Software Engineering Laboratory

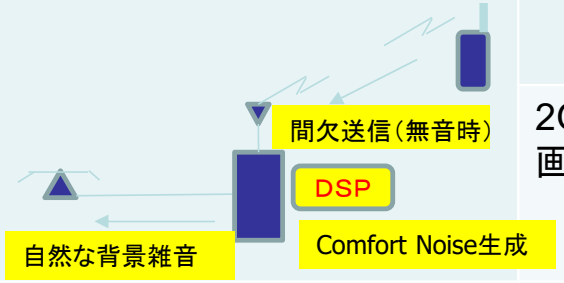
組込みシステムの開発事例に基づく効率的な改良保守に関する研究

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
大江　秀幸

目次

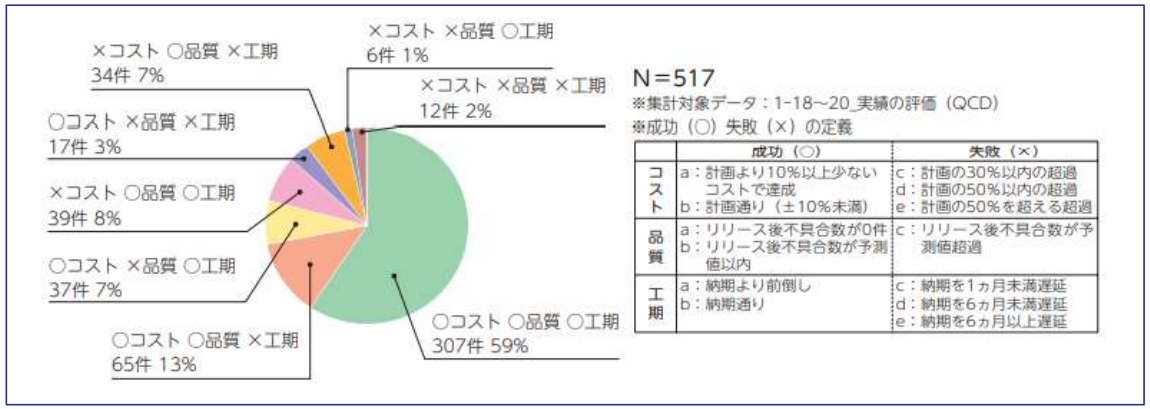
1. はじめに

1．1　経歴紹介

年	社名	主な開発対象システム	開発環境等	職位・役割等
1991～2002	NECテレコムシステム株式会社 （現NEC通信システム株式会社） 	電子ファイリングシステム	OS/2、 Windows3.1/ NT C言語	担当
		2G/3G携帯電話システムでの音声・画像CODECカード開発	Linux DSP アセンブラ C言語	主任、 PL
2002～2018	松下AVCマルチメディアソフト株式会社 （現パーソルAVCテクノロジー株式会社） 主に組込みソフトウェア開発に従事	AV家電  	µItron、Linux、 LTSA、 VDM++、 C/C++言語	チームリーダ、PM、 PL、CMM/CMMI 社内アセッサ
		車載ECU 	Autosar、 C/C++言語	アシスタントGM、 PL
2018～現在	関西デジタルソフト株式会社	無人受付端末	Windows10、 C#	課長、アーキテクト、 PM、PL

1．2　研究の動機・目的

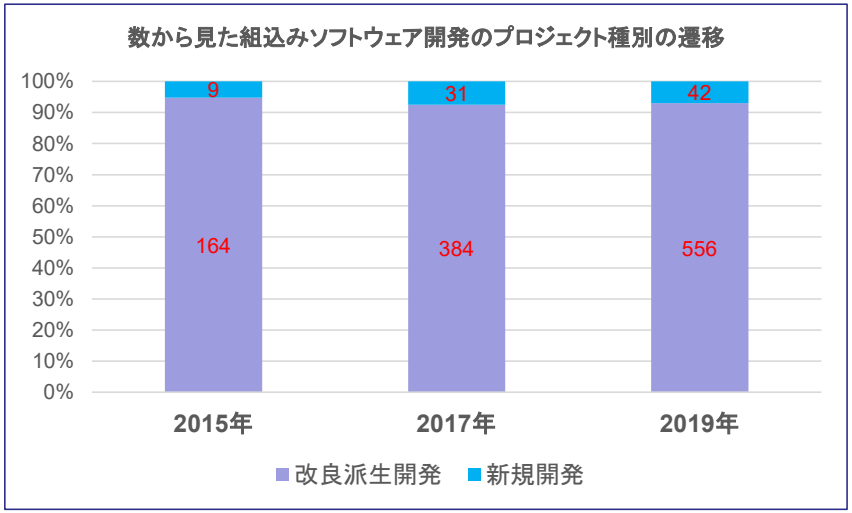
組込みソフトウェアのQCD別の実績評価



	成功（○）	失敗（×）
コスト	a：計画より10%以上少ないコストで達成 b：計画通り（±10%未満）	c：計画の30%以内の超過 d：計画の50%以内の超過 e：計画の50%を超える超過
品質	a：リリース後不具合数が0件 b：リリース後不具合数が予測値以内	c：リリース後不具合数が予測値超過
工期	a：納期より前倒し b：納期通り	c：納期を1ヵ月未満遅延 d：納期を6ヵ月未満遅延 e：納期を6ヵ月以上遅延

IPA 組込みソフトウェア開発データ白書2019より

40%以上がQCDいずれかで「失敗」との評価



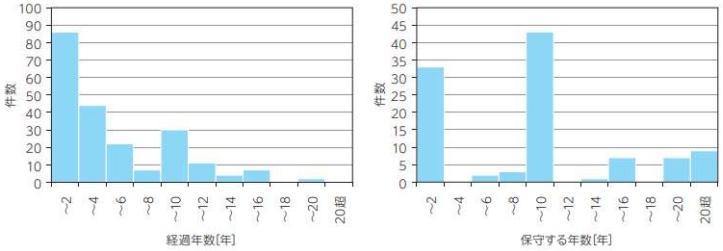
IPA 組込みソフトウェア開発データ白書2015, 2017, 2019を基に作成

90%以上が「保守」プロジェクトで傾向変わらず

実務経験上、「保守」には新規開発には無い困難さ（特に修正箇所、影響範囲の特定）あり

「保守」特有の開発の困難さを解消したい

1．3　組込みシステムの特徴
研究対象システム



組込みソフトウェア開発データ白書2017より

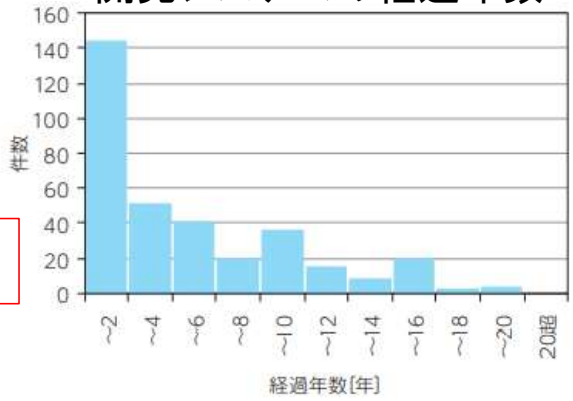
JR東でSuica登場
ウィキペディア発足
東京ディズニーシー／USJ開園
サイバーショットP1
ムシキング
ブロードバンド（流行語）
A.I.（洋画）
千と千尋の神隠し
ヒカルの碁

同じ傾向

これまで保守されていた期間

保守必要な期間
（製品寿命）

開発システムの経過年数



N＝339
※集計対象データ：1-5a_経過年数

開発システムの保守年数

N＝214
※集計対象データ：1-5b_保守する年数

組込みソフトウェア開発データ白書2019より

「ふた昔」前のシステムも稼働。一般に10年程度の保守が必要。

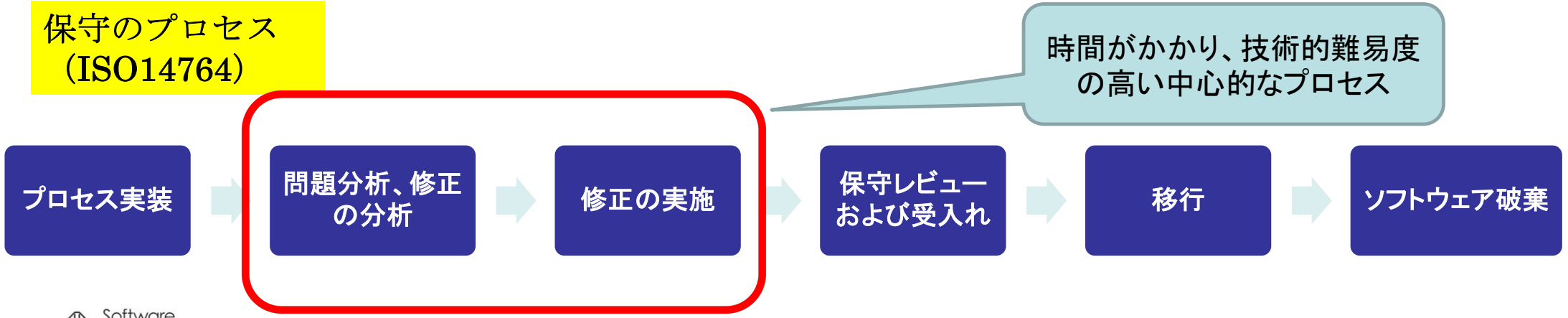
1．4　ソフトウェア保守について
研究のスコープ

ソフトウェア保守＝運用中のソフトウェアの改変

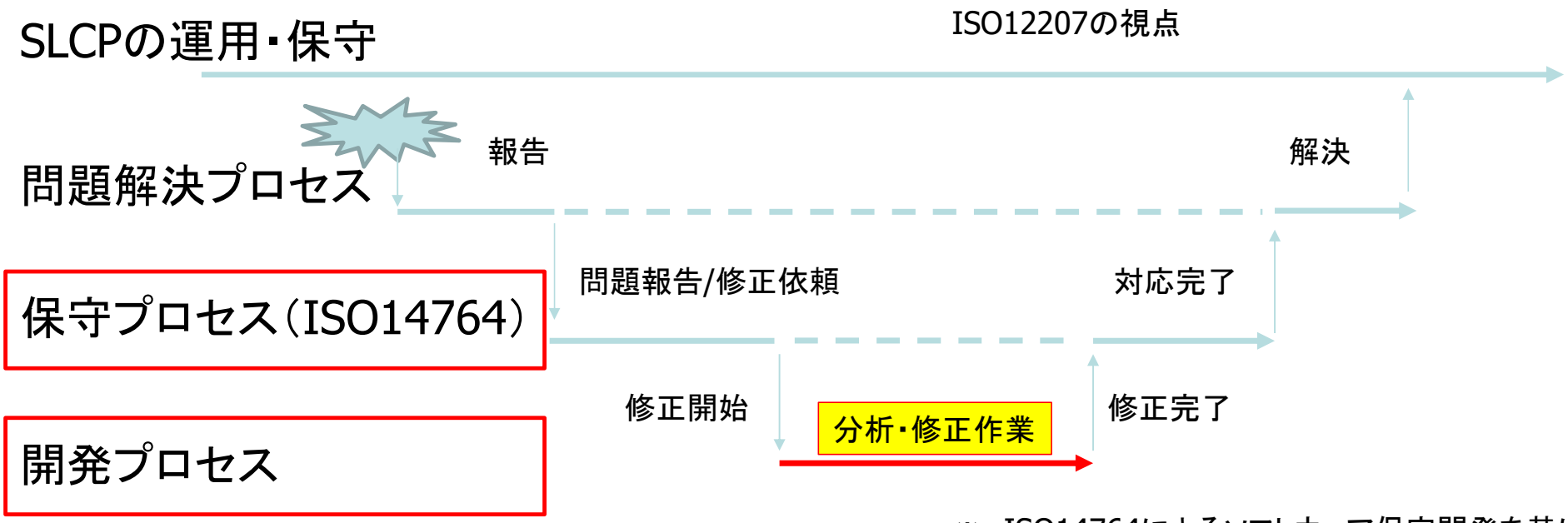
保守の分類（ISO14764）

訂正保守（従来の保守）	改良保守（進化・発展につながる保守）
是正保守	適応保守
緊急保守	完全化保守
予防保守	—

保守のプロセス（ISO14764）



1．5　保守プロセスの位置づけ



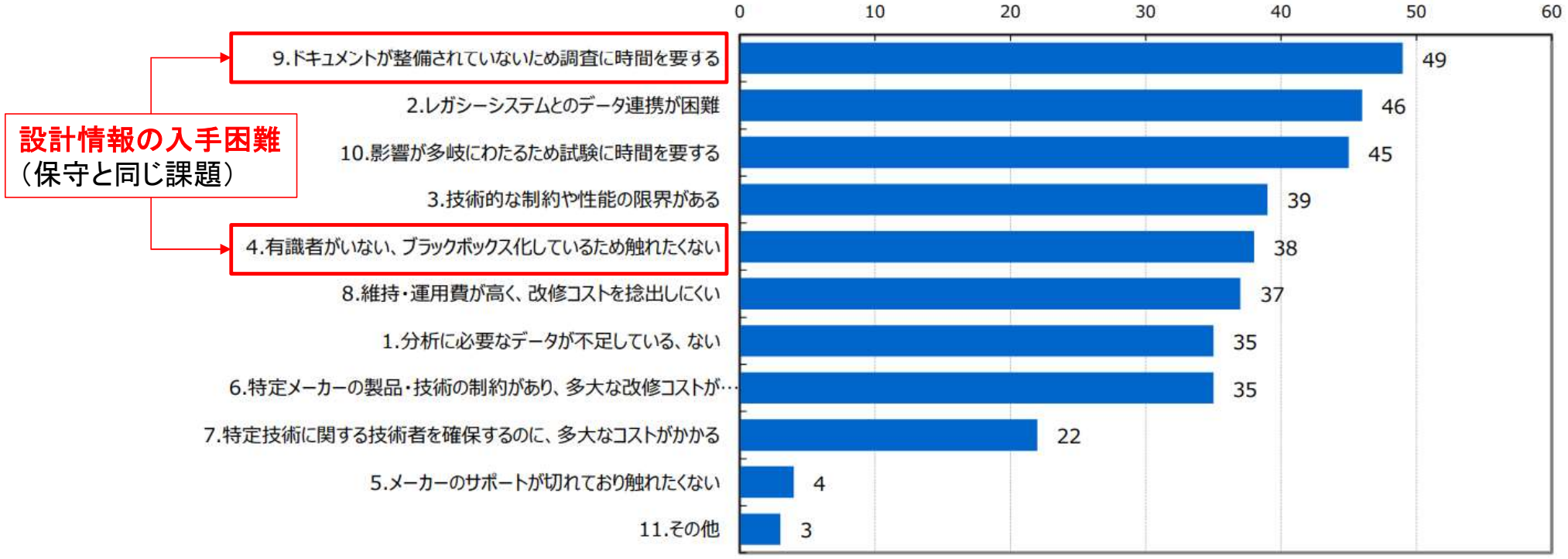
※　ISO14764によるソフトウェア保守開発を基に作成

保守プロセスは開発プロセスを呼出す＝保守特有の技術解の定義なし

（新規開発と同じ技術で保守を行う）

1．6　保守に関する近年の動向
DXとの関係

レガシーシステムが足かせと感じる理由⇒ DX（デジタルトランスフォーメーション）化を阻害

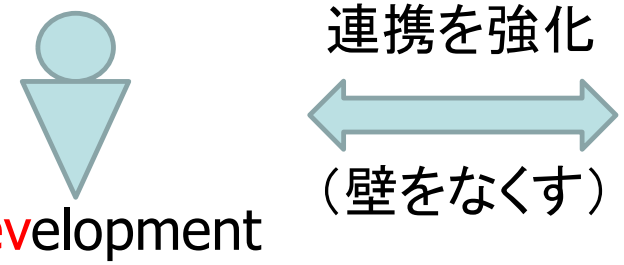


一般社団法人日本情報システム・ユーザ強化「デジタル化の進展に対する意識調査」（平成29年）より引用

1．6　保守に関する近年の動向
DevOpsとの関係

DevOps　目標：組織文化の醸成を通して、市場投入までの時間短縮、リリース品質向上などを目指す

開発部門と運用部門が協力して、継続的にユーザ要求の早期実現（開発）と安定稼働（運用・保守）を同時に実現する、組織としての改善活動



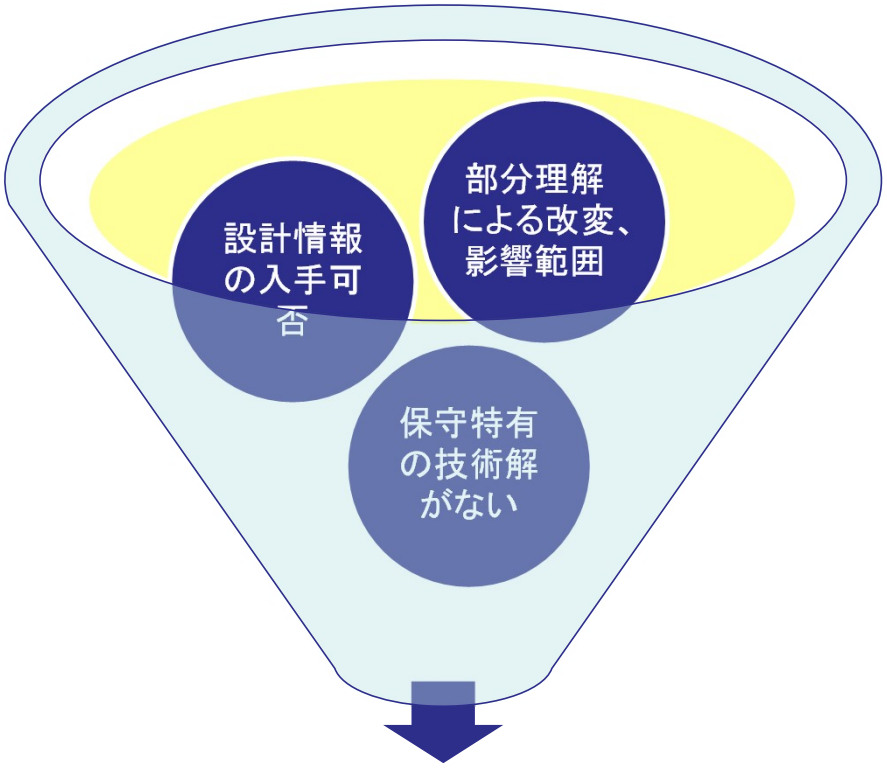
各種ツール　継続的インテグレーション　リリース自動化　コミュニケーション　etc.

https://ja.wikipedia.org/wiki/DevOps より転記

本研究が対象とする、稼働中のシステムの分析・修正技術とは直接の関係はない

1．7　研究課題

よく知られた保守の課題



保守への各種開発技術の適用手順、方法の整理が必要

1．8　研究業績との対応

2章（設計情報が利用できない場合の保守）

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”，デジタルプラクティス，10(3)，pp. 588-602（2019-07）.

3章（設計情報が利用できない場合の保守）

- 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, Vol.62, No.4, pp.1051-1055 (2021-04).

4章（設計情報が利用できる場合の保守）

- 大江秀幸, 松下誠, 井上克郎, “予約受付端末の他サービス転用に向けたソフトウェア改造事例”（投稿準備中）

Software Engineering Laboratory

２．組込み機器開発における2038年問題への対応事例

設計情報が利用できない場合の保守

Software Engineering Laboratory

2.1 2038年問題について

2．1　2038年問題について

2038年問題とは

問題現象

2038年1月19日3時14分7秒

↓ 1秒後

①2038年1月19日3時14分8秒
②1970年1月1日0時0分0秒*

①実際の時刻 (UTC)
②システム時刻（UTC）

発生メカニズム

1970年1月1日0時0分0秒 (epoch)	
time_t型の値	0x00000000

↓

2038年1月19日3時14分7秒	
time_t型の値	0x7FFFFFFF

↓

2038年1月19日3時14分8秒	
time_t型の値	0x80000000

“time_t”型で扱う時刻情報は、毎秒カウントアップ
32bitシステムでは “*32bit signed int*” 型で定義

*** 時刻が不正となる例.**

2．2　解決手法

2．2 解決手法
対象システムの例

外部システムから表示情報取得
（ブロードキャスト信号）

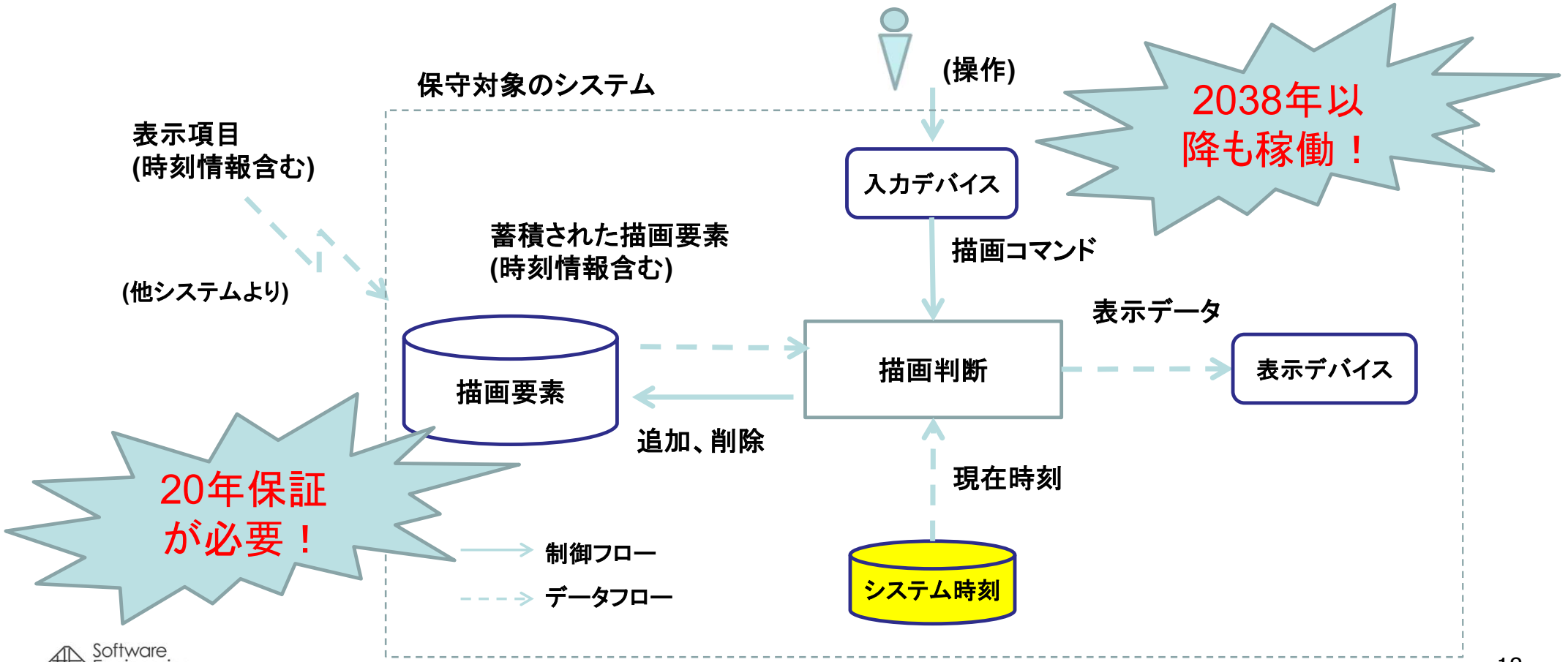
写真はイメージ

時刻を必要とする
各種の項目表示がある。

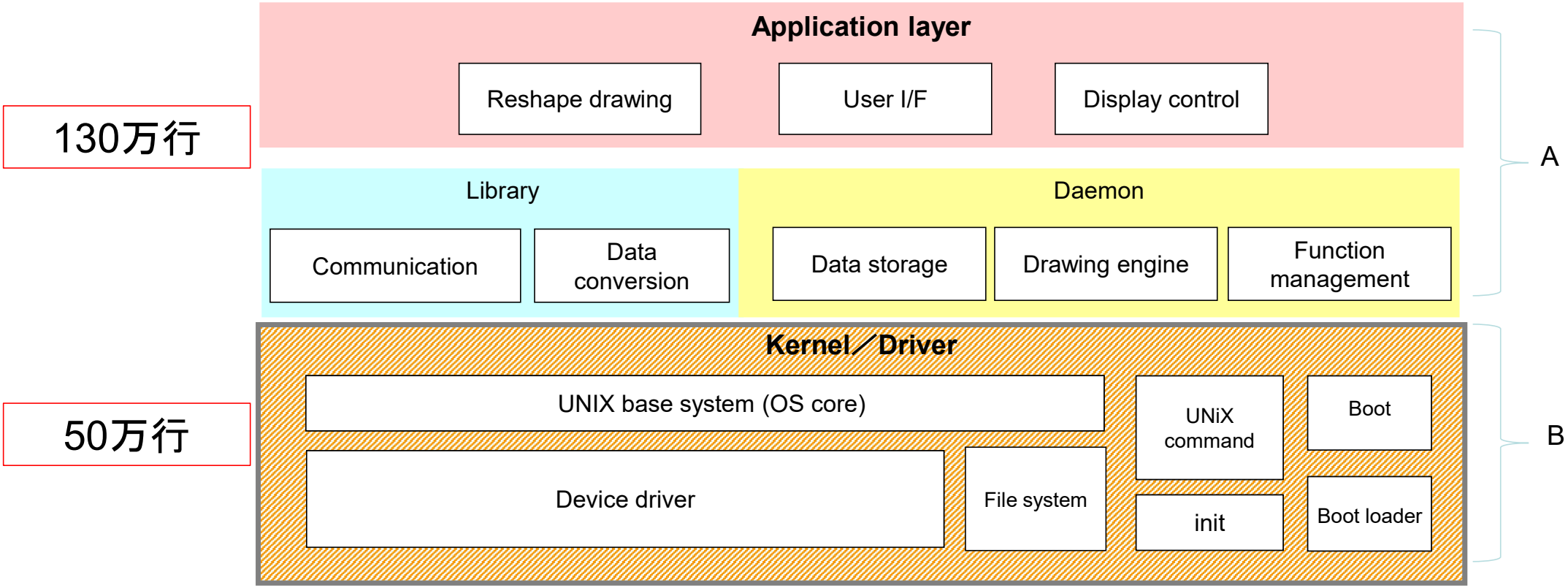
TV番組, 電子番組表, OSD, …

システム時刻が1970年に戻った場合 …

2．2 解決手法
対象システム概要図



2．2 解決手法
ソフトウェアの静的構成図



A： 自社開発部, B : OS部

2．2 解決手法
プログラム変更規模

解決方法の検討に先立ち、おおよその変更規模を調査

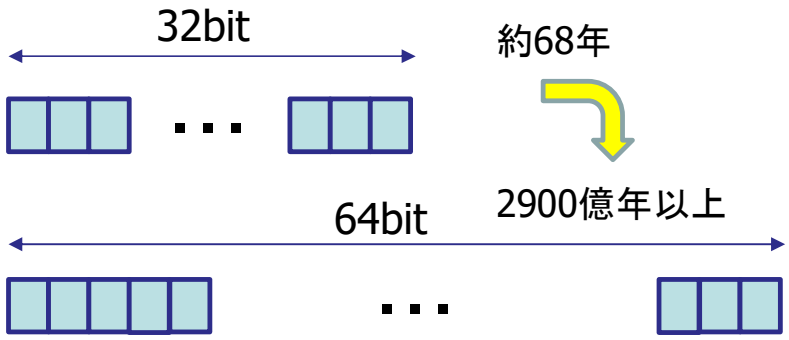
調査の必要なデータの使用箇所	変更箇所数
OS部(デバイスドライバ、OSS、標準ライブラリ含む)	2546
自社開発部	945

OS部の改変は避けたい。なぜなら…

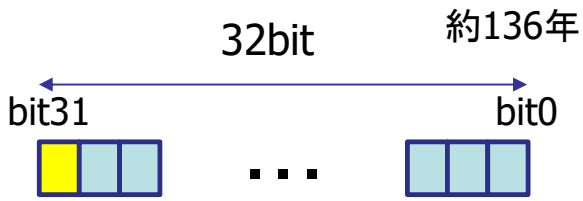
開発期間に対して変更規模が大き過ぎる
修正にコストが掛かり過ぎる

2．2 解決手法
解決方法のアイデア

(a) time_t型の64bit化、もしくは64bit OSの利用

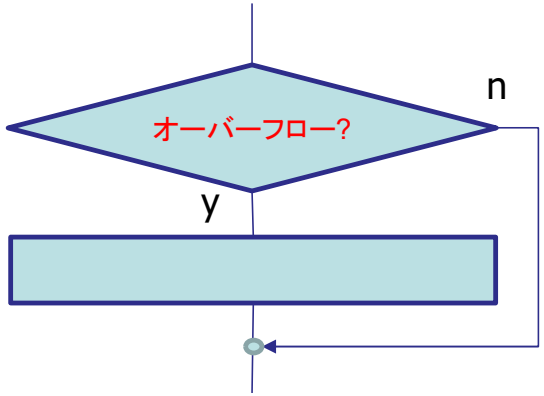


(b) time_t型をunsigned int型に改造

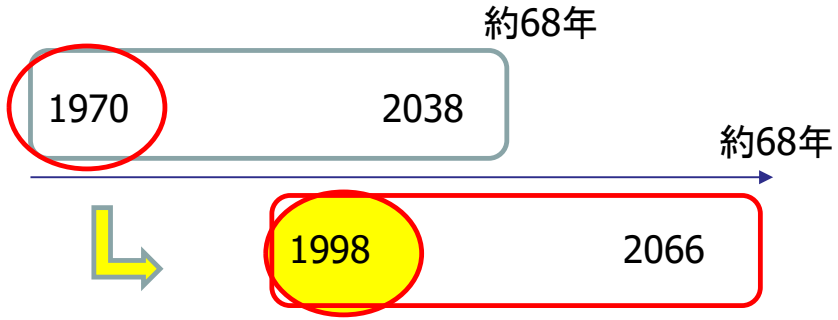


サインbitも時刻情報として利用する

(c) オーバーフロー発生時にプログラムで制御



(d) "epoch" を変更する



2．2 解決手法
設計案の比較

対策案	開発量見込み	OS部修正有無	修正による影響範囲見込み
(a) time_t 型を 64bit にする	大	有	大
(b) time_t 型を unsigned int (32-bit)にする	大	有	大
(c) time_t 型を変更せず,年月日変換・大小比較する箇所で桁上がりを考慮する	大	有	大
(d) time_t 型を変更せず，ラッパー関数を用いて起点（epoch）を変更してシステム時刻を管理する	中	無	中

2．2 解決手法
設計に際しての検討事項

- 1970年(epoch)から何年分遅らせるか
- どのようにしてepochを変更するか
- どのような種類のデータを確認するか
- どのような種類のライブラリ関数を確認するか

これらを解決すれば、プログラムの設計が可能

2．2 解決手法
1970年（epoch）から何年分遅らせるか

日付時刻（UTC）

	1970年1月1日 0:00:00	1998年1月1日 0:00:00	2038年1月19日 3:14:07	2066年1月19日 3:14:07
(time_t 型変数値)				
1970年起点	0x0000 0000	0x34AA DC80	0x7FFF FFFF	0xB4AA DC7F
1998年起点	—	0x0000 0000	0x4B55 237F	0x7FFF FFFF

28年ずらす理由

20年保証が必要 (20年以上ずらす必要あり)
閏年を考慮すると4の倍数が望ましい
2099年まで、曜日計算の変更が不要

2．2　解決手法

1970年（epoch）から何年分遅らせるか

28年　28年

2021年6月

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

2049年6月

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

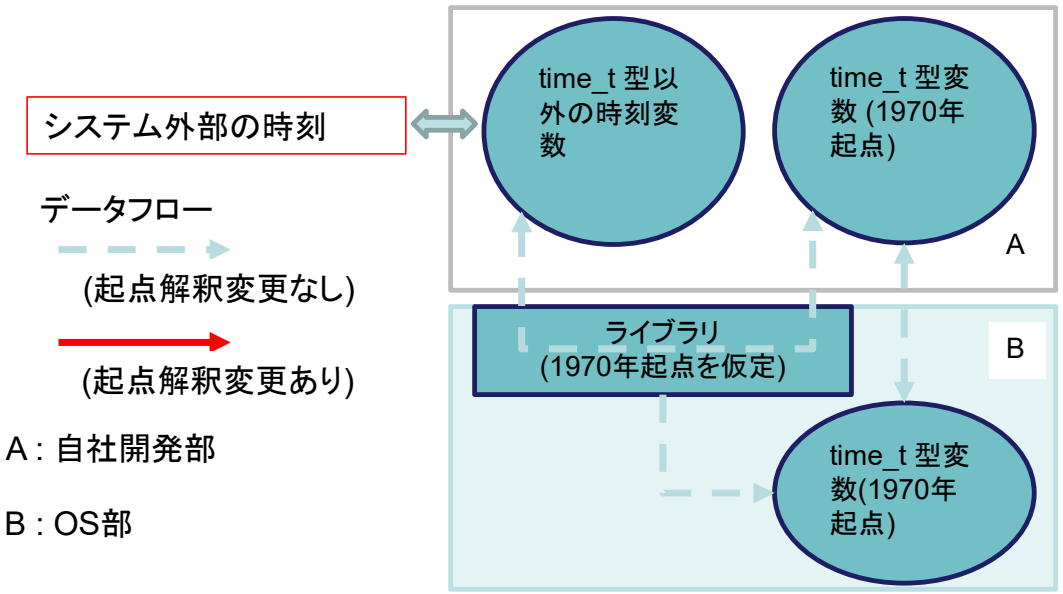
2077年6月

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

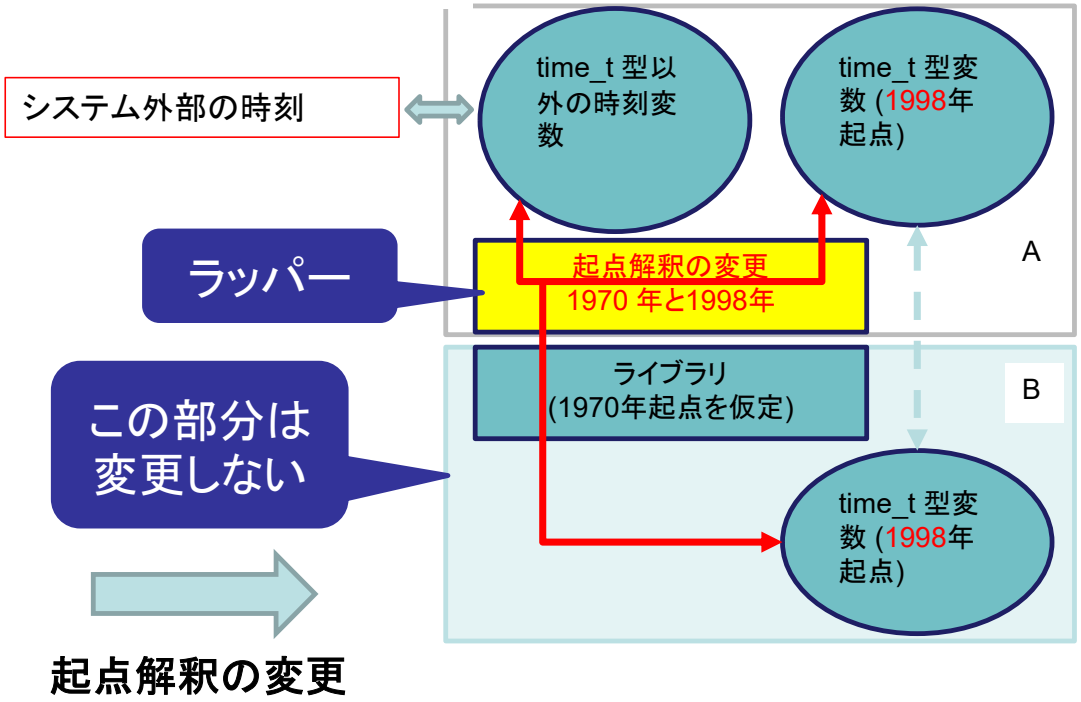
2099年までであれば、28年毎に曜日の並びが同じになる。
（2021年現在を表現できないため、56年はずらせない：1970+56=2026）

2．2 解決手法
どのようにしてepochを変更するか

起点解釈変更前



起点解釈変更後



2．2 解決手法
どのような種類のデータを確認するか

番号	データ型	概要
1	time_t	システム時刻を保存するためのデータ型．標準Cライブラリで定義され，32bit FreeBSDでは，32bit符号付き整数型で定義されている．
2	struct timeval	time_t型変数(tv_sec)とsuseconds_t型変数(tv_usec)をメンバに持つ構造体．tv_secでは秒を，tv_usecではマイクロ秒の情報を格納する．
3	struct tm	年，月，日，曜日などの要素別時刻情報をメンバとし，各メンバをint型で格納する構造体．

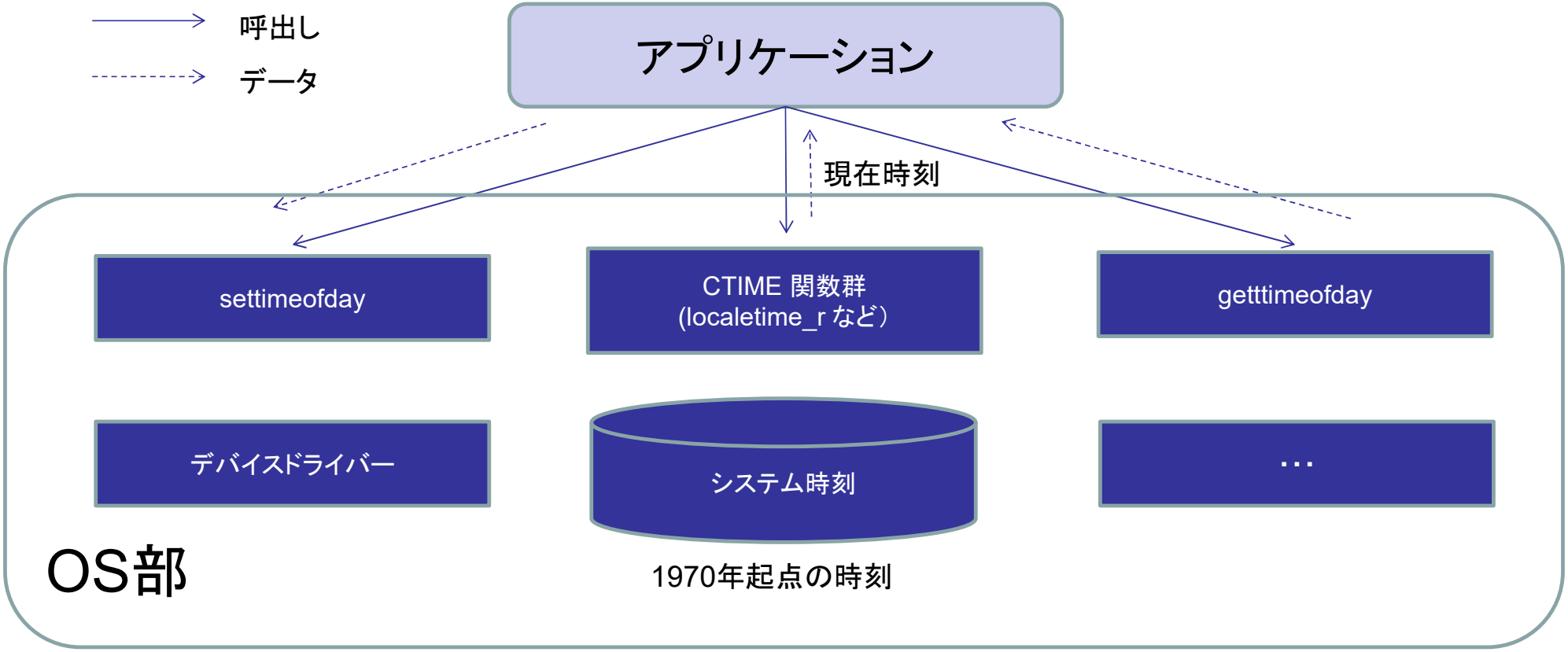
2．2 解決手法

どのような種類のライブラリ関数を確認するか

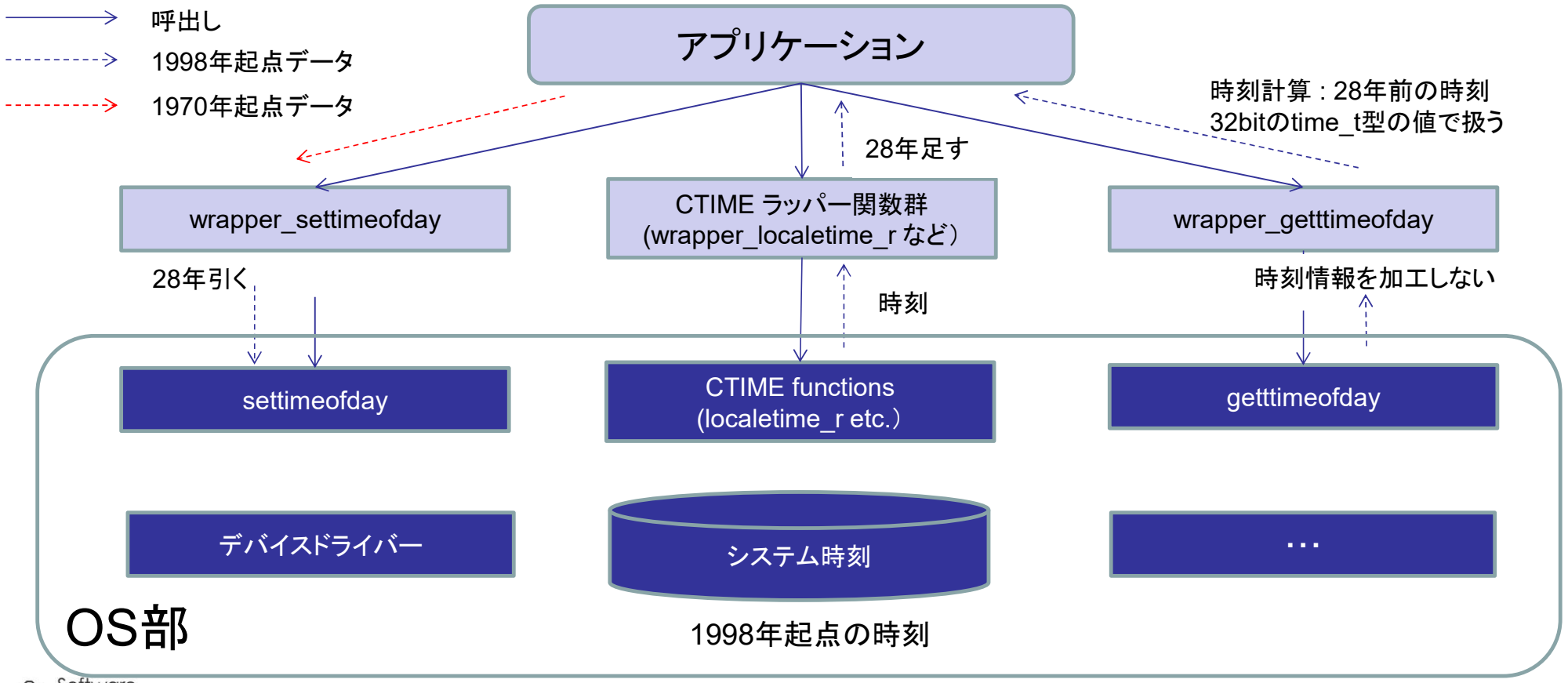
番号	関数名	ラッパー関数の作成
1	clock_t clock(void)	—
2	char *ctime(const time_t *)	必要
3	double difftime(time_t, time_t)	—
4	int gettimeofday(struct timeval *, struct timezone *)	必要
5	struct tm *gmtime(const time_t *)	必要
6	struct tm *localtime(const time_t *, struct tm *)	必要
7	time_t mktime(struct tm *)	必要
8	int settimeofday(const struct timeval *, const struct timezone *)	必要
9	time_t timegm(struct tm *)	必要
・	・	・
・	・	・
60	void tzsetwall(void)	—

2．2 解決手法

修正前の時刻情報のデータと制御フロー



2．2　解決手法
修正後の時刻情報のデータと制御フロー



2.3 実装

2．3　実装
マクロ・型定義

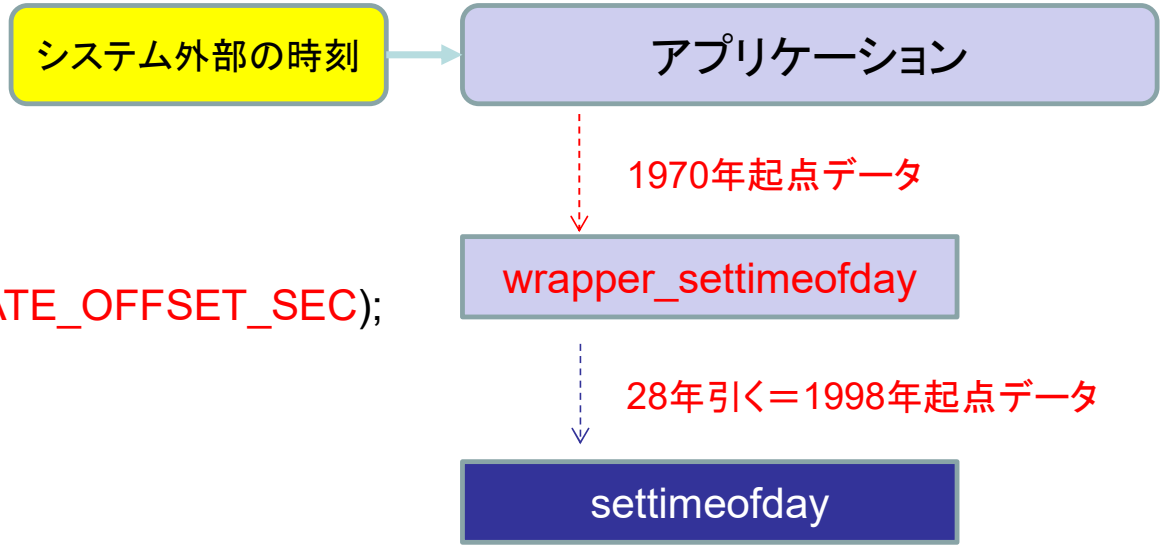
```
#define DATE_OFFSET_YEAR     28         /* 28years */
#define LEAP_DAYS    (DATE_OFFSET_YEAR >> 2)
#define DATE_OFFSET_SEC      ((DATE_OFFSET_YEAR * 365 + LEAP_DAYS) * 60 * 60 * 24)
・・・
```

DATE_OFFSET_YEAR	：	epochのシフト量（28年）
LEAP_DAYS	：	28年中の閏日の日数
DATE_OFFSET_SEC	：	28年を秒で表現

```
typedef struct wrapper_timeval_t {    unsigned int tv_sec;     /* sec base time 1970/1/1 00:00:00 */
                                      suseconds_t tv_usec;     /* usec */}
wrapper_timeval;
```

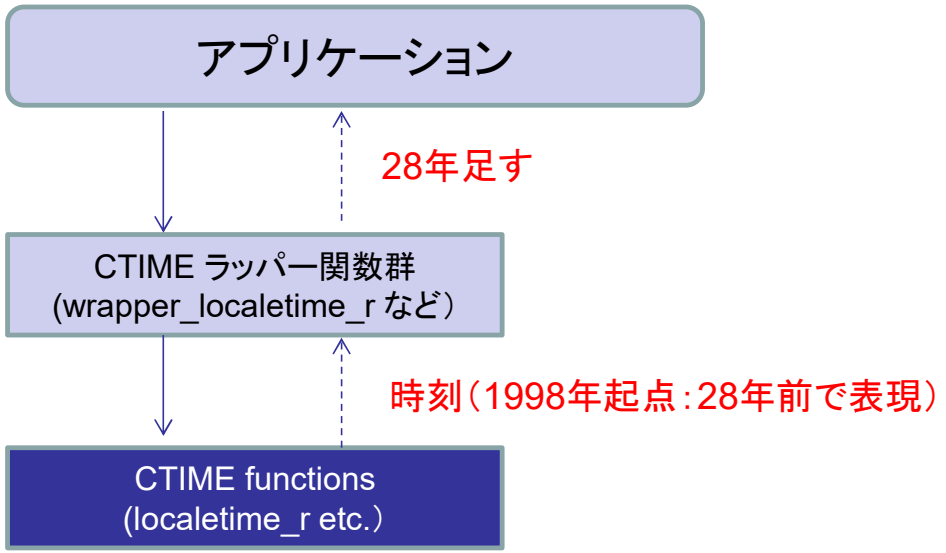
2．3　実装
ラッパー関数（1）

```
int wrapper_settimeofday(const wrapper_timeval * tv , const struct timezone * tz)
{
  struct timeval tv_tmp, *tvp;
  int ret;
  if (NULL == tv)
  {
    tvp = NULL;
  }
  else {
    tv_tmp.tv_sec = (time_t)(tv->tv_sec – DATE_OFFSET_SEC);
    tv_tmp.tv_usec = tv->tv_usec;
    tvp = &tv_tmp;
  }
  ret = settimeofday(tvp, tz);
  return ret;
}
```



2．3　実装
ラッパー関数（2）

```
struct tm* wrapper_localtime_r (const time_t *clock, struct tm *result)
{
  struct tm *tm_tmp;
  if ((tm_tmp = localtime_r(clock, result)) != NULL)
  {
    result->tm_year += DATE_OFFSET_YEAR;
  }
  return tm_tmp;
}
```



2．4　結果

2．4　結果
テスト項目数とテスト結果

	テスト項目数	検出不具合数
単体テスト	165	0
結合テスト	103	1※

※異常値（-1）を時刻の計算に用いた。

2．4　結果
修正規模

	ライブラリ名	ラッパー関数規模（行）	呼出し箇所数	呼び出し部の修正規模（行）
1	ctime	11	0	0
2	ctime_r	11	0	0
3	gettimeofday	6	25	1262
4	gmtime	9	0	0
5	gmtime_r	9	1	21
6	localtime	9	2	12
7	localtime_r	9	30	1131
8	mktime	22	10	307
9	settimeofday	15	2	127
10	strftime	81	2	48
11	timegm	22	1	28
12	timelocal	22	0	0
13	Common functions etc.*	38	-	-
合計		264	73	2936
修正規模（合計）				**264 + 2936 = 3200**

* 複数のラッパー関数で共用する関数と，“include”，“define” 文を含む．

2. 4 結果
修正工数

$$修正工数 = \frac{修正規模}{開発効率}$$

開発効率 = 3.45 steps / 人時

2038年問題での修正規模 = 3200 steps

2038年問題での工数 = 3200 / 3.45 = 927.54 人時

8 時間 / 日, 20 日 / 月

927.54 / (8 * 20) = 5.80 人月

2038年問題だけの工数採取は困難
修正規模と開発効率を用いて近似

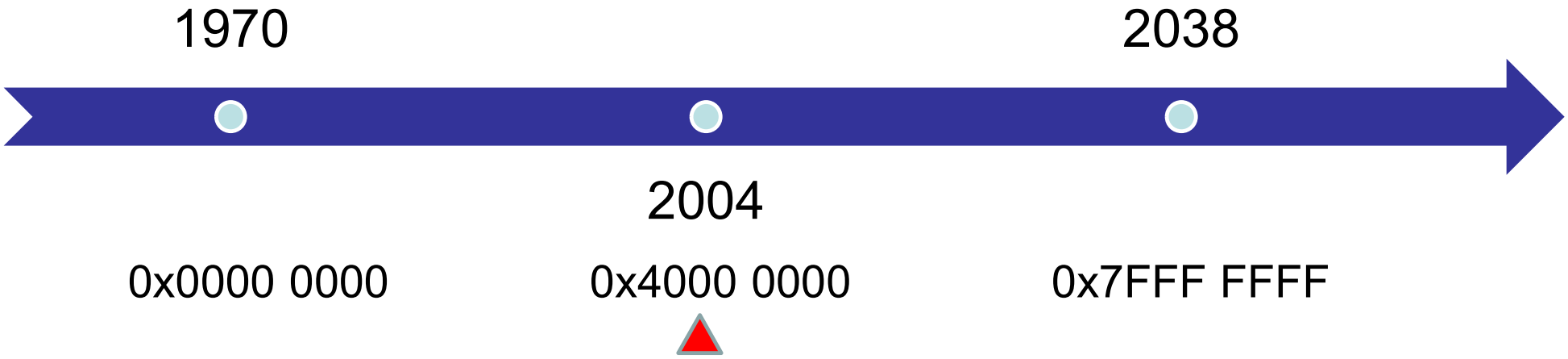
Software Engineering Laboratory

2.5 議論

2．5 議論
ソフトウェアでの日時の取扱い

OS	bit幅	64bit化の状況
Windows	64bit	Visual C++ 2005より前のバージョンのVisual C++とMicrosoft C/C++では，time_t は32bit.
NetBSD	64bit	NetBSD は対応するすべてのアーキテクチャで time_t 型は64bitに変更された.
MacOS/iOS	64bit	時間を2001年1月1日UTCからの経過秒数で表現．現在は64bitアプリケーションしか許容しないよう規約で定めている.
FreeBSD	64bit/32bit	引き続き32bit OSも利用し続けることができるが，2038年問題が生じる.
Raspbian/ Raspberry Pi OS（β版）	32bit/64bit	2020年2月リリースの Raspbian Buster は，まだ32bitである. 2020年5月リリースの raspios（Pi4）より64bit が選択できるようになった.
Linux	64bit/32bit	Linuxカーネル5.6では，time_t型変数の64bit化が行われている．32bitシステム下で作成したファイルの属性等を含めて考えた場合，2038年問題は解決しない.

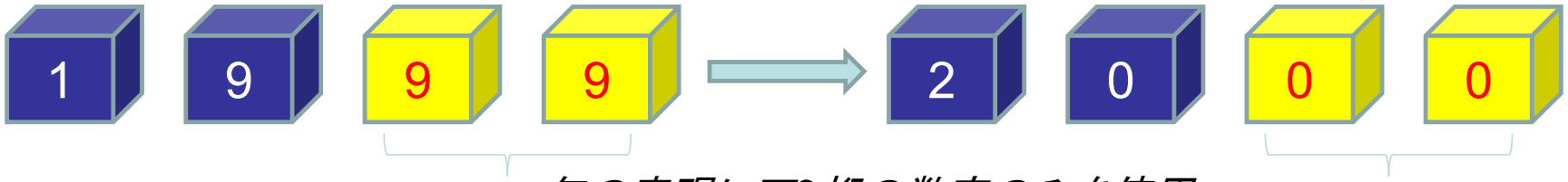
2．５ 議論
UNIXでの2004年問題



オーバーフロー発生原因

- 桁あふれを考慮せずに日付同士を足し合わせた
- 0.5秒単位で時刻を認識するシステムで，最大桁数を増やさなかった

2．5　議論
2000年問題



年の表現に下2桁の数字のみを使用

(a) 日付計算の間違い

Ex) 1996年から2000年までの期間：00 – 96 = -96

(b) 日付比較の間違い

Ex) 96 > 00 より，1996年の方が新しいと誤判断

(c) 入出力による誤り

Ex) 2000年以降のデータを1901年より古いデータとして登録（99 > 00）

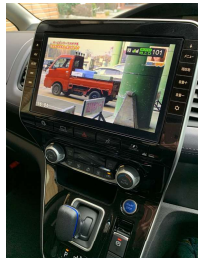
事前に問題点やリスクが周知，対策されたため大きな問題にはならなかった．

２．６　2038年問題対応事例のまとめ

2．6　2038年問題対応事例のまとめ
修正作業の評価

プロジェクト	2038年問題の修正は計画通りに完了
	特に困難な作業などは無かった
手法の特徴	ラッパー関数を使ったepoch変更 (1970年から1998年)による簡単な手法
	修正後のシステムでは1998年以前を扱うことができない点には注意が必要

2．6　2038年問題対応事例のまとめ
まとめ



取組み評価

32bit FreeBSDシステムにおいて，
epochを1970年から1998年にシフトすることで

- （2018年から）20年保証の製品を開発
  - 閏年についても考慮
  - 曜日の計算を変更しない

修正コスト：130万行のアプリケーションで 5.80人月

2038年問題をOS部を変更せずに完遂した

今後

現在、多くの32bitシステムが市場で稼働
例えば、“Raspbian OS” は、
IoT システムの構築等に安価で人気

2038年問題を内包したIoTシステムは多い
気付かずに使い続けられる恐れあり

この問題の解決は今後、より重要になる

効率よく、2038年問題を解決する手法を研究したい

Software Engineering Laboratory

3．32bit 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

設計情報が利用できない場合の保守

３．１　プログラム修正の問題と課題

2038年問題をOS部を変更せずに解決するため、以下の調査が必要

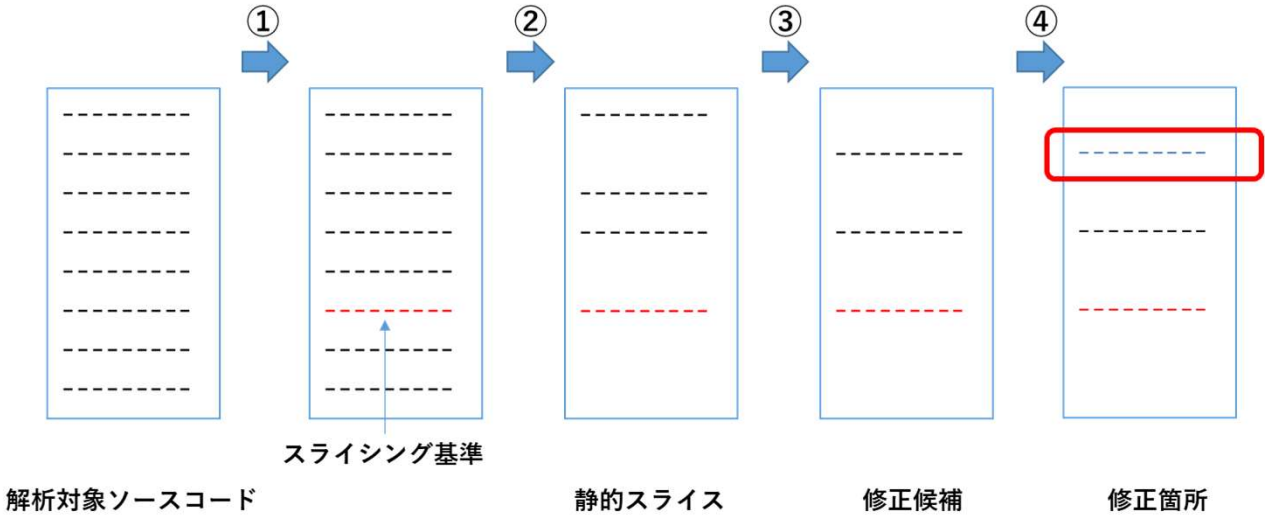
- 日付時刻に関わるライブラリ関数
- time_t型変数を使用する箇所
  - ライブラリ関数の引数
  - 一時的に時刻情報を保存する変数
  - キャストしたものを別の型の変数に代入した箇所

漏れなく抽出し、修正対象箇所かどうかを人手で確認が必要

修正箇所をいかに正確に効率よく見つけるかが課題

3．2　提案手法

プログラムスライシングを用い、修正箇所の特定までを手順化



ステップ① ：スライシング基準選定
ステップ② ：静的スライス抽出
ステップ③ ：修正候補絞り込み
ステップ④ ：修正箇所特定

3．3　修正ステップについて

ステップ①
- **スライス基準選定**
  - time_t型変数が該当プログラムの実行系列で最後に利用される箇所を見つけて，スライシング基準とする

コードレビュー等

ステップ②
- **静的スライスの取得**
  - スライシング基準より，データ依存行，制御依存行を求めて，静的スライスを得る

CodeSonar等

ステップ③
- **絞り込み**
  - 2038年問題に関係しないスライス文を対象から除外

呼出し関数の機能
アプリケーションの機能
スライス文の処理内容

ステップ④
- **特定と修正**
  - 修正候補の各文に対して，修正箇所を特定して修正

標準ライブラリ関数：ラッパー経由
時刻評価の箇所：起点変更分を考慮

3．4　修正コマンド実行例

FreeBSDのtouchコマンド、statコマンド、dateコマンドを修正

端末

ファイル(F)　編集(E)　表示(V)　検索(S)　端末(T)　ヘルプ(H)

```
ooe@FreeBSD:~/test % touch -t 204001010000 aaa
touch: out of range or illegal time specification: [[CC]YY]MMDDhhmm[.SS]
```

修正前のtouchコマンドで2040年を指定

```
ooe@FreeBSD:~/test % touch.new -t 204001010000 aaa
ooe@FreeBSD:~/test % stat.new -F aaa
-rw-r--r-- 1 ooe ooe 0 Jan  1 00:00:00 2040 aaa
ooe@FreeBSD:~/test %
```

修正後のtouchコマンド、statコマンドの実行例

端末

ファイル(F)　編集(E)　表示(V)　検索(S)　端末(T)　ヘルプ(H)

```
ooe@FreeBSD:~ % date
1992年  7月19日  日曜日  22時00分08秒  JST
ooe@FreeBSD:~ % date.new
2020年  7月19日  日曜日  22時00分11秒  JST
ooe@FreeBSD:~ % date -j 204001010000
date: nonexistent time
ooe@FreeBSD:~ % date.new -j 204001010000
2040年  1月  1日  日曜日  00時00分00秒  JST
ooe@FreeBSD:~ %
```

dateコマンド実行例

３．５　提案手法の成果

32bit FreeBSDの標準コマンドを改造し、2038年問題に対応

コマンド	元の行数*	静的スライス結果	修正候補の確認行数	修正行数	追加行数
touch	303	273 (10%)	178 (41%)	18	15
date	871	564 (35%)	258 (70%)	11	35
stat	771	565 (27%)	249 (68%)	3	10

修正候補の確認行数（調査規模）の大幅な低減に成功
2038年問題の修正コストの低減に期待

３．６　まとめ

- 2038年問題の対策コストを下げる方法を提案した
  - プログラムスライシングを応用
  - 修正箇所を特定する手順を示した
- FreeBSDの3つのコマンドで2038年問題に対応した
  - touch, date, stat
- 3つのコマンド全てで調査規模を小さくできた
  - 修正手順の確立と調査規模削減により、対策コスト低減に期待

Software Engineering Laboratory

4. 予約受付端末の他サービス転用に向けたソフトウェア改造事例

設計情報が利用できる場合の保守

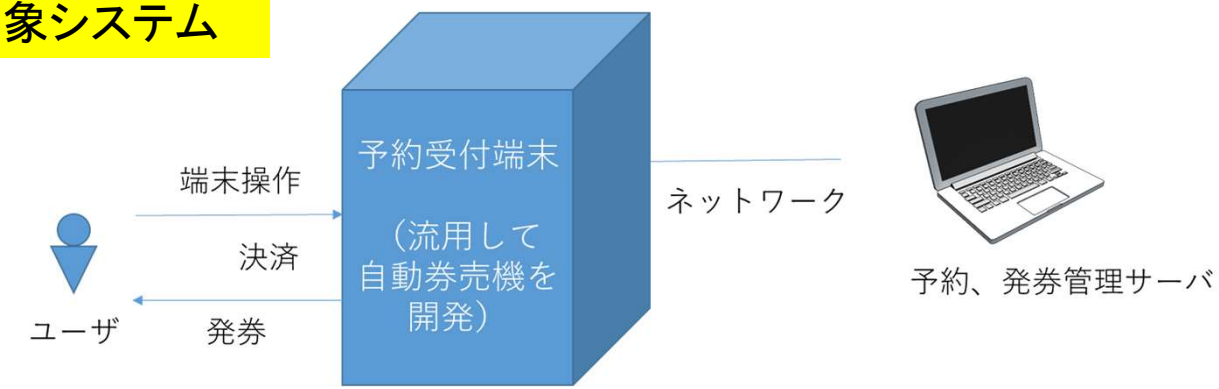
4．1　プログラム修正の問題・課題と対象システム

プログラム修正の問題・課題

修正箇所をいかに正確に効率よく見つけるかが課題

漏れなく抽出し、修正による影響範囲（テスト範囲）をどのように定めるか

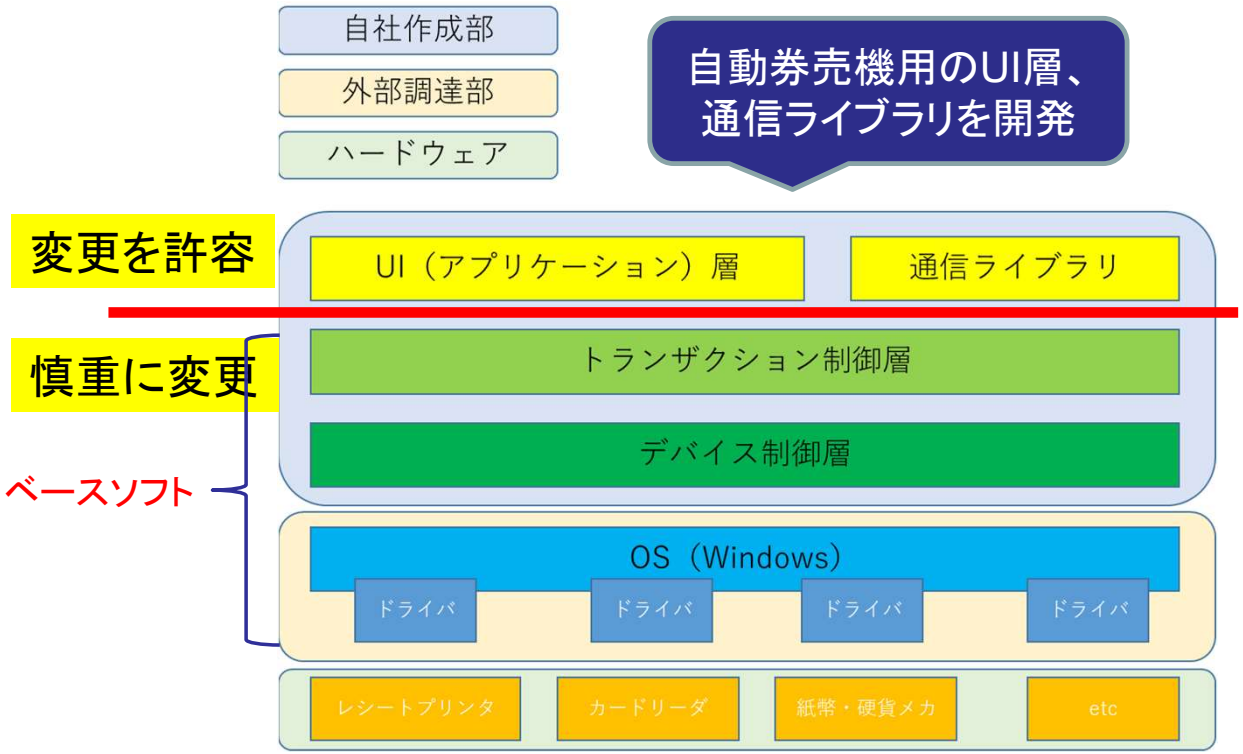
対象システム



適用対象となるサービスを限定せず、様々な用途に展開する構想

4．2　提案手法

対象システムの静的構成図



設計構造により

- 変更を許容する箇所
- 変更を慎重に判断する箇所

を定め、保守方針とする

保守対象箇所を早期に明確化できる
保守作業時に迷いが少ない

ただし、細かな調整は必要

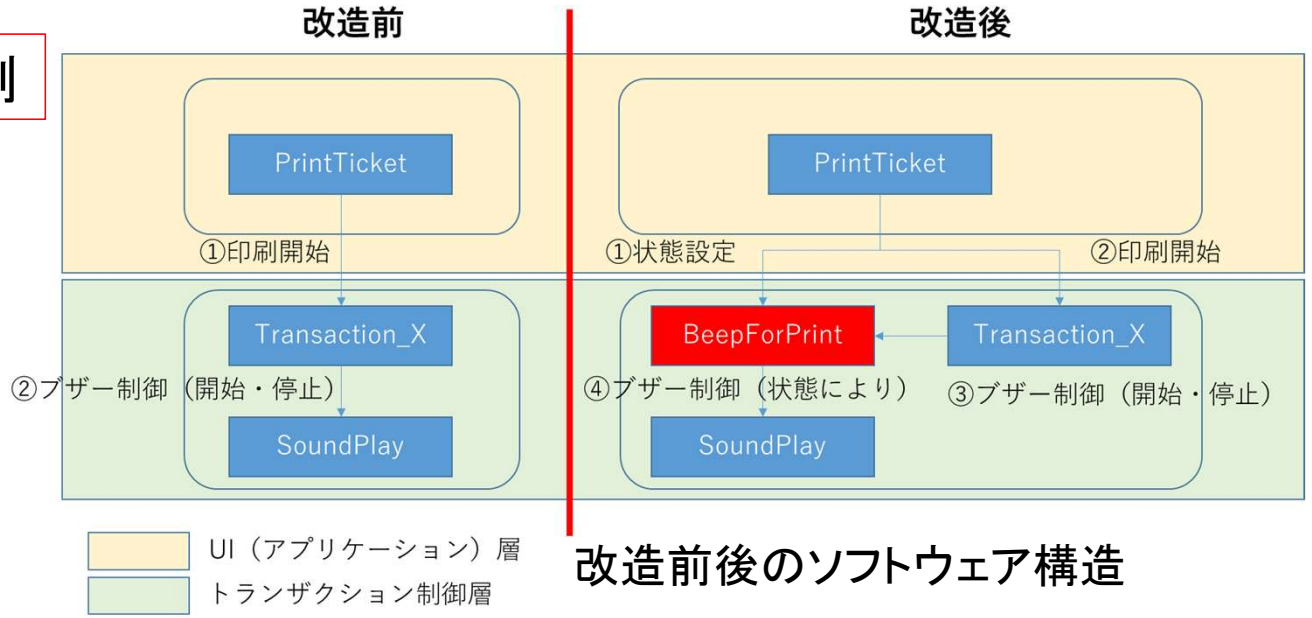
4．3　修正事例

システム開発時に解決した問題例

複数枚のチケット印刷時、
トランザクション制御層に

- まとめて依頼（1）
- 1枚ずつ依頼（2）

という2通りの考え方
既存サービスでは（1）を前提の設計
新サービス（2）でブザー制御に問題発生



改造前後のソフトウェア構造

UI層は積極的に変更
トランザクション制御層は新規モジュール追加
Transaction_Xの呼び出し手順を変更：レビューによる慎重な判断

4．4　設計情報が利用できる場合の修正手順

修正方針決定
- ソフトウェア改変の動機を検討（全体か特定のターゲットか）
- 変更を慎重に行う箇所の修正を含む可能性があるか

変更箇所特定
- 動機と設計構造から変更箇所を特定
- 新規モジュール追加の場合、変更許容度合いにより配置を決定

修正実施
- 元の設計思想を壊さない
- 変更内容につき関係者とのレビュー実施

４．５　まとめ

- トランザクション制御層を互換性を保ち変更した
  - UI層によるブザー鳴動制御の選択を可能とした
- 変更がベースソフトに組み込まれた
  - 現在、2つの異なるサービスで利用されて市場で稼働中
- 変更箇所と変更による影響範囲を容易に特定できた

Software Engineering Laboratory

5. おわりに

5. おわりに
まとめ

改良保守の以下の問題を解決

- 組込み機器開発における2038年問題への対応事例
  - 文字列一致検索等により修正箇所特定
  - ラッパー関数作成によりOS部を変更せずに対応、対応工数等を実証的に示した
- **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**
  - プログラムスライシングの応用等、技法を用いることで修正箇所の特定を効率化した
- 予約受付端末の完全化保守開発事例
  - 設計情報が利用できる場合の修正箇所の特定例と設計上の工夫を示した

DXの実現にも関わる古くて新しい問題

保守の設計開発技術の一般化に向けて研究を進めて行きたい