特別研究報告

題目

UNIX の 2038 年問題に対するプログラム修正ツールの作成

指導教員
井上 克郎 教授

報告者
水上 陽向

令和 3 年 2 月 9 日

大阪大学 基礎工学部 情報科学科

令和 2 年度 特別研究報告

UNIX の 2038 年問題に対するプログラム修正ツールの作成

水上 陽向

内容梗概

UNIX ベースのシステムでは，時刻情報として 1970 年 1 月 1 日を起点とした UNIX 時刻を用いている．また，この UNIX 時刻を扱うために，データサイズがシステム依存の整数型である，time_t 型が用いられている．UNIX ベースシステムにおける 2038 年問題とは，符号付 32bit の time_t 型が 2038 年 1 月 19 日にオーバーフローを起こすことに起因する問題である．これにより，32bit アーキテクチャを用いるシステムでは，異常な動作やエラーによる機能停止など，種々の不具合が生じることが予想されている．

この 2038 年問題に対し，システム内で扱う UNIX 時刻の起算点を変更してオーバーフローの発生を先送りすることにより，修正を行う箇所を最小限にとどめる対応方法が提案されている．また，その手法の実行に対し，プログラムスライシングを用いることで，プログラムから修正必要箇所の候補を効率的に特定する手法が提案されている．

本研究では，これらの先行研究の手法に対して，具体的な実装を行って 2038 年問題への対応作業を効率化・省力化するためのツール searcher2038 を作成した．本ツールでは，先行研究で提案された，時刻起算点の変更による 2038 年問題への対応方法に基づいて，プログラム内の修正必要箇所の特定・出力を行う．具体的にはまず，商用のソースコード解析ツール Understand を用いて，対象ソースコードの静的解析を行う．この解析情報をもとに，C 言語標準の時刻を扱う型を起点に依存情報を追うことで，時刻情報を扱う箇所の判定を行い，修正箇所の特定を行う．修正必要箇所となるのは，時刻情報を扱う外部関数呼び出し箇所と，起算点変更後の時刻情報の比較を行う箇所である．

作成ツールに対して，先行研究で用いられている，手動での修正必要箇所の特定結果と比較することによって，評価を行った．オープンソースの UNIX ベースシステムである，FreeBSD のコマンドのソースファイルを対象に作成ツールを実行し，結果の比較を行った結果，本ツールは修正が必要とされる箇所を漏れなく指摘できていることを確認した．

主な用語

2038 年問題
UNIX 時刻
データフロー解析

目 次

1　まえがき

現在広く用いられている UNIX ベースのシステムでは，時刻情報として 1970 年 1 月 1 日を起点とした UNIX 時刻を用いている [1]．また，この UNIX 時刻を扱うために，UNIX 標準では，time_t 型という変数型が用いられている．time_t 型は，データサイズがシステム依存の整数型であり，32bit アーキテクチャでは主に符号付 32bit 整数が用いられている．UNIX ベースのシステムにおける 2038 年問題は，この time_t 型が，2038 年 1 月 19 日に 32bit の符号つき整数で表せる範囲を超えてオーバーフローを起こすことに起因する問題である [2]．これにより，32bit アーキテクチャを用いるシステムでは，異常な動作やエラーによる機能停止など，種々の不具合が生じることが予想されている．時刻を表す値の桁あふれに起因する類似の問題には，2000 年問題 [3] や，GPS における週情報のオーバーフロー問題 [4] などさまざまな問題が存在し，実際に不具合が発生している [5, 6, 7]．

この 2038 年問題に対し，大江らの先行研究 [8] では，32bit アーキテクチャを採用した組込システムを対象として，64bit システムへの移行を行わず，修正を行う箇所を最小限にとどめた対応方法を提案している．この手法では，システム内で扱う UNIX 時刻の起算点を変更し，オーバーフローの発生を先送りすることで，2038 年問題への対応を行っている．また大江らは，その手法に対し，修正箇所の特定を効率化する研究も行っている [9]．この研究では，プログラムスライシングを用いることで，プログラムから修正必要箇所の候補を絞り込む手法が提案された．これにより，確認の必要な箇所を減らし，修正コストの低減が可能となった．

本研究では，これらの先行研究 [8, 9] の手法に対して，具体的な実装を行い，2038 年問題への対応のさらなる効率化・省力化を図るツール searcher2038 を作成した．searcher2038 では，[8] 及び [9] で提案された，時刻起算点の変更による 2038 年問題への対応について，プログラム内の修正必要箇所の特定・出力を行う．修正箇所の特定には，ソースコード解析ツールである Understand[10] を利用し，対象ソースコードの解析を行う．この解析情報をもとに，制御フローグラフとプログラム内のデータ依存情報を構築する．その後，C 言語標準の時刻を扱う型を起点に依存情報を追うことで，時刻情報を扱う箇所の判定を行い，修正箇所の特定を行う．修正必要箇所となるのは，時刻情報を扱う外部関数呼び出し箇所と，起算点変更後の時刻情報の比較を行う箇所である．

searcher2038 に対し，大江らによる先行研究 [9] で述べられている，手動での修正必要箇所特定結果と比較し，評価を行った．対象としたファイルは，オープンソースの UNIX ベースシステムである，FreeBSD[11] のコマンドのソースファイルである．date，stat，touch の 3 つのコマンドのソースファイルを対象に searcher2038 を実行し，結果の比較を行った．その結果，searcher2038 は手動での修正必要箇所特定結果に含まれている箇所をすべて発見で

きることを確認した．

以降 2 節では，研究の背景となる 2038 年問題について，また先行研究やその課題について論じる．3 節では，searcher2038 の作成において用いた手法を記す．4 節では，具体的な実装について記す．5 節では，searcher2038 の評価を行う．最後に，6 節で，まとめと今後の課題点を記している．

2 背景

2.1 2038 年問題

UNIX ベースシステムにおける 2038 年問題とは，UNIX システムにおいて時刻情報を扱う変数型である，time_t 型の桁あふれが発生することに起因する問題である [2]．本節では，この問題について述べる．

表 1 に，2038 年問題の概要を示している．UNIX ベースのシステムでは，時刻情報として，1970 年 1 月 1 日を起点とする UNIX 時刻が用いられている [1]．これを表すために，起算点からの秒数に相当する変数型である，time_t 型が存在する．time_t 型は整数型であるが，データサイズはシステムに依存となっている．このため，32bit アーキテクチャにおいて一般的な，符号付 32bit の time_t 型は，2038 年 1 月 19 日に桁あふれを起こし，負数となってしまう．この値を UNIX 時刻として解釈すると，誤った時刻として扱われてしまう．これにより，システムで以下のような誤動作が想定される．

- 正しい時刻情報との矛盾による通信エラー
- 負値となることによる時刻の大小比較の逆転
- 負値の時刻を想定したエラー処理の実行

現在では，システムの 64bit への移行が進んでいる [12]．これによって時刻情報が 64bit へと拡張されたシステムでは，time_t 型は符号付 64bit で定義されるのが普通であるため，2038 年に桁あふれが発生することはない．

しかし，こうした 64bit への拡張を行うには，ハードウェアの更新や，OS やアーキテクチャの変更を含む，大規模な対応が必要となる．このため，開発コストの制約や，開発期間の制約から，64bit への拡張による 2038 年問題への対応が困難な場合も少なくない．

表 1: 2038 年問題の概要

時刻 (UTC)	1970/1/1 00:00:00	... ...	2038/1/19 03:14:07	2038/1/19 03:14:08
符号付 32bit time_t 型値	0x0000 0000	...	0x7FFF FFFF	0x8000 0000
10 進表示	0	...	2147483647	**-2147483648**
時刻解釈	1970/1/1 00:00:00	... ...	2038/1/19 03:14:07	**1901/12/13 20:45:52**

2.2 時刻情報に関連する類似の問題

2038 年問題のように，時刻変数の桁あふれが原因となって生じる類似の問題は，多数存在する．また，桁あふれとは異なる原因で発生する，時刻情報に関する問題も多数存在する．本節では，これらの問題をいくつか紹介する．

2.2.1 2004 年 1 月 10 日の問題

2004 年 1 月 10 日は，UNIX 時刻の起算点である 1970 年 1 月 1 日から，符号付 32bit の time_t 型が桁あふれを起こす 2038 年 1 月 19 日の，中間となる日である．このため，time_t 型の時刻情報を 2 つ足し合わせると，符号付 32bit の範囲を超え，桁あふれが発生する．これにより，桁あふれを考慮していないシステムで，誤動作が生じることがあった [2].

国内では，曜日が間違って判定されたことにより携帯電話の料金請求が誤った値となった事例がある．また，通信システムが正確な日時情報を取得できなくなったことで，銀行 ATM での取引が行えなくなる事例も発生した．

2.2.2 GPS の桁あふれ問題

GPS では，週の情報を 10bit の 2 進数で処理している．このため，1024 週が経過すると，オーバーフローを起こして 0 週に戻ってしまう．1024 週でカウンタがリセットされてしまうことは仕様に含まれているが [13]，問題を考慮せずに製造されたシステムで，誤動作が発生しうる [4]．この問題は，1024 週ごとに生じるため，これまでに 1999 年と 2019 年の 2 回，発生している．

1999 年には，カーナビの動作不良や，位置情報ずれなどの問題が発生した [6]．2019 年にも，既に一度経験した問題にも関わらず，バイクのナビシステムで，時刻が正常に表示されなくなるなどの事例が発生している [7].

2.2.3 2000 年問題

2000 年問題は，Y2K 問題としても知られる問題である．2000 年は，コンピュータが実用化されて以来，初めて到来する下二桁が “00” の年であり，初めて上二桁が変わった年であった．このため，1900 年代であることを前提とし，年情報の上二桁を省略して扱っていたシステムでは，2000 年を 1900 年と解釈してしまい，誤作動やエラーの発生が起こり得た [3]. 国内では，気象台の地震観測システムのデータ送受信に不具合が生じたり，FAX 情報サービスに不具合が生じるなどの影響があった [14].

この問題に関しては，比較的早くから危険性が指摘され，予め対策が進められていたため，結果として想定されていたような大きな混乱が生じることはなかった [5].

2.3 先行研究と課題

2.3.1 先行研究

2.1 節で述べたように，2038 年問題に対する 64bit への拡張による対応は，全てのシステムで行えるわけではない．

2038 問題に対し，大江らによる先行研究 [8] では，コストの制約の大きい組み込み機器を対象として，UNIX 時刻の起算点を変更することによる対応を行った．大江らの手法では，本来 1970 年である UNIX 時刻の起算点を後にずらすことで，2038 年に発生する桁あふれを先送りしている．起算点を 28 年後にずらして 1998 年 1 月 1 日とすることで，桁あふれの発生も 28 年先送りされ，2066 年になる．28 年の起算点変更を行った際の time_t 型値の遷移を，表 2 に示している．この手法による修正は，標準ライブラリや OS 部の変更を行うことなく完了することが可能であり，2038 年問題への対応にかかるコストを抑えることに成功している．

これに加え，大江らは [9] において，上記の時刻起算点変更による対応のために，修正箇所の特定を効率化する手法を提案している．この研究では，プログラムスライシング [15] の手法を用いることにより，時刻情報を扱う箇所の特定を容易にしている．[9] では，以下のような手順で修正箇所の特定を行っている．

1. time_t 型変数の利用箇所を基準とした，静的スライスを作成する
2. 以下に該当する，2038 年問題に関わらないスライス文を除外する
    - 時刻情報を扱わない関数の文
    - 時刻情報を変更しない機能に関する文
    - 時刻情報のオーバーフローを発生させない文
3. 以下に該当する，修正必要箇所を特定する

表 2: 起算点変更を行った場合の time_t 値の推移

時刻 (UTC)	1970/1/1 00:00:00	1998/1/1 00:00:00	... ...	2038/1/19 03:14:07	2038/1/19 03:14:08
符号付 32bit time_t 型値	—	0x0000 0000	...	0x4B55 237F	0x4B55 2380
10 進表示	—	0	...	1263870847	1263870848
時刻解釈	—	1998/1/1 00:00:00	... ...	2038/1/19 03:14:07	**2038/1/19 03:14:08**

- 時刻情報を扱う標準ライブラリ関数の呼び出し
- 起算点変更後の時刻情報を評価する箇所

[9] では，この手法の評価のために，オープンソースの UNIX ベースシステムである FreeBSD[11] の，date，stat，touch の 3 つのコマンドのソースファイルを対象として，修正の実施を行っている．これにより，修正必要箇所を特定するために，確認が必要なプログラムの量を減少させる効果が確認された．

2.3.2 課題と本研究の目的

時刻起算点の変更による 2038 年問題への対応を一般化した大江らの研究 [9] は，手法の提案のみにとどまっている．この対応手法においては，実際の修正箇所の特定から修正の実行までを手動で行う必要があり，修正のためのコストは小さいとは言えない．そこで，本研究では，大江らによる手法の実装を行い，修正箇所の特定をより効率化するためのツールの作成を目指す．

3　プログラム修正手法

2.3 節で論じた通り，大江らの先行研究 [8, 9] では，システム内の UNIX 時刻の起点を後にずらして扱うことで，time_t 型の変数がオーバーフローする時刻を遅らせる手法をとっている．本節では，3.1 節にて，手法の前提とした既存ツールについて述べる．その後，3.2 節で，時刻起算点の変更による対応手法について述べ，3.3 節以降で，[8, 9] の手法をもとにした，プログラム修正箇所の特定手法を述べる．

3.1　ソースコード解析ツール Understand

本研究では，ソースコードの解析を行うために，ソースコード解析ツール Understand[10] を用いている．Understand は，ソースファイルを静的に解析するツールであり，プログラムの制御フローや構造，クラス継承，関数や変数の関係，構文解析情報など，様々な情報を取り出すことができる．また，解析情報の視覚化，ファイルへの書き出しに加え，C，Python，Java，Perl の 4 つの言語の API が存在し，これを通して，プログラム内で解析情報の取り出しを行うことができる．図 1 は，テキストファイルに出力した，Understand の分析結果の例の一部である．

Understand のような，ソースコードの解析を行うツールには様々なものが存在する．今回は，API やファイル出力を通して，必要な情報をプログラムから容易に取り出せる点や，コマンドラインから実行可能で，プログラムから容易に解析を実行できる点を評価し，Understand を選択した．

```
buf   (Local Object)
  Declared as: char [32]
    Define [date.c, 231]      printisodate
    Use [date.c, 237]         printisodate
    Use [date.c, 237]         printisodate
    Use [date.c, 243]         printisodate
    Use [date.c, 243]         printisodate
    Use [date.c, 246]         printisodate

century   (Local Object)
  Declared as: int
    Define [date.c, 258]      setthetime
    Set [date.c, 296]         setthetime
    Set [date.c, 301]         setthetime
    Use [date.c, 304]         setthetime

ch   (Local Object)
  Declared as: int
    Define [date.c, 94]       main
    Set [date.c, 111]         main
    Use [date.c, 112]         main
```

図 1: Understand の分析結果例

3.2　時刻起算点の変更

2.3 節で述べたように，大江ら [8, 9] の手法では，本来 1970 年である UNIX 時刻の起算点を後にずらすことで，2038 年に発生する桁あふれを先送りしている．起算点を 28 年後にずらして，1998 年 1 月 1 日とすることで，桁あふれの発生も 28 年先送りされ，2066 年になる．この修正手法は，システム内部で扱う時刻情報のみを変更することによるものである．このため，標準ライブラリなどの修正を必要とせず，対応コストを抑えることができる．

3.3　時刻情報の判定と追跡

システム内の時刻情報の起点を変更するために，ソースコード内で時刻情報を扱う変数を特定し，その変更や参照を追跡する必要がある．この追跡を行うために，制御フローグラフとデータ依存の情報を構築し，利用する．

3.3.1　制御フロー

制御フローは，プログラム中のソースコードが，どのような順序で実行されうるかを表したものである．

通常，ソースコードは，先頭行から最終行へと順に実行される．しかし，if 文などの分岐，for 文などのループ，goto 文などのジャンプが存在する場合，その限りではない．これらの制御述語による実行順の変更を含む，プログラムの実行の順序を表したものが，制御フローである．

制御フローは，一般には 1 つの関数やメソッドといった単位で構成される．制御フローを表したグラフは，Control-Flow Graph（CFG）と呼ばれる．

3.3.2　データ依存

データ依存は，変数の参照によって生じる，プログラムの文や述語の間での依存関係である．プログラム中のある点 A で定義・編集された変数の値が，他の定義や編集を挟まず点 B で参照されうる場合，B は A にデータ依存するという．

図 2 は，データ依存が存在するコード片の例である．この例において，プログラムは 1 行目から 5 行目まで，順に実行されていく．このとき，2 行目で変更された a の値が，3 行目で参照されている．また，4 行目で変更された a の値が，5 行目で参照されている．これらから，3 行目は 2 行目にデータ依存し，5 行目は 4 行目にデータ依存する．一方，2 行目で変更された a の値は，5 行目に到達する前に，4 行目で上書きされている．このため，5 行目は 2 行目にはデータ依存しない．

```
1.  int a, b, c;
2.  a = 20;
3.  b = a + 10;
4.  a = 50;
5.  c = a - 30;
```

図 2: データ依存の例

3.4 修正箇所の特定

本研究では，大江らの先行研究 [9] に従い，以下の 2 つを，2038 年問題への対応のための修正必要箇所と定める．

- 外部関数の呼び出し箇所
- 時刻変数の直接評価箇所

これらの定義については，続く 3.4.1，3.4.2 項で述べる．

3.4.1 外部関数の呼び出し箇所

修正が必要な箇所の 1 つは，時刻情報を扱う変数に関連する外部関数の呼び出し箇所である．

大江らの先行研究 [8, 9] の手法における 2038 年問題への対応は，3.2 節で述べた通り，システム内部で扱う時刻情報の起算点を変更することにより行う．そのため，起算点変更を前提としないシステム外部の時刻情報と，起算点変更を行ったシステム内部の時刻情報をやり取りする，外部関数の呼び出し箇所は修正が必要となる．

図 3 の左側は，時刻情報を扱う外部関数呼び出しを含む，コード片の例である．ここでは，C 言語の標準ライブラリ関数である，mktime 関数が使用されている．mktime 関数は，tm 構造体を引数に取り，その値を time_t 型に変換して返す関数である [16]．大江らの先行研究 [9] では，mktime 関数のラッパー関数を作成し，その内部で引数の年情報の減算を行ってから，mktime 関数を呼び出すようにする修正を行っている．図 3 の右側は，同図の左側のコード片に対し，修正を行った例である．なお，*defyear* は，時刻起算点変更分の年数，すなわち 28 を表す定数である．

3.4.2 時刻変数の直接評価箇所

修正が必要なもう 1 つの箇所は，時刻起算点変更後の時刻情報を，起算点変更を行っていない値と比較する箇所である．時刻起算点が変更された値と，起算点変更を行っていない値

```
time_t tv;                          time_t tv;
struct tm ts;                       struct tm ts;
...                          ⟶      ...
tv = mktime(&ts);                   tv = wrapper_mktime(&ts);
                                      ...
                                    static time_t
                                    wrapper_mktime(struct tm *ptm){
                                       time_t ret;
                                       ptm->tm_year -= defyear;
                                       ret = mktime(ptm);
                                       return ret;
                                    }
```

図 3: 時刻情報を扱う外部関数呼び出しの例

```
time_t tv;                                time_t tv;
long num;                                 long num;
tv = time(NULL) − defsec                  tv = time(NULL) − defsec
...                                       ...
if(tv > num){                      ⟶     if(tv > num − defsec ){
...                                       ...
}                                         }
```

図 4: 時刻情報の直接比較の例

を比較すると，想定と異なる結果となってしまう．このため，修正が必要となる．

図 4 の左側は，起算点変更を行った時刻情報を，long 型変数と比較しているコード片の例である．また，図 4 の右側は，これに対して修正を行った例である．ただし，*defsec* は，起算点変更分の秒数を表す定数である．比較対象の値に，起算点変更分に相当する減算を行うことで，正しい比較が行えるようになる．

4　実装

4.1　開発言語と想定環境

searcher2038 の作成は，Python[17] を用いて行った．使用したバージョンは，Python3.9 である．Python を利用したのは，Understand API のうち，Python API が，今回必要な機能を最も容易かつ十全に扱うことができたためである．searcher2038 の実行には，python3.9 が実行できる環境が必要となる．また，3.1 節に記したように，既存ツール Understand[10] を searcher2038 内で使用するため，これが利用できる環境が必要となる．

searcher2038 の動作は，Windows10 のコマンドプロンプトから Python を実行して確認している．

4.2　処理の流れ

本節では，searcher2038 での処理の流れについて概説する．図 5 に，処理の流れの概図を示している．searcher2038 は，ソースコードの解析，データ依存情報の構築，修正箇所の特定，結果の出力の，4 つのステップで処理を行う．このうち，ステップ 2 にあたる，データ依存情報の構築は，制御フローグラフの構築を行う段階と，それを用いてデータ依存情報の構築を行う段階に分けられる．また，ステップ 3 にあたる修正箇所の特定は，時刻情報の判定と追跡を行う段階と，それを用いて修正箇所の特定を行う段階に分けられる．

ステップ 1 では，ソースコード解析ツール Understand[10] を用いて，対象ソースファイルの解析を行う．ステップ 2 では，解析情報をもとに，プログラムの各文の依存情報を構築する．ステップ 3 では，構築した依存情報と，解析情報をもとに，修正が必要な箇所の特定を行う．最後に，ステップ 4 で特定した修正箇所の情報を出力する．

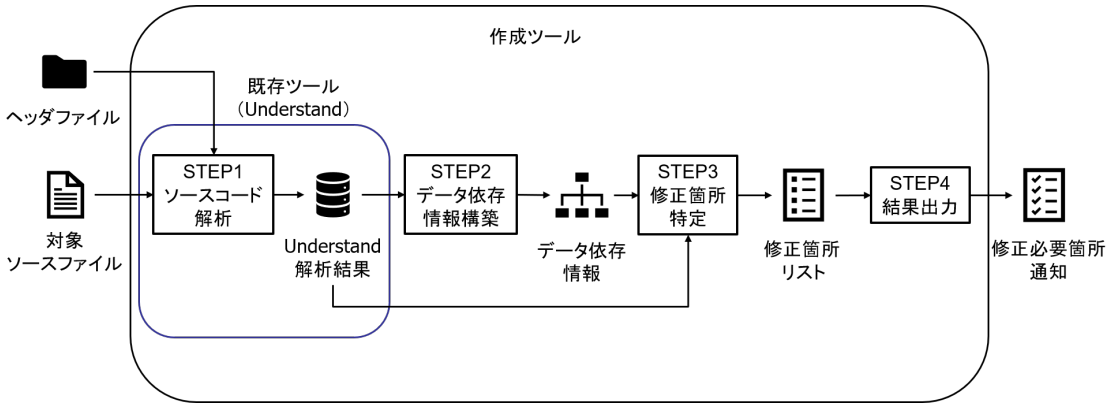

図 5: 処理の流れ

4.3 searcher2038 への入力

searcher2038 は，修正対象である C 言語のソースファイルと，そのコンパイルに用いるヘッダファイルを入力とする．ソースファイルはファイルのパスを，ヘッダファイルは存在するフォルダのパスを，それぞれ searcher2038 の実行時に標準入力から受け付ける．

4.3.1 ソースファイルの入力

ソースファイルは，1 個以上の入力を受け付ける．“None” 以外の文字列が入力されるたびに，それをソースファイルのパスとして受けつけ，リストに格納する．“None” の文字列が入力された場合，そこでソースファイルの受け付けを終了する．入力されたソースファイルのパスは，受け付け終了後にファイルへの書き込みを行っておく．また，パスを “/” または “\” を区切りとして分割した末尾を取り出すことで，ファイル名部分を抜き出し，これらを格納したファイル名リストを作成する．

4.3.2 インクルードパスの入力

インクルードパスは，0 個または 1 個の入力を受け付ける．“None” の文字列が入力された場合，インクルードパスの入力がないものとみなす．それ以外の入力があった場合は，それをインクルードパスの入力とみなす．

4.4 STEP1: ソースコード解析

本節では，対象ソースコードの解析について記す．対象ソースファイルの解析は，Understand [10] を利用して行う．

Understand では，解析対象としたソースファイル内の，全ての関数や変数，およびファイル自身が，“エンティティー” として扱われる．エンティティーには，名前と型の情報や利用箇所の情報，関数とパラメータの間などの親子関係など，解析で得られる情報が紐付けられている．今回用いた Python API では，これらエンティティーや，それらの利用箇所 1 つ 1 つに対して，対応するインスタンスが作成され，メンバメソッドを通して情報の参照が行えるようになっている．

4.4.1 前処理

Understand での解析を行う前に，対象ソースファイルに簡単な前処理を行う．ソースファイル中にタブ文字が存在すると，Understand で解析を行った際に，単一の特殊な空白文字として認識される場合と，複数の空白文字の連続として認識される場合がある．このため，

解析を行う前に，入力されたソースファイル全てに対し，タブ文字を空白文字に置換する処理を行っておく．

4.4.2 Understand での解析

Python の標準モジュールである，subprocess モジュール [18] の run コマンドを用いて，Understand をコマンドラインから実行することで，ソースコードの解析を行う．searcher2038 は，データベースの作成，ソースファイルの追加，インクルードパスの設定，解析の実行の順でコマンドを実行する．Understand への対象ソースファイルの入力は，4.3.1 項で作成した，パスを記したファイルを与えることで行う．また，インクルードパスの入力は，入力されたパスをそのまま与えることで行う．インクルードパスの入力がなかった場合（“None” が入力された場合）は，インクルードパスの設定は行わない．

解析を実行することで，Understand は解析結果のデータベースファイルを作成する．Understand では，このデータベースファイルを指定して，結果をテキストや html などのファイル形式で書き出すことや，API を通して利用することができる．

4.4.3 解析情報の取り出し

Understand による解析の終了後，Understand の Python API を通して，データベースファイルから基礎的な解析情報の取り出しを行う．ここでは，取り出した情報をもとに，全てのエンティティーを格納したリストの作成，ファイルエンティティーのリストの作成，各ファイルの構文解析情報のリストの作成を行う．

4.5 STEP2-1: 制御フローグラフの構築

解析対象のソースファイルに含まれる全ての関数に対し，制御フローグラフの構築を行う．制御フローグラフの情報は，Understand から DOT 形式 [19] として取り出すことができる．ここでは，この DOT 形式のファイルを読み出し，プログラム内で扱える連結リストとして再構築する．

4.5.1 関数インスタンスの作成

制御フローグラフの作成は，対象ソースファイル内の関数それぞれに対して行う．作成した制御フローグラフの情報を保持させるため，クラス TheFunction を設け，対象の関数ごとにインスタンスを作成する．TheFunction インスタンスは，以下のメンバを保持するインスタンスとして作成される．

- Understand API における，当該関数のエンティティー

- 関数名
- パラメータ型のリスト
- 関数自身の型
- 関数の制御フローグラフの先頭と末尾のノード
- 関数が外部関数であるか

TheFunction インスタンスの保持する情報のうち，関数名，パラメータ型のリスト，関数自身の型，外部関数であるかの情報は，インスタンス作成時に設定する．外部関数か否かの判定は，対象ソースファイル内で定義が行われているかによって行う．Understand API から，当該関数の出現箇所の情報を取り出し,「定義（define）」の出現が存在すれば，そのファイルのエンティティーを，インスタンスに保持させる．存在しなければ，外部関数とみなし，None を保持させる．残る制御フローグラフの情報は，次項の手順において付加する．

4.5.2 制御フローグラフの構築

DOT 形式ファイルには，グラフ全体の形式，ノードのラベルや形状，辺の位置とラベルといった情報が，テキストによって記されている．Understand によって書き出された制御フローグラフでは，各ノードがソースファイルの 1 文に対応する．各ノードのラベル内には，以下の情報が含まれている．

- 文の種別（分岐文，ループ文など）
- 対応するソースコード片
- 当該文の開始位置の行及び列，終了位置の行及び列
- 分岐文またはループ文であれば，その終了ノード
- ノードから出る辺の行先（複数あれば全て）

これらの情報を読み取り，Node インスタンスを作成する．作成時，インスタンスには，以下の情報を保持させる．

- ノード番号
- 文種別
- 開始・終了位置の行および列番号

- 属する関数の TheFunction インスタンス

ノード情報の読み取りが完了した後は，辺情報を読み取り，ノードの接続を行う．各ノードのインスタンスは，直前のノードを格納するリスト prevnode と，直後のノードを格納するリスト nextnode を持ち，これを以下の手順で埋めていく．

1. 辺の根本のノード番号と，行先のノード番号を確認し，その番号を持つノードインスタンスを探索する．
2. 根本のノードインスタンスの nextnode リストに，行先ノードのインスタンスを加える．
3. 行先ノードインスタンスの prevnode リストに，根本のノードのインスタンスを加える．

最後に，関数の制御フローグラフの先頭と末尾のノードを，当該関数の TheFunction インスタンスに保持させる．先頭と末尾のノードのラベルには，それぞれ “start” と “end” の文字列が特定の位置に存在するため，これを読み取ることでそれらのノードを特定する．

以上で，制御フローグラフが連結リストとして構築され，その先頭と末尾のノードが，TheFunction インスタンスに保持された状態となる．

4.5.3 オブジェクトの出現情報の付加

データ依存情報の構築を行うために，制御フローグラフの各ノードに，ソースコードの当該箇所での，変数の出現情報のリストを保持させる．Understand の解析結果に含まれる全てのエンティティーの，全ての出現情報について，存在するファイルと，行及び列の情報を確認する．これらの情報が以下の条件を満たしていれば，その出現が，当該ノードに対応するソースコードで発生しているとみなして，リストに追加する．

- 出現箇所のファイルが，当該ノードが含まれる関数の定義されたファイルと一致する
- 出現箇所の行が，当該ノードの保持している開始行から終了行の範囲に含まれる
- 出現箇所の行が，当該ノードの保持している開始行と一致していれば，出現箇所の列が，ノードの開始列以降に存在する
- 出現箇所の行が，当該ノードの保持している終了行と一致していれば，出現箇所の列が，ノードの終了列以降に存在する

ノードインスタンスには，それぞれの出現につき，Understand API のリファレンスインスタンスと，出現しているオブジェクト（変数，関数など）の，Understand API のインスタンスを組として，出現情報を保持させる．

4.6 STEP2-2: データ依存情報構築

本節では，3.3.2 項で述べた，データ依存情報の構築について記す．データ依存情報は，4.5.3 項でノードインスタンスに付加した，オブジェクトの出現情報を用いて構築する．

依存情報は，それぞれのノードインスタンスに，依存先ノードのリストと，依存元ノードのリストとして保持させる．それぞれのノードインスタンスに対し，以下の手順を実行することで，依存情報の構築を行う．

1. ノードインスタンス n が持つ出現情報のリストから，変数の参照にあたるものを探索する．
2. 1 で発見した参照に対し，同じ変数の参照以外の出現を含むノードインスタンスが見つかるまで，制御フローグラフを後ろ向きに辿る．探索は深さ優先探索で行う．変更や定義といった出現が該当する．
3. 2 で該当する出現が見つかった場合，そのノードの依存元ノードに，ノード n を追加する．また，ノード n の依存先ノードに，見つかったノードを追加する．

4.7 STEP3-1: 時刻情報の判定と追跡

本節では，時刻情報を扱う変数の特定と，その追跡について記す．はじめに，searcher2038 において時刻情報を扱っているとみなす変数の定義を述べる．その後，実際の特定と追跡の手順を述べる．この手順では，どの変数が時刻情報を扱っているかに加え，それらの変数が，対象ソースコード内のどの位置において，時刻情報を保持しているかも調べる．このために，TimeVal クラスを定義し，それぞれの時刻情報を扱う変数ごとにインスタンスを作成して，以下の情報を保持させる．

- Understand API における，当該変数のエンティティー
- 変数の種別（ローカル変数，グローバル変数，パラメータなど）
- 当該変数が時刻情報を保持している状態で出現する箇所に対応する，(4.5 節で作成した)Node インスタンスのリスト

この Node インスタンスのリストは，以下では timepoint リストと呼称する．

4.7.1 時刻情報を扱う変数の定義

searcher2038 において，時刻情報を扱っているとみなす変数には，以下の 2 種類が存在する．

- C 言語の標準ライブラリで定義された時刻用の型である，time_t 型または struct tm 型を持つ変数
- 他の時刻情報を扱う変数を参照して，値が変更された変数

後者には，例えば time_t 型の値を代入して初期化された整数型の変数が該当する．前者に分類される変数を直接参照せずとも，後者に該当する他の変数を参照していれば，それは時刻情報を扱う変数とみなされる．

次項以降で述べる，時刻変数の特定は，前者に該当する変数を特定・追跡と，後者に該当する変数の特定・追跡の 2 ステップに分けて行う．

4.7.2 C 言語標準の型を持つ時刻変数の特定

はじめに，C 言語標準の時刻型を持つ変数の特定を行う．C 言語標準の型を持つ変数は，Understand API から，変数の型情報を取り出すことで特定できる．

これらの変数の特定と追跡は，以下のように行う．

1. 発見された時刻変数の TimeVal インスタンスを格納するリストと，追跡対象の変数を格納するスタックを作成する．
2. 4.4.3 節で作成した，全てのエンティティーを格納したリストから，time_t 型か struct tm 型を持ち，関数ではないものを取り出す．
3. 2 で取り出したエンティティーそれぞれについて，TimeVal インスタンスを作成する．また，これを 1 で作成したリストとスタックに加える．
4. 1 で作成したスタックから要素を 1 つ取り出す．
5. 4 で取り出した要素が，グローバル変数の TimeVal インスタンスであれば，以下を行う．
   (a) 対象ソースファイル内から，当該変数の出現箇所を探す．
   (b) 出現箇所に対応する Node インスタンス全てを，当該 TimeVal インスタンスの，「時刻情報を保持している状態で出現する箇所」リストに加える．
   (c) (b) の Node インスタンスにデータ依存する Node インスタンスのうち，当該変数の参照にあたる出現があるものを，当該 TimeVal インスタンスの，timepoint リストに加える．
6. 4 で取り出した要素が，グローバル変数以外の TimeVal インスタンスであれば，以下を行う．

(a) 対象ソースファイル内から，当該変数の，参照以外（定義，変更などが該当）の出現箇所を探す．

(b) (a) の出現箇所に対応する Node インスタンスを，当該 TimeVal インスタンスの，timepoint リストに加える．

(c) (b) の Node インスタンスにデータ依存する Node インスタンスのうち，当該変数の参照にあたる出現があるものを，当該 TimeVal インスタンスの，timepoint リストに加える．

7. 4 から 6 を，スタックが空になるまで繰り返す．

4.7.3 依存関係によって生じる時刻変数の特定

次に，既に発見された時刻変数を参照する変数の特定を行う．この処理では，新たな時刻情報を扱う変数を発見するだけではなく，既に発見されている時刻変数に対し，それが時刻情報を保持している箇所の情報を更新することも行う．この対象には，前項で特定された時刻変数も含まれる．

1. 発見された時刻変数の TimeVal インスタンスを格納するリストと，追跡対象の変数を格納するスタックを作成する．それぞれ，4.7.2 節で発見された時刻変数の TimeVal インスタンスのリストで初期化を行う．

2. 1 で作成したスタックから要素を 1 つ取り出す．

3. 2 で取り出した TimeVal インスタンスの，timepoint リストに含まれる Node インスタンスを 1 つ取り出す．

4. 3 で取り出した Node インスタンスの保持する，オブジェクト出現情報から 1 つを取り出す．

5. 4 で取り出した出現情報が，2 で取り出した TimeVal インスタンスの時刻変数以外かつ関数でないオブジェクトのもので，参照以外の出現（定義，変更などが該当）である場合，以下を行う．そうでなければ，7 へ進む．

   (a) 4 で取り出した出現情報のオブジェクトに対応する TimeVal インスタンスが既に存在すれば，それを以降の手順における newtv として扱う．

   (b) 4 で取り出した出現情報のオブジェクトに対応する TimeVal インスタンスが存在しなければ，新たに生成し，1 で作成した時刻変数の TimeVal インスタンスのリストに追加する．また，これを以降の手順における newtv として扱う．

6. 3 で取り出した Node インスタンスが，newtv の timepoint リストに含まれていなければ，新たに追加し，以下を実行する．

   (a) newtv が 1 で作成したスタックに含まれていなければ，追加する．

   (b) 3 で取り出した Node インスタンスの全ての依存元ノードの，全てのオブジェクト出現情報に対し，それが newtv に対応する変数オブジェクトのもので，参照にあたる出現であれば，当該依存元ノードを newtv の timepoint リストに加える．

7. 3 で取り出した Node インスタンスの保持する，オブジェクト出現情報全てについて，4-6 を行う．

8. 2 で取り出した TimeVal インスタンスの，timepoint リストに含まれる Node インスタンス全てについて，3-7 を行う．

9. 1 で作成したスタックが空でなければ，1 へ戻る．

本項に記した処理が終了した時点で，時刻変数のリスト（上記 1 で作成したもの）が，特定すべき全ての時刻変数を格納した状態となる．

4.8 STEP3-2: 修正箇所の特定

本節では，修正が必要な箇所の特定について記す．修正が必要な箇所には，3.4 節で論じたように，外部関数の呼び出し箇所と，時刻変数の直接評価箇所の 2 種類がある．ここまでの処理で集めた情報を用いて，それぞれ別個に特定を行う．

4.8.1 外部関数呼び出し箇所の特定

修正が必要な外部関数呼び出し箇所の情報を記憶するために，CallToFix クラスを作成する．それぞれの呼び出し箇所ごとにインスタンスを作成し，以下の情報を保持させる．

- Understand API における，当該箇所での関数の出現のインスタンス
- 当該箇所の行番号
- 当該箇所に対応する，Node インスタンス
- 当該箇所で出現する時刻変数のリスト

また，生成された CallToFix インスタンスを全て格納するためのリストを，はじめに作成する．

4.7 節で特定された全ての時刻変数の TimeVal インスタンスにおける，timepoint リスト内の全ての Node インスタンスについて，保持しているオブジェクト出現情報を調べる．関数の出現で，かつその関数が対象ソースファイル内で定義されていなければ，それを外部関数の呼び出しとみなし，以下を行う．

- その呼び出しについての CallToFix インスタンスが存在しなければ，作成し，リストに加える．
- その呼び出しについての CallToFix インスタンスが存在すれば，その時刻変数リストに，現在注目している時刻変数を加える．

全ての時刻変数に対して以上の処理を行うことで，修正が必要な外部関数呼び出しの特定が完了する．

4.8.2 直接評価箇所の特定

修正が必要な直接評価箇所の情報を記憶するために，CompareToFix クラスを作成する．それぞれの評価箇所ごとにインスタンスを作成し，以下の情報を保持させる．

- Understand API における，当該箇所での時刻変数の参照のインスタンス
- 当該箇所の行番号
- 当該箇所に対応する，Node インスタンス

また，生成された ComparaToFix インスタンスを全て格納するためのリストを，はじめに作成する．

4.7 節で特定された全ての時刻変数の TimeVal インスタンスにおける，timepoint リスト内の全ての Node インスタンスについて，その時刻変数の出現情報を調べる．それが参照にあたる出現であれば，Understand API の構文解析機能を用いて，比較を行う箇所かを調べる．まず，その時刻変数の出現箇所の前後のトークンを，比較演算子が見つかるか，改行または予約語に行き着くまで順に辿っていく．比較演算子が見つかった場合，時刻変数の比較が行われている箇所とみなし，比較対象を調べる．修正が必要となるのは，修正後の時刻変数を保持していない値との比較である．注目しているノードが，現在調べている以外の TimeVal インスタンスの timepoint リストに含まれていない場合，比較対象が時刻変数ではないとみなし，CompareToFix インスタンスを作成して修正対象リストに加える．そうでない場合，以下の手順で，比較対象を確認する．

- 比較演算子が時刻変数の前方にあった場合，以下の手順では，前方のトークンを調べる．後方にあった場合，後方のトークンを調べる．

- 識別子，リテラルが見つかるか，改行または予約語に行き着くまで，前方または後方のトークンを辿る．
  - 識別子が見つかった場合，それが TimeVal インスタンスの timepoint リストに現在のノードを含んでいる時刻変数でなければ，CompareToFix インスタンスを作成して修正対象リストに加える．
  - リテラルが見つかった場合，CompareToFix インスタンスを作成して修正対象リストに加える．
  - 改行，予約語，また時刻変数の識別子が見つかった場合，追加は行わない．

全ての時刻変数の参照に対し，以上の処理を行うことで，修正が必要な時刻変数の評価箇所の特定が完了する．

4.9 STEP4: 結果出力

本節では，結果の出力について記す．4.8 節で特定した，修正が必要な箇所を，順に出力する．

修正必要箇所は，4.8.1 項で作成した外部関数呼び出しのリストと，4.8.2 項で作成した直接評価箇所のリストに格納されている．出力は，まず外部関数呼び出しの一覧，次に直接評価箇所の一覧の順で行う．リスト内のそれぞれに対し，存在するファイル，出現行に加え，外部呼出し箇所については，その関数と，当該箇所に出現する時刻変数のリストを，直接評価箇所については評価されている時刻変数を，出力する．ユーザーは，この出力で示されたソースコード行に対し，3.4.1 ，3.4.2 項で例示したような修正を加えることで，2038 年問題への対応を行うことができる．

searcher2038 の出力例として，図 6 に記した簡単なプログラムに対する出力を，図 7 に示している．なお，紙面の都合から，図 6 では，プログラム内の空行は省いている．

```
#include¡stdio.h¿
#include¡time.h¿
void haribote(struct tm);
void main(){
    time_t timer;
    long a, b, c, d;
    struct tm *timestruct;
    time(&timer);
    printf("time -> %ld\n", timer);
    a = timer;
    b = a + 100;
    c = a + b;
    printf("a -> %ld\n", a);
    printf("b -> %ld\n", b);
    printf("c -> %ld\n", c);
    *timestruct = localtime(&timer);
    haribote(*timestruct);
    d = (long)*timestruct->tm_year;
    do {
      b–;
      if(a < 0){
        return;
      }
    } while (a < b);
    if(a == b){
      printf("%d", a);
      printf("OK.");
    }
}
void haribote(struct tm ts){
  return;
}
```

図 6: サンプルプログラム

```
Result : The following are points may be fixed

=== Callings those may be fixed =================

testprogram.c line11 <time>
    < timer time >

testprogram.c line13 <printf>
    < timer printf >

testprogram.c line23 <localtime>
    < timer timestruct localtime >

testprogram.c line19 <printf>
    < printf a >

testprogram.c line36 <printf>
    < printf a >

testprogram.c line20 <printf>
    < printf b >

testprogram.c line21 <printf>
    < printf c >

=== Comparings those may be fixed ===============

testprogram.c line31 <a>

=== End ========================================
```

図 7: 出力例

5　評価

5.1　先行研究との比較

searcher2038 の評価として，先行研究 [9] による修正結果との比較を行った．

5.1.1　対象と手法

大江らの先行研究 [9] では，提案手法の評価として，オープンソースの UNIX ベースシステムである FreeBSD[11] のソースファイルを対象に，2038 年問題に対応する修正を行っている．本研究では，先行研究 [9] と同じソースファイルを対象として searcher2038 を実行し，結果の比較を行う．

対象としたのは，FreeBSD 12.1-RELEASE の，date，stat，touch の 3 つのコマンドのソースファイルである，date.c，stat.c，touch.c である．これらのソースファイルを入力として searcher2038 を実行し，出力された修正必要箇所と，先行研究の手法による修正箇所との間で比較を行った．図 8，9，10 は，それぞれ date.c，stat.c，touch.c に対して実行した，searcher2038 の出力である．

5.1.2　結果

表 3 に，大江らの先行研究 [9] の手法による修正箇所と，searcher2038 が検出した修正が必要な箇所の個数を記している．3 つの対象ファイルのいずれについても，先行研究 [9] による修正と比較して，多数の修正箇所を検出している．直接評価箇所については，[9] では 3 ファイルのいずれでも一つも修正が行われていないのに対し，searcher2038 では全ファイルで検出されている．

次に，実際に検出された箇所を確認し，先行研究 [9] の手法における修正箇所と一致しているかを確かめた．表 3 には，[9] の手法による手動での修正箇所を正解とした，結果の適合率と再現率をまとめている．

直接評価箇所については，先行研究の手法による修正箇所が存在しないため，再現率は定義できない．外部関数の呼び出し箇所と，合計に関しては，再現率は 100%となった．一方，適合率は 50%を切る低い値となっている．このように，searcher2038 は，修正が必要な箇所を全て検出することに成功しているが，同時に不要な箇所を多数検出していることがわかった．

```
Result : The following are points may be fixed

=== Callings those may be fixed =================

date.c line157 <strtoq>
    < tval strtoq >

date.c line191 <time>
    < tval time >

date.c line219 <localtime>
    < tval lt localtime >

date.c line287 <localtime>
    < tval lt localtime >

date.c line368 <mktime>
    < tval lt mktime >

date.c line373 <netsettime>
    < tval netsettime >

date.c line222 <vary_apply>
    < lt badv vary_apply >

date.c line241 <strftime>
    < lt strftime >

date.c line293 <strptime>
    < lt t strptime >

date.c line224 <fprintf>
    < badv fprintf >

date.c line299 <fprintf>
    < t fprintf strlen >

date.c line301 <strlen>
    < t fprintf strlen >

=== Comparings those may be fixed ===============

date.c line220 <lt>

date.c line288 <lt>

date.c line294 <t>

date.c line298 <t>

=== End ========================================
```

図 8: date.c に対する実行結果

```
Result : The following are points may be fixed

=== Callings those may be fixed =================

stat.c line760 <localtime>
    < tm localtime >

stat.c line765 <strftime>
    < tm strftime >

stat.c line763 <localtime>
    < tm tv_sec localtime ts >

stat.c line987 <snprintf>
    < tv_sec ts snprintf >

stat.c line1040 <snprintf>
    < tv_sec ts snprintf l >

=== Comparings those may be fixed ===============

stat.c line761 <tm>

=== End ========================================
```

図 9: stat.c に対する実行結果

```
Result : The following are points may be fixed

=== Callings those may be fixed =================

touch.c line272 <mktime>
    < tv_sec t tvp mktime >

touch.c line305 <mktime>
    < tv_sec t mktime tvp >

touch.c line348 <mktime>
    < tv_sec t mktime tvp timegm >

touch.c line348 <timegm>
    < tv_sec t mktime tvp timegm >

touch.c line227 <time>
    < now time >

touch.c line228 <localtime>
    < now t localtime >

touch.c line289 <time>
    < now time >

touch.c line290 <localtime>
    < now t localtime >

touch.c line328 <strptime>
    < t p strptime >

=== Comparings those may be fixed ===============

touch.c line228 <now>

touch.c line290 <now>

touch.c line329 <p>

touch.c line332 <p>

touch.c line332 <p>

touch.c line332 <p>

touch.c line341 <p>

touch.c line345 <p>

=== End ==========================================
```

図 10: touch.c に対する実行結果

表 3: 結果の比較

	先行研究の手法による修正箇所			searcher2038 による特定箇所		
	外部関数呼出	直接評価箇所	合計	外部関数呼出	直接評価箇所	合計
date.c	3	0	3	11	4	15
stat.c	3	0	3	9	8	17
touch.c	2	0	2	5	1	6

	外部関数呼出		直接評価箇所		合計	
	適合率	再現率	適合率	再現率	適合率	再現率
date.c	0.27	1.00	0.00	N/A	0.20	1.00
stat.c	0.33	1.00	0.00	N/A	0.18	1.00
touch.c	0.40	1.00	0.00	N/A	0.33	1.00

5.1.3 結果の考察

5.1.2 項で示したように，今回の評価において，searcher2038 は，再現率は高かったものの，適合率は低かった．この結果について，修正が必要と誤って検出された箇所のソースコードを確認し，原因の推定を行った．

はじめに，外部関数呼び出しの誤検出箇所は，時刻情報と文字列の間の変換を行う関数の呼び出しを行う箇所が主であった．これに該当する誤検出箇所は，C 言語の標準ライブラリ関数である，strptime，strftime，strtoq の呼び出し箇所である．strptime は，文字列を tm 構造体に変換する関数である．strftime は逆に，tm 構造体を文字列に変換する関数である．strtoq は，文字列を整数に変換する関数である．これらは，時刻情報を扱っているが，引数の表す値（時刻情報）を変更することなく，型を変えて返している．このため，呼び出しを修正せずとも，問題は発生しない箇所である．図 11 に，date.c で確認された，時刻情報と文字列の間の変換を行う関数の呼び出しを行う誤検出箇所の例を示している．searcher2038 では，241 行目の，時刻情報を扱う変数 lt を引数とする strftime 関数の呼び出しが，修正必要箇所として検出された．

次に，時刻情報の直接評価の誤検出箇所は，NULL との比較を行っている箇所が主であった．Understand での構文解析時，NULL は識別子として扱われる．また，時刻情報を扱う変数ではないため，searcher2038 においては，修正が必要な比較箇所として判定されてしまう．しかし，変数が NULL であるかどうかは，起算点変更が行われた前後に影響されないため，修正を行わずとも，問題は発生しない箇所である．図 12 に，date.c で確認された，時刻情報を NULL と比較する誤検出箇所の例を示している．searcher2038 では，220 行目の，時刻情報を扱う変数 lt の NULL との比較が，修正必要箇所として検出された．

また，touch コマンドでは，コマンドの引数として与えられた時刻を扱う関数内の箇所で，誤検出が多く見られた．touch コマンドは，引数として時刻を受け取ることができる．これを扱うための関数では，tm 型などの時刻変数が出現するため，複数の修正必要箇所が検出された．しかし，これらの箇所は，手動での特定では修正対象とはなっていない．図 13 に，

```
 93.  int
 94.  main(int argc, char *argv[])
 95.  {
            ...
241.        (void)strftime(buf, sizeof(buf), format, lt);
242.        printdate(buf);
243.  }
```

図 11: 時刻情報と文字列を変換する誤検出箇所の例

```
93.  int
94.  main(int argc, char *argv[])
95.  {
                ...
219.        lt = localtime(&tval);
220.        if (lt == NULL)
221.              errx(1, "invalid time");
                ...
```

図 12: 時刻情報と NULL と比較する誤検出箇所の例

```
219.  static void
220.  stime_arg1(const char *arg, struct timespec *tvp)
221.  {
              ...
227.        now = time(NULL);
228.        if ((t = localtime(&now)) == NULL)
229.              err(1, "localtime");
              ...
```

図 13: コマンドの引数として与えられた時刻を扱う関数内の誤検出箇所の例

touch.c の，コマンドの引数として与えられた時刻を扱う stime_arg1 関数内の誤検出箇所の例を示している．searcher2038 では，227 行目の time 関数，228 行目の localtime 関数の呼び出しが，それぞれ修正必要箇所として検出された．

これらの箇所の誤検出は，searcher2038 が，ソースコードの意味解釈を行わないことが原因であると考えられる．手動での修正箇所特定では，上記のような，形式上は修正必要箇所と同じでも修正の不要な箇所は，ソースコードの意味解釈を行うことで判別が行える．しかし，searcher2038 では，詳細な意味解釈を行わずに修正箇所特定を行うため，これらを区別できなかった．

6　まとめ

本研究では，先行研究 [8, 9] の手法に基づく 2038 年問題に対するプログラム修正のために，修正が必要な箇所の特定を行うツール，searcher2038 を作成した．searcher2038 では，[9] の手法による手動での特定と比較し，同じ修正箇所を特定することに成功した．

searcher2038 による修正箇所の特定は，再現率は高いものの，適合率が低い結果となった．このため，2038 年問題への対応としてプログラム修正を行うには，実際に修正が必要な箇所の絞り込みをさらに行う必要がある．このため，修正箇所の特定を手動で行う必要があるという，先行研究 [9] の課題点を完全に克服するには至っていない．今後の課題として，searcher2038 の修正箇所特定の精度をより高め，手動での確認を不要とすることがあげられる．

また，今回 5 節で行った評価で対象としたソースファイルでは，高い再現率を示すことができた．しかし，searcher2038 の手法では，関数を跨いだ時刻情報の追跡精度に課題があるため，より大規模なソースファイルにおいても同様の性能を発揮できるかについては疑問が残る．この点について調査を行い，精度の低下が見られた場合は，searcher2038 の改良を行いたい．

searcher2038 の改善点に加え，先行研究のもう 1 つの課題点として，修正の実行を手動で行う必要がある点が残されている．searcher2038 の精度向上とともに，修正の実行を自動化するツールを作成したい．

謝辞

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上克郎 教授には，ご多忙の中，研究活動に対して多くの貴重な御助言や御指導を賜りました．心より深く感謝申し上げます．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠 准教授には，研究の着想から研究活動の直接の御指導，論文の執筆に至るまで，あらゆる場面で多くの御指導を賜りました．松下誠 准教授の適切な御指導により，本論文を完成させることができました．心より深く感謝申し上げます．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 神田哲也 助教には，発表の問題点の提示において，多くの御助言を賜りました．心より深く感謝申し上げます．

大阪大学大学院情報科学研究科コンピュータサイエンス専攻 伊藤薫 氏，嶋利一真 氏には，研究室での生活や，研究活動において，日々様々な御支援や御助言を賜りました．心より深く感謝申し上げます．

最後に，その他様々な御指導，御助言等を賜りました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に，心より深く感謝申し上げます．

参考文献

[1] time(3), FreeBSD 12.1-RELEASE Library Functions Manual (2003).

[2] 鈴木孝知, 中村建助: 「西暦 2038 年問題」でトラブル相次ぐ, 日経コンピュータ（2004-4-1）, http://tech.nikkeibp.co.jp/it/members/NC/ITARTICLE/20040325/1/ （2021 年 2 月 9 日閲覧）.

[3] 横田隆夫：西暦 2000 年問題の意味と対応策，コンピュータソフトウェア，Vol.13, No.5,pp.412-419 (1996).

[4] 喜多充成，久保信明: 4 月 7 日（日本時間）に 2 度目の「GPS 週数ロールオーバー」 — 衛星測位入門 — みちびき（準天頂衛星システム）(2019-2-25）, https://qzss.go.jp/column/gps-rollover_190225.html （2021 年 2 月 9 日閲覧）.

[5] コンピュータ西暦２０００年問題 に関する報告書, 内閣コンピュータ西暦二千年問題対策室（2000-3-30）, https://www.kantei.go.jp/jp/pc2000/houkokusyo/honbun.html （2021 年 2 月 9 日閲覧）.

[6] 贄川俊: ２０年に１度ＧＰＳ時間リセット 前回は混乱，今回は？ — 朝日新聞デジタル（2019-4-6）, https://www.asahi.com/articles/ASM423RZ2M42UTIL00W.html （2021 年 2 月 9 日閲覧）.

[7] 「GPS 週数ロールオーバー」に関する重要なお知らせ＜バイク専用ポータブルナビゲーション＞ — Yupiteru, https://www.yupiteru.co.jp/corp/important/190405_bikenavi.html （2021 年 2 月 9 日閲覧）.

[8] 大江秀幸, 安藤友康, 松下誠, 井上克郎:組込み機器開発における 2038 年問題への対応事例, 情報処理学会デジタルプラクティス Vol.10 No.3, pp.603–620 (2019).

[9] 大江秀幸, 松下誠, 井上克郎: 32bit UNIX システムの 2038 年問題に対するプログラム修正法の提案, 情報処理学会論文誌 (2021, to appear).

[10] Understand^{TM} by SciTools, https://www.scitools.com/ （2021 年 2 月 9 日閲覧）.

[11] The FreeBSD Project, https://www.freebsd.org/ （2021 年 2 月 9 日閲覧）.

[12] 鈴木淳也: Windows 10 は全て 64bit になる 32bit から 64bit への完全移行は間もなく — ITmedia PC USER（2020-5-20）,

https://www.itmedia.co.jp/pcuser/articles/2005/20/news062.html （2021 年 2 月 9 日閲覧）.

[13] GPS Week Number Rollover — Naval Oceanography Portal, https://www.usno.navy.mil/USNO/time/gps/gps-week-number-rollover （2021 年 2 月 9 日閲覧）.

[14] コンピュータ西暦 2000 年問題 — 首相官邸, https://www.kantei.go.jp/jp/pc2000/（2021 年 2 月 9 日閲覧）.

[15] Frank Tip: A Survey of Program Slicing Techniques, Journal of Programming Languages 3, pp.121–189 (1995).

[16] CTIME(3), FreeBSD 12.1-RELEASE Library Functions Manual (1999).

[17] Welcome to Python.org, https://www.python.org/ （2021 年 2 月 9 日閲覧）.

[18] subprocess — Subprocess management — Python 3.9.1 Documentation, https://docs.python.org/3/library/subprocess.html （2021 年 2 月 9 日閲覧）.

[19] The DOT Language — Graphviz - Graph Visualization Software, http://www.graphviz.org/doc/info/lang.html （2021 年 2 月 9 日閲覧）.