組込みソフトウェア開発における
工数削減および期間短縮に関する研究

提出先　　大阪大学大学院情報科学研究科

提出年月　2016年1月

岡 本　周 之

主要論文

I. 学会論文誌等掲載論文

(1) 岡本周之, "既存資産の拡張開発に適したドライバ層エミュレーションによる実機レス開発方式," 情報処理学会論文誌コンシューマ・デバイス＆システム(CDS), Vol.3, No.3, pp.30-38, 2013.

(2) 岡本周之, 藤原貴之, 楠本真二, 岡野浩三, "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," IPA/SEC journal No.39, pp.8-17, 2015.

関連論文

I. 学会論文誌等掲載論文

(1) 藤原貴之, 岡本周之, "大規模組込み機器向けテスト自動化拡大方式," 情報処理学会論文誌コンシューマ・デバイス＆システム(CDS), Vol.4, No.4, pp.1-10, 2014.

II. 国際会議，研究集会等発表論文（査読付）

(1) K. Yoshimura, J. Shimabukuro, T. Ohara, C. Okamoto, Y. Atarashi, S. Koizumi, S. Watanabe, K. Funakoshi, "Key Activities for Introducing Software Product Lines into Multiple Divisions: Experience at Hitachi," Software Product Line Conference (SPLC) 2011, pp.261-266, 2011.

(2) 藤原貴之, 岡本周之, "大規模組込み機器向けテスト自動化拡大方式," 情報処理学会研究報告 モバイルコンピューティングとユビキタス通信研究会研究報告(MBL), 2013-MBL-67(9), pp.1-8, 2013.

内容梗概

近年，製品やサービスにおけるソフトウェアの重要性が高まっており，ソフトウェア開発において，ソフトウェアの品質，開発コスト，納期の３要素の改善が求められている．ソフトウェア開発技術とは開発効率を向上させて，(a)品質を低下させずに開発コストの削減と納期の早期化を実現するか, (b)開発コストの増加と納期の遅れを生じさせずに品質を向上させる技術と言える．品質には定量評価が困難な特性や，マネジメント要素が複雑に関与する特性があるため，本研究では(a)のアプローチを採用した．開発コストの削減では，初期コストよりも設計・実装・テストコストが大きいため，開発工数の削減が課題である．また納期の早期化では，並行開発による開発期間の短縮が課題である．

また近年では，家電製品，制御機器，移動機器など，特定の機能を持ったソフトウェアがハードウェアに組み込まれた**組込みシステム**が我々の生活に広く深く浸透しており，社会的に重要な位置を占めている．品質，開発コスト，納期の影響が大きく，組込みシステムに組み込まれた**組込みソフトウェア**の開発効率向上が強く求められている．組込みシステムおよび組込みソフトウェアは，以下の特徴を備えることが多い．(1)拡張開発される，(2)理想的なソフトウェア構造の維持が困難，(3)仕様書の記述が不十分，不正確，(4)ハードウェアリソースの制約が厳しい，(5)ハードウェアや OS などプラットフォーム（PF）の変更がある，(6)ハードウェアとソフトウェアが並行開発される．このため組込みソフトウェアで開発効率を向上させるためには，ソフトウェア構造が崩れ，かつ仕様書に頼れないという状況を考慮して，既存ソフトウェアの PF 間移植やハードウェアとの並行開発について，検討する必要がある．

本研究の目的は, 組込みシステムおよび組込みソフトウェアの開発効率の向上である．具体的には，組込みシステムおよび組込みソフトウェアを対象に，(1)設計・実装工程における既存ソフトウェアの PF 間移植と，(2)テスト・デバッグ工程における並行開発の開発工数削減および開発期間短縮を目指した．

先ず，設計・実装工程における既存ソフトウェアの PF 間移植では，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．この課題の解決に際し，既存ソフトウェアを改変せずに PF 間で移植できれば理想的であるが，実際にはソフトウェアの階層構造が崩れ，エンディアンやパディング有無などの PF 特性に依存する実装（以降，PF 依存部と呼ぶ）が点在していることが多く，PF 依存部の抽出・修正が必要となる．さらに仕様書が不完全，不正確のため PF 依存部の抽出に活用できないという状況にも対応できるよう，PF 特性に依存する実装パターン（以降，PF 依存種と呼ぶ）検索によるソースコードからの PF 依存部抽出手法を考案した．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．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結果，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．

次に，テスト・デバッグ工程における並行開発では，実際の製品版ハードウェア（実機）が不十分な開発の初期段階で，ソフトウェアのテスト・デバッグ環境を構築することが課題である．ハードウェア不足を補う実機レス開発方式などが提案されているが，従来の方式では，ミドルウェア層の開発，ユーザ操作による処理，既存資産の拡張開発に同時には対応できていなかった．本研究ではこれらに対応できるよう，ドライバ層エミュレーションによる汎用 PC 上での実機レス開発方式について考案した．本方式では，(A)実機のドライバ層およびハードウェア層と同等の機能を，ドライバ層エミュレータおよび汎用ハードウェア（PC）を用いて実現させ，かつ，(B)ドライバ層エミュレータを，基本機能エミュレータとその上に被せた実機ドライバ I/F からなる構成とした．また，エミュレータ適用に向け必要となるアプリケーション修正については，ドライバ層およびハードウェア層を PF と考えると，実機 PF からエミュレータ適用 PF への PF 間移植に相当することに着目し，(1)の PF 間移植の開発効率向上技術を適用した．さらに，既存資産の拡張開発を行うデジタル TV 向けに，本方式を用いたドライバ層エミュレータを開発し，これを用いた実機レス開発環境を構築して，実機のみの開発ではできなかった並行開発を実現した．デジタル TV における，ミドルウェアと組み合わせたユーザ操作アプリケーションの実際の製品開発に適用した結果，テスト・デバッグ期間を 58%短縮し，組み合わせテスト時のテスト・デバッグ効率を 29%向上させる効果が得られ，高い費用削減効果が得られることを実証した．

謝辞

まず初めに，本研究の全般に関し常日頃より適切なご指導を賜りました，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 楠本真二教授に心から感謝申し上げます.

また，本論文を執筆するに当たりご助言をいただきました，信州大学 学術研究院工学系 岡野浩三准教授，大阪大学大学院 情報科学研究科 肥後芳樹准教授ならびに日立製作所 尾崎友哉氏に感謝申し上げます.

個別の研究につきましては, 既存ソフトウェアのプラットフォーム間移植における開発効率向上の研究に関して，ツール開発，適用評価にご協力いただきました，日立製作所 藤原貴之氏，日立アドバンストデジタル（現 日立産業制御ソリューションズ）開発チームの皆様，ならびに Network Systems & Technologies（NeST，現 QuEST Global）開発チームの皆様に御礼申し上げます.

実機レス開発方式による並行開発における開発効率向上の研究に関しては, 研究開発をご指導いただきました, 日立製作所 大條成人氏（現 日立産業制御ソリューションズ）, ツール開発にご協力いただきました，Simbed の Alexandre Payet 氏（現 Alstom Grid），ならびに適用評価にご協力いただきました, 日立コンシューマエレクトロニクス（現 日立マクセル）開発チームの皆様に御礼申し上げます.

また，本研究を大阪大学大学院 情報科学研究科で進める機会を与えていただきました，日立製作所 森正勝氏，森谷真寿美氏，ならびに原田泰志氏に感謝申し上げます.

最後に，楠本研究室の皆様のご協力に御礼申し上げると共に，家族の理解と支援に感謝いたします.

目次

図一覧

表一覧

第1章 はじめに

1.1 本研究の背景と目的

1.1.1 ソフトウェア開発と QCD の向上

近年，製品やサービスにおけるソフトウェアの重要性が高まっている．利用者が製品やサービスに求める価値は，従来から重要視されてきた機能や性能の他に，近年では心地よい操作感，サービスを通じて得られる体験価値（UX: User eXperience）など多岐に渡っている[Kanbara2010][Muramatsu2011][Kurosu 2012]．これらの価値の実現にはソフトウェアが多くを担っており，ソフトウェアの価値が製品やサービスの価値を左右すると言える．

ソフトウェアの価値を高めるためには，ソフトウェア開発において，ソフトウェアの(1)品質/Quality，(2)開発コスト/Cost，(3)納期/Delivery の３要素（それぞれの頭文字からまとめて QCD とも呼ばれる）の継続的な改善が必要である[Button1996][Manalo2010].

(1) ソフトウェアの品質

ソフトウェアの品質は，ISO 8402:1994 [ISO1994]において「明示又は暗黙のニーズを満たす能力に関する，ある『もの』の特性全体」と表現されており，ソフトウェアの価値そのものである[Tamaki2014]．この特性については，ISO/IEC 9126:1991[ISO1991] (JIS X 0129:1994[JIS1994])で 6 種類定義されており，2001 年の規格改定等を経て，後継規格の ISO/IEC 25010:2011[ISO2011] (JIS X 25010:2013[JIS2013])では，利用時の品質として 5 種類の特性と 11 種類の副特性が，システム/ソフトウェア製品品質として 8 種類の特性と 31 種類の副特性が，それぞれ定義されている．JIS X 25010:2013 における品質の特性と副特性を表 1.1 に示す．これらの品質特性をそれぞれ高めることで，ソフトウェア品質が向上し，ソフトウェアの価値が向上する．

なお本論文では，**ハードウェア**と**ソフトウェア**を統合したものを**システム**と呼ぶ．ハードウェアに搭載されたソフトウェアがハードウェアを制御し，システムとして機能する．また図表などでは，ハードウェアをハード，ソフトウェアをソフトと略して表記する場合がある．

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．

表 1.1　JIS X 25010:2013 における品質の特性と副特性

	特性	副特性
利用時の品質	有効性	有効性
	効率性	効率性
	満足製	実用性，信用性，快感性，快適性
	リスク回避性	経済リスク回避性，健康・安全リスク緩和性，環境リスク緩和性
	利用状況網羅性	利用状況完全性，柔軟性
システム/ソフトウェア製品品質	機能適合性	機能完全性，機能正確性，機能適切性
	性能効率性	時間効率性，資源効率性，容量満足性
	互換性	共存性，相互運用性
	使用性	適切度認識性，習得性，運用操作性，ユーザエラー防止性，ユーザインタフェース快美性，アクセシビリティ
	信頼性	成熟性，可用性，障害許容性（耐故障性），回復性
	セキュリティ	機密性，インテグリティ，否認防止性，責任追跡性，真正性
	保守性	モジュール性，再利用性，解析性，修正性，試験性
	移植性	適応性，設置性，置換性

(2) ソフトウェアの開発コスト

製品におけるソフトウェアの重要性が高まるに連れて，ソフトウェア開発のコストが，製品開発コストの多くを占めるようになっている．情報処理推進機構による国内企業 397 社に対するソフトウェア産業実態調査[IPA2013]では，ソフトウェア開発コストが組込みシステムで 53.6%，エンタプライズ システムで 50.1%を占めることが判明した．文献[IPA2013]を基に作成した開発費用の内訳（開発対象別）を図 1.1 に示す．

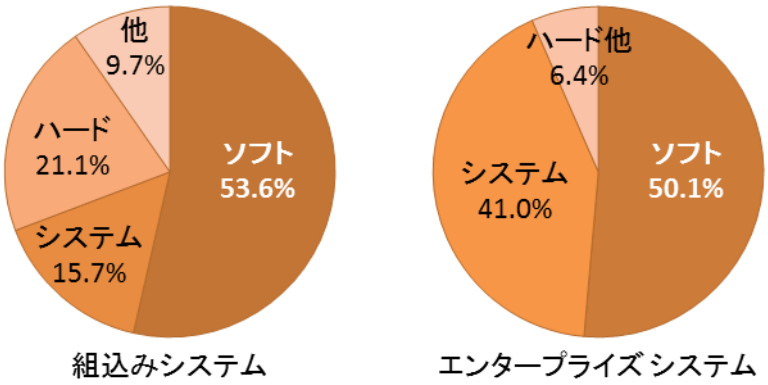


図 1.1　[IPA2013]を基に作成した開発費用の内訳（開発対象別）

ソフトウェア開発においては，ハードウェア開発のように部品コストがかからないため，ソフトウェア開発コストは(式 1)のように表すことができる．なお，本論文では，単に**開発**と述べた場合は**ソフトウェア開発**を指し，ハードウェアを含む製品全体の開発は**製品開発**と述べて区別する．

ソフトウェアの開発コスト ＝　(a) 初期コスト（ライセンス料，開発環境構築コスト，教育コストなど） 　＋(b) 設計・実装・テストコスト（開発者の開発工数単価×開発工数）	**(式 1)**

前述の情報処理推進機構による調査[IPA2013]では，組込みシステムでは，(a)に相当するソフトウェア・ハードウェアの購入・レンタル費および教育研修費が合計 13.5%，(b)に相当する社内人件費，人材派遣費，外部委託費が合計で 86.4%であり，エンタプライズ システムでは，それぞれ 13.7%，85.3%であると判明しており，一般的に(a)より(b) の方が大きいと言える．[IPA2013]を基に作成した開発費用の内訳（費用別）を図 1.2 に示す．

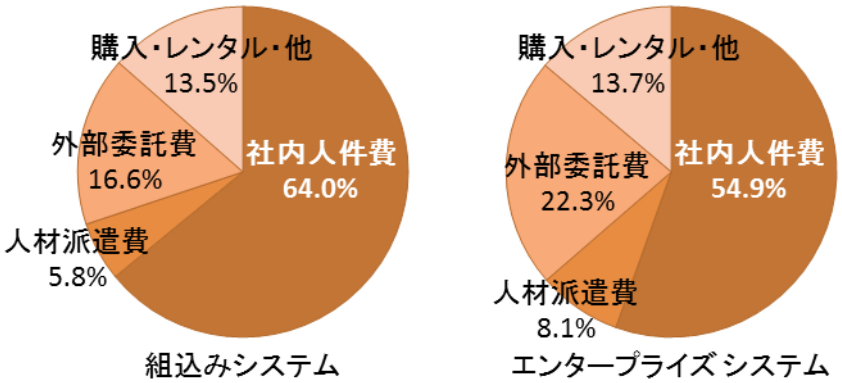


図 1.2　[IPA2013]を基に作成した開発費用の内訳（費用別）

(a) 初期コスト

開発の初期コストは，OSS (Open Source Software)や仮想マシンの活用で削減できる．

OSS[Raymond1999]とは，ソースコードが無償で公開されており，再頒布が自由なソフトウェアである．近年では，OS(Operating System)，デバイスドライバ（以下，ドライバ），ミドルウェア，アプリケーション，開発ツールなど様々な種類の OSS が充実しており，Linux Foundation は，エンド ユーザーや SI 企業が安心して導入出来る OSS ツール選定の際の一つの目安として，12 分野 550 種の OSS を紹介している[LF2015]．経済産業省が Linux Foundation オープンソースソフトウェア活用動向調査[LF2013]を基にまとめた調査[TIS2015]によると，国内における OSS 導入数は，2008 年から 2013 年で 2.1 倍に増加している．

OSS は無償のため，開発環境構築[Yamada2007]に用いると開発ツールベンダへ支払うライセンス料を削減できる．また，製品ソフトウェアの一部として他社のソフトウェア（ライブラリ等）を導入すれば，自社開発の人件費は削減できるが，ライブラリベンダへのライセンス料の支払いが必要となる. ここで製品ソフトウェアの一部に OSS を用いれば，ライセンス料を削減できる[Noda2009].

また，仮想マシンとは，仮想化ソフトウェアによって，あるコンピュータの動作環境を別なコンピュータの上で再現する仕組みである．ARM アーキテクチャの実行環境を X86 アーキテクチャの PC 上で再現する等が可能な異種 CPU エミュレータの QEMU [Bellard2005]や, Linux の実行環境を Windows 搭載 PC 上で再現する VMware の Player[VMP], Workstation [VMW] [Ward2002]等がある．仮想マシンの動作状態は，イメージ情報としてホストマシンの記憶装置に保持されているので, このデータを別なホストマシンに展開することで，全く同じ動作状態の仮想マシンを再現できる[Yamagata2006].

仮想マシンを開発環境に導入することで，統一された開発環境を各開発者の PC 上に正確かつ迅速に構築できる. 複数の開発ツールを各 PC にインストールし, 開発プロジェクトに合わせてツール毎の設定を行う方式に比べ，環境構築工数，すなわちコストが削減できる．文献[Yamagata2006]では，仮想マシンのイメージを用いて計算機環境を構築する方法により，環境構築時間を 12%削減できたと報告されている.

また，教育コストは，ツールや技術を共通化して使うことで削減できる．そのためには，ソフトウェア基盤[Shimamoto1995]を整備してソフトウェアを階層構造[Inoue2005]とし，機能ごとにソフトウェアを部品化して再利用[William2005] [Northrop2006]しやすくすることが重要である．特定製品の異なるモデルや異なる製品間で，共通のソフトウェア基盤やソフトウェア部品を使えば，ソフトウェア基盤のシミュレータ（エミュレータ）や，ソフトウェア部品の利用方法なども共通化され，新規に導入する場合に比べて教育コストが削減できる．ソフトウェアとツール以外の共有対象として，文献[Takeuchi2007]では構成の定義, 性能やセキュリティなどの非機能管理情報, 開発作法, 開発ノウハウなどが挙げられている.

(b) 設計・実装・テストコスト

設計・実装・テストコストは，(式 1)で示した通り，開発者の開発工数単価×開発工数で表される．開発工数単価は，人件費が比較的安価な地域で開発を行うオフショア開発を導入することで削減できる．ただしオフショア先が国外の場合は，言語や考え方の違いを埋めるために，通訳や翻訳，日本人同士なら不要な前提条件の確認などオーバヘッド工数がかかる[Saito2007]．また，開発工数単価は，開発者との契約や開発拠点の物価など, 開発技術以外の要素が大きく影響するため, 本研究では検討の対象外とする.

開発工数は，開発者数分の延べ開発期間で表される．開発効率（生産性とも言う）を向上することで, 同じ作業を完了させるのに必要な時間, すなわち開発工数を削減できる．開発効率の向上技術については，テスト自動化 [Fewster1999]，モデルベース開発（モデル駆動開発）[Szekely1996][Jackson2002][France2007]，形式手法[Abrial2005][Nakajima2007]，ソフトウェアプロダクトライン[Pohl2005][SPL][Yoshimura2011]など多数あり，第 2 章で詳細を述べる．

(3) ソフトウェアの納期

ソフトウェアの納期とは，ソフトウェア開発のある段階が完了し，次のステップに進む時期，または社内の別部署や顧客に開発したソフトウェアを納める時期を指す．つまり，ソフトウェアの開発終了時期と言い換えることができる．ソフトウェアの納期，すなわち開発終了時期を早めることで，製品開発の終了時期も早まり，市場動向に合わせて適切なタイミングで製品投入ができるようになる．

開発終了時期は，開発開始時期と開発期間から(式 2)のように求められる．ただし，ソフトウェアの開発開始時期は，市場調査や企業戦略，顧客との契約や開発人員の確保など開発技術以外の要素が大きく影響するため，本研究では検討の対象外とする．

$$\text{開発終了時期} = \text{開発開始時期} + \text{開発期間} \quad \textbf{(式 2)}$$

ソフトウェアの開発期間は，見積もりモデルの一種である COCOMO(Constructive Cost Model) [Boehm1981]では(式 3)の形で表され，開発工数と密接な関係にあるが，開発工数は(2)(b)で考慮済みのため，ここではハードウェアとソフトウェアの並行開発による，開発期間の短縮について検討する．

$$\text{開発期間} = A \times \text{開発工数}^{B} \quad \textbf{(式 3)}$$

ハードウェアとソフトウェアが並行開発される場合, 片方が完成していないと統合したシステムテストができないという問題が生じる．特にソフトウェアの場合は，テスト工程で不具合を検出し，実装工程や設計工程に遡って修正した後，再度テストを行う，というサイクルを繰り返す．この工程に時間がかかる[Fewster2014]ため，テスト工程を早期に開始することが開発期間の短縮に重要となるが, ハードウェア開発が完了していない，または十分な数のハードウェアが準備できていない等，ハードウェアの整備が不十分な状況では, ソフトウェアのテスト工程を開始できない. この問題の対策としては，仮想マシンの活用が報告されている[Miyauchi2009][Harashima2012].

製品ハードウェア（実機）を必要としないソフトウェアの動作環境（以降，実機レス開発環境と呼ぶ）を仮想マシンを用いて構築することで，実機の整備が不十分な状況でもソフトウェアの動作確認テストを行うことができる．また，実機と完全に同じ動作環境でなくとも，一定のソフトウェアテストを実施できる．このため，ハードウェア開発中に可能な限りのソフトウェアテストを済ませてしまい，ハードウェアが完成したらすぐに，そのハードウェアでしか確認できないテストを実施することで，開発期間を短縮することが可能となる．実機レス開発環境を適用した開発プロセスを図 1.3 に示す．たとえば文献[Miyauchi2009]では，実機レス開発環境の適用によりデバッグ工数の 45%を実機完成前に実施でき，実機デバッグ期間を 60%短縮できたと報告されている．

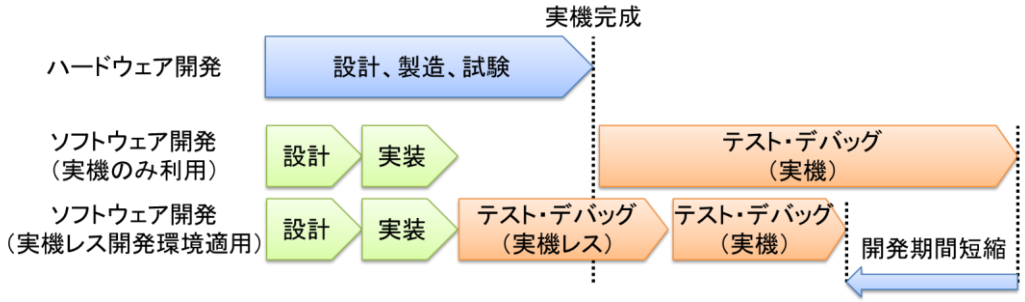


図 1.3　実機レス開発環境を適用した開発プロセス

1.1.2 本研究の目的

通常，QCD の 3 要素は互いにトレードオフの関係にある[Uchiya2015]．例えば，品質を向上させるために開発工数を増やすと，開発コストが増えるか納期が遅れてしまう．また，開発コスト削減のためにテスト工程を削減すると，品質が劣化してしまう．また，品質を維持したまま納期を早めるためにより多くの開発者を投入すると，開発コストが増えてしまう．

ソフトウェア開発技術とは，開発効率を向上させて，(a)品質を低下させずに開発コストの削減と納期の早期化を実現するか，(b)開発コストの増加と納期の遅れを生じさせずに品質を向上させる技術と言える．1.1.1 (1)で述べた品質には，定量評価が困難な特性や，マネジメント要素が複雑に関与する特性がある[Nonaka2010]ため，本研究では(a)のアプローチを採用した．

また，1.1.1 (2)で述べた通り，開発コストの内(a)初期コストよりも(b)設計・実装・テストコストの方が一般的に大きいため，本研究では，(b)設計・実装・テストコストの削減を主な検討対象とした．また，1.1.1 (2)(b)で述べた通り，開発工数単価は検討対象外であるため，本研究の課題は開発工数の削減である．また納期では，1.1.1 (3)で述べた通り，並行開発による開発期間の短縮が課題である．

以上をまとめると，本研究の目的はソフトウェアの開発効率向上であり，より具体的には，開発工数削減および開発期間短縮である．

1.1.3 本研究の対象

本研究の対象は, 筆者が所属する企業で取り扱っている組込みシステムおよび組込みソフトウェアとした．ここで**組込みシステム**とは，特定の機能を実現するため，特定の機能を持ったソフトウェアが主に専用のハードウェアに組み込まれたシステムであり，汎用ハードウェア上でソフトウェアを追加・変更・削除して多様な用途に用いることができる汎用システムと対を成す概念である. 組込みシステムに組み込まれたソフトウェアのことを**組込みソフトウェア**と呼ぶ.

組込みシステムの種類は, テレビ, デジカメ, エアコンなどの家電製品や, 工場のモータ，自動車のエンジン，エレベータなどの制御機器，POS (Point of Sale)端末，ATM (Automated Teller Machine), 自販機などの支払機器, 心電計, CT (Computed Tomography), MRI (Magnetic Resonanse Imaging)などの医療機器，監視カメラ，カードリーダ，指紋読取機などのセキュリティ機器，船舶，航空機，人工衛星などの移動機器，リモコン，マウス，キーボードなどの入力機器，スピーカ，モニタ，アラームなどの出力機器，明るさセンサ，人感センサ，雨滴センサなどのセンサ類と，多岐に渡っている．

また近年では, ハードウェアのコスト低下や小型化, 性能向上などにより, ソフトウェアを組込み易くなり，組込みシステムが急激に増えている．総務省の平成 27 年度情報通信白書[MIC2015]によると, 電子部品デバイスの価格は 1986 年からの 24 年間で 75%, 1997 年からの 12 年間で 50%低下しており，一方で IoT (Internet of Things)の数は 2011 年からの 10 年間で 5.1 倍に増えると予測されている．[MIC2015]を基に作成した(a)電子部品・デバイス物価指数の推移，および(b)IoT デバイス数の推移・予測を図 1.4 に示す.

一方, 組込みシステムおよび組込みソフトウェアは, 以下の特徴を備えることが多い. 特徴の説明と，これにより引き起こされる問題や解決すべき課題は，第 2 章で述べる.

(1) 拡張開発される
(2) 理想的なソフトウェア構造の維持が困難
(3) 仕様書の記述が不十分，不正確
(4) ハードウェアリソースの制約が厳しい
(5) ハードウェアや OS などプラットフォーム（PF）の変更がある
(6) ハードウェアとソフトウェアが並行開発される

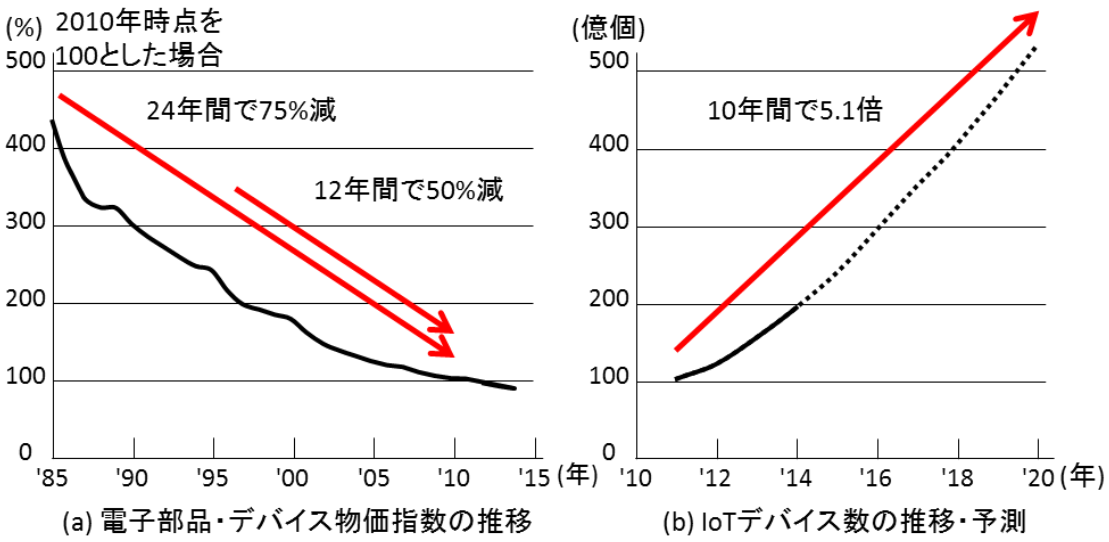


図 1.4　[MIC2015]を基に作成した電子部品価格および IoT 数の推移・予測

このため組込みソフトウェアで開発効率を向上させるためには，ソフトウェア構造が崩れ，かつ仕様書に頼れないという状況を考慮して，既存ソフトウェアの PF 間移植やハードウェアとの並行開発について，検討する必要がある．

1.2 本研究の概要

本研究が対象とするソフトウェア開発の工程は，設計，実装，テスト，デバッグからなる．本研究では，組込みシステムを対象に，設計・実装工程における既存ソフトウェアの PF 間移植と，テスト・デバッグ工程における並行開発の開発効率向上を図った．

1.2.1 既存ソフトウェアの PF 間移植

まず，設計・実装工程における既存ソフトウェアの PF 間移植について述べる．ここでは，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．

この課題の解決に際し，既存ソフトウェアを改変せずに PF 間で移植できれば理想的であるが，実際にはソフトウェアの階層構造が崩れ，エンディアンやパディング有無などの PF 特性に依存する実装（以降，PF 依存部と呼ぶ）が点在していることが多く，PF 依存部の抽出・修正が必要となる．さらに仕様書が不完全，不正確のため PF 依存部の抽出に活用できないという状況にも対応できるよう，PF 特性に依存する実装パターン（以降，PF 依存種と呼ぶ）検索によるソースコードからの PF 依存部抽出手法を考案した．

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．本手法により，以下の効果が得られる．

(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．

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．

1.2.2 テスト・デバッグの並行開発

次に，テスト・デバッグ工程における並行開発について述べる．ここでは，実際の製品版ハードウェア（実機）が不十分な開発の初期段階で，ソフトウェアのテスト・デバッグ環境を構築することが課題である．

この課題の解決には，エミュレータを用いた実機レス開発が有効であるが，組込みソフトウェア開発に多い，

(1) 製品の重要（差別化）機能がミドルウェアで実装されており，アプリケーションがそのミドルウェアと密接に連携している場合
(2) ユーザ操作による処理がある場合
(3) 既存資産の拡張開発を行っている場合

に適した，実機レス開発方式の確立が求められる．これを満たすドライバ層エミュレーションによる汎用 PC 上での実機レス開発方式について考案した．

本方式では，(1)実機のドライバ層およびハードウェア層と同等の機能を，ドライバ層エミュレータおよび汎用ハードウェア（PC）を用いて実現させ，かつ，(2)ドライバ層エミュレータを，基本機能エミュレータとその上に被せた実機ドライバ I/F からなる構成とした．

これにより，ドライバ層でエミュレートするためミドルウェア層とアプリケーション層のテスト・デバッグが行える．また，ハードウェア層のレベルで詳細にエミュレート

しないため，オーバーヘッドが小さく処理遅延が少なくなり，処理時間に関するテストに適用できる．また，基本のエミュレーション機能と独立してドライバ I/F を容易に変更可能なため，既存資産の拡張開発が可能となる．

また，エミュレータ適用に向け必要となるアプリケーション修正については，ドライバ層およびハードウェア層を PF と考えると，実機 PF からエミュレータ適用 PF への PF 間移植に相当することに着目し，前述の PF 間移植の開発効率向上技術を適用した．

既存資産の拡張開発を行うデジタル TV 向けに，本方式を用いたドライバ層エミュレータを開発し，これを用いた実機レス開発環境を構築して，実機のみの開発ではできなかった並行開発を実現した．デジタル TV における，ミドルウェアと組み合わせたユーザ操作アプリケーションの実際の製品開発に適用した結果，テスト・デバッグ期間を 58%短縮し，組み合わせテスト時のテスト・デバッグ効率を 29%向上させる効果が得られ，高い費用削減効果が得られることを実証した．

1.3 本論文の構成

1.3 では，本論文の構成について示す．まず第 2 章で，本研究の目的であるソフトウェアの開発効率向上について，アプローチと主な技術，関連研究を述べる．本研究の対象である組込みシステムと組込みソフトウェアの開発に固有の課題と関連研究を述べ，本研究の課題を定義する．

続いて第 3 章では，本研究の課題である，設計・実装工程における既存ソフトウェアの PF 間移植の開発効率向上について，PF 依存種検索によるソースコードからの PF 依存部抽出手法の研究を述べる．課題とアプローチ，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，PF 依存部抽出支援ツールの開発，製品ソースコードへの適用評価を述べる．

続いて第 4 章では，本研究のもう 1 つの課題である，テスト・デバッグ工程における並行開発の開発効率向上について，既存資産の拡張開発に適したドライバ層エミュレーションによる実機レス開発方式の研究を述べる．課題とアプローチ，既存資産のドライバ層をエミュレートする実機レス開発方式，デジタル TV 向け実機レス開発環境の開発，デジタル TV 開発への適用評価を述べる．

最後に第 5 章で，本研究を総括し，今後の方針を述べる．

なお，本論文では図表などにおいて，ハードウェアをハード，ソフトウェアをソフト，アプリケーションをアプリ，ミドルウェアをミドルと略して表記する場合がある．

第2章 組込みソフトウェア開発の効率向上技術と関連研究

第 2 章では，まず，本研究の目的であるソフトウェアの開発効率向上について，アプローチと主な技術，関連研究を述べる．次に，本研究の対象である組込みシステムと組込みソフトウェアの開発に固有の課題と関連研究を述べ，最後に本研究の課題を定義する．

2.1 ソフトウェアの開発効率向上技術

2.1 では，本研究が対象とするソフトウェアの開発工程について述べた後，開発効率向上技術についてアプローチ別に整理し，代表例としてモデルベース開発，形式手法，ソフトウェアプロダクトライン（SPL）について述べる．

2.1.1 ソフトウェアの開発工程

本研究が対象とするソフトウェア開発の工程は，設計，実装，テスト，デバッグからなる．図 2.1 にソフトウェア開発の工程を示す．

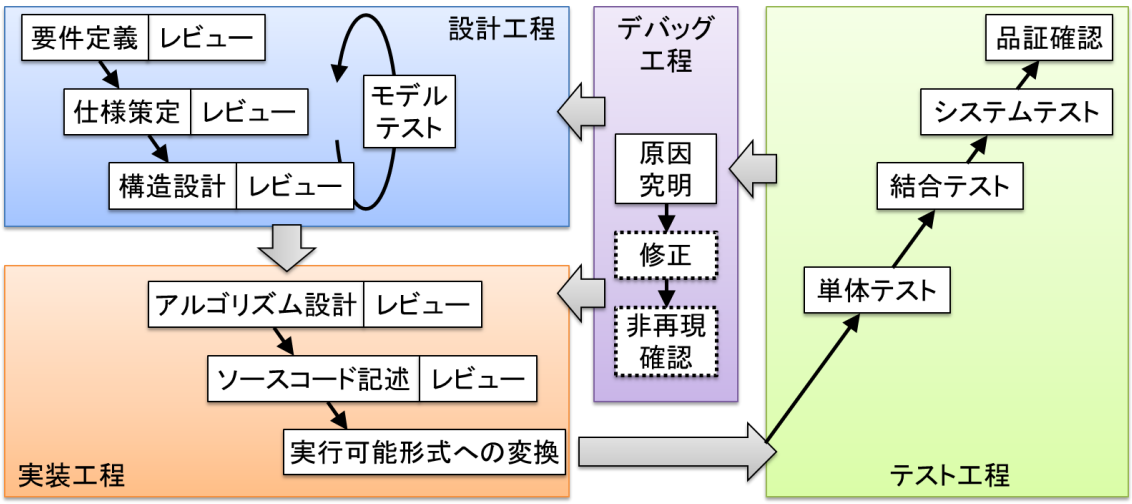


図 2.1　ソフトウェア開発の工程

(1) 設計工程

設計工程は，主に要件定義，仕様策定，構造設計とそれらのレビューからなる．要件定義は，要求分析とも呼ばれ，システムに対する要件からソフトウェアに対する要件を定義する．仕様策定では，ソフトウェア要件を満たすよう，ソフトウェアの仕様を策定する．構造設計では，仕様を満たすようソフトウェア構造を設計する．設計対象となる

構造の要素としては，ソフトウェア部品や情報テーブルなどがあり，これらの入出力や振る舞いを定義する．実装工程に進む前に設計情報（モデル）を基にモデルテスト（モデル検証）を行う場合もあるが，本研究ではこれを設計工程とした．

(2) 実装工程

実装工程は，アルゴリズム設計，ソースコード記述それらのレビュー，実行可能形式への変換からなる．アルゴリズム設計では，設計工程で定義した設計情報に従って，ソフトウェアの詳細な処理アルゴリズムを策定する．処理アルゴリズムは，抽象度の高いものは設計工程で定義する場合もあるが，本研究では実装工程とした．その後，処理アルゴリズムを特定の言語でソースコードとして記述し，記述したソースコードを実行可能形式のプログラムに変換する．

(3) テスト工程

テスト工程は，ソースコードの解析や実行可能形式プログラムの動作により，前工程で定義された設計，仕様，要件を満足していること（＝不具合が検出されないこと）を確認する工程である．詳細アルゴリズム設計を確認する単体テスト，構造設計を確認する結合テスト，ソフトウェア仕様を確認するシステムテスト，製品としての要件を確認する品証確認からなる．

(4) デバッグ工程

デバッグ工程は，テスト工程において不具合を検出した場合，その不具合の原因究明と，その対策としての修正，修正後に不具合が再発しないことを確認する非再現確認を行う工程である．実際には不具合の原因に応じて，設計工程または実装工程で修正を実施した後，テスト工程で非再現確認するが，本研究では便宜上，いずれもデバッグ工程とした．

ソフトウェア開発工程では，前工程の不具合が後工程に影響を与える．例えば，設計工程で正しく設計されていない場合，実装工程で設計通りにソフトウェアを実装しても，要求や仕様が満たされないという問題が生じる．

2.1.2 開発効率向上のアプローチ

ソフトウェア開発の効率向上に関して，2.1.1 で述べたソフトウェア開発工程に従い，アプローチ別に整理して関連技術を述べる．開発効率向上のアプローチには，表 2.1 に示す 3 種類がある．

表 2.1　開発効率向上のアプローチ

#	アプローチ	例
(1)	作業の自動化	ツールによる自動化：ソースコード実装，テストケース作成，テストの操作・記録・繰返し，デバッグ用の情報測定・可視化など
(2)	作業量の削減	組合せ最適化によるテストケースの最小化，部品化・オープン化による開発規模の削減など
(3)	不具合の早期検出	レビュー，モデルテストなどによるフロントローディング

(1) 作業の自動化

各開発工程の作業を自動化することで，手動で作業する場合に比べて開発効率を向上できる．一般的に自動化した場合は，作業時間が短くなるし，作業精度が高まるからである．また自動化により，詳細情報の記録と追跡や，同じ作業の正確な再現が実現できることも開発効率向上に寄与する．

例えば，テスト工程を自動化[Fewster1999][Fujiwara2014]した場合を考える．手動テストでは，担当者がテスト指示書を読み，理解し，操作して，その結果を読み取り，記録する必要がある．また，デバッグ工程での原因解析のために，どのような条件や操作タイミングで不具合が顕在化するのか，記録する必要もある．さらに，仕様やソースコードを修正する度に，同じテストを繰り返す必要がある．

一方，自動テストでは，テスト指示書を読んで理解する時間が不要となるし，理解や操作での誤りを排除できるため，テスト工数を削減できる．また，詳細なテスト条件や厳密な操作タイミングを自動で記録し，高精度に再現できるようになるため，デバッグ工程での原因解析や，修正後の再テストの効率を向上できる．手動テストと自動テストの工程比較を図 2.2 に示す．文献[Fujiwara2014]では，ソフトウェア内部にテスト自動実行エンジンを埋め込み，動作状態の直接取得や操作ログの活用による自動化方式を提案し，不具合調査テストや非再現確認テストなどの工数を合計 71.6%削減できたと報告した．

また，デバッグ工程で解析効率を高めるためには，十分な情報が入手できることが重要である．これに関しては，ソフトウェア統合開発環境（Eclipse [ECL], Microsoft の Visual Studio [VST]など）のデバッガ機能，OS のトレーサ機能（Linux の ftrace [FTR]など），性能解析ツールであるプロファイラ（gprof [GPR]，OProfile [OPR]など），組込みシステム向けに CPU 機能をエミュレートする ICE (In Circuit Emulator)などが活用されている．これらを用いることで，自動的に測定，記録された必要な情報を効率よく入手でき，可視化

されて表示されるため，デバッグ効率を向上できる．トレースログの可視化ツールについては，ログの標準形式変換と可視化ルールの整備により，汎用性と拡張性を高める手法などが提案されている[Goto2010].

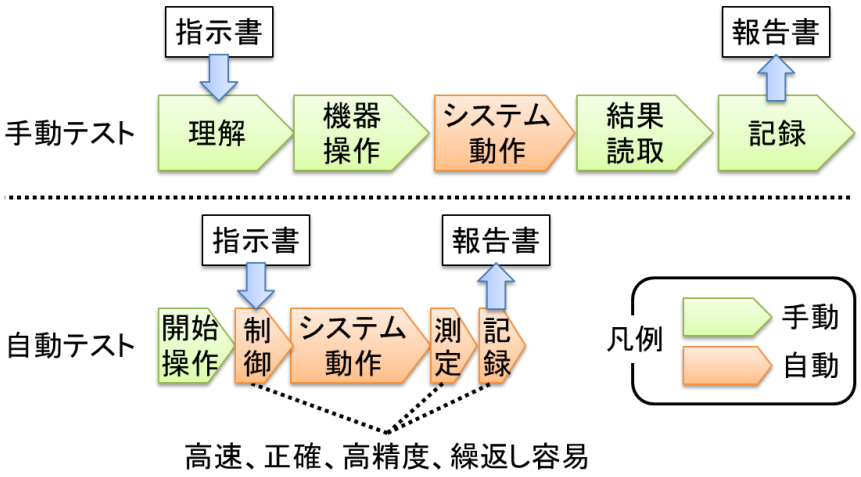


図 2.2　手動テストと自動テストの工程比較

その他の自動化技術としては，要求仕様からのソースコード自動生成や，仕様からのテストケース自動生成などがある．組込みソフトウェア開発に特化した研究として，文献[Yamanaka1999]では，状態遷移仕様から時間制約付き周期処理に対応したコード自動生成法が，また文献[Nakajima2001]では，状態遷移仕様のイベントアクションをデバッガのメモリ値設定に関連づける自動テスト手法が，また文献[Ohta2009]では，動作モデルと状態遷移系を用いた最小テストケース設計手法が提案されている．

(2) 作業量の削減

各開発工程の作業を削減することで，工数を削減し，開発効率を向上できる．重複した作業や不要な作業を削減し，必要最低限のものに絞り込むことで実現する．

例えばテストを行う際，複数の入力を組み合わせる場合は，入力の条件（因子）や値（水準）の種類の多さによっては組み合わせが膨大な数になる．この組み合わせを合理的に削減する手法として，ペアワイズ（オールペア）法[Tsuchiya2007]，直交表[Gomi2015]などがある．ペアワイズ法では，テストケースのどこかで２つの水準の全組み合わせが１つ以上設定され，直交表ではさらに，全組み合わせが同じ数だけ設定される．

また，ソフトウェアの開発規模を削減することで，各工程の作業を削減することもできる．1.1.1 で述べた通り，ソフトウェア基盤[Shimamoto1995]を整備してソフトウェアを階層構造[Inoue2005]とし，機能ごとにソフトウェアを部品化して再利用[William2005][Northrop2006]しやすくすることで，同じ機能を二重に開発することを避けられる．ま

た，OSS や他社が提供するソフトウェア部品を導入すれば，自社が開発するソフトウェアの規模を削減することができる[Noda2009]．これを，組込みソフトウェア開発のオープン化（または水平分業）と呼ぶ[Takada2004]．部品化に関しては，既存ソフトウェアの特定機能を部品化するだけでなく，あらかじめ共通部品とする機能を定め，それに最適化して開発する SPL 開発手法も多数研究されてきた. SPL については 2.1.3 で述べる．

(3) 不具合の早期検出

各工程で作りこまれた不具合を早期に検出することで，手戻り工数を削減でき，開発効率を向上できる．ここで手戻り工数とは，ある工程で見つけた不具合を，工程を遡って修正するために必要となる工数のことである．例として，設計の不具合を修正する場合について，不具合の検出工程の違いによる手戻り工数の比較を図 2.3 に示す．(a)不具合をテスト工程で検出した場合は, 設計工程までさかのぼって設計を修正し, 再度実装, テストを行う必要がある．一方，(b)不具合を設計工程で検出した場合は，手戻り工数は設計工程内のやり直し工数だけで済むため，全体の工数削減につながる．

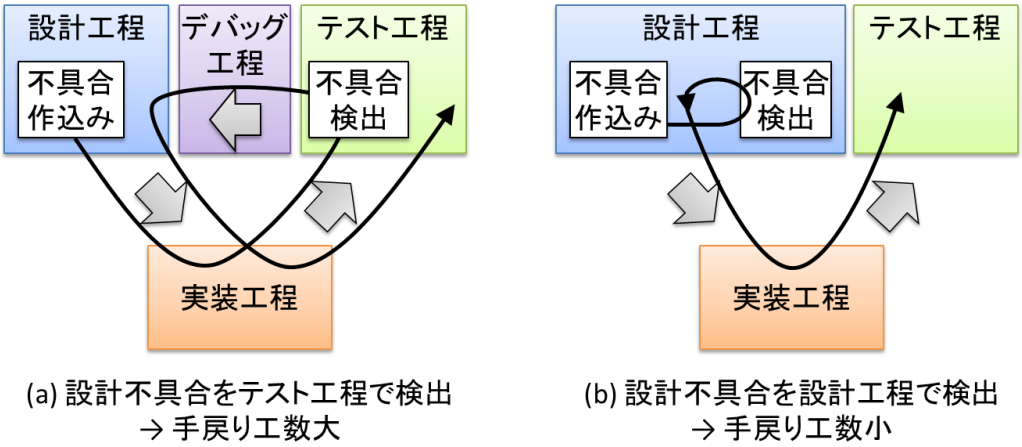


図 2.3　設計不具合の検出工程の違いによる手戻り工数比較

手戻り工数は，不具合の検出が後工程になるほど大きくなる[Wiegers2004][METI2011][Shinkai2015]という特徴があり，たとえば文献[Tassey2002] [Yamamoto2008]では，どちらも上流工程，単体試験工程，結合試験工程，システム試験工程，サービス開始後の誤り修正コストの比を 1：5：10：15：30 としている．このため，設計工程など開発工程の初期で不具合を検出することが重要となる．

これは，フロントローディング[Ikeda2007][Yoshizawa2012]という考え方で，設計工程の検証を，テスト工程ではなく設計工程で行う．代表的な手法として，設計レビュー[Kudo2003] [Yamada2005]の他，モデルベース開発（モデル駆動開発）や形式手法によるモデルテストなどがある．モデルベース開発と形式手法については 2.1.3 で述べる．

2.1.3 モデルベース開発，形式手法と，ソフトウェアプロダクトライン

2.1.2 で述べた開発効率向上の考え方を複合的に活用している開発技術として (1)モデルベース開発（モデル駆動開発），(2)形式手法，(3)ソフトウェアプロダクトラインがある．それぞれの概要と課題を述べる．

(1) モデルベース開発

モデルベース開発（Model Based Development: MBD）は，モデルを用いて仕様を定義し，このモデルを基にして開発する手法である．モデル駆動開発（Model Driven Development: MDD）とも呼ばれる．モデルとは，特定の観点に特化して抽象化した情報であり，図，表，数式など特定の様式で表現される．モデリング言語には，振舞定義用の状態遷移表や状態遷移図，フローチャート[Jyaku2005]，構造や振舞いを示すための 13 種類の図法がある UML (Unified Modeling Language) [ISO2012]など多数ある．モデルベース開発の適用工程の一例を図 2.4 に示す．

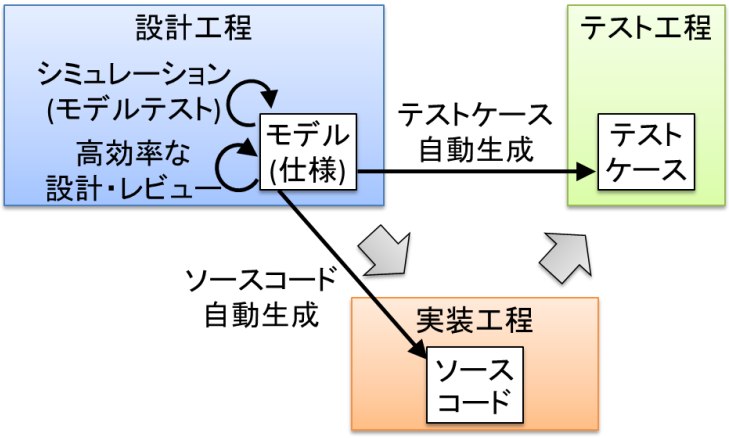


図 2.4　モデルベース開発の適用工程

モデルは，特定の観点に特化して仕様が抽象化されているため，自然言語のみで記された仕様に比べて理解しやすく勘違いを起こしにくい．このため，設計工程におけるデザイン効率やレビュー効率が向上する．

また特定様式に従って論理が記されているため，シミュレーションやコード自動生成が実現できる．キャッツの ZIPC [ZPC]や MathWorks の MATLAB/Simulink [MSL]などのシミュレーションツールを利用することで，設計工程の中でモデルテストにより設計不具合を検出することができ，手戻り工数を削減できる[Watanabe2004][Oho2009]．また，IBM の Rational Rhapsody [RPS]や OSS の blanco Framework [BLA]などのコードの自動生成ツールを利用することで，手動での実装誤りを防ぐことができ，手戻り工数を削減できるとともに，自動化により実装工数を削減できる．

さらに，モデルには仕様情報が含まれているため，テスト開発に流用できる．テストケースの作成やテスト実行環境の構築にモデルを流用することで，テスト工数の削減ができる．前述の ZIPC など，状態遷移モデルに基づくテストケース作成ツールを活用できる．

組込みシステムの制御ソフトウェアでは，MATLAB/Simulink を用いたモデルベース開発が普及しており，これに関した開発効率向上研究が多くなされている．たとえば文献[Kamiyama2010][Tamura2011]では，制御設計からソフトウェア設計への円滑な移行を目的とした Simulink モデルから UML モデルへの変換方法が提案されている．また文献[Ohara2014]では，SILS (Software In the Loop Simulation)の信号量削減により，シミュレーション時間を 90%削減できたと報告されている．

一方でモデルベース開発には，情報処理機器への適用や，既存ソフトウェアのモデル化という課題がある．

コントローラの制御ロジック等，外部環境が連続値を取り数理的に表現可能な場合は，MATLAB/Simulink などのツールが整備されておりシミュレーションが可能だが，IT 機器の情報処理ロジック等，外部環境が離散値を取りタイミングが重要な処理では，モデルが複雑になり過ぎて，シミュレーション環境の構築が困難である．

また，ソフトウェア全体が新規開発の場合は，モデル利用を前提に全体を設計できるが，既存ソフトウェアの拡張開発時にモデルベース開発を導入する場合は，モデルベース開発で作られていない既存部分のモデル化が課題となる．

組込みソフトウェアに多い既存部分のモデル化に対する取り組みとしては，アプリケーション層をモデル化した例[Takahashi2010]や，仕様設計にモデルを導入した例[Kawakami2011]がある．また，理想的なモデルを設計した後で既存ソフトウェアを活用するのではなく，リファクタリングにより既存ソフトウェアの構造と処理の複雑さを軽減した後，ソースコードベースでモデル化する手法が提案されている[Asada2014].

(2) 形式手法

形式手法とは，論理学や離散数学を基にした技法で，仕様記述や検証に用いられる[Nakajima2007]．仕様，設計，実装が正しいことを，テスト工程を待たずに，かつ，正確に検証できるため，工数が削減できる．形式手法の適用工程の一例を図 2.5 に示す．

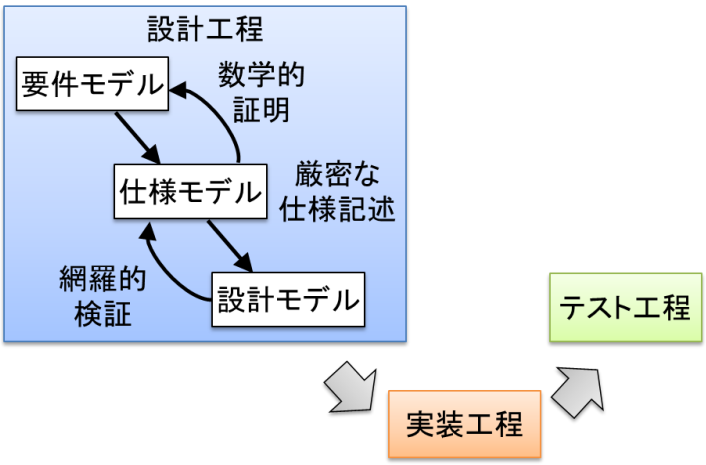


図 2.5　形式手法の適用工程

形式仕様記述とは，B[Abrial2005][Kuruma2007]や VDM++ [Sahara2007][Ishikawa2011]などの仕様記述言語により，仕様を数学的に厳密に記述する手法である．仕様に限らず要件や設計を記す場合もある．形式仕様記述では，論理式など数学的な表現を用いるため，自然言語や図表に比べて仕様を正確に表すことができる．このため，設計時の誤解の原因となる仕様の曖昧さを排除でき，かつ，要件に対する仕様の正しさや仕様に対する設計の正しさなどを厳密に判定できる．

形式検証とは，ソフトウェアの仕様や特性を数学的に検証する手法である．ソフトウェアで検査すべき状態が多い場合，全状態を検証するための膨大なテストケースを作成してテストを実施するには工数がかかるし，無限の場合はテストでは網羅できない．形式検証を用いれば，テストケースを作成することなく検証できるため，このような場合でも保証できる[Nakajima2001]．検証方法には大きく２つ，定理証明とモデル検査がある．

定理証明は，ソフトウェアの振る舞いの正しさ（定理）を検証（証明）する手法である．自動証明系ツールの Vampire[VAM]や対話的証明系ツールの Coq [COQ]などを用いて，数式で表した仕様モデルから推論を積み重ねて数学的に証明するため，ソフトウェアで検査すべき状態を挙げて調べることなく，ソフトウェアの安全性や不具合がないこと等を保証できる[Imai2011].

モデル検査では，モデル記述言語を用いて表したソフトウェアの振る舞いをモデル検査ツールで網羅的に検証する．ツールにより全状態，全遷移の網羅的検査が自動実行される[Nakajima2006]．言語とツールには，Promela と SPIN [Holzmann1997] [Holzmann2004]などがある．たとえば文献[Jyaku2005]では，画面遷移図と処理フロー図から作成したモデルが，画面遷移仕様を表した命題時相論理を満たすかどうかを網羅検証している．

一方で形式手法には，複雑世界の適切なモデル化や，数学的知識の支援という課題がある．

現実の世界は複雑で，ソフトウェアの入出力や振る舞いも複雑である．これらを全て数学的に表現することは不可能なため，ある特性に特化して抽象化してモデル化する必要がある．ここで特性を絞り込み過ぎたり抽象化し過ぎたりすると，仕様記述や検証に必要な情報が欠落してしまう．一方で，抽象化が不足していると現実的な時間内に計算を終えられない．このため，適切な特性を抽出し，適切な抽象度でモデル化する必要がある．

組込みソフトウェアへのモデル検査適用の課題に対し，文献[Mizuguchi2005]では，動作環境のモデル化，検査精度と状態数と反例解析を考慮した適切なモデル化および抽象化，適切な検査項目の設定が挙げられている．解決策の１つとして文献[MCBOK2009]では，状態爆発回避の方法として，モデル化対象の分析によりモデル化範囲を縮める方法，および，一旦モデル化して機械的に縮小し妥当性を示す方法が提示されている．

また，形式手法を用いるためには，数学的な記述や検証の仕組み，適切な手法適用方法などの知識が求められるが，従来のソフトウェア開発者は，この知識を十分に有していない場合が多い[Offutt2008][Ogawa2011]．数学者でないプログラマが形式仕様記述を行うために，また，数学者でないテスタが形式検証を行うためには，ツールなどにより数学的知識を補う，または隠ぺいする必要がある．

(3) ソフトウェアプロダクトライン

ソフトウェアプロダクトライン（Software Product Line: SPL [Pohl2005][SPL]）とは，同系列の製品群のソフトウェアを，共通性と可変性に基づいて全体最適化して開発する手法である．SPL の最大の特徴は，共通化するために戦略的に部品（コア資産）を開発する点である．従来の部品化は製品を開発してから一部を部品化するのに対し，SPL では開発前から共通部品のことを考慮する．

SPL におけるコア資産と製品の関係を図 2.6 に示す．共通部は，プラットフォームなど全製品で共通利用されるソフトウェア部品であり，可変部は，ライブラリ(Lib)など一部製品で共通利用される部品である．これらをまとめてコア資産と呼び，製品開発と分けて開発する．個別製品部分は，再利用を前提としない，アプリケーション(App)など各製品毎に異なるソフトウェア部品である．

従来は，特定製品に最適化してソフトウェアを開発した後で，その一部を切り出して他の製品向けに部品化していた．このため他の製品からすると，その部品がそのままでは使えない場合があり，流用するために部品を改良することもある．これを繰り返すと，

似ているが完全に共通ではない部品が増え，個別に維持管理することになるため，部品化，共通化の利点が限定的になってしまう．

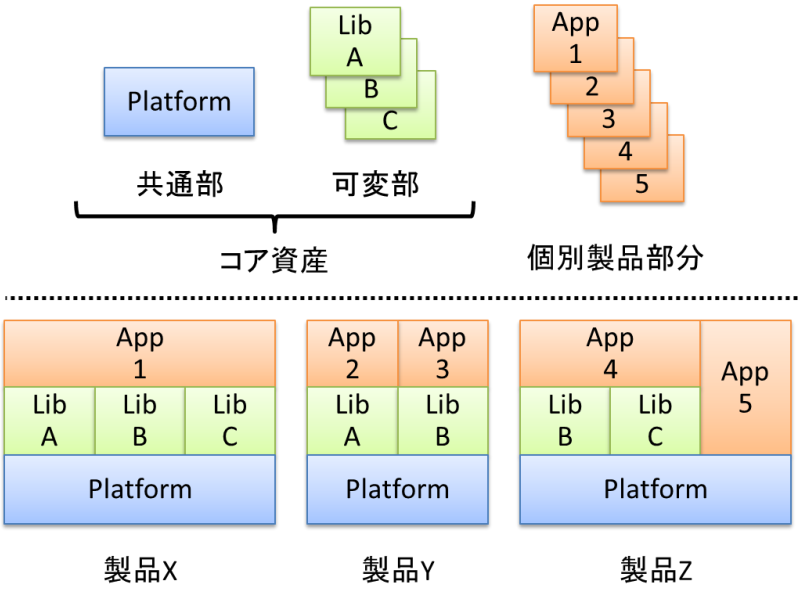


図 2.6　SPL におけるコア資産と製品の関係

これに対して SPL では，共通部品を作る前から，部品を利用する製品群（スコープ），共通化するフィーチャ（機能，特徴）と製品毎に個別開発するフィーチャ，各フィーチャの実装方法と製品毎の切り替え方法，コア資産（共通部と可変部）と個別製品の開発体制などを考慮する[SPL]．このため，コア資産を効率よく流用できる．また，コア資産の設計，実装，テスト工程を集約できる．また，コア資産開発時に，個別製品開発の事情に過度に引っ張られることなく，製品系列の全体最適が実現できる[Pohl2005]．

SPL は，主に組込みソフトウェア向けに発展してきたこともあり，組込みソフトウェア開発に特徴的な既存ソフトウェアへの適用についての研究が多い[Yoshimura2006] [Yoshimura2007] [Yoshimura2008]．既存製品を活用したプロダクトラインの構築や保守進化においては，製品群における派生・バリアントの関係の分析や，要求・コード間のトレーサビリティ（リンク）の分析が，以降のより効率的かつ効果的な製品開発・保守を進めるうえで重要な研究課題となっている[Kishi2013]．SPL 開発に関するトップカンファレンスである Software Product Line Conference（SPLC）においても，SPLC2013 では Traceability & Evolution という研究トラックが設けられ，関連研究[Tsuchiya2012]が発表されている．

一方 SPL には，プラットフォーム変更への対応という課題がある．ソフトウェアを共通部，可変部，個別製品部分に分けて考える際，ソフトウェアの構造としては，OS などのプラットフォームを共通部とし，その上に，取り替え可能な可変部と個別製品部分

を載せるのが自然である．しかし，プラットフォーム変更が予想される場合は，プラットフォーム自体を可変部とし，その他の部分をなるべくプラットフォーム非依存にしなければならない．

2.2 組込みソフトウェア開発の特徴と課題

2.2 では，本研究の対象である組込みシステムと，それに組み込まれた組込みソフトウェアの開発に固有の特徴を述べ，これにより引き起こされる問題や解決すべき課題を述べる．

1.1.1 で述べた通り，ソフトウェア開発のコストが，製品開発コストの半分以上を占めるようになっている．また 1.1.3 で述べた通り，組込みシステムは我々の生活に広く深く浸透しており，社会的に重要な位置を占めている．このため，組込みソフトウェアの開発効率向上が強く求められている．

一方，組込みシステムおよび組込みソフトウェアは，以下の特徴を備えることが多い．これにより引き起こされる問題や解決すべき課題と合わせて表 2.2 に示す．組込みソフトウェア開発では，これらを考慮する必要がある．

表 2.2　組込みシステムおよび組込みソフトウェアの特徴，問題，課題

#	特徴	問題，課題
(1)	拡張開発される	・構造の見直しが必要で，移植等の工数がかかる ・ドキュメントがない場合はコード解析が必要 ・開発手法適用時に既存ソフトウェア見直しが必要
(2)	理想的なソフトウェア構造の維持が困難	・変更時の影響分析工数がかかる ・コードの整形が必要
(3)	仕様書の記述が不十分，不正確	・コードからの情報入手が必要
(4)	ハードウェアリソースの制約が厳しい	・ソフトウェア処理高速化が必要で構造が崩れ易い ・既存ソフトウェア再利用時に仮想化技術が使えない
(5)	プラットフォーム（PF）の変更がある	・仮想化技術が使えない場合，移植工数がかかる
(6)	ハードウェアとソフトウェアが並行開発される	・シミュレータによる実機レス開発環境が必要

(1) 拡張開発される

組込みシステムは，特定の機能を持ったソフトウェアが主に専用のハードウェアに組み込まれているため，製品を開発する度にソフトウェアもハードウェアも新規に作り直していると，開発費がかさむ．このため組込みソフトウェア開発では，既存システムのソフトウェアを流用し，これを基にして拡張開発（差分開発，派生開発とも言う）を行うことが多い．経産省による組込み関連企業 226 社対する組込みシステム産業の実態把握調査[METI2011]では，拡張（差分/派生）開発が 65.5%を占めることが判明した．文献[METI2011]を基に作成したプロジェクト件数比率を図 2.7 に示す．

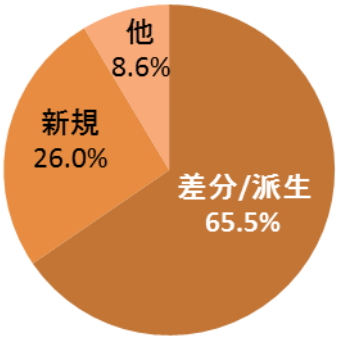


図 2.7　[METI2011]を基に作成したプロジェクト件数比率

具体的には，同じ製品シリーズ（例えばテレビ）であれば，ソフトウェアの基本的な製品機能（例えば放送受信と画面表示）はシリーズを通して同じであるため，製品の新しいモデル（例えば録画機能付き高解像度テレビ）を開発する場合，現行モデルを元に新機能部分（例えば録画機能）を追加で開発したり，性能（例えば画面表示の解像度）を向上したりする．

拡張開発は，適用初期は開発工数の削減に貢献するが，繰り返すにつれてソフトウェア構造の不適合などの問題を引き起こす．拡張開発を繰り返すとソフトウェアの規模が拡大し，機能も高度化，複雑化するため，初期モデルでは最適と考えられていたソフトウェア構造が，最新モデルでは最適ではなくなる．これに対応するため，ソフトウェア構造を定期的に見直す必要がある．ソフトウェア構造を見直す際は，新しいソフトウェア基盤の構築はもちろん，既存ソフトウェアを部品として再利用するにも新基盤との調整などの移植作業工数がかかる．ソフトウェア構造の見直し例を図 2.8 に示す．

例えば，当初は単機能アプリケーション（APP）または単純な機能で基本ソフトウェア（OS）を用いない OS レスの構造であったソフトウェア[Iijima2008]が，機能追加を繰り返すうちに，タスクのスケジューリングやメモリの集中管理が必要となり，μITRON [ITR]や VxWorks [VXW]などのリアルタイム OS を導入する．さらに，製品が通信機能や

セキュリティ機能，高度な GUI などを備えるようになり，Windows [WIN]や Linux [LIN]などの高機能 OS や，HTML [HTM]や Java [JAV]などのアプリケーション基盤（フレームワーク）を取り入れる．

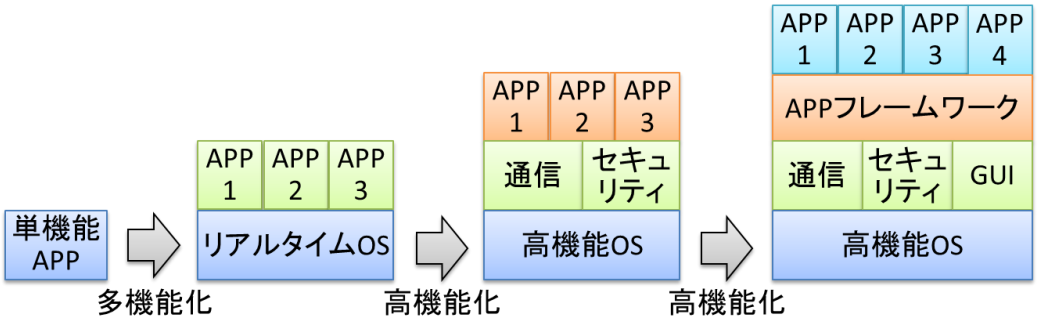


図 2.8　ソフトウェア構造の見直し例

また拡張開発では，過去に開発した際のドキュメントなどが残っていないことが多く，ソースコードからソフトウェア構造や仕様を理解する必要がある[Iijima2008]．さらに，開発技術を幅広く適用するためには，新規開発部分だけでなく既存部分にも適用することが求められる．しかし既存ソフトウェアの開発時には将来適用される技術のことが考慮されていないため，構造や開発工程が最適化されておらず，適用時に見直すなどの対応が必要となることが多い．これに対しては 2.1.3 で述べた通り，既存ソフトウェアへの MBD 適用に関する研究[Takahashi2010][Kawakami2011][Yamaguchi2012] [Asada2014]や，既存ソフトウェアへの SPL 適用に関する研究[Yoshimura2006][Yoshimura2007][Yoshimura2008][Tsuchiya2012]などがある．

(2) 理想的なソフトウェア構造の維持が困難

組込みソフトウェアでは，理想的な階層構造を維持できていない場合が多い．例えば，ソフトウェア構造は，性能要件と保守性のトレードオフを考慮して設計するのが望ましい[IPA2012]が，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[Takada2004]．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．さらに，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]．ソフトウェア構造を保つ・崩す呼出し方の例を図 2.9 に示す．

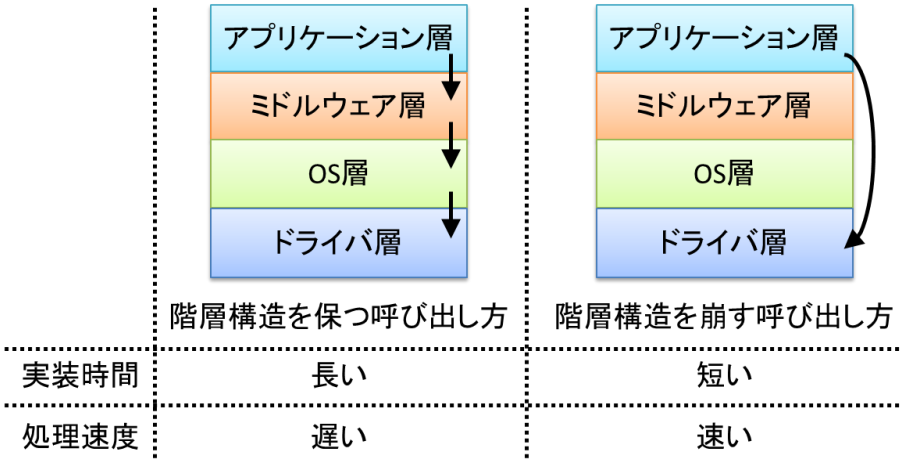


図 2.9　ソフトウェア構造を保つ・崩す呼び出し方の例

理想的なソフトウェア構造が維持されている場合は，ソフトウェアを効率良く開発することができる．例えば，図 2.9 のようにソフトウェアが階層毎に分離されており，下位層と上位層の独立性が高い場合は，ドライバ層のロジックを変更してもアプリケーション層に影響を与えない．しかし，前述のような状況でソフトウェア構造が崩れ，アプリケーション層からドライバ層のロジック部分を直接呼び出している場合，ドライバ層を変更する際に，ソースコードを解析して影響範囲を特定し，影響を与える先のソースコード（例えばアプリケーション層）も合わせて変更する必要がある．

組込みソフトウェアの開発では，開発着手後の仕様変更や設計変更などさまざまな変更が入ることが少なくない．変更時には，(1)変更要求情報の記録，(2)変更による影響分析，(3)変更計画立案・実行，(4)変更結果の確認が必要であり，特に，ソフトウェア内部で独立性が低い，あるいは，周囲との強結合を有する部分，つまり理想的なソフトウェア構造が崩れている部分の変更時は，事前の影響分析（波及効果解析）[Lehnert2011]が重要である[IPA2007].

このように，理想的なソフトウェア構造が維持できていない場合は変更工数が増加してしまう．これに対して，保守・派生開発を対象としたソフトウェア進化研究が多数なされている[Omori2012B]．ソフトウェアの振舞いを変えずにソフトウェア構造を整形するリファクタリング技術[Fowler1999] [Hatano2003][Higo20011]については，前述の影響分析をコードを元に行うプログラムスライシング技術[Weiser1981][Maruyama2002]，コピー＆ペーストの多い組込みソフトウェアのリファクタリングに必要なコードクローン（類似コード片）の検出技術[Kamiya2002][Higo2008]および分析・評価技術[Ueda2003][Kadota2003]，組込みソフトウェアに対するリファクタリング前後のプログラム等価性検証技術[Ichii2015]などが報告されている．

(3) 仕様書の記述が不十分，不正確

組込みソフトウェアでは，仕様書やソースコードのコメントに，必要な情報の記載がない, または記載があっても誤っていることが多い[Spinellis2006][Morisaki2014]. 例えば，将来の拡張（ミドルウェアの変更など）のことを十分に考慮していない場合は，その拡張開発時に必要となる情報（ミドルウェアへの依存情報など）が記載されないことがある．また，開発時間が不足している場合は，コードのみを修正し，仕様書やコメントを修正しないことがある．

このように組込みソフトウェアでは，仕様書やソースコードのコメントが不十分，不正確な場合があるため，これらから必要な情報を正しく得られるとは限らない．これに対応するためには，ソースコードのプログラム処理部分（コメント以外の部分）から必要な情報を取得する手法が必要である．ソースコードを解析してソフトウェアの情報を取得するリエンジニアリング技術としては，前述のコードスライシング技術[Weiser1981][Maruyama2002]のような分析（プログラム解析）技術の他に，構文解析技術[Yanagi2010]，可視化技術[Uehara2010]がある．

(4) ハードウェアリソースの制約が厳しい

組込みシステムは，主にその製品に固有の専用ハードウェアを用いており，また，大量生産されたり安価に販売される製品が多いため，コスト制約が厳しい場合が多い．このため，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することが求められる[Nakagawa2007]．例えば，CPU の演算処理能力が高くないシステムで, 利用者の利便性や操作感を高めるために素早い反応を実現しようとすると，ソフトウェア処理の高速化が必要となる．これは(2)で述べた通り，ソフトウェアの階層構造を崩す要因となる．

また，既存ソフトウェアの再利用や複数 OS の共存に際しては，有効な手段の１つとして 1.1.1 で述べた仮想化技術[Bellard2005][Ward2002]があるが, 仮想化技術は大量のメモリを消費するため，メモリ容量が大きくないシステムでは採用できない[Jones2011].

(5) プラットフォームの変更がある

組込みシステムではハードウェアや OS 等の PF の変更を行う場合がある．例えば，部品コスト削減，供給停止に伴う代替，性能向上を目的とした CPU の変更や，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，開発効率向上を目的とした OS の変更がある．

ここでコスト削減のため CPU を安価なものに変更する場合，他のハードウェアやソフトウェア開発費も削減が求められる．CPU 変更の際，同時にメモリを高額な大容量の

ものに変更することはできないため，(4)で述べた通り，仮想化技術など豊富なメモリを必要とする技術は適用できない[Jones2011]．このような場合は，既存ソフトウェアの PF 間移植が必要となり，また，PF 間移植工数の削減が求められる．PF 変更と PF 間移植の関係を図 2.10 に示す．

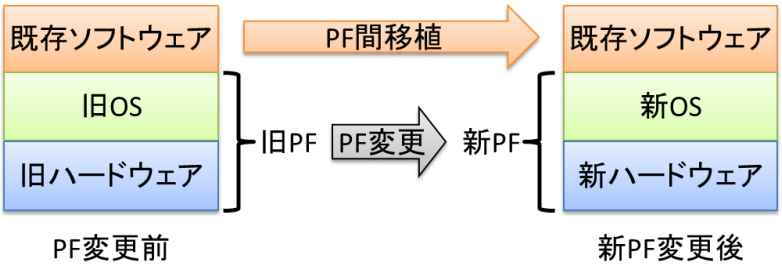


図 2.10　PF 変更と PF 間移植の関係

組込みソフトウェアの PF 間移植の開発効率向上に関しては，ソフトウェア構造に多層アーキテクチャを用いて移植性を高める方式[Abe2003]が主流だが，最初からその構造で設計するか，構造見直しにより実現する[Omori2012A]必要がある．開発時から移植性を考慮して設計するという観点では，2.1.3 で述べた SPL もこれに寄与するが，適切なソフトウェア部品と構造が求められる[Yoshimura2006]点は同じである．

(6) ハードウェアとソフトウェアが並行開発される

組込みシステムでは，ハードウェアもソフトウェアもそのシステムに固有であることが多いため，通常，製品を開発する場合，ハードウェアとソフトウェアの両方とも開発する必要がある．このとき，製品全体の開発期間短縮のために，ハードウェアとソフトウェアが並行して開発される[Miyauchi2009]（図 1.3 参照）．

並行開発では，開発の前半の段階において，ソフトウェアの一部分が完成した時に，まだハードウェアが完成していないという状況が生じる．この場合，先行して完成したソフトウェアの動作確認をしようにも，動作環境が整備できない．また，製品開発途中のハードウェアの製造は試作であるため，製造速度が遅く，価格も高い．このため，製品開発期間を通じて，ソフトウェアのテスト・デバッグのために用意できる最新ハードウェア数が不足する[Harashima2012]．

これに対応するためには，実際の製品版ハードウェア（実機）を用いずに，ソフトウェアのテスト・デバッグ環境を構築する必要があり，シミュレータ（エミュレータ）を用いた実機レス開発環境の構築が課題となる．組込みシステム向けの実機レス開発環境については，1.1.1 で述べた通り，文献[Miyauchi2009] [Harashima2012]で報告されている．

2.3 本研究の課題

2.1 で述べた通り，開発効率向上に関して，モデルベース開発や形式手法，SPL など多くの開発技術が研究されているし，効果も確認されている．しかし，2.2 で述べた通り，組込みソフトウェア開発には固有の制約や課題があり，これらの開発技術の適用が困難な場合がある．

例えば，モデルベース開発や形式手法では適切なモデル作成が重要であるため，**(1)拡張開発される**ことが多い組込みソフトウェアでは，新規開発部分だけでなく既存部分のモデル化も必要である．しかし，その既存部分は**(2)理想的なソフトウェア構造の維持が困難**なため，モデル化に適していない構造となっている場合が多い．理想的でない構造をそのままモデルにしても，十分な開発効率向上は期待できない[Asada2014]．また，形式検証では厳密な仕様を活用するが，組込みソフトウェア開発では**(3)仕様書の記述が不十分，不正確**な場合があるような状況であり，現状のソースコード重視の開発体制から仕様重視の開発体制に移行するのは困難と考える．

このため本研究では，モデルと形式手法を導入せず，また仕様書に頼ることなく，既存ソフトウェアのソースコードから得られる情報を中心に，開発効率向上を図ることにした．さらに 2.2 に述べた組込みソフトウェア開発の特徴を考慮し，本研究の課題として以下の２つを定義した．各課題の開発工程における位置づけを図 2.11 に示すとともに，第 3 章，第 4 章で詳細を述べる．

(1) 既存ソフトウェアの PF 間移植手法の考案と，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の工数削減（設計・実装工程）

(2) 拡張開発向け実機レス開発方式の考案と，並行開発におけるソフトウェア開発の効率向上および期間短縮（テスト・デバッグ工程）

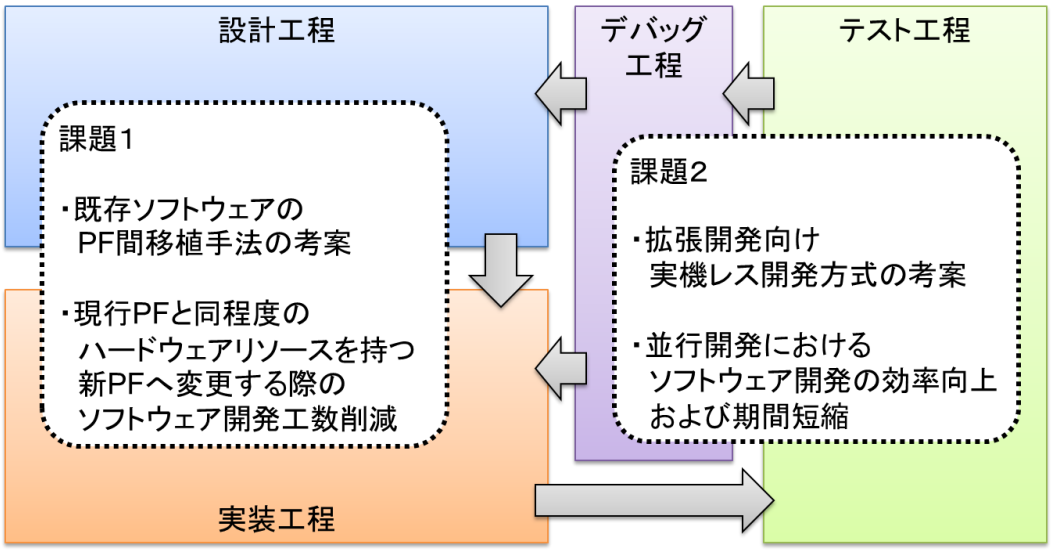


図 2.11　本研究の課題とソフトウェア開発工程における位置づけ

第3章 プラットフォーム依存部抽出手法の研究

第 3 章では，エンディアンやパディング有無などのプラットフォーム（PF）特性に依存する実装（以降，PF 依存部と呼ぶ）と，PF 特性に依存する実装パターン（以降，PF 依存種と呼ぶ）に着目し，本研究の課題である，設計・実装工程における既存ソフトウェアの PF 間移植の開発効率向上について，PF 依存種検索によるソースコードからの PF 依存部抽出手法の研究を述べる．課題とアプローチ，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，PF 依存部抽出支援ツールの開発，製品ソースコードへの適用評価を述べる．

3.1 導入

2.2 で述べた通り，組込みシステムではハードウェアや OS 等の PF の変更を行う場合がある．例えば，部品コスト削減，供給停止に伴う代替，性能向上を目的とした CPU の変更や，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，開発効率向上を目的とした OS の変更がある．

一方 1.1.1 で述べた通り，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）削減のためソフトウェア開発効率向上が必要である．また，2.2 で述べた通り，組込み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）開発が多く，さらに，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．本研究では，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 依存部抽依存部抽出手法を考案した．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，2.1.3 で述べた通り，モデルベース開発，形式手法，ソフトウェアソフトウェアプロダクトライン（SPL）開発などがある．本研究で考案した手法は，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，SPL 開発技術と組み合わせて用いることも可能である．

3.2 課題とアプローチ

3.2 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，ならびに本研究で考案した手法とその効果を述べる．

3.2.1 PF 間移植手順

PF の変更方法には次の(1)～(3)がある（表 3.1 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．

(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．

表 3.1　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CPU → 新ソフト / 新OS / 新CPU	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CPU → 既存ソフト / 既存OS / 仮想CPU	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CPU → 既存ソフト 修正 / 新OS / 新CPU	既存ソフトを新PF向けに一部改修	△（中）	○ 既存ソフトと同等

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．そこで本研究では，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した．

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，既存ソフトウェアの 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，2.2 で述べた通り，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（図 2.9 参照）．

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（図 2.9 参照）．

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 3.1 に示す．

(1) PF 依存層（ドライバ層など）を新 PF（ハードウェア，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（ミドルウェア層，アプリケーション層など）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．

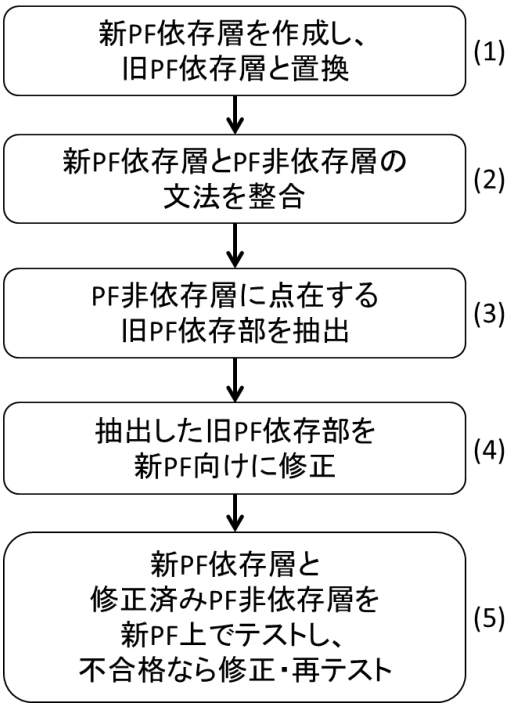


図 3.1　PF 間移植手順

3.2.2 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，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 3.2 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．

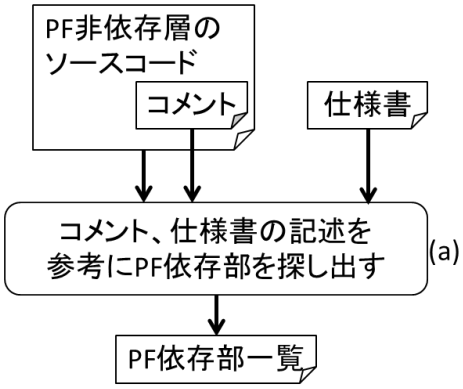


図 3.2　従来の PF 依存部抽出手法

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[METI200B]．

3.2.3 考案した 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，本研究で考案した 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．

本研究で考案した PF 依存部抽出手法を図 3.3 に示す．本手法では，(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．

本研究で考案した手法の特徴は，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．

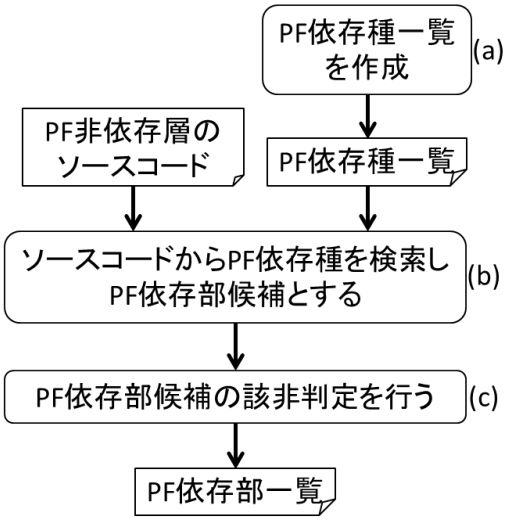


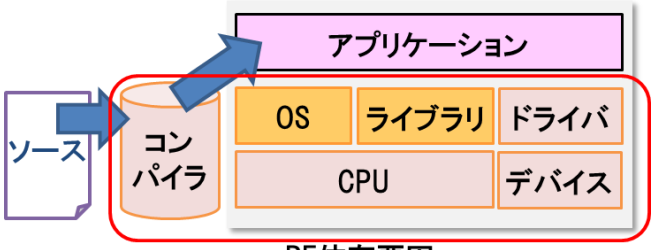
図 3.3　考案した PF 依存部抽出手法

3.3 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

3.3 では，本研究で考案した 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．

3.3.1 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への影響と，対応方法を図 3.4 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，CPU や CPU 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．



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		
CPU		
コンパイラ		

図 3.4　PF 依存要因と対応方法

3.3.2 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，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 3.2 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**という PF 特性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 3.10 に記す．

また，移植時の修正難度を「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．修正難度の基準と該当する PF 依存種数を表 3.3 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 3.4 に示す．これにより，例えば，修正難度が高く修正重要度が低い PF 依存部よりも，修正難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 3.2　PF 依存種一覧（全 39 種）の抜粋

PF依存種 （PF特性に依存する 実装パターン）	PF依存種が 依存する PF特性	修正 難度	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 3.3　修正難度の基準と該当する PF 依存種数

修正 難度	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

表 3.4　修正重要度の基準と該当する 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

3.4 PF 依存部抽出支援ツールの開発

本研究で考案した 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．3.4 では，本ツールについて述べる．

3.4.1 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 3.5 に示す．本ツールは，新旧それぞれの PF（組込みシステムの試作ボード等）の上で動作する(1)Check モジュールと，OS として Windows を搭載した汎用 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．同じく OSS である cppcheck[CPC]を改良してソースコードの静的解析エンジンとして用い，検索機能として OSS の grep コマンドを用いた．ツールの開発規模は数 kLOC で，C および C++ で実装した．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 3.2 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．

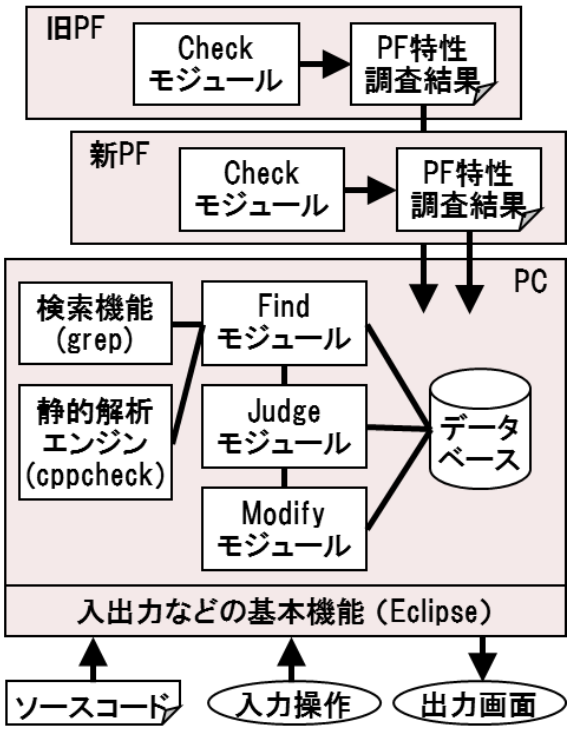


図 3.5　PF 依存部抽出支援ツールの概略機能構成

(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(1)を図 3.6 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．

図 3.6　PF 依存部抽出支援ツールの出力画面例(1)

(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(2)を図 3.7 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．

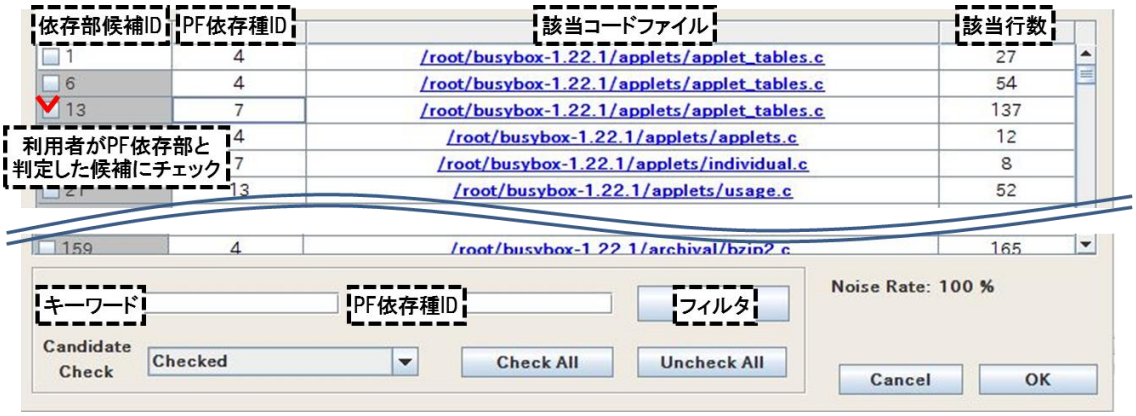


図 3.7　PF 依存部抽出支援ツールの出力画面例(2)

(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 つを選択すると，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(3)を図 3.8 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．

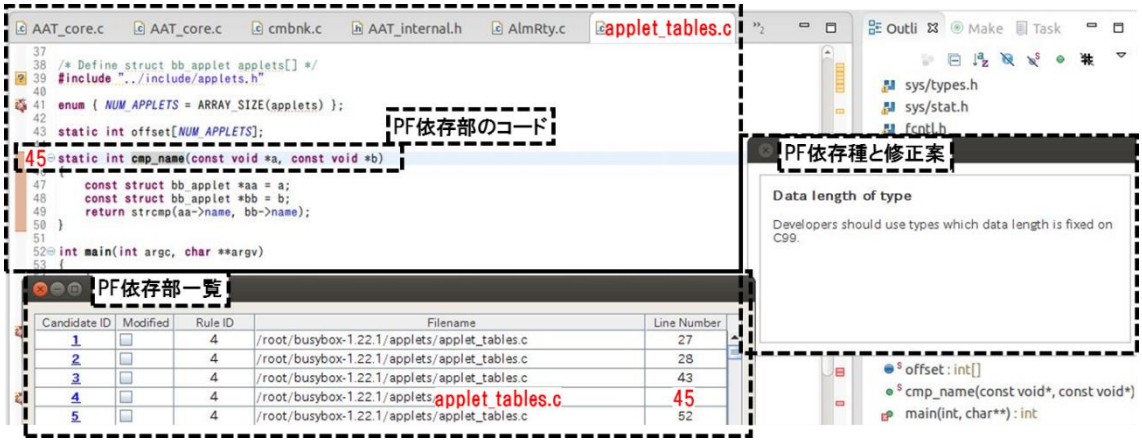


図 3.8　PF 依存部抽出支援ツールの出力画面例(3)

なお，図 3.6，図 3.7，図 3.8 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．

3.4.2 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．

3.4.3 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（図 3.9 参照）．

```
char buf1[];
unsigned char *buf2;
void *buf3;

send(sockfd, buf1, len, flags);  // 抽出しない
send(sockfd, buf2, len, flags);  // 抽出しない
send(sockfd, buf3, len, flags);  // 抽出する
```

図 3.9　関数検索＋名前確認＋引数確認方式での抽出処理例

(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 と signed の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．

3.5 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，本研究で考案した PF 依存部抽出手法を適用し評価した．

3.5.1 適用対象

本評価における考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，CPU 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，3.4 で述べた PF 依存部抽出支援ツールとして実装した．

3.5.2 適用手順

本評価における PF 間移植では，図 3.1 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，考案した手法を適用した．

3.5.3 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，図 3.1 の手順(1)(2)を移植工程(A)**文法上の整合**，同手順(3)(4)を移植工程(B)**PF 依存部の修正方法の検討と実装**，同手順(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 3.5 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 依存部の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．

表 3.5　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	

(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 3.6 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，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．

表 3.6　PF 特性調査機能の有効範囲

考案手法で用いる全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%

(3) PF 依存部候補数

図 3.1，図 3.3 の 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 3.7 に示す．抽出された候補数は，(3a)手動時の場合の約 28k 個に対して，(3b)ツール利用時の場合は約 17k 個（3a に対し 60%）に低減されていた．

表 3.7　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%

(4) 考案手法の再現率

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，考案手法による抽出可否で分類した．

考案手法の再現率の評価結果を表 3.8 に示す．全修正 1107 件の内，(4a)考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)考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．(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．(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た．

また，(4a)で修正した PF 依存部（5 種）は，修正難度が「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，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)ツール利用で抽出された PF 依存部候補 16,904 個の内，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

個であるので，考案手法の適合率は 5.5%（=935/16904）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)手動で抽出された PF 依存部候補 28,139 個は比較用に求めたものであり，移植には利用していない．

表 3.8　考案手法の再現率

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)考案手法で抽出可能な PF依存部の修正	5種	935件	85%
(4b)考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%

(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 3.9 に示す．(4)考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)．

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．

表 3.9　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

3.5.4 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)考案手法の再現率の評価における(4a)考案手法で抽出可能な PF 依存部の修正件数（935 件）の 9 割以上を，修正難度「易」と「普」の PF 依存部が占めていた．修正難度「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誤検出を避けられる（図 3.10 参照）．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．

```
// VxWorksの関数「socket」「i」を抽出しようとする際
int socket;                    // 本ツールは抽出しない（キーワード検索のみだと抽出）
for (i==0; i<10; i++){ ... };  // 本ツールは抽出しない（キーワード検索のみだと抽出）
```

図 3.10　本ツール適用時に誤検出を回避できる例

(4) 考案手法の再現率

本評価では，考案手法の再現率は(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．一方で適合率が 5.5%と低いのは，本研究では PF 依存部の抽出漏れによる手戻り工数を削減するため，適合率よりも再現率の向上を優先したためである．

また，考案手法の再現率が 85%であり，かつ，(4a)考案手法で抽出可能な PF 依存部の修正件数（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)で示した不要な PF 依存部候補のツールによる除外は，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．

3.5.5 その他の議論

(1) ツールの作成，教育/習熟コスト

考案手法の場合，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，考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

考案手法は，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

考案手法は，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．

3.6 まとめ

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 間移植手法を考案した．

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考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．

3.7 付録

表 3.10　PF 依存種一覧

分類	PF依存種(PF特性に依存する実装パターン)
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

第4章 実機レス開発方式の研究

第 4 章では，本研究の課題である，テスト・デバッグ工程における並行開発の開発効率向上について, 既存資産の拡張開発に適したドライバ層エミュレーションによる実機レス開発方式の研究を述べる．課題とアプローチ，既存資産のドライバ層をエミュレートする実機レス開発方式，デジタル TV 向け実機レス開発環境の開発，デジタル TV 開発への適用評価を述べる．

4.1 導入

近年, 組込みシステムの開発ではシステムの多機能化に伴いソフトウェア開発規模が増加している. 製品開発費に占めるソフトウェア開発費は，文献[Tamaru2012]によると，2002 年度の 36.3%から 2010 年度の 50.0%にまで増加しており，情報処理推進機構による 2012 年度のソフトウェア産業実態調査[IPA2013]では，53.6%と判明している．一方，開発製品の競争力強化のために, 事業課題として開発期間の短縮と開発コストの低減がよりいっそう求められている．

また，2.2 で述べた通り，組込みシステムではハードウェアとソフトウェアが並行して開発されるという特徴があるため,

(1) 開発の前半の段階において，ソフトウェアを動作させるハードウェアがない
(2) 開発期間を通じて，テスト・デバッグのために用意できるハードウェア数が不足する傾向にある

という問題がある．これらの問題に対しては，実機ハードウェアを必要としないソフトウェアのテスト・デバッグ環境（以降，実機レス開発環境と呼ぶ）を用いた開発が有効である．

本研究では，AV コンシューマ機器などの組込みソフトウェア開発に多い,

(1) 製品の重要（差別化）機能がミドルウェアで実装されており，アプリケーションがそのミドルウェアと密接に連携している場合
(2) ユーザ操作による処理がある場合
(3) 既存資産の拡張開発を行っている場合

に適した，ドライバ層エミュレーションによる汎用 PC 上での実機レス開発方式について考案した．

4.2 課題とアプローチ

4.2.1 従来の実機レス開発方式とその問題

従来の実機レス開発方式とその問題点を図 4.1 に示す．

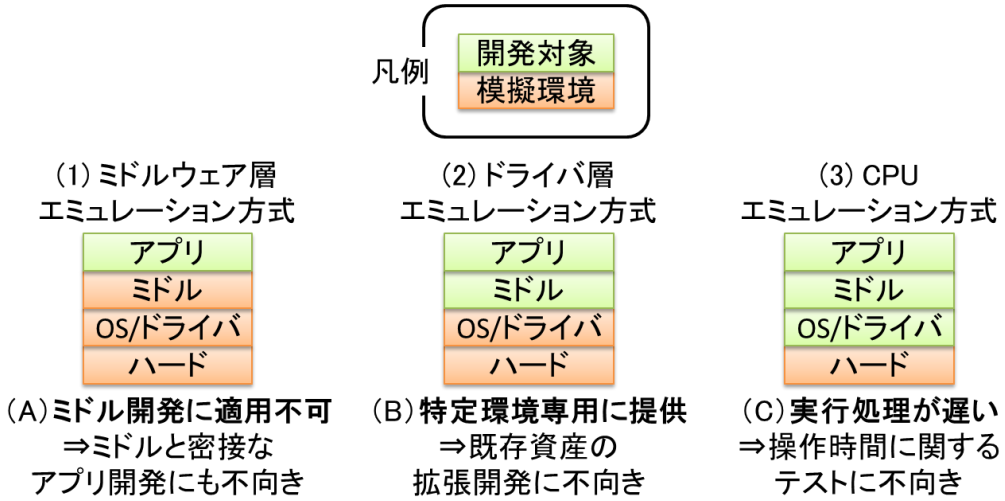


図 4.1　従来の実機レス開発方式とその問題点

(1) ミドルウェア層エミュレーション方式

アプリケーション プラットフォームである BREW[BRE]や brewmp[BRM]では，アプリケーション開発用にミドルウェア層エミュレーションが用いられている．しかし，実機の機能をミドルウェア層でエミュレートするため，ミドルウェア層の開発には適用できない．このため，ミドルウェア層と密接に絡んだアプリケーション層の開発にも適用困難である．

(2) ドライバ層エミュレーション方式

Android[AND][Saha2008]などのプラットフォームでは，ドライバ層エミュレーション方式がプラットフォーム設計時から考慮され，プラットフォームと共に提供されている．しかし，このエミュレーション機能を利用するためには，当該プラットフォームの利用が前提となるため，過去に開発済みのソフトウェア既存資産を用いた拡張開発に適用する場合，プラットフォーム間でのソフトウェア移植作業が発生し，工数がかかるという問題がある．

(3) CPU エミュレーション方式

CPU エミュレータ[Bellard2005][David2008][Matsuda2011]は，CPU の動作をエミュレーションすることでミドルウェア層の実機レス開発を実現する．しかし，CPU エミュレータは，CPU の動作を逐一置き換えるため，実行処理が遅くなる．このため，ユーザのリモコン操作後に一定時間が経過した時の機器の反応テストなど，時間に関わるテストには適用困難である．

4.2.2 考案した実機レス開発方式とその効果

上記問題点を解決するため，本研究で考案した実機レス開発方式では，(1)実機のドライバ層およびハードウェア層と同等の機能を，ドライバ層エミュレータおよび汎用ハードウェア（PC）を用いて実現させ，かつ，(2)ドライバ層エミュレータを，基本機能エミュレータとその上に被せた実機ドライバ I/F からなる構成とした．本方式の構成を図 4.2 に示す．

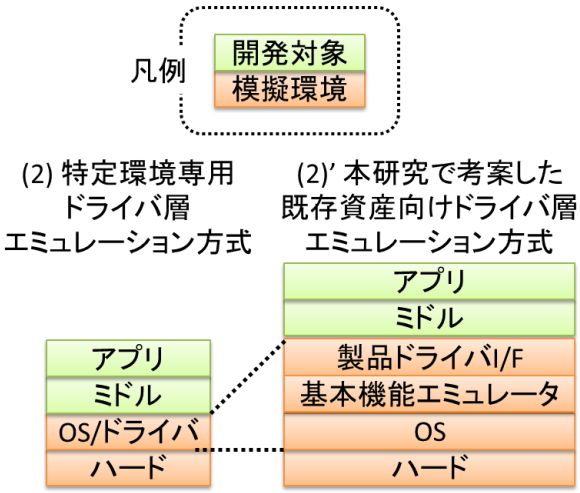


図 4.2　考案した実機レス開発方式の構成

これにより，ドライバ層でエミュレートするためミドルウェア層とアプリケーション層のテスト・デバッグが行える．また，ハードウェア層のレベルで詳細にエミュレートしないため，オーバーヘッドが小さく処理遅延が少なくなり，処理時間に関するテストに適用できる．また，基本のエミュレーション機能と独立してドライバ I/F を容易に変更可能なため，既存資産の拡張開発が可能となる．

本研究では，実機レス開発方式の要件を以下のように４件定めた．

(1) 実機を使わずに PC 上でアプリケーションとミドルウェアの動作確認が可能であること

(a) PC のディスプレイに実機の表示画面，ソフトウェア動作確認用のログ出力，及びソースデバッガを表示

(b) リモコン画像とマウス操作，またはキーボード入力によりリモコンまたはボタン類を操作

(c) キーボード入力により，実機での記録媒体の挿抜を擬似的に実現

実機の応答（表示画面とログ出力）とソースデバッガが一つのディスプレイに統合表示されることで，効率よいテスト・デバッグが可能となる．また，実機におけるリモコン本体ボタン類，記録媒体の操作を，PC 付属のデバイス（マウスまたはキーボード）で実現することで，テスト・デバッグ時の入力が容易となる．

(2) PC は，従来から開発に利用している Windows/x86 環境をそのまま活用できること

ソフトウェア開発において，仕様書表示やソースコード編集などに用いるために各開発者に配備されている PC の OS は，Windows である場合が多い．実機レス開発環境を構築する際には，この Windows PC をそのまま活用した方が，実機レス開発環境とその他の開発環境の統合や，担当者の習熟期間などの面で効率的である．

(3) 既存資産を流用した拡張開発が可能であること

(4) リモコン操作時間に関するテストが可能であること

4.3 既存資産のドライバ層をエミュレートする実機レス開発方式

4.2 で示した要件を満たすよう本研究で考案した，実機レス開発方式の特徴を以下に示す．

4.3.1 Windows と Linux のマルチ OS 環境

Windows と Linux のマルチ OS 環境による実機レス開発環境の構成を図 4.3 に示す．

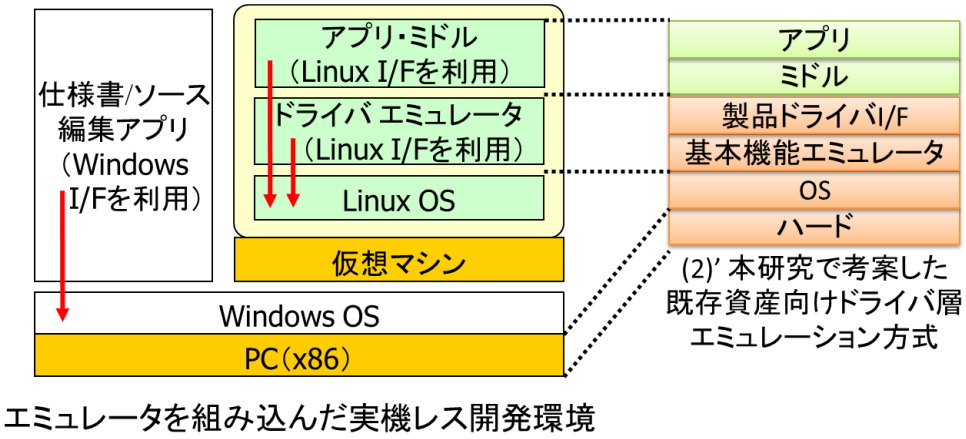


図 4.3　Windows と Linux のマルチ OS 環境による実機レス開発環境の構成

ドライバ エミュレータは，実機のアプリケーション・ミドルウェア層のプログラムを汎用ハードウェア層（PC）の上で動かすため，実機ドライバのエミュレーション機能を PC 上で提供する環境である．

組込み機器では，高度なネットワーク機能やオープンソースとの親和性などから Linux を採用する場合が増えている．このため，エミュレータも OS として Linux を用いれば，実機とエミュレータの OS 部分が共通化できるため，効率的に開発でき，かつ，より正確に動作をエミュレートすることができる．一方，4.2 で示した通り，PC は，従来から開発に利用している Windows/x86 環境をそのまま活用できることが求められる．

そこで，本研究の実機レス開発環境では，Windows PC 上で Linux 実行環境を提供可能な仮想マシン（VMware Player）を利用した．仮想マシンは，Windows アプリケーションとして Windows OS 上で動作しつつ，内部では Linux 環境を提供することができる．これにより，Windows アプリケーションでソースコード編集などを行なった後，クリック 1 つで仮想マシンに切り替えて，仮想マシンの Linux OS 上でプログラム生成や動作確認を行なうことができる．

4.3.2 本研究で考案したエミュレータの実装方式

本研究で考案したエミュレータの実装方式について，図 4.4 に示す．図は，下から汎用ハードウェア層（PC），OS 層（Linux カーネル，Linux ライブラリ，Linux デバイス ドライバ，ファイル システム），ドライバ エミュレータ層，開発対象であるアプリケーション・ミドルウェア層を示している．前述のように，本研究の実機レス開発環境では OS 層において，Linux OS は Windows OS 上で動作する仮想マシン（VMwarePlayer）の中で動作している．なお，Linux OS に含まれる Linux カーネルと Linux デバイス ドライバは図から省略している．

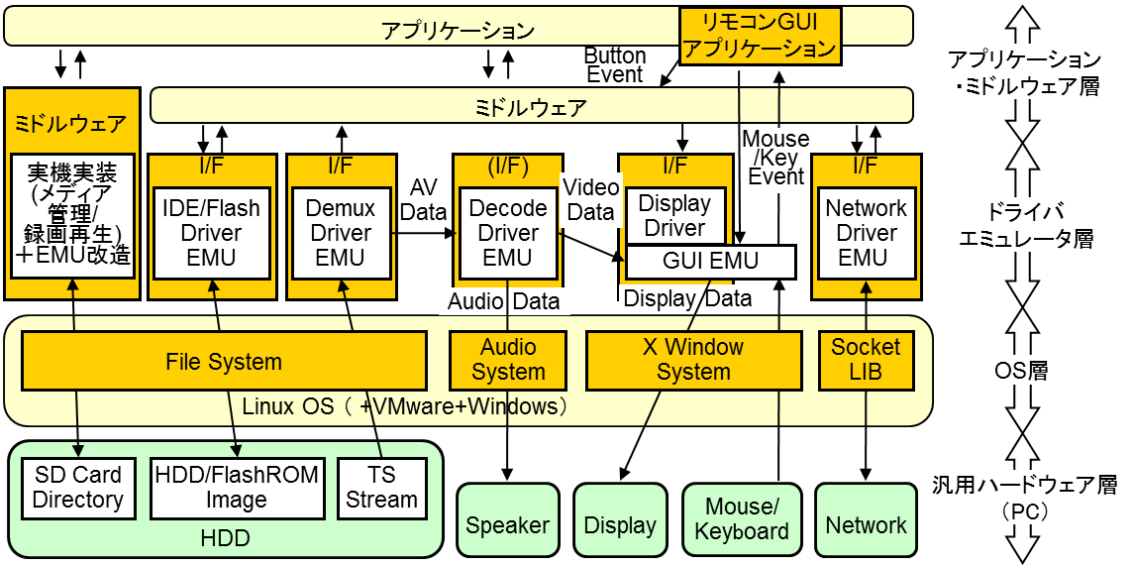


図 4.4　ドライバ エミュレータの実装方式

本研究で考案した各ドライバ エミュレータは，ミドルウェアに対して実機ドライバと同じ I/F（API）を提供する．内部では基本機能エミュレータとして固有のエミュレーション処理が実装してあり，これが下層の Linux ライブラリを呼び出す．実機用のソースコードから流用したドライバ I/F と，新たに開発する内部のエミュレーション処理の組み合わせにより，ドライバ エミュレータとして動作させる．以下では，エミュレーション処理，Linux OS，PC ハードウェアの関係について，エミュレーション機能の実装方式ごとに代表例を挙げて述べる．

(1) Linux ライブラリ利用方式

例えばネットワーク ドライバ エミュレータ（以降，EMU と呼ぶ）は，ミドルウェアから通信設定用 I/F（API）を呼ばれると，同等機能の Linux ソケット ライブラリ通信設定用 API を呼び出し，PC のネットワークデバイスを介して通信する．

本研究の実機レス開発環境では，Linux OS と PC ハードウェアの間に仮想マシンと Windows OS が介在するが，仮想マシン上の Linux 環境は，Windows が存在せず Linux だけが OS として動く Linux PC と同じ環境である．このため以降では，仮想マシンと Windows OS の説明を省略する．

(2) Linux ファイルシステム利用方式

IDE（Integrated Drive Electronics）ドライバ（EMU）や Flash ドライバ（EMU）のようにデータストレージ（HDD や FlashROM）を読み書きするドライバ（EMU）の場合は，ストレージをイメージ ファイルとして PC の HDD 上に置き，Linux のファイルシステムを用いて管理する．

例えば実機の Flash ドライバが提供する FlashROM の読み書き機能を，PC の FlashROM を用いてエミュレートすると，PC のシステムデータが書き換えられて PC が動作不良を起こす可能性がある．そこで本研究で考案したエミュレータは，PC の HDD 上に実機の FlashROM イメージファイルを作成して利用する．Linux では，FlashROM などのハードウェアに対する読み書き API と，ファイル システムに対する読み書き API が共通化されているため，実機の Flash ドライバ内のシーケンスを変更することなく，エミュレーションを実現できる．

(3) Linux 入出力システム利用および専用アプリケーション作成方式

実機では，高速処理用にハードウェアで実装した図形描画機能を表示ドライバが呼び出して利用する一方，PC のハードウェアには図形描画機能が備わっていない場合が多い．このため本研究で考案した GUI エミュレータは，同等の図形描画機能をソフトウェアで実装し，提供する．

また，実機では，ユーザの操作端末として赤外線リモコンが用いられる場合がある．例えば，実機でリモコンのボタンが押されると，ハードウェアの赤外線受光部がリモコンからの信号を受けて，実機のリモコンドライバが動作する．しかし，PC のハードウェアには赤外線受光機能が備わっていないため，本研究の実機レス開発環境では，PC のマウスまたはキーボードでリモコン操作を模擬する．

具体的には，Linux の X Window System（以降，X と呼ぶ）という GUI 環境構築用の入出力システムを用いて，ディスプレイ，マウス，キーボードなどの GUI を統合管理する GUI エミュレータを開発する．表示ドライバ（EMU）とリモコン GUI アプリケーションは，GUI エミュレータを呼び出して実機の表示画像とリモコン画像を表示する．また，リモコン画像のボタン部分をマウスで操作するかキーボードでキーを押すと，GUI エミュレータが, X 経由で受けたマウス/キーボードのイベントをリモコン GUI アプリケーションに転送し，リモコン GUI アプリケーションが，ボタン部分を押された状態の画像に描画し直して，ボタンイベントをイベント管理ミドルウェアに転送する．

また，実機の表示機能では，静止画像と動画像（ビデオ データ）で制御するハードウェアが異なるためドライバも異なる場合が多い．しかし，PC では表示ハードウェアが１つであるため，両方とも GUI エミュレータが表示する．

(4) ミドルウェア層エミュレーション方式

本研究で考案した実機レス開発環境では，基本的にはドライバ層でエミュレートし，アプリケーション・ミドルウェア層の実機ソースコードをそのまま使えるように設計している．しかし，製品開発適用時のテスト・デバッグ効率の向上が図れることから，SD カードなどの記録媒体の読み書き機能については，ミドルウェア層でエミュレートする．

例えば実機では，記録媒体の挿抜を検出してマウント状態を管理するメディア管理ミドルウェアが，Linux OS が提供する Linux 記録媒体ドライバの読み書き API を呼び出す．一方エミュレータでは，Linux が管理するファイル システム上の特定ディレクトリを，マウントした記録媒体用ディレクトリに見せかけるようメディア管理ミドルウェアを修正する．これにより，PC 上でファイルをコピーするだけで，記録媒体に様々なデータを入れた状態をテストできる．

また，メディア管理ミドルウェアを修正してマウント状態を擬似的に管理する機能を実装し，キーボードから擬似的にマウント状態を変更してやることで，記録媒体の挿抜テストを簡単にエミュレートできるようにする．これにより，PC の記録媒体用ハードウェアを用いたドライバ層のエミュレータを実装し，各種データが入った記録媒体を用意して PC に記録媒体を挿抜してテストする場合よりも，効率良くテスト・デバッグを行なうことができる．

4.3.3 エミュレータの開発手順

本研究で考案したドライバ エミュレータは，処理遅延を抑えつつ，アプリケーション・ミドルウェア層のテスト・デバッグを PC 上で実行可能とするものである．このため，デバイスの温度変化によるドライバ動作の変化や，スレッド切り替えタイミングの変化など，システムの個体毎に異なるような厳密な動きはエミュレーションの対象外とした．これにより，ドライバ エミュレータの開発を容易化できるだけでなく，基本機能エミュレータの部分を，TV，レコーダ，カーナビゲーションシステム，ビデオカメラなどの多様な製品へ転用し易くなる．本研究で考案したドライバ エミュレータの開発手順を図 4.5 に示す．これは 4.3.2 で述べた実装方式の内，「(3)の専用アプリケーションと(4)のミドルウェア層エミュレータ」以外に適用できる．

まず，(1)実機ドライバの機能仕様書を元にドライバの機能テストを作成する．次に，(2)作成したテストを実機のドライバ上で動かして，テストの動作を確認し，必要なら合格するようにテストを修正する．

次に，(3)実機ドライバの機能仕様書を元にドライバ エミュレータを開発し，(4)開発したドライバ エミュレータの単体機能を検証する．ここでは，(2)で確認したテストを用いて実機ドライバと同様のテスト結果になることを確認する．(3)は，(1)，(2)と並行開発してもよい．

最後に，(5)アプリケーションとミドルウェアの実機依存部を修正（＝PC 対応 : 後述）して(4)で開発したドライバ エミュレータ上で動作するようにし，(6)アプリケーションとミドルウェアが実機システムと同じ動作をすることを確認して，システム機能を検証する．必要なら同じ動作になるようにアプリケーションとミドルウェアを修正する．

この開発手順により，仕様書と最新（実装）の仕様が一致しない場合でも，実機開発環境と実機レス開発環境で同じ動作を確認できるようになる．

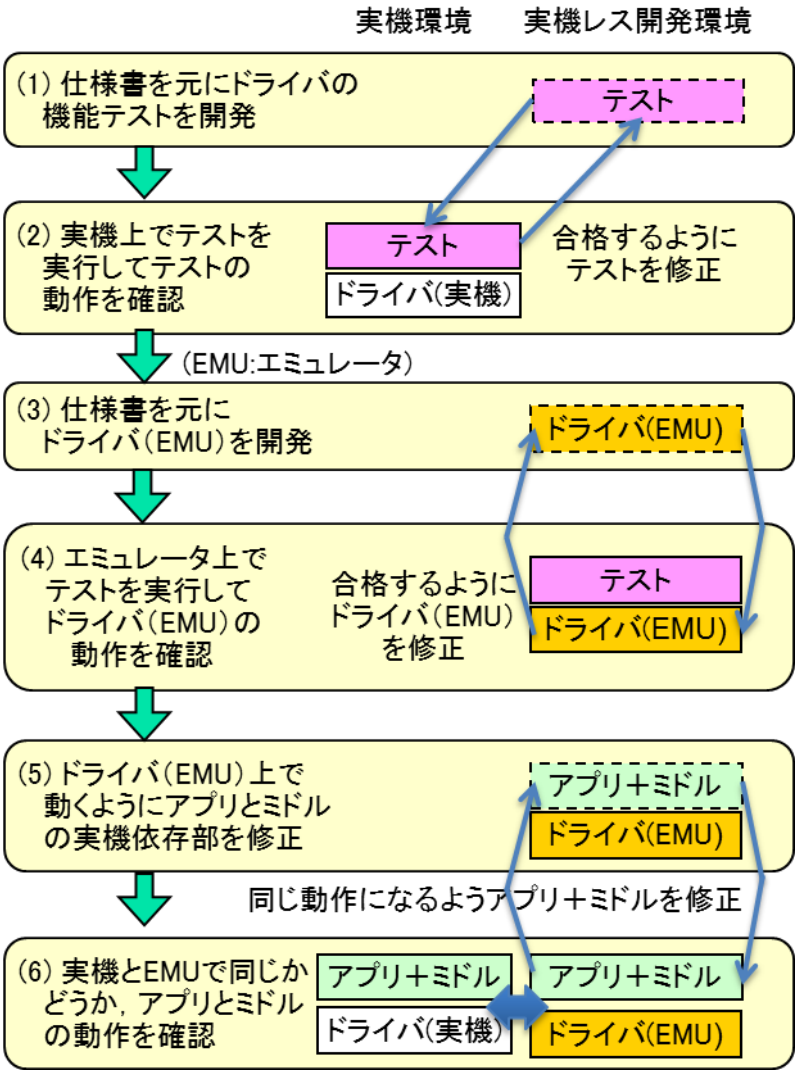


図 4.5　ドライバ エミュレータの開発手順

4.4 デジタル TV 向け実機レス開発環境の開発

4.3 で示した実機レス開発方式に従い，デジタル TV 製品のソフトウェア開発用に実機レス開発環境を構築した．

4.4.1 デジタル TV 向け実機レス開発環境の概要

デジタル TV（以降，DTV と呼ぶ）向けの実機レス開発環境を Windows PC 上に構築した例を図 4.6 に示す．図では，右前面にリモコンと TV 画面，左背面にソースデバッガのソースコード，ログ，変数の値などが表示されている．マウスやキーボードでリモコンを操作し，その反応を TV 画面とログ画面で確認する．また，ソースデバッガを操作し，ソースコードの任意の場所でプログラムを止めて，変数の値を確認しながら 1 行ずつプログラムを動作させることなどができる．図には示していないが，同じ PC（Windows）

上で，従来通り仕様書を表示したり，使い慣れたテキスト編集ツールでソースコードを編集することも可能である．DTV 向け実機レス開発環境で動作確認可能な主な TV 機能を以下に示す．

・TV 放送（表示，選局，各種設定）
・番組情報（番組表，番組検索，録画予約など）
・ネットワーク（映像機器との連携）
・HDD（録画，一覧表示，再生），SD カード（写真表示）

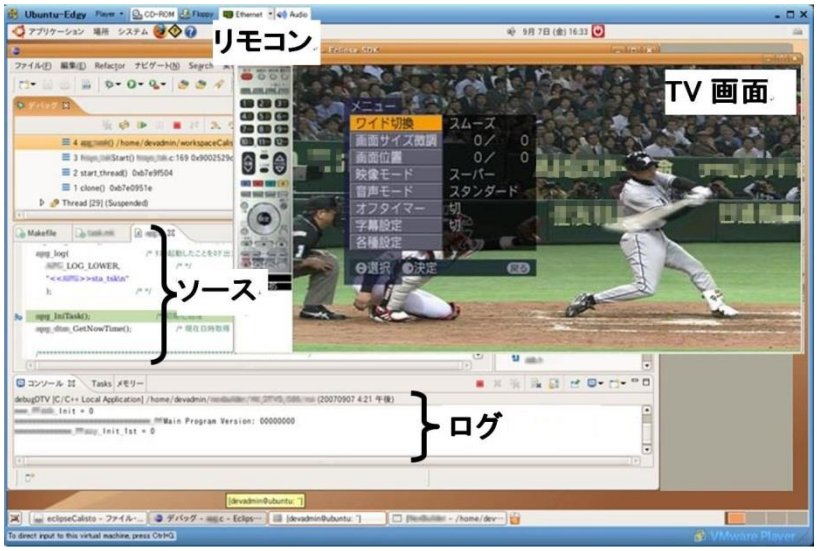


図 4.6　実機レス開発環境の画面例

4.4.2 アプリケーションとミドルウェア内の実機依存処理の対策

本研究で考案した実機レス開発方式では，ハードウェアとドライバをエミュレータ実装し，その上でアプリケーションとミドルウェアを開発できるようにした．実機レス開発環境のハードウェア（PC）と実機のハードウェア（組込み機器）の違いは基本的にエミュレータが吸収するが，開発対象であるアプリケーションやミドルウェアに実機依存処理が含まれている場合，開発対象にも対策が必要となる．これは，実機のハードウェアとドライバ，また PC とエミュレータを，それぞれ開発対象のプラットフォーム（PF）と考えると，第 3 章で述べた PF 間移植における PF 依存部の修正に相当する．

本研究の実機レス開発方式を DTV に適用した際の，主な実機依存項目，その洗い出し方法と対策を表 4.1 に示す．

表 4.1　実機依存処理の洗い出し方法とその対策

#	実機依存内容	洗い出し方法	対策（ソースコード修正）
(1)	固定アドレス利用	8 桁の 16 進数値を検索（マスク等は目視で除外）	アドレス管理モジュールでエミュレータ対応
			レジスタ アクセス等の処理を個別に削除
(2)	エンディアン依存	シフト演算，2 バイトコードを検索	両方（ビッグ・リトル）の場合を記述し#define でビルド時に切替
(3)	アセンブラコード	ビルドエラーを参照	実機用と PC 用のアセンブラ実装ファイルをビルド時に Makefile で切替

(1) 固定アドレス利用

実機ではメモリアドレスは固定の設定だが, エミュレータではメモリアドレスが動的に設定され, 起動の度にアドレスが変化する. このため, アドレス管理モジュールのソースコード中で固定アドレスを用いている部分を，malloc を利用した動的なアドレス指定方法に変更した.

また，レジスタやバスへのアクセス処理にも固定アドレスが利用されていたが，これらの処理は実機依存であり実機レス開発環境では利用できない. システム起動時のこれらのアクセス処理はハードウェアの初期化指示であり, 実機レス開発環境上では不要であるため，ソースコードから削除した．また，レジスタを一時的なデータ保存場所として利用している場合は，エミュレータ内に代わりのデータ保存場所を確保した.

固定アドレスを利用している箇所は，8 桁の 16 進数値をソースコードから検索して洗い出した．この時，マスク処理なども該当してしまうが，目視で判断して除外した.

(2) エンディアン依存

実機環境と実機レス開発環境とでエンディアンが異なるため, 実機用ソースコード中のエンディアン依存の部分に, 実機レス開発環境用エンディアンに対応させた実装を併記し，#define でビルド時に切り替えることで対応した．エンディアン依存処理は，シフト演算や 2 バイトコードをソースコードから検索して洗い出した.

(3) アセンブラコード

実機環境と実機レス開発環境では CPU が異なり，実機用のアセンブラコードはそのままでは実機レス開発環境で動かない．このため，実機と実機レス開発環境の両方の CPU 用のアセンブラ実装ファイルを用意し, Makefile でビルド時に切り替えて対応した. 対応していない CPU のアセンブラコードは, ビルド時にコンパイルエラーが出るため, エラーメッセージを参照して洗い出した.

4.5 デジタル TV 開発への適用評価

既存資産への拡張開発を行っている，実際の DTV 製品のソフトウェア開発に，4.4 で開発した DTV 向け実機レス開発環境を適用し，評価した.

4.5.1 製品開発での適用対象

DTV の約半数のアプリケーション開発における，Unit Test（以降，UT と呼ぶ），Combination Test（以降，CT と呼ぶ）およびデバッグに DTV 向け実機レス開発環境を適用した．アプリケーション（以降，APP と呼ぶ）全体に対する適用対象分の規模を表 4.2 に示す.

表 4.2　適用対象ソフトウェアの規模

	APP 全体		
		適用対象	適用比
モジュール数	68	33	49%
完成ステップ数	688k	187k	27%

4.5.2 適用手順

(1) UT

UT では, 関数単独での動作を確認するため, ミドルウェア層やドライバ層の関数と, アプリケーション層の関数の間での呼び出しは考慮しない. 各関数の入力や内部変数などの条件に応じて, アプリケーションの各関数内で正しい経路が実行され正しい値が出力されることを確認する．このため，実機レス開発環境を適用し，デバッグ用ツールまたはログ出力を用いて，経路や値の確認を行った.

(2) CT

アプリケーションの CT では，ユーザによるリモコン操作に応じた，アプリケーションと，それが呼び出すミドルウェアの組み合わせによる応答を確認する．このため，これらから呼び出されるドライバ API の機能がエミュレータで提供されているか否かにより，実機レス開発環境で CT を実行可能かどうかが決まる．本研究では，以下の手順で実機レス開発環境における制約を詳細に調査した後，実際のテストを行なった.

(a)テスト対象であるアプリケーションをエミュレータ上で実行し, アプリケーションの実機用チェックリストに従い，１項目ずつテスト（動作確認）の可否を調査

(b) (A)エミュレータ上でテストできる項目，(B)スタブ作成によりエミュレータ上でテストできる項目，(C)実機を使わないとテストできない項目に分類

(c) 必要なスタブを作成

(d) (A)(B)をエミュレータ上でテスト

(e) (C)を実機上でテスト（エミュレータが非対応の機能，TV 放送の受信が必要な機能，画質・音質調整など）

4.5.3 評価項目・評価方法

実機レス開発環境の適用評価について，評価項目と評価方法を表 4.3 に示す.

表 4.3　実機レス開発環境の評価項目と評価方法

#	評価項目	評価方法
(1)	適用可能テスト件数	実機レス開発環境のみで（実機がなくても）確認可能なテスト項目の件数を計上
(2)	テスト・デバッグ期間 （実機準備完了後の開発期間）	実機レス開発環境と実機の両方を用いた場合（実績値）と，実機レス開発環境の適用がない場合（実機のみの利用を仮定して算出）で，実機準備完了後の開発期間を比較
(3)	テスト・デバッグ効率 （工数当たりのテスト件数）	実機レス開発環境で実施したテストと，実機で実施したテストで，工数当たりのテスト件数を算出して比較
(4)	実機レス開発環境の構築工数	ドライバエミュレータの実装と，テストの実装及び動作確認（実機依存処理の対策を含む）に要した工数（開発全体比）を算出

4.5.4 評価結果

(1) 適用可能テスト件数

UT と CT のそれぞれについて，(a)今回の適用対象ソフトウェアでのテスト件数，(b)その内実際に適用可能であったテスト件数，及び(c)適用対象ソフトウェア（ソフト）のテストにおける適用比率（=b/a）を表 4.4 に示す.

表 4.4　実機レス開発環境の適用可能テスト件数

	(a)適用対象		
		(b)実際に適用可能分	(c)適用対象ソフトのテストにおける適用比率（=b/a）
UT 件数	12015	12015	100%
CT 件数	15174	6100	40%

(2) テスト・デバッグ期間

実機は，テスタとプログラマ全員で 1 台を共有し，実機レス開発環境は，1 人 1 台ずつ配備した．実機レス開発環境と実機での UT と CT の実績値から，実機レス開発環境を適用しない場合，すなわち実機 1 台のみでのテスト・デバッグ期間を，下記の手順で推定した．

(a) (A)テスト・デバッグ工数，(B)月当たりの作業時間，(D)テスト・デバッグ期間の実績値から，(C)月当たりのテスト・デバッグ工数を算出（C=(A/B)/D）．

(b) (A)の実績値を合計して，実機レス開発環境を適用しない場合の(A)を算出.

(c) 実機レス開発環境を適用しない場合の(B)と(C)は, 実機における(B)と(C)の実績値と同じだと推定して設定.

(d) 実機レス開発環境を適用しない場合の(A)，(B)，(C)から，同じく適用しない場合の(D)を算出（D=(A/B)/C）．

以上の結果を表 4.5 に示す．実機レス開発環境の導入により，実機を用いたテスト・デバッグの期間（実機準備完了後の開発期間）を 6.2 ヶ月間（58%）短縮することができたと考えられる．

表 4.5　テスト・デバッグ期間の短縮効果

		適用時の実績値				適用なしを仮定
		実機レス開発		実機開発		
	テスト環境	人数分の PC		1 台の実機		1 台の実機
	テスト分類	UT	CT	UT	CT	UT・CT
A	テスト・デバッグ工数（人・時間）	1100	800	0	1440	3340 (*2)
B	月当たりの作業時間（時間／月）	200	200	−	240	240 (*3)
C	月当たりのテスト･デバッグ工数＝実質作業者数（人）	1.3 (*1)	1.6 (*1)	−	1.3 (*1)	1.3 (*3)
D	テスト・デバッグ期間（ヶ月間）	4.2	2.5	0	4.5	10.7 (*4)
					適用効果	6.2 ヶ月間(58%)短縮

*1) 実績値 A, B, D より算出，*2) 実績値 A を合計して算出，
*3) 実機の実績値と同じだと推定して利用，*4) 実機レス適用なしの A, B, C より算出

(3) テスト・デバッグ効率

実機レス開発環境と実機開発環境で, UT と CT それぞれでのテスト･デバッグ効率（工数当たりのテスト件数）を算出して比較した．(A)テスト・デバッグ工数は，(2)テスト・デバッグ期間の評価に用いた実績値である. 同じく実績値である(E)テスト件数と合わせて(F)テスト・デバッグ効率を算出した（F=E/A）．ただし，UT では 100%に実機レス開発環境を適用したため，実機のテスト・デバッグ効率が求められないので，参考データとして，ソフトウェア開発全体でのテスト・デバッグ効率を表 4.6 に示す.

表 4.6　テスト・デバッグ効率の向上効果

		適用時の実績値			
		実機レス開発		実機開発	
	テスト環境	人数分の PC		1 台の実機	
	テスト分類	UT	CT	UT	CT
E	テスト件数（件）	12015	6100	0	8481
A	テスト・デバッグ工数（人・時間）	1100	800	0	1440
F	テスト・デバッグ効率（件／人・時間） ＝工数当たりのテスト件数	10.9 (*1)	7.6 (*1)	(10.2) (*2)	5.9 (*1)
適用効果（実機レス/実機）		(107%)	129%		

*1) 実績値 E, A より算出,

*2) 実際の開発データがないため，ソフトウェア開発全体での参考値

(4) 実機レス開発環境の構築工数

DTV ソフトウェア開発全体の工数を 100 とした場合の，実機レス開発環境の構築工数比を表 4.7 に示す．なお，テストの実装・動作確認は実機依存処理の対策を含む.

表 4.7　実機レス開発環境の構築工数比

DTV ソフトウェア開発全体	100
ドライバ エミュレータの実装	3.1
テストの実装・動作確認	5.0

4.5.5 考察

(1) 適用可能テスト件数

本適用評価では，UT の 100%，CT の 40%に実機レス開発環境を適用可能であった．UT において 100%適用可能となったのは，本適用対象がアプリケーションであり，CPU レジスタやバスなどの制約事項に影響されなかったためである．一方，CT は，スタブを作成することで非実装機能のテストを可能にしたが，実際の放送電波を受信しないとテストできないなどの制約の影響を受け適用率が 40%に留まった．リモコン操作時間に関するテストは可能であった.

(2) テスト・デバッグ期間

表 4.5 の結果から，実機レス開発環境がなく実機 1 台でテスト・デバッグを実施した場合は，実機レス開発環境を適用した場合に比べ，テスト・デバッグの終了が 6.2 ヶ月

間遅延すると考えることができる．実機レス開発環境の適用によりこの遅延を回避できたのは，実機準備が完了する前からテスト・デバッグの前倒し着手が可能になったことによる開発期間短縮の効果である．これにより，実機レス開発環境によるハードウェアとソフトウェアの並行開発の効果を確認することができた．

(3) テスト・デバッグ効率

実機レス開発環境で実施した CT は，実機開発環境で実施した CT に対して，テスト・デバッグ効率が 29%向上している．一方 UT の場合は，参考値との比較のため単純には評価できないが，107%とほぼ同等である．これは，テスト・デバッグ工数に占めるデバッグ工数の割合が UT では小さく CT では大きいため，実機レス開発環境の適用によるデバッグ効率向上の効果が，CT に顕著に現れていると考えられる．

(4) 開発費用

実機レス開発環境の適用による開発（テスト・デバッグ）での費用削減効果を概念的に図 4.7 に示す．横軸を開発期間 T，縦軸を並行開発可能な環境数 P とし，面積で開発工数または費用（人件費）W を表している．実機のみ利用時の開発期間を t1，実機レス開発環境利用時の開発期間を t2，実機数を p1，開発者数を p2 とする．並行開発可能な環境数 P は，実機レス開発環境を利用した開発では開発者数 p2 と同じだが，実機のみ利用した開発では実機数と同じ p1 に限られる．このため，実機のみ利用時の開発工数は W1（=t1×p1），実機レス開発環境利用時の開発工数は W2（=t2×p2）で表される．

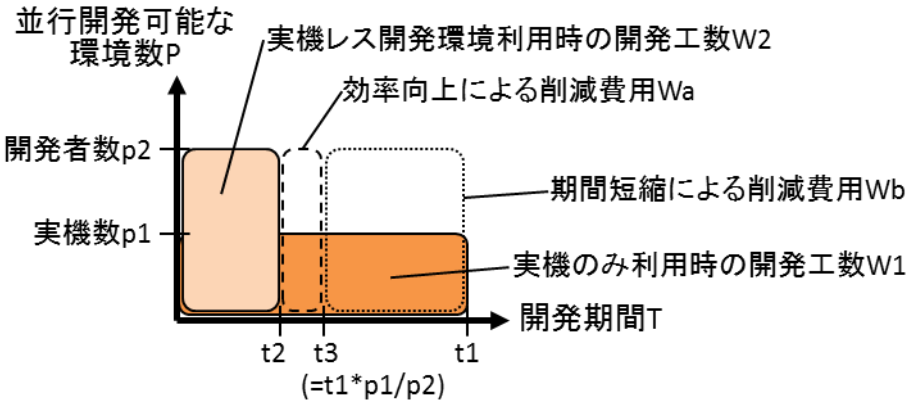


図 4.7　費用削減効果の概念

(a) テスト・デバッグ効率向上による費用削減

実機のみ利用した開発で，開発者数 p2 と同数の実機が用意できたと仮定すると，開発工数は W1 と同じなので，開発期間は t3（=W1/p2=t1*p1/p2）となる．実機レス開発

環境利用時の開発工数 W2 は，テスト・デバッグ効率向上分だけ小さくなり，その削減費用 Wa は，W1-W2=t1*p1-t2*p2=(t3-t2)*p2 と表される．

(b) テスト・デバッグ期間短縮による費用削減

実機のみ利用した開発では，並行開発可能な環境数が実機数と同じ p1 であっても，開発期間 t1 の間，開発者数 p2 分の人件費が発生している．一方，実機レス開発環境利用時の開発では，並行開発可能な環境数を開発者数 p2 分に増やせるため，(a)の効率向上効果を差し引いて考えると開発期間を t3 に短縮できる．その削減費用 Wb は，(t1-t3)*p2 と表される．

(c) 費用対効果

(a)および(b)による削減費用 Wa+Wb は，(t3-t2)*p2+(t1-t3)*p2=(t1-t2)*p2 と表され，本適用では DTV のソフトウェア開発全体の工数を 100 とした場合，11.9 であった．実機レス開発環境の構築工数は表 4.7 の通り同 8.1（=3.1+5.0）であるため，本適用によりソフトウェア開発全体の 3.8%（=11.9-8.1）の費用対効果が得られたと言える．

また，本研究で考案した実機レス開発環境は，基本機能エミュレータ部分を，同じ製品系列の次世代モデルや，他の類似製品へ容易に転用できるため，より少ない構築工数（費用）で同様の削減効果が得られることが期待できる．

4.6 まとめ

テスト・デバッグの期間短縮と効率向上を目的に，実機レス開発方式を考案し，既存資産の拡張開発を行うデジタル TV 向けのドライバ層エミュレータを開発した．これを用いた実機レス開発環境を構築し，ミドルウェアと組み合わせたユーザ操作アプリケーションの製品開発に適用した．実機のみの開発ではできなかった並行開発を実現し，テスト･デバッグの期間短縮と効率向上により，高い費用削減効果が得られることを実証した．今後は既存資産の拡張開発を行う他の Linux 搭載製品へも適用を図る．

第5章 むすび

5.1 まとめ

本研究では, 組込みソフトウェア開発における工数削減および期間短縮を目的として, (1)設計・実装工程における既存ソフトウェアのプラットフォーム（PF）間移植と，(2)テスト・デバッグ工程における並行開発の開発効率向上を図った.

(1) 設計・実装工程における既存ソフトウェアの PF 間移植

PF 間移植に必要となる，エンディアンやパディング有無などの PF 特性に依存する実装（以降，PF 依存部と呼ぶ）の抽出に関し，仕様書が不完全，不正確のため活用できないという状況に対応できるよう，PF 特性に依存する実装パターン（以降，PF 依存種と呼ぶ）検索によるソースコードからの PF 依存部抽出手法を考案し，本手法を用いた PF 依存部抽出支援ツールを開発した.

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 依存部を抽出するため，以下の効果が得られる.

(A) 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.

(B) 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.

(C) 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.

本研究では，PF 依存部抽出支援ツールを実際の製品ソースコードに適用した結果，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.

(2) テスト・デバッグ工程における並行開発

実際の製品版ハードウェア（実機）が不十分な開発の初期段階で，ソフトウェアのテスト・デバッグ環境を構築するに当たり，ミドルウェア層の開発，ユーザ操作による処理，既存資産の拡張開発に同時に対応できる，ドライバ層エミュレーションによる汎用 PC 上での実機レス開発方式について考案した.

本方式では，(A)実機のドライバ層およびハードウェア層と同等の機能を，ドライバ層エミュレータおよび汎用ハードウェア（PC）を用いて実現させ，かつ，(B)ドライバ層エミュレータを，基本機能エミュレータとその上に被せた実機ドライバ I/F からなる構成とした．

これにより，ドライバ層でエミュレートするためミドルウェア層とアプリケーション層のテスト・デバッグが行える．また，ハードウェア層のレベルで詳細にエミュレートしないため，オーバーヘッドが小さく処理遅延が少なくなり，処理時間に関するテストに適用できる．また，基本のエミュレーション機能と独立してドライバ I/F を容易に変更可能なため，既存資産の拡張開発が可能となる．

また，エミュレータ適用に向け必要となるアプリケーション修正については，ドライバ層およびハードウェア層を PF と考えると，実機 PF からエミュレータ適用 PF への PF 間移植に相当することに着目し，前述の PF 間移植の開発効率向上技術を適用した．

さらに，既存資産の拡張開発を行うデジタル TV 向けに，本方式を用いたドライバ層エミュレータを開発し，これを用いた実機レス開発環境を構築して，実機のみの開発ではできなかった並行開発を実現した．デジタル TV における，ミドルウェアと組み合わせたユーザ操作アプリケーションの実際の製品開発に適用した結果，テスト・デバッグ期間を 58%短縮し，組み合わせテスト時のテスト・デバッグ効率を 29%向上させる効果が得られ，高い費用削減効果が得られることを実証した．

5.2 今後の研究方針

本研究では，組込みソフトウェア開発における工数削減および期間短縮を目的として，PF 間移植と並行開発の開発効率向上を図った．今後も引き続き，組込みソフトウェアの開発効率向上を支援する開発手法の確立に取り組んでいく．特に，仕様書に頼れずソフトウェア構造が理想的でない既存ソフトウェアへの対応など，実際の開発現場の制約や課題を考慮した手法の確立と，実製品開発への適用評価，さらに複数製品への展開と手法の改善を進めていきたい．

本研究で考案した PF 依存部抽出手法を例に挙げると，手法適用の際に，これまで主に検討してきた PF 依存部の検索・判定工程だけでなく，工数の大きい修正工程の開発効率向上も実際の開発では重要となる．

修正工程では，手当たり次第にソースコードを修正するのではなく，各種 PF 依存部の数や修正方式毎の見積もり工数などを考慮して，修正方式や優先順を決定してから修正作業を行う．たとえば，もしエンディアン依存部が 1 箇所だけなら，依存部を直接修正してエンディアンを変換する方法が考えられるし，多数あるなら，エンディアン変換

関数を別途作成し，依存箇所でこれを呼び出すことも考えられる．しかし，本研究で考案した手法は，PF 依存部の数を算出するのには有用であるが，修正工数の見積もりには対応していないため，正確で効率的な見積もり方法の確立が課題である．

これには，複数プロジェクトでの手法の適用と評価が有効と考える．例えば，実際の製品開発における 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，見積もりの実現が期待できる．

このように，実際の製品開発データを蓄積，分析して手法を改善していく．

参考文献

[Abe2003] 安部田章, "多層化による組込みソフトウェアの移植性の向上," 情報処理学会研究報告 ソフトウェア工学(SE), 2003(22), pp.117-122, 2003.

[Abrial2005] J. R. Abrial, A. Hoare, "The B-book: assigning programs to meanings," Cambridge University Press, 2005.

[AND] Android, "Android Emulator," http://developer.android.com/tools/help/emulator.html, 2015.12.4 参照.

[Asada2014] 浅田幸則, 大條成人, 松本紀子, "既存ソフトウェアに対する実践的なモデル化方法およびデジタルテレビへの応用," 情報処理学会デジタルプラクティス= Journal of digital practices, 5(3), pp.231-238, 2014.

[Button1996] G. Button, W. Sharrock, "Project work: the organisation of collaborative design and development in software engineering," Computer Supported Cooperative Work (CSCW) 5.4, pp.369-386, 1996.

[Bellard2005] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," Proc. USENIX 2005 Annual Technical Conference, pp.41-46, 2005.

[BLA] "Blanco Framework," https://osdn.jp/projects/blancofw/, 2015.12.5 参照.

[Boehm1981] B. Boehm, "Software Engineering Economics," Englewood Cliffs, Prentice-Hall, 1981.

[BRE] Qualcomm Technologies, Inc., "BREW," https://www.qualcomm.com/products/brew, 2015.12.4 参照.

[BRM] Qualcomm Technologies, Inc., "Brew MP," https://developer.brewmp.com/home, 2015.12.4 参照.

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

[COQ] INRIA, "The Coq Proof Assistant," https://coq.inria.fr/, 2015.12.5 参照.

[CPC] Sourceforge, "Cppcheck," http://cppcheck.sourceforge.net/, 2015.12.4 参照.

[CPT] Techmatrix, "C++Test C/C++対応自動テストツール," http://www,techmatrix,co,jp/quality/ctest/, 2015.12.4 参照.

[David2008] F. M. David, E. M. Chan, J. C. Carlyle, R. H. Campbell, "QInject: A Virtual Machine based Fault Injection Framework," In International Conference on Architectural Support for Programming Languages and Operating Systems, 2008.

[ECL] The Eclipse Foundation, "Featured Eclipse Project," http://www.eclipse.org/, 2015.12.5 参照.

[Frankel2003] S. Frankel, "MDA モデル駆動アーキテクチャ," 日本アイ・ビー・エム TEC-J MDA 分科会, エスアイビー・アクセス, 2003.

[Fewster1999] M. Fewster, D. Graham, "Software test automation: effective use of test execution tools," ACM Press/Addison-Wesley Publishing, 1999 (テスト自動化研究会 訳, "システムテスト自動化 標準ガイド," 翔泳社, 2014).

[Fowler1999] M. Fowler, "Refactoring: Improving the Design of Existing Code," Addison-Wesley, 1999, (児玉公信, 友野晶夫, 平沢章, 梅沢真史 訳, "リファクタリング：プログラムの体質改善テクニック," ピアソン・エデュケーション, 2000).

[France2007] R. France, B. Rumpe, "Model-driven development of complex software: A research roadmap," in 2007 Future of Software Engineering, IEEE Computer Society, pp.37-54, 2007.

[FTR] Linux Foundation, "ftrace - Function Tracer," https://www.kernel.org/doc/Documentation/trace/ftrace.txt, 2015.12.5 参照.

[Fujiwara2014] 藤原貴之, 岡本周之, "大規模組込み機器向けテスト自動化拡大方式," 情報処理学会論文誌コンシューマ・デバイス＆システム(CDS), Vol.4, No.4, pp.1-10, 2014.

[Gomi2015] 五味弘, "現場で使うためのオールペア法, 直交表の基本," atmarkIT, http://www.atmarkit.co.jp/ait/kw/ait_allpairtest.html, 2015.12.5 参照.

[Goto2010] 後藤隼弐, 本田晋也, 長尾卓哉, 高田広章, "トレースログ可視化ツール," コンピュータ ソフトウェア, 27(4), pp.8-23, 2010.

[GPR] "GNU gprof," https://sourceware.org/binutils/docs/gprof/, 2015.12.5 参照.

[Harashima2012] 原嶋秀次, 蔭山佳輝, 河込和宏, "仮想化技術による実機レステスト環境の構築," 東芝レビュー, Vol.67, No.8, pp.31-34, 2012.

[Hatano2003] 秦野克彦, 乃村能成, 谷口秀夫, 牛島和夫, "ソフトウェアメトリクスを利用したリファクタリングの自動化支援機構," 情報処理学会論文誌, 44(6), pp.1548-1557, 2003.

[Higo2008] 肥後芳樹, 楠本真二, 井上克郎, "コードクローン検出とその関連技術," 電子情報通信学会論文誌 D, 91(6), pp.1465-1481, 2008.

[Higo2011] 肥後芳樹, 吉田則裕, "コードクローンを対象としたリファクタリング," コンピュータ ソフトウェア, 28(4), pp.43-56, 2011.

[Holzmann1997] G. J. Holzmann, "The Model Checker Spin," IEEE Transaction on Software Engineering, Vol.23, No.5, pp.279-295, 1997.

[Holzmann2004] G. J. Holzmann, "The SPIN Model Checker - Primer and Reference Manual," Addison-Wesley, 2004.

[HTM] WHATWG, "Welcome to the WHATWG community," https://whatwg.org/, 2015.12.6 参照.

[IBM2008] 日本 IBM, "モデル駆動型開発手法によりソフトウェアの効率化と品質の向上を実現," IBM PROVISION, No. 57, pp.18-25, 2008.

[Ichii2015] 市井誠, 小川秀人, "モデル変換を用いたリファクタリング検証手法," コンピュータ ソフトウェア, 32(3), pp.70-76, 2015.

[Ikeda2007] 池田義雄, "フロントローディングによる上流設計力強化," 東芝レビュー, Vol.62, No.9, pp.2-8, 2007.

[Iijima2008] 飯島三朗, 桜庭恒一郎, "組込みソフト開発における課題と対応策," ZIPC WATCHERS Vol.12-10, www.zipc.com/instance/files/vol12/Vol12-10.pdf, 2015.12.6 参照.

[Imai2011] 今井敬吾, 今井宜洋, 小笠原啓, "対話的定理証明支援系によるミドルウェア検証," 第 9 回クリティカルソフトウェアワークショップ, 2011.

[Inoue2005] 井上栄, "携帯電話用ソフトウェアプラットフォーム," 東芝レビュー, Vol.60, No.9, pp.16-20, 2005.

[IPA2007] IPA/SEC, "【改訂版】組込みソフトウェア向け 開発プロセスガイド ESPR," IPA, 2007.

[IPA2012] IPA/SEC, "組込みソフトウェア向け 設計ガイド ESDR[事例編]," IPA, 2012.

[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," IPA, 2013.

[Ishikawa2011] 石川冬樹, 荒木啓二郎監修, "VDM++による形式仕様記述（トップエスイー実践講座）," 近代科学社, 2011.

[ISO1991] International Organization for Standardization (ISO), "ISO/IEC 9126:1991: Software engineering -- Product quality," ISO, 1991.

[ISO1994] International Organization for Standardization (ISO), "ISO 8402:1994: Quality Management and Quality Assurance-Vocabulary," ISO, 1994.

[ISO2011] International Organization for Standardization (ISO), "ISO/IEC 25010:2011: Quality Management and Quality Assurance-Vocabulary," ISO, 2011.

[ISO2012] International Organization for Standardization (ISO), "ISO/IEC 19505:2012: Information technology -- Object Management Group Unified Modeling Language (OMG UML)," ISO, 2012.

[ITR] 名古屋大学大学院 情報科学研究科 情報システム学専攻, ERTL, "μITRON4.0 仕様," http://www.ertl.jp/ITRON/SPEC/mitron4-j.html, 2015.12.6 参照.

[Jackson2002] D. Jackson, "Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM), 11(2), pp.256-290, 2002.

[JAV] ORACLE, "Java.com: あなたと Java," https://www.java.com/ja/, 2015.12.6 参照.

[JIS1994] 日本工業標準調査会, "JIS X 0129:1994: ソフトウェア製品の品質," 日本規格協会, 1994.

[JIS2013] 日本工業標準調査会, "JIS X 25010:2013: システム及びソフトウェア製品の品質要求及び評価（SQuaRE）－システム及びソフトウェア品質モデル," 日本規格協会, 2013.

[Jones2011] M. T. Jones, "組み込みシステムの仮想化: ハイパーバイザーが小型機器に組み込まれるに至った経緯と理由," IBM developerWorks 日本語版,

https://www.ibm.com/developerworks/jp/linux/library/l-embedded-virtualization/, 2015.12.5 参照.

[Jyaku2005] 崔銀惠, 河本貴則, 渡邊宏, "画面遷移仕様のモデル検査," コンピュータ ソフトウェア, 22(3), pp.146-153, 2005.

[Kadota2003] 門田暁人, 佐藤慎一, 神谷年洋, 松本健一, "コードクローンに基づくレガシーソフトウェアの品質の分析," 情報処理学会論文誌, 44(8), pp.2178-2188, 2003.

[Kamiya2002] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," Software Engineering, IEEE Transactions on, 28(7), pp.654-670, 2002.

[Kamiyama2010] 神山達哉, 添田隆弘, 兪明連, 横山孝典, "組み込み制御ソフトウェア開発のための Simulink・UML モデル変換ツール," 情報処理学会 研究報告組込みシステム(EMB), 2010(7), pp.1-7, 2010.

[Kanbara2010] 神原典子, "ソフトウェアユーザエクスペリエンス設計," 日経 BP 社, pp.8-22, 2010.

[Kawakami2011] 川上真澄, 小川秀人, "デジタル家電ドメインに特化したモデルベース開発環境," 情報処理学会論文誌, Vol.52, No.12, pp.3184-3191, 2011.

[Kishi2013] 岸知二, 他 10 名, "ソフトウェアプロダクトライン国際会議 (SPLC2013) 参加報告," 電子情報通信学会技術研究報告, ソフトウェアサイエンス(SS), 113(269), pp.209-211, 2013.

[Kudo2003] 工藤裕, 平井千秋, 川辺博史, 降旗由香理, 大野治, "議事録を利用した設計レビュー管理システムの開発と評価," 情報処理学会論文誌, 44(5), pp.1404-1412, 2003.

[Kurosu2012] 黒須正明, "ユーザエクスペリエンスにおける感性情報処理," 放送大学研究年報, No.30, pp.93-109, 2012.

[Kuruma2007] 来間啓伸, 中島震監修, "B メソッドによる形式仕様記述－ソフトウェアシステムのモデル化とその検証（トップエスイー実践講座）," 近代科学社, 2007.

[Langlois2005] B. Langlois, J. Barata, D.Exertier, "Improving MDD Productivity with Software Factories," International Workshop on Software Factories, San Diego, California, USA, 2005.

[LF2013] Linux Foundation, "オープンソースソフトウェア活用動向調査," Linux Foundation, http://www.linuxfoundation.jp/content/osssurvey, 2013, 2015.12.4 参照.

[LF2015] Linux Foundation, "OSS Finder," http://ossfinder.linuxfoundation.jp/, 2015.12.4 参照.

[LIN] Linux Foundation, "Linux.com: 日本のリナックス/OSS 情報ポータル," https://jp.linux.com/, 2015.12.6 参照.

[Lehnert2011] S. Lehnert, "A review of software change impact analysis," Technische Universitat Ilmenau, 2011, http://www.db-thueringen.de/servlets/DerivateServlet/Derivate-24546/ilm1-2011200618.pdf, 2015.12.7 参照.

[Manalo2010] R. G. Manalo, M. V. Manalo, "Quality, Cost and Delivery performance indicators and Activity-Based Costing," Management of Innovation and Technology (ICMIT), 2010 IEEE International Conference, pp.869-874, 2010.

[Maruyama2002] 丸山勝久, "基本ブロックスライシングを用いたメソッド抽出リファクタリング (<特集> オブジェクト指向技術)," 情報処理学会論文誌, 43(6), pp.1625-1637, 2002.

[MSL] MathWorks, "Simulink," http://in.mathworks.com/products/simulink/, 2015.12.5 参照.

[Matsuda2011] 松田稔彦, 北村俊明, "計算精度低下を検出する PC エミュレータの開発," 情報処理学会研究報告, Vol.2011-HPC-132, No.24, pp.2-3, 2011.

[MCBOK2009] 西原秀明, 他, "MCBOK 2008: ソフトウェア開発のためのモデル検査知識体系, " 産業技術総合研究所システム検証研究センター, 2009.

[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-," 経済産業省, 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-," 経済産業省, 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," 経済産業省, 2011.

[MIC2015] 総務省, "平成 27 年版情報通信白書 ICT 白書：ICT の過去・現在・未来," 総務省, 2015.

[Miyauchi2009] 宮内孝, 小林大介, 藤田和明, "設備制御ソフトウェア開発の効率を向上させる実機レス デバッグシステムの適用," 東芝レビュー, Vol.64, No.5, pp.10-13, 2009.

[Mizuguchi2005] 水口大知, 渡邊宏, "システム検証の科学技術 組み込みソフトウェア開発におけるモデル検査の適用事例," コンピュータ ソフトウェア, 22(1), pp.77-90, 2005.

[MSR] The Motor Industry Software Reliability Association, "MISRA C," http://www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx, 2015.12.4 参照.

[Muramatsu2011] 村松孝倫, "レグザエンジン CEVO で操作感を向上させて液晶 TV 向け GUI アプリケーションー「高速レグザ番組表」と「レグザメニュー」ー," 東芝レビュー, Vol.66, No.11, 2011.

[Nakagawa2007] 中川雄一郎, 小川秀人, "企業における組込みソフト開発の課題," 組込みシステム技術に関するサマーワークショップ, 2007.

[Nakajima2001] 中島毅, 別所雄三, 山中弘, 広田和洋, "状態遷移モデルで記述された要求仕様に基づく組込みソフトウェアの自動試験法," 電子情報通信学会論文誌 D, 84(6), pp.682-692, 2001.

[Nakajima2001] 中島震, "オブジェクト指向デザインと形式検証手法（チュートリアル）," コンピュータ ソフトウェア, 18(5), pp.17-46, 2001.

[Nakajima2006] 中島震, "新しいソフトウェアの実現 モデル検査法のソフトウェアデザイン検証への応用," コンピュータ ソフトウェア, 23(2), pp.72-86, 2006.

[Nakajima2007] 中島震, "ソフトウェア工学の道具としての形式手法ー彷徨える形式手法ー," National Institute of Informatics (NII), NII Technical Report, 2007, (ソフトウェア工学の道具としての形式手法 ソフトウェアエンジニアリング最前線, 近代科学社, pp.27-48, 2007).

[Noda2009] 野田哲夫, 丹生晃隆, "オープンソース・ソフトウェアの開発モチベーションと労働時間に関する考察," 経済科学論集, 35, pp.71-93, 2009.

[Nonaka2010] 野中誠, "ソフトウェア品質の定量的管理における曖昧さ--ソフトウェア欠陥測定の原則," 経営論集, 76, pp.99-109, 2010.

[Northrop2006] L. M. Northrop, "Software Product Lines: Reuse That Makes Business Sense," IEEE Software Engineering Conference 2006 Australian (ASWEC2006), pp.1-3, 2006, (http://www.sei.cmu.edu/library/assets/ASWEC2006.pdf, 2015.12.4 参照).

[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.

[Offutt2008] J. Offutt, et al., "Programers ain't Mathematicians, and Neither are Testers," 10th International Conference on Formal Engineering Methods, LNCS5356, p.2, 2008.

[Ogawa2011] 小川秀人, "企業におけるソフトウェア開発とソフトウェア工学," コンピュータ ソフトウェア, 28(3), pp.2-11, 2011.

[Ohara2014] 大原貴都, 藤平達, 茂岡知彦, 新田泰広, 岩崎力, 杉山達也, "モデル駆動開発を支援するためのシミュレーション高速化に関する検討 (開発手法, 組込み技術とネットワークに関するワークショップ ETNET2014)," 電子情報通信学会技術研究報告, ディペンダブルコンピューティング(DC), 113(498), pp.25-30, 2014.

[Oho2009] 於保茂, 青野俊宏, 鈴木邦彦, "エンジン制御モデルベース開発の先進技術 (特集 環境, 安全, 快適を実現するオートモティブシステム開発技術)," 日立評論 91.10, pp.792-795, 2009.

[Ohta2009] 太田暁率, 進博正, 渡邊竜明, "高品質なソフトウェアを効率よく開発できるモデルベーステスト技術," 東芝レビュー, 64(8), pp.24-27, 2009.

[Omori2012A] 大盛善啓, 樋口靖和, 菊池匡晃, "家電機器への IP リモコン機能組込みを容易にするソフトウェアプラットフォーム," 東芝レビュー, 67(6), pp.24-27, 2012.

[Omori2012B], 大森隆行, 丸山勝久, 林晋平, 沢田篤史, "ソフトウェア進化研究の分類と動向," コンピュータ ソフトウェア, 29(3), pp.3_3-3_28, 2012.

[OPR] "OProfile," http://oprofile.sourceforge.net/news/, 2015.12.5 参照.

[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/, 2015.12.4 参照.

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

関翻訳, "ソフトウェアプロダクトラインエンジニアリング―ソフトウェア製品系列開発の基礎と概念から技法まで," エスアイビーアクセス, 2009).

[Raymond1999] E.S. Raymond, "The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary," O"Reilly Media, 1999, (山形訳, "伽藍とバザール：オープンソース・ソフト Linux マニフェスト," 光芒社, 1999).

[RPS] IBM, "Rational Rhapsody," http://www.ibm.com/software/products/ja/ratirhapfami, 2015.12.5 参照.

[Saha2008] A. Saha, "A Developer's First Look At Android," Linux For You, 2008(January), pp. 48-50, 2008.

[Sahara2007] 佐原伸, 荒木啓二郎, "先端ソフトウェアツール オブジェクト指向形式仕様記述言語 VDM++支援ツール VDMTools," コンピュータ ソフトウェア, 24(2), pp.14-20, 2007.

[Saito2007] 齊藤邦浩, "中国オフショア開発におけるコミュニケーション・マネジメント: オフショア開発成功の鍵 (< 特集> グローバル・プロジェクトマネジメント)," プロジェクトマネジメント学会誌, 9(1), pp.26-31, 2007.

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

[Shimamoto1995] 島本憲夫, 阿久津正幸, 中山好一郎, 黒川章, "ソフトプラットフォーム流用についての検討," 電子情報通信学会ソサイエティ大会講演論文集, 1995(2), 86, 1995.

[Shinkai2015] 新海良一, "上流工程（要件開発工程）から実践する QCD 向上施策," Prowise Business Forum in Tokyo 第 62 回, 日立ソリューションズ, 2015, http://www.hitachi-solutions.co.jp/forum/tokyo/vol62/pdf/pb_seminar62_3.pdf, p.24, 2015.12.4 参照.

[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).

[SPL] Software Engineering Institute, Carnegie Mellon University, "A Framework for Software Product Line Practice, Version 5.0," http://www.sei.cmu.edu/productlines/frame_report/index.html, 2015.12.4 参照.

[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.

[Szekely1996] P. Szekely, "Retrospective and challenges for model-based interface development ," Springer Vienna, pp.1-27, 1996.

[Takada2004] 高田広章, 岸田昌巳, 宿口雅弘, 南角茂樹, "リアルタイム OS と組み込み技術の基礎：実践μITRON プログラミング: 第 1 章 組み込みシステム概論," CQ 出版社, pp.7-16, 2004, http://www.cqpub.co.jp/hanbai/books/33/33281/33281_1syo.pdf, 2015.12.7 参照.

[Takahashi2010] 高橋知信, 南光孝彦, "組込み分野へのモデル駆動型開発手法の適用," Panasonic Technical Journal, Vol.56, pp.60-62, 2010.

[Takeuchi2007] 竹内真弓, 平山秀昭, 位野木万里, "ソリューションビジネスを成功へと導くプロセス技術とプロダクト技術," 東芝レビュー, Vol.62, No.9, pp.30-33, 2007.

[Tamaki2014] 玉置彰宏, "ソフトウェア工学の勧め: 第 5 章," http://www.tamakiseoffice.jp/software_engineering/Chap_05.pdf, 2014, 2015.12.4 参照.

[Tamaru2012] 田丸喜一郎, "組込みソフトウェア産業の現状と課題," http://sec.ipa.go.jp/events/2012/esec_02/events_esec_20120510-13.pdf, 2012, 2015.12.4 参照.

[Tamura2011] 田村雅成, 神山達哉, 添田隆弘, 兪明連, 横山孝典, "振る舞いモデル生成機能を持つ Simulink・UML モデル変換ツール", 電子情報通信学会技術研究報告 ソフトウェアサイエンス(SS), 110(458), pp.43-48, 2011.

[Tassey2002] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, RTI Project, 7007(011), 2002.

[TIS2015] TIS 株式会社, "平成２６年度我が国経済社会の情報化・サービス化に係る基盤技術 （クラウドコンピューティグ時代におけるオープンソースソフトウェアの活用に関する調査事業）調査報告書," 経済産業省, pp.58-59, 2015.

[Tsuchiya2007] 土屋達弘, 菊野亨, "ペアワイズテスト―ソフトウェアテストの効率化を求めて―," 電子情報通信学会論文誌 D, 90(10), pp.2663-2674, 2007.

[Tsuchiya2012] 土屋良介, 鷲崎弘宜, 深澤良彰, 加藤正恭, 川上真澄, 吉村健太郎, "派生プロダクト群における要求・実装間のトレーサビリティリンク抽出," 電子情報通信学会技術研究報告, ソフトウェアサイエンス(SS), 112(275), pp.123-128, 2012.

[Uchiya2015] 打矢隆司, 椿広計, 木野泰伸, "ソフトウェア計量管理のための QCD 統合構造モデル," 品質, 45(1), pp.98-115, 2015.

[Ueda2003] 植田泰士, 神谷年洋, 楠本真二, 井上克郎, "開発保守支援を目指したコードクローン分析環境," 電子情報通信学会論文誌 D, 86(12), pp.863-871, 2003.

[Uehara2010] 上原伸介, 小林隆志, 大須賀俊憲, 山本晋一郎, 阿草清滋, "多粒度な可視化によるソフトウェア理解支援," コンピュータ ソフトウェア, 27(2), pp.112-117, 2010.

[VAM] Andrei Voronkov, Krystof Hoder, "Vampire's Home Page," http://www.vprover.org/, 2015.12.5 参照.

[VMP] "VMware Player," https://www.vmware.com/products/player, 2015.12.4 参照.

[VMW] "VMware Workstation," http://www.vmware.com/products/workstation/, 2015.12.4 参照.

[VST] Microsft, "Visual Studio," https://www.microsoft.com/ja-jp/dev/, 2015.12.5 参照.

[VXW] Wind River, "VxWorks," http://windriver.com/products/vxworks/, 2015.12.6 参照.

[Ward2002] B. Ward, "The Book of VMware: The Complete Guide to VMware Workstation," No Starch Press, 2002.

[Watanabe2004] 渡辺政彦, "組み込みソフトウェア向け開発支援環境 (特集 モデリングとツールを駆使したこれからのソフトウェア開発技法ーモデル駆動開発手法を中心としてー)," 情報処理 45.1, 2000.

[Wiegers2004] K. E. Wiegers, 大久保雅一監訳, "ピアレビュー: 高品質ソフトウェア開発のために," 日経 BP 社, 2004.

[William2005] B. F. William, K. Kang, "Software Reuse Research: Status and Future," IEEE Transactions on Software Engineering, Vol.31, No.7, pp.529-536, 2005.

[WIN] Microsoft, "Windows," http://www.microsoft.com/ja-jp/windows, 2015.12.6 参照.

[Weiser1981] M. Weiser, "Program slicing," in Proceedings of the 5th International Conference on Software Engineering, IEEE Press, pp. 439-449, 1981.

[Yamada2005] 山田茂, 冨高功介, "ソフトウェア信頼性に影響を及ぼす品質工学的アプローチに基づく人的要因分析と信頼性予測 (< 特集>「ソフトウェア信頼性工学の新展開」)," 日本信頼性学会誌, 信頼性, 27(7), pp.439-448, 2005.

[Yamada2007] 山田暁広, "OSS を使用したインテグレーション作業の自動化," UNISYS TECHNOLOGY REVIEW, 94, pp.119-131, 2007.

[Yamagata2006] 山形育平, 高宮安仁, 中田秀基, 松岡聡, "グリッド上における仮想計算機を用いたジョブ実行環境構築システムの高速化," 情報処理学会研究報告, 計算機アーキテクチャ研究会報告(ARC), 167, pp.127-132, 2006.

[Yamaguchi2012] 山口鉄平, 新田泰広, 稲葉雅美, 秋山義幸, 杉山達也, 川上真澄, 島袋潤, 小川秀人, "制御ソフトウェアのモデルベース開発への移行方法," 情報処理学会研究報告 ソフトウェア工学 (SE), 2012(1), pp.1-6, 2012.

[Yamamoto2008] 山本修一郎, "要求工学(第 48 回)：要求モデリングと誤り," ビジネスコミュニケーション 2008 年 10 月号, 2008, http://www.bcm.co.jp/site/youkyu/youkyu48.html, 2015.12.4 参照.

[Yamanaka1999] 山中弘, 田村直樹, "要求仕様に基づくソフトウェアソースコードの自動生成法," 情報処理学会研究報告 ソフトウェア工学研究会報告(SE), 99(28), pp.93-100, 1999.

[Yanagi2010] 柳慶吾, 石尾隆, 井上克郎, "ソフトウェア部品利用例抽出のためのデータフロー解析手法の提案と評価," 情報処理学会研究報告 ソフトウェア工学(SE), Vol.2010-SE-167, No.29, pp.1-8, 2010.

[Yoshimura2006] K. Yoshimura, D. Ganesan, D. Muthing, "Defining a strategy to introduce software product line using the existing embedded systems," Proc. of 6th ACM & IEEE International Conference on Embedded Software, pp.63-72, 2006.

[Yoshimura2007] 吉村健太郎, D. Ganesan, D. Muthig, "プロダクトライン導入に向けたレガシーソフトウェアの共通性・可変性分析法," 情報処理学会論文誌, vol.48, no.8, pp.2482-2491, 2007.

[Yoshimura2008] K Yoshimura, F. Narisawa, K. Hashimoto, T. Kikuno, "Factor analysis based approach for detecting product line variability from change histrody," Proc. of MSR 2008, pp.11-18, 2008.

[Yoshimura2011] K. Yoshimura, J. Shimabukuro, T. Ohara, C. Okamoto, Y. Atarashi, S. Koizumi, S. Watanabe, K. Funakoshi, "Key Activities for Introducing Software Product Lines into Multiple Divisions: Experience at Hitachi," Software Product Line Conference (SPLC) 2011, pp.261-266, 2011.

[Yoshizawa2012] 吉澤智美, 西康晴, "ソフトウェアテストの最新動向とフロントローディング (< 特集> 我が国のソフトウェア品質技術の潮流)," 品質, 42(4), pp.467-477, 2012.

[ZPC] キャッツ, "ZIPC," http://www.zipc.com/product/zipc/, 2015.12.5 参照.