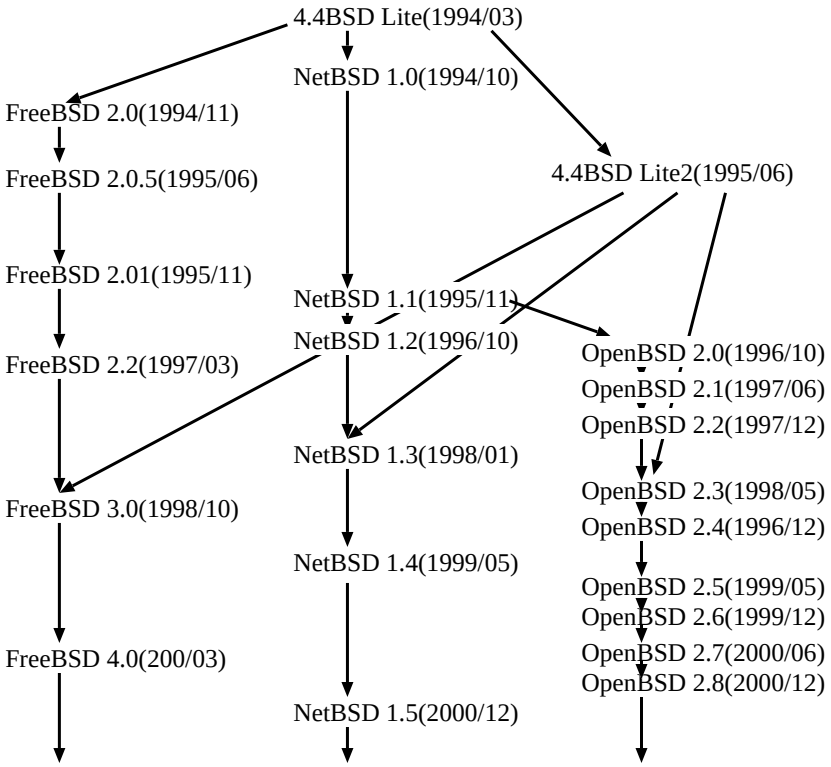

図 4.6: BSD 系 UNIX 派生図

次に，NetBSD と OpenBSD の分類を見てみる．OpenBSD 2.0 を除いた全ての OpenBSD のバージョンが同一クラスタに統合され，NetBSD 1.1 と統合されている．これは，OpenBSD が NetBSD 1.1 から派生していることを示している．

このように，類似度を用いたクラスタ分析を行うことでバージョンの系統図が作成可能である．多数のバージョンを作成してそれらの派生の特定が困難になった場合，それらのバージョンの類似度を計測しクラスタ分析を行うことで派生関係が分かる．

各個体間の距離を S_{line} 以外の類似度を用いてクラスタ分析を行った．ここでは，前述した S_{fn} を類似度として用いて作成した樹状図を図 4.8 に示す．この結果より図 4.7 と同様の結果が得られることが分かる．この理由は，これらの OS すべてが BSD 系由来のファイル構造，ファイル名を持つためだと考えられる．ファイル名命名規則やファイルの階層構造が決まったポリシーで開発が行われている場合は，単純にファイル名を用いて派生図を作成することが可能である．

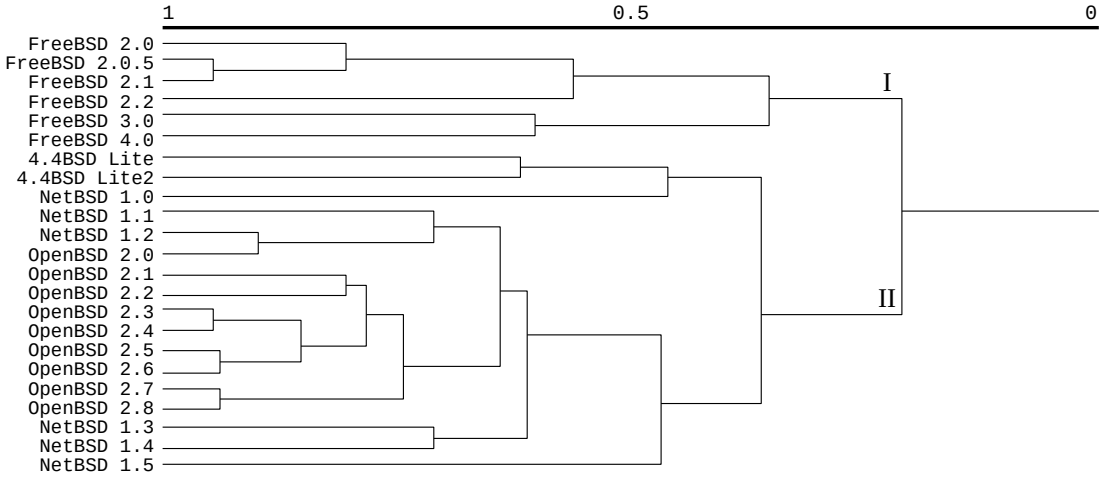

図 4.7: 類似度 S_{line} を用いた樹状図

表 4.3: 学生実験の S_{line}

	A	B	C	D	E	F	G	H
A	1	0.009	0.024	0.038	0.034	0	0.054	0.047
B	0.009	1	0.040	0.001	0	0	0	0.023
C	0.024	0.040	1	0.060	0.042	0.088	0.118	0.170
D	0.038	0.001	0.060	1	0.010	0.040	0.069	0.039
E	0.034	0	0.042	0.010	1	0.022	0.172	0.237
F	0	0	0.088	0.040	0.022	1	0	0
G	0.054	0	0.118	0.069	0.172	0	1	0.797
H	0.047	0.023	0.170	0.039	0.237	0	0.797	1

4.4.3 開発系統が異なる OS の類似度

由来が異なる二つの OS であり同時期にリリースされた FreeBSD 4.0 と Linux2.2.15[24] の S_{line} を求めたところ，0.031 と低い値を示した．対応する行はデバイスドライバがほとんどである．異なる開発系統の OS は S_{line} によって容易に区別することができる．

4.4.4 S_{fn} との比較

S_{line} の簡便な代替メトリクスとして S_{fn} が用いられるかどうか検討した．今回は，大阪大学基礎工学部情報科学科の Pascal コンパイラの開発演習において作成されたシステムに対し，S_{line} と S_{fn} を計測した．本演習では，指導書によって提

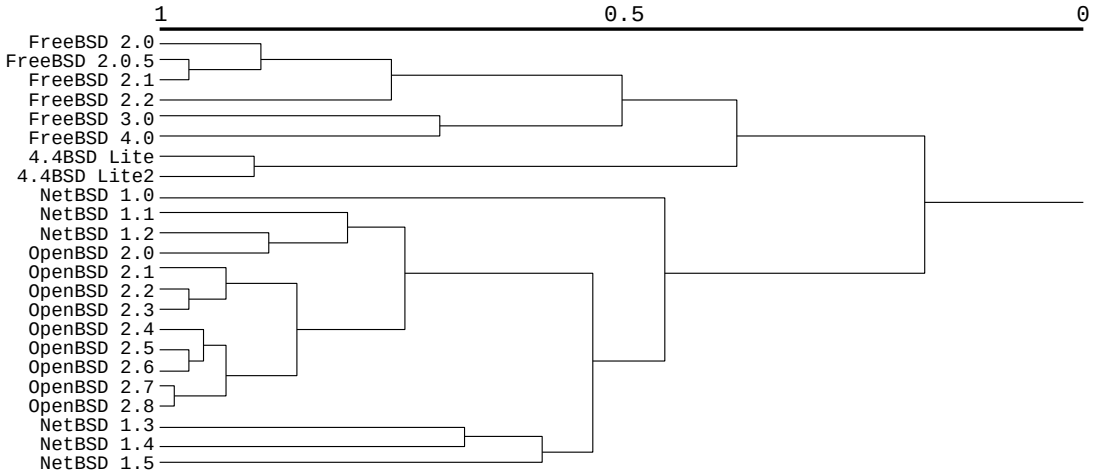

図 4.8: 類似度 S_{fn} を用いた樹状図

表 4.4: 学生実験の S_{fn}

	A	B	C	D	E	F	G	H
A	1	0	0	0.113	0	0	0	0
B	0	1	0.666	0.125	0.600	0.666	0.105	0.428
C	0	0.666	1	0.137	0.857	1	0.125	0.545
D	0.113	0.125	0.137	1	0.133	0.137	0.102	0.117
E	0	0.600	0.857	0.133	1	0.857	0.235	0.500
F	0	0.666	1	0.137	0.857	1	0.125	0.545
G	0	0.105	0.125	0.102	0.235	0.125	1	0.190
H	0	0.428	0.545	0.117	0.500	0.545	0.190	1

出すべきシステムの実行ファイル名が指定されているが，ソースファイルの名前は指定されていない．収集できた計 8 人分のシステムに対する総当たりでの S_{line}, S_{fn} の計測結果をそれぞれ表 4.3 と表 4.4 に示す.

S_{line} が高いシステムは他の学生との同一コードが多数含まれていることを意味する．また，S_{fn} が高いシステムは同一ファイル名が多数含まれていることを意味する．各 S_{line} と S_{fn} の相関は-0.004 と低い．本演習では，提出する実行ファイル名が決められているため，ソースファイルのファイル名も実行ファイルと同じファイル名を付けている学生がおり，それらの学生同士では，S_{fn} が高い値を示す．つまり，学生のファイル命名規則が同じであれば，高い類似度を示すことになり，S_{fn} は必ずしもシステムの類似度を表しているとはいえない．表 4.3 に示す S_{line} の値を見ると，学生 G と学生 H の間の類似度が 0.797 であり他の学生同士の類似度と

表 4.5: バージョン間のリリース間隔（ヶ月）

	FreeBSD 2.0	FreeBSD 2.0.5	FreeBSD 2.1	FreeBSD 2.2	FreeBSD 3.0	FreeBSD 4.0
FreeBSD 2.0	0	7	12	28	47	64
FreeBSD 2.0.5	7	0	5	21	40	57
FreeBSD 2.1	12	5	0	16	35	52
FreeBSD 2.2	28	21	16	0	19	36
FreeBSD 3.0	47	40	35	19	0	17
FreeBSD 4.0	64	57	52	36	17	0

表 4.6: リリース間隔との相関

S_{line}	S_{fn}	行数差	$(1 - S_{line}) \times$ 総行数
-0.973	-0.964	0.937	0.911

比べると非常に高い値である．しかし，S_{fn} は 0.190 と低い．実際，学生 G と学生 H のソースコードを見比べると非常に似ており．S_{line} がシステムの類似度を反映しているといえる．

4.4.5 リリース間隔と類似度の相関

図 4.6 に示したリリース時期と類似度との間に相関があるかどうか計算をした．FreeBSD のリリース間隔を計算した結果を表 4.5 に示す．表 4.6 にリリース間隔と 4 つの値との相関を求めたものを示す．4 つの値とは，類似度 S_{line}，類似度 S_{fn}，二つの OS の総行数の差，$(1 - S_{line}) \times$ 二つの OS の総行数 である．$(1 - S_{line}) \times$ 二つの OS の総行数 は，二つの OS 間で一致しない行数を表す．

この結果から，いずれの値ともリリース間隔と高い相関を持つことが分かる．特に S_{line} との相関は最も高い値を持つ．そのため，ある二つのバージョンの S_{line} を計測することで，そのリリース間隔を知ることが可能である．また，S_{line} は，言い換えるとソースコードの変更していない部分の割合を示しているため，相関が高いということは，開発が滞りなく一定に行われているといえる．

4.5 結論と課題

本章では，ソースプログラムとして与えられた二つのソフトウェアシステムの類似度を定義し，それを計測するメトリクス S_{line} の提案を行った．実際に SMMT

を作成し，種々のソフトウェアシステムに適用し，その有効性を確認した．さらに，得られた類似度を基にして，クラスタ分析を行い，各 OS のバージョンを分類し，バージョンの樹状図を作成した．その結果，提案する類似度メトリクスを用いて作成した樹状図は，OS の分類を派生通りに表していることが分かった．具体的には，類似度はクローン検出ツール CCFinder と diff コマンドを用いて対応を持つ行を検出し計算される．CCFinder だけを用いた手法では，対応を持つ行は完全に検出することができず，diff だけを用いた手法では，多くの時間を要するといった問題がある．そのため，CCFinder と diff を組み合わせて対応を求めることで，より有効な対応する行を検出することが可能になる．

そして，実際に類似度を FreeBSD，NetBSD，OpenBSD といった種々の UNIX OS のいくつかのバージョンに適用した．適用結果から，類似度は二つのソフトウェアシステム間の類似性を正しく表していると考えることができる．さらに，得られた類似度を基にして，クラスタ分析を行い，各 OS のバージョンを分類し，バージョンの樹状図を作成した．その結果，提案するメトリクスを用いて作成した樹状図は，OS の分類を派生通りに表していることが分かった．また，二つのバージョンの間の S_{line} と開発期間の間に強い相関があることも確認できた．

S_{line} は類似度を表す値であるが，S_{line} を計算する過程で各行の対応を求めている．この各行の対応を用いると，実際にファイルのどの行が他のファイルのどの行と同じであるか知ることが可能である．S_{line} の値だけでなく，これらの行の対応の情報もソフトウェアシステムを参照，改変，保守する場合に役に立つと考えられる．

今後の課題として，さらなる類似度の妥当性の検証，類似度と再利用プログラミングとの関係の計測などが挙げられる．

第5章　むすび

5.1　まとめ

本研究では，ソフトウェアの構築や保守における版管理とソフトウェア再利用に着目した．既存の手法の問題点を解決し，版管理とソフトウェア再利用の支援を目的とした三つのシステムを提案し，構築した．

版管理は，バージョン（改訂毎に作成されるプロダクトのこと）に基づいてプロダクトの作成や変更作業の逐次記録や追跡を行う．これらを目的とした様々なモデルが提案され，そのモデルに基づくバージョン管理システムが実装されている．しかし，既存のバージョン管理システムでは，バージョンの保存や取り出しなどを行う際には各ユーザがそれらの命令をシステムに指示する必要がある．また，バージョンの保存はユーザの恣意的な作業で行われるため，その作業を忘れてしまう，あるいは，保存間隔がユーザによって異なるといった問題点がある．そこで，ソフトウェア開発中にソフトウェア開発者の作成するプロダクト（ファイル）の全てを自動的に採取する新たなバージョン管理ファイルシステム Moraine を提案し，構築した．Moraine を用いることで，開発中のプロダクトに関する全ての履歴を保存することが可能になった．全ての履歴を保存するため，過去のどの時点でのソフトウェアにも復元可能になった．また，バージョンの保存作業はシステムが自動で行うため，Moraine を導入したとしても，開発者の作業に影響を及ぼさないという特徴を持つ．さらに，Moraine を導入した場合のファイルの読み出し，書き込みの処理時間を計測し，導入する前の処理時間と比較を行い実用的であるか評価した．その結果，導入したとしても十分に高速な動作が行え，実用的なシステムを構築することができた．

また，Moraine で収集された履歴を基にメトリクスを計測するシステム MAME を構築した．MAME は，開発者の開発環境を変更することなく，メトリクスデータを収集可能である．そして，MAME を導入して開発を行うと，開発作業後でもファイルの開発者，行数，バージョン数，日時などを指定し，ファイルに関するメトリクスデータを，細かい粒度で取得可能になる．取得するメトリクスをソフトウェア開発を始める前に決定する必要がなく，開発の進行中や終了後に，メトリクスデータを決定し，収集することができる．さらに，MAME をある学生のプ

ログラム演習に適用し，MAME から得られたメトリクスデータを用いて学生の活動の分析を行った．MAME を用いることでメトリクスデータの収集及び解析は容易なものとなることが分かった．解析したメトリクスデータを用いることで，ソフトウェアプロセスの改善を行い，ソフトウェアの開発を支援することが可能になる．

最後に，ソフトウェア再利用を支援するシステムの構築を行った．再利用では，既存のソフトウェアの中から再利用可能な部品を検索し，検索した部品に変更を加えて利用することでソフトウェアの生産性，信頼性，性能の向上を行う．しかし，既存の研究では，コンポーネント，ライブラリといった部品を対象にしており，より大規模なソフトウェアシステム単位での再利用は考慮してない．同種のソフトウェアを開発し，その派生が分からなくなると，新たなソフトウェアを開発する際，過去に開発したどのソフトウェアを利用し開発すればよいかを見極めるのが困難になる．そのため，開発したソフトウェアがどのように派生してきたかを自動的に調べることができれば，再利用ソフトウェアを検索する作業の効率の向上が期待される．そこで，二つのソフトウェアシステムの類似度を計測し，計測した類似度からソフトウェアの派生の様子を調べるシステム SMMT の構築を行った．SMMT では，類似度はクローン検出ツール CCFinder と diff コマンドを用いて対応を持つ行を検出し，類似度の計算を行う．計測した類似度からクラスタ分析を行うことで，ソフトウェアの複数のバージョンを分類し，そのソフトウェアのバージョンの樹状図を作成することが可能になる．樹状図から過去に開発したソフトウェアの派生の様子が分かり，新しくソフトウェアを開発する際に，どのソフトウェアの利用すればよいかの目安になる．

5.2 今後の研究方針

今後は，ソフトウェアの再利用をより容易に利用できる環境の構築を行いたい．ソフトウェアの部品化や再利用といった研究は，ソフトウェアの改変を行う際に既にあるソフトウェアを利用して今後のソフトウェアをどのように進化させるといった研究になる．そこで，ソフトウェアの部品化や再利用に着目したソフトウェア進化を支援する方法論やシステムの設計，構築が今後の研究として挙げられる．既存のソフトウェアを整理し，部品化，再利用する際，本研究で提案した SMMT を用い類似度が高いものを部品化し再利用することで，より品質の高いソフトウェアを低コストで開発できると考えられる．

参考文献

[1] A. Aiken. Moss (measure of software similarity) plagiarism detection system. “http://www.cs.berkeley.edu/ moss/”.

[2] Atria Software Inc. ClearCase product summary. Technical report, Atria Software Inc., 24 Prime park Way, Natick, Massachusetts 01760, 1994.

[3] W. A. Babich. *Software Configuration Management*. Addison-Wesley, Reading, Massachusetts, 1986.

[4] V. R. Basili, G. Caldiera, and H. D. Rombach. Goal Question Metric Paradigm. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, Vol. 1, pp. 528–532. John Wiley & Sons, 1994.

[5] V. R. Basili and H. D. Rombach. The TAME Project: Towards improvement–oriented software environments. *IEEE Transactions on Software Engineering*, Vol. SE-14, No. 6, pp. 758–773, June 1988.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pp. 368–378, Montpellier, France, 1998.

[7] S. Carson and S. Setia. Optimal write batch size in log-structured file systems. *Computing Systems*, Vol. 7, No. 2, pp. 263–281, Spring 1994.

[8] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, Vol. 30, No. 2, pp. 232–280, June 1998.

[9] Diffutls. “http://www.gnu.org/software/diffutils/diffutils.html”.

[10] H. Ehrig, W. Fey, H. Hansen, M. Löwe, and D. Jacobs. Algebraic software development concepts for module and configuration families. In *Proceedings of*

the 9th Conference on Foundation of Software Technology and Theoretical Computer Science, Vol. LNCS 405, pp. 181–192, Bangalore, India, December 1989. Springer-Verlag.

[11] J. Estublier and R. Casallas. The Adele configuration manager. In Walter Tichy, editor, *Configuration Management*, pp. 99–133. John Wiley and Sons, Ltd., Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1994.

[12] B. S. Everitt. *Cluster Analysis*. Edward Arnold, 3rd edition, London, 1993.

[13] P. H. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, March 1991.

[14] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, Vol. 31, No. 3, pp. 288–298, March 1988.

[15] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, Vol. 12, No. 1, pp. 58–89, February 1994.

[16] J. K. Hubbard. RELEASE NOTES FreeBSD Release 3.0-RELEASE. “http://www.freebsd.org/releases/3.0R/notes.html”.

[17] A. John. Dynamic vnodes – design and implementation. In *Proceedings of the USENIX 1995 Technical Conference*, pp. 11–23, New Orleans, LA, USA, January 16–20 1995.

[18] 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及び評価実験”. 電子情報通信学会技術研究報告、SS2000-42〜52, Vol. 100, No. 570, pp. 41–48, 2001.

[19] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 3–25, February 1992.

[20] D. G. Korn and E. Krell. A new dimension for the Unix file system. *Software–Practice and Experience*, Vol. 20, No. S1, pp. 19–34, June 1990.

[21] E. S. Laymond. The cathedral and the bazaar. “http://www.tuxedo.org/%7Eesr/writings/cathedral-bazaar/cathedral-bazaar.ps”.

[22] S. J. Leffler, M. K. McKusick, M. J. karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[23] C. Lewerents and F. Simon. A product metrics tool integrated into a software development environment. In *In Proc. European Software Measurement Conference FESMA98*, pp. 603–608, 1998.

[24] Linux Online. “http://www.linux.org/”.

[25] M. Matsushita and M. Oshita and H. Iida and K. Inoue. Conceptual issues of object-centered process model. In *Asia-Pacific Software Engineering Conference*, pp. 519–520. IEEE Computer Society Press, December 1997.

[26] 松下誠, 飯田元, 井上克郎. ”オブジェクトモデルを用いたソフトウェアプロセスの表現方法”. 電気学会論文誌 C, Vol. 118-C, No. 12, pp. 1786–1791, December 1998.

[27] M. Matsushita, H. Iida, and K. Inoue. Web-based Process Management System with Object-Centered Process Modeling. In *The International Symposium on Internet Technology*, pp. 92–97, April 1998.

[28] K. McCoy. *VMS File System Internals*. Digital Press, 1990.

[29] M. K. McKusick, K. Bostic, M. J. karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD UNIX Operating System*. Addison-Wesley, 1996.

[30] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pp. 181–197, August 1984.

[31] Merant, Inc. Merant pvcs version manager & configuration builder. “http://www.merant.com/products/pvcs/”.

[32] Microsoft, Inc. Visual source safe. “http://msdn.microsoft.com/ssafe/”.

[33] W. Miller and E. W. Myers. A file comparison program. *Software- Practice and Experience*, Vol. 15, No. 11, pp. 1025–1040, 1985.

[34] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, Vol. 1, pp. 251–256, 1986.

[35] 長橋賢児. ”類似度に基づくソフトウェア品質の評価”. 情報処理学会研究報告 2000-SE-126, Vol. 2000, No. 25, pp. 65–72, 2000.

[36] OpenBSD. “http://www.openbsd.org/”.

[37] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. In *Proceedings of the USENIX 1995 Technical Conference*, pp. 25–33, New Orleans, LA, USA, January 16–20 1995.

[38] L. Prechelt, G. Malpohl, and M. Philippsen. Jplag: Finding plagiarisms among a set of programs. Technical Report 2000-1, Fakultat fur Informatik, Universitat Karlsruhe, Germany, 2000.

[39] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 4, pp. 364–370, December 1975.

[40] M. Russell and M. Bush. Introduction to metkit. *Journal of Information and Software Technology*, Vol. 35, No. 2, pp. 108–110, 1993.

[41] W. Schneider. The unix system family tree: Research and bsd. “ftp://ftp.freebsd.org/pub/FreeBSD/branches/-current/src/share/misc/bsd-family-tree”.

[42] The FreeBSD Project. “http://www.freebsd.org/”.

[43] The NetBSD Foundation Inc. “http://www.netbsd.org/”.

[44] W. F. Tichy. RCS – a system for version control. *Software–Practice and Experience*, Vol. 15, No. 7, pp. 637–654, July 1985.

[45] K. Torii, K. Matsumoto, K. Nakakoji, Y. Takada, S. Takada, and K. Shima. Ginger2: An environment for caese (computer-aided empirical software engineering). *IEEE Transactions on Software Engineering*, Vol. 25, No. 4, pp. 474–492, 1999.

[46] E. Ukkonen. Algorithms for approximate string matching. *INFCTRL: Information and Computation (formerly Information and Control)*, Vol. 64, pp. 100–118, 1985.

[47] K. L. Verco and M. J. Wise. YAP3 : Improved detection of similarities in computer program and other texts. In *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, pp. 130–134, New York, 1996.

[48] 山本哲男, 松下誠, 井上克郎. ”monoprocess を用いた開発作業連絡支援システムの提案”. 情報処理学会研究報告 98-SE-120, Vol. 98, No. 64, pp. 11–18, July 1998.