Vol.2017-SE-195 No.24
2017/3/12

# ファイルへの入出力に基づく実行履歴の構造化手法の提案

加藤 宗一郎[1,a)]　吉田 敦[1,b)]　蜂巣 吉成[1,c)]　桑原 寛明[2,d)]　阿草 清滋[1,e)]

**概要**：コマンドラインを用いた作業では，実行履歴からコマンドを指定して再実行を行うことがあるが，複数のコマンドをまとめて指定したり，それらを複数のファイルに適用するといった応用が難しい．本研究では，コマンドの入出力に用いられたファイルを特定し，実行コマンド間の依存関係やコマンドのオプションを解析し，コマンド列の構造化を行う手法を提案する．これにより，作業の目的に合わせてコマンド列の選択やカスタマイズを可能にする．また，ファイルの入出力はファイルの更新情報を調べることで，コマンドの仕様に依存しない．

Soichiro Kato[1,a)]　Atsushi Yoshida[1,b)]　Yoshinari Hachisu[1,c)]　Hiroaki Kuwabara[2,d)]
Kiyoshi Agusa[1,e)]

## 1. はじめに

一般に，プログラムや文書の作成，サーバシステムの運用テストなどでは，一度の作業で目的を達成することは難しく，成果物や動作状況などを確認しながら繰り返し作業を続けていく．シェルは直前までに実行したコマンドをリスト化した実行履歴を持ち，同じ作業を行いたいときは，コマンドを指定して再実行させることができる．コマンドの指定は，キー操作で履歴を遡ったり，通し番号や検索キーワードの指定など，簡単な入力操作で実現でき，コマンド名の入力やオプションの設定などを省略できるので，効率良く作業を繰り返せる．ただし，選択できるコマンドは 1 つであり，複数のコマンドを組み合わせるときは，各コマンドごとに指定する必要がある．

複数のコマンドを繰り返すときは，Makefile やシェルスクリプトなどの作業指示のためのファイルを作成し，それを用いる方が効率が良い．ただし，コンパイルなどの定型的な作業ではない場合には，作業の内容が明確ではないことも多く，実行しながら手順を考える．一通り実行した時点で，履歴から一連の作業を取り出すことができれば，指示ファイルの作成も容易になる．そこで，実行履歴を構造化し，関係性が強いコマンド群を構成することを考える．すなわち，短い時間間隔で実行したコマンド群は，一つの目的を達成する一連の作業である可能性が高い．また，コマンドの入出力ファイルに着目すれば，コマンド間の依存関係を推測でき，ある目的を達成するために必要なコマンド群を取り出すことができる．これを応用することで，複数の作業の繰り返しを簡単に指定したり，シェルスクリプトなどの作業指示ファイルの雛形を生成できる．

本研究では，コマンド群の再実行やシェルスクリプトの作成を支援する基盤として，実行時間やファイルの更新情報を記録した実行履歴の構造化方法を提案する．また，応用として，一連の作業と推測されるコマンド群を生成できることを示す．なお，実用性の観点から，既存のシェルを，そのシェルが持つ機能を利用して拡張し，履歴情報を残す．本論文では，bash を前提として拡張方法を説明する．

## 2. 実行履歴の構造化

### 2.1 シェルの実行履歴の機能

シェルにおける実行履歴とは，直前までにコマンドラインで指定された指示のリストである．以前に実行したコマンドを再度実行する際には，実行履歴から実行したいコマンドを指定し，実行する．bash は，表 1 の履歴展開機能を提供しており，コマンドライン上で，再実行したい履歴を指定して展開したうえで，実行する．

csh 系のシェルでも同様の機能が存在する．例えば，

[1] 南山大学理工学部
Faculty of Science and Engineering, Nanzan University
[2] 南山大学情報センター
Center for Information, Nanzan University
[a)] 13se075@nanzan-u.ac.jp
[b)] atsu@nanzan-u.ac.jp
[c)] hachisu@nanzan-u.ac.jp
[d)] kuwabara@nanzan-u.ac.jp
[e)] agusa@nanzan-u.ac.jp

表 1　履歴展開機能の一覧

| 指示 | 意味 |
|---|---|
| !$n$ | $n$ 個目のコマンドを参照 |
| !$-n$ | 現在から $n$ 個前のコマンドを参照 |
| !! | 直前のコマンドを参照 |
| !*cmd* | *cmd* から始まるコマンドのうち，現在から最も近くに実行されたコマンドを参照 |
| !?*cmd*? | *cmd* を含むコマンドのうち，現在から最も近くに実行されたコマンドを参照 (*cmd* の直後が改行のとき，後ろの?は省略可能) |
| ^*str1*^*str2*^ | *str1* を *str2* に置き換えて直前のコマンドを繰り返し実行 |
| !# | これまでに打ち込まれたコマンド全体 |

tcsh[6] では，履歴展開と同様の機能として，“History substitution” と呼ばれる機能が存在する．

これらの履歴展開機能は便利ではあるが，複数の作業を繰り返すときの支援は十分ではない．それぞれの展開機能は，1 つのコマンドの履歴を指定して展開するので，複数の処理をまとめてやりたいときに，該当する履歴をすべて指定する必要がある．コマンド名を指定して検索する場合，空白を含むことができないので，オプションが少し異なる処理が履歴に含まれているときには指定できないことがあり，代わりに，1 つずつ通し番号を確認しながら指定する必要がある．

履歴からシェルスクリプトを作成する場合にも，history 命令では，すべての履歴が出力されるので，tail や sed 等のツールで範囲を絞って取り出したり，エディタ内で不要な部分を除去するなどの手作業が必要になる．作業者は，一連の作業は短時間の間に連続して行い，その作業の連続を 1 つの単位と考えて指定をしたいが，シェルの実行履歴には実行間隔に関する情報はないので，実行履歴から自分が考える単位の部分を探す必要がある．例えば，図 1 は，実際の実行履歴に対し，各コマンドを実行するまでに，直前のコマンドから何秒であったかを追加した例である．連続作業を見極めやすくするために，10 秒以上の間隔があった場合に空行を入れている．LaTeX のファイルの句読点の変換とコンパイルおよび PDF への変換までの一連の作業における時間間隔は短く，それ以外の箇所は相対的に長いことが読み取れる．

スクリプトを作成する前は，試行錯誤を行う可能性があり，オプションを確認したり，マニュアルを表示させる，カレントディレクトリ内のファイルの状態を確認するなど，本来スクリプトには含めない作業も実行履歴に含まれる．man や ls など，典型的なコマンドは grep などでパターンを指定して除去はできるが，コマンドのヘルプを表示した場合や，作業の中に ls の結果をファイルに書き込んでいる場合など，パターンを指定できないことがある．作業の多くは，成果物をファイルとして構成することを考えると，ファイルへの書き込みを行わない処理は，試行錯誤の過程で行なった処理であり，繰り返したい処理には含まれないと考える方が妥当である．しかし，実行履歴からは，そのことは判定できない．

```
[    ] $ cd myfolder/gr-thesis/
[  0] $ ls
[  5] $ atom analysis.tex

[ 47] $ cd
[  1] $ ls

[ 18] $ cp task_confirm.tex makefile.tex
[  2] $ vim makefile.tex
[  5] $ platex makefile.tex
[  8] $ dvips makefile.dvi -E -o makefile.eps
[  3] $ cd ../

[197] $ clear

[ 13] $ sed -i '' -e 's/、/，/g' analysis.tex
[  5] $ sed -i '' -e 's/。/．/g' analysis.tex
[  5] $ platex thesis.tex
[  1] $ platex thesis.tex
[  5] $ dvipdfmx thesis.dvi

[114] $ sed -i '' -e 's/、/，/g' analysis.tex
[  2] $ sed -i '' -e 's/。/．/g' analysis.tex
[  2] $ platex thesis.tex
[  2] $ dvipdfmx thesis.dvi

[ 23] $ sed -i '' -e 's/、/，/g' analysis.tex
[  2] $ sed -i '' -e 's/。/．/g' analysis.tex
[  2] $ platex thesis.tex
[  2] $ dvipdfmx thesis.dvi

[380] $ sed -i '' -e 's/、/，/g' analysis.tex
[  3] $ sed -i '' -e 's/。/．/g' analysis.tex
[  1] $ platex thesis.tex
[  2] $ dvipdfmx thesis.dvi

[ 72] $ cd fig
[  4] $ ls -F
[  7] $ mkdir history_log
[  4] $ cd history_log/
[  5] $ pwd
```

図 1　作業の時間間隔の例

Makefile を作成する場合は，ファイルの生成の関係に基づいて処理を分ける必要があるが，実行履歴にはファイルの参照や書き込みに関する情報は含まれないので，作業者がコマンドが持つファイルに関する処理を意識しながら区別する必要がある．

### 2.2　実行履歴の拡張

複数のコマンドからなる作業を実行履歴から取得するには，少なくとも以下の情報が必要となる．

- コマンドの実行開始時刻
- コマンドの実行終了時刻
- コマンドが参照したファイル

- コマンドが書き込んだファイル

これらの情報の取得方法については，後述する．

これらの情報を取得するうえで，実用性の観点から次の前提を置く．

- ユーザは通常通りシェルを使い，コマンドを起動するたびに情報取得のための専用コマンドの指定しない．
- コマンドの機能に関する前提知識はなく，任意のコマンドに対して使えるようにする．
- 既存のシェルを，それ自体が持つ機能で拡張し，シェルのソースコードを変更するといった専用のシェルの作成は行わない．

コマンドの実行時間は，time コマンドのように，引数で指定されたコマンドを子プロセスで実行し測定する方法はある．しかし，毎回，専用のコマンドをユーザが指定する方法では，指定し忘れが起きやすく，作業者への負担も考えると，現実的ではない．また，各コマンドの仕様が与えられれば，ファイルの入出力情報はわかるが，そのような情報を事前に用意することも難しく，また，用意したとしても，使用できるコマンドが限定されることになる．よって，シェルの拡張が必要であるが，ソースコードを修正して専用のシェルを作ることはバージョンアップへの追従などの課題があり，運用上，望ましくない．

ここで，bash の機能に着目する．bash には，コマンドの実行開始前と後に別のコマンドを実行する仕組みがある．

- 実行開始前: trap コマンドのシグナル DEBUG
- 実行終了後: PROMPT_COMMAND

これらは，シェルを起動後，コマンドラインから動的に設定が可能である．また，実行開始前と実行終了後のタイミングで起動できるので，そのときの時刻，また，起動したコマンドを記録することも容易である．

ファイルからの読み込みと書き込みについても，コマンドの実行前後のファイルの状態から判定できる．ファイルシステムには，最終アクセス時間と最終修正時間が記録されており，それらの情報の差を見ることで，コマンドの実行時に参照または変更が行われたかどうかがわかる．ただし，この方法では，情報の取得に係る時間を減らすことや，デーモンなどの処理による更新の影響を避けるために，ファイルシステム全体ではなく，カレントディレクトリ以下のファイルに限定するといったことが必要である．また，エディタをバックグラウンドで起動していると，コマンドの実行中にエディタがファイルを保存することがあり，コマンドの入出力を正確に把握できるわけではない．

### 2.3 実行履歴の構造化

実行履歴から，どのような単位で作業を切り出したいかは，ユーザの考えに依存し，また，コマンドの入出力を正確に把握することも難しい．そこで，取得できる情報に基づき，実行履歴を構造化し，各種支援の基盤とする．

図 2 は，実行履歴を構造化したものをクラス図で表現している．実行履歴は，タスクのリストで構成され，さらにタスクは実行コマンドのリストで構成される．タスクは，ある作業を達成するために入力された一連の実行コマンドをまとめたオブジェクトを表す．作業の区切りは，後述するように，基本的には，実行時の時間間隔で決める．

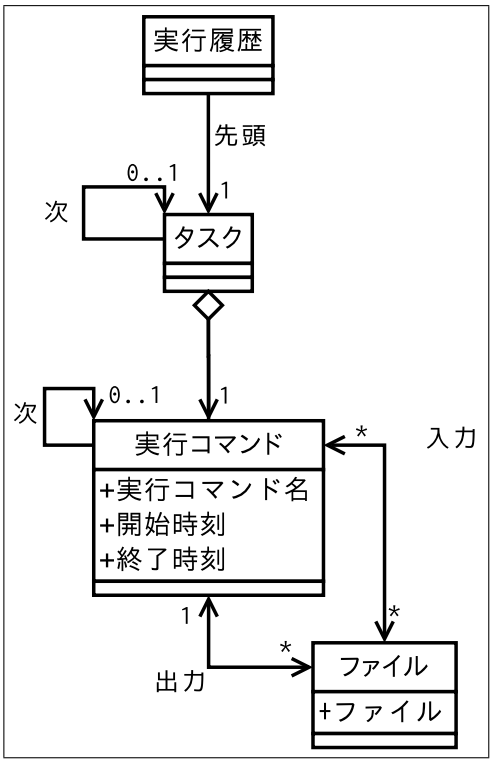


**図 2** 実行履歴の構造

バックグラウンドなどでエディタを起動しているときには，コマンドラインでコマンドを実行中にファイルを更新する可能性がありえる．その場合，コマンドラインで起動したコマンドによる更新として扱われる．ただし，一般的には，コマンドライン内にファイル名が現われることが多く，ファイル名の有無により，ある程度は区別可能である．また，実行コマンド内のファイル名を特定することで，複数ファイルに対して同じ処理を繰り返したいときに，ファイル名を変数に置換したスクリプトに変換できる．

## 3. 構造化実行履歴を用いた解析

構造化した実行履歴を用いた解析として，時間間隔に基づく作業の分割と抽出，コマンド間の依存関係に基づく作業の抽出，コマンド列内のファイル名の変数名置換を示す．これらの解析手法を用いることで，再実行したいコマンド列を生成し，それを実行したり，シェルスクリプトなどの作業指示ファイルを生成できる．

### 3.1 時間間隔に基づく作業の分割と抽出

コマンドラインで行う作業の多くは，ソースとなるファ

イルを出発点として，データの変換を行い，最終結果をファイルに保存をする．その手順が決まっていれば，一連の操作は連続して行われるので，コマンド実行の時間間隔は短くなる傾向にある．そこで，2 つのコマンドの時間間隔が一定時間以下の場合は 1 つの作業を行なっているとみなし，その作業のことを，本論文ではタスクと呼ぶ．図 1 では，時間間隔が 10 秒以上の箇所に空行を入れているが，その空行で区切られた各コマンド列がタスクに相当する．また，図 3 のように，各コマンド列に通し番号を割り振り，実行されたコマンド列とともに表示することで，作業者が一意にコマンド列を指定し，実行できるようになる．

```
[#0] --------------------
  cd myfolder/gr-thesis/
  ls
  cd fig/
  ls
  clear
  ls
  atom thesis.tex
[#1] --------------------
  atom desimplement.tex
[#2] --------------------
  platex thesis.tex
  platex thesis.tex
  dvipdfmx thesis.dvi
```

**図 3** 実行履歴の表示例

問題として，作業者としては，1 つの作業のつもりでも，入力するべきコマンドを検討するために時間を要した場合などに，タスクが分かれ，作業者の意図と合わないことがある．しかし，基準となる時間間隔をそのときの状況に応じて調整することは難しい．そこで，一定時間の時間間隔で区切ったうえで，必要に応じて作業者が補正できるようにする．図 2 で，タスクを表現している理由は，コマンドの区切りを変更することを想定しているからである．すなわち，単純に一定時間で分割するのみであれば，実行履歴を表示するときに分割していけば良いが，それではタスクごとの調整が難しい．そこで，あらかじめ実行履歴をタスクに分けておくことで，2 つのタスクの結合などの処理ができるようにする．

また，タスクの分割が正しいとしても，各タスクが再実行したり，シェルスクリプトに変換する処理そのものになっているとは限らない．作業を進めて行く過程では，入力ファイルの名前を確認するためにカレントディレクトリの内容を表示したり，コマンドのヘルプを表示，あるいは，man でマニュアルを確認するといった，作業を決めるための作業が加わっていることが多い．一般的に，このようなコマンドは，作業自体を進めるものではないので，図 4 に示す**コマンド 1** や**コマンド 4**，**コマンド 6** のように作業で用いるファイルそのものは更新しない．よって，出力ファイルがないコマンドを取り除くフィルタを作ることで，不要な作業を除去できる．

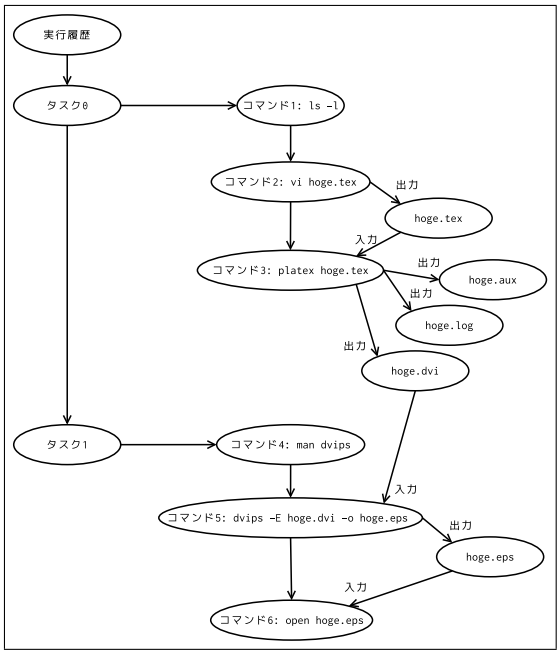


**図 4** 不要なコマンドを含む実行履歴

### 3.2 ファイルの依存関係に基づくコマンド列の生成

成果物の生成作業を繰り返し行う場合，必要なコマンドを取得するためには，生成に必要なファイルを更新するコマンドを取得する必要がある．ファイルの入出力から，作業に必要なコマンドを取得するには，ファイルの依存関係を用いる．例えば，プログラムを作成する際には，ソースコードを編集した後に，コンパイルを行うことで，目的とするプログラムを得る．プログラムの作成作業を繰り返し行うためには，目的プログラムを生成するコンパイル作業から，ファイルの依存関係を用いて，必要なコマンドを取得する．複数のコマンドを実行する際に，ファイルの依存関係をたどることでユーザが生成したいファイルを生成するために必要なコマンド列を連鎖的に取得できる．例えば，図 5 では，`hoge.c` が更新された場合，入力として `hoge.c` を読み込み，`hoge.o` を出力するコマンド `cc -c hoge.c` と，`hoge.o` を入力として読み込み，`prog` を出力するコマンド `cc -o prog main.c *.o` が実行に必要なコマンドとなる．

取得した依存関係の書式を変換することで，Makefile などのスクリプトの生成にも応用できる．

### 3.3 複数のファイルへの適用

複数の画像ファイルに対してサイズや特定の加工処理を

```
$ vi hoge.c
$ cc -c hoge.c
$ vi piyo.c
$ cc -c piyo.c
$ vi main.c
$ cc -o prog main.c *.o
$ vi hoge.c
$
```

図 5　C プログラムのコンパイル作業例

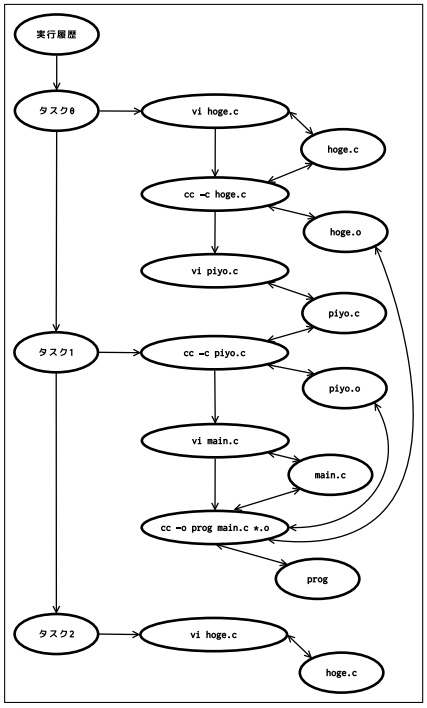


図 6　図 5 の依存関係のモデル

適用する場合，まず 1 つのファイルに適用して効果を確認し，それから，残りのファイルに適用することが多い．このような処理の場合，実行履歴には，最初に用いたファイルに対する処理が残っており，それ自体は他のファイルに適用できない．よって，3.1 節や 3.2 節の手法で得られたコマンド列に対して，ファイル名を変数名に置き換え，for などの繰り返し命令で用いられるコマンド列を生成する．コマンドライン上のファイル名を解析した際に，入力ファイルと同じ拡張子で，異なる名前のファイルが指定された場合，コマンド列を指定されたファイルに関する処理に変換して生成する．ただし，本研究では，ファイル名の拡張子以外の部分が一致する場合のみ支援の対象とする．

コマンド列の入出力に用いられたファイルについて，入力と出力で拡張子が異なるだけで，拡張子以外が同じ名前である場合は，一般にデータの変換処理とみなせる．例えば，図 5 中の，`cc -c hoge.c` や `cc -c piyo.c` のように C プログラムのオブジェクトファイルを出力する場合，入力ファイルの拡張子が`.c` で，出力ファイルの拡張子が`.o` と拡張子が異なっているが，拡張子以外はそれぞれ `hoge`, `piyo` と同じである．

入力ファイルと同じ拡張子で，異なる名前のファイルが指定された場合，コマンド列を指定されたファイルに関する処理に変換して生成する．図 7 は `foo.tex` から `foo.eps` へ変換する処理の一連である．この作業を `bar.tex` を入力ファイルとして指定する場合，図 7 中のファイル名のうち，`foo` を `bar` に置き換えて実行する．本研究では，ファイル名の拡張子以外の部分が一致する場合のみ支援の対象とする．

```
$ vi foo.tex
$ platex foo.tex
$ dvips -E foo.dvi -o foo.eps
$
```

図 7　ファイルの変換作業例

## 4. 実装と評価

### 4.1 システムの構成

実行履歴の取得および解析するシステムを構築した．実行履歴の取得では，bash に履歴取得用の関数を設定し，コマンドや実行時間，ファイルの更新情報をログファイルに書き出す．また，そのログファイルを読み込み，ファイルの依存関係などを解析して，目的に合わせたコマンド列を生成する．

### 4.2 実行履歴の取得

作業者がコマンドを実行するたびに，以下の情報を収集し記録する．

- 実行コマンド
- コマンドの開始時刻および終了時刻
- ファイルの更新履歴

この情報を取得するためには，コマンドの実行前後をトラップする必要があるが，bash の trap と環境変数 `PROMPT_COMMAND` を使用した．実行履歴の取得の機能は，図 8 の処理を source 命令でシェルに読み込ませ，コマンドの実行の前後で情報を取得する．

実行履歴の情報は，シェル関数として定義した `saveHistory` を用いる．`saveHistory` は，コマンド起動前と起動後のそれぞれで呼び出され，起動前はコマンド起動時刻やファイル更新情報の取得の準備を，起動後はコマンドの終了時刻やファイルの更新情報の取得を行い，ログに書き出す．また，一定時間を経過したときは,，1 つの作業が終ったものとして，ログを分割する．なお，分割するタイミングは動的に変更できる．また，この分割されたログがタスクに相当する．

ここで，`PROMPT_COMMAND` で呼び出す関数では何もせ

```
after_hook()
{
:  # nothing to do
}
PROMPT_COMMAND="after_hook"

# record command logs with file dependencies
saveHistory()
{
  if [ "${BASH_COMMAND}" = "${PROMPT_COMMAND}" ]; then
    if [ -z "${CMD_BEGIN_T}" ]; then
      return;
    fi

    CMD_END_T=$(date '+%s')

    # record the times of begining and ending of command execution
    printf "%d %d\t%s\n" ${CMD_BEGIN_T} ${CMD_END_T} "${CMD_LAST_CMD}" \
        >> ${LOG_DIR}/cmpstamp.log

    # check if the command is in the current task
    if [ -n "${CMD_LAST_T}" ]; then
      if (( CMD_BEGIN_T - CMD_LAST_T > CMD_TASK_SEP_T )); then
        (( CMD_COUNTER++ ))
      fi
    fi

    # record file dependencies with executed command
    (
      printf "%d %d\t" ${CMD_BEGIN_T} ${CMD_END_T}
      getHistory -c ${LOG_DIR}/criterion -d ${PWD} "${CMD_LAST_CMD}"
    ) >> ${LOG_DIR}/task${CMD_COUNTER}

    CMD_LAST_T=${CMD_END_T}
  else
    CMD_BEGIN_T=$(date '+%s')
    touch ${LOG_DIR}/criterion
    CMD_LAST_CMD=$BASH_COMMAND
  fi
}

trap saveHistory DEBUG
```

**図 8**　実行履歴の記録処理

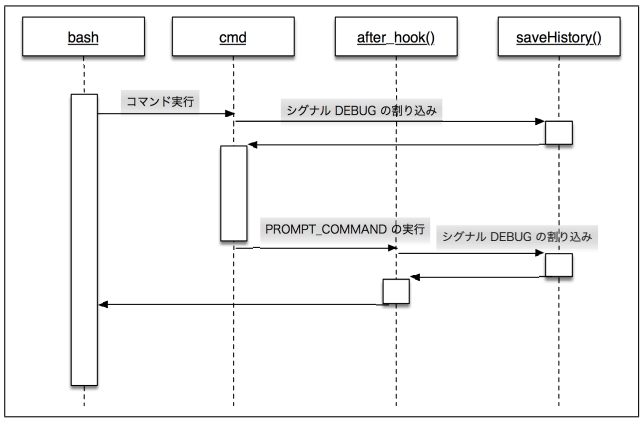


**図 9**　記録処理の呼び出しの流れ

ず，saveHistory ですべての記録を取っている理由は，PROMPT_COMMAND でのコマンドに対しても trap による割り込みがかかるからである．trap を用いて，シグナルに DEBUG に割り込み用のコマンドを指定すると，各コマンドの起動の直前にその割り込み用コマンドを起動する．一方，PROMPT_COMMAND は，プロンプトを表示する直前に指定されたコマンドを実行するので，実質的にその直前のコマンドの実行直後にコマンドを起動できることになる．ただし，PROMPT_COMMAND で実行するコマンドに対してもシグナル DEBUG による割り込みが有効に働く．よって，PROMPT_COMMAND で実行履歴を記録しようとすると，それ自体も記録され，不要な情報が残ることになる．そこで，PROMPT_COMMAND には何もしないシェル関数を設定し，図 9 に示すように，その呼び出しによって，saveHistory が呼び出されるようにする．シグナル DEBUG の処理中は，それ自体の割り込みは発生しないので，どのようなコマンドを実行しても実行履歴に影響を与えない．

ファイルの読み込みや書き込みは，ファイルの最終アクセス時刻と最終修正時刻に基づいて解析する．コマンド起動の直前に，その時点の時刻を記録するために一時ファイルを作成し，コマンド終了時に，その一時ファイルより最終アクセス時刻または最終修正時刻が新しいファイルを検索する．最終アクセス時刻が新しいファイルはコマンドが読み込んだファイルとして扱い，最終修正時刻が新しいファイルはコマンドが書き込んだファイルとして扱う．

### 4.3　実行履歴の解析

構造化した実行履歴を取得することで，コマンドの再実行の支援ができることを示すために，次の 3 つの解析を行うツールを実現した．

- 作業ごとに分割した実行履歴の取得
- ファイルを更新しないコマンドの除去
- 特定のファイルの生成に必要なコマンド列の取得

#### 4.3.1　作業ごとに分割した実行履歴の取得ツール

実行履歴を記録する時点で，各作業ごとにタスクに相当するログが構成されているので，タスクの通し番号順に各ログの内容を表示する．また，タスク番号を指定した場合には，その作業の内容を表示し，オプションにより再実行を指示できる．

すでに述べたように，必ずしも一定の時間の経過が作業の切れ目になるとは限らない．特に，作業内容を考えながらコマンドを入力しているときは，思案中に時間が経過し，作業者の意図と反して，作業が分割される．その場合，その分割を例外として分割をキャンセルする仕組みが必要であるが，それ自体を管理することは，システムを複雑にする．そこで，各ログは時間順に通し番号が割り振られたファイルとして保存し，それらのファイルを結合することで，履歴を結合する．

### 4.4　ファイルを更新しないコマンドの除去

man や ls など，作業内容を決めるために実行するコマンドを除去するために，タスクの表示の際に，ファイルを更新しないコマンドを除去するフィルタを構成した．例えば，図 4 では，コマンド 1，コマンド 4，コマンド 6 を除去する．

### 4.5　特定のファイルの生成に必要なコマンド列の取得

作成したいファイルを指定すると，それを生成するために必要なコマンド列を生成する機能を作成した．実行履歴の中には，試行錯誤の過程で同じファイルを別の方法で作成した記録が残っている可能性があるが，時間的に新しいものを有効となるように実装した．ただし，過去のタスク

での作業の方を優先したい場合があるので，タスクの番号を指定した場合には，そのタスクから実行履歴をたどるようにした．

直感的には，依存関係を辿るのみであるので，タスクは直接関係しないが，経験的に，2 つの方法を実装した．1 つは，タスク内だけで閉じて取得する方法で，もう 1 つは，タスクを無視して，すべての実行履歴から取得する方法である．このような 2 つの実装を行なった理由は，ファイルの依存関係が正しくない場合があるからである．典型的な例は次のものである．

- バックグラウンドで動作するエディタが更新したファイルをコマンドが更新したファイルとして扱っている
- LaTeX のように，実行時に生成したファイルを次の実行時に読み込む

後者は，厳密には，依存関係としては正しいが，タスクを無視すると，実行履歴中のすべてのコンパイルのコマンドが依存関係を持つことになり，作業者の意図と合わない．一般には，コンパイルは 2 回，あるいは 3 回連続して実行していると想定されるので，タスク内で閉じた解析の方が有効である．

### 4.6 評価

提案手法を評価するために，本論文の執筆や本システムの拡張などの作業で，実際に実行履歴を取得し，ある作業の再実行を行いたときに，意図した作業を取り出せるかどうかを検証した．その結果，エディタの取り扱いで問題が生じた．

```
vim rep.tex
platex rep.tex
dvipdfmx rep.dvi
open rep.pdf
vim fig.tex
platex fig.tex
man dvips
dvips -E fig.dvi -o fig.eps
vim rep.tex
platex rep.tex
platex rep.tex
dvipdfmx rep.dvi
vim fig.tex
platex fig.tex
dvips -E fig.dvi -o fig.eps
vim rep.tex
platex rep.tex
platex rep.tex
dvipdfmx rep.dvi
vim fig.tex
tasks
again -n 0 -o rep.pdf
```

**図 10** 実際の作業

```
[#0] --------------------
  vim rep.tex
  platex rep.tex
  dvipdfmx rep.dvi
  open rep.pdf
[#1] --------------------
  vim fig.tex
  platex fig.tex
  man dvips
  dvips -E fig.dvi -o fig.eps
  vim rep.tex
  platex rep.tex
  platex rep.tex
  dvipdfmx rep.dvi
[#2] --------------------
  vim fig.tex
  platex fig.tex
  dvips -E fig.dvi -o fig.eps
  vim rep.tex
  platex rep.tex
  platex rep.tex
  dvipdfmx rep.dvi
  vim fig.tex
  tasks
[#3]--------------------
  again -n 2 -o rep.pdf
```

**図 11** tasks の実行結果

図 10 は，LaTeX の文書を作成する作業の実行履歴である．履歴の最後の 2 行のうち，`tasks` は，タスクの一覧を表示するコマンドであり，図 11 のように表示される．`again` は，生成したいファイルを指定してコマンドを再実行させる指示であり，この場合は最後のタスクに対して PDF ファイルを指定している．

```
vim fig.tex
platex fig.tex
dvips -E fig.dvi -o fig.eps
vim rep.tex
platex rep.tex
platex rep.tex
dvipdfmx rep.dvi
```

**図 12** 実行されたコマンド列

実行されたコマンドを図 12 に示す．この例では，LaTeX のソースファイル `rep.tex` では，別の LaTeX のソースファイル `fig.tex` から生成した図を EPS に変換して取り込んでいる．コマンドライン上には現れないが，`rep.tex` をコンパイルする時点で EPS ファイル `fig.eps` が読み込まれるので，依存関係を辿って `fig.tex` のコンパイルが履歴に含まれている．さらに，各 LaTeX のソースファイルはエディタで編集をしているので，そのエディタの起動も含まれている．よって，依存関係は正しく解析できている．

依存関係の解析は正しいが，作業者の意図を考えると，

```
platex fig.tex
dvips -E fig.dvi -o fig.eps
platex rep.tex
platex rep.tex
dvipdfmx rep.dvi
```

**図 13** 期待するコマンド列

この結果は望ましくない．一般的に，複数のファイルで構成される文書やソースコードを対象とする場合，どれを修正するかは，そのときどきで変わるので，エディタによる編集とコンパイルなどの変換作業は分けて考えたい．つまり，ここで期待するコマンドは，図 13 に示すように，LaTeX のソースファイルから PDF ファイルを生成するまでの一連のものである．

この問題を解決する一つの方法は，エディタの編集を同定し，タスクを分割することである．エディタの編集を同定する方法の一つとして，ファイルへの読み書きの情報を参照する方法がある．すでに 1 度でもファイルが生成されたあとの編集であれば，同じファイルに対して読み込みをしているものは，エディタとみなせる．ただし，図 1 では sed を用いて句読点を変換しているが，このような処理との区別がつかない．また，最初にファイルを作成する時点では，読み込みが発生しないので，やはり区別できない．別の方法として，どれがエディタであるかは作業者自身はわかっているので，コマンド名を与えてタスクを分割できる仕組みを提供することである．

本研究では，バックグランドで動作するエディタがファイルを更新した場合のことは考慮していない．更新することで，その時点で実行中のコマンドに本来は存在しない依存関係が生じることになる．多くの場合，コマンドライン上には，関係するファイル名またはファイル名のうち拡張子を除いた部分が出現するので，対応するものがない場合には依存関係を辿らないような仕組みが有効と考えられる．ただし，C 言語のコンパイラが a.out を生成するように，必ずしも正確に取り扱えるわけではない．

本研究では，ファイルの属性の変更など，ディレクトリの情報が変わった場合に，そのファイルに関する依存関係を記録できない．また，ファイルの削除についても定義しておらず，読み書き以外のファイルへの操作に対する依存関係の取り扱いについては今後の課題である．

## 5. 関連研究

次に実行すべきコマンドを予測する研究は古くから行われている．Korvemaker らは，コマンドの実行履歴からユーザが次に実行するコマンドを確率的に予測する手法を提案している [1]．予測には，直前の 2 つのコマンドを基準とした確率を用いている．Yoshida らは，コマンドの実行履歴とファイルの入出力の情報を用いてコマンド間の関係を解決し，次に実行されるコマンドを予測している [2]．ファイルの入出力の情報の取得は，UNIX オペレーティングシステムを改造することで実現している．Ruvini らは，Smalltalk の開発環境の VisualWorks を対象に，ユーザの行動を機械学習を用いて学習し，次に実行する操作を予測する手法を提案している [3]．本研究は，予測は行わず，ユーザが指定するときの選択肢を提示する．また，既存の研究の予測は 1 つのコマンドであるが，本研究では複数のコマンドの列を選択肢にする点でも異なる．

## 6. おわりに

本研究では，ユーザが再実行したいコマンド列を実行履歴中から簡潔に選択できるためのコマンド候補列を生成する手法を提案し，その支援システムを実現した．また，実行履歴に対してコマンド列への分割やコマンド間の依存関係の解析などを行うことで実行履歴を構造化し，構造化した実行履歴から再実行の候補となるコマンド列の生成や絞り込みを行う方法を提案した．本研究で提案した手法を実現するためのシステムを実装し，実際の実行履歴に適用した結果から，コマンド候補列の生成と，生成されたコマンド列の選択ができることを確認した．今後の課題として，候補列の生成と絞り込みの方法を広げることで，生成するコマンド列候補の精度を向上させることが挙げられる．

### 参考文献

[1] B. Korvemaker, and R.Greiner, “Predicting UNIX Command Lines: Adjusting to User Patterns,” *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pp. 230–235, 2000.

[2] K. Yoshida, “User Command Prediction by Graph-Based Induction,” *Tools with Artificial Intelligence*, pp. 732–735, 1994.

[3] J.Ruvini, and C.Dony, “APE: Learning User’s Habits to Automate Repetitive Tasks,” *Proceedings of the 5th international conference on Intelligent user interfaces* , pp. 229–232, 2000

[4] 下村隆夫，“プログラムスライシング技術と応用，” 共立出版，1995

[5] Free Software Foundation, Inc., “*Bash Reference Manual,*” `https://www.gnu.org/software/bash/manual/bash.html#History-Interaction`

[6] “*History substitution,*” `http://www.tcsh.org/tcsh.html/History_substitution.html`

**謝辞** 本研究の一部は，JSPS 科研費 26350344, 2016 年度南山大学パッヘ奨励金 I-A-2 の助成を受けて実施した．