ソフトウェアの共同開発作業における細粒度ロックの有効性の検証

A Study on the Validity of the Fine Grain Locking Mechanism in Cooperative Software Development Activities

佐藤 友章 † 杉山 安洋 ‡
Tomoaki SATO Yasuhiro SUGIYAMA

† 日本大学大学院工学研究科情報工学専攻
Graduate School of Engineering, Nihon University
tomoaki@ssl.ce.nihon-u.ac.jp

‡ 日本大学工学部情報工学科
Department of Computer Science, College of Engineering, Nihon University
sugiyama@ce.nihon-u.ac.jp

ソフトウェアの共同開発を支援するシステムでは，編集作業の競合などを回避するため，共有資源の排他制御が必要不可欠である．しかしながら，既存の排他制御方式は，編集作業の並行性が低いなどという問題があり，オープンソースによる開発の支援に十分であるとは言えない．本稿では，既存の排他制御方式の問題を解消する方式として細粒度ロックを提案し，その有効性をオープンソースによる開発プロジェクトを例にして検証する．

1 はじめに

インターネットに代表されるネットワーク環境が社会に浸透しつつある現在，ネットワークを介したソフトウェアの共同開発もさほど珍しいことでない．とりわけ，オープンソースによる開発では，不特定多数のプログラマが自由に開発に参加できることが前提で，それを受け入れられる一種の“ 寛容さ ”が求められる．それは，あらかじめ厳密に設計などが行われず，あくまでプログラマの自由な開発を束縛することなく，それでもなお開発が支障をきたさないということである．

私たちは，Java によるソフトウェアの開発において，クラスやインタフェースの関連，そしてクラスに定義されているフィールドやメソッドなどの視覚的な理解を補助し，プログラムの構造に基づく編集作業を支援するシステムとして，ClassFactory [2] を開発している．現在，ClassFactory は，ネットワークを介した共同開発を支援するために，マルチユーザ化が進められており，既に初期バージョンが完成している．図 1 に初期バージョンの GUI を示す．

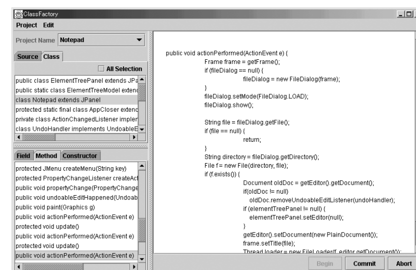

図 1: ClassFactory の GUI

ClassFactory のマルチユーザ化に伴い，複数のプログラマによって共有されるプログラムの同じ部分が同時に編集されるという編集作業の競合が発生する可能性が生じた．そうした競合に対する排他制御方式として，初期バージョンの ClassFactory ではファイルのロックを採用しているが，ファイルのロックでは開発作業を円滑かつ柔軟に行うことは難しい．

本稿では，ファイルのロックを含めた既存の排他制御方式の問題を明らかにするとともに，その問題点を解決する細粒度ロック [5] [6] の概要について述べ，その有効性についてオープンソースによる開発を対象に検証する．

2 既存の排他制御方式

2.1 ファイルのロック

初期バージョンの ClassFactory で採用したファイルのロックは，SCCS[4] や RCS[7] などのバージョン管理システムで確立された排他制御方式である．

初期バージョンの ClassFactory は，クライアントサーバシステムという形で構成した．編集対象のソースコードは，サーバ上のデータベースで一括管理されている．編集作業を始めるとき，編集したいファイルのチェックアウト（サーバ上のファイルをクライアントのデータベースにコピー）を行い，サーバ上のファイルをロックする．それにより，他のプログラマは編集目的でそれをチェックアウトすることができなくなる．そして，編集作業を終えると，編集したファイルのチェックイン（サーバ上のファイルをクライアントで編集されたファイルに更新）を行い，ファイルのロックを解除する．ロックが解除されたファイルは自由にチェックアウトすることができる．

しかし，ファイルを粒度としてロックするため，同一ファイル中の異なる場所を編集しようとしても同時に行えないため，編集作業の並行性が低いという問題がある．

2.2 マージ

マージは，厳密には排他制御方式でないが，競合を解消する方式の 1 つとして，CVS[3] などでも採用されている．

マージによる競合の解消では，プログラマがファイルをチェックアウトするとき，ファイルをロックしない．従って，複数のプログラマが同じファイル，さらにプログラムの同じ部分でも並行して編集することができる．ただし，編集したファイルをチェックインする際，その編集作業によって競合が発生した場合は，他のプログラマによる編集作業と自分の編集作業とをマージすることで競合を解消することになる．

確かにマージによる競合の解消では，編集作業の並行性は非常に高いものの，プログラムの意味まで考慮されたマージを自動化することは難しい．従って，競合が発生した場合，最終的にプログラマ自身が手動でマージしなければならず，プログラマへの負担が大きくなるというという問題がある．

また，競合の発生はチェックインするまで分からないため，プログラムの同じ部分を複数のプログラマが編集した場合，重複した作業の一部が無駄になることもある．

3 細粒度ロックの概要

細粒度ロックは，ロックの粒度を細かくすることによって，複数のプログラマが同時に編集できるプログラムの範囲を広めながら，あくまで競合を未然に回避できる排他制御方式である．細粒度ロックの粒度はプログラムの構文要素である．ClassFactory は Java による開発を支援するため，クラスやインタフェース，クラスに定義されるフィールドやメソッドなどが含まれる．

構文要素は構文木という階層構造をなすため，構文要素をロックするとき，それに含まれている構文要素も再帰的に全てロックの対象になる．例えば，クラスをロックするとき，そのクラスに定義されているフィールドやメソッドもロックされる．また，同時に複数の構文要素をロックすることも可能である．

細粒度ロックは，構文要素を粒度としてロックを行うため，ファイルのロックにおける編集作業の並行性の低さを解消できる．例えば，図 2 に示すように，プログラマ A がファイルの一部を編集する場合は，それに相当する構文要素だけロックされる．従って，別のプログラマ B は A と異なる部分であれば，同じファイル内であってもロックして並行して編集作業を行うことができる．

なお，ソフトウェアを構成する構文要素は相互に関連しており，ある構文要素に対する編集作業が，それだけで問題がなくとも，全体として矛盾を発生させることがある．例えば，クラスを編集するとき，そのスーパークラスが他のプログラマによって編集されている場合，スーパークラスのフィールドやメソッドの削除，シグネチャの変更により，結果としてサブクラスにおけるフィールドの参照やメソッドの呼び出しが矛盾することになる．こうした矛盾を回避するため，細粒度ロックでは構文要素の関連に基づくロックとして，構文要素を編集する際にあらかじめ関連する構文要素のロックが行える．

4 細粒度ロックの有効性の検証

細粒度ロックの有効性を検証するため，オープンソースによるソフトウェアの開発を対象に調査した．本来であるならば，ClassFactory が Java による開発を支援することから，Java を対象にすべきところで

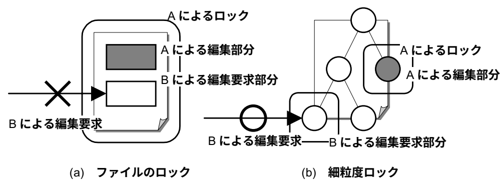

図 2: ファイルのロックと細粒度ロック

あるが，適当な開発が見当たらなかったため，C による開発を対象として調査した．今回はその調査の中から，GNU プロジェクトによる Emacs 20.7 の開発に関する調査の結果について報告する．

Emacs は，CVS を用いて開発されており，今回の調査では，ソースプログラムとともに配布される ChangeLog というログファイルをもとに調査を行った．ChangeLog には，プログラムが編集された日付，プログラマの名前，そして変更箇所としてファイルの名前，関数や変数などの名前が記述されている．

4.1 ファイルのロックに対する優位性

CVS による編集作業では，もともとロックは行われないため，ある編集作業が他の編集作業と競合するために作業が行えないという問題は発生しない．しかし，実際にはマージが必要な編集作業の競合が発生していた可能性がある．そこで，ログと同じスケジュールでロックを行って作業を行ったと仮定し，先行する編集作業と競合してロックが行えなくなる作業の割合を調査した．ロックの粒度としては，編集対象の関数のみをロックした場合と，編集対象の関数を含むファイル全体をロックした場合について調査した．

図 3 は，ログ中の編集作業のロック期間がすべて同一であると仮定した場合に，ロック期間と競合の発生率の平均の関係をグラフとして表したものである．なお，同一プログラマによる同一関数の編集は，たとえ開発期間が重なっても競合とは見なさないこととした．これは，ロック期間の延長として考えることができるためである．

同図に示されるように，ファイルを粒度としてロックした場合，編集期間が延びるにつれ競合が推測される編集作業の占める割合が明らかに高くなることが分かる．それに比べ，関数をロックの粒度とする場合，割合が高くなることは確かであるが，その傾きはなだらかである．

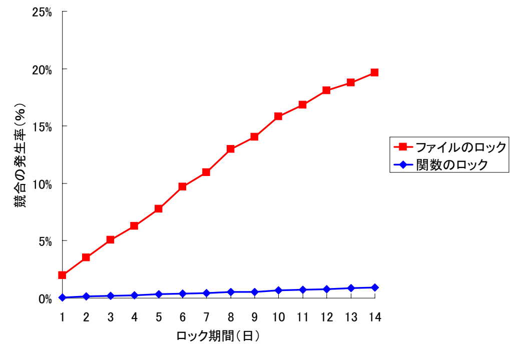

図 3: ロック期間と競合の発生率の関連

期間を 1 日つまり同じ日付に複数の編集作業が行われた場合は，ファイルをロックした場合と関数をロックした場合との差は約 2 %である．しかし，ロックの期間を 2 日，3 日に延ばすにつれ，その差は大きく広がる．期間を 14 日としてファイルを単位にロックが行われた場合，編集作業のおよそ 20 %に競合が推測されることになる．従って，競合を回避するため，編集作業の 5 件に 1 件を拒否することになり，編集作業の並行性が低下することは否めない．しかし，関数を単位としてロックを行った場合には，拒否する必要がある編集作業の要求は少なくとも数%以下に抑えることが可能である．つまり，細粒度ロックはファイルのロックより高い並行性で編集作業を行うことができる．

さらに図 4 は，ロックの期間を，編集対象となる関数の粒度に合わせて変更した場合の，変更対象の関数の粒度（行数）と競合の発生率の関係を表したグラフである．なお，ロックの期間を求めるにあたっては， COCOMO[1] を用いた．COCOMO を用いると，開発対象のコンポーネントの行数をもとに，その開発に必要な工数や開発期間を求めることができる．ここでは，編集対象となる関数の行数をもとに，その開発期間を求め，それをロックの期間として採用した．なお，ロックの期間は，COCOMO におけるコーディング工程のみを対象とし，設計等の工程に必要な期間は除いてある．

このグラフからわかることは，まず，ファイルのロックよりも関数のロックの方が，競合が発生する可能性が低いことがわかる．さらに，関数のロックでは，多少ばらつきは見られるものの，プログラムの行数が少ないほど競合の発生率も少ないことが分

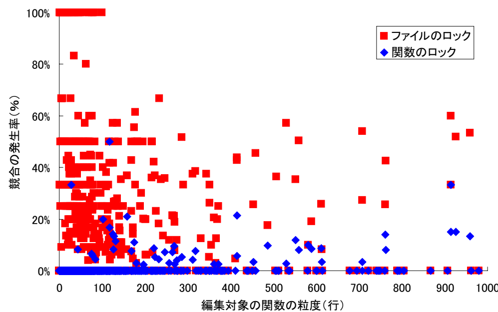

図 4: 関数の粒度と競合の発生率の関連

かる．なお，ファイルのロックを行った場合には，関数が小さいものの方が，競合の発生率が高い場合もあるという結果も得られている．これは，関数が小さいと，ひとつのファイルが多くの関数を含むことができるため，ロックの競合が発生しやすくなるためであると考えられる．

4.2 マージに対する優位性

今回の解析対象としている CVS のログにおいては，編集完了後のチェックイン時に競合の存在が判明し，それを回避するためのマージ作業が行われた可能性がある．これを，ロックを行う場合の状況に換言すれば，ロックの競合が発生している場合に他ならない．しかし，その実態はログには記述されていない．そこで，ログのデータから，ロックを行うと仮定した場合のロックの競合の発生率をより詳細に分析してみた．

COCOMO では，システムの新規開発のみならず，変更作業に関する工数や作業期間を求めることができる．例えば，あるコンポーネントを既存のコンポーネントを変更して開発する場合，既存のコンポーネントの 行数 $ADSI$ [1]に対して，$EDSI = (0.4DM + 0.3CM + 0.3IM) \times ADSI$ を変更作業と等価な開発行数[2]として用いる．ここで， DM は設計に変更が必要な割合，CM はコーディングに変更が必要な割合，IM はシステム統合のために変更が必要な割合である．この $EDSI$ を用いて，開発工数や開発期間を計算する．

A と B という二人のプログラマが編集作業を行ったとする．これらの作業に競合が発生しない場合には，プログラマ A に対しては，$EDSI_A =（0.4DM_A + 0.3CM_A + 0.3IM_A）\times ADSI$ ，プログラマ B に対しては，$EDSI_B =（0.4DM_B + 0.3CM_B + 0.3IM_B）\times ADSI$ で，等価な開発行数が求められる．なお，DM_A や DM_B 等における添え字は，作業をしたプログラマを表している．ところが，マージが必要な場合は，状況が変わってくる．A がまず編集作業を完了し，続いて B が作業を完了した段階で，競合の発生が判明し，マージが必要となった場合を仮定する．この場合，A については変わらないが，B については，もともとの B の作業 $(0.4DM_B + 0.3CM_B + 0.3IM_B) \times ADSI$ に加え，A の行った作業に変更を加える作業 $(0.4DM_{AB} + 0.3CM_{AB} + 0.3IM_{AB}) \times ADSI$ と，B 自身の行った作業に変更を加える作業 $(0.4DM_{BB}+0.3CM_{BB}+0.3IM_{BB})\times ADSI$ がさらに必要となる．ここで，評価を簡単にするために，少々乱暴ではあるが，すべての変更の割合を 20 %と仮定すると，マージを行わない場合の等価な編集行数は，$EDSI_{nomerge} = 0.2ADSI$ となり，一方マージを行う場合の等価な編集行数は，$EDSI_{merge} = 0.6ADSI$ となる．

これを，今回の調査に当てはめれば，前者が細粒度ロックを用いてマージの必要を無くした場合に相当し，後者が CVS を用いてマージを行いながら作業をした場合に相当する．また，今回の場合には，$ADSI$ は変更対象となる関数の行数である．これをもとに，作業期間を計算すると，マージを行った場合の方が，作業期間が長くなるため，ロックの必要な期間も長くなる．このロック期間を用いて，競合の発生率を求めたグラフが図 5 である．グラフでは，横軸に編集対象の関数の粒度，縦軸に競合の発生率をプロットしている．なお，粒度が小さい関数が多いため，それらの状態を詳しく表示するため，横軸には今回は対数メモリを用いている．

このグラフからわかることは，マージを行うためにロックの期間が長くなることにより，競合の発生率が 2 倍近くに増加している編集作業が数多く存在することがわかる．これが，細粒度ロックのマージに対する優位性にほかならないと考えられる．

4.3 調査結果の考察

これまでの調査で，より細かいロックの粒度を採用することで，共同開発における編集作業の並行性を向上させることができることを検証した．それに

[1] Adapted Delivered Source Instructions
[2] Equivalent numbers of Delivered Source Instructions

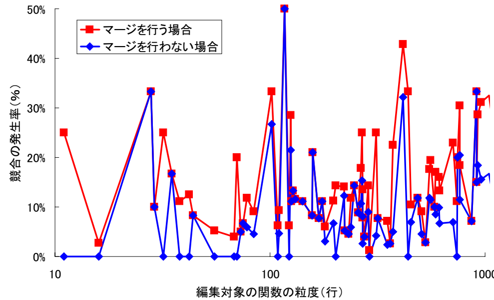

図 5: マージを考慮した競合発生率

よって，細粒度ロックの方が，ファイルのロックより高い並行性を確保できるため，優れていることが証明できた．しかしながら，細粒度ロックはロックを行う以上，既にロックされたプログラムに対する編集作業の要求は受け付けられない．従って，CVSなどで採用されるマージによる競合の解消に比べ，編集作業の並行性という面で劣ることは否めない．しかし，マージによる競合の解消には，競合が発生した場合，プログラマが自分で編集作業をマージしなければならないという，本来の編集作業とは別の作業があることを考慮する必要がある．今回，マージのための編集期間の増加を考慮して評価を行ったが，ロックを細粒度で行って，マージを行わない方が，結果的には良い結果が予測できることが観察できた．

5 まとめと今後の課題

本稿では，ソフトウェアの共同開発の支援における排他制御方式について検討し，細粒度ロックの概要を述べた．そして，細粒度ロックを用いた編集作業では，編集作業の並行性を確保しながら，競合の発生を未然に回避できることを，オープンソースによる開発の調査をもとに検証した．

今後の課題としては，細粒度ロックは現在実装中のため，まず細粒度ロックの実装を完了させることが挙げられる．また，実装においては，ロックの粒度を細かくすることに伴う，ロックするという作業自体の煩雑化を避けるため，ロックを容易に行うための機能およびGUIの提供が必要である．

また，本稿における有効性の検証は，ログを用いた机上の検証であった．そのため，実装が完了した段階で，今回の予測が正しいものであることを，再度確認する必要があると考えている．また，今回はロックの競合の発生によるシステム全体の開発期間の増加については考察することができなかった．競合の発生率のみならず，競合を回避することによる工程の遅延等についても調査する必要があると考えている．さらに，調査対象がCによる開発であったため，オブジェクト指向における継承などの関連が考慮されておらず，主に構文要素のロックに対する検証であった．従って，構文要素の関連に基づくロックの有効性の検証も必要である．

参考文献

[1] B. Boehm, Software Engineering Economics, Prentice-Hall, 1981.

[2] 市原潤, 杉山安洋, ClassFactoryにおけるパターン支援機能の提案. 情報処理学会第 *57* 回全国大会講演論文集, No. 1, pp. 223–224, 1998.

[3] K. Fogel, Open Source Development with CVS, Coriolois Open Press, 1999.

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

[5] 佐藤友章, 杉山安洋, ClassFactoryを用いたグループ開発作業における排他制御方式の検討. ソフトウェア工学の基礎 *VIII*, pp. 83–92, 2001.

[6] 佐藤友章, 杉山安洋, プログラムの構成要素間の関連に基づく細粒度ロックの検討. 第 *44* 回日本大学工学部学術研究報告会講演要旨集, pp. 1–4, 2001.

[7] Walter F. Tichy, RCS - A System for Version Control. *Software - Practice and Experience*, Vol. 15, No. 7, pp. 637–654, 1985.