プラットフォーム依存種検索によるソースコードからのプラットフォーム依存部抽出手法




岡本　周之[†1 †2]　藤原　貴之[†1]　楠本　真二[†2]　岡野　浩三[†2]

組込みシステム開発では，プラットフォーム（PF）変更に伴うソフトウェア移植作業の効率向上が求められる．本稿では，PF 間移植工数の削減を目的とし，PF 依存種検索によるソースコードからの PF 依存部抽出手法を提案する．本手法に従って PF 依存部抽出ツールを開発し，製品ソースコードの移植に適用した結果，PF 依存部の検索・判定・修正工数を 49%、同判定工数を 40% 削減でき，本手法の有効性を検証できた．

Abstracting Method of Platform Dependent Code with Analyzing Platform Dependency in Source Code

Chikashi Okamoto[†1†2], Takayuki Fujiwara[†1], Shinji Kusumoto[†2], Kozo Okano[†2]

The embedded system development requires efficiency improvement in software migration with platform (PF) change. We propose an abstracting method of PF dependent code (PFD) with analyzing PF dependency in source code for reducing the migration cost. We have developed a PFD finding tool using the method and have migrated production source code with the tool. It can reduce PFD finding, judging and modifying cost, and PFD judging cost by 49% and 40%, respectively.

1. はじめに

組込みシステムでは，部品コスト削減，供給停止に伴う代替，性能向上のために CPU などのハードウェアを変更する場合がある．また，機能追加や海外展開の容易化，新技術への追従性向上，ライセンス料削減，開発効率向上のために OS を変更する場合がある．つまり，CPU や OS 等のプラットフォーム (PF) の変更を行う場合がある．

一方，組込みシステムにおけるソフトウェア開発では，既存ソフトウェアを活用する拡張（差分 / 派生）開発が多い．プロジェクト件数比率を図 1.1 に示す [METI2011].

また，組込みシステムでは機能と開発費の多くをソフトウェアが占めており，工数（コスト）削減のためソフ

【脚注】
† 1　（株）日立製作所 横浜研究所
Yokohama Research Laboratory, Hitachi, Ltd.
† 2　大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
a)　本稿で述べられたシステムおよび製品名は，一般に各社の商標または登録商標である．

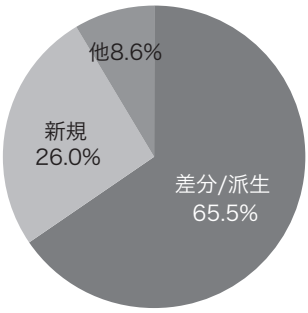


図 1.1　プロジェクト件数比率

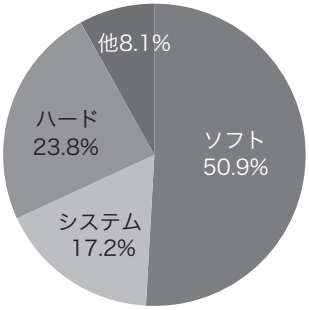


図 1.2　開発費内訳

ト開発効率向上が必要である．開発費内訳を図 1.2 に示す [METI2011].

さらに，組込みシステムでは，厳しいコスト制約のため限られたハードウェアリソース（CPU の演算処理能力，メモリ容量など）で動作するようソフトウェアを開発しなければならない．

このため，開発済みの組込みシステムにおいて，現行 PF と同程度のハードウェアリソースを持つ新 PF へ変更する際のソフトウェア開発工数削減が課題となる．

本稿では，この課題の解決に必要な，PF 間移植における既存ソフトウェアからの PF 依存部抽出について，「PF 依存種検索によるソースコードからの PF 依存部抽出手法」を提案する．本手法では，あらかじめ PF 依存種（とその検索方法）の一覧を作成しておき，これを用いて既存ソフトウェアのソースコードを解析して PF 依存部候補を検索した後，候補から PF 依存部を抽出する．

ソフトウェア開発工数削減に対する既存の手法としては，ソフトウェアプロダクトライン（SPL）開発 [Clements2002] [Pohl2005]，モデル駆動型開発（MDD）[Selic2003] などがある．本稿での提案手法は，これらの既存手法と比較して，PF 変更時の既存ソフトウェア移植に最適化している．また，組み合わせて用いることも可能である．

本提案手法に従って PF 依存部抽出ツールを開発し，実際の製品ソースコードの移植に適用した．この結果，PF 依存部の検索・判定・修正工数を 49%、同判定工数を 40% 削減できるとの見通しを得，本提案手法の有効性を検証できた．

2.　課題とアプローチ

2 章では，PF 間移植手順について述べた後，手順の 1 つである PF 依存部抽出について，従来手法とその問題点，ならびに本稿で提案する手法とその効果を述べる．

2.1. PF 間移植手順

PF の変更方法には次の (1) ～ (3) がある（表 2.1 参照）．

(1) 既存ソフトウェアを新 PF に合うように新規に作成し直す場合，新規作成工数（大）が必用である．

表 2.1　PF の変更方法

PF 変更方法	概要	開発工数	要求リソース
(1) 既存ソフト / 既存 OS / 既存マイコン → 新ソフト / 新 OS / 新マイコン	全ソフトを新 PF 向けに新規作成	×（大）	○ 既存ソフトと同等
(2) 既存ソフト / 既存 OS / 既存マイコン → 既存ソフト / 既存 OS / 仮想マイコン	仮想化技術で既存ソフトをそのまま使用	○（小）	× 仮想化用 CPU/ メモリが必要
(3) 既存ソフト / 既存 OS / 既存マイコン → 既存ソフト 修正 / 新 OS / 新マイコン	既存ソフトを新 PF 向けに一部改修	△（中）	○ 既存ソフトと同等

(2) 仮想化技術を用いて既存ソフトウェアをそのまま使用する場合，仮想化のための CPU/ メモリが必要である．

(3) 既存ソフトウェアを新 PF に合うように一部改修して（PF 間で移植して）使用する場合，改修工数（中）がかかる．

新 PF が，現行 PF と同程度のハードウェアリソースしか持てない場合，(2) は不向きである．ソフトウェア開発工数を考慮すると (1) よりは (3) が現実的である．そこで本稿では，PF 変更方法に (3)，すなわち PF 間移植を用いる場合を対象とする．

PF 間移植において，工数削減のため移植作業を必要最小限にするには，既存ソフトの PF 依存部と PF 非依存部を分離し，PF 依存部のみを修正する必要がある．

ここで，デバイスドライバ層を PF 依存部とするなど，階層構造からある程度 PF 依存部を分離できるが，組込みシステムでは，理想的な階層構造を維持できない場合が多い．例えば，厳しいコスト制約のため限られたリソースの中でソフトウェア処理の高速化が求められることがある．この場合，ソフトウェアの階層構造を理想的に保つことよりも，例外的な呼び出し方を許してでも処理の高速性が優先されてしまうことがある．

また，出荷間際で開発期間が不足する中で，不具合に対するソフトウェア修正が求められることもある．この場合，不具合解消に有効な手段の内，実装時間がかかる理想的な手段よりも，理想的ではないが短時間で実装可能な手段が優先されてしまうことがある．

拡張開発を繰り返した場合は，このような状況が積み重なるため，理想的な構造の維持がさらに困難となる [Fowler1999][Ochiai2002].

この結果，PF 依存部が集約された PF 依存層と，PF 依存部が含まれていない PF 非依存層について，PF 層のすぐ上位に集約されるべき PF 依存部分が，ソフトウェア全体に離散してしまい，一括して把握することが困難になる．

このような状態に対応するための，一般的な PF 間移植手順を図 2.1 に示す．

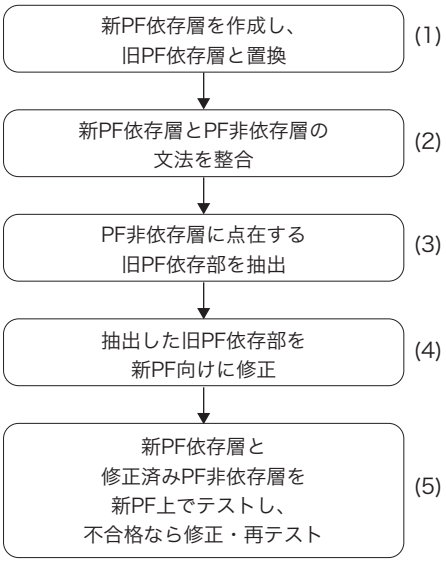


図 2.1　PF 間移植手順

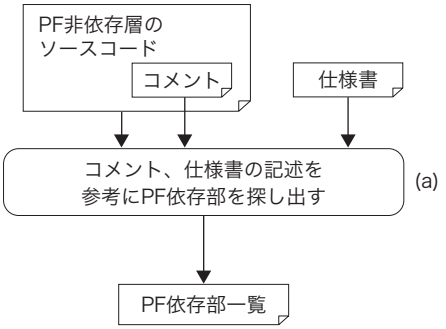


図 2.2　従来の PF 依存部抽出手法

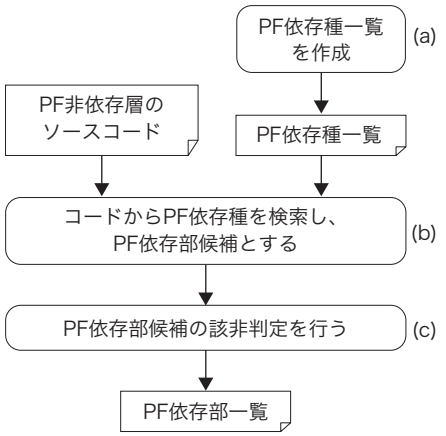


図 2.3　提案する PF 依存部抽出手法

(1) PF 依存層（ドライバ層など）を新 PF（ハード，OS 層など）用に作成し，旧 PF 依存層と置換する．

(2) 新 PF 依存層と，PF 非依存層（ミドル層，アプリ層など）について，文法上の整合を取る．

(3) PF 非依存層に点在する旧 PF 依存部を抽出する．

(4) 抽出した旧 PF 依存部を新 PF 向けに修正する（新規作成関数との置換もある）．

(5) 新 PF 依存層と修正済みの PF 非依存層を新 PF 上でテストし，不合格なら修正して再度テストする．

2.2. 従来の PF 依存部抽出手法の問題

PF 間移植手順 (3) について，社内の過去の移植事例と参考文献を基に，従来の PF 依存部抽出手法の問題点を述べる．

従来の PF 依存部抽出手法を図 2.2 に示す．従来手法では，ハードウェアや OS 等 PF との関連が記された仕様書，コード内の PF 依存に関するコメントなどと，ソースコードを照らし合わせて PF 依存部を探し出し，PF 依存部一覧を作成する．

従来手法では，仕様書にハードウェア依存部である旨が記載されている部分を，PF 依存部と認識していた．また，ソースコードのコメントに同様の記述があることをもって，PF 依存部と認識していた．

しかし，仕様書やソースコードのコメントに，PF 依存部に関する記載がない，または記載があっても誤っていることが多い [Spinellis2006]．例えば，将来の拡張のことを十分に考慮していない場合（OS が変更になることは想定していなかった等）や，開発時間の不足でコードのみ修正し，仕様書やコメントを修正しなかった場合などである．その結果，PF 依存部の抽出が不完全となり，テスト不合格で再度修正するケースが増え，手戻り工数が増える．

また，近年，組込みシステムの多機能化，大規模化，複雑化に伴い，完全に網羅的にテストを実施することは，困難になっており，潜在的な不具合がテストで顕在化せず，製品出荷後に問題を引き起こすことがある [Sugiyama2003]．出荷後の不具合は対策費の増大を招き，経産省による 193 事業部門への調査では，2008 年度に不具合が発生した事業部門の 17% において対策費総額が 1 億円以上に上ったことが判明した [METI2010B]．

2.3. 提案する PF 依存部抽出手法とその効果

上記問題点を解決するため，本稿で提案する「PF 依存種検索によるソースコードからの PF 依存部抽出手法」では，ドキュメントやコメントではなく，ソースコード（実行命令部分）の情報を用いて PF 依存部を判定する．

ソースコードは，更新されていない可能性がある仕様書やコメントと異なり，旧 PF で動いているソフトウェアを作成するのに用いられる最新情報であるため，現状のプログラムと乖離することを避けられる．

本稿で提案する PF 依存部抽出手法を図 2.3 に示す．本提案手法では，(a) あらかじめ PF 依存種（とその検索方法）の一覧を作成しておき，(b)PF 非依存層のソースコードから PF 依存種を検索した結果を PF 依存部候補とする．(c) この PF 依存部候補について，PF 依存部か否かの該非判定を行って PF 依存部を得，一覧を作成する．

本提案手法の特徴は，次の (1) ～ (3) にまとめられる．

(1) PF 依存種が一覧化されているため，過去の事例から判明した PF 依存種を早い段階で漏れなく抽出でき，同じテストを何回も実施しなくて済み，潜在的な不具合がテストで顕在化しないリスクを減らせる．

(2) 複数の PF 依存部をまとめて対応することができるため，工数が削減でき，対応方法を最適化できる．すなわち，リファクタリングが効率的に実施できる．

(3) PF 依存種を定型化することで，自動化ツールによる作業支援が可能になるため，更なる工数削減が可能となる．

3. PF 依存種検索によるソースコードからの PF 依存部抽出手法

3 章では，提案する PF 依存部抽出手法のキーアイディアである PF 依存種とその一覧を説明する．

3.1. PF 依存要因

PF 間移植における，PF 依存要因ごとの既存アプリケーション（アプリ）への影響と，対応方法を図 3.1 に示す．

ライブラリ及び OS への依存部は，ライブラリ API やシステムコールの I/F 変換ラッパを用いることで，アプリケーションの修正を行うことなく対応することができる．ただし，変換ラッパを用いる I/F を抽出する必要がある．

一方，デバイスドライバ（ドライバ）などのソフトウェア，マイコンやマイコン周辺デバイス（デバイス）などのハードウェア，コンパイラなどの開発環境への依存部は，これを抽出して個別に修正する必要がある．

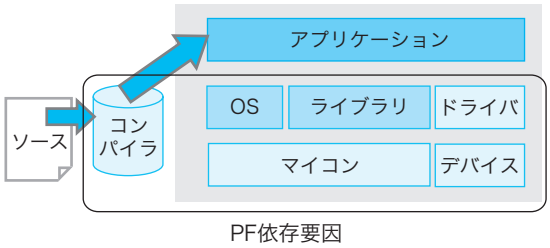
3.2. PF 依存種

本研究では，PF 移行に際して修正が必須となる可能性のある PF 依存部を，PF 依存内容別に 39 種類に分類した．この分類では，社内の過去の移植事例から 23 種を集め，その他 MISRA-C[MSR] にある他の 16 種と合わせた．なお，移植性を高めるためのリファクタリングなど，PF 移行に必ずしも必要ではない修正は含めていない．

本研究で整理した PF 依存種一覧の抜粋を表 3.1 に示す．例えば，旧 PF 用の既存コードを検索した結果「ソケット通信用構造体へのデータ入出力」という PF 依存種に該当するコードが検出された場合，PF の「エンディアン」の種別（ビッグかリトルか），または「パディング有無」に依存している可能性があることを示している．もし旧 PF 用の既存コードがビッグエンディアン依存であり，かつ，新 PF がリトルエンディアンである場合は，PF 移行に際して PF 依存部の修正が必要となる．

なお，PF 依存種の一覧を付録の表 Ex.1 に記す．

また，移植時の修正難易度を「易」「普」「難」の 3 段階で分類し，大まかな修正工数把握を可能とした．修正難易



PF依存要因

PF 依存要因	対応方法	アプリ修正
ライブラリ	I/F 変換ラッパを利用	不要
OS		
ドライバ	PF 依存部を抽出して個別に修正	要
デバイス		
マイコン		
コンパイラ		

図 3.1　PF 依存要因と対応方法

表 3.1　PF 依存種一覧（全 39 種）の抜粋

PF 依存種（検索対象コード）	PF 依存内容	修正難易度	修正重要度
ソケット通信用構造体へのデータ入出力	エンディアン、パディング有無	難	2
負整数の除算、剰余算	小数部の切上げ切捨て	普	1
レジスタへの直接アドレス参照	アドレス値	難	3
ポインタから固定幅整数型へのキャスト	ポインタサイズ	普	2
構造体メンバへの直値オフセット参照	パディング有無	難	1
システムコールの呼出	システムコール名	易	4

表 3.2　修正難易度の基準と該当 PF 依存種数

修正難易度	基準	該当依存種数	
		移植事例	MISRA-C
易	移植前後のコードがほぼ 1:1 で機械的に変換可能	7	5
普	変更方法が 2 通り以上あり、状況に応じた選択が必要。PF 依存部 1 行に対し変換後コードが数行	8	11
難	PF 依存部以外に関連箇所への影響あり。ハードウェアに依存。PF 依存部 1 行に対し数十～数百行のコード変更が必要	8	0

度の基準と該当する PF 依存種数を表 3.2 に示す．更に，移植時の修正重要度を 1 ～ 4 の 4 段階で分類し，大まかな修正順の決定を可能とした．修正重要度の基準と該当する PF 依存種数を表 3.3 に示す．これにより，例えば，修正難易度が高く修正重要度が低い PF 依存部よりも，修正難易度が低く修正重要度が高い PF 依存部を優先して対応すべきであるが，この様な判断を容易化できる．

表 3.3　修正重要度の基準と該当 PF 依存種数

修正重要度	基準	該当依存種数	
		移植事例	MISRA-C
1	コンパイルは通るが、動作に問題がでる可能性がある	9	13
2	コンパイルは通るが、ほぼ確実に動作に問題が生じる（値が不正になるなど）	8	3
3	コンパイルは通るが、動作に致命的な問題が生じる (メモリの範囲外アクセス、無限ループなど)	3	0
4	コンパイルが通らない	3	0

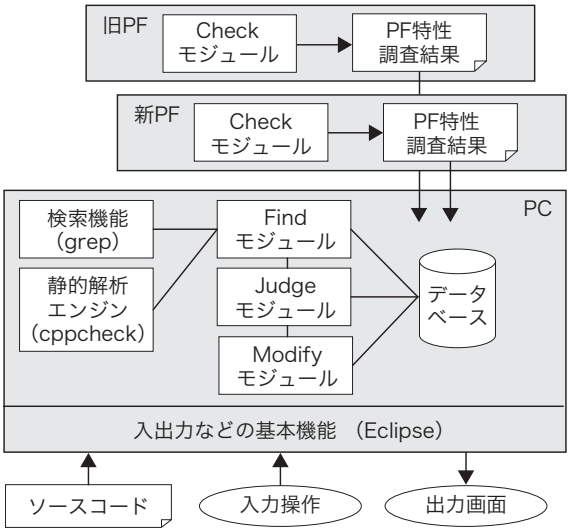


図 4.1　PF 依存部抽出支援ツールの機能構成

4.　PF 依存部抽出支援ツールの開発

本稿で提案する PF 依存部抽出手法の適用を支援するための PF 依存部抽出支援ツールを開発した．4 章では，本ツールについて述べる．

4.1. PF 依存部抽出支援ツールの概要

本ツールの概略機能構成を図 4.1 に示す．本ツールは，新旧それぞれの PF 上で動作する (1)Check モジュールと，PC 上で動作する (2)Find モジュール，(3)Judge モジュール，(4)Modify モジュール，検索機能，静的解析エンジン及びデータベースからなる．

組込み機器開発で取り扱われる言語の 60 ～ 70% は C 言語であるため [METI2010A][IPA2013]，本ツールの解析対象は C 言語ファイルとした．

また，入出力などの基本機能を容易に実装するため，OSS の統合開発環境として広く普及している Eclipse[ECL] をベースに用いた．また，同じく OSS である Cppcheck[CPC] を改良し，ソースコードの静的解析エンジンとして用いた．検索機能としては grep コマンドを用いた．以下，各モジュールについて説明する．

(1) Check モジュール

Check モジュールは，新旧それぞれの PF 上で動作し，表 3.1 の PF 依存内容に相当する PF 特性を調査する．例えば，旧 PF のエンディアンがビッグで，新 PF のエンディアンがリトルであることなどを得る．PF 特性調査結果はファイルとして出力される．

(2) Find モジュール

Find モジュールは，データベースに保存された PF 依存種に該当するコード（PF 依存部）を，入力されたソースコードから検索する．検索結果は，PF 依存部候補一覧としてデータベースに保存される．

なお，Find モジュールは，新旧 PF における PF 特性調査結果を元に，両者で PF 特性が異なる PF 依存種のみを検索対象とする．本ツールでの出力画面例を図 4.2(a) に示す．この画面は，旧 PF のエンディアンがビッグで，新 PF のエンディアンがリトルであるため，エンディアンに関連する PF 依存種が自動選定された場合の例を示している．

(3) Judge モジュール

Judge モジュールは，データベースに保存された PF 依存部候補一覧を利用者に提示する．また，利用者がソースコードを解析し，それぞれの PF 依存部候補が PF 依存部か否かの該否判定を行った後，Judge モジュールはその該非判定結果を受け付ける．本ツールでの出力画面例を図 4.2(b) に示す．この画面は，PF 依存部候補の内 ID13 は PF 依存部である，と利用者が判定しチェックをつけた場合の例を示している．該非判定結果で PF 依存部とされた箇所は，PF 依存部一覧としてデータベースに保存される．

(4) Modify モジュール

Modify モジュールは，データベースに保存された PF 依存部一覧を利用者に提示する．また，利用者が PF 依存部の 1 つを選択した場合，該当するコードと，PF 依存種およびその修正案を提示する．本ツールでの出力画面例を図 4.2(c) に示す．この画面は，PF 依存部一覧の内 ID 4 の PF 依存部を利用者が選択し，該当する「applet_tables.c の 45 行目」とその前後のコード，PF 依存種およびその修正案が表示された場合の例を示している．

なお，図 4.2(a)，(b)，(c) に示した画面例は，各機能の説明用にそれぞれ典型的なものを選んだ．同一解析過程での画面ではないため，表示項目は相互に対応しているとは限らない．

4.2. PF 依存部抽出支援ツールの利用手順

PF 依存部抽出支援ツールの利用手順を以下に示す．

(1) ツールの Check 機能を用いて旧 PF と新 PF の PF 特性を調査する．

(2) ツールの Find 機能を用いて PF 特性が異なる PF 依存種をソースコードから検索し，PF 依存部候補を得る．

(3) ツールの Judge 機能を用いて PF 依存部候補の該非判定を行い，判定結果をツールに入力する．

(4) 各 PF 依存部について，ツールの Modify 機能が出力した PF 依存種別の修正方法を参考に修正する．

図 4.2(a)　PF 依存部抽出支援ツールの出力画面例

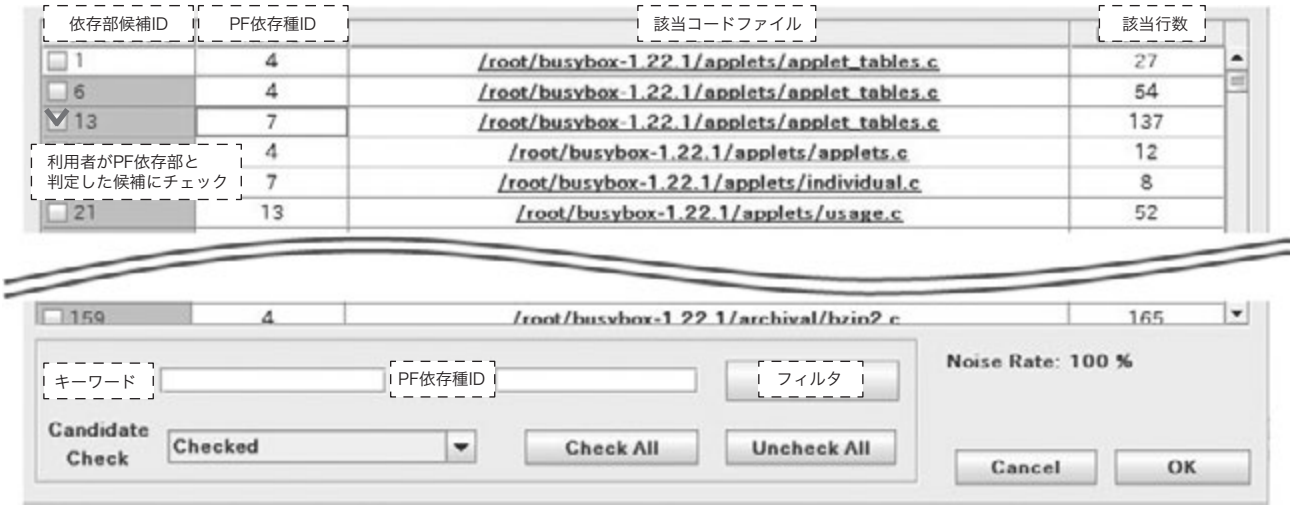


図 4.2(b)　PF 依存部抽出支援ツールの出力画面例

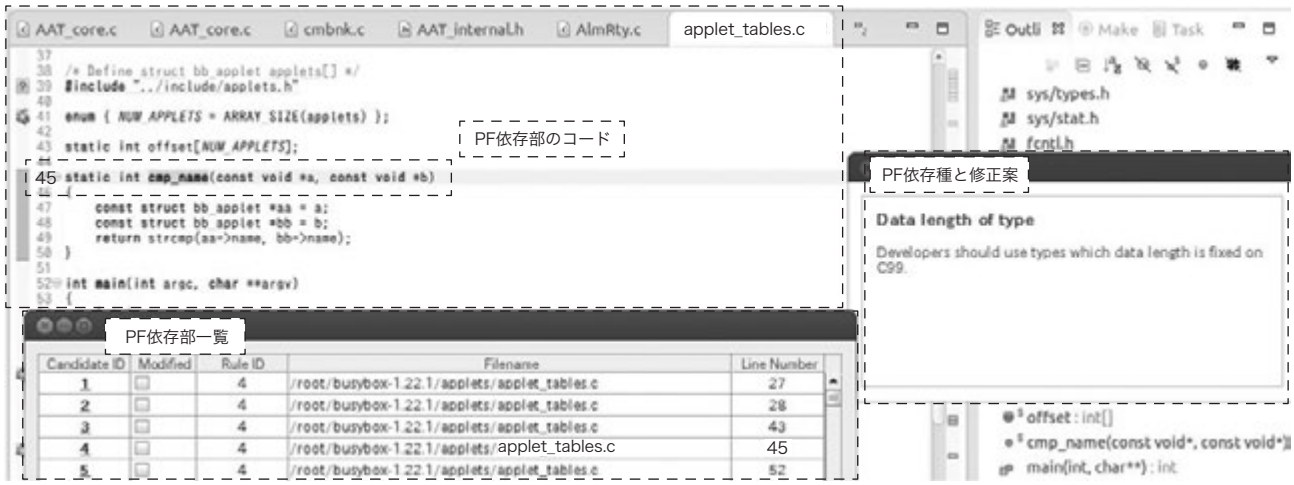


図 4.2(c)　PF 依存部抽出支援ツールの出力画面例

4.3. PF 依存部抽出支援ツールの処理方式

PF 依存部抽出支援ツールの代表的なモジュールである Check モジュールと Find モジュールの処理方式について，以下に示す．

(1) Check モジュール

Check モジュールでは，例えば PF 特性としてエンディアンを調べる場合，エンディアンがビッグかリトルかによって値が変化するようなプログラムのコードを用意しておく．コードを新旧 PF 用にそれぞれコンパイルして，各 PF 上で実行し，その値からエンディアンの種別を判定する．型のサイズを調べる場合なども同様に行う．本 Check モジュールでは，判定処理もプログラム化し，判定結果を PF 特性結果ファイルとして出力させた．

(2) Find モジュール

PF 依存部の抽出を行う Find モジュールの処理方式，すなわちソースコードの検索方式は，抽出する PF 依存種により異なる．代表的な方式を以下に示す．

(i) 単純キーワード検索方式

本方式では cppcheck を使わず，grep により特定キーワードをソースコードから検索する．例えば「#pragma」という文字列による検索が該当した部分を，「プラグマの使用」という PF 依存部の候補として抽出する．

(ii) 数値検索＋値確認方式

本方式では cppcheck を使わず，grep により特定の種類の数値を検索し，その値が特定範囲に含まれているものを抽出する．例えば「0x」という文字列で検索した 16 進数が，レジスタに割り当てられたメモリアドレスの範囲に含まれていた場合，その数を利用する部分を「レジスタへの直接アドレス参照」という PF 依存部の候補として抽出する．

(iii) 関数検索＋名前確認方式

本方式では，cppcheck を使って関数シンボルを検索し，そのシンボルが特定の文字列と一致するものを抽出する．例えば関数シンボルが，システムコール一覧の「semTake」と一致した場合，そのシンボルの利用部分を「システムコールの呼出」という PF 依存部の候補として抽出する．

(iv) 関数検索＋名前確認＋引数確認方式

本方式では，(iii) に加えて，引数の型も条件に加える．例えば，関数シンボルが「send」であり，かつ，第 2 引数の型が「char または unsigned char の，ポインタまたは配列」でない場合，「ソケット通信用構造体へのデータ入出力」という PF 依存部の候補として抽出する．

(v) 代入検索＋キャスト確認＋サイズ確認方式

本方式では，cppcheck の機能を複合的に使う．「変数 = 変数」の形を検索し，該当部分が「unsigned」「signed」間でキャストしており，かつ，右辺のサイズが左辺のサイズより大きい場合，「明示的，暗黙的な型変換」という PF 依存部の候補として抽出する．

5. 製品ソースコードへの適用評価

産業用インフラシステム向け情報機器の製品ソースコードの移植に，提案する PF 依存部抽出手法を適用し評価した．

5.1. 適用対象

本評価における提案手法の適用対象は，産業用インフラシステム向け情報機器の製品ソースコード（コード規模は数百 KB，言語は C）のうち，主要機能（コード規模は製品全体の約 50%）部分とした．本手法を適用した移植では，マイコンのアーキテクチャを SH から ARM に，OS を VxWorks から Linux に，それぞれ変更した．

また，MISRA-C で定義されている PF 依存部は，例えば富士通ソフトウェアテクノロジーズの PGRelief[PGR] やテクマトリックスの C++Test[CPT] など，既存の静的解析ツールで抽出可能なので，これらを除いた 23 種の PF 依存種を評価用の自動化対象とし，4 章で述べた PF 依存部抽出支援ツールとして実装した．

5.2. 適用手順

本評価における PF 間移植では，図 2.1 に示した PF 間移植手順 (3) において PF 依存部抽出支援ツールを用い，提案手法を適用した．

5.3. 評価項目・評価方法・評価結果

本評価の評価項目ごとに，評価方法，評価結果を述べる．

(1) 移植工数

PF 依存部抽出支援ツールを用いた PF 間移植手順を 3 つの移植工程に分類し，本評価における各移植工程の工数を計測した．具体的には，手順 (1)(2) を移植工程 (A)「文法上の整合」，手順 (3)(4) を移植工程 (B)「PF 依存部修正方法の検討と実装」，手順 (5) を移植工程 (C)「動作確認とデバッグ」とした．移植工程 (A) には，ビルド環境構築，OS・ドライバ整備，リンクエラー修正などが含まれる．

移植工数の評価結果を表 5.1 に示す．本評価の移植では，移植工程 (A)「文法上の整合」に 77.5 時間（全工程に対し 24%），移植工程 (B)「PF 依存部修正方法の検討と実装」に 115.0 時間（同 35%），移植工程 (C)「動作確認とデバッグ」に 132.0 時間（同 41%）を費やした．また，移植工程 (B) の内，ツールで抽出可能な PF 依存部に関する作業工数 (B1) は 104.5 時間（同 32%），それ以外の工数 (B2) は 10.5 時間（同 3%）であった．

(2) PF 特性調査機能の有効範囲

移植前後で特定の PF 特性が同じである場合に，その PF 特性に関連する PF 依存種は，PF 間移植時の修正が必須ではない．例えば移植前後の PF がどちらもビッグエンディア

ンの場合，移植対象コードがビッグエンディアン依存であったとしても，その PF 依存部を修正することなく PF 間で移植できる．つまり，PF 依存種に関連する PF 特性をあらかじめ得ておけば，不要な（移植に必須ではない）PF 依存部調査・修正工数の削減につながる．

そこで，PF 特性調査機能，すなわち PF 依存部抽出支援ツールの check 機能を用いて得られる PF 特性について，関連する PF 依存種の数と割合を求めた．

「PF 特性調査機能の有効範囲」の評価結果を表 5.2 に示す．全 PF 依存種 39 項目中 19 項目 (49%) について，関連する PF 特性を得られた．この 19 項目の PF 依存種全てが，社内の過去の移植事例から得た 23 種に含まれており，MISRA-C から得た 16 種のものはなかった．

(3) PF 依存部候補数

PF 間移植手順 (3)(b) において，PF 依存部候補の抽出件数が少なければ，その分，手順 (3)(c) の該否判定工数が削減できる．そこで，PF 依存部候補の抽出件数について，手動時の場合とツール利用時の場合を測定し，比較した．

PF 依存部候補数の評価結果を表 5.3 に示す．抽出された候補数は，手動時の場合の約 28k 個 (3a) に対して，ツール利用時の場合は約 17k 個 (3b)（3a に対し 60%）に低減されていた．

(4) 提案手法の再現率

移植工程 (B)(C) を合わせて，最終的に移植に修正が必要だった箇所を，提案手法による抽出可否で分類した．

「提案手法の再現率」の評価結果を表 5.4 に示す．全修正 1107 件の内，(4a) 提案手法で抽出可能な PF 依存部の修正が 5 種 935 件 (全修正に対し 85%)，(4b) 提案手法では抽出不可だが技術的に抽出可能な PF 依存部の修正が 6 種 159 件 (同 14%)，(4c) 完全に対応するのは技術的に困難な PF 依存部の修正が 3 種 3 件 (同 0.3%)，(4d)PF 依存と無関係の修正が 4 種 10 件 (同 0.9%) であった．また，(4a) で修正した PF 依存部（5 種）は，修正難易度が「易」2 種，「普」1 種，「難」2 種であり，修正重要度が「2」1 種，「3」2 種，「4」2 種であった．また，(4a) で修正した PF 依存部（5 種）の全てが，社内の過去の移植事例から得た 23 種に含まれており，MISRA-C から得た 16 種のものはなかった．

(3)PF 依存部候補数の評価との関係を述べると，「(3b) ツール利用で得た 16,904 個の PF 依存部候補」の内，実際に PF 依存であり移植に際して修正を要したのが (4a) の 935 個である．その他の修正 (4b)(4c)(4d)（合計 172 箇所）は，(4a) に関する移植工程 (B)(C) で見つけたものであるが，「(3a) 手動で得た 28,139 個の PF 依存部候補」は比較用に求めたものであり，移植には利用していない．

(5) 手動時とツール利用時の再現性の比較

本評価の移植において実際に変更が必要だった PF 依存

表 5.1　移植工数

移植工程	PF 間移植手順	工数（時間）	全工程に対する割合
(A) 文法上の整合	(1) (2)	77.5	24%
(B) PF 依存部の修正方法の検討と実装	(3) (4)	115.0	35%
(B1) ツール抽出可		104.5	32%
(B2) ツール抽出不可		10.5	3%
(C) 動作確認とデバッグ	(5)	132.0	41%
合計		324.5	

表 5.2　PF 特性調査機能の有効範囲

提案手法で用いる全 PF 依存種	PF 特性調査機能で得られる PF 特性に関連する PF 依存種	全 PF 依存種に対する割合
39 項目	19 項目	49%

表 5.3　PF 依存部候補数

抽出された PF 依存部候補数 (3a: 手動時)	抽出された PF 依存部候補数 (3b: ツール利用時)	手動時に対する割合
28,139 個	16,904 個	60%

表 5.4　提案手法の再現率

修正内容	修正種類	修正件数	全修正に対する割合
全修正	18 種	1107 件	-
(4a) 提案手法で抽出可能な PF 依存部の修正	5 種	935 件	85%
(4b) 提案手法で抽出不可だが技術的に抽出可能な PF 依存部の修正	6 種	159 件	14%
(4c) 完全に対応するのは技術的に困難な PF 依存部の修正	3 種	3 件	0.3%
(4d)PF 依存と無関係の修正	4 種	10 件	0.9%

表 5.5　手動時とツール利用時の再現性の比較

PF 依存部の区分	該当数 (*1)	(5a) に対する割合
(5a) 変更が必要	741 個	-
(5b) 手動で抽出可能	741 個	100%
(5c) ツールで抽出可能	732 個	99%

*1: 比較可能な PF 依存部のみ

箇所の内，手動時の場合とツール利用時の場合とで，それぞれ何箇所を PF 依存部候補として抽出できたかを測定し，比較した．なお，キーワード検索と静的解析を併用して抽出した PF 依存部は，正確に比較できないため評価対象外とした．

「手動時とツール利用時の再現性の比較」の評価結果を表 5.5 に示す．(4) 提案手法の再現率の評価における「修正した PF 依存部 (4a)(4b)(4c)（合計 1097 箇所）」の内，比較対象となる PF 依存部は 741 個あった (5a)．この 741 個の PF 依存部は，(3)PF 依存部候補数の評価における「(3a) 手動で得た 28,139 個の PF 依存部候補」の中に 741 個 (5a に対し 100%) 含まれており (5b)，(3)PF 依存部候補数の評価における「(3b) ツール利用で得た 16,904 個の PF 依存部候補」の中に 732 個 (同 99%) 含まれていた (5c)．

5.4. 考察

(1) 移植工数

本評価では，移植工程 (A)「文法上の整合」の工数は全体の 24% であったが，今回の移植対象では OS・ドライバ整備の工数が発生していない．

汎用 OS を搭載した汎用ハードウェア機器に移植する場合は，標準ライブラリや標準ドライバが入手可能であり，本工数はそれほど大きくならないが，特殊な周辺デバイスを持つ機器への移植などの場合は，ドライバの新規作成が必要となり，本工数がより大きくなると想定される．

また，今回の移植では，(4) 提案手法の再現率の評価における「提案手法で抽出可能な PF 依存部の修正 (4a)」件数（935 件）の 9 割以上を，修正難易度「易」と「普」の PF 依存部が占めていた．修正難易度「難」の PF 依存部が多い場合は，同程度の修正件数であっても移植工数がより大きくなると想定される．

(2)PF 特性調査機能の有効範囲

本評価では，PF 依存部抽出支援ツールにより，全依存種の 49%（19 項目）に関連する PF 特性を得られ，この 19 項目全てが，本ツールの自動化対象であった．

PF 依存箇所が全 PF 依存種に平均的に分布しており，かつ，ツールで得られた PF 特性が PF 間移植前後で全て同じである場合は，PF 依存リストに従って闇雲に抽出・判定するのと比較して，49% の工数削減が可能であると言える．

また，ツールで抽出可能な PF 依存部に関する修正方法の検討と実装作業の工数 (B1) は全移植工程の 32% であったので，同程度の割合の工程においてこの工数削減効果が得られると言える．

(3)PF 依存部候補数

PF 依存部候補の抽出件数は，手動時の場合に対して，ツール利用時の場合は 60% に低減されていた．これは，手動時がキーワード検索のみであるのに対して，ツール利用時はソースコードの静的解析結果を併用することで不要な候補が除外されているためである．例えば，VxWorks で利用されている socket のような一般名称や i などの短い名称の関数を抽出する際，キーワード検索だけでは同名の変数も抽出されてしまうが，静的解析との併用によりこれを避けられる．

PF 依存部候補の該否判定工数が均一な場合は，ツール利用により PF 依存の該否判定工数が 40% 削減可能であると言える．

(4) 提案手法の再現率

本評価では，提案手法の再現率は (4a)85% であった．ただし本評価の移植では，全 PF 依存種に関して修正が必要だったわけではないので，今回の移植で評価されなかった PF 依存種の再現率が 85% とは限らない．

また，本評価では (4b) 提案手法では抽出不可だが技術的に抽出可能な PF 依存部の修正があった．(4b) の例としては，関数に対して仕様で未定義の値が引数として渡されている場合があった．これは，仕様未定義の関数をリストアップしておけばツールで対応可能である．

また，本評価では (4c) 完全に対応するのは技術的に困難な PF 依存部の修正があった．(4c) の例としては，ROM コードを書き換えている場合があった．const 変数を非 const 変数にキャストしている場合などは文字列検索により抽出可能だが，const 変数のアドレスを直接参照している場合などへの対応は困難と考える．

また，提案手法の再現率が 85% であり，かつ，「提案手法で抽出可能な PF 依存部の修正 (4a)」件数（935 件）の 9 割以上を，修正重要度が「3」と「4」の PF 依存部が占めていた．(4a) に該当する PF 依存部の種類が 5 種類と少ないため参考情報としてだが，修正重要度の高い PF 依存部から修正する方針は正しそうである．

(5) 手動時とツール利用時の再現性の比較

本評価では，手動時とツール利用時とで再現性はほぼ同じであった．これにより，考察 (3) で示した「ツールによる PF 依存部候補の除外」は，PF 依存部を残しつつ PF 非依存部を除外しており，正しく機能していると言える．

なお，ツール利用時に抽出できなかった 9 個の PF 依存部（= (5a)741 − (5c)732）は，従来通りのテストで検出された．これらは，ポインタ関数など，静的解析エンジンとして用いた cppcheck の仕様が原因で抽出できなかったものであり，ツールの改造により対応可能である．

5.5. その他の議論

(1) ツールの作成，教育 / 習熟コスト

提案手法の場合，ツールの作成，教育 / 習熟コストが増えることが考えられる．しかしこれは最初の 1 プロジェクトのみに必要なコストであり，次回以降の適用では不要と

なる．

(2) その他の工数削減効果

本評価では，PF 依存部候補の検索を手動で行う場合，PF 依存種の一覧が既にあることを前提にした．PF 依存種の一覧がなく，多数の PF 依存部分を動作確認テストの不具合解析で発見し修正する場合は，手戻り工数が大きくなるため，提案手法によりこの工数の削減が期待できる．

(3) 工数削減以外の効果

提案手法は，既存ソフトウェアの PF 間移植を容易化するだけでなく，ソースコードの移植性を高めることで，実機レス開発の実現 / 導入を容易化できる．

(4) 汎用性

提案手法は，C 言語で記述されたソースコードならば他の PF 間移植にも適用可能である．つまり，SH/ARM から VxWorks/Linux への移植以外にも応用できる．この場合，関連キーワードの設定ファイルを追加 / 変更するだけで良い．

表 Ex.1　PF 依存種一覧

分類	PF 依存種（検索対象コード）
社内の過去の移植事例から得たもの	レジスタへの直接アドレス参照
	SRAM、DRAM への直接アドレス参照
	FlashROM への直接アドレス参照
	型のデータ長
	ポインタから固定幅整数型へのキャスト
	明示的、暗黙的な型変換
	char の符号
	符号付き整数のシフト
	エンディアン変換コード
	整数型をバイト配列にキャストしてのアクセス
	unicode 文字列の入出力
	ソケット通信用構造体へのデータ入出力
	キャラクタデバイスとの通信
	バイナリファイルの入出力
	構造体メンバへの直値オフセット参照
	ビットフィールド
	負整数の除算、剰余算
	double のバイト幅
	浮動少数点の整数キャスト時の丸め誤差
	プラグマの使用
	アセンブラの使用
	システムコールの呼出
	OS 依存のデータ構造
MISRA-C より得たもの	自動変数の初期化
	標準識別子、定義、マクロの変更
	オブジェクトの重複領域でのコピー
	31 文字以上の識別子
	複数の異なる外部定義
	シフトによるビット拡張
	関数ポインタのキャスト
	const、volatile 変数のキャスト
	式の実行順序
	型のビット幅以上のシフト
	浮動小数点数のビット演算
	インクリメント、デクリメント演算子の演算順
	浮動少数点の比較演算
	異なる配列ポインタの演算、比較
	消滅したオブジェクトの使用
	共用体のメモリ配置

6.　おわりに

開発済みの組込みシステムにおいて，PF を現行と同程度のハードウェアリソースを持つ新 PF へ変更する際のソフトウェア開発工数削減を目的に，ソースコードの PF 依存部抽出による既存ソフトウェアの PF 間移植手法を提案した．

本手法を用いた PF 依存部抽出支援ツールを開発し，実際の製品ソースコードに適用した．典型的な条件では，PF 依存部の検索・判定・修正工数を 49%、PF 依存の該否判定工数を 40% 削減可能であるとの見込みを得，高い工数削減効果が得られることを実証した．

実際の開発では移植方式の検討にかけられる工数は小さいため，正確で効率的な見積り方法の確立が今後の課題である．

また，複数プロジェクトでの提案方式の適用と評価も必要である．例えば，PF 依存種ごとの対応工数の違い，適合率の違いを分析し，プロジェクトごとに変化が大きい要素と，共通的な要素を分類することで，より正確な修正支援，見積りを行うことが可能となる．

【参考文献】

[Clements2002] P. Clements and L. M. Northrop, Software Product Lines: Practices and Patterns. Addison-Wesley, 2002.

[CPC] Sourceforge: Cppcheck. http://cppcheck.sourceforge.net/

[CPT] Techmatrix: C++Test C/C++ 対応自動テストツール . http://www.techmatrix.co.jp/quality/ctest/

[ECL] The Eclipse Foundation: Featured Eclipse Project. http://www.eclipse.org/

[Fowler1999] M. Fowler: Refactoring: Improving The Design of Existing Code, Addison-Wesley (1999).

[IPA2013] IPA: 2012 年度「ソフトウェア産業の実態把握に関する調査」調査報告書 , (2013).

[METI2010A] 経済産業省 : 2010 年版 組込みソフトウェア産業実態調査報告書 - プロジェクト責任者向け調査 -, (2010).

[METI2010B] 経済産業省 : 2010 年版 組込みソフトウェア産業実態調査報告書 - 事業者責任者向け調査 -, (2010).

[METI2011] 経済産業省 :「平成２２年度中小企業システム基盤開発環境整備事業（組込みシステム産業の施策立案に向けた実態把握のための調査研究）」事業報告書 , (2011).

[MSR] The Motor Industry Software Reliability Association: MISRA C. http://www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx

[Ochiai2002] 落合竜一 , 鈴木正人 : リファクタリングとコンポーネント技術による既存ソフトウェアの拡張手法 , 情報処理学会研究報告 ソフトウェア工学研究会報告 2002(23), pp.87-94, (2002).

[PGR] 富士通ソフトウェアテクノロジーズ : PGRelief C/C++. http://jp.fujitsu.com/group/fst/services/pgr/

[Pohl2005] K. Pohl, G. Böckle, and F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., (2005).

[Selic2003] B. Selic, The pragmatics of model-driven development, IEEE Software, vol.20, no.5, pp.19–25, (2003).

[Spinellis2006] D. Spinellis: Code Quality: The Open Source Perspective, Pearson Education, (2006), トップスタジオ (訳): コード・クオリティ , 毎日コミュニケーションズ , (2007).

[Sugiyama2003] 杉山泰一 , 携帯電話のバグを無線ネット経由で修復 , 日経コミュニケーション , 2003 年 8 月号 , pp.70-72, (2003).