コンシューマ向け組込みシステム開発における不具合解析と性能検証の効率化

長野　岳彦

電気通信大学　大学院　情報システム学研究科

博士（工学）の学位申請論文

2022 年 12 月

コンシューマ向け組込みシステム開発における不具合解析と性能検証の効率化

博士論文審査委員会

主査　大須賀　昭彦　教授
委員　田中　健次　教授
委員　広田　光一　教授
委員　田原　康之　准教授
委員　石川　冬樹　客員准教授

Improving Efficiency of Software Bug Analysis and Performance Verification for Consumer Embedded Systems

Takehiko Nagano

Abstract

The scale of software development of embedded systems has increased every year. For example, around 1981, automobile ECUs software scale was about 50,000 lines. By the mid-2010s, ECUs software scale had grown to 200 million lines.

Consumer embedded systems development must address the following three restrictions.

1: Short-term development (shipped twice a year, such as summer model and winter model).

2: Derivative development considering grades and regulations.

3: Adoption of OSS and external products to support large-scale development.

Furthermore, high reliability is also required.

In this way, we must satisfy conflicting demands of ensuring quality while improving product development efficiency. For this problem, we introduced enterprise development methods for a consumer embedded systems when adopting linux for digital television. Above methods include adoption of development standards, performance estimation and adoption of various tools such as Linux Kernel State Tracer (LKST). Consequently, we were able to ship initial products that adapted Linux. However, new problems have occurred, such as delayed delivery and response to software failure after shipping. As a result of investigating these problems through in-house cases, we needed to solve following two problems.

(1) Improvement of failure analysis efficiency in debugging process

(2) Improvement of performance verification efficiency in design process

We continued to analyze above two problems through our in-house cases. We analyzed 2600 work information of debugging process for problem (1). As a result, we clarified that following three solutions are necessary.

a) Reliable tracing of failure information for low repeatability software bug
b) Narrowing huge trace information
c) Easier understanding of trace information

We decided to solve above problems by providing following three functions.

a. Long time trace function
b. Analysis target filter function
c. Format conversion (Visualization) function

Moreover, we analyzed problem (2) through interview with storage products developers. As a result, we have clarified that accuracy can not be obtained by the statistical performance estimation method (ex. queueing theory). The reason was that the information on the derived part could not be obtained sufficiently. Therefore, we decided to use model checking technology for efficient verification. When developing consumer embedded system, we reuse source code of past products. Therefore, the man-hours for creating verification model of reused part becomes a problem. Hence, we propose and evaluate the performance verification model creation method using existing product source code.

We applied the failure analysis environment consisting of above three functions to twenty-four cases of six products. As a results, we solved eighteen bugs. As an example of results, we were able to solve the problem that used to take 8 days to solve, in 1.91 days. Moreover, the work man-hours could be reduced by 76.13% from the original. Next, by using our proposal performance verification method, we achieved an average modeling development hour reduction of 52.76 % in small-scale model development and 69.07% in actual product scale model development.

コンシューマ向け組込みシステム開発における不具合解析と性能検証の効率化

長野　岳彦

概要

組込みシステムのソフトウェア開発規模は年々増加しており，例えば自動車の車載コントローラのソフトウェア規模は 1981 年の段階で 5 万行程度であったが，2010 年代中盤には 2 億〜3 億行に増加している．組込みシステムの中でもコンシューマ向けの組込み製品は，夏モデル・冬モデルといった年二回の出荷に代表される短期開発，グレードや地域の法規を考慮した多数の派生開発，大規模開発に対応するための Linux 採用に代表されるオープン化や導入品の採用といった制約に対応しつつ，信頼性も確保した上で製品を提供しなくてはならない．

このように，コンシューマ向け組込みシステムの製品開発では効率化を進めつつ，品質を確保するという，相反する要望を満足する必要がある．このような問題に対し，執筆者の所属企業では，コンシューマ向け組込み機器に対する Linux 導入のタイミングで，エンタープライズ系のシステム開発で培った開発基準や性能見積もりに代表される手法，Linux Kernel State Tracer(LKST)に代表されるツールの導入を実施した．それらを用いて製品開発を進めることで，Linux に対応した初期製品を出荷した．その中で想定していた納期からの遅延や，出荷後の障害発生に伴うソフトウェアの更新対応などの新たな問題が発生した．これら問題に対し社内事例を調査したところ，組込みシステムの(1)デバッグ工程における不具合解決の効率化と(2)設計工程における性能検証の効率化という二つの課題解決が必要であることが明確になった．

我々はこの二つの課題に対し，それぞれ更に社内事例を通して分析を続けた．課題(1)に対しては，製品開発時のデバッグ工程における 2600 件の作業情報を分析し，デバッグ工程において再現性の低い障害を発生時に確実に記録すること，膨大な情報の絞り込みと，情報の理解容易化が必要であることを明らかにし，長時間トレース機能，解析対象絞り込み機能，形式変換機能からなる障害解析環境によるデバッグ工程の効率改善に取り組むことにした．課題(2)に対しては，課題解決のニーズが高かったストレージ製品の開発担当者に対するヒアリングを通じて研究内容を決めた．その結果，これまでの待ち行列理論など統計を用いた

性能見積もり技術では，派生開発が多い製品の開発においては見積もり精度が出ないことが判明した．それに対し，モデル検査を用いて効率的かつ網羅的な検証を実施したいというニーズを明らかにした．その上で製品開発が前世代のプログラムを流用して新製品のシステムを開発する差分開発になっており，性能検証モデルを作成する際に，流用箇所も含めてモデル化する工数が大きな課題であることを明らかにし，差分開発における性能モデル検査技術の導入を目的とした，既存ソースコードを流用した性能検証用モデル開発の効率化を提案し，評価した．

評価の結果，障害解析環境は 6 製品 24 件の障害に適用し，18 件の障害解決を実現した．そのうち 1 例として既存手法では解析に 8 日間かかっていた問題を，1.91 日に短縮し，作業工数を 76.13%削減することができた．また，性能検証用モデル開発の効率化においては，ソートアルゴリズムなど小規模のモデルで平均 52.76%，実製品開発規模である Hard Disk Drive(HDD)のキャッシュモデルで平均 69.07%の開発時間削減を達成することができた．

目次

図目次

表目次

第1章　序論

本論文では，コンシューマ向け組込みシステムの開発工数削減を目的とし，課題とした障害解析と性能検証の効率化に関する研究について述べる．本章では，1.1 で背景となる組込みシステム開発における概況について述べる． 1.2 で組込みシステムについて述べ，1.3 で本研究の対象であるコンシューマ向け組込みシステムについて述べ，1.4 でその開発が抱える公知の課題と，執筆者所属企業における課題について述べる．1.5 では，本論文の取り組みの位置づけについて述べる．最後に，1.6 で本論文の構成について述べる．

1.1　本研究の背景

組込みシステムではインターネット接続機能や，ユーザインタフェースの高度化など求められる機能が増加したため，搭載されるソフトウェアの規模が増大している．例えば自動車 1 台に搭載される機器のソフトウェア行数の合計は，1981 年には 5 万行程度であった．しかし 2000 年代半ばには 1 億行をこえ，2014 年の段階では 2 億行から 3 億行に，現在では更に増加している[1].

一方で，コンシューマ家電などの組込みシステムは，安価な労働力で価格を削減する中国メーカや，モジュールなど水平分業から参入する台湾メーカのような新興国企業による影響を受け，コモディティ化*が進んでいる．その結果，商品ライフサイクルの短命化，急激な価格低下の影響を受けている[2]．そのため，開発にかけられるコストも制限があり，開発の効率化によるコスト削減が望まれている[3].

このような要望に対し，これまで多くの取り組みがされてきた．その結果，開発コストの削減に対する効果が出てきており，2016 年度は 43.0%をソフトウェア開発費が占めていたものの，2019 年は 27.3%にまで削減されている．一方でいまだに開発費のコスト削減要求が非常に高い[4].

この様に高いコスト削減要求にもかかわらず，高い信頼性が求められるのも組込みシステムの特徴である．例えば，ミッションクリティカルシステムに使われ，耐久性や信頼性が最初から高く求められる組込みシステムでなく，テレビに代表されるコンシューマ向け組込みシステムでも 10 年～20 年と長い製品寿命が求められる．なぜならば，これまでの製品は，10 年～20 年といった長い期間顧客の要求に応えてきたため，これまで同様の高い品質をメーカが保証すると認知されているからである[2].

1.2　組込みシステム

---

* コモディティ化：ある商品が一巡して汎用品化が進み，競合商品間の差別化が難しくなって，価格以外の競争要素がなくなること
https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h21/html/l212c000.html（2022 年 5 月 21 日現在）

本節では，組込みシステムについて述べる．組込みシステムとは，各種の機器に組み込まれてその制御をおこなうコンピュータシステムのことをいう．例えば，プラント制御システムに Personal Computer（PC）ベースのハードェアが用いられることが多々ある．この場合，ハードウェア的には PC そのものであり，組み込まれているとはいいにくい．一方でプラントという「機器」に組み込まれていて，プラントを制御しているコンピュータシステムと考えれば，組込みシステムに含める．一方，Personal Digital Assistant(PDA)やゲームマシンはコンピュータそのものであるとも考えられ，組み込まれているかどうかは明確ではない．しかし携帯電話はその出自から組込み機器と考えられている．

以上より，組込みシステムをこれらの境界例も含むと広く定義すると，PC やワークステーションなどのいわゆる汎用システム以外の，ほぼすべてのコンピュータシステムが組込みシステムと同等の意味でつかわれる場合もある[5].

このような考えをふまえ，一般的に組込みシステムといわれるものには，以下表 1-1 に示すようなものがあげられる[6].

表 1-1　組込みシステムの例

分類	例
家庭	デジタルテレビ，ビデオ機器，オーディオ機器，エアコン，洗濯機，電子レンジ，炊飯器，電子楽器，テレビゲーム，カーナビ，携帯電話など
屋外	自動販売機，信号機制御，アーケードゲームなど
オフィス	複合機（プリンタ，コピー，スキャナなど），ストレージサーバ，照明制御システム，空調制御システム，セキュリティ監視制御など
通信機器	多機能電話，ファクシミリ，放送機器，無線システム，アンテナ制御装置，衛星制御装置，交換機，Automatic Teller Machine(ATM)装置，ルータなど
工場	Factory Automation(FA)用コンピュータ，工業用ロボット，測定器，郵便物自動区分機，自動倉庫システムなど
交通	自動車，航空機，鉄道車両，飛行制御，ロケット姿勢制御，列車運行状態監視制御システムなど

1.3　コンシューマ向け組込みシステム

本節では，本研究の対象であるコンシューマ向け組込みシステムについて述べる．コンシューマ向け組込みシステムは，主に個人・家庭向けに作られた組込みシステム製品を指し，法人・事業者や，生産者向けの製品と区別するためにこのような定義をする．1.2 で述べた組込みシステムのうち，テレビ，ビデオ，携帯電話，カーナビ，ハードディスクなどが代表的なものである．

次に製品を例にコンシューマ向け組込みシステムの役割について説明する．一つはこれ

らシステムが扱う特徴的な処理であり，テレビやビデオなどでいう映像や音声の情報の処理である．これらの情報はデジタル化された上に，圧縮に代表される高度な情報処理や，画像処理を実現する必要があるため，高い演算性能が必要となる．また，消費電力やリアルタイム性が求められるため，汎用 Central Processing Unit(CPU)ではなく，専用の回路を用いることがある．もう一つは付帯機能の処理である．これらのシステムの多くは，メニュー画面の処理，リモコン処理などのユーザインタフェースや，インターネットの接続など，周辺機能の実現も求められる[7].

以上のような要求の実現は，車における携帯電話回線を利用した通信を用いてソフトウェア更新をする Electronic Control Unit (ECU)や，ナビに代表される Human Machine Interface(HMI)の処理をする ECU でも存在する.

1.4 コンシューマ向け組込みシステム製品の開発における課題

本節では，本研究の対象であるコンシューマ向け組込みシステムの製品の開発における課題について述べる．最初に公知の課題について述べ，その後執筆者所属企業における課題について述べる.

コンシューマ向け組込みシステムは，以下の様な公知の課題を持つ.

(1) 短期開発

デジタルテレビやビデオカメラなどの家電製品を中心に，夏モデル・冬モデルといったように，年 2 回の製品リリースが予定されており，その予定に合わせて製品開発をする必要がある．同様に，自動車においても，車両販売はマイナーアップデート，メジャーアップデートなどのモデルチェンジがあらかじめ2年, ないし4年となっており[8]，その期間に製品，または製品に関連する部品を開発しなくてはならず，法人・事業者や生産者向け組込みシステムに比べ製品開発期間が短期である.

(2) 多数の派生開発

家電や自動車などでは，製品のグレードによって搭載される機能を差別化しており，機能を実現するハードウェアやソフトウェアが異なっている．また，国ごとの法規や規格などへの対応も必要であり[9][10]，多くの派生開発が必要である.

(3) 急速なオープン化への対応

2000 年代前半までコンシューマ向け組込みシステムは，リアルタイム Operating System(OS)と専用アプリといったような，旧来の組込みシステムとおなじ構成であった．その後，テレビ，ビデオカメラ，携帯といったコンシューマ向け家電製品を中心にシステムのデジタル化，ネットワーク化が進んだ．その対応のため，Linux†といった汎用 OS や，ネットワーク対応のため TCP/IP プロトコルスタック，Graphical User Interface(GUI)開発のためのミドルウェア，Web ブラウザといった製品の導入ないし

† Linux®は Linus Torvalds の米国及びその他の国における登録商標です.

外部調達が進んだ[11]．このような流れは自動車分野でも HMI 製品を中心に進んでおり，Automotive Grade Linux や AUTOSAR‡-AP の導入が始まっている[13]．

また，これら 3 つの公知の課題に加え，コンシューマ向け組込みシステムは諸外国の開発参入により，生存競争が過酷になっている．それにより，商品ライフサイクルは短く，機種展開数は増加した一方で価格は低下し，家電業界は消耗戦を強いられている．具体的には，Cathode Ray Tube(CRT)テレビは 13 年，Plasma Display Panel(PDP)は 4 年，Video Home System(VHS)デッキは 6 年，Digital Versatile Disc(DVD)プレーヤは 4 年，DVD レコーダは 2 年で価格が半額に下落している．一方で，2000 年以降製造された薄型ＴＶは，いまだに使用されているケースもあり，顧客はデジタル家電製品や家電メーカに対し，高い品質の製品を提供し，保証するという認知が変わらずされている[2]．その結果，デジタル家電におけるソフトウェアも，たとえ安価な製品だとしても，高い性能・品質が求められている．

次に，執筆者所属企業における課題について述べる．上記したコンシューマ向け組込みシステム公知の課題に起因し，執筆者所属企業では以下 2 点の課題を抱えていた．詳細は第 2 章で述べるため，本項では概要のみ述べる．

一つ目の課題は，製品のデバッグ工程における不具合解決の効率化である．本研究の着手時は，上記した(1)短期開発と(3)急速なオープン化への対応をしつつ，コンシューマ向け組込みシステムの高機能化に対応しなくてはならない状況であった．例えば DTV 開発では，ネットワーク機能の強化や，デジタル化への対応が必要であった．この対応のため，Linux やネットワーク機能などに関連する多数のソフトウェア/ハードウェア部品を導入した．結果，自社開発のアプリだけでなく，導入品のソフトウェア，ハードウェア，デバイスドライバなどの組み合わせによるリソースの競合など，組み合わせパターンや発生タイミングがわからない再現性の低い不具合の解決に多くの時間を費やしていた．

二つ目の課題は，製品の設計工程における性能検証の効率化である．上記した(1)短期開発と(2)多数の派生開発をしつつ目標とする性能を達成するためには，本研究着手時点で実施していたテスト工程における性能テストと改善だと，手戻りが大きく開発工数を圧迫することがあった．そのような問題に対しモデル検査技術が有効であることが公知であった．しかし性能を達成するための実装はプログラムの中に散在するため，前世代のプログラムを流用して新製品のシステムを開発する差分開発をしているコンシューマ向け組込みシステムにモデル検査を導入するには，導入タイミングで非常に大規模の検証用モデルを開発しなくてはならず，モデル検査の導入が困難であった．

1.5　本論文の位置づけ

本論文は，執筆者所属企業における複数のコンシューマ向け組込みシステムの開発事例を通じて製品開発上の課題を導出し，一連の解決事例を示したことを特徴とする事例研究

‡ AUTOSAR は，アウトザール・ゲゼルシャフト・ビュンゲルヒッヒェン・レヒツの日本及びその他の国における登録商標です．

である．導出された課題に対し，執筆者らの過去の製品開発における知見をもとに解決手法を立案，複数製品を対象に評価・検証を行った．本研究で得た新たな知見は，特に近年増加している高機能なコンシューマ向け組込みシステムを対象としている．以下，各章で述べる知見と，貢献すると考える対象を列挙する．

3 章で述べるデバッグ工程における障害解析効率化の内容は，①解析対象絞り込み機能，②長時間トレース機能，③形式変換機能からなり，Linux 搭載のコンシューマ向け組込みシステム向けに検討した内容になっている．これらは，コンシューマ向け組込みシステムの開発をする際に，開発工程を遅延させる原因となっていた再現性の低い不具合の原因分析を効率化するために検討されたものである．再現性の低い不具合は，複数のソフトウェア，ハードウェアの動作の組み合わせやタイミングにより引き起こされる．以上より，本機能はマルチタスクシステム上のアプリケーション開発に係わる開発者や，機能の横串を通して製品の不具合解析に関わる品質保証部門の担当者に適している．

４章で述べる設計工程における性能検証効率化の内容は，既存製品のソースコードを利用した差分開発[12]をする場合における性能検証を効率的に行う一手法であり，こちらは差分開発をしているコンシューマ向け組込みシステムにおいて，性能の検討を行っている設計者，開発者に適した内容となっている．

1.6　論文の構成

本論文は，コンシューマ向け組込みシステム向けに，高い品質を維持しつつ製品開発を効率化することを目的として，執筆者が取り組んだ内容について述べる．本論文は 5 章から構成される．第 1 章の序論では，本研究の背景となる問題について述べ，特に本研究にて取り組むコンシューマ向け組込みシステム開発における背景となる課題について述べた．第 2 章の先行研究と取り組む課題では，組込みシステム開発について述べた後，関連する既存研究の動向，執筆者所属企業における状況，執筆者所属企業における課題について述べた後，執筆者の取り組む課題について述べる．第 3 章，第 4 章では，明確化された課題に対する研究内容について述べる．3 章では，デバッグ工程における障害解析作業の効率を改善する障害解析環境機能の開発について述べる．4 章では設計工程におけるモデル検査を用いた性能検証手法の効率的な導入方法と，性能探索の実現方法について述べる．最後 5 章は結論として，全体の研究成果をまとめ，今後の課題について述べる．

第2章　従来施策と本研究の課題

本章では，2.1 にて組込みシステム開発の工程とその内容について述べ，そこで執筆者が取り組む箇所について説明する．2.2 にて各工程において行われている開発効率化に関連する既存施策について述べる． 2.2.1 では，開発効率化に関連する既存技術の調査状況について述べる． 2.2.2 では 2.1 で述べた全ての工程について概略を述べ，2.2.3 と 2.2.4 では，本研究で執筆者が取り組むデバッグ工程と設計工程における既存手法の詳細について述べる．2.3 にて執筆者所属企業におけるコンシューマ向け組込みシステム開発に関する状況について述べ，2.4 にて本研究にて取り組む課題について述べる．

2.1　組込みシステム開発の工程

本節では，組込みシステムの開発の工程とその内容について述べる．一般的なソフトウェア開発工程を規定するモデルとして，V モデルがある[14]．また，V モデルを拡張し，組込みシステムを考慮し，デバッグ工程を追加したモデルが定義されている[15]．これらのモデルを統合し，以下図 2-1 に組込みソフトウェア開発の工程モデルを示す．開発は設計，実装，テスト，デバッグの４工程からなる．それらの各工程は更に詳細な工程からなる．これらの各工程の内容と，工程の問題について以下に述べる．

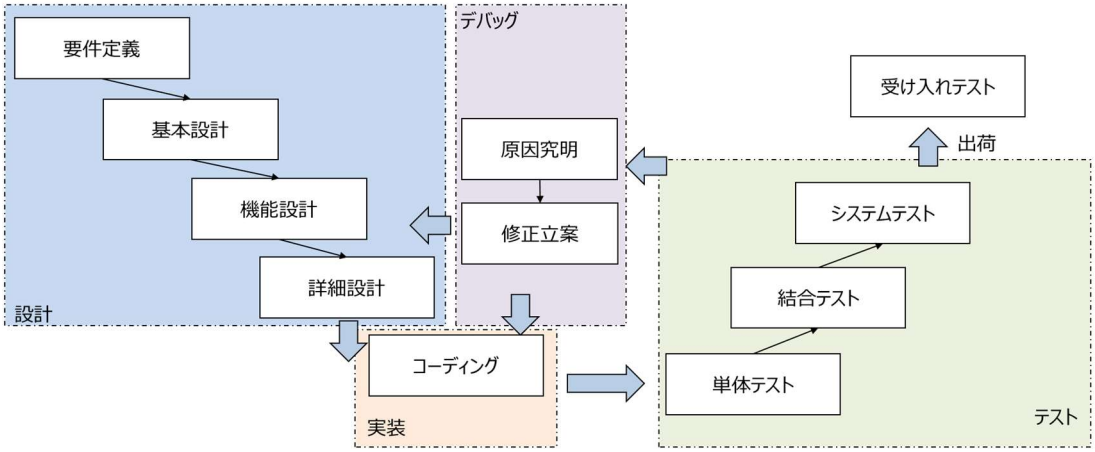

図 2-1　組込みシステムの開発工程

（1） 設計工程

設計工程は，要件定義と基本設計，機能設計，詳細設計の各詳細工程からなる．要件定義は，要求分析とも呼ばれ，組込みシステムの要件や，組込みソフトウェアに対する要件を定義する[15]．基本設計は，要件定義（分析）と機能設計の間の工程で，対象物の機能や構成などの大枠や基本的な仕様を決める工程である．機能設計は，基本設計と詳細設計の間の工程で，システムの機能ごとに仕様を定義する工程である．詳細設計は，機能設計とコーディングの間の工程で，前工程で定義された要素の仕様や動作の詳細を定義する工程である．

（2） 実装工程

実装工程は，コーディングの詳細工程からなる．コーディングは，設計工程で定義した設計情報に従ってソフトウェアのソースコードを記述し，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，単体テスト工程，結合テスト工程，システムテスト工程の各詳細テスト工程からなる．設計工程で定義された要件，仕様を満足しているかを確認する工程である．単体テスト工程は，実装工程で作成された実行可能形式のプログラムを用い，詳細設計で定義された仕様を満足するかを確認する工程である．結合テスト工程は，機能設計で定義された仕様を満足するかを確認する工程である．システムテスト工程は，基本設計で定義された仕様を満足するか確認する工程である．

（4） デバッグ工程

デバッグ工程は，原因究明と，修正立案の各詳細工程からなる．テスト工程にて不具合を検出した場合，実施される．原因究明工程は，テスト工程で発見した不具合を，再現条件をもとに再現し，再現時の情報をもとに，不具合の原因を突き止める工程である．修正立案工程は，不具合の原因をもとに，設計の変更や，実装の変更といった修正内容を立案する工程である．

以降，本論文は上記各工程のうち，設計工程とデバッグ工程について詳しく述べる．その理由は，研究着手時に執筆者所属企業における事例を通し開発の問題を分析したところ，課題が設計工程，並びにデバッグ工程における効率の悪さに起因することが判明したためである．上記事例分析結果については，2.3 にて詳述する．

2.2 開発効率化に関連する既存研究の動向

2.2.1 開発効率化における調査の状況

組込みソフトウェア産業における課題の調査は，大手企業から中小企業まで多くの企業が携わっており，その実態の把握は難しい．その実態把握のため，2003～2010 年度までの 8 年間にわたり，経済産業省が主体となって，「組込みソフトウェア産業実態調査」を実施した[16]．その後を受けて 2011 年～2012 年度の 2 年間は，IPA/SEC が主体となり「ソフトウェア産業の実態把握に関する調査」を実施した[17]．その後 2016～2018 年度は「組込みソフトウェア産業の動向把握等に関する調査」[18]，2019 年度以降は「組込み/IoT 産業の動向把握等に関する調査」が実施されている[19]．

これらの調査は国内外の組込みシステムに関連する企業に対して，アンケートを用いた

調査を実施している．例えば組込みソフトウェア産業実態調査の場合，経営者や事業者といった層に対する調査，開発プロジェクト責任者に対する調査，技術者個人に対する調査など幅広く行っている．その中で，開発プロジェクト責任者に対し，ソフトウェア品質管理に関する設問も行っており，企業における組込みシステム開発に関する課題などが調査されている． この調査では，製品に使用された OS や言語といった情報から，本論文で扱っている品質課題に関連し，どのようなツールを導入したか，そのツールの課題は何かといった内容や，ソフトウェアの不具合が顕在化した工程や今後のプロジェクトで改善すべき項目の調査結果が精緻に述べられている．

また類似した調査として，JEITA ソフトウェア事業委員会が 2006 年から 2014 年まで「組込み系ソフトウェア開発の課題分析と提言」というアンケート調査を用いた実態把握，課題分析，課題解決に向けての提言を行っている．こちらの報告では，例えばアンケート調査した対象のプロジェクトの 54%が出荷後に重度の障害を検出しているといった実態報告や，品質に寄与する要因が開発規模の大規模化と短納期化にあることや，ソースコードの自動生成やテスト自動化といった自動化に関する手法の適用時の課題についても述べられており，各社の実践的な開発の状況とその課題を知ることができる．

2.2.2 各工程における既存研究の概略

2.1 で述べた各工程において，各工程における公知の課題と，それに対する既存手法を以下に述べる．

（1）設計工程

設計工程においては，主に設計品質の低さに起因する課題がある．具体的には，以下に示す課題がある．

・システムに対する要求の明確化が不十分，やりたいことが後から顕在化
・システムの性能に代表される非機能は，各機能担当者の業務領域を超えた部分の影響により変化することから，非機能要件の定義が不十分
・設計は自然言語を用いて記述するため曖昧な部分が含まれる．それに起因して不具合発生
・設計検証の網羅性が低く，仕様の正しさの判定が厳密でない

これらの課題は，組込みシステムの特性であるリアルタイム性の確保や，高い信頼性の確保に必須である．更に開発規模の増大から，ますます重要度を増している．それに対し，要件定義の進め方，成果物，技法，ノウハウをまとめたガイドラインの作成や，設計の曖昧さを防ぐためのモデルベース開発技術，設計において仕様記述や検証に用いられる形式手法などが研究・開発されてきた．

ここでは本研究で取り上げるモデル検査を包含する手法である形式手法について説明する．形式手法とは，論理学や離散数学など数学的基盤に基づく形式仕様記述言語を用いた正確な仕様記述を核とした技法であり，仕様記述や検証に用いられる．仕様，設計，実装の正しさを，テスト工程を待たずに正確に検証できるため，組込みシステム開発工数の工数削減

が期待できる．形式手法は，仕様記述言語，検証方法，作業過程の３要素から構成される．仕様記述言語は，システム並びに満たすべき性質の厳密な記述・表現にかかわり，どのような性質を表現するかに応じて様々な言語が提案されている．検証方法は，仕様記述言語を用いて書いた仕様と満たすべき性質が整合していることを機械的に確認する方法であり，代表的な手法として演繹法とモデル検査法がある[20]．

演繹法は，検証したい性質が仕様によって満たされるかを，推論規則に基づいて証明する．人手で証明する他に，定理証明系の支援技術として，Vampire[21]や対話証明系ツールのCoq[22]などがある．モデル検査法は，仕様からシステムのモデルを作成し，検証したい性質が満たされていることをモデル上で検査する．主に振舞いの検証に用いられ，ツールを使った自動検証も可能である．言語とツールには，Promela と SPIN[23]などがある．

次に，性能問題の品質を扱う手法について述べる．それらは主に数理モデルを用いた性能の予測・検証と，設計モデルを用いた性能の予測・検証が行われている．

数理モデルを用いた手法としては，待ち行列理論[24][25]や Markov モデル[26]を用いた検証があげられる．設計モデルを用いた手法としては，時間オートマトンで記述した時間制約付きモデルを検証するツールの UPPAAL などがあり[27]，プロトコルの時間制約の検証で実適用されている[28]．また，統計モデルを扱える PRISM などの検証器もある[29]．

（２）実装工程

実装工程においては，少ない開発費で目的の開発を達成するための，開発効率の向上に起因する課題がある．具体的には以下に示す課題がある．

組込みシステムは第 1 章に示した通り，機能の増加，およびそれによるソフトウェア規模の増大による開発工数の増加が問題となっている．一方で量産品を中心にコスト削減が要求され，それに伴い開発費の削減が要求され，開発工数に制約がかかるといった，相反する要望を解決する必要がある．

これらの課題に対し，以下に示す研究や開発が行われてきた．

・Open Source Software（OSS）やプロプラエタリなソフトウェアの導入
・仮想開発環境の導入による開発の仮想化と実機の削減
・ソフトウェア部品の再利用を効率よく実施する Software Product Line(SPL)技術の導入
・要件や仕様からソースコードを出力できるコード自動生成技術の導入

（３）テスト工程

テスト工程においては，テスト品質の低さと，テスト効率の悪さに起因する課題がある．テスト品質については，テストが十分ではない，テストが十分か判断できないといった課題があった．また，テストの効率の悪さについては，何度も同じテストをしなくてはならないことや，テスト条件（パラメータ）が多様なため，テストケースが膨大になってしまう課題などがあった．

これらの課題は，1.4 に示したコンシューマ向け組込みシステム開発の課題である高い信頼性の確保に直結するため，改善が求められるが，ソフトウェア開発規模の増大により，ま

すます改善が難しくなっている．

このような課題に対し，以下に示す研究や開発が行われてきた．

・テストによる品質の確保を定義するテスト方法論の導入

・テストの効率改善を目的とした，テストケース自動生成の導入

・テスト自動化技術である Continuous Integration and Continuous Deployment(CI/CD)などをはじめとするテスト自動化技術[30]の導入

・テストケースを削減する手法（ペアワイズ法/オールペア法）[31]の導入

（4）デバッグ工程

デバッグ工程においては，デバッグの効率の低さに起因する課題がある．具体例を，以下に述べる．

デバッグ時の重大な課題の一つに不具合の解析に十分な情報が無く，デバッグに着手できない課題がある．具体的には不具合の発生源が別のアプリや導入品にある場合，不具合の再現条件を特定することが難しく，不具合が再現せずにデバッグに必要な情報を十分に取得できない課題がある．また，組込みシステムのデバッグの場合，システムのリソースが少なく，解析に十分な情報が取れず解析ができない課題や，そのために試行錯誤的に何度も情報を取らなくてはいけない課題がある．また，組込みシステムのソフトウェア開発規模増大に伴う機能の肥大化により，同時に解析しなくてはならない情報が増え，解析が困難になっているという課題もある．この詳細は，以降詳しく述べる．

これらの課題に対し，以下の研究や開発が行われてきた．

・再現に必要な情報を取得するためのトレース機器の導入

・OS レベルでソフトウェア全体の情報をトレースする技術である LKST[32][33][34]，Kprobes§，LTTng**の開発と導入

・膨大なトレース情報を可視化，効率よく解析する技術の開発と導入

・性能に着目し，解析を支援するプロファイラ技術である perf††，OProfile‡‡の導入

2.2.3 デバッグ工程における既存研究

本項では，本研究で取り組む工程の一つである，デバッグ工程における既存手法について詳しく説明する．組込みシステムの障害解析においては，製品が市場投入されるまでの時間が問題となっており，デバッグフェイズの最適化が必要である．そこで実行トレースを用いた分析が強力であることは既にわかっており[35]，多くの研究がされている．例えば，トレース分析結果を可視化する研究は様々あり，Java で実装されたプログラムの実行履歴を可

§ Kernel Probes(Kprobes), https://www.kernel.org/doc/Documentation/kprobes.txt (2022 年 4 月 3 日現在)

** LTTng is an open source tracing framework for Linux, https://lttng.org/ (2022 年 4 月 3 日現在)

†† Linux kernel profiling with perf, https://perf.wiki.kernel.org/index.php/Tutorial(2022 年 4 月 3 日現在)

‡‡ OProfile, https://oprofile.sourceforge.io/news/(2022 年 4 月 3 日現在)

視化する JIVE をはじめ[36]，マルチプロセッサ環境で動作するソフトウェアデバッグ用のトレース可視化ツール TLV[37]，リアルタイム性を持つ組込み機器向けに開発された OSS の可視化ツールである Timedoctor[§§]など，多くの研究，ツールや手法が報告がされている．また，組込みシステム向けを含め多くのトレースツールや[33] [38][39]，プロファイルツールなど解析ツールの開発もされている．さらに，大量に取得したデータの効率化については，パターン認識を用いて周期性のある振る舞いの情報を抽出する研究がおこなわれている[35]．

これらの研究のほかに，デバッグ工程における性能対策も取り組まれている．開発プロセスの下流工程では，主に二つの性能対策が実施されている．まずは，開発したソフトウェアやシステムの性能が，目標の性能を達成しているかを評価し，問題があった場合の要因を分析するための性能テストとその結果の分析がある．そしてその分析の結果を元に，システム性能を改善するための再設計やチューニングなどを行う[40]．これらの行為は，実際のシステムに対しワークロード分析を行い，次世代の予測を実施した例等も報告されている[41]．さらにデータベースソフトウェアに代表されるソフトウェアには，性能チューニングのためのパラメータが準備されている．そして，その使いこなしのためのドキュメントやツールも準備されている[42]．

これらの報告は，トレース自体，可視化自体を対象に進めているものが多いが，我々が課題に挙げている再現性の低い障害の解決には，十分に考慮されていない．例えば，トレースの取得では，再現性の低い障害が発生した際のトレース情報を，障害発生時に必ず記録しておかなければならない．このような問題に対し，OS トレーサの研究・開発はエンタープライズ系のシステムや，PC を中心にされていた[34]．そのためコンシューマ向け組込みシステムと異なり，CPU，メモリ，ストレージに代表されるリソースは豊富で，長時間のトレースを取るためにシステムのリソースが足りなくなり挙動が不安定になることや，メモリからログを出力するストレージが足りなくなるといった課題が大きな問題になることはなかった．一方でコンシューマ向けの組込みシステムの開発では既に問題になっており，JTAG-ICE などを使い，外部にトレースを出力し，記録する仕組みが提供されていた[***]．しかしこの仕組みは，トレース容量が 1Gbyte と多くないことと，公開されている API を用いてトレースを取得する必要があることから，改めて OS トレースを取得するための Linux Kernel の修正とテストが必要になり，工数が増えるという問題を有していた．

更にこれら OS トレースの情報を収集できても，その際に取得した膨大な情報を効率良く解決しなくてはならない.この課題に対し，パターン認識を用いた検出が検討されているが[35]，我々が取り扱う再現性の低い不具合情報は周期性がないため，検出するには別のアプ

---

§§ David Legendre, Francois Audeon: Detection and Resolution of Real-Time Issues using TimeDoctor, https://elinux.org/images/3/3e/Detection-of-RT-issues-with-TimeDoctor.pdf (2022 年 4 月 3 日現在)

*** 京都マイクロコンピュータ社 Partner Jet: https://www.kmckk.co.jp/jet/(2022 年 5 月 19 日現在)

ローチの検討が必要であった．また，問題点の絞り込みによる効率化という観点では，プロファイラを用いた研究があるが，こちらはCPU使用率の観点でボトルネックを検出するものであり[40]，トレース情報の絞り込みへの適用には工夫が必要であった．

最後に，トレース結果の可視化による理解容易化について述べる．この課題については，特定のトレーサの出力に特化した可視化や**，収集したトレース情報を汎用的に可視化するといった目的で作られていた[37]．そのため，再現性の低い不具合の情報に特化した可視化については，ツールの使用者が個々に検討する必要があった．

2.2.4　設計工程における既存研究

本項では本研究で取り組む工程のもう一つである，設計工程における既存手法について詳しく説明する．組込みシステムの性能問題に対し，これまで開発現場では，人材投入や動作環境の並列化といった対応を行ってきた．またその一方で科学的なアプローチとして，以下に列挙する様な性能対策が実施されてきた．その事例と課題を紹介する．

システムの性能問題に対し，我々は大きく分けて二つの対策を実施している．1)開発工程の前半にて性能の予測検討・設計を実施するアプローチ．2)開発工程の後半で，性能の検査を実施し，問題を分析・改善をするアプローチの二つである[44]．この対策において，開発工程の前半に十分な性能の予測・検査を行わず，不十分な検討に基づく設計に起因する性能不具合が開発工程後半で顕在化した場合，デバッグ工程で情報を収集・分析・対策立案後，再度設計をし，改めて開発，テストをすることになり，結果作業工数が増加する．そのため，我々は工数の増加を抑止することを目的に，開発工程前半における技術に注目する．

上記した開発工程前半における性能の検査技術としては，性能モデルを用いた検査技術がある．それは，数理モデルを用いた検査技術や，設計モデルを用いた検査技術などがある．ここで性能を達成するための実装は，プログラムの中に散在している[45].そのため，性能の検査を網羅的に実施しなくては性能の問題の確認をすることは難しい．そこで本研究では，システムの網羅的な検査を目的としたモデル検査技術に注目することにした．以下，その詳細について述べる．

2.2.4.1　開発プロセスの上流における性能対策

開発プロセスの上流で実施されるソフトウェア設計の際に，システム全体の性能予測や，アルゴリズムの性能検討を実施している．その際，主に以下二手法が実施されている．A)数理モデルを用いた性能の予測・検討，B)設計モデルを用いた性能検査である．まず，A)数理モデルを用いた性能の予測・検討について説明する．代表的なモデルとして待ち行列理論による統計モデルがある．様々な分野で適用されており，実績は多い[24] [25]．また，同様の数理モデルとして，Markovモデルを用いた例もある[26]．次に，B)設計モデルを用いた性能検査について述べる．設計モデルを用いたものには，プロセス代数ベースのアプローチや，ペトリネットベースのもの[46]，　Unified Modeling Language(UML)の設計モデルに時間

概念を扱うための拡張である Modeling and Analysis of Real-Time & Embedded Systems(MARTE)を使い[47], 性能要件を記述したモデルを作成し, それを用いて性能の設計・検討を実施するもの[40]などがある.

本論文で用いるモデル検査を用いた性能設計・検討もここに含まれる. モデル検査器として有名なものに SPIN がある. SPIN はモデルの振る舞いを網羅的に計算機上で展開する検査手法で, 実行系列を明示的に算出して検索する明示的モデル検査を行う検査ツールである. 検査内容は, 線形時間命題時相論理(Propositional Liner-time Temporal Logic(PLTL) もしくは Liner Temporal Logic(LTL))で記述し, その式を Büchi オートマトンに変換し, 更にシステムの振る舞いと合成した Büchi オートマトンにより, その性質が成り立つかどうかを検査する. SPIN ではモデルを Promela と呼ばれるモデリング言語で記述する. Promela では, 動作を C 言語に似た文法で記述する. 動作として, 変数の読み出し, 書き込み, 各種算術・論理演算がサポートされる. また, 構文 if, do は条件分岐, ループの制御構造, 非決定的分岐がサポートされ, これら可能性がある全ての分岐をたどり性質が成り立つか検査する[48]. SPIN と Promela は機能面での検査に実適用例が多い. 一方, 今回対象とする性能の面では, 時間制約の検査に用いられる UPPAAL が有名で, 実適用例も多い. UPPAAL は時間オートマトン[27]を利用したモデル検査環境で, プロトコルの時間制約の検査等でも既に実適用されている[28]. また, 先進的なモデル検査技術の導入のため, ユニファイドプラットフォームストレージのソフトウェアを対象に, SPIN, NuSMV と CBMC†††の 3 つのモデル検査環境を, 検査時間と検査時におけるメモリ使用量の 2 点で比較し, その 2 点で優位であった CBMC にて製品の検査に用いた報告がある[49]. 他にも PRISM などの統計モデルを扱える検査器もある[29]. 更に時間制約以外の性能の設計・検討の例としては, ソフトウェアの動作によって消費する電力量を, CPU の命令セット単位でモデル化し, 予測する手法が提案されている[50].

一方で, これらのモデル検査技術の多くは, 機能検証のための技術として開発されており, 性能検証に用いるには改善が必要であった. 特に本研究で対象とする検証を効率化する技術については, C のソースコードを直接検証する CBMC†††[51]や, C のソースコードから機能検証用の Promela のモデルを生成する *Fever* がそのような技術に該当しており[52] そのままでは適用できないという問題がある. また, 性能検証を効率化する技術として, UML のモデルから性能検証を実施する UPPAAL のモデルを生成する異種モデル間連携基盤が提案されているが[53], こちらは生成された UPPAAL のモデルと, 実際の製品のソースコードの挙動の一致の証明や, 設計者への説明のための準備に工数が必要で, 迅速な導入が難しいという問題を有していた.

---

††† Carnegie Mellon University : Carnegie Mellon CBMC Homepage(online), https://www.cprover.org/cbmc/, (2022 年 4 月 3 日現在)

2.3　企業における状況

2.3.1　　研究着手前における状況

本研究の取り組みは，執筆者所属企業におけるコンシューマ向け組込みシステム製品開発を対象に，その開発効率を改善するために推進した施策に関するものである．そのため，本節では2000年代中頃の取り組み開始時における執筆者所属企業における開発の状況について述べる．次にエンタープライズ系の技術を用いた初期の取り組みについて述べる．最後に，その取り組みを通して得た，本研究にて取り組む課題について述べる．

2.3.2　　コンシューマ向け組込みシステムの Linux 導入

2000 年代当時，デジタルテレビやレコーダ，カメラといった家電は，デジタル化により生存競争が過酷になっていた．それにより，商品ライフサイクルは短く，機種展開数が増加した．一方で価格は低下し，家電業界は消耗戦が強いられていた．具体的には，CRT テレビは 13 年，PDP は 4 年，VHS デッキは 6 年，DVD プレーヤは 4 年，DVD レコーダは 2 年で価格が半額に下落している状況であった．一方で 2000 年以降に製造された薄型 TV といったデジタル家電製品は，未だ使用されているケースもあり，顧客は家電メーカに対し，高い品質の製品を提供し，保証するという認知が変わらずされている．結果，デジタル家電におけるソフトウェアも，たとえ安価な製品だとしても高い性能・品質が求められている．このような背景のもと，安価でも高い品質の製品を実現するため，デジタル家電の Linux 化が各社で検討された．たとえば松下電器の場合，以下の様な要求でデジタルテレビの Linux 化が検討された[2].

・複数アプリケーションの同時動作要求の拡大

例 : テレビを見ながら，裏番組を録画といった，同時動作要求の対応に対し，マルチプロセス対応の Linux 導入が検討された.

・ネットワークアクセス要件の拡大

例 : PC で醸成されたネットワークアクセスによる付加価値（インターネット接続，ストリーミングサービスの利用など）拡大を取り込むため，プロトコルスタックが充実し，動作実績が多い Linux の導入が検討された.

・ソフトウェア要求の拡大

例 : ブラウザの利用やさまざまなフォーマットの画像の表示など，多くの要件を満たすため，ソフトウェア規模が拡大した．このため，多くの機能を自前開発しなくて済む Linux の導入が検討された.

・自前での品質担保

例 : デジタル家電の品質は，メーカが最終的に担保しなくてはならない．このため，ソースコードが公開されていて，自前でテストし品質が担保できる Linux の導入が検討された.

本論文の執筆者所属企業においても，デジタルテレビをはじめとしたコンシューマ向け組込みシステムの Linux 化について，特にネットワークアクセス要件の拡大を視野に入れ

検討が進んだ．しかし Linux 導入の重要な後押しとなったのは，他の理由であった．2006 年の 4 月に発売するデジタルテレビ H/HR9000 シリーズで採用した System on a Chip(SoC) に対し，その SoC の半導体ベンダがμITRON のサポートを打ち切るからであった．

これらの要因により，本論文の執筆者所属企業では， H/HR9000 シリーズより，デジタル家電プラットフォームを Linux 化することを決定した．

2.3.3 Linux 化における開発環境の整備

2.3.2 で述べた通り，デジタルテレビ製品 H/HR9000 シリーズより，デジタルテレビの Linux 化を開始することになった．そのため，H/HR9000 シリーズを開発する際に開発環境を整備した．当時執筆者の所属企業では，エンタープライズ向けシステムの開発部門を中心に，Linux を用いたシステム開発が行われていた．そのため， 図 2-2 に示す通りエンタープライズ系事業部にてすでに導入されていた基準やツールなどを中心に施策を導入し，開発環境を整備・拡充した．設計工程では，設計プロセス，設計ルールやコーディングルールなどからなる各種設計基準と性能見積もりの手法を導入した．実装工程では開発プロセスや，使用する開発環境，コーディングルールなどからなる開発基準を導入した．テスト工程では，テスト基準と各種テスト手法を導入した．デバッグ工程では，OS トレーサやプロファイラ，OS が提供するリソース使用状況の分析ツールなどからなる解析ツールと，組込みシステムでよく使われるデバッグ対象の基板の制御や，CPU の機能を活用したデバッグのために使う JTAG-ICE などの Linux 向け各種デバッガを導入した．

我々はまた，開発工程全体において，開発している製品の変化点と，変化点における構成要素であるソフトウェア及び関連するドキュメントを管理するための構成管理の基準とツールも合わせて導入することにした．

以降施策の結果と，得た課題について述べる．

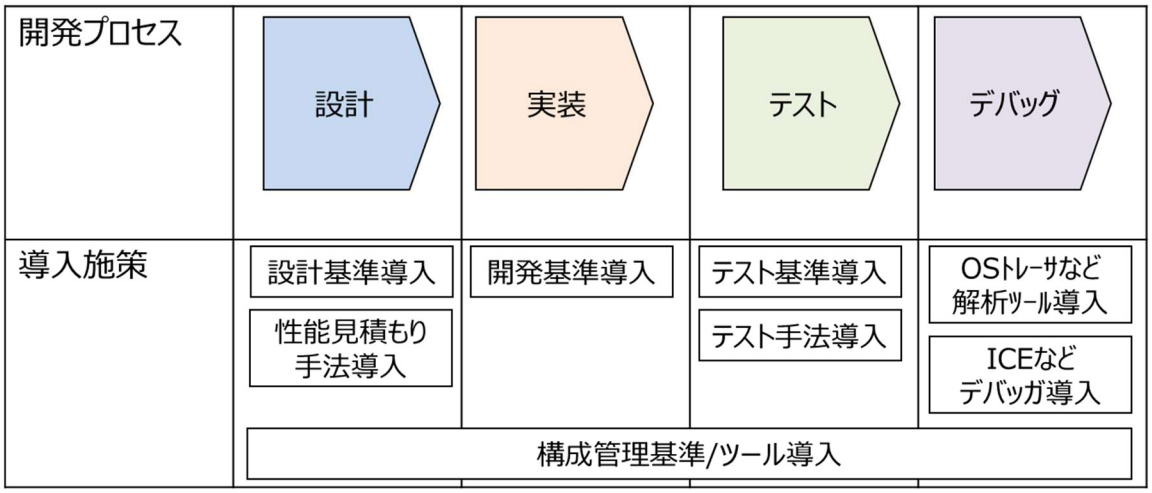

図 2-2 エンタープライズ系システムの基準やツールの導入例

2.4　本研究で取り組む課題

これまで述べた内容を踏まえ，本研究で取り組む課題を 2 点述べる．

2.4.1　　課題 1　デバッグ工程における不具合解決の効率化

2.3.3 に示した各種技術の導入により，H/HR9000 の出荷をした．一方で問題が出た．製品開発の下流工程（テスト工程，デバッグ工程）に導入したツールは効果的であったが，効率が悪かった．そこで我々は実際の製品事例を通して発生した不具合とその対策に関連する情報を調査したところ，再現性の低い（今回，我々は 1 週間= 5 営業日再現しない不具合を再現性が低いと定義）不具合が散見され，その不具合の平均対策期間が 19.8 日（営業日）と，通常の不具合（再現性のある不具合）の 10.7 日（営業日）という対策期間に比べて，2 倍近く長くなっていることが発覚した．その結果，出荷直前の不具合対策に追われていることが判明した．

その分析を進めたところ，大きく 3 つの問題があることがわかった．一つ目は，障害の再現性が低いこと，二つ目は，OS トレーサの結果が膨大で解析対象の特定が難しいこと，三つ目は，解析情報がトレースデータのままでは非常にわかり辛いことである．以下，その三点について説明する．

まず，障害の再現性が低い点について説明する．導入品（OS，ミドルウェア，ハードウェア及びデバイスドライバなど）や複数のアプリケーションからなる組込みシステムを開発している場合，テスト条件の入力だけでは不具合が発生しないことが多々ある．そのため，テスターからすると，不具合が偶然に発生しているように見えてしまう．このような不具合は，テスト対象のアプリ以外のアプリや，導入品ソフトウェア，更にはハードウェアの動作と組み合わさって発生するため，テスターのテスト入力条件だけでは再現せず，不具合の再現条件特定を難しくしていることがわかった．

次に，OS トレーサの結果が膨大で解析対象の特定が難しい点について説明する． OS のトレーサによるトレース結果は，システム全体の解析を可能とし，非常に有効である一方，短時間のトレースでも取得できるトレースの量が膨大になることや，それらのうちの多くのログは解析に不要であるため，真因に到達するまでの問題の絞り込みが困難であった．一例を示すと，前者では 10 秒程度のテストで 5 万行のログ出力があること．後者ではデジタルテレビは 194 タスク，ビデオカメラ（BD-CAM(Blu-ray Disk CAMera)）は 78 タスクが同時実行されるが，解析に影響のあるタスクは 10 タスクもないケースが多い．

最後に，解析情報がトレースデータのままでは非常にわかり辛い点について説明する．OS のトレース情報は，組込みシステムの OS で管理されていたシステムの挙動情報であり，プロセス名と ID やシステムコール番号，割り込み番号などの名前解決がされてない．その結果どのログがどのタスクの情報か，どのシステムコールに関する情報か，どのハードウェアの割り込みに関する情報かが一見ユーザからはわからない．更には，プロセスなどの並列解析ができないと，複数のアプリの組み合わせによる因果関係の解析が難しいが，トレース情報そのままだと，時系列の連続情報にしか見えない．その上で先に述べた通り，解析に影

響のない情報も多数含まれるため，解析のための情報整理に時間がかかっていることがわかった．

そこで本研究では，これらの先行研究・ツールなどを活用・応用しつつ，より効率良く再現性の低い障害を解決する機能について検討・評価する．今回は障害発生時に不具合情報を必ず記録しておくため，LKST を用いた長時間トレースを実現する．そのため，LKST の仕組みを利用して低い負荷で大量の情報を取得する技術について検討する．また，長時間のトレースによって取得した膨大な情報の絞り込みのために，プロファイラを活用した問題点の絞り込みについて検討する．更に，再現性の低い不具合の解析容易化に特化した可視化手法についても検討をする．その内容について 3 章に記載する

2.4.2　課題 2　設計工程における性能検証の効率化

デバッグ工程における不具合解決の効率化を進めていくと，2000 年代後半から性能問題が開発工程の終盤で発見される製品開発プロジェクトが増えてきた．この背景として，機能のソフトウェア化が進み，柔軟な開発が可能になった結果，多くの仕向け先に合わせて機能や性能の最適化がソフトウェアの修正でできるようになったことが挙げられる．例を挙げると，デジタルテレビのブラウジング機能を，各国の放送規格に合わせて変更・追加していた．また，ストレージ製品のキャッシュアルゴリズムを変更することで，機能や性能の最適化を実施していた．

このような開発に対し，我々は図 2-2 に示す通り，設計工程において性能見積もり技術を導入した．導入した性能見積もり技術は，待ち行列理論に代表される統計を用いたものであった．この技術は，主要機能・共通機能では高い精度で予測もでき，成果があった．しかし，上記のような派生開発において増えた開発に対しては，見積もりに十分な情報が無く，高い精度の見積もりが不可能であった．

この様な問題に対し，2.2 で述べた各性能見積もり・検証技術から，有効な技術を検討したところ，モデル検査技術を用いることで，開発する機能の動作の振舞いを利用し，網羅的に性能の検証が可能と考えた．

一方で性能を達成するための実装は，プログラムの中に散在している[45]ことが既にわかっており，差分開発により，前世代のプログラムを流用して新製品のシステムを開発するコンシューマ向け組込みシステムにおいては，モデル検査を導入するタイミングでモデル開発工数が増加するため，モデル検査の導入が困難であることがわかっていた．

そこで，我々は上記の問題を解決するために，設計工程における性能検証の効率化技術を検討し，開発，評価を行った．その内容について 4 章に記載する．

第3章　デバッグ工程における障害解析効率化

3.1　導入

コンシューマ向け組込みシステムの開発には，さまざまなハードウェア，ソフトウェアの組み合わせが用いられる．製品の使用目的，要求される性能，かけられるコストなど要求がさまざまなためである．また近年のハードウェア高性能化に伴い，ソフトウェアによる機能実装が増加している．結果，ソフトウェアの開発規模は増加の一途をたどっている．

このようなハードウェアの高性能化及びソフトウェアによる機能実装の増加により，組込みシステムでは，これまでのような特別なハードウェアを前提としたソフトウェアの実行だけでなく，OS，ミドルウェアの実行や，機能を実現するアプリケーション（以降アプリ）が複数まとめて動作する機会が増えている．結果これまで発生しなかったような，アプリ間のリソース競合による障害などが発生するようになり，その影響評価や実行順序の妥当性検証に工数を要することが増えてきた[54]．

このようにソフトウェア開発工数は増加しているものの，製品コスト削減の要求から，開発期間の削減が求められている[5]．開発期間の削減には，多数の開発チームによる並列開発や，テスト工数の削減，デバッグの効率化が必要である．

我々は，この中でデバッグの効率化に着目し，デバッグツールを見直すことにした．デバッグツールには，動作中のソフトウェアを制御しながら解析をする対話型デバッグツールや，プログラムの動作情報を記録し動作終了後に挙動を解析するトレースツールや，可視化ツールなどがある[55]．ここで，これらのツールを組み合わせ，障害解析全体を効率良く進める方法を検討し，評価することにした．

本論文では，コンシューマ向け組込みシステムの一つであるデジタルテレビ（Digital Tele Vision(DTV)）やビデオカメラ（BD-CAM）のアプリ開発を対象に，ソフトウェアの障害解析における課題を抽出し，それに対する改善手法を提案，実装し，DTV やビデオカメラなどを中心とする組込みシステム 6 製品に適用した結果について述べる．

3.2　組込みシステム向け障害解析環境の課題導出

これまで執筆者所属企業では，DTV やビデオカメラなどをコンシューマ向け組込みシステムの開発を多数行ってきた．その中でも特に，組込みシステムの Linux 化，ネットワーク対応を契機に，ソフトウェアの複雑さは増し，障害解析に必要な時間は増加している．

そこで，我々はこれまで開発してきた DTV とビデオカメラの開発実績をもとに，障害解析におけるどのような作業が開発工数に影響を与えている要因なのかを調査した．その結果は，以下の通りであった．

要因 1)障害の再現性が低く，再現をさせるまでに時間がかかる

要因 2)システム全体の膨大なトレース結果を解析するのに時間がかかる

　　要因 2-1)試行錯誤しながらトレースを取る

要因 2-2)トレースを取れる時間が少ないが，解析者からすると情報量が膨大であり，
解析に時間がかかる

以下それぞれの分析結果について述べる．

3.2.1 要因 1)に関する分析

コンシューマ向け組込みシステムの開発では，障害の再現性が低く，障害解析が進まないケースが散見される．例えば，我々が 2007 年に製品開発をした BD-CAM の障害 2600 件のうち，19 件（0.7%）の障害が 1 週間（5 営業日）以上再現しなかった「再現性の低い障害」に該当する障害であった．再現性の低い障害は，件数は少ないが解決しないと製品出荷ができない内容で，開発におけるボトルネックとなっていた．

この 19 件は，再現性のある障害に対し，平均対策期間で長い時間を必要とした．再現性のある障害の対策期間が平均 10.7 日であったのに対し，再現性の低い障害の対策期間は平均 19.8 日と，1.85 倍の対策期間を要した．

上記再現性の低い障害の内容を解析した結果，原因は当該アプリ以外のアプリや，アプリレイヤ以下の OS や，デバイスドライバなどに原因があった．そのため，障害を発生させるためには担当者の開発したアプリ以外のさまざまなシステムの動作条件を揃える必要があり，テスト担当者による再現が困難であった．結果，一度不具合が発生した際に解析に十分な情報が取得できて無い場合，解析に必要な情報の取得に時間がかかり，解析のボトルネックとなっていた．以上より，以下の課題を得た．

課題 1:再現性の低い障害の情報を障害発生時に確実に記録できるようにする必要がある．

ここで確実に記録するとは，記録内容の正確さを意味するものではなく，「テスト実施時に常時もれなく情報を記録する」という意味である．

3.2.2 要因 2)に関する分析

組込みシステムの障害解析において，我々は 2006 年出荷の DTV 開発のタイミングで，OS トレーサである LKST[32][33]を導入した．それにより，OS を介して動作するシステム全体の一元的な解析を可能化し，2.4.1 に示したアプリレイヤ以外で発生する障害などに対応可能にした．一方で，OS トレーサはアプリ開発者には十分に普及しなかった．

その原因を調査するため，DTV 開発チーム及び BD-CAM 開発チームに対し，使用しない理由をヒアリングした結果，以下の様な回答を得た．

a)LKST のトレース結果が，膨大でありどこから見てよいかわからない

b)LKST のトレース結果は膨大である一方，長時間のトレースができないため，2.4.1 に述べたような再現性の低い障害の解析情報収集に時間がかかる

c)障害情報収集後に，アプリ名称，割り込み名称，システムコール名称といった解析に必要な情報を，トレース結果に引き当てる必要があるが，その作業が難しいうえ，量が膨大

順にヒアリング結果について説明する．a)については，LKST はカーネルのメモリ領域に

トレース結果を保存し，その結果をテスト完了後にコマンドを使い抜き出す．当時は組込みシステムのメモリ領域のうち，LKST のトレース結果保存に使える領域は多くて 10MB 程度であった．このメモリを使い OS のトレースを取ると，システムの動作状況に依存するが，10 秒弱，5 万行～6 万行程度のトレースが出力される．この量は，要因 2-2)に示す通り，アプリ開発者にとっては非常に膨大である．

次に b)については，解析するデータ量は膨大である一方，10 秒という時間の中で障害を再現させることは難しく，要因 2-1)に示す通り，試行錯誤を繰り返し，何度もトレースを取る必要があった．そのため，課題 1 同様，再現性の低い障害の情報を障害発生時に確実に記録できるようにする必要がある．一方で，DTV は 194 タスク，BD-CAM は 78 タスクが同時に実行されるが，その多くは解析対象の不具合に関係無いことが多い．以上より，以下の課題を得た．

課題 2:アプリ開発者からすると，LKST のトレース結果が膨大．そのため，解析対象を絞り込む必要がある．

最後に c)については，DTV は 194 タスク，BD-CAM は 78 タスクが同時に実行され，id で管理されている．しかしその id は OS の起動や，アプリの動作のタイミングで OS によって番号が割り当てられるため，開発者の認識しているタスク名と一致させるにはトレース結果を名前解決する必要があった．同様に，アプリの動作のトリガとなるハードウェアの情報や，OS とのやり取りをおこなうシステムコールの情報も，システム上は名前ではなく，システムの動作時に割り当てられる id で管理されるため，人間が解析するには，名前解決をしないと理解が難しい．さらには，文字の情報としてトレースを直接解析することは困難で，可視化など理解容易化が必要である．以上より，以下の課題を得た．

課題 3:開発者が理解しやすい形で解析情報を提供する必要がある．

3.2.3　関連研究

本項では，3.2.2 までに導出した課題に対する，既存の研究について説明する．組込みシステムの障害解析では，製品が市場投入されるまでの時間が問題となっており，デバッグフェイズの最適化が必要である．そこで実行トレースを用いた分析が強力であることは既にわかっており[35]，多くの研究がされている．

例えば，トレース分析結果を可視化する研究は様々あり，Java で実装されたプログラムの実行履歴を可視化する JIVE をはじめ[36]，マルチプロセッサ環境で動作するソフトウェアデバッグ用のトレース可視化ツール TLV[37]，リアルタイム性を持つ組込み機器向けに開発されたオープンソースの可視化ツールである Timedoctor など，多くの報告がされている．また，組込みシステム向けを含め多くのトレースツールや[33] [38] [39]，プロファイルツールなど解析ツールの開発もされている．さらに，大量に取得したデータの効率化については，パターン認識を用いて周期性のある振る舞いの情報を抽出する研究がおこなわれている[35]．

一方で，これらの報告は，トレース自体，可視化自体を対象に進めているものが多いが，我々が課題に挙げている再現性の低い障害の解決には，十分に考慮されていない．例えば，トレースを取得する点においては，再現性の低い障害が発生した際のトレース情報を，障害発生時に確実に記録しなければならない．またその際に取得した膨大な情報を効率良く解決しなくてはならないが，このような情報は周期性が無い場合も多いため，検出するには別のアプローチの検討が必要と考えらえる．

そこで本研究では，これらの先行研究・ツールなどを活用・応用しつつ，より効率良く再現性の低い障害を解決する機能について検討・評価する．

3.3 障害解析環境の効率改善機能の要件

本研究では，3.2 で示した課題 1～3 それぞれを解決するため，以下に示す要件 R1～R3 を満足するソフトウェアを開発することを目標とする．

R1:長時間トレース機能の実現

R2:膨大なトレース結果から解析対象を絞り込む機能の実現

R3:システム全体のトレース結果を取得し，アプリ開発者でもわかる形式に変換する機能の実現

R3-1:時系列上にプロセス毎にトレース結果を表示できる機能

R3-2:時系列上にハードウェアからの割り込み処理の開始契機を割り込み信号ごとに表示できる機能

R3-3:システムコールの発行タイミングをシステムコールごとに記録できる機能

以降，本章ではそれぞれの詳細について説明する．

3.3.1 R1：長時間トレース機能の実現

本要件は，3.2.1 で示した課題 1 を解決するための要件である．再現性の低い障害の解析を効率良く解析するためには，偶然障害が再現した際に確実に情報を記録していることが求められる．また，3.2.2 に示した通り，LKST は，カーネルが確保したメモリ領域の一部にトレースを記録する．そのためハードウェアリソースに制限のある組込みシステムでは，記録時間が少なくなり再現性の低いバグの記録や，長時間の耐久テストなどでは使用できない．そこでトレース結果をメモリ領域に記録せず，代替装置に記録できる機能の実現を目標とする．

3.3.2 R2：解析対象を絞り込む機能の実現

本要件は，3.2.2 で示した課題 2 を解決するための要件である．2.4.1 に示した通り， OS トレースは，OS を介して処理される全てのプロセスやハードウェアとのやり取りの情報に関係するイベント情報を取得する．そのため，取得できる情報の量は膨大であり，問題点を絞り込まないと解析の効率が下がる．そのため，実行時に CPU を多く利用しているものな

ど，何らかの観点から解析対象を絞り込める機能の実現を目標とする．

3.3.3 R3：形式変換機能の実現

本要件は，3.2.2 で示した課題 3 を解決するための要件である．2.4.1 に示した通り，LKST は OS で実行されるイベントについてトレースを取る．そのトレース結果解析の簡略化を目的に，アプリ開発者でもわかる形式に変換する．そのため，以下に述べる 3 つの機能の実現を目標とする．

3.3.3.1 R3-1：時系列上にプロセスごとにトレース結果を表示できる機能

OS 上でアプリはプロセスの単位で実行される．ここで OS トレーサは，プロセスの切り替えや，システムコールの発行，セマフォの確保といったアプリの動作に関係する様々な情報を記録している．それらの結果は，Process ID (PID)などプロセスを識別する単位で記録されている．一方，プロセスは OS がプロセスを起動する際に OS によって PID が振られるため，システムテストなどトレースを取得するたびに PID の番号が変わってしまう．そこでテスト実行時における PID の情報をテスト実行時に取得し，その結果を解析時に突合して解析を可能にすることを目標とする．そのため，PID の情報とプロセス名称の情報を突合できるように，/proc/[PID]/以下の情報を一括取得し，PID 番号とプロセス名を突合して解析を可能とすることを目標とする．最後に，取得した各プロセスの情報から，CPU を占有している時間を把握可能にすることで，プロセス間のやりとりなどが視覚的に把握可能となるため，その実現も目標とする．

3.3.3.2 R3-2:時系列上にハードウェアからの割り込み処理を割り込み信号ごとに表示できる機能

組込みシステムは，ハードウェアを用いた制御をおこなうことが多く，ハードウェアからの動作指示を契機にアプリが起動することがよくある．そのため，割り込みの情報とアプリの動作の因果関係を把握できる必要がある．よって本研究では，割り込み情報を，割り込みをかけるハードウェアの割り込み信号ごとにプロセスの情報と同じ時間軸上で解析できる機能の実現を目標とする．

3.3.3.3 R3-3:システムコールの発行タイミングをシステムコールごとに記録できる機能

組込みシステムは OS の機能を利用して，ハードウェアを操作したり，OS のリソースを利用したりする．ゆえに，アプリからシステムコールを発行してハードウェアを制御することや，やり取りをすることなどがよくある．そのため，システムコールの発行とアプリの動作の因果関係を把握する必要がある．よって本研究では，システムコール発行情報をプロセスの情報と同じ時間軸上で解析できる機能の実現を目標とする．

3.4　障害解析環境の効率改善機能の設計

3.4.1　　障害解析環境の効率改善機能の全体像

今回開発する機能群の全体像を，以下図 3-1 に示す．

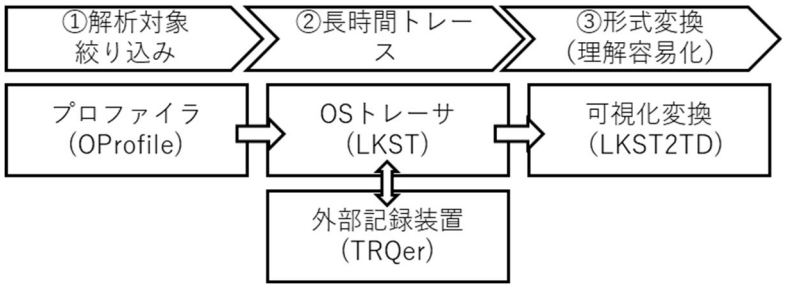

図 3-1　今回開発する機能群の全体像

今回開発する内容は，要件 R1 を満足する②長時間トレース機能，要件 R2 を満足する①解析対象絞り込み機能，要件 R3 を満足する③形式変換機能の 3 つである．

これらの機能を用いた作業推進方法は，以下の通りである．テスト前に，①解析対象絞り込み機能を使い，プロファイリングを実施する．次に，②長時間トレースを使い，テスト実施中のトレースを可能な限り取得する．最後に③形式変換機能を使い，内部情報を可視化し，①で得たプロファイル結果を参考に，まずは負荷の高いタスクなどを中心に，情報を選択し原因分析を進める．ただし，原因分析がうまく進まない場合は，分析対象の情報を変更して解析を継続する．以降，これらの機能の設計について説明する．

3.4.2　　設計の具体的な進め方

今回開発する機能の設計では，主に以下 2 点を実施した．

・既存手法の調査，分析と評価

・経験者・有識者の知見の抽出とそれを用いた改善

前者は，既に開発で使われている機能や，広く市場で使われている機能，学術的に検討されている手法などを製品カタログや論文などを用いて調査し，有望なものを試験導入・評価した上で改善点を洗い出した．

後者は，前者で洗い出された改善点に対し，製品開発経験者を中心にディスカッションを通して過去の製品開発において獲得した知見の抽出，知見に基づいて改善手法を立案し設計に反映した．ディスカッションは，製品開発部門（製品ごとの開発部署，左記開発部署の横串を通すプラットフォーム開発部署）から，プラットフォーム開発部署のエキスパートエンジニア１名，各製品開発部署から中堅～若手エンジニア数名を選出，また研究部門のエキスパート研究者と執筆者が参加し実施した．

3.4.3　　長時間トレース機能の実現

今回の長時間トレース機能は，再現性の低い不具合の解析をするため，特定のタスクのト

レースをするのではなく，システム全体の解析をするために，OS トレーサのイベントを，トレースできる機能を再利用する[56]．解析に必要な情報は LKST で取得するものとし，LKST の記録を外部に保存する仕掛けを考える．

ここで 2.2.3 に述べている通り，既存の OS トレーサや，JTAG-ICE などを用いた OS のトレース方式は，コンシューマ系組込みシステムにおける長時間トレースの実現が困難であった．具体例として従来の LKST を用いた長時間トレースの実現について説明する．LKST で長時間トレースをするには，図 3-2 に示す通り，メモリ上に複数面のバッファを確保し，そのうちの 1 面のバッファでイベントが発生するごとに内容を記録，バッファを使い切った後バッファを切り替え（①）トレースの継続をする．並行して元のバッファの記録結果を外部媒体にファイルとして出力する（②）．これらを組み合わせることで，長時間トレースを実現している．しかし，組込みシステムに搭載されるメモリは容量に制限がある．また，出力先のストレージデバイスも無い場合が多い．さらには，ファイルへの出力は，ファイルを格納するためのストレージデバイスへの書き込み負荷が，テスト対象のシステムの動作を変更してしまう可能性がある．

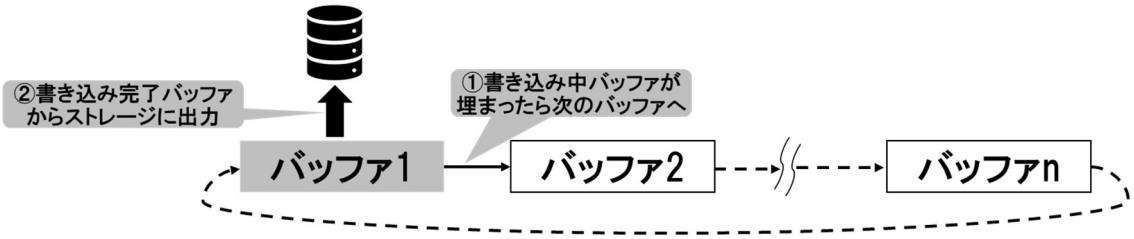

図 3-2　従来の LKST を用いた長時間トレースの仕組み

そこで，提案手法では図 3-3 に示す通り，開発対象のコンシューマ向け組込みシステムが外部にデータを出力する外部出力バスを持っていることを前提とし，外部バス経由で記憶容量が大きい外部記録装置に低負荷で OS の挙動情報を出力し，長時間記録する方式を提案する．

今回は，LKST のトレース結果を送出するための外部出力モジュールをデバイスドライバとして実装し，挙動情報を LKST のバッファ領域外に出力可能にした．この出力モジュールは，LKST のイベントハンドラを用い，トレース内容をカーネル領域のメモリに記録する処理を Hook して，本来メモリに記録する 32bit×4 のデータ列を 16bit×8 のデータ列に分解し，16bit ずつ外部バスに出力する．外部バスには，HDD を保有する長時間トレースハードウェア（今回は当時横河デジタルコンピュータ/現 DTS インサイトの TRQer‡‡‡） を接続し，バスにイベントの出力が流れる度，長時間トレースハードウェアに記録する方式とした．

‡‡‡ TRQer は DTS インサイトの登録商標です

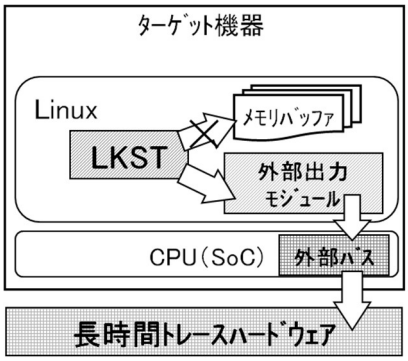

図 3-3　長時間トレース機能

3.4.4　解析対象絞り込み機能

3.2.2 に示した通り，本研究の対象であるコンシューマ向け組込みシステム上で動作するタスクの数は多いため，トレース結果を解析する際に全てを解析することは困難である．そこで解析を絞り込む方法を検討した．検討した方式は，1)対象とするタスク数を制限する方式，2)対象とする時間を制限する方式の 2 点である．検討の結果，我々は 1)対象とするタスク数を制限する方式を採用した．その理由について以下に述べる．

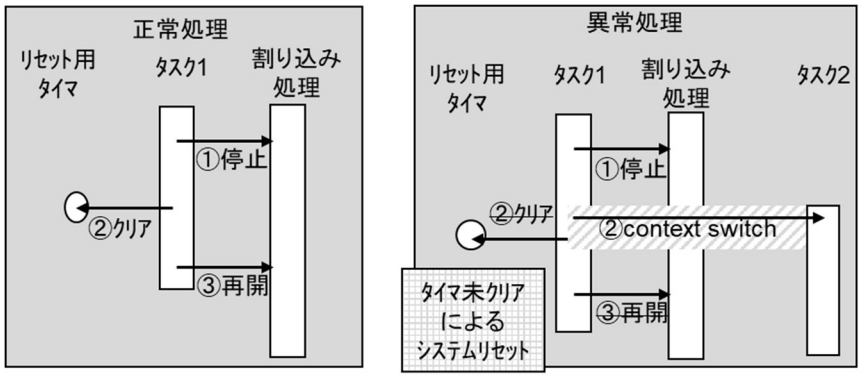

図 3-4　時間による制限をかけにくい障害の例

これまでの開発において LKST の仕組み上，カーネル内部で確保できるメモリの制約により，時間的な記録制限を受けることが多かった．しかし，図 3-4 に示すようなタイマ未クリアにおけるシステムリセットが発生する障害の場合，タスク 1 が割り込み処理を停止してから，リセット用タイマをクリアするまでの時間が長い場合，2)方式のように時間で解析制限をかけると，原因箇所のトレース結果が新しい挙動情報に上書きされてしまい，解析ができない可能性が有ることがわかっていた．同様に障害の原因発生時刻と顕在化時刻の間隔が長い事例には，メモリーリークや排他処理，優先度変換処理など多数考えられる．そのため，今回は 2)対象とする時間を制限する方式は採用しないことにした．

1)対象とするタスク数を制限する方式は，3.2.1 に示したような，原因がデバッグ中のア

プリ以外の部分にあるような障害に対して有効であることが経験としてわかっていた．上記障害は，我々の経験ではリソースの競合（例：ロック，メモリなど）や，ハードウェアの動作に起因する場合が多い．そのような不具合の原因となるタスクと，顕在化するタスクは同時期に並行処理されていること．また，障害発生時には全ての動作が高い頻度で動作しているわけではなく，リソースを確保しようとしているタスクや，特定のハードウェアに紐づくタスクが高い頻度で動作していることがわかっていた．そのため，高い頻度で動作するタスクに解析対象を絞り込み，制限することで，効率よく解析ができる．以上より今回は，1)対象とするタスク数を制限する方式を検討することにした．具体的にはCPUの使用率に着目し，動作していないタスクを取り除き，動作頻度の高いタスクから優先的に解析できるようプロファイラを用いて解析対象を絞り込むことにした．

次に適用するプロファイラについて述べる．本研究着手時点では，OProfileがLinuxでは標準的に使われていたため，本研究ではOProfileを使った絞りこみを行う．OProfileはLinux向けプロファイラの一種であり，OS上で動作する全てのプログラムのプロファイル情報を取得するプロファイラである．OProfileはハードウェアタイマやソフトウェアタイマを使用し周期的に割り込みを発生させ，その割り込みによって，Exception Program Counter(EPC)レジスタに退避されたProgram Counter(PC)の値を取得する．さらに，取得したPCの値を，アプリ，関数などの単位で統計的に解析し，ユーザに提示する．ユーザはOProfileの解析結果を基に，実行される頻度の高いアプリ名や関数名を知ることができる．また，分析データをファイルに保存可能であるため，長時間のデータ採集に適応可能なこと，ソースコードに改変を加えることなくプロファイリングが可能という特徴を持つ．

OProfileのデータ収集機能は，図 3-5に示すデータをサンプリングする割り込みハンドラ部分からなるOProfileモジュールと，ユーザインタフェースであるOProfile制御・解析コマンドと，デーモンプログラムとして常駐し，カーネルからプロファイル結果を取得し，プロファイル結果がどのプログラムのものかを引き当ててデータとして保存するOProfileデーモンからなる．

OProfileはDTV開発時には，DTVに採用されていたMIPS§§§アーキテクチャ向けMontaVista**** Linux3.1(Kernel 2.4)に対応していなかったため，必要な機能を移植することにした[57]．当時既にKernel2.6のMIPS向けにはOProfileが開発されていた．一方で，i386及びia64アーキテクチャ向けには，2.4x,2.6xともにOProfileが提供されていたため，我々はi386/ia64向けの2.4向けOProfileを2.6向けのOProfileと比較し，カーネルバージョンの違いでどのようなプログラムの差分があるかを調査した．その結果，図 3-5に示したOProfileモジュール内において，3つの実装が異なることがわかった．1つ目は，タイマユニット関連でCPUのキャッシュヒットミスの検出方法の実装，2つ目は割り込み間隔の設定に伴うハードウェアタイマの設定の実装，3つ目はアプリがライブラリをダイナミッ

§§§ MIPSはMIPS Technologies, Inc.の登録商標もしくは商標です．
**** Monta VistaはMontaVista Software LCC社の登録商標です．

クリンクする際に発生するシステムコールの情報を収集するために，システムコールをフックして情報を収集している箇所の実装であった．我々は，上記 3 か所を 2.6 の MIPS 向け OProfile から移植した[57]．また，2.4 系のカーネルと，2.6 系のカーネルでは，デバイスドライバの I/F が変更されていることから，そこの部分の対応も併せて行った．

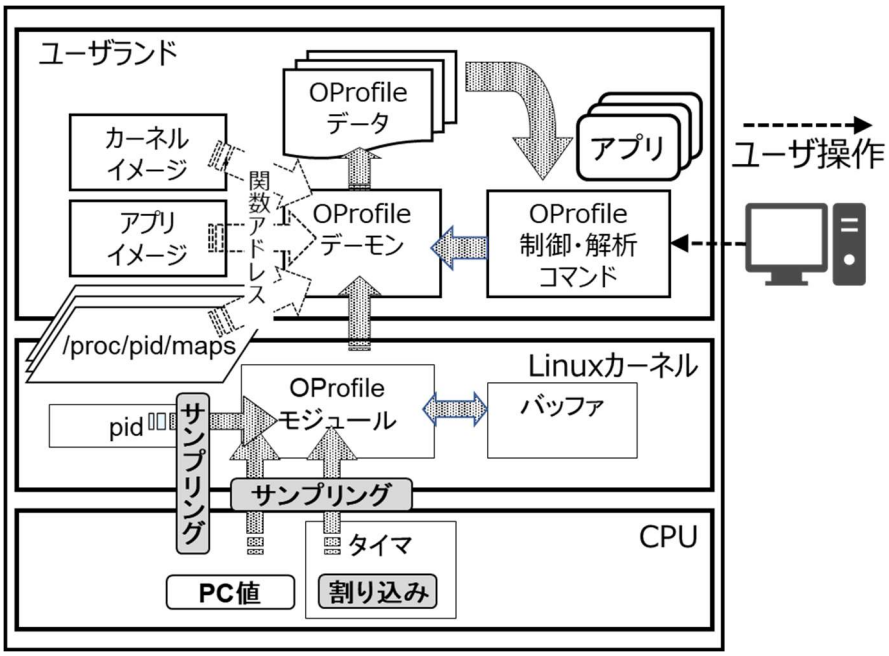

図 3-5　OProfile 概要

3.4.5　　形式変換機能

形式変換では長時間トレース機能で保存された LKST のトレース結果を，3.3.3 に示した要件で出力する．今回はプロセスや割り込み，システムコールごとに名前解決をしつつ，OSS の可視化ツール TimeDoctor で出力できる変換を行う．

LKST で取得できるイベントは以下表 3-1 に示す 118 のイベントからなる．ここで，今回#1 に示したプロセス管理のイベント，#2 に示した割り込み関係のイベント，#4 に示したシステムコールイベント，その他のイベントに分け，それぞれプロセス名，割り込み名，システムコール名を引き当て，可視化する．上記イベントの取得・可視化を中心とする目的は，本研究着手以前の LKST 活用の知見に基づくものである．組込みシステムの開発においては，リソースの競合やハードウェア起因の問題が多く発生していたため，個々のプロセスの動作状況の情報に加え，リソースの確保など OS の機能を利用する情報（システムコールの情報），定期的な割り込みの発生（割り込み間隔の確認）や，タスク動作の要因となる割り込みの確認をするための割り込み情報，OS の各種イベントの情報を取得することで，問題の切り分けが可能になる．特に，リソースの競合になる場合は，問題が顕在化する前に，他のタスクが必ずリソースにアクセスをしているため，上記取得情報を，同一時系列上に併記可視化することで，原因タスクと顕在化タスクの判別が容易になる．

上記知見を踏まえ，以下図 3-6 に示す結果取得処理の順でトレース結果及び関連情報を取得する．まず①トレース停止処理で LKST を停止する．次に②プロセス情報取得処理で/proc/[PID]/以下の情報（例えば cmdline 情報）などを取得し，PID とプロセス名を対応付けたファイルを出力する．次に③割り込み情報取得処理では/proc/interrupts 情報を取得し，割り込み番号と割り込み名称を対応付けたファイルを出力する．最後に④システムコール情報取得処理では，/asm/unistd.h 情報を取得し，システムコール番号とシステムコール名称を対応付けたファイルを出力し，終了する．以上の処理により，解析に必要な情報を取得する．

表 3-1　LKST で取得できるイベント

#	Category	Event 数	備考
1	Process management	13	PROCESS_CONTEXTSWITCH,WAKEUP, SIGSEND など
2	Interrupts	10	INT_HARDWARE_ENTRY,TASKLETHI_ENTRY など
3	Exceptions	6	EXCEPTION_ENTRY,EXIT など
4	System calls	2	SYSCALL_ENTRY,EXIT など
5	Memory Management	15	MEM_SWAPOUT,SWAPIN,MALLOC など
6	Networking	5	NET_PKTSEND,PKTRECV など
7	SysV IPC	11	SYSV_IPC_SEMOP など
8	Locks	8	LK_SPINLOCK,WRLLOCK など
9	Timer	5	TIMER_RUN,ADD など
10	Oops	1	OOPS_PGFAULT
12	Others	4	O_PORTIN,PORTOUT など
13	Page	18	PAGE_ALLOC_ENTER など
14	IPv4	5	NET_V4RTIN_ENTER など
15	LKST	15	LKST_INIT,BUFF_SHIFT など
合計		118	

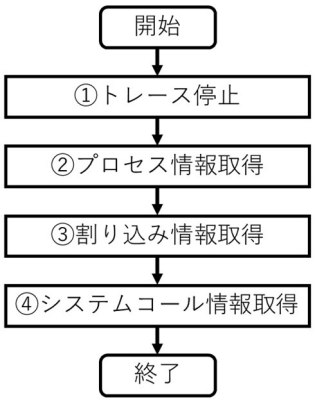

図 3-6　結果取得処理

上記した結果取得処理の後，TRQer から取得した LKST トレース結果と合わせ図 3-7 に示した形式変換処理（LKST2TD）を行う

①のトレース文結合処理で，TRQer の仕様に合わせて保存されている LKST 情報を，元の LKST 形式の情報に変換する．具体的には，LKST1 イベントの情報が表 3-2 に示す 10 個の TRQer メッセージで出力される．10 個を一つのトレース文に統合する処理を行う．

次に，統合された各トレース文を 1 行ずつ取り込み，②イベント（LKST イベント）とのマッチングをとり，各イベントに合わせた処理を行う．プロセス管理イベントの場合，③プロセス切り替え変換処理に遷移し，結果取得処理で取得したプロセス情報を用いてプロセス ID 毎にテーブルを作り，プロセスが実行状態・停止状態に遷移したかを時間情報と合わせて保存する．同様に割り込みイベントの場合，④割り込み変換処理に遷移し，結果取得処理で取得した割り込み情報を用いて割り込み番号毎にテーブルを作り，割り込み発生時刻と終了時刻，割り込まれたタスク名を合わせて保存する．また，システムコールイベントの場合，⑤システムコール変換処理に遷移し，結果取得処理で取得したシステムコール ID 毎にテーブルを作り，システムコールの発行時刻とシステムコール名称を合わせて保存する．これらの処理を，全てのトレース文が終了するまで行い，可視化ツールのフォーマットにあわせて出力する．

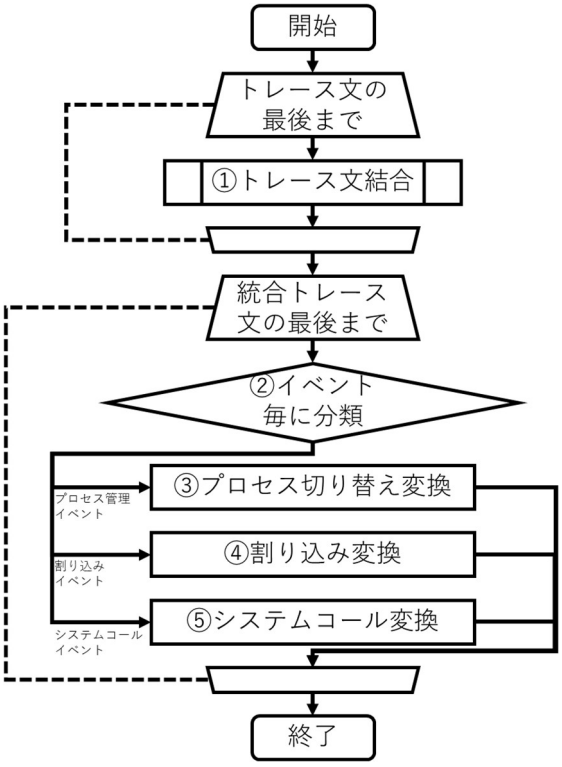

図 3-7　形式変換処理

表 3-2　TRQer に保存したトレース結果

#	TRQer ID	内容
1	A	LKST event ID
2	1	第 1 パラメータ上位 16bit
3	2	第 1 パラメータ下位 16bit
4	3	第 2 パラメータ上位 16bit
5	4	第 2 パラメータ下位 16bit
6	5	第 3 パラメータ上位 16bit
7	6	第 3 パラメータ下位 16bit
8	7	第 4 パラメータ上位 16bit
9	8	第 4 パラメータ下位 16bit
10	9	イベント発生時刻

3.5　実装と評価

3.5.1　　長時間トレース機能

本研究では 3.4.3 で提案した長時間トレース機能の実用性評価を，汎用の評価ボードを用いた原理試作により実施した．これは，トレースデータを外部出力するためのハードウェア

改造や治具の作成が必要な DTV やデジタルビデオカメラなどの実製品と比較して，あらかじめ外部バスを備えた評価ボードを用いて原理試作を行うことで，開発工数を抑制し提案手法の評価・問題点抽出を素早くできるためである．

3.5.1.1 評価環境と方法

本評価においては，汎用評価ボードとしてアットマークテクノ社 Armadillo†††-300 を用いた．また，評価対象に対して外部から負荷をかけるための環境として，Armadillo とローカルネットワークで接続した Linux PC に搭載した Apache‡‡‡ Bench を用いた．

本評価では，LKST を用いた長時間トレースを実施している際に，外部から評価対象に対し負荷をかけ，CPU 負荷と記録時間を測定する．具体的には，Armadillo 上で web サーバ（thttp）を動作させ，そこに Apache Bench からアクセスを発行し，負荷を与える．使用したコマンドラインは“ab –n 30000 –c5 アクセス先 URL”である．CPU の使用率は，Snap Gear 社の Greg Ungerer 氏が OSS で公開している cpu.c を用いて計測した．cpu.c は /proc/stat 以下にある CPU の動作状況から，システム時間，ユーザ時間等を算出するツールである．

3.5.1.2 評価結果

CPU 使用率による負荷の評価結果を，以下表 3-3 に示す．

表 3-3 CPU 負荷計測結果

		CPU 使用率	内 System
LKST 無し		92.00%	36.80%
従来方式	LKST メモリ記録	92.45%	42.32%
	LKST ファイル記録	95.02%	34.47%
提案方式		92.23%	56.94%

評価項目のうち，CPU 使用率を用いて長時間トレース自体の負荷について評価する．System の比率は，CPU 使用率における負荷の要因の評価分析に用いる．

LKST 無しの場合に対し，従来の LKST メモリ記録，3.4.3 に示した従来の長時間トレース手法である LKST ファイル記録と，提案方式を比較する．手法のうち，LKST ファイル記録の CPU 負荷が最も高く，LKST 無しの場合と比べ，CPU 使用率が 3.02%上昇している．それに対し，提案方式の場合 0.23%の上昇と，それ程 CPU 負荷に影響が出てない．また，LKST メモリ記録より 0.22%低く出ている．この結果より，従来の長時間トレース手法である LKST ファイル記録と比較し，低い CPU 使用率で長時間トレースできることのめどをつけることができた．

---

†††Armadillo は株式会社アットマークテクノの登録商標です
‡‡‡Apache は Apache Software Foundation の登録商標または商標です

次に，LKST メモリ記録に対し，提案方式の CPU 使用率が低く出た原因を分析する．LKST は OS に組み込まれるため，CPU 使用率のうち OS が CPU を使用した比率である System の項目を用いる．提案方式が，LKST 無しの場合に比べ 20.14%，LKST メモリ記録方式と比べ 14.62%上昇している．理由は，提案方式がイベント発生毎に外部バスにトレース結果を出力するための，カーネルモジュールが頻繁に動作しているからである．以上より，提案方式は LKST メモリ記録より，長時間トレース自体の負荷は上昇していると考えられる．これに関連し，LKST ファイル記録方式の System の比率が提案方式より低い原因を分析する．理由は，メモリからファイルに結果を出力する際，ユーザランドのデーモンプログラムが動作し，メモリからストレージに結果を出力しているため，OS の動作比率が下がっているからである．ここで，デーモンプログラムを含めた CPU 使用率は LKST ファイル記録方式の方が提案方式より高いため，負荷も高いと考えられる．以上より，提案方式は LKST ファイル記録方式よりも低負荷で長時間トレース可能と判断した．

最後にトレース時間の評価結果について述べる．上記環境でヒートランをした結果，トレースデータを 190GB，実行時間で約 105 時間程度の記録を確認できた．また，実際の障害としては，1000 秒程度のトレース結果を用いてメモリーリークの解析を行ったのが最長の結果となった．

3.5.2　解析対象絞り込み機能

3.4.4 で提案した解析対象絞り込み機能をテストするため，DTV（HR-01）にて発生した，図 3-4 に示すタイマ制御処理障害の解析を通して評価した．その結果を以下表 3-4 に示す．

ここで示す通り，実際に動作していたタスク数は 194 あったが，OProfile にて障害発生時刻周辺で CPU を占有していたタスクを上位 10 まで絞り込み，解析を行った．これにより，ソースコードのチェックする規模も 248 万ステップから 12 万 8 千ステップ程度まで削減することができ，解析範囲を約 1/20 まで減らすことができた．

表 3-4　絞り込みの結果

	調査タスク数	調査ステップ数
従来方式	194	2,486,836
提案方式	10	128,180

3.5.3　形式変換機能

3.4.5 で提案した形式変換機能を，DTV（HR-01）と，トレースログ可視化ツールである TimeDoctor を使い評価した，その結果例を以下図 3-8 を用いて示す．

形式変換機能では図 3-7 に示した通り，③プロセス切り替え変換，④割り込み変換，⑤システムコール変換を行っている．それぞれの変換結果は図 3-8 の①～④に出力される．図 3-8 の①は，カレントプロセスの切り替えを TASK としてタイムライン上にプロセス名と対応付けて表示している．②は割り込みの開始・終了を ISR(Interrupt Service Routine)と

してタイムライン上に割り込み名称と対応付けて表示している，③は LKST イベントのマーキングであり，例えば④で表示するシステムコールイベントの発生タイミングがわかる．④はシステムコールの開始と終了を AGENT にシステムコールとしてタイムライン上に表示している．システムコールを発行したプロセス名と一緒に開始から終了までの時間を把握することができる．このように，トレース結果を形式変換してアプリ開発者が理解しやすい形式へ変換することができた．

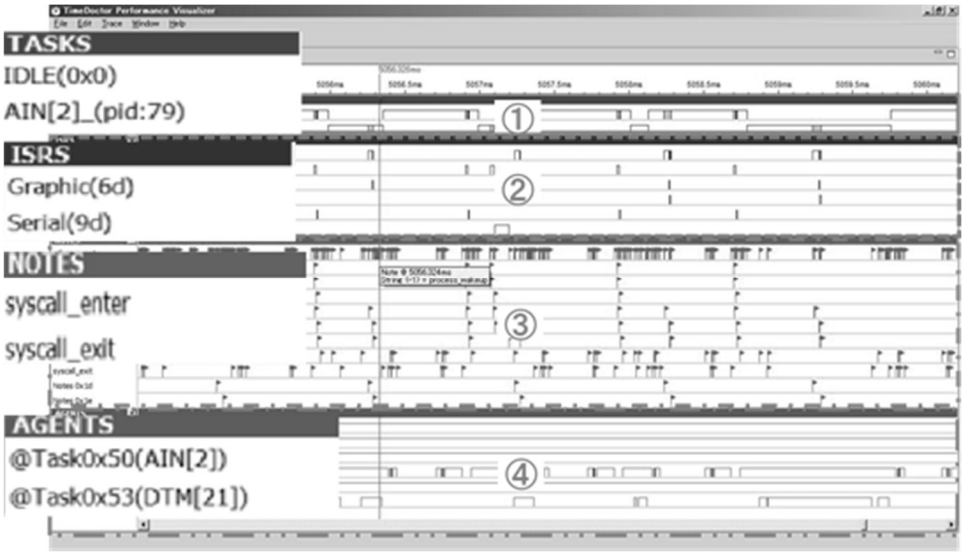

図 3-8　形式変換結果例

3.5.4　　提案した障害解析環境の製品開発適用結果と考察

提案した環境は，当時の DTV 開発（日本・欧州・北米向け各製品）時の解析困難な障害（5 営業日以内に担当者が解決できなかった障害）60 件に対し，開発をしながら試行的に適用することができた．そのうち図 3-4 に示すタイマ処理制御障害を一例に，提案手法においてどのような工数短縮結果があったかを表 3-5 に示す．実際の対策工数は，6.09 日短縮することができた．短縮できた箇所はトレース取得及び解析の日数で，7.5 営業日を，トレース取得・解析を 1 日へ短縮することができた．一方で絞り込みのための 0.41 日分は工数が増加した．

表 3-5　工数短縮結果

	所要時間	内訳
従来方式	8 日	トレース取得・解析 7.5 日，開発・テスト 0.5 日
提案方式	1.91 日	絞り込み 0.41 日，トレース取得・解析 1 日，開発・テスト 0.5 日

更にこれら環境を情報系組込み機器 6 製品（BD-CAM，アンドロイド携帯電話，液晶プロジェクタなど）24 件の障害の解析困難な障害に実適用し，そのうち 18 件で効果を確認した．そこで，本障害解析環境がどのような障害に有効であったかを分析した．その結果を以

下に示す.

発生した障害の顕在化箇所を表 3-6 を用いて説明する．それらの障害の顕在化箇所は，テスト時においてほぼアプリの障害として検出されたものであった.

表 3-6　障害顕在化箇所

#	障害顕在化箇所	件数	比率
1	アプリ	17	94.44%
2	他	1	5.56%

実際に障害の原因となった箇所を表 3-7 を用いて説明する．その多くは#1,2 にある様に OS，ドライバといったプラットフォーム部分で起きた障害であった．また，#3 アプリにおいても，障害が顕在化したアプリ以外の，別のアプリが原因であった.

表 3-7　障害原因箇所

#	障害原因箇所	件数	比率
1	OS	5	27.78%
2	ドライバ	7	38.88%
3	アプリ	3	16.66%
4	ミドル	1	5.56%
5	ツールチェーン	1	5.56%
6	データ	1	5.56%

以上の結果より，本障害解析環境は，障害発生個所と顕在化箇所が別にあるような障害に対して有効であることが明らかになった．また，それらの多くは OS やドライバ，ミドルといったアプリ以外の導入品に含まれる障害であった.

今回対策した障害解析の内容の汎用性について，表 3-8 を用いて説明する．最初に，どのような不具合に適用できたかその分類を示す．最も多く適用できた不具合は，性能（処理遅延）に関する不具合である．これが全体の 12 件，66%強を占めている．要因としては，他タスクの負荷高騰に伴う対象タスクの処理遅延や，自タスク/他タスクの優先度設定ミスによる処理遅延，自タスクの無駄処理による処理遅延の検出といった内容であった．次に，システムのクラッシュと再起動がそれぞれ 2 件ずつ，11%強となった．要因としては，クラッシュの場合，カーネルのリンクレジスタ書き込み不正や，Floating Point Unit(FPU)レジスタ退避のバグであった．再起動の場合は，割り込み禁止区間を設定したタスクに正しく設定が反映されてないことに起因する Watch Dog Timer (WDT) によるシステムリセットや，別々のアプリがそれぞれにリセットタイマを設定することによる互いに想定しないタイミングでの WDT によるリセット発生であった.最後にフリーズと起動失敗が 1 件ずつあり，こちらの要因は mutex_lock の処理不正並びにドライバサスペンド時の処理フラグ未クリアによる，リジューム失敗であった.

表 3-8　障害内容分析結果

#	不具合分類	件数	比率	不具合要因
1	性能 （処理遅延）	12	66.67%	他タスクの負荷高騰 優先度設定ミス 無駄処理の検出等
2	クラッシュ	2	11.11%	Kernel のレジスタ操作ミス（書き込み，保存）
3	再起動	2	11.11%	割り込み禁止区間設定ミス，2 重タイマ設定によるリセット
4	フリーズ	1	5.56%	mutex_lock の処理不正
5	起動失敗	1	5.56%	ドライバ処理フラグのクリアミスに伴う起動失敗

以上の結果より，本障害解析環境で解析できた内容は，性能障害や，システムクラッシュ，再起動，フリーズ，起動失敗などであり，その要因は，Kernel のレジスタ操作ミスから開発タスクの無駄処理，タスク間の動作関係に起因する要因までさまざまであることがわかった．そのため，本環境は多くの障害に汎用的に対応できる見込みを得た．一方で評価件数が少ないため引き続き評価が必要である．

本障害解析環境が有効でなかったケースは，各機能固有の解析ツールや，メーリングリストの情報などで解決した事例であり，本環境を使う前の調査段階で対応が可能であった．

3.5.5　提案する障害解析環境採用による障害解析作業の変化

本項では，提案する障害解析環境の使用前後における作業の変化について述べる．

これまでは，以下図 3-9 の左側に記したプロセスのとおり障害解析を行っていた．問題は，再現条件を特定する際とトレースの取得と解析をする際に，試行錯誤を繰り返し，多大な工数をかけて解析を行っていた点にある．ここでいう試行錯誤とは，再現条件を特定するための試行錯誤，トレースの取得に伴う試行錯誤，可視化による試行錯誤がある．

再現条件特定のための試行錯誤は，コンシューマ向け組込み機器に搭載されるメモリやストレージの容量が少ないため，テスト期間中絶えずトレースを取ることができず，再現条件を特定しないと原因分析に必要な情報が取れないことに起因する．そのため，何度も条件を変えてテストし，再現条件を特定する必要がある．

トレースの取得に伴う試行錯誤には，トレース内容の試行錯誤と，トレースタイミングの試行錯誤がある．トレース内容の試行錯誤は，トレース可能な容量・時間の制限を解決するために，どの情報を取るか（プロセス切り替えを取るのか，割り込みを取るのか，システムコールをとるのかなど）を取捨選択し，何度もトレースを取る試行錯誤のことである．トレースタイミングの試行錯誤とは，トレース内容の試行錯誤同様，トレース可能な容量・時間の制限内で，障害の原因となる挙動の記録のために，いつトレースの取得を開始し，いつまで記録するかを決定するために何度もトレースを取る試行錯誤である．

可視化による試行錯誤とは，トレース内容の試行錯誤と連動し，取得した情報を用いてどのように可視化すれば原因分析できるかを考え，実際に可視化する作業の試行錯誤である．

これまではこのような試行錯誤を繰り返し，障害解析を行ってきたが，本研究で提案する手法を導入することにより，障害解析の手順は以下図 3-9 の右側に示すプロセスのように変化した．その結果，これまで行ってきた試行錯誤を伴う再現条件の特定，テストの実施（トレース取得）と可視化手法の検討といった３つの作業が不要となった．一方，提案手法の導入により，一度のプロファイリング，一度のテスト実施（トレース取得）と，試行錯誤的な分析箇所の選択の作業が追加となった．

これまでの作業が不要になった理由を以下に述べる．再現条件特定のための試行錯誤と，テストの実施（トレース取得）のための試行錯誤は，長時間トレース技術により，テスト中止めることなく全ての解析情報の記録が可能になるため不要となった．可視化手法検討と可視化の試行錯誤は，形式変換機能による一斉可視化のため不要となった．

次に追加された作業について述べる．解析対象絞り込み機能により，プロファイリングの作業が増える．また解析対象絞り込み機能と形式変換機能を活用し，解析を進めるために分析箇所選択の作業は増える．しかし，プロファイリングにより大量の分析対象（トレース情報）から優先的に何を解析するかがわかることと，分析箇所選択はトレース結果のフィルタリングなので，それまでのテスト実施と可視化手法検討に対して作業としては軽微なため，原因分析の試行錯誤の工数を減らすことが期待できる．

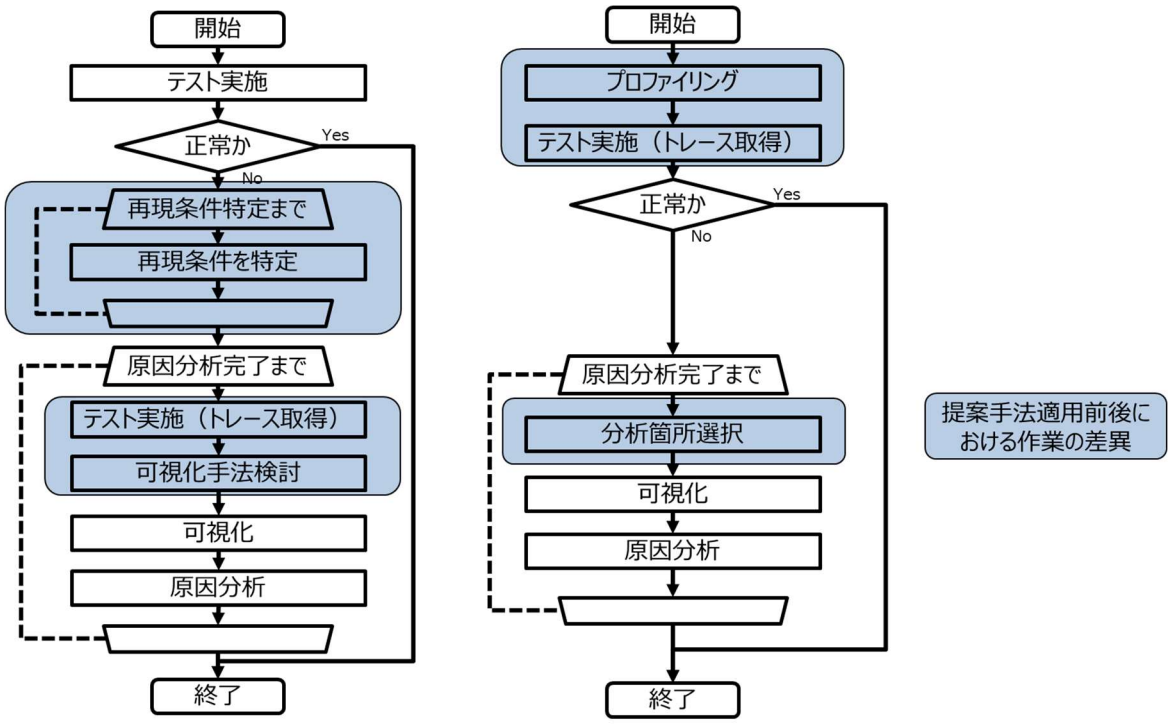

図 3-9　提案手法適用前後の作業フロー及びその差異

3.6 まとめ

本章では，組込みシステムにおける障害解析環境について検討し，Armadillo を用いた試験環境及び，執筆者所属企業のコンシューマ向け組込みシステムを中心とした製品開発において評価を行った．課題抽出にあたっては，それまで主に DTV 及びビデオカメラに代表される家電製品の開発において，開発工数に悪影響を与えていた作業を分析し，対策が必要な課題 3 点と，その課題に着目した要件 3 点を抽出した．それをもとにそれらを解決するための①解析対象絞り込み機能，②長時間トレース機能，③形式変換機能からなる構成で障害解析の効率改善機能を設計した．

次にそれぞれの機能の特徴を列挙する．①解析対象絞り込み機能は，それまで CPU リソースのボトルネック解析で使われていたプロファイラを，我々の製品開発の知見から，複数のタスクの組み合わせで発生する不具合の解析対象の絞り込みに使えることを明らかにし，解析効率を改善できたことである．②長時間トレース機能は，それまでエンタープライズ系のシステム開発では，OS トレースの時間が問題になることがあまりなかったこと，組込みシステムの開発においては JTAG-ICE などによるトレースが主流であったが，OS トレースの容量や導入の工数に課題があることを明らかにしたこと．その課題に対し，OS トレースの結果をカーネル領域のメモリに記録するタイミングにて外部出力バス経由で出力し，市販されている記録装置に保存することにより，低負荷で長時間トレースを実現し，再現性の低い不具合が偶然再現した際に確実に情報を記録できるようにしたことである．③形式変換機能は，それまで特定のトレーサを可視化することに特化した取り組みや，収集できるトレース情報を汎用的に可視化できるよう，トレース内容を一般化する方向で研究が進んでいた．それに対し本研究では，これまでの製品開発の知見を利用し，再現性の低い不具合で解析が必要な，リソースの競合に代表される複数タスク間の競合や，割り込み間隔の確認，不具合の原因となる OS 機能の利用と動作のタイミングといった可視化項目を明らかにし，形式変換機能を開発，その効果を明らかにできたことである．

そしてこれら 3 機能を試験環境と，DTV（HR-01）で評価し，その後組込み製品 6 製品の実開発に適用し，その有効性を確認できた．提案する障害解析環境の採用により，これまで行ってきた不具合再現条件特定のための試行錯誤，テスト実施（トレース取得）のための試行錯誤と可視化手法検討の試行錯誤を不要にすることができた．一方で，プロファイルと分析箇所選択による解析対象の絞り込み作業は増加した．ここで，提案手法による解析対象の絞り込みでは，障害の内容によっては解析に必要なトレース情報を可視化対象外にする可能性がある．その場合は障害要因を含むトレース情報を可視化するまで解析を続ける必要があるため，表 3-5 に示したような効率化はできないことも考えられる．しかし，解析対象の絞り込みで行う分析箇所選択はトレース結果のフィルタリングなので，それまでのテスト実施と可視化手法検討に対して作業としては軽微なため，原因分析の試行錯誤の工数を減らすことが期待できる．また，解析に必要なトレース情報は長時間トレース機能で取得できているため，これまでのような試行錯誤を伴う作業をすることなく確実に障害要因

を特定することができる．

この環境は 2007 年の BD-CAM（DZ-BD-7H）の開発から実適用を開始し，現在，これらの環境は当時の環境から一部変更し，執筆者所属企業における産業機器などの開発で適用を続けている．変更の例としては，OS トレーサを LKST から現在の Linux トレースの標準となっている ftrace に移行した．また OProfile は Perf に移行した．

現在直面している課題としては，形式変換ツールの出力先の可視化ツール TimeDoctor の置き換えである．現在，同ツールはメンテナンスがされてない状態であり，今後前提となる OS や Java のアップデートに伴い，使用できなくなることが考えられる．そのため，今後も本環境を維持する上で，ツールの選定と，それに伴う形式変換ツールの修正が必要である．

第4章 設計工程における性能検証効率化

4.1 導入

コンピュータシステムの 20%は何らかの実行時間の未達成に代表される性能問題を抱えたまま出荷されている[40]．それらの幾つかは製品出荷後に性能遅延等の問題が発覚している．その結果，出荷遅延による収入の損失やペナルティが発生し，プロジェクト中止や顧客との関係悪化に繋がっている[44]．

特にコンシューマ向け組込みシステム開発では，製品グレードごとの機能の差別化や，仕向け地ごとの標準や法規への対応などに基づく機能の変更がある[9][10]．その機能の変更に伴い性能が悪化した場合，対策としてプログラムの改善や，アルゴリズムの見直しなどによる性能チューニングが必要になる．例えば HDD の場合，コンシューマ向けのドライブは単一ユーザがマルチメディアデータを長時間読み書きするようなユースケースにあわせたチューニングが望まれるが，エンタープライズ向けのドライブは，RAID サーバ等に用いられるため，複数のユーザからランダムに細切れの読み書きをするようなケースに合わせたチューニングが望まれる．そのため，メーカはそれぞれの顧客のユースケースに合わせて，目標とする性能を達成するために，キャッシュ機能などの性能チューニングが必要となる．他にも，デジタルテレビのソフトウェアは国ごとの放送規格の指定により，番組表を特定のプログラミング言語で作成しなおすことが必要な場合などがある．その結果，個々の仕向けに合わせて作り直すことで他機能などの性能が劣化してしまい，それに対応した更なる性能チューニングが必要になる場合がある．このように様々な顧客の要望や仕様毎に性能を達成するチューニングをするため，対応する追加工数が必要となる．

この様な性能に関する問題は，本論文執筆者の所属企業でも発生しており，その対策が必要となっていた．それに対し我々は，コンシューマ向け組込みシステムの性能に関する不具合事例を分析した結果，図 4-1 の様な手戻りが発生し，開発の遅延につながっていることを明らかにした．コンシューマ向け組込みシステムでは，設計，実装，テストの前半部分はハードウェアとソフトウェアが平行して実施されるが，ハードウェアとソフトウェア両方のテストが完了して，初めてシステムテストが実施できる．当時の執筆者所属企業では，この段階で初めて性能を検証できる状況になり，システムテストを通じて性能の評価を行う．また，その結果を持って性能最適化に取り組むことができる．ここで設計段階の性能見積もり・検証の精度が低いと，システムテストやデバッグの工程で不具合が顕在化し，その対策のためにテストやデバッグ工程自体が遅延する問題や，最悪の場合は設計段階まで手戻りが発生し，大幅な工数増加につながる．結果，コンシューマ向け組込みシステムの開発全体の効率を悪化させていた．

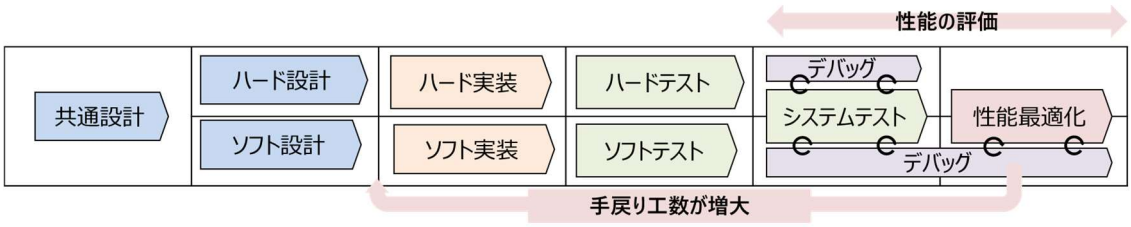

図 4-1　性能問題における手戻りの例

このような問題に対し，研究着手時に，設計段階で統計的な性能見積もりモデルを作成し解析することと，第 3 章で示した障害解析環境を用いてデバッグの効率をあげることの二つの手法を適用していた．しかし前者は，派生開発部においてトレース情報を統計的に十分と言える量が確保できてないため，結果精度が高い予測ができないことが判明していた．後者は，開発工程の後半で不具合が発見できても設計段階において根本的な解決ができないため，図 4-1 に示す手戻りを抑止できないことが判明していた．

一方，これらの課題に対し，モデル検査技術を用いて設計段階で網羅的な性能検証をすることで，問題のある設計を検出，対策ができること．その結果，手戻り発生を抑止する検討はできていた．しかし性能を達成するための実装はプログラムの中に散在するため，差分開発をしているコンシューマ向け組込みシステムでは，モデル検査導入時においてモデル化対象となるソースコードの規模が大きくなってしまい，導入するタイミングで非常に多くの工数がかかることが課題であることがわかった．

そこで本研究では，性能検証の効率化，特に性能モデル検査導入の効率化を取り組む課題とし，その解決により，開発の工数を削減しつつ，精度の高い性能の見積もり・検証を実現することで，デバッグ工程からの手戻りも併せて削減する技術について検討し，ハードディスク製品の事例を通して評価した結果について報告する．

4.2　コンシューマ向け組込みシステム向け性能見積もり・検証に関する課題

上記のような状況において， 図 4-1 に示すような手戻り発生の問題が出ていたため，コンシューマ向け組込みシステムの開発部署に問題の詳細をヒアリングした．その結果，バリエーション毎に細かい設計の変更やチューニングが要求される製品の開発では，2.4.2 に示したような性能見積もり技術，特に統計的な性能見積もり技術では，十分な統計情報（トレース結果などに基づく性能検証用の情報）が無いため，性能見積もり・検証の精度が出ないことが判明した．

上記に対し，既存技術のうちモデル検査技術を用いた網羅的な性能検証技術を用いることで，上流工程における性能品質の改善を実施し，下流工程における手戻り発生を抑止できないかを検討した．

モデル検査は，一般的に検査技術自体の難しさと，従来技術とのすり合わせの難しさの 2 点から，導入が難しいとされている．検査技術自体の難しさとしては，検査モデル作成，検

査条件作成，ツール操作方法の習得，反例解析の難しさなどがあげられる．従来技術とのすり合わせの難しさとしては，設計時に検証モデル作成の工程を追加するといった開発プロセスの変更の必要性と，検査技術習得までの人的リソース確保の難しさが挙げられる[58]．

本研究では，上記のうち，検査モデル作成の難しさに着目する．検査モデル作成の難しさには，モデル検査用の言語が従来のプログラム言語と異なる点，検査用のモデルの妥当性を示すことが困難な点，性能を達成するための実装がプログラムの中に散在しているため，性能モデルの作成には，製品全体のモデル化が必要な点が挙げられる[45]．特に最後の点については，組込みシステムの製品開発において，特に大きな課題となる．理由としては，組込みシステムの多くは，開発コスト削減のため，短期で製品開発をしなくてはならない．そのため，既にあるソフトウェアを利用し，機能を追加・改造をする差分開発が多くされている．よって，新たに性能検証の手段としてモデル検査を導入する場合，導入のタイミングでシステム全体を考慮した性能検証用のモデルを作成する必要がある．しかし既に述べている通り，組込みシステムは開発コストの削減が望まれているため，導入に際し大量の開発コストをかけることはできない．その結果，性能検証用のモデル検査導入時には大量のモデリング対象から性能に関して必要な部分だけを抽出し，効率良くモデルを作成する必要がある．以上を課題の観点１とする．

次に，機能検証用のモデル作成と，性能検証用モデル作成の違いについて述べる．性能検証用のモデルを，既存システムのプログラムがある製品に組み込むためには，以下 4 つの作業の実施が必要となる．

（作業 1）　既存システムの分析・構造の把握
（作業 2）　既存システムの実行時間解析
（作業 3）　既存システムの性能検証用モデル作成
（作業 4）　作成した検証用モデルの妥当性検討

上記各作業の詳細と，既存の対策手法について説明する．まず，(作業 1)既存システムの分析・構造の把握であるが，この作業では，検証用のモデルの作成のため，既存システムの設計ドキュメントを読んだり，システムを実行したりすることで，ソフトウェアの制御構造やデータ構造を可視化し，既存システムの構造を把握する．この作業の支援ツールとしては，OSS の場合，gcc のオプションである“—fdump-rtl-expand”と，可視化ツールである egypt などを使い処理フローを可視化する方法や[59]，プロプラエタリなソースコードの解析ツールである Understand§§§§などを利用して処理フローを可視化する方法により，構造を迅速に把握することができる．次に，（作業 2）既存システムの実行時間解析であるが，この作業では既存システムを実行し，実行時間や実行回数を，機能単位で取得する．この作業の支援ツールとしては，OSS の場合だと gprof などのプロファイリングツールや，gcc のコンパイルオプションである“—finstrument-function”を使い，関数の入り口と出口でフックを

§§§§ TechM@trix(R)：Scitools Understand，(online)，
https://www.techmatrix.co.jp/product/understand/index.html，(2022 年 4 月 3 日現在)

かけ，関数名と時刻をトレースする手法がある．これらのツールを使うことにより，モデル検査の検証に使う実行時間情報を取得し，モデル化に使用できるようになる．（作業 3）既存システムの性能検証用モデル作成では，（作業 1），（作業 2）の結果を基に，性能検証用のモデルを作成する．この作業に関連するツールとして，プログラムのソースコードを直接利用しモデル検査を実施するツールや，モデル検査器自体にソースコードを再利用する機能があるものがある．また，時間オートマトンを用いて性能に特化した検証をする UPPAAL 等がある．まず，既存ソースコードを利用する手法について説明する．例えば Fever[52]や CBMC[51]といったツールにより，C 言語のプログラムから，モデル検査用のモデルを生成・検証，または直接モデル検査することができる．また SPIN では，C 言語のプログラムを呼び出すことができる[23]．次に UPPAAL についてであるが，こちらは検証モデルを時間オートマトンで新たに作成し，検証するものである．最後に，（作業 4）作成した検証用モデルの妥当性検討では，今回の性能の場合，処理の実行を組み込みシステム・モデル双方で実施し，組込みシステムの実行に要した時間と，検証用モデルと検証器で検証した結果出力された時間を比較し，妥当性を検証する．

このような性能検証用モデル作成の作業に対し，（作業 3）関連技術は今回対象とする性能（実行時間など）に対応しておらず，性能の検証を行う場合は，検証用モデルを新規に作るか，ツールで取得した検証モデルに性能（例えば，モデルに実行時間算出のロジックを組込み，目標時間以内に終了状態に到達可能か）を検証できるように作り直さなくてはいけないという課題があった．また UPPAAL については，モデルの作成・検証を試行し評価したところ，モデルの作成工数が大きくなること，作成した時間オートマトンのモデルが，既存プログラムと同じ動作をすることを証明，説明するための工数が新たに追加されることから，作業工数が増えてしまうという課題もでた．以上 2 点を課題の観点 2 とする．

以上，課題の観点１，観点２を踏まえ，本研究では，モデル化対象の既存のソースコードを再利用し，必要最低限のモデル化で性能検証を実現する手法により，性能モデル検査導入を効率化する手法を提案し，評価する．

4.3　ソースコードを再利用した性能検証用モデル作成手法

4.3.1　　本研究における性能の定義と検証したい内容

性能には，実行時間・スループットという定義があるが[60]，本研究では，当時の開発の状況から，扱う性能は実行時間に限定する．そのため実施する性能検証は，目標とする実行時間以内に，目的とする処理の実行が完了状態に到達することを検証するものとする．今回取り上げる HDD のキャッシュの検証の場合，ワークロードと呼ばれるコマンド列の最初のコマンドを HDD のキャッシュが受理してから，最後のコマンドを受理完了するまでの時間が，目標とする時間より短いかを検証する．

4.3.2　　提案手法の概要

提案手法では，対象とするプログラムが，目標とする時間以内に実行完了することを検証可能にする．そのため，検証するモデルは，プログラムの実行開始から終了までの実行時間を積算し，目標時間内に完了したかを検証するものとする．

本研究で提案する手法は，既に大量のソフトウェアが開発されており，新たにモデル検査を用いた性能検証を導入する場合において，性能検証に必要な部分をモデル化する手法である．

ここで，検証用のモデル作成に関し，モデルを新規に作成する必要がある部分とそうでない部分を検討し，どのようにモデルの作成をすればよいかを表 4-1 にまとめた．以降，表 4-1 に従い説明する．まず，(1)について説明する．性能検証の開始/終了処理，性能検証のためのパラメータ定義，再利用する関数の定義，再利用するソースコードの呼び出し，検証式による判定といった性能検証部，及び新規開発機能などのモデル作成は，既存の製品コードには含まれないため，新たに検証用のモデルを作成する必要がある．新規に必ず作る必要があるモデルは(1)の機能であり，残りの部分は，製品の既存コードがある場合，再利用すれば良い．

次に(2)～(4)の既存の製品コードを再利用する部分について説明する．今回，検査対象とした HDD のキャッシュ模擬プログラムを対象に，どのようにソフトウェアの性能に影響するかを分析したところ，以下 3 点から成ることが判明した．(2)実行時間が長く性能に直接影響する部分（例：ヘッドのシークエミュレーション）(3)実行時間は短いが，その後の処理に影響する((2)の処理へ分岐する)部分（例：コマンド受信後の処理）(4)その他の 3 点である．

以上の分類結果をもとに，必要な部分だけの実行時間のみを取得することを考慮すると，(2)に該当する部分は，既存コードの処理を流用しつつ，実行時間の算出機能を持たないプログラムの場合，その算出をモデル化する必要がある．(3)に該当する部分は，(2)の出現パターンを網羅して実行時間を計算する必要があるため，制御構造を維持しつつモデル化する必要がある．(4)については性能の検証のため，処理の結果は必要であるが，それ自体をモデル化し検証する必要は無い．以降製品ソースコードの再利用方法について説明する．

表 4-1　モデル作成の指針

#	モデル化方針	モデル化対象機能・処理		説明箇所
(1)	新規作成	検証開始/終了処理 検証用パラメータ定義 再利用関数定義 再利用コードの呼び出し 性能検証部 新規開発機能		4.3.3.2 にて説明
(2)	既存コード再利用	性能に直接影響する処理（実行時間が長い処理）		4.3.3.3 にて説明
(3)		その他	(2)に分岐する処理	
(4)			その他	4.3.3.4 にて説明

ここで，作業効率化を進めるため，モデリング言語には，Promela を使い，検証器には SPIN を使うことにする．今回 Promela/SPIN を使った理由は，執筆者の所属企業では製品開発に C 言語を使っていたため，Promela/SPIN が提供している C 言語の処理をそのまま呼び出すためのインタフェースを使うことで，開発コード・開発済みのコードが再利用できる見込を得ていたからである[23]．関連する研究として，C のソースコードから Promela のモデルを生成する *FeaVer*[52]や，C のソースコードを直接モデル検査の対象にできる CBMC[51]がある．しかしこれらの検証対象は機能検証であり，これらを用いてモデルを作成しても，非機能要件である性能の検証はできない．そのため，性能をどのようにモデル化し，検証するかを 4.3.3 以降で説明する．

4.3.3　製品のソースコードを利用した性能検証用モデルの作成方法

4.3.3.1　Promela と SPIN を用いた実行時間検証

ここでは，Promela と SPIN を使い，実行時間の検証をどのようにするかについて説明をする．本論文では，4.3.1 に示した通り，実行時間に限定し，検証方式を説明する．

検証の方式は幾つか考えられるが，我々は（1）動作時間を外部のプロセスで生成し検証する方式と，(2)動作時間を処理の中で加算し検証する方式の 2 方式を検討した．そして，時間経過の情報が実機などの情報と同様で開発者が理解できる（2)の方式を採用することにした．そのやり方についてそれぞれ説明する．

(1)の方式は 図 4-2 に示す通り，プロセスの実行時間検証をしたい場合，1～7 行目に記したタイマ生成プロセスを使い，検証するシステムの時刻を生成．その後処理の中の 10,12,13 行目のように経過時間を計測し，検証に使うというものである．

```
1 | int clock = 0;
2 | active proctype timer() /*タイマ生成プロセス*/
3 |{
4 |   do
5 |   ::clock = clock + 1
6 |   od
7 |}
8 | active proctype a()
9 |   {
10|       start_time = clock;
11|        //各種処理
12|       end_time = clock;
13|       SystemTime = SystemTime + (end_time – start_time);
14| }
```

図 4-2　動作時間を外部生成して検証する方式

上記の場合最後に SystemTime を使い，モデルの終了状態の箇所において例えば assert(SystemTime < 目標時間)のようなアサート文を導入して検証することで，目標時間以下でプログラムが完了するか検証ができる．

(2)の方式は，以下図 4-3 の通り，モデルで時間を生成するのではなく，既存プログラムや，既存システム，及び新規設計関数などの平均実行時間を計測及び推定した値を準備しておき，処理実施時に 6 行目のように処理事に経過時間を加算し，(1)の方式同様，モデルの終了状態の箇所において例えば assert(SystemTime < 目標時間)のような assert 文を導入して検証することで，目標時間以下でプログラムが完了するか検証ができる．

```
1 | int clock = 0;
2 | active proctype a()
3 |   {
4 |       //各種処理
6|   SystemTime = SystemTime + (result_a_time); /*計測した平均時間などを加算*/
7| }
```

図 4-3　動作時間の計測情報を活用して検証する方式

4.3.3.2　　新規作成部分のモデル化

ここでは，4.3.2 に示した(1)新規作成部分に関して，モデル化する例を図 4-4 に示す．今回は簡単のため，reuse_func という関数を再利用し，実行時間を求める処理に対し，実行時

間を管理するパラメータ SystemTime が 100 以下で終了するか検証するモデルを例に説明する．

表 4-1 に示した通り，新規作成部は検証開始/終了処理，検証用パラメータ定義，再利用関数定義，再利用コードの呼び出し，性能検証部，新規開発機能などをモデル化する．

検証処理の開始/終了は，検証モデルの主な処理になり，図 4-4 の 9 行目～16 行目 model_main()部が相当する．その中で，10 行目のような検証用のパラメータを定義する．そのパラメータを 4 行目～8 行目に示す再利用した関数 reuse_func を用いて，実行時間を演算，変更する．この再利用の仕方については，4.3.3.3 に示す．また再利用のための定義は，1 行目～3 行目が相当する．この定義の仕方については， 4.3.3.4 で示す．性能の検証は，15 行目の assert 文が行い．この場合は，実行時間 SystemTime が 100 未満であれば，検証は成功であり，100 以上であれば，検証が失敗し，反例が出力される．このほか，新規作成部をモデル化，適宜呼び出しをするようにする．13 行目の check_p は，reuse_func に引き渡すための変数で，このように定義することで，呼び出し先の C プログラムに Promela で定義した変数とその値を引き渡すことができる．

```
 1| c_decl{
 2|   int reuse_func(int); //再利用する関数の定義
 3| }
 4| c_code{//関数の再利用
 5|   int reuse_finc(int){
 6|   //略
 7|   }
 8| }
 9| proctype model_main(){ //開始
10| int SystemTime =0; //検証用パラメータ定義
11|   c_code{
12|     //再利用コード呼び出し
13|     Pmodel_main->SystemTime= reuse_func(Pmodel_main->check_p)
14|   }
15| assert(SystemTime < 100); //性能検証部
16|}//終了
```

図 4-4　新規作成部の例

4.3.3.3　既存コードを再利用したモデル化（性能に直接影響する処理/または性能に直接影響する処理に分岐する処理）

ここでは，表 4-1 にて説明した(2)性能に直接影響する部分と，(3)性能に直接影響する部

分に分岐する処理部分の２つの方法について，Promela にてモデル化する例を示す．

図 4-5 に，1 例として 4 章でモデル化する HDD キャッシュ模擬プログラムのソースコードを示す．このソースコードは，キャッシュに登録されたデータがディスクドライブに出力された際の経過時間を取得するソースコードである．図 4-6 にそれをモデル化（Promela 化）したものを示す．図 4-5 において，4.3.2 の(2)　性能に直接影響する処理は 2，4，5 行目になる．また，(3)に該当する分岐は，1，3，6 行目になる．

```
1| if(LRUDumpTime ==0){
2|       SystemTime += TimeInterval;
3 | }else{
4|       SystemTime += LRUDumpTime;
5|       LRUDumpTime = 0;
6| }
```

図 4-5　C ソースコード例

```
 1| if
 2|     ::c_expr{ LRUDumpTime == 0} ->
 3|       c_code{
 4|               Pcache_main->SystemTime += TimeInterval;
 5|       };
 6|    ::else ->
 7|       c_code{
 8|          Pcache_main->SystemTime += LRUDumpTime;
 9|          LRUDumpTime = 0;
10|       };
11| fi;
```

図 4-6　Promela モデルの例

(2)性能に影響する部分をモデル化した例を図 4-6 に示す．図 4-6 において，性能に影響のある部分をモデル化した個所は，3,4,5 行目と 7,8,9,10 行目となる．今回の検証では，実行時間を検証するため，処理に応じて実行時間を計算するコードを検証モデルに追加するか，実行時間を計算する処理を再利用し，処理結果を参照できるように既存コードをモデル化する必要がある．今回の例の場合は後者であり，図 4-5 の 2 行目及び 4 行目が，それぞれ処理が発生しない場合のインターバル経過時間の加算と，処理が発生した場合の時間経過の加算を表しているので，その処理を再利用する．したがって，処理自体は C 言語のま

ま実行するが，得た実行時間はモデル検査にて検証可能にする必要がある．そのため，図 4-6 の 3,5 行目，7,10 行目のように c_code{}にて処理自体を囲み，C 言語として処理を実施するが，のちに検証可能にするために，4,8 行目のように Pcache_main といったように，Promela プロセス cache_main の変数 SystemTime に結果を渡す処理を追加する．次に，(3) 性能に直接影響する部分に分岐する処理については，図 4-6 の 1,2,6,11 行目がそれに相当する．if 等，C 言語に対応する制御構造は，Promela でもほぼ使用可能であり，それを使いモデル化する．例えば，if 文については，C 言語では図 4 の様に使用する場合，Promela では図 2 の 1,11 行目の様に if~fi;で囲んだ間に，2 行目と 6 行目の様に条件文を入れて使用する．条件式は，c_expr というプリミティブを用いて表現する．

4.3.3.4　既存コードを再利用したモデル化（処理全体の再利用）

ここでは，表 4-1 の(4) その他の処理に関するモデル化手法について説明する．性能に関係ない部分は，システムの振る舞いを再現するためには必須であるが，性能検証のための時間経過に直接影響しない．そのため，処理結果だけを利用することを目的に，既存コードを丸ごと再利用する．その例を図 4-7，図 4-8 に示す．C 言語で記載された関数を丸ごと再利用する場合は，図 4-7 に示すように，c_code というプリミティブで対象のソースコードを囲めば良い．ヘッダの再利用の場合は，図 4-8 に示すように c_decl というプリミティブで対象のソースコードを囲めばよい．

```
 1| c_code{
 2|     //compare function
 3|     int comp_segment(const void *seg1,const void *seg2)
 4|     {
 5|         int Time1,Time2;
 6|          SegmentUnit *Unit1 = *(SegmentUnit **)seg1;
 7|          SegmentUnit *Unit2 = *(SegmentUnit **)seg2;
 8|
 9|          Time1 = Unit1->Time;
10|          Time2 = Unit2->Time;
11|
12|          return Time1 - Time2;
13|     }
14|}
```

図 4-7　コード再利用の例

```
1| c_decl{
2|     int comp_segment(const void *, const void *);
3| }
```

図 4-8　ヘッダ再利用の例

4.3.4　　性能検証用モデル開発の流れ

4.3.3 までに記載したモデル化手法を用いて，性能検証をするためのモデルの開発手順を図 4-9 に示す.

まず，1.既存コードの分析を行う．このステップで，既存コードを機能/処理単位でどのようなものかを理解する．具体的には，4.2 の（作業１）で説明した内容を行う．次に 2.処理/機能の分類を行う．このステップでは，ステップ 1 の理解を基に，既存コードの処理/機能を 4.3.2 で説明した性能面における影響に従い，分類する．次に 3.性能測定を行う．これは，モデル化する場合において，実行時間の計算を検証用のモデルで行う場合に実施する．具体的には 4.2 で説明した（手順 2）に従い実施する．次に 4.モデルの作成を行う．このステップでは，ステップ 2 の分類と，ステップ 3 の計測結果を用い，前節までに説明したモデリング手法に基づき，性能検証モデルを作成する．次に，ステップ 5 では，作成した性能検証モデルが，既存ソースコードでビルドされたプログラムと同じふるまいをするか，テストデータなどを作成し，検証する．ふるまいが異なる場合はステップ 4 に戻り，モデルを修正する．ふるまいが同じになったならば，ステップ 6 に進み，性能検証を実施する．ここで，本研究において「ふるまいが同じ」とは，4.3.1 に示した検証したい内容である「ワークロードと呼ばれるコマンド列の最初のコマンドを HDD のキャッシュが受理してから，最後のコマンドを受理完了するまでの時間」が，モデル作成の元となった対象（今回の場合，キャッシュエミュレーションプログラム）と同じ時間になることと定義する.

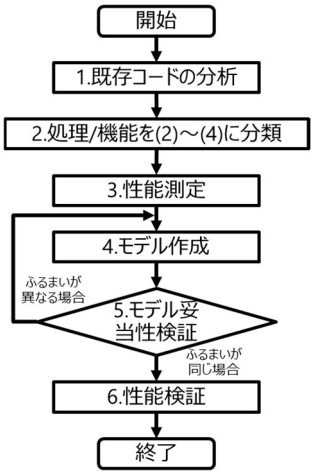

図 4-9　性能検証用モデル開発の流れ

4.4　モデルの作成と妥当性の検証

本章では，これまで述べた性能検証・最適化手法を HDD キャッシュ機能プログラムに対して適用した結果について述べる．

適用の流れは 4.3 に示した通りである．まず，一般的なハードディスクの概要について 4.4.1 で述べる．次に，モデル化対象となるキャッシュ機構の概要について 4.4.2 で述べる．その後，ステップ 1．既存コードの分析の結果について 4.4.3 で述べる．更にステップ 2.処理/機能の分類結果について 4.4.4 で述べる．最後にステップ 3,4 に従い作成したモデルの 5.妥当性検証結果と，6.性能検証結果について，4.4.5 で評価する．

4.4.1　　HDD の概要

今回は HDD(Hard Disk Drive)のキャッシュ機能の性能検証・性能最適化を対象とする．HDD の I/O 性能は，ディスクアクセスの頻度に左右される．何故ならば，ディスクのヘッド到達時間（シーク時間＋回転待ち時間）は，7200 回転のドライブの場合で 8.2msec+8.33msec の計 16.53msec なのに対し，キャッシュメモリの制御に必要な時間はμ秒単位の処理である．このことから，HDD の I/O 性能に対し，ディスクアクセスの時間は支配的であることがわかっている[61]．そのため HDD はディスクアクセスを削減するために，アクセスしたデータをメモリに保持するキャッシュ機能を備えており，キャッシュの利用効率を高めることで，HDD 全体の性能が発揮される．今回は，このキャッシュ機能を対象として，性能を探索・評価する際にモデル検査を利用した結果を示す．また，キャッシュの制御の結果，処理の時間が発生するドライブ部分の処理時間については，今回は一律データサイズに対して平均的にかかる時間を用いて値を返却するものとする．

4.4.1.1　　HDD の構成とキャッシュメモリ

HDD の構成を図 4-10 に示す．HDD は FW に代表されるソフトウェア部分と，I/F コントローラ（プロトコルチップ），メモリ，ドライブ，その他制御用のコントローラ等のハードウェアから成る．

次に処理の流れを，ライト処理を例に説明する．HDD は，ホストから発行された処理を I/F controller で受け取る．受け取った情報は FW に渡される．FW のキャッシュコントローラ部は，Cache に空きがあるか判定し，空きがない場合は Drive Controller 機能と Memory controller を使用し，Cache 上のデータを Drive に書き込むことで Cache 上に空きを作る．さらに Memory controller を使用し，Cache へデータを書き込む．

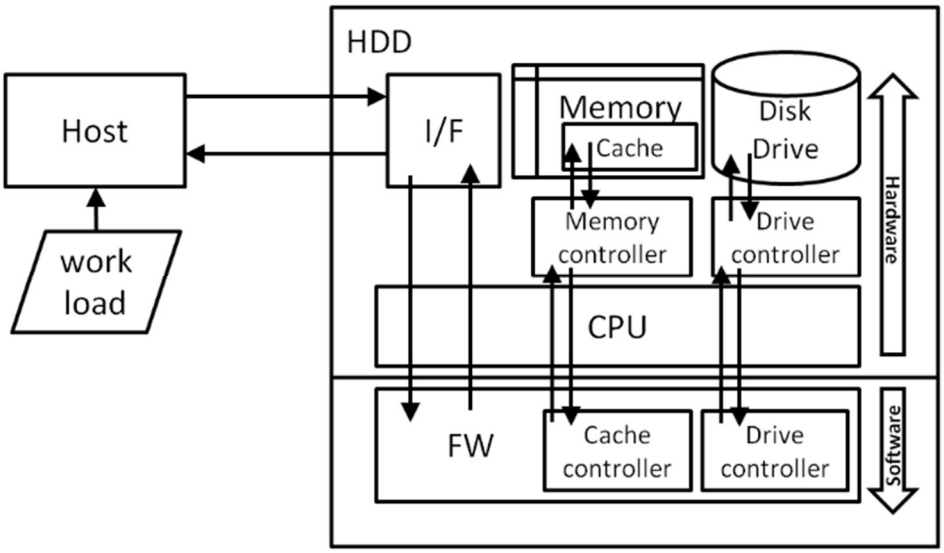

図 4-10　HDD 概要図

4.4.1.2　　モデル検査にて検証する内容

我々は，今回 HDD のキャッシュ機能に関連する性能について検証したい．コンピュータシステムの性能とは，4.3.1 に示した通り，実行時時間とスループットである．HDD の性能も同様の定義で，詳細には HDD の性能は，以下の通り定義されている．

・スループット：単位時間当たりのデータ転送速度(MB/s)
（IOPS:単位時間当たりの I/O 数）
・実行時間（応答時間）：I/O 要求を受けてから応えるまで

本論文では，これらの性能のうち実行時間に絞って議論をする．

上記の性能の定義に基づき，我々はワークロードを構成する先頭コマンドの送信開始から送信完了までの時間（ドライブ側の処理の実行時間）を検証する内容とする．

次に，今回作成するモデルの検証結果は，他社製品や，自社製品の先行モデルの実機を用いた性能の計測結果と比較することを目標とする．そのため，評価結果をそのまま比較に使える程度の模擬精度を求める．よって，時間精度に関しては抽象化の対象としない．

4.4.1.3　　評価に伴うパラメータ

今回，評価に用いるパラメータは，キャッシュ模擬プログラムに準ずるものとする．以下表 4-2 に，そのパラメータ情報を示す．

表 4-2　検証用のパラメータ一覧

パラメータ名	概要
Rotational speed	1 分間あたりのプラッタの回転数
Sector Size	トラックの分割単位（サイズ）（512 or 4096 byte）
Cache Size	キャッシュメモリ容量
Average seek time	読み書き対象の位置までヘッドを移動する時間
Max segment count	キャッシュメモリの分割数
Max sector count	トラックあたりの最大セクタ数

4.4.2　　今回モデル化する HDD キャッシュ機構

4.4.1.2 に示したモデル化の方針に従い，キャッシュ機能の模擬コードから Promela で記述したモデルを生成する．

キャッシュの処理の概要を，図 4-11 に示す．キャッシュ処理は，最初にキャッシュ内に纏めて出力可能なデータが有るかを調べ，ある場合はドライブに出力する．無い場合は，コマンド I/F が受け取った処理を受信し，キャッシュがオーバーフローするかを調べ，オーバーフローする場合は，キャッシュのポリシーに従い，ドライブに書き戻すデータを選択し，出力をする．その後，I/F が保持するデータをキャッシュに保存する．

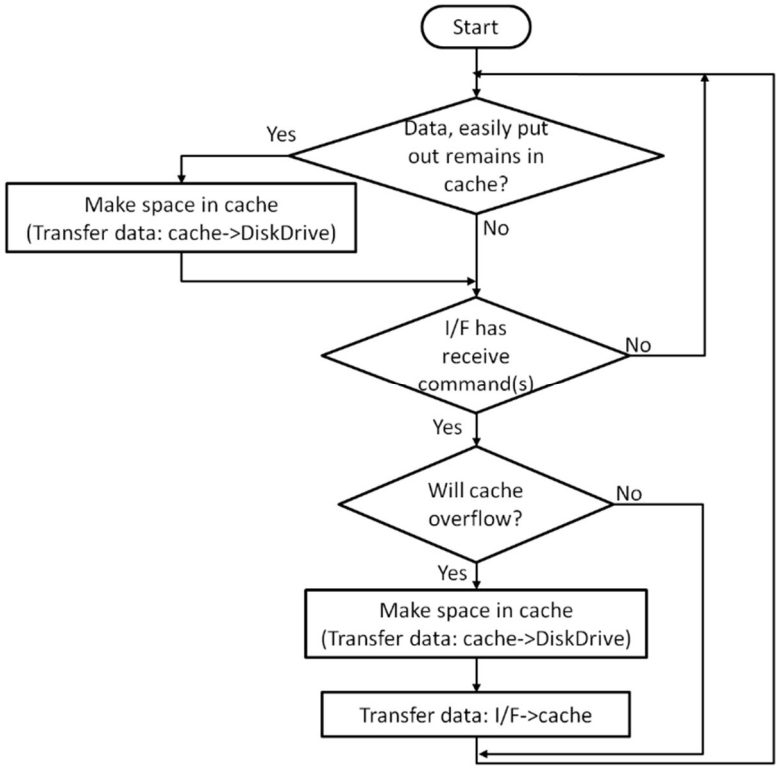

図 4-11　キャッシュ処理の概要

以上の概要を踏まえ，図 4-12 に今回性能検証用のモデルを作成するキャッシュ模擬プログラムの状態遷移図を示す．今回作成したモデルは，ホスト部分の処理は無く，HDD 自体でワークロード情報を読み込み，Cache の状態の模擬を実施する．また，ドライブの処理時間については，実機を用いるのではなく，今回は平均処理時間を返却する仮想的なモデルを呼び出すものとする．

状態遷移図の各状態は，以下の通り
q0: Workload check (End)
q1: Segment count check
q2: Create drive access list using cache data
q3: Judge existing access list
q4: Calc drive Access Time and cache clear
q5: Check exist any drive access
q6: Set lapsed time by drive access
q7: Set interval time.
q8: Update system time.
q9: Get commands within update time
q10: Create new segment
q11: Modify hit segment
q12: Check cache size
q13: Decide destage segment
q14: Calc drive access time and clear cache
q15: Transfer data from I/F to cache
q16: Finish

次に動作の流れについて説明する．動作が開始すると，workload を読み出し，q0 に遷移してコマンドリストの残り数をチェックする．残りが有れば q1 に遷移し，なければ q16 に遷移して動作を完了し，実行時間を検証する．q1 ではキャッシュ内部の segment カウントを見て，ドライブに出力するかしないかを判定する．出力する場合は q2 に遷移し，しない場合は q5 に遷移する．q2 では，ドライブアクセス用のキャッシュセグメントリストを作成し，q3 に遷移する．q3 では，ドライブアクセスリストの有無を検査し，ある場合は q4 へ．ない場合は q5 へ遷移する．q4 では，ドライブアクセスの時間を，アクセスリストの先頭 LBA の位置と，アクセスするデータの長さから計算し，取得する．アクセス処理が終了すると，q5 に遷移する．q5 では，（q4,q14 において）ドライブアクセスが有ったどうか確認する．ドライブアクセスが有る場合は q6 へ，ない場合は q7 へ遷移する．q6 では経過時間にドライブアクセス時間の合計を設定し，q8 へ遷移する．q7 では，経過時間に設定済みの

一定経過時間を設定し，q8 へ遷移する．q8 では設定された経過情報を元に，システム時刻を更新し，q9 へ遷移する．q9 では，q8 で更新されたシステム時間の範囲で，到着したコマンドを読み込む．コマンドが無ければ q0 に戻る．コマンドが有る場合は，キャッシュをチェックし，hit/miss hit の判定をする．miss hit の場合は q10 に遷移し，hit した場合は q11 に遷移する．q10 では，新規にセグメントを確保する際のサイズを計算する，サイズ情報を引き渡し q12 へ遷移する．hit の場合は，hit した cache セグメントの更新情報を計算し，サイズ情報を引き渡し q12 に遷移する．q12 では，更新されたキャッシュ情報を元に，模擬するキャッシュサイズを超えるか否かを判定する．キャッシュサイズを超えない場合は q9 に遷移し，越える場合は q13 に遷移する．q13 では，キャッシュを更新するスケジューリング（例 LRU）を用いて，ディスクに出力する若しくはクリアするキャッシュセグメントを決定し，q14 に遷移する．q14 では，キャッシュのデステージを実行し，write キャッシュの場合にはドライブ出力を実施する．その後 q15 へ遷移する．q15 では，I/F 部に到着しているコマンドのデータをキャッシュへ転送する．転送完了後，q16 へ戻る．

以上が今回モデル化する対象のキャッシュ模擬プログラムの処理シーケンスである．

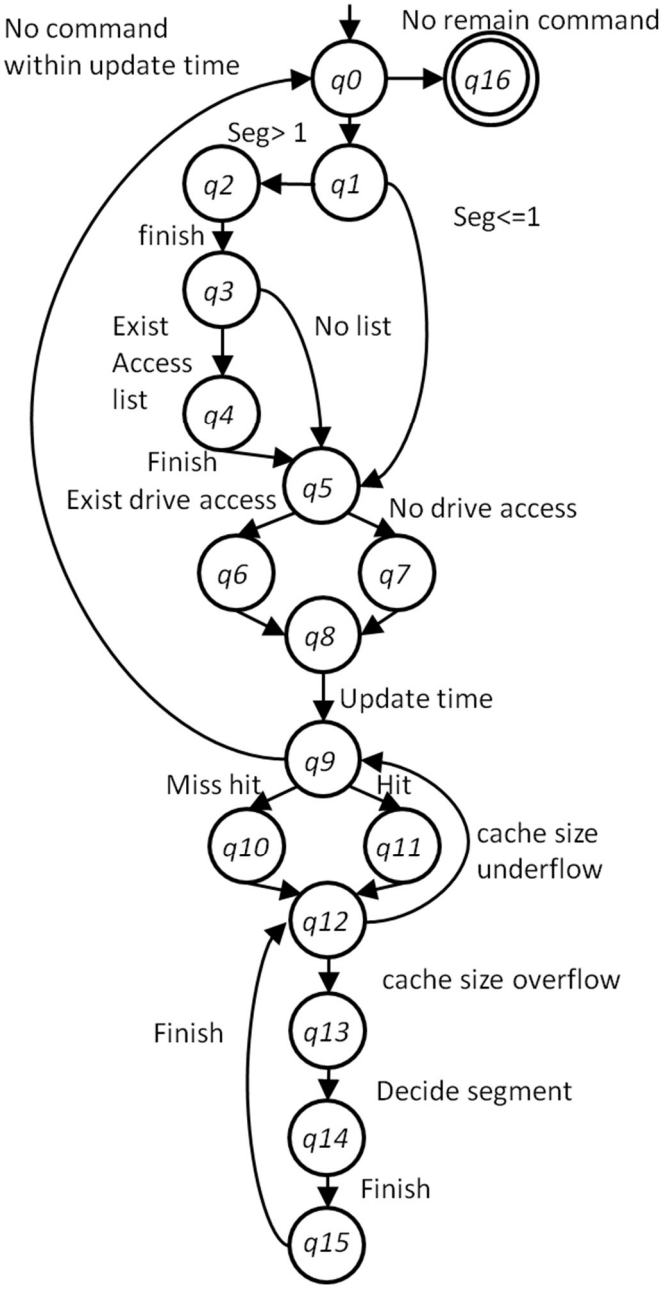

図 4-12　キャッシュ模擬プログラムの状態遷移図

4.4.3　　キャッシュ模擬プログラムの分析

本節では，キャッシュ模擬プログラムの分析について説明する．今回は，既存の C プログラムで作成されたキャッシュ機能の模擬プログラムを再利用し，検証用のモデルを作成する．そのため，C プログラムを元に，再利用する箇所と，しない箇所をどの様に判定したかを以下に述べる．

4.4.1.2 に基づき，我々は対象となるキャッシュ模擬プログラムとソースコードの分析を実施した．本項では，分析の結果について述べる．

4.4.1 で述べた通り, HDD の I/O 性能は, ディスクアクセス時間が支配的となっており, キャッシュの処理自体は検証時の誤差程度の時間しか必要としない. そこで, 今回の実行時間の検証において, 経過時間の計算はドライブアクセス箇所に絞り込んだ. しかし, ドライブアクセスが発生する契機については, コマンド到着時間に依存するため, コマンド到着時間の情報を基に, 経過時間を計算することにした.

また, 上記から, ドライブ処理の状態の有無を判定するために必要な分岐と, コマンドの送受信に伴う分岐は, 経過時間に影響を与えるため, C 言語のプログラムの呼び出し対象から外し, Promela のモデル化対象とした.

次に, 上記の方針より, ドライブのアクセス内容を規定する処理については, 実行結果のみがドライブのアクセス時間に影響があり, アクセス内容を生成する処理の過程は性能に影響が無いと考えた. よって, ドライブアクセス内容を決定する処理は C 言語のプログラム呼び出しの対象とした.

更にドライブ自体の処理については, 今回は性能検証の対象がキャッシュ機能であることから, ドライブのデータ長に合わせ平均処理時間を返却するものとし, その処理自体を C 言語のプログラムの呼び出し対象とした.

4.4.4 キャッシュ模擬プログラムを再利用した性能検証モデルの開発

4.4.4.1 性能検証モデルの作成

図 4-12 に示した状態遷移図に対し, 4.4.3 の分析結果を元に, 表 4-1 に示した (2) ～ (4) の手法でモデル化する箇所を決定した. その結果を図 4-13 に示す.
点線で囲った箇所は表 4-1 (2) (4), 4.3.3.4 に示した処理の再利用によってモデルを作成した. 特にグレーでハッチングした箇所は, 性能に影響する部分である. ここでは時間経過情報をモデルに引き渡すために, 図 4-6 の 4,8 行目で示した時間経過情報をモデルへ引き渡す処理を適用した. 次に実線で囲った箇所は点線箇所にて取得した結果を用い, 表 4-1 の (3), 4.3.3.3 に示した手法を適用してモデル化を実施した.

図 4-13 の実装例の一部は, 既出の図 4-6 である. 図 4-6 は, q5,q6,q7 の 3 状態からなる状態遷移を示している. 図 4-6 の 1,2,6,11 行は q5 を, 3~5 行目は q7 を, 7~10 行目は q6 を示している. そして 3~5,7~10 行目はそれぞれ c_code{}で処理が埋め込まれており, C 言語プログラムが再利用されていることがわかる. 他の処理箇所も同様に 4.3.3 に記載した手法を用いてモデルの作りこみを行った.

4.4.5 作成したモデルの妥当性検証

本節では, まず検証に用いた環境について説明する. 次に, 今回提案した手法で作成した検証モデルを用いて得られた実行時間が妥当であるかを検証する. 最後に, 作成した性能検証モデルによる実行時間検証について示す.

4.4.5.1　　検証に用いたデータと環境

(1) **検証に用いたワークロード**

今回の検証では，表 4-3 に示す workload を使用した．

表 4-3　検証用ワークロードの仕様

Name	Value
command count	6510
command input time range(μ sec)	0~ 35529817
Start LBA range	95~1953512383
Data length(sector)	1~256

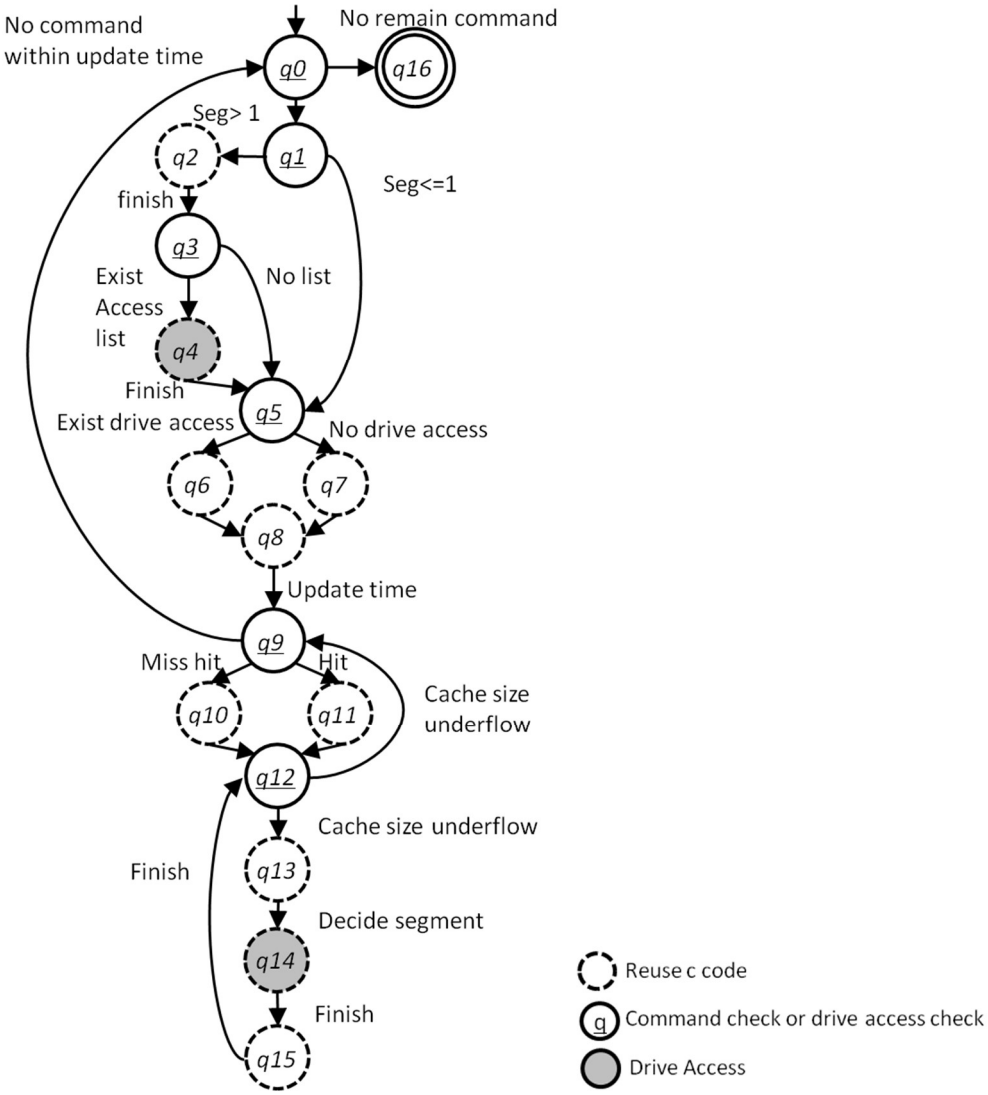

図 4-13　作成するモデルの分析結果

(2) **検証に用いたパラメータ**

今回の検証では表 4-4 に示すパラメータを使用した.

表 4-4　検証用パラメータ一覧

Parameter	Value
Rotational speed	7200 rpm
Sector Size	512 byte
Cache Size	4,8,16,32,64 MB
Average seek time	8.2 msec
Max segment count	2048
Max sector count	2048

(3) **検証に使用した PC**

今回の検証では，表 4-5 に示す PC を使用した.

表 4-5　検証環境 PC の仕様

Name	Value
Name	Dell Precision T1500
CPU	Intel(R)Core(TM)i7-860 2.8GHz
Memory	16GB DDR3 SDRAM(1066MHz)
Chip Set	Intel(R) H57

(4) **用いた検証器と C コンパイラ**

今回の検証では，SPIN Version　6.4.8 を用いた．また C コンパイラは gcc 4.8.4 を用いた.

4.4.5.2　　提案手法妥当性の検証

今回我々は，4.3.4 に示した通り，モデル化対象のキャッシュ模擬プログラムと，そのプログラムを再利用し作成した性能検証用モデルのそれぞれに対し同一のワークロードを入力し，ワークロードの最初のコマンドを受理してから，最後のコマンドを受理するまでの処理時間を比較し，作成した性能検証モデルが妥当であるか検証した.

具体的には，表 4-3 に示したワークロードを，キャッシュ模擬プログラムと性能検証用のモデルそれぞれ入力し，実行時間を求めた．また，その際に表 4-4 の検証用パラメータのうちキャッシュサイズを 4MB，8MB，16MB，32MB，64MB と変更して検証した．入力したコマンドは pan –c0 –v –m10000000 である．ここで-c0 オプションはエラーが発生しても止めない，-v は SPIN のバージョン表示をする，-mN は探索する深さの設定を示している．今回は-m オプションを上記値に設定しないと検証が終了しなかったため，上記の通り設定した.

まず，図 4-14 に性能検証用モデルに対し，4MB のキャッシュサイズを設定し，ワーク

ロードを入力した際の結果を示す．1 行目の System Time が実行時間である．この例では 46,058,984 μ 秒を要した結果となった．ここで反例が出ているのは，検証に使用した目標とする処理完了時間の設定を 40,000,000 μ 秒に設定しているためである．

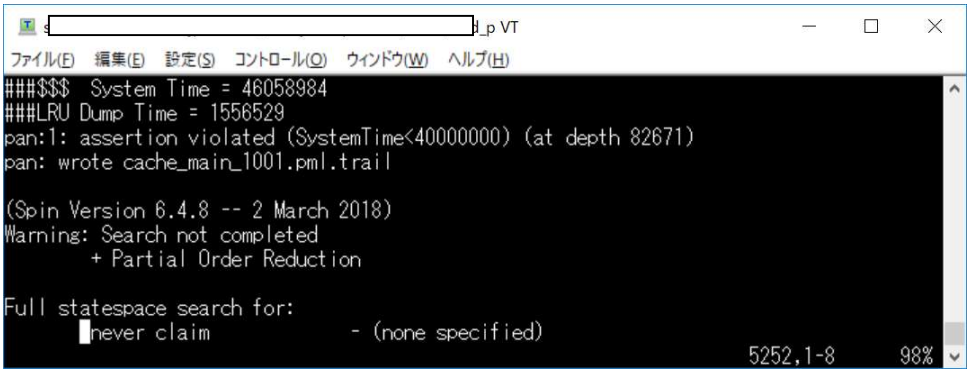

図 4-14　検証結果の例

次に上記と同条件でキャッシュ模擬プログラムを実行した．その結果を図 4-15 に示す．最終行に示した System Time が模擬プログラムの実行結果である．結果，図 4-14 の性能検証用モデルにて得られた実行時間と同じ 46,058,984 μ 秒となった．

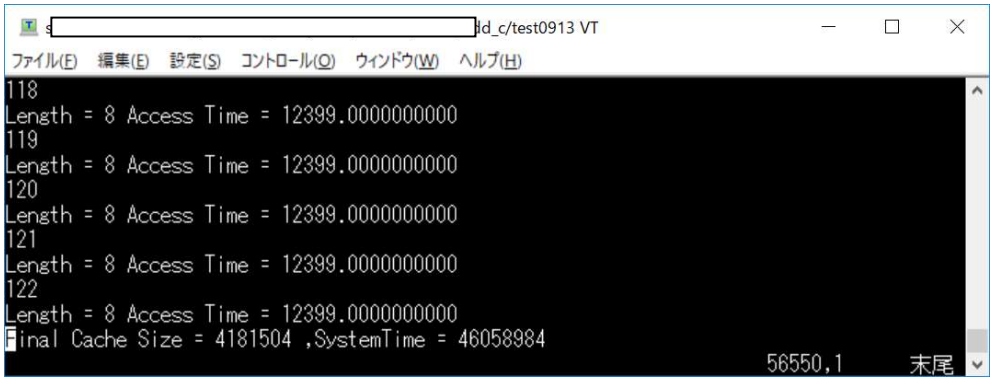

図 4-15　シミュレーションコード実行結果

上記の他，キャッシュサイズを 8MB，16MB，32MB，64MB と変更した際も，以下表 4-6 に示す通り同じ実行時間となった．この比較結果より今回我々が作成したモデルは，実行時間の検証の観点で，検証対象となるキャッシュ模擬プログラムと同様にふるまうことがわかった．以上より，性能検証をおこなうのに妥当であると判断した．

表 4-6　得られた実行時間

#	キャッシュサイズ	性能検証モデル(μ 秒)	模擬プログラム(μ 秒)
1	8MB	42,735,898	42,735,898
2	16MB	39,996,690	39,996,690
3	32MB	36,268,521	36,268,521
4	64MB	35,330,000	35,330,000

4.4.5.3　　作成した検証モデルを用いた性能検証

次に，作成した検証モデルを用いて性能検証を行った例を示す．今回は，4.3.1 に示した通り，HDD の処理コマンドを受信した後，キャッシュもしくはドライブへの直接の読み/書き処理が完了するまでの時間が，目標とする時間より短いかを検証する．そのため，今回の検証モデルでは目標時間内に終わるか，終わらないかを図 4-13 の q16:finish state に到達した際の SystemTime が，目標時間より大きくなるか小さくなるかで検証することにした．そのために，目標時間 Target Time を設定し，assert(System Time < Target Time)という検証条件をモデルに組み込み検証した．また，検証の際には，4.4.5.2 同様表 4-3 のワークロードを入力した．入力したコマンドは pan –c0 –v –m10000000 である．

上記の結果，得られた結果は 4.4.5.2 に示した通りである．図 4-14 に示す通り，例えばキャッシュサイズを 4MB に設定し，上記条件で検証した場合，実行時間は 40,000,000 μ 秒を超える．結果として 3 行目に示す通り反例が検出され，4 行目に示す通り，trail が生成された．同様に 8MB で検証した場合でも反例が検出，trail が生成された．この結果から，提案した手法によって作成したモデルにより，性能検証が可能になったと判断した．

4.5　提案手法の評価

本節では，提案した，ソースコードの再利用による性能モデル作成手法により，実際に提案した手法でモデル化工数の削減ができたかを評価する．以下その内容と結果について述べる

4.5.1　　評価する内容

本項では，評価の内容について述べる．本研究では，提案した手法が，既存の性能モデル作成手法よりも効率が良い手法なのか，またこれまで実製品における検証規模の問題に適用した場合，既存の検証手法であるシミュレータの実装と比べて効率化されているのかを評価する．そのため以下 2 点の評価をする．

評価 1)小規模な実行時間検証用の性能モデルを作成し，Promela によるモデリングと，提案手法によるモデリングを実施し，開発に必要とした工数を比較

評価 2)実製品を想定し，4.4 に示した HDD キャッシュモデルの開発にかかった工数を，既存手法（性能検証用のシミュレーションプログラムを開発）の開発工数と比較

上記の評価の詳細を以降述べる．

4.5.1.1　　小規模な実行時間検証用モデルを用いた評価

本評価では，小規模な実行時間検証をする性能モデルを複数評価し，Promela によるモデリングと提案手法によるモデリングを実施し，工数を比較する．評価項目は，以下表 4-7 に示す．検証項目はモデルの作成時間であり，その構成要素は，実装時間，評価時間である．

実装時間は，設計した内容を，既存手法である Promela や，提案手法で実際にモデルを

作成する時間である．評価時間は，作成したモデルを実際に動作させ，動作が正しいかを評価・検証する時間である．

表 4-7　評価に用いる項目

検証項目（モデル作成時間）	
実装時間	評価時間
Promela や提案手法で実際にモデルを作成する時間	モデルのふるまいが正しかを評価・検証する時間

次に，評価時に作成するモデルについて述べる．今回は実行時間を検証するモデルを作成し，指定した時間内で処理が完了するかを検証するモデルを作成する．モデルとして表 4-8 に示す 2 種類 5 モデルを作成して検証する．

表 4-8　作成するモデル

#	種別	詳細
1	ソートアルゴリズム	バブルソート
2		クイックソート
3		マージソート
4	多項式演算	順次演算
5		Horner 法

次に，評価の対象と実施内容，その流れについて表 4-9 を用いて説明する．今回は，Promela 未経験の組込みシステムエンジニアに対し，#1 に示す通り，執筆者から，Promela と提案手法の教育を 1 か月かけて実施する．次に，#2～#4 の作業を，Promela 未経験のエンジニアが表 4-8 に記載した 5 つの各アルゴリズムに対して実施する．その際に表 4-7 に記載した項目を計測する．

ここで，#2 で行う C 言語による実装は，Promela 及び提案手法で実装するアルゴリズムの学習と，#4 の提案手法を用いたアルゴリズムの実装のために行う．この意図は，実際の製品開発では流用開発を行うため，#2 の作業は既にされているという前提に立っている．

表 4-9　評価の対象と実施内容

#	実施内容	
1	執筆者による Promela と提案手法の教育	
2	モデル毎に実施	C 言語でアルゴリズムの実装
3		Promela でアルゴリズムの実装
4		提案手法でアルゴリズムの実装
5	評価	

最後に評価値の算出について述べる．評価には工数改善率を使う．工数改善率は以下（1）式を使い算出する．既存手法に対し，提案手法によって工数（作業時間）がどれだけ短縮されたかの比率である．

$$工数改善率(\%) = 100 \times \left\{1 - \left(\frac{提案手法工数(hour)}{既存手法工数(hour)}\right)\right\} \cdots (1)$$

4.5.1.2　HDD のキャッシュモデルによる評価

本評価では，実製品の開発を想定し，4.4 に示した HDD キャッシュモデルの開発にかかった工数を，既存手法（性能検証用のシミュレーションプログラムを開発）の開発工数と比較する．

評価の流れを表 4-10 を用いて説明する．#1 では，Promela の知識を既に持っているエンジニアに，4.4 に示した設計情報と，既存のキャッシュシミュレータプログラムを提供し，提案手法を用いて検証モデルを開発してもらう．#2 では#1 同様の開発を 4.5.1.1 担当のエンジニアにも実施してもらう．最後に#3 ではそれらの結果情報をもとに，提案手法によりモデル化の工数が削減できたかを評価する．評価式は 4.5.1.1 に示した(1)式を用いて，効率改善率を算出して評価する．

表 4-10　評価の流れ

#	担当者	実施内容
1	Promela の知識を持つエンジニア	4.4 設計に基づく実装
2	4.5.1.1 担当エンジニア	#1 とおなじ作業の実施
3	執筆者	#1,2 の結果に基づく評価

4.5.2　評価結果

本項では提案した，ソースコードの再利用による性能モデル作成手法により，実際に提案した手法でモデル化工数の削減ができたかを評価した結果について示す．まず，4.5.1.1 にて説明した，小規模な実行時間検証用モデルによる評価の結果を示し，その次に，4.5.1.2 にて説明した，HDD のキャッシュモデルによる評価の結果について示す．

4.5.2.1　　小規模な実行時間検証用モデルを用いた評価の結果

小規模な実行時間検証用モデルに対し，評価式(1)を用いて算出した工数改善率に基づく評価の結果を，表 4-11 に示す．今回対象とした小規模のモデリングでは，全アルゴリズムの全評価項目における平均で 52.76%の工数削減を達成した．次に，実装，評価の各工程の平均においては，実装工程が 68.64%改善されている結果となった．一方で，評価工程の効率は，36.67%の改善に留まった．

次に，担当者へのヒアリング並びに作業記録による分析結果について述べる．まず，全体的に効率が改善している理由は，提案手法の目論見通り，モデル化する箇所がソースコードの一部だけになっているため，作業が削減された結果であった．次に，評価工程の改善が他項目より低い理由について分析した結果について述べる．この理由は，本来再利用した箇所は，評価の対象外のはずだが，Promela 化した部分と，再利用部分のインタフェース間における整合性を確認する作業が発生したため，想定よりも評価の工数が増加し，その結果工数改善率が低くなった結果であった．

表 4-11　小規模な実行時間検証用モデルによる工数改善率評価の結果

#	評価項目	ソートアルゴリズム			多項式		平均
		バブル	クイック	マージ	順次	Horner 法	
1	実装	83.33	80.00	75.00	52.94	52.94	68.84
2	評価	66.67	66.67	50.00	0.00	0.00	36.67
3	平均	75.00	73.34	62.50	26.47	26.47	52.76

最後に，アルゴリズムの種別で効率に差が出た結果について表 4-12 を用いて分析する．表 4-12 は，実装した各アルゴリズムの Promela，C 言語（提案手法にて再利用するためのコード，アルゴリズムの学習用），提案手法によりモデル化した際それぞれのステップ数を記載した結果である．この結果を見てわかる通り，多項式アルゴリズムの実装コードが，ソートアルゴリズムと比べ大幅に少ない（開発規模が小さい）ため，Promela によるモデル化に時間が多くかからず，結果として効果が下がったことが判明した．

表 4-12　各モデルのステップ数

#	評価項目	ソートアルゴリズム			多項式	
		バブル	クイック	マージ	順次	Horner 法
1	Promela	57	151	109	16	24
2	C 言語 (再利用)	93	101	98	27	29
3	提案手法	67	142	102	31	33

4.5.2.2　　HDD のキャッシュモデルによる評価の結果

HDD のキャッシュモデルに対し，評価式(1)を用いて算出した工数改善率に基づく評価の結果を，表 4-13 に示す．この結果より提案手法は，経験者，未経験者双方で開発効率が改善し，平均 69.07%開発工数が効率化する結果となった．

表 4-13　HDD のキャッシュモデルによる工数改善率評価の結果

	既存方式	提案手法		平均 (提案手法)
担当者	Pormela 経験者		Promela 初心者	
開発工数(日)	20	5.75	6.63	6.19
工数改善率	-	71.25	68.88	69.07

4.6　まとめ

本章では，組込みシステムにおける性能検証について現状の問題を分析し，以下 2 点の課題を導出した．一つは大量の機能を含む製品を短期に開発するために，差分開発を行っている．そのため，モデル検査導入時に大量のソースコードからモデルを作成しなくてはならず，多大な工数が発生すること．二つ目は，既存の検証技術を使うにはモデルの作成の工数，性能検証用のモデルに変更する工数に加え，動作の妥当性の説明・証明の工数が追加されるため，更に工数が増加することである．

このような課題に対し，我々は，モデル化対象の既存のソースコードを再利用し，必要最低限のモデル化で性能検証を実現する手法により，性能モデル検査導入を効率化しつつ，モデルの動作を設計者が理解可能にする手法を提案した．

また，上記提案した手法を，ソートアルゴリズム 3 種と多項式演算 2 種からなる小規模な実行時間検証用モデルと実製品を想定した HDD のキャッシュモデルを用いて評価を行った．その結果，4.5 に示した通り， ソートアルゴリズムなど小規模のモデル開発で平均 52.76%，実製品開発規模である HDD のキャッシュモデルで平均 69.07%の開発時間削減を達成することができた．

第5章　結論

5.1　まとめ

本研究では，高い品質を維持したままコンシューマ向け組込みシステムの製品開発を効率化することを目的とし，執筆者所属企業における製品開発事例を通し，上記目的を達成するために以下二点の課題解決が必要であることを明らかにした．

課題１：デバッグ工程における不具合解決の効率化
課題２：設計工程における性能検証の効率化

執筆者はこの二つの課題に対し，それぞれ更に社内事例を分析した．課題 1 に対しては，製品開発時のデバッグ工程における 2600 件の作業情報を分析し，デバッグ工程において再現性の低い障害を発生時に確実に記録すること，膨大な解析対象の絞り込みと，解析情報の理解容易化が必要であることを明らかにし，長時間トレース機能，解析対象絞り込み機能，形式変換機能からなる障害解析環境の提供によるデバッグ工程における効率改善に取り組むことにした．課題 2 に対しては，課題解決のニーズが高かったストレージ製品の開発担当者に対しヒアリングをし，対策を検討した．その結果，これまで用いてきた待ち行列理論など統計を用いた性能見積もり技術では，派生開発が多い製品の開発では見積もり精度が出ないため，モデル検査を用いて効率的かつ網羅的な検証に着手したいというニーズを明らかにした．その上で製品開発が差分開発になっており，性能検証モデルを作成する際に，流用箇所も含めてモデル化する工数が大きな課題であることと，モデルの妥当性の説明・証明の工数が必要なことも明らかにした．以上より，差分開発における性能モデル検査技術の導入を目的とした，既存ソースコードを流用した性能検証用モデル開発の効率化を提案し，評価した．

本論文における技術的な特徴は，課題 1 に記載した 3 点と，課題 2 に記載した 1 点の計 4 点である．それぞれの概要を以下に記す．

課題１に対して開発した３つの技術の特徴を以下に列挙する．一つ目の特徴は，膨大なトレース情報から，効率的に問題点を検出する①解析対象絞り込み機能である．解析対象絞り込み機能は，それまで CPU リソースのボトルネック解析で使われていたプロファイラを，我々の製品開発の知見から複数のタスクの組み合わせで発生する不具合の解析対象の絞り込みに使えることを明らかにし，解析効率を改善した．二つ目の特徴は，リソース競合などタイミングに依存して発生するような再現性の低い障害発生時における挙動情報を確実に記録するための②長時間トレース機能である．長時間トレース機能は，組込みシステム固有の課題（1.エンタープライズ系のシステム開発では，OS トレースの時間が問題になることがあまりなかった．2.組込みシステムの開発においては JTAG-ICE などによるトレースの

実現が主流であったが，OS トレースの容量や導入の工数が問題であった）を明らかにしたこと．その課題に対し，OS トレースの結果をカーネル領域のメモリに記録するタイミングにて外部出力バス経由で出力し，市販されている記録装置に保存することにより，低負荷で長時間トレースを実現し，再現性の低い不具合が偶然再現した際に常時解析対象の挙動情報を記録可能にした．三つ目の特徴は，取得し，絞り込んだトレース結果を，これまでの障害解析の知見を用いて可視化し，解析を容易化・効率化する③形式変換機能である．それまでのトレースや可視化の取り組みは，特定のトレーサを可視化することに特化した取り組みや，収集できるトレース情報を汎用的に可視化できるよう，トレース内容を一般化する方向で研究が進んでいた．それに対し本研究では，これまでの製品開発の知見を利用し，再現性の低い不具合で解析が必要な，リソースの競合に代表される複数タスク間の競合や，割り込み間隔の確認，不具合の原因となる OS 機能の利用と動作のタイミングといった可視化項目を明らかにし，形式変換機能を実現した．

最後に課題 2 に対して開発した，四つ目の技術の特徴について述べる．4.2 に述べた課題に対し我々は，モデル化対象の既存のソースコードを再利用し，必要最低限のモデル化で性能検証を実現する手法を提案した．それにより，性能モデル検査導入の効率化だけでなく，製品コードの流用によりモデルの動作を設計者が理解可能にした．

評価の結果，それぞれ以下の結果を得た．課題 1 に対する障害解析環境の提供によるデバッグ工程における効率改善では，本研究で開発した障害解析環境を 6 製品 24 件の障害に適用し，18 件の障害解決を実現した．そのうち 1 例として既存手法では解析に 8 日間かかっていた問題を，1.91 日に短縮し，作業工数を 76.13%削減することができた．この提案する障害解析環境の採用により，これまで行ってきた不具合再現条件特定のための試行錯誤，テスト実施（トレース取得）のための試行錯誤と可視化手法検討の試行錯誤を不要にすることができた．一方で，プロファイルと分析箇所選択による解析対象の絞り込み作業は増加した．ここで，提案手法による解析対象の絞り込みでは，障害の内容によっては解析に必要なトレース情報を可視化対象外にする可能性がある．その場合は障害要因を含むトレース情報を可視化するまで解析を続ける必要があるため，表 3-5 に示したような効率化はできないことも考えられる．しかし，解析対象の絞り込みで行う分析箇所選択はトレース結果のフィルタリングなので，それまでのテスト実施と可視化手法検討に対して作業としては軽微なため，原因分析の試行錯誤の工数を減らすことが期待できる．また，解析に必要なトレース情報は長時間トレース機能で取得できているため，これまでのような試行錯誤を伴う作業をすることなく障害要因を特定することができる．

課題 2 に対する提案である既存ソースコードを流用した性能検証用モデル開発の効率化では，提案した手法を，ソートアルゴリズムのような小規模な開発と，実製品の位置機能である HDD のキャッシュ機能の開発の二点における性能検証モデル作成にて評価した．その結果，ソートアルゴリズムなど小規模のモデル開発で平均 52.76%，実製品開発規模である HDD のキャッシュモデルで平均 69.07%の開発時間削減を達成することができた．

最後に，本研究の特徴が今回の研究を通して生まれた点について考察した．コンシューマ向け組込みシステムの開発は，2.3.2 に示すとおりデジタル化を契機に LinuxOS の導入や高機能化が進んだ．また 2.4.2 に示すように，仕向け先や法規対応のための派生開発の増加も進んだ．一方で 1.4 に示した開発の低コスト化・短期化に対する要求や，製品コスト削減のための搭載ハードウェアの低性能化や削減という状況は変わらず，結果として本研究で導出した課題の解決が必要となった．

5.2 今後の課題

本研究では，コンシューマ向け組込みシステム開発における製品開発を効率化するため，コンシューマ向け組込みシステムのデバッグ工程における不具合解決の効率化，設計工程における性能検証の効率化の二点に取り組んだ．

この結果，執筆者の所属企業で抱えていた課題を解決した．これらの技術のうち障害解析技術は今も運用を続けている．一方で， 取り組んだ上記二点については，それぞれ課題が残っている．障害解析効率化の提案機能開発で採用したツールや開発環境自体がメンテナンスを終了するなど，ユーザ数が減った結果別のものに移行しなくてはならないといった「障害解析環境の維持」という新しい課題が発生している．

上記課題の具体例を，我々の製品開発事例を基に紹介する．一つ目の事例は開発者（製品及び障害解析環境）の使用する言語のトレンドが変化，その変化に追従するため形式変換機能の実装を 2014 年に Perl から Python に変更した件である．この修正は，トレース機能と形式変換機能で 1 か月程度の作業が発生している．二点目の事例は，トレース可視化用のフロントエンドツール TimeDoctor のメンテナス終了に伴い，KernelShark[62]に代表される他ツールへの変更が今現在必要となっている問題であり，こちらの問題はまだ解決できていない．

このような課題については，ツールや開発環境とのインタフェースを抽象化し，ツールへの依存性を下げることで，「障害解析環境の維持」にかかる維持作業工数を削減する方法について検討をする必要がある．また，本研究で扱った OS トレーサについては，当初，自社も開発にかかわった LKST[32][33][34]を使用し一定の成果を得た一方，最終的には Linux カーネルに組み込まれた OS トレーサである ftrace[38]へ移行した．その理由は，LKST が Linux カーネルに組み込まれなかったため，自分達で OS トレーサの維持・管理をする工数が負担になったからである．このような状況を回避するためには，自分達でツールを作るだけでなく，コミュニティに積極的に情報を発信し，ツールを多くの開発者に知ってもらうことや，コミュニティ活動の中で標準化に取り組むこと，ツールのメンテナンスにも積極的に参加することなどが必要である．

次に，既存ソースコードを流用した性能検証用モデル開発の効率化にて提案した手法に関する課題について述べる．我々は性能の要素である「実行時間」「スループット」[60]のうち，「実行時間」のみを評価対象としている．そのため，性能のもう一つの要素である「ス

ループット」における評価をしなくてはならない．また，別の課題として状態爆発の課題がある．モデル検査はソフトウェアが取りうる状態を網羅的に探索するため，調べる状態が非常に多くなる問題がある．この問題は実行が分岐するだけ，乗算で状態数が増加する[63]．そのため，コンシューマ向け組込みシステムの開発のように，差分開発をしており，モデル化対象のソフトウェア規模が大きい開発において，検証する範囲を拡大するなど，内容を複雑にすると，状態爆発が大きな課題になることが考えられる．

上記課題の具体例を，我々の製品開発事例を基に紹介する．まずは性能の要素であるスループットは，本研究でも取り組んだハードディスクにおいても，性能の指標として採用されているため，その検証実現が求められている．また，デジタルテレビ製品でも， CPU とメモリを繋ぐバスのスループットが，同時録画や再生などで問題になることがあり，同じく検証の実現が求められている．次に，状態爆発であるが，我々がキャッシュ機能を検証する際に，HDD のコマンド列を非同期に生成・発行するようなテスト用のモデルをつくり検証したところ，簡単に状態爆発が発生してしまった．

これら述べた課題については，本研究で取り組んだように，実製品開発の中で評価したい項目から研究項目を明確化し，取り組むことで具体的な解決方法を検討する必要がある．近年の事例では，自動車分野の自動運転用 ECU のソフトウェア開発などにおいて，画像情報を多く扱うことからバスのスループットを考慮して開発しなくてはならない製品がある．そのような製品への適用を通して，スループットに対する性能検証手法の具体的な研究が可能と考える．また，状態爆発の面については，これまでも多くの手法が検討されており，既にアルゴリズムの最適化や，効率的なデータ表現法などが進んでいる[63]．よって今後は，それらのアルゴリズムの適用の仕方，効率的なデータ表現法を用い，いかに効率の良いモデルを作るか，製品開発事例を通じて積みあげ，使用法のナレッジベースを構築することが有効と考えられる．

謝辞

本論文は，執筆者が株式会社日立製作所において行った研究を中心に，トップエスイー並びに電気通信大学にてご指導を頂き，研究成果としてまとめたものです．執筆にあたり，大変多くの方々にご指導，ご協力を頂きました．ここに感謝の意を表します．

博士課程の在学期間，指導教員として多大なご指導を頂いた電気通信大学大須賀昭彦教授，田原康之准教授には，研究の差別化ポイント，論文執筆など研究活動全般にわたり多大なご指導を賜りました，また進捗が思わしくない際には温かいお言葉もかけて頂きました．ここに深く感謝致します．

トップエスイーの修了制作におけるモデル検査を用いた性能検証に関する取り組みの立ち上げから，今日に至るまで，早稲田大学の吉岡信和上級研究員には，研究の方向性に関する議論，進め方，論文執筆など，多大なご指導・ご助言を頂きました．ここに深く感謝致します．

本論文をまとめるにあたり，大変ご多忙な中審査を快く引き受けていただき，論文や発表内容に対して多くの有益なご指導・ご助言を頂きました電気通信大学大学院の田中健次教授，広田光一教授，石川冬樹客員准教授に深く感謝致します．

研究の立ち上げ時に組込みシステム研究者・開発者としての基本をご指導頂き，供に業務に取り組ませて頂きました日立製作所の小口琢夫氏，茂岡知彦氏に深く感謝いたします．

博士課程に入学する機会を与えて頂いた日立システムズの藤林昭氏，本取り組みを進める上で業務上の様々なご指導を頂いた日立アステモの芹沢一氏，日立製作所の寺岡秀敏氏，岡田明正氏，前岡淳氏に深く感謝致します．また業務面でご支援いただきました今井光洋氏，亀山達也氏をはじめとする職場の皆様に深く御礼申し上げます．

最後に，大須賀研究室の皆様のご協力に御礼申し上げると共に，博士課程での研究活動や，社会人学生生活を心身両面から支えてくれた家族に感謝致します．

参考文献

[1] IEEE Spectrum : THIS CAR RUNS ON CODE, https://spectrum.ieee.org/this-car-runs-on-code, (2022 年 4 月 3 日現在)

[2] 梶本一夫: デジタル家電ソフトウェアものづくり, https://www.jst.go.jp/crest/crest-os/osddeos/event/200811/ET2008-kouen02.pdf (2022 年 4 月 3 日現在)

[3] IPA: 2011 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, 2012.

[4] IPA: 2019 年度組込み/IoT 産業の動向把握等に関する調査」事業“組込み/IoT に関する動向調査”調査報告書, IPA, 2019

[5] 高田広章: 組込みシステム開発技術の現状と展望，情報処理学会論文誌，Vol.42, No.4,pp.930-938, 2001

[6] 中本幸一，高田広章，田丸喜一郎 : 組込みシステム技術の現状と動向，情報処理学会論文誌，Vol.38, No.10, 1997

[7] 高田広章 : 組込みシステム概要，映像情報メディア学会誌, Vol.63, No.1, pp49-51, 2009

[8] 金丸輝康 : 自動車 3 社の製品ラインとフルモデルチェンジ，大阪学院大学商・経営学論集,第 41 巻 1 号,pp35-64, 2015

[9] JASIC(Japan Automobile Standards Internationalization Center):自動車法規フォロー, https://www.jasic.org/j/08_publication/hh/10_regulations.htm (2022 年 4 月 3 日現在)

[10] 一般社団法人電波産業会 : デジタル放送システムの ARIB 標準規格体系, https://www.arib.or.jp/kikaku/kikaku_taikei/taikei02.html（2022 年 4 月 3 日現在）

[11] 桑原禎司: デジタル家電のプラットフォーム技術，日立評論, Vol.87, No.5, pp.499-504, 2005

[12] 上野秀剛，亀井靖高，門田暁人，松本健一 : 原価率とプロジェクトメトリクスに着目したソフトウェア開発プロジェクトの特徴分析，プロジェクトマネジメント学会誌, Vol.12,No.5,pp.25-30,2010

[13] MONOist，トヨタ自動車が車載 Linux「AGL」を車載情報機器に全面採用，「他社も続く」: https://monoist.itmedia.co.jp/mn/articles/1801/12/news033.html（2022 年 3 月 27 日現在）

[14] 平林雅之，鵜林尚靖 : ソフトウェア工学，オーム社，2017

[15] 岡本周之 : 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，大阪大学大学院情報科学研究科博士論文，2016.

[16] 経済産業省 : 組込みソフトウェア産業実態調査報告書について : https://warp.da.ndl.go.jp/info:ndljp/pid/1167911/www.meti.go.jp/policy/mono_info_service/joho/ESIR/index.html（2022 年 6 月 27 日）

[17] IPA : 2012 年度「ソフトウェア産業の実態把握に関する調査」の報告書を公開, https://www.ipa.go.jp/sec/reports/20130426.html（2022 年 6 月 27 日）

[18] IPA :「2018 年度組込みソフトウェア産業の動向把握等に関する調査」事業「組込み/IoT に関する動向調査」の調査結果を公開, https://www.ipa.go.jp/ikc/reports/20190327.html（2022 年 6 月 27 日）

[19]IPA : 組込み/IoT に関する動向調査, https://www.ipa.go.jp/ikc/our_activities/rs_05.html（2022 年 6 月 27 日）
[20]來間啓伸： B メソッドによる形式仕様記述，近代科学社，2007
[21]Alexandre Riazanov, Andrei Voronkov: The design and implementation of VAMPIRE, AI Communications, Volume15, Issue 2,3 pp91-110, 2002
[22]INRIA, The Coq Proof Assistant, available from, https://coq.inria.fr/,(2022 年 4 月 3 日現在)
[23]Holzmann, G : The model checker SPIN, IEEE Transactionsions on software engineering ,Vol 23, No 5, IEEE, pp279-295, 1997
[24]Trivedi, K : Probability and Statistics with Reliability, Queuing, and Computer Science Applications. Wiley , 2001
[25]Liu, H : Software performance and scalability, Wiley, 2009
[26]Qinriu, Q. and Pedram, M: Dynamic power management based on continuous-time Markov decision processes, Proc. Design Automation Conference, IEEE, pp.555-561 , 1999
[27]Alur, R. and Dill, D : A theory of timed automata, Theoretical Computer Science, Vol.126, pp183-235, 1994
[28]Havelund, K. Skou, A. Larsen, K, et al : Formal Modelling and Analysis ofan Audio/Video Protocol: An Industrial Case Study using UPPAAL, Proc. Real-Time System Symposium, IEEE , pp.2-13, 1997
[29]Nagaoka, T. Ito, A. Okano, K. et al : QoS Analysis of Real-time Distributed System Based on Hybrid Analysis of ProbabilisticModel Checking,IEICE Transactions on Information and Systems, Vol.E94-D, No.5, pp.958-966, 2011
[30]Mark Fewster, Dorothy Graham：システムテスト自動化標準ガイド，凸版印刷株式会社，2014
[31]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, Vol.J90-D ,No.10 ,pp.2663-2674, 2007
[32]IPA, OSS 性能・信頼性評価，障害解析ツール開発プロジェクト ―OSS ミドル評価手順書と Linux 障害解析ツールを開発― , https://www.ipa.go.jp/about/jigyoseika/04fy-pro/open/2004-621d.pdf(2022 年 4 月 3 日現在)
[33]畑崎恵介，中村哲人，芹沢一:システムの挙動に対応して動作の切替えが可能なイベントトレーサ LKST の開発，情報科学技術フォーラム一般講演論文集 2002(1), pp.177-178, 2002
[34]中村哲人，畑崎恵介， 芹沢一:カーネル拡張モジュール機能を用いた OS デバッグの実現，第 65 回全国大会講演論文集 2003(1), pp.11-12, 2003
[35]Patricia, C and Aurelie, B et al : Debugging embedded multimedia application traces

through periodic pattern mining, Proc. 10th ACM international conference on Embedded Software(EMSOFT'12), pp 13-22, 2012

[36]Paul, G. and Bharat, J : Methodology and archtecture of JIVE, Proc. ACM symposium on Software visualization(SoftVis '05), pp95-104, 2005

[37]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 TraceLogVisualizer (TLV),コンピュータ ソフトウェア,2010,27 巻,4 号,p. 4_8-4_23, 2010

[38]Steven rostedt ftrace – Function Tracer , https://www.kernel.org/doc/Documentation/trace/ftrace.txt (2022 年 4 月 3 日現在)

[39]Mohamad Gebai, Michel R. Dagenais: Survey and analysis of kernel and userspace tracers on Linux: design, implementation, and overhead, ACM Computing Surveys, vol. 51, no 2, 2018

[40]Woodside, M. Franks, G. and Petriu, C : The Future of Software Performance Engineering, Proc. Future of Software Engineering(FOSE '07) , IEEE Computer Society ,pp. 171-187, 2007.

[41]Barber, S : Creating Effective Load Models for Performance Testing with Incomplete Empirical Data, Proc. 6th IEEE Int. Workshop on Web Site Evolution, IEEE Computer Society, pp. 51-59, 2004

[42]Oracle(R): Database Performance Tuning Guide, https://docs.oracle.com/cd/E11882_01/server.112/e41573/toc.htm, (2022 年 4 月 3 日現在)

[43]須永高浩，笹田耕一，Ruby 用リアルタイムプロファイラの設計と実装，情報処理学会論文誌　プログラミング, Vol4, No 3, pp1-15, (2011)

[44]Smith, C. and Williams, L : Performance solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley Publishers, 2001

[45]鵜林尚靖：組み込みソフトウェアの設計モデリング技術，　情報処理，Vol45，No 7, pp682-689, 2004

[46]Hamadi, R. and Benatallah, B: A Petri net-based model for web service composition, Proc. 14th Australasian database conference, pp191-200, Australian Computer Society, Inc, 2003

[47]Object Management Group : UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, http://www.omg.org/spec/MARTE/, (2022 年 4 月 3 日現在)

[48]吉岡信和，田辺良則，田原康之，長谷川哲夫，磯部祥尚：モデル検査による設計検証，コンピュータソフトウェア，Vol31, No4, pp40-65, 2014

[49]Kim, M. and Kim, Y : Automated Analysis of Industrial Embedded Software,　Proc. 9th International Symposium(ATVA 2011), Springer-Verlag Gerlin Heidelberg, pp. 51-59,

2011
[50]Tiwari,V. Malik, S. and Wolfe, A　: Power Analysis of Embedded Software: A First Step Towards Software Power Minimization, IEEE Transactions on VLSI Systems, Vol2, pp. 437-445, 1994
[51]Edmund, C. Daniel, K. and Flavio, L : A tool for checking ANSI-C programs, Proc. 10th international conference on Tools and Algorithms for Construction and Analysis of Systems(TACAS '04), Springer, pp. 168-176, 2004
[52]Holzmann, G. and Smith,　M : Automating software feature verification, Bell Labs Technical Journal, Vol5, Issue2, ,pp.72-87, 2000
[53]竹辺靖昭，鈴木康文，川上真澄:モデルベース開発における異種モデル間連携基盤の開発，研究報告組込みシステム（EMB）2010.34, pp1-8, 2010
[54]田中勇樹，石郷岡祐他： 高応答性と統合容易性を両立するマルチコア活用ソフトウェア統合ミドルウェア，情報処理学会研究報告, Vol.2019-EMB-50 No.6, 2019
[55]Jonathan B. Rosenberg 著，吉川邦夫訳: デバッガの理論と実装 アスキー出版局, 1998
[56]長野 岳彦，亀山 達也 ：組込みトレース技術とその応用，研究報告組込みシステム（EMB），2011-EMB-20, PP1-6, 2011
[57]長野岳彦: OProfile porting on MIPS architecture, CE Linux Forum Japan technical jamboree #16 講演資料 , 2007
[58]池田信之，今村紀子，高田沙都子：実用化に向けたモデル検査適用手法の開発，東芝レビュー, Vol.62, No.9，pp46-49, 2007
[59]Andreas Gustafsson: egypt – create call graph from gcc RTL dump, (online), https://www.gson.org/egypt/egypt.html, (2022 年 4 月 3 日現在)
[60]Patterson, D. Hennessy, J. 成田光彰(訳）：コンピュータの構成と設計第 3 版（上），日経 BP 社, 2006
[61]Jacob, B. Ng, S. and Wang, D : Memory Systems Cache, DRAM, Disk, Morgan Kaufmann Publishers, 2008
[62]Kernel Shark : available from https://kernelshark.org/, (2022 年 4 月 3 日現在)
[63]吉岡信和，青木利晃，田原康之：SPIN による設計モデル検証，近代科学社，　2008

研究業績

1. 査読付き学術雑誌

[1] 長野岳彦，小口琢夫，吉岡信和，田原康之，大須賀昭彦：組込みシステム向け障害解析環境の効率改善，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 12 No2,pp27-37(2022/5)

※本論文の第３章に関連する業績

2. 査読付き国際会議発表

[1] Takehiko Nagano, Kazuyoshi Serizawa, Nobukazu Yoshioka, Yasuyuki Tahara and Akihiko Ohsuga: Performance Exploring Using Model Checking A Case Study of Hard Disk Drive Cache Function, Proc. The Tenth International Conference on Software Engineering Advances(ICSEA2015), IARIA, pp31-39（2015/11）

※本論文の第４章に関連する業績

3. 国内口頭発表

[1] 長野岳彦，小口琢夫，吉岡信和，田原康之，大須賀昭彦：組込みシステム向け障害解析環境の効率改善，情報処理学会研究報告コンシューマ・デバイス＆システム(CDS),2022-CDS-32(5),pp1-8,(2021/8)

[2] 長野岳彦，吉岡信和，田原康之，大須賀昭彦：既存プログラムを再利用した効率的な性能検証法，情報処理学会研究報告コンシューマ・デバイス＆システム(CDS),2019-CDS-24(27),pp1-8(2019/1)

[3]細木浩二，長野岳彦，石川誠，井上和則：HDD 向けキャッシュ性能シミュレーションによる I/O 性能設計の一手法，情報科学技術フォーラム講演論文集，11(1),pp305-306,(2012/9)

[4]長野岳彦，吉岡信和：モデル検査技術を用いた性能最適化手法の提案，ソフトウェア工学の基礎 18 日本ソフトウェア科学会 FOSE2011,pp257-258(2011/11)

[5]大林浩気，長野岳彦，茂岡知彦：待ち行列理論による抽象化を用いたモデル検査手法の検討，情報科学技術フォーラム講演論文集，11(1),pp239-240,(2012/9)

[6]長野岳彦，亀山達也：組込みシステムトレース技術とその応用，情報処理学会研究報告

組込みシステム(EMB),2011-EMB-20(22),pp1-6,(2011/3)

[7]長野岳彦，亀山達也：組込みシステムのソフトウェア障害予兆検出に適した挙動情報収集手法の検討，情報科学技術フォーラム講演論文集，9(1),pp445-446, (2010/8)

[8]長野岳彦，今井光洋：長時間トレース技術を用いた組込みソフトウェア開発の効率向上に関する検討，情報科学技術フォーラム講演論文集，8(1),pp499-500, (2009/8)

4. その他テクニカルカンファレンスなどにおける発表

[1]長野岳彦:OProfile porting on MIPS architecture, CE Linux Forum Japan technical jamboree #16(2007/8)

[2]茂岡知彦，長野岳彦：Linux Kernel State Tracer(LKST)の組込み向けポーティングと活用事例，CE Linux Forum Japan technical jamboree #6(2006/1)

5. 受賞歴

[1]IARIA, The Tenth International Conference on Software Engineering Advances(ICSEA2015), Best Paper Award, 2015

[2]情報処理学会，CDS 研究会，優秀発表賞，2021

[3]情報処理学会，山下記念研究賞，2022

関連論文の印刷公表の方法及び時期

1. 査読付き学術雑誌

[1] 長野岳彦，小口琢夫，吉岡信和，田原康之，大須賀昭彦：組込みシステム向け障害解析環境の効率改善，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 12 No2,pp27-37(2022/5)

※本論文の第３章に関連する業績

2. 査読付き国際会議発表

[1] Takehiko Nagano, Kazuyoshi Serizawa, Nobukazu Yoshioka, Yasuyuki Tahara and Akihiko Ohsuga: Performance Exploring Using Model Checking A Case Study of Hard Disk Drive Cache Function, Proc. The Tenth International Conference on Software Engineering Advances(ICSEA2015), IARIA, pp31-39（2015/11）

※本論文の第４章に関連する業績

著者略歴

長野　岳彦

1995 年 4 月　東京学芸大学　教育学部　情報環境科学課程　教育情報科学専攻入学
1999 年 3 月　東京学芸大学　教育学部　情報環境科学課程　教育情報科学専攻卒業
1999 年 4 月　北陸先端科学技術大学院大学　情報科学研究科　博士前期課程入学
2001 年 3 月　北陸先端科学技術大学院大学　情報科学研究科　博士前期課程卒業
2001 年 4 月　株式会社　日立製作所　入社
2014 年 9 月　電気通信大学　大学院情報システム学研究科　博士後期課程　社会知能情報学専攻入学
2022 年 12 月　電気通信大学　大学院情報システム学研究科　博士後期課程　社会知能情報学専攻修了