UNIXの2038年問題に対する問題箇所特定ツール

水上 陽向[1,a)]　松下 誠[1,b)]　井上 克郎[1,c)]

概要：UNIX ベースシステムでの 2038 年問題とは，時刻を表す符号付 32bit の time_t 型がオーバーフローを起こすことによる問題である．これにより，32bit アーキテクチャを用いるシステムでは，種々の不具合が予想されている．本研究では，先行研究で提案された，時刻起算点の変更を用いた 2038 年問題への対応方法に対して，具体的な実装を行い，問題への対応作業を省力化するツールを作成した．作成ツールは，プログラム内の修正必要箇所の特定・出力を行う．また先行研究での修正箇所特定結果と比較することで，作成ツールの評価を行った． FreeBSD のコマンドのソースファイルを対象に比較を行い，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．さらに，ソースコードの修正を手動で行う必要があるという課題に取り組むため，自動修正機能の設計を行った．

キーワード：2038 年問題，UNIX 時刻，データフロー解析

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．UNIX ベースのシステムにおける 2038 年問題は，この time_t 型の変数が 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．

この 2038 年問題に対し，大江らの先行研究 [3] 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，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 年問題に対応する手法を提案している．また大江らは，その手法に対し，修正箇所の特定を効率よく行う手法に関する研究も行っている [4]．

本研究では，これらの先行研究 [3], [4] の手法に対して，具体的な実装を行い，2038 年問題への対応の省力化を図るツール searcher2038 を作成した．searcher2038 では，[3] 及び [4] 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．

searcher2038 に対し，大江らによる先行研究 [4] 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．オープンソースの UNIX ベースシステムである，FreeBSD[6] 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．

また，ソースコードの修正を手動で行う必要があるという課題に取り組むため，自動修正機能の設計を行った．自動で特定を行った修正箇所に対し，ソースコードの書き換えを行い，修正を完了させる．外部関数呼び出し箇所に対しては，先行研究 [3], [4] の手法に沿って，ラッパー関数の作成と，呼び出し箇所の書き換えによって修正を行う．時刻情報の比較を行う箇所に対しては，比較を行う変数から起算点変更分の減算を行った上で比較するよう，コードを書き換えて修正を行う．

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について，また 4 節ではその実装について述べる．5 節では searcher2038 の評価を行う．6 節では自動修正機能の設計概要について記す．最後に，7 節でまとめと今後の課題について述べる．

[1] 大阪大学大学院情報科学研究科
1-5 Suita, Osaka 565–0871, Japan
[a)] h–mizukm@ist.osaka-u.ac.jp
[b)] matusita@ist.osaka-u.ac.jp
[c)] inoue@ist.osaka-u.ac.jp

2. 背景

2.1 2038 年問題

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である符号付 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い，誤動作の発生が想定される．

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**

現在では，システムの 64bit への移行が進んでいる [7]．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，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．

2.2 先行研究と課題

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 [3] 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．

表 2 起算点変更を行った場合の time_t 値の推移

時刻 (UTC)	**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

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 [4] 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 [8] 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．[4] 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[6] 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．

ただ，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 [4] 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，修正箇所の特定に関わる手間を軽減できるツールの作成を目指す．

3. プログラム修正手法

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，[3], [4] 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[5] 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．

3.2 時刻起算点の変更

2.2 節で述べたように，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．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，関数単位で制御フローグラフとデータ依存の情報を構築する．

3.4 修正箇所の特定

本研究では，大江らの先行研究 [4] 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 節で述べる．

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．

変更前	変更後
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){
	return mktime(ptm) - *defsec*;
	}

図 1 時刻情報を扱う外部関数呼び出しの例

図 1 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 [9]．大江らの先行研究 [4] 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 1 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，*defsec* は，時刻起算点変更年数に相当する秒数を表す定数である．大江ら [4] は，引数の減算を行ってから mktime 関数の呼び出しを行うラッパー関数を作成しているが，この例では，引数はそのままに mktime 関数を呼び出し，その返り値を減算してから返す形で作成している．

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．

変更前	変更後
struct tm timer;	struct tm timer;
int y;	int y;
timer=gmtime(time(NULL)); →	timer=gmtime(time(NULL));
timer.tm_year + = *defyear*;	timer.tm_year + = *defyear*;
…	…
if(timer.tm_year > y){	if(timer.tm_year − *defyear* > y){
…	…
}	}

図 2 時刻情報の直接比較の例

図 2 の左側は，起算点変更を行った時刻情報を，int 型変数と比較しているコード片の例である．また，図 2 の右側は，これに対して修正を行った例である．ただし，*defyear* は，起算点変更分の年数を表す定数である．比較を行う値に，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[10] バージョン 3.9 を用いて行った．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．また，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

図 3 に，処理の流れの概図を示している．

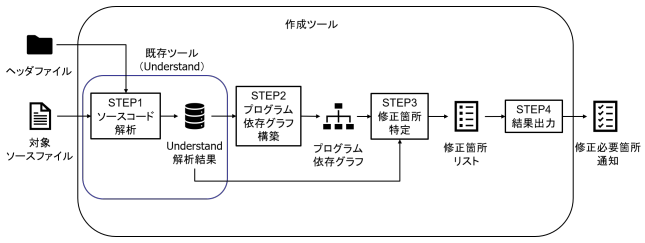

図 3 処理の流れ

searcher2038 は，ソースコードの解析，プログラム依存グラフの構築，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，プログラム依存グラフの構築は，制御フローグラフの構築を行う段階と，それにデータ依存情報の付加を行う段階に分けられる．また，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．

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．ソースファイルは 1 個以上，インクルードパスは 0 個または 1 個の入力が可能である．

4.4 STEP1: ソースコード解析

対象ソースファイルの解析は，Understand [5] 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．

4.4.1 Understand を用いた解析

Python の標準モジュールである，subprocess モジュール [11] 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．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．

4.4.2 解析結果の保存

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: プログラム依存グラフ構築

この手順では，制御フローグラフに対し，変数のデータ依存情報を付加したグラフの作成を行う．本稿では，このグラフをプログラム依存グラフと呼称する．

4.5.1 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 [12] 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．

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，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，の情報を保持するインスタンスとして作成される．

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．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．

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，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），の情報が含まれている．

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，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 インスタンスを保持させる．

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．

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．出現箇所が，ノードの属するファイルと同じファイルで，ノード

インスタンスが保持する行・列の範囲内にあれば，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．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．

4.5.2 STEP2-2: 制御フローグラフに対するデータ依存情報の付加

データ依存情報は，4.5.1 節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．データ依存情報は，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．

以上で，プログラム依存グラフの構築が完了する．

4.6 STEP3: 修正箇所特定

4.6.1 STEP3-1: 修正候補の抽出

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を，4.5 節で作成した，プログラム依存グラフを用いて調べる．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，4.5.1 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．

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の特定・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．

C 言語標準の型を持つ時刻変数

はじめに，C 言語標準の時刻型を持つ変数の抽出を行う．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行う．

依存関係によって生じる時刻変数

次に，既に発見された時刻変数を参照する変数の抽出を行う．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．変数間の参照情報は，4.5 節で構築した，プログラム依存グラフから取り出す．

ここで述べた処理が終了した時点で，特定すべき全ての時刻変数を格納した時刻変数のリストが完成した状態となる．

4.6.2 STEP3-2: 修正候補からの絞り込み

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．

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，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，の情報を保持させる．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．

4.6.1 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．

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，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，の情報を保持させる．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．

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．

4.7 STEP4: 結果出力

4.6.2 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，4.6.2 節で作成した外部関数呼び出しのリストと，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 節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，簡単なプログラムに対する出力を，図 4 に示している．

```
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 4 出力例

5. 評価

searcher2038 の評価として，先行研究 [4] による修正結果との比較を行った．

5.1 対象と手法

本研究では，先行研究 [4] 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．評価対象は，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．

5.2 結果

表 3 に，大江らの先行研究 [4] 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 [4] 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，[4] 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 [4] 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，[4] 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．

5.3 結果の考察

5.2 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．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．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，struct 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．

これらの箇所の誤検出は，searcher2038 が，ソースコー

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

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. 自動修正機能の設計

本研究で作成したツール searcher2038 では，2038 年問題への対応のために，修正が必要な箇所の特定を行うことができた．しかし，先行研究の課題点として，修正の実行を手動で行う必要がある点が残されている．今後の課題として，特定した修正箇所の修正を自動で行うことができるツールの作成が挙げられる．本節では，修正の自動化を行うための方針を述べる．

6.1 外部関数呼び出し箇所の修正

時刻情報を扱う外部関数呼び出し箇所の修正は，大江らの手法に従い，ラッパー関数の作成と，呼び出し箇所の書き換えによって行う．この手法を自動化するために，全体を呼び出し箇所の特定と関数のリスト化，ラッパー関数の作成と呼び出し文の書き換え，の 2 つに分割して考える．

なお，2038 年問題への対応を行う上では，標準ライブラリ関数の呼び出しと，ユーザー定義の外部関数の呼び出しの両方の修正が必要となる．しかし，ユーザー定義関数については，静的解析での，関数の入出力仕様や関数の機能の正確な判定が難しく，自動での修正の対象とすると，十分な精度・再現率を実現することが困難になると予想される．そのため，修正の自動化においては，標準ライブラリ関数の呼び出し箇所のみを修正対象とする．

6.1.1 呼び出し箇所の特定と関数のリスト化

ここでは，予め時刻情報を扱うための標準ライブラリである，time.h 内の関数を調査し，修正対象となる関数の一覧を作成する．大江らの研究 [3] でも同様の関数の調査が行われているため，この結果を参考にして改めて関数の処理内容を調査し，time_t 型時刻情報と，他の型の時刻情報の間の変換が行われる関数を修正対象の関数とする．具体的には，以下の関数が該当する．

- ctime，ctime_r（time_t 型 → 文字列型）
- gmtime，gmtime_r（time_t 型 → tm 構造体）
- localtime，localtime_r（time_t 型 → tm 構造体）
- mktime（tm 構造体 → time_t 型）
- gmtime（tm 構造体 → time_t 型）
- timelocal（tm 構造体 → time_t 型）

Understand[5] の解析情報を用いて，対象ソースコード内から，これらの修正対象ライブラリ関数の呼び出し箇所を特定する．また，ラッパー関数の作成を行うために，対象ソースファイルそれぞれに対し，そのファイル内で呼び出されている修正対象ライブラリ関数のリストを作成しておく．

6.1.2 ラッパー関数の作成と呼び出し文の書き換え

続いて，修正対象ライブラリ関数の呼び出し箇所情報をもとに，ラッパー関数の作成と，呼び出し文の書き換えを行う．修正は，3.4.1 節や，図 1 に示したように，大江らの手法に従い，修正対象関数の引数や戻り値に対して足し引きを行うラッパー関数を作成した上で，呼び出し箇所をラッパー関数の呼び出しへと書き換える．

修正対象関数の処理内容に基づき，ラッパー関数は，以下のような処理を行う関数として作成する．

- 与えられた引数に，修正年数に相当する秒数を加えて，それを引数として修正対象関数を呼ぶ．（ctime，ctime_r）
- 与えられた引数で修正対象関数を呼び，その返り値に含まれる年情報に，修正年数を加えて返す．（gmtime，gmtime_r，localtime，localtime_r）
- 与えられた引数で修正対象関数を呼び，その返り値から，修正年数に相当する秒数を引いて返す．（mktime，timegm，timelocal）

対象ソースファイルそれぞれに対し，その内部に呼び出しが存在する修正対象関数のラッパー関数の作成を行い，既存の修正対象関数の呼び出し箇所を，全て対応するラッパー関数の呼び出しに置換する．

6.2 直接評価箇所の修正

起算点変更後時刻情報の直接評価箇所の修正は，大江ら [4] の手法に従い，時刻起算点の変更を考慮した評価へと変更することによって行う．具体的には，起算点変更後の年情報を扱う変数の代わりに，その変数から変更年数を引いた値を比較に用いることで，正しい評価が行えるよう修正する．この手法を自動化するために，以下の手順に分割する．

(1) 追跡が必要な時刻情報を扱う変数を特定する
(2) 変数の追跡を行い，修正すべき評価箇所を特定する
(3) 評価箇所の書き換えを行う

なお，ctime 関数で変換した場合のように，時刻情報は文字列型で扱われることがある．この場合も，比較は正常に行えないため，修正が必要となるが，文字列の修正は，単純な加減算では行えないため，自動化が難しい．よって，これらの箇所は，自動での修正は行わず，修正が必要な旨を出力するのみにとどめる．

6.2.1 追跡が必要な変数の特定

修正が必要な評価箇所を特定する上で，追跡が必要な変数は，時刻起算点の変更処理が行われた，時刻を扱う変数である．これに該当するのは，6.1 節の内容で作成された，ラッパー関数から返された変数となる．

また，4.6.1 節と同様に，追跡中の変数の値を参照して値が変更された変数も，追跡対象とする必要がある．これらについては，次項の追跡中に，随時特定と追加を行っていく．

6.2.2 変数の追跡と評価箇所の特定

起算点変更を行った時刻情報を扱う変数の追跡は，4.6.1 節と同様に，制御フローグラフを辿ることで行う．追跡の実行中には，評価箇所の特定と，追跡中の値を参照した他の変数の調査を行う必要がある．これらはそれぞれ，Understand[5] の解析情報から，変数の参照情報と構文解析の情報を用いて特定する．また，追跡中の変数に，他の変数や定数の値が代入された場合など，起算点変更後の時刻を表さなくなった場合は，そこで追跡を終了する．

6.2.3 書き換えの実行

直接評価箇所の修正は，上述の通り，図 2 のように，時刻情報を扱う変数を減算してから比較するよう，コードを書き加えることで行う．

7. まとめ

本研究では，先行研究 [3], [4] 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，[4] 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．また，修正の実行を自動化するツールを実現するため，自動修正機能の設計を行った．

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．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確認できていない．この点についてさらなる調査を行い，精度の低下が見られた場合は，searcher2038 の改良を行いたい．さらに，6 節で述べた自動修正機能の実装も今後の課題としてあげられる．

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] 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).
[4] 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).
[5] Understand^{TM} by SciTools, https://www.scitools.com/（2021 年 2 月 9 日閲覧）.
[6] The FreeBSD Project, https://www.freebsd.org/（2021 年 2 月 9 日閲覧）.
[7] 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）.
[8] Frank Tip: A Survey of Program Slicing Techniques, Journal of Programming Languages 3, pp.121–189 (1995).
[9] CTIME(3), FreeBSD 12.1-RELEASE Library Functions Manual (1999).
[10] Welcome to Python.org, https://www.python.org/（2021 年 2 月 9 日閲覧）.
[11] subprocess — Subprocess management — Python 3.9.1 Documentation, https://docs.python.org/3/library/subprocess.html（2021 年 2 月 9 日閲覧）.
[12] The DOT Language — Graphviz - Graph Visualization Software, http://www.graphviz.org/doc/info/lang.html（2021 年 2 月 9 日閲覧）.