FeliCa IC チップ開発への実行可能な形式仕様記述の実践に基づく設計・構成法の提案

中津川, 泰正
Graduate School of Information Science and Electrical Engineering, Kyushu University

https://doi.org/10.15017/21763

出版情報：九州大学, 2011, 博士（工学）, 課程博士
バージョン：
権利関係：

FeliCa IC チップ開発への
実行可能な形式仕様記述の
実践に基づく設計・構成法の提案

中津川 泰正

平成 24 年 2 月

九州大学大学院システム情報科学府

情報工学専攻

論文梗概

組織的に開発を進める必要があるソフトウェア開発現場において, 上流工程の文書を中心に多くの問題を抱えており, 品質の高いシステムを組織的に効率よく開発することができる手法として, フォーマルメソッドが注目されている. フォーマルメソッドの利用に関して, 3 段階の適用レベルが示されることがあるが, そのうち, 本論文では, 形式仕様記述を行うレベル 0 の段階を扱う. レベル 0 の段階とは, 証明までは行わないが, 数学的な記法を用いて厳密な仕様を記述する段階である. 筆者らは FeliCa IC チップ開発を通してレベル 0 における形式仕様記述手法の適用の効果を得ることができた. しかしながら, Parnas は 2010 年に発表した論文の中で, フォーマルメソッドが実践的手法として広く利用されるような「当然の技術」として, 開発現場に浸透していない点を指摘した. 本論文では, 「当然の技術」として産業界に浸透するための課題を分析し, 形式仕様記述手法を適用する際に必要不可欠となる形式仕様記述手法の設計・構成法を提案する.

形式仕様記述手法が広く利用されるためには, 形式仕様記述手法の専門家ではない人が, 形式仕様記述を読み, 理解し, 活用していく必要がある. つまり, そのような形式仕様記述を書くための技術と, 活用するための技術が必要である. 本論文では, 以下の 3 つの提案を行う. 1 つ目は, 理解容易性に優れた形式仕様を記述するための, 形式仕様記述の設計・構成法の提案である. 2 つ目は, 従来からある既に開発現場に浸透しているレビューやテスト手法といった検証技術を主体として, その中で形式仕様記述手法を活用していく方法の提案である. 3 つ目は, 筆者らが使用してきた形式仕様記述手法では取扱いが難しかった, 動的な振る舞いに関する検証方法の提案である.

1 つ目の, 理解容易性に優れた仕様を記述するためには, Hayes と Jones が指摘した「実行可能性の影響」と Jackson が指摘した「実装の影響」を考慮する必要がある. まず, 「実行可能性の影響」とは, 仕様アニメーションとして知られている方法により記述した仕様を動作させる場合, この「動かすこと」を仕様記述の目的に含めてしまうことにより, 仕様の理解容易性に影響を与えるというものである. 次に, 「実装の影響」とは次のような課題である. 一般に, 「解決すべき問題」を定義した仕様と, 「問題の解決方法」を定義した設計を分けることで, 仕様策定者と設計者が分業して組織的に開発を行うことができると言われている. しかし, 実際には, この 2 つを分けることは容易ではないため, 仕様書に「問題の解決方法」を含めてしまうことで, 設計者を過剰に制約してしまうという課題である. 「実行可能性の影響」も「実装の影響」も, 仕様の記述が複雑になり, 過剰に設計を制約してしまうことが課題である. これらの課題を解決するために, 「実行可能性の影響」に対しては, 仕様伝達部と非伝達部からなる仕様記述スタイルを提案する. 「実装の影響」に対しては, 仕様と設計において定義するデータ構造に着目した形式仕様記述の設計・構成法を提案する.

2 つ目の提案として, 形式仕様記述手法の専門家ではない, 様々なレビューの目的を持った読者にとって, 読みやすい仕様について分析を行い, レビューの目的とテストの目的を考慮した仕様記述フレームワークを提案する. 構造が簡潔な点とテストとの親和性の点で, ディシジョンテーブルの構造を用いる方法と, 仕様の概要と詳細を区別して記述する方法を提案する. また, 実際に, 形式仕様記述から体系的にテスト項目を作成し, テスト項目とテストスクリプトの網羅性を機械的に検証する方法を提案する.

3 つ目は, 形式仕様記述手法では取扱いが容易でない動的な振る舞いに関する課題に対して, 同時並行に動作する動的な振る舞いをモデル検査を用いて検証する方法を提案する.

本論文では, これらの形式仕様を記述するための方法と活用するための方法を提案することによって, 形式仕様記述手法が, 産業界において適用可能で有効な手法であることを開発実践を通して示した.

九州大学大学院 システム情報科学府 情報工学専攻

目 次

第1章

序論

本章では, まずソフトウェア開発の現状を述べ, フォーマルメソッドの適用における課題を考察する. 次に, 本研究において扱うフォーマルメソッドとして, 形式仕様記述手法 VDM とモデル検査の概要を述べる. 最後に, 本研究の基となる筆者らのフォーマルメソッドの適用事例を紹介し, 本研究において扱う課題の概要を述べる.

これらの課題を本章に続く 2 章と 3 章において詳細に議論する. 4 章以降では, それぞれの課題に対する解決方法を提案していく.

1.1 ソフトウェア開発の現状

ソフトウェアは, 社会基盤となる情報システムや日常生活に必要不可欠な様々な製品に組み込まれて利用されることが多いため, ソフトウェアの信頼性や安全性をいかに確保するかということが, 社会的に重要な課題となっている. また, 開発対象が大規模, 複雑化していく中で, 品質の高いソフトウェアを効率的に開発するために, 開発現場ではソフトウェア開発工程全体をいくつかの工程に分け, 専門性を持った人たちからなるチームが各工程を担当することで, 分業して組織的に開発を進める必要がある.

工程は, 開発現場ごとに様々ではあるが, 要求分析, 仕様策定, 設計検討, 実装, テスト, 運用・保守に分かれていることが多い. 一般に開発工程全体をソフトウェアライフサイクルと呼び, 開発の進め方を開発プロセスと呼ぶ. 本節では, まず本研究において想定す

る開発プロセスについて述べた後に, ソフトウェア開発における現状について述べる.

1.1.1　本研究が想定する開発プロセス

図 1.1 に本研究において前提とする開発プロセスを示す. 図は, ソフトウェア開発のライフサイクルを模式的に表す上で一般に用いられている「 V 字型モデル」に準じたものである. ウォータフォール型のライフサイクルモデル [Roy70] を拡張した「 V 字型モデル」は, 左側に要求分析と仕様策定と設計検討と実装の流れを示し, 右側にはテストの流れを示している. 左側と右側の高さが同じ箇所は, 開発の詳細さの段階が同じであることを表している. 図 1.1 は次のような記載となっている. 左側に, 各工程名と各工程を担当するチームの名称を示し, 中央には, 左側の各工程で作成するものを記載し, 右側に左側に対応するテストの工程名を示した.

以下では, 図 1.1 の各工程ごとに, 本論文において用いる用語を定義する. 本論文では, 要求分析を行うチームをドメインエンジニアと呼び, 要求分析で決定する事柄を要求あるいは要求仕様, 作成する文書を要求仕様書と呼ぶことにする. 仕様策定を行うチームを仕様策定者, 仕様策定で決定する内容を仕様, 作成する文書を仕様書と呼ぶことにする. 設計検討を行うチームを設計者, この工程で決定する事柄を設計, 作成する文書を設計書と呼び, 実装を行うチームを実装者, この工程で作成するソースコードを実装コードと呼ぶことにする. テストを行うチームを評価者またはテスト実施者と呼ぶこととする. テストの工程では, 図 1.1 に示すように, 対応する工程に基づくテストを実施する. 図 1.1 の右側の上から, 要求仕様書を基に実装コードが要求仕様を満たしていることを確認するテストを, 要求仕様に基づくテストと呼び, 仕様書を基に実装コードが仕様を満たしていることを確認するテストを, 仕様に基づくテスト, 設計書を基に実装コードが設計を満たしていることを確認するテストを, 設計に基づくテスト, 実装者が行うテストを実装に基づくテストと呼ぶこととする. 要求仕様と仕様と設計に基づくテストは, テストの基になる文書が存在するが, 実装に基づくテストは, 実装者の意図を基準として実装コードのテストを行う. 図 1.1 には示していないが, 運用・保守を行うチームを運用者あるいは運

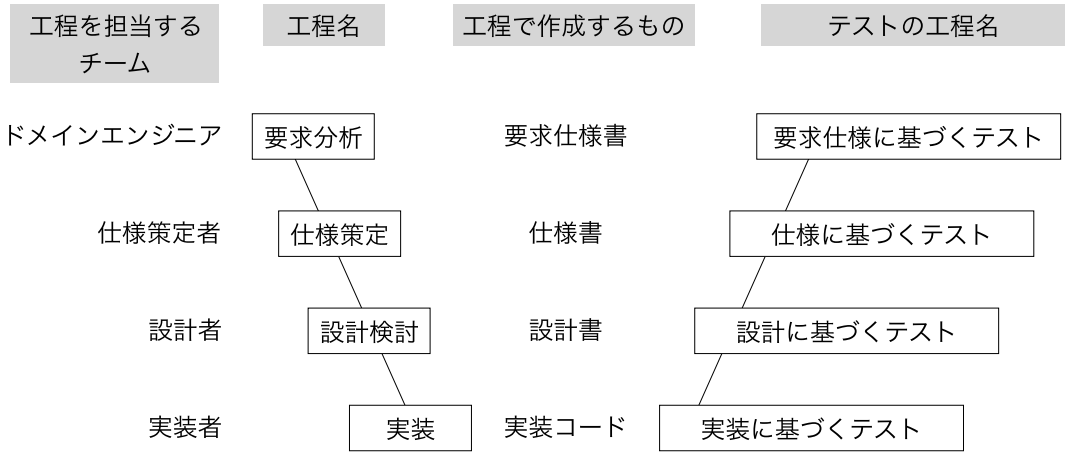

図 1.1　本論文において前提とする「 V 字型モデル」に準じた開発プロセス

用・保守者と呼ぶことにする.

1.1.2　ソフトウェア開発の現状

多くの開発現場では, 工程間で文書を受け渡すことで開発を進めている. 文書の厳密性や一貫性, 簡潔性, 理解容易性などの品質が, 開発効率および最終製品の品質に大きく影響を与えることが知られている. 特に, 開発工程の前段階である要求分析および仕様策定工程において品質の高い仕様書を作成することで, その後の設計や実装, テスト, 運用・保守を正確かつ組織的に行うことができるため, 開発効率を高めることができる.

しかし, 開発現場では上流工程の文書の品質を中心に多くの問題を抱えている. 以下に, 上流工程の文書の品質に起因する, 起こりうる具体的な問題について述べる.

テストの工程においては, 次のような問題が考えられる. 曖昧な仕様に起因する手戻りが発生した場合, 信頼性を高めるためのテスト工数およびテスト期間が見積もりを大幅に超える. そのため, スケジュールが遅延する. また, テストの入力となる仕様が曖昧である場合, テストの観点がもれてしまうので, 十分なテストが行えない. そのため, 市場において品質の問題を起こすことが考えられる.

設計と実装の工程においては, 次のような問題が考えられる. 本来は, 仕様策定工程で決めるべきことが決定されておらず, また, 仕様書における記述が曖昧であるために, 仕

様策定工程から設計工程に移っているにもかかわらず, 仕様を決定するための議論が多くなって開発効率が悪い. また, 明確に定義した仕様がないので, 詳細な仕様について仕様策定者と設計者が口頭で話し合った結果, 十分な意思疎通が行えないまま, 仕様とは異なる振る舞いをする製品が出荷されてしまう. そのため, 市場において品質の問題を起こすことが考えられる.

運用・保守の工程においては, 次のような問題が考えられる. 曖昧な仕様を基に実装を行い, 実装中に明らかになった仕様を仕様書に反映していないため, 正確な仕様書がない. そのため, 運用・保守工程において, 顧客からの質問に対して実装者しか回答を行えない状況となってしまう. この実装者が異動した場合, ソースコードの保守および顧客のサポートが難しくなることが考えられる.

これらの問題を解決するために, 上流工程において, 仕様を厳密に記述し検証するための数多くの開発手法が提案されている. フォーマルメソッド (formal methods) は, 計算機科学における数学を基盤としたソフトウェアおよびハードウェアシステムの仕様記述, 開発, 検証を行う技術の総称である. 早期に仕様の検証を行うことで後工程における手戻りを減らし, 厳密な仕様を基に設計およびテスト, 運用・保守を行うことで, 上流工程から下流工程において体系的な品質確保の枠組みを構築し, 品質の高いシステムを組織的に効率よく開発することができる.

1.2　フォーマルメソッドの適用における課題

本節では, フォーマルメソッドの概要について述べ, 産業界への適用における課題を分析する.

1.2.1　フォーマルメソッドの概要

近年, 品質の高いシステムを効率よく開発する手法として, フォーマルメソッドに対する関心が高まっている [CWA+96, BH06, 中島 07, 荒木 08].

例えば, 情報システムや製品のセキュリティ機能について第三者が評価・認証を行う

表 1.1 ISO/IEC 15408 [ISO09c] において定められた評価保証レベル

評価保証レベル	説明
EAL1	機能的なテストの保証
EAL2	構造的なテストの保証
EAL3	系統的なテストおよび確認の保証
EAL4	系統的な設計, テスト, レビューの保証
EAL5	準形式的な設計およびテストの保証
EAL6	準形式的な設計の検証およびテストの保証
EAL7	形式的な設計の検証およびテストの保証

ために国際的な評価基準を定めた ISO/IEC 15408[ISO09a, ISO09b, ISO09c] において, 7 段階の評価保証レベル EAL (Evaluation Assurance Level) が定められており, 最高レベルであるレベル 7 を取得するためには設計が形式的に検証されていなければならない. **表 1.1** にそれぞれの評価保証レベルを示す. レベル 1 からレベル 4 では, 形式的な設計は求められていない. レベル 5 においては準形式的な設計, レベル 6 においては準形式的な設計の検証が求められる. ISO/IEC 15408 は情報システムやセキュリティ製品の政府調達基準として利用されている. また, 電気・電子関連の国際安全規格として 2000 年に制定された IEC 61508[IEC10] においてもフォーマルメソッドの適用を推奨している.

フォーマルメソッドという用語は技術の総称として用いられ指し示す技術領域は広い. 一般的にフォーマルメソッドは, 計算機科学における数学を基盤としたソフトウェアおよびハードウェアシステムの仕様記述, 開発, 検証を行う技術の総称として用いられている. つまり, 数学を基盤とした記述言語により開発対象の特性を仕様として記述し, 定理証明や機械的な検査により記述の正しさを検証する手法である. あるいは, 開発対象の特性である仕様の記述から段階的に証明を行いながら詳細化を行いプログラムを導出する手法である.

フォーマルメソッドには多種多様な理論や手法, ツールが提供されており, その利用に関して近年では以下の 3 段階の適用レベルが示されている [荒木 08, Wik].

レベル 0 : 形式仕様記述

数学的な記法を用いて厳密な仕様を記述する. この仕様記述を基にしてプログラム

を開発する. 証明や分析までは行わない.

レベル 1 : 形式的開発および検証

プログラムの性質を証明したり, 詳細化により仕様からプログラムを作成する.

レベル 2 : 機械支援による証明

定理証明器や証明支援器を用いてプログラムの性質を証明する.

本研究において述べるフォーマルメソッドの適用レベルは, レベル 0 の形式仕様記述の段階である. レベル 0 の段階であってもフォーマルメソッドの適用による効果を得ることができる. フォーマルメソッドの適用により, 完全に正しいシステムを作ることはできないが, フォーマルメソッドを適用しなければ見つけることが難しい矛盾, 曖昧さ, 不完全さが明らかになり, またシステムに対する理解を深めることができる.

1.2.2　フォーマルメソッドの現状

フォーマルメソッドは 1970 年代後半から 1980 年代にかけて研究が始められ歴史のある技術である. また, 数多くのフォーマルメソッドの有効性を示す適用事例や適用における知見が発表されており [HK91, BH95, CWA+96, Hal96, Abr06, BH06], 産業界において十分に適用可能な技術である. しかしながら, 2010 年に David L. Parnas は, フォーマルメソッドが実践的手法として開発現場において広く利用されるような「当然の技術」として使われていない点を指摘し, フォーマルメソッドの問題点を示した [Par10]. Parnas は広く適用が進んでいない理由を次のように述べている.

> 産業界への実践的なフォーマルメソッドの適用は, フォーマルメソッドの利用が共通のプラクティスではないことを確認するような例外的な事例のままである. … 逆説的に, フォーマルメソッドの成功事例に関する報告は, 標準的な手順として, フォーマルメソッドの産業界への適用に失敗していることを明らかにしている. もし, これらの手法の利用がありふれた手順であれば, 手法の利用に関する論文は発表されることはないからである.

つまり, フォーマルメソッドの産業界への適用に関する論文が, 現在もなお取り上げられていることが, ありふれた「当然の技術」として, フォーマルメソッドが産業界に浸透していない現状を表していると述べている. Parnas は 研究者の立場から, フォーマルメソッドの現状を指摘し, 「当然の技術」としてフォーマルメソッドが用いられるために必要な研究の指針を与えた.

1.2.3 フォーマルメソッドの設計・構成法の課題

前述のとおり, 本研究において述べるフォーマルメソッドの適用レベルは, レベル 0 の形式仕様記述の段階である. このレベル 0 の段階においても, 産業界ではありふれた「当然の技術」として用いられていない. 自然言語を用いた仕様記述には, 正確性や簡潔性などの側面において様々な問題があることが 20 年以上前から知られている. Bertrand Meyer は 1985 年に仕様記述に自然言語を用いることによる欠陥と, 自然言語の代わりに数学的なフォーマルな表現を用いた場合の利点を明らかにした [Mey85]. 開発現場では多くの問題を抱えているにもかかわらず, フォーマルメソッドが産業界に浸透しないのはなぜなのであろう.

以下では, フォーマルメソッドの適用者の立場から, フォーマルメソッドが「当然の技術」として産業界に浸透するための課題を分析する.

課題を分析するために, レベル 0 の適用レベルを レベル 0-a とレベル 0-b の 2 つの段階に分けて考える. レベル 0-a は, 形式仕様の記述者が, 厳密に仕様を記述するという適用のレベルである. これにより, 形式仕様の記述者の開発対象に対する理解が深まり, 記述の不十分さや, 曖昧さに起因する不具合を早期に見つけることができる. レベル 0-b は, 形式仕様の記述者以外である, ドメインエンジニアや設計者, 実装者, 評価者が, 形式仕様記述を参照し, 形式仕様記述に基づいて設計と実装とテストを行う適用のレベルである. これにより, それぞれの開発工程において品質を確保し, 開発効率を高めることができる.

レベル 0-a からレベル 0-b になることで, 形式仕様記述を活用する工程が, 仕様策定工

程から開発工程全体に広がる．そのため, 形式仕様を記述する上で必要な技術も異なると考える．レベル 0-a において, 開発プロジェクトの中で形式仕様記述を活用する人は, 形式仕様の記述者のみの限られた人である．その人たちが形式仕様記述手法を習得し, その人たちの中で理解される形式仕様記述を作成すればよい．一方で, レベル 0-b において, 形式仕様記述を活用する人は, 開発プロジェクトのほぼ全ての工程の人たちとなる．ドメインエンジニア, 設計者, 実装者, 評価者が, 形式仕様記述を読み, 理解し, 活用していく必要がある．これらの人たちが, 既に形式仕様記述手法を習得していることは希であり, 数日の研修によって形式仕様記述手法を学ぶことになる．形式仕様記述手法の専門家ではない人に, 形式仕様記述をありふれた「当然の技術」として活用してもらう必要がある．そのために, 形式仕様記述を読みやすいと感じ, 自然言語よりも形式仕様記述言語を用いていることにメリットを感じることが大切である．つまり, レベル 0-b の適用段階において, そのような形式仕様を記述するための技術と, 活用するための技術が必要となる．この課題を解決することで, 形式仕様記述手法が, ありふれた「当然の技術」として開発プロジェクトにおいて広く活用されるようになり, さらには, 産業界においても広く活用されることにつながると考える．

本研究では, 前述のレベル 0-b の段階に到達するために必要な, 形式仕様を記述するための技術と活用するための技術を研究課題として設定した．記述するための技術として, 様々な役割を担った開発者にとって, 理解容易性に優れた仕様を記述する技術が必要である．ここでいう様々な役割を担った開発者とは, ドメインエンジニア, 設計者, 実装者, 評価者である．活用するための技術としては, 従来からある既に開発現場に浸透しているレビューやテスト手法といった検証技術を主体として, その中で形式仕様記述手法を活用していく技術が必要である．

また, 本研究では, 前述の形式仕様を記述するための技術として, 形式仕様記述の設計・構成法に着目した．ソフトウェア開発において, 構成法 (construction) とは設計・実装・デバッグ・開発者テストなどを中心とする活動である．これらの活動を省略してソフトウェアを完成させることは難しい．形式仕様記述手法の適用においても同様に, 設計・構成法に関する技術が必要不可欠である．

筆者は, フォーマルメソッドの導入を検討している段階において, フォーマルメソッドが我々の課題を解決できることは分かった. しかし, 手法を試みる段階において適用技術の設計・構成法に関する知識が不足していたため, 「何から着手すべきであるか」ということが分からなかった. そのため, 初めは設計・構成法の技術を身につけるためにフォーマルメソッドの有識者から学びながら「分からなくても, とにかく手を動かしてみる」という方法を取った. 有識者から学びながら手法の適用を進めることは, フォーマルメソッドに限らず新しい技術の習得において必要なことが多い. 新しい技術の習得・適用には, 良き指導者を見つけるとよい [BH95, HO09]. 良き指導者を見つけることで, 筆者は実際に適用するときの設計・構成法に関する知識不足の課題を解決した. 設計・構成法は, 具体的な詳細設計や実装の技術であるため, 抽象化および一般化することが難しいものが多い. そのため, 新しい技術を適用するために, 開発現場ごとに良き指導者を見つけることで, 設計・構成法の課題を解決している.

本論文の目的は, FeliCa IC チップ開発への適用経験を通して得られた課題と具体的な設計・構成法を提示し, フォーマルメソッドが産業界の問題を解決する「当然の技術」たる所以を示すことである. 本研究の基となる FeliCa IC チップ開発の概要について, 1.5 節で述べる. フォーマルメソッドとして, 形式仕様記述手法とモデル検査 [CGP99] を用いた. FeliCa IC チップ開発の仕様策定工程において形式仕様記述手法を用い, 仕様策定後の設計工程においてモデル検査を用いた. モデル検査を用いることで, 形式仕様記述手法では取扱いが容易でない動的な振る舞いを検証した. 本節に続く 1.3 節では形式仕様記述手法について述べ, 1.4 節ではモデル検査について述べる.

1.3 形式仕様記述手法 VDM

本節では, 筆者らが FeliCa IC チップ開発において用いた, 形式仕様記述手法の概要を述べる. 仕様策定工程において, モデル指向型の形式仕様記述手法 VDM (Vienna Development Method) [BJ78, Jon90] を用いた. まず, 形式仕様記述手法 VDM について概観した後に, VDM の統合開発環境である VDMTools について述べる. 最後に, 実行可能

仕様について議論する.

1.3.1　形式仕様記述手法 VDM の概要

形式仕様記述手法とは, 仕様を数学あるいは論理学で使われるような形式化 (formalization) による抽象化を行う形式的な仕様化の技法である. 形式化の方法により, モデル指向型と性質指向型に分類される. 性質指向型とは仕様化の対象となるシステムの外から見た性質を公理的な方法で定義するものである. 性質指向型の手法としては Larch[GHJ+93] や OBJ[Gog79] などの抽象データ型の代数仕様による意味定義の方法がよく知られている. 一方, モデル指向型は, 状態機械の仕様をその状態を構成するデータ構造と, それに対する操作の入出力仕様で定める方法である. モデル指向型の手法としては VDM (Vienna Development Method)[BJ78, Jon90], Z 記法 [Abr96], B メソッド [Abr96] が有名である. VDM には VDM-SL[ISO96] と, VDM-SL の言語仕様をオブジェクト指向に拡張した VDM++[FLM+05] がある. VDM-SL は, 1970 年代に IBM ウィーン研究所でプログラミング言語 PL/I コンパイラの正しさを検証するプロジェクトで用いられた技術を, 一般の仕様記述向けに拡張したものである. 1975 年から 1990 年代前半にかけて開発され, 1996 年に ISO 標準となった [ISO96]. VDM-SL にオブジェクト指向と同時並行処理記述を付加した VDM++ は, 1990 年代前半にヨーロッパにおける ESPRIT 計画 の Afrodite プロジェクトの中で, 産業界への適用を狙って開発された.

VDM-SL と VDM++ は, Floyd-Hoare 論理 [Flo67, Hoa69] における事前条件 (precondition) と事後条件 (postcondition) および不変条件 (invariant) によって仕様を記述する枠組みを提供する.

1.3.2　統合開発環境 VDMTools

VDM-SL と VDM++ には, 統合開発環境 VDMTools[SCS, 佐原+07] が SCSK 株式会社から提供されている. VDMTools は以下の機能を提供する.

- 仕様の構文検査 (syntax check)・型検査 (type check)

- 実行可能仕様のアニメーションを行うための評価実行とデバッグ支援
- 実行可能仕様のコードカバレッジ計測
- ソースコードの清書：コードカバレッジ情報を含んだ自然言語によるドキュメントと VDM-SL, VDM++ のソースコードを混合させた文書を生成する
- 実行可能仕様から Java, C++ コードの生成
- 証明課題生成
- UML リンク：クラス図とソースコード, ソースコードとクラス図の双方向変換
- CORBA API : 実行可能仕様を CORBA サーバとして, C や Java のクライアントから呼び出して実行する

仕様の構文検査や型の検査により, 機械的な検証を行いながら仕様を記述することができる. 実行可能仕様と仕様アニメーションについて, 次の 1.3.3 節において説明する.

1.3.3 VDM++ による実行可能仕様

形式仕様記述言語は意味定義が厳密に決まっているため, 言語処理系を構築し, 仕様アニメーションにより検証を行うことができる. 仕様アニメーションとは, 記述した仕様を言語処理系の上で実行することである. プログラムの実行と区別して, 仕様アニメーションと呼ぶことが多い. 本論文では, 仕様アニメーションよる実行可能な仕様を実行可能仕様と呼び, 形式仕様記述言語を用いて記述した仕様記述を形式仕様記述と呼ぶこととする. VDMTools における, 実行可能仕様の記述例と実行可能ではない仕様の記述例を示す. 次の (1) の記述は実行可能であり, (2) の記述は実行可能ではない.

```
(1) {x | x in set {5,6,7} & x > 5 }
(2) {x | x : nat          & x > 5 }
```

(1) は {5, 6, 7} に含まれる x のうち 5 より大きい数値の集合を表す式であり, 実行することができる. VDMTools の評価実行環境を用いて次のように実行する.

```
> print {x | x in set {5,6,7} & x > 5 }
{ 6,7 }
```

行頭の「>」は VDMTools がコマンド入力待ちであることを表す．「 print 」は式を評価して結果を出力するという VDMTools の命令である．評価した結果, {6, 7} を得る．一方, (2) の記述は 5 より大きい自然数 (nat) の集合を表す式であり, 無限の値の集合は, VDMTools の評価実行環境の制約により実行することができない．

一般に記述した仕様を実行するためには次の 2 つの方法がある．

- 仕様記述を実行可能コードに変換して動作させる
- 仕様記述を解釈して動作させるインタープリタを構築する

仕様アニメーションによる検証の際に, いずれの方法を選択するかは, 形式仕様記述言語ごとに提供されている統合開発環境が, 実現するための機能を備えているか否かによる．VDMTools は C++ コード生成機能とインタープリタ機能の両方を備えており, 前述の 2 つの方法を選択することができる．

インタープリタ機能を使用した場合, VDM++ コードを逐次解釈しながら実行する．C++ コード生成機能を使用した場合, VDM++ コードを最終的にマシン語に変換し, バイナリコードを実行する．あらかじめマシン語に変換する C++ コード生成機能を用いた方が, インタープリタ機能を使用した場合よりも, 実行速度が速い．そのため, 仕様アニメーションによる検証において, 実行速度が課題となる場合, C++ コード生成機能を用いることを検討する必要がある．また, C++ コード生成機能により生成したコードを製品の実装コードとして使用できる場合, C++ コード生成機能を使用する利点がある．

しかし, C++ コード生成機能を使用する場合, インタープリタ機能を用いる場合よりも, 仕様記述に用いることができる VDM++ の言語仕様 [SCS10] の制約が厳しくなる．例えば, 関数の引数に関数型を指定する高階関数を VDM++ は定義することができるが, C++ コード生成機能はこの高階関数には対応していない．

我々は, VDM++ コードの評価実行環境として VDMTools のインタープリタ機能を使用した．理由は, 以下の 3 点である．(1) 仕様記述に高階関数を使用しており, C++

コード生成機能の言語仕様の制約を満たしていなかった. (2) インタープリタ機能を用いても実行速度は大きな課題とはならなかった. (3) FeliCa IC チップのファームウェア実装コードは, セキュリティ実装を行う必要があり, C++ コード生成機能を利用することができたとしても, 出力した C++ コードをそのままファームウェア実装コードとして用いることはできなかった.

1.4 モデル検査

本節では, 筆者らが FeliCa IC チップ開発において用いた, フォーマルメソッドの 1 つであるモデル検査の概要を述べる. 仕様策定後の設計工程において, 形式仕様記述手法では取扱いが容易でない動的な振る舞いについて, モデル検査を用いて検証を行った.

モデル検査とは, システムの仕様を表現したモデルがシステムに要求される性質を満たすかどうかを自動的に検証する手法である. 特に並行性を含むシステムを対象に, モデルをラベル付き遷移システム (labelled transition system) で記述し, 性質を時相論理 (temporal logic) で表すものをモデル検査と呼ぶことが多い [CGP99]. 以下ではラベル付き遷移システムと時相論理について述べる.

ラベル付き遷移システムとは, 記述対象の振る舞いを状態および状態間の遷移で構成されるグラフで表す. 一般にこのグラフは状態遷移図と呼ばれ, 遷移に動作 (action) のラベルを付けたものである. 時相論理とは, 動作の進行順序に関する性質を表現する論理体系である. 時相論理式で表現する主な性質として, 安全性 (safety) と活性 (liveness) がある. 安全性は, 望ましくない事象がどのような遷移経路をたどっても決して起こらないという性質である. 安全性の例としては, 次の遷移が存在しないデッドロック (deadlock) や, リソースの排他アクセス違反などがある.

安全性を時相論理式で表すと次のようになる.

$$AG\neg P$$

ここで, A は「どのような遷移経路をたどっても」を意味する限量子であり, G は「いつでも」を意味する時相演算子, P は望ましくない性質を表す述語である.

活性は, 望ましいことがいつか必ず起こるという性質である. 望ましいことは, 記述対象によって異なり, 通常は仕様に記述される. 例えば, 本論文で述べる FeliCa IC チップにおいては, いつかは, 処理を受け付けることができる受信待ちに戻ることといった性質である.

活性を時相論理式で表すと次のようになる.

$$AFQ$$

ここで, F は「いつかは」を表す時相演算子であり, Q は望ましい性質を表す述語である.

モデル検査の技術は, ラベル付き遷移システムで表現したモデルを, 網羅的に探索し, 時相論理式で記述した性質を満たしていることを自動的に調べる技術である. これにより, 設計レビューやテストにより見つけることが難しい不具合を見つけることができる. しかし, 網羅的に全状態空間の探索を行い検証が終了するためには, ラベル付き状態システムとして記述する検査対象の状態数が有限である必要がある. また, コンピュータの性能向上や, アルゴリズムの工夫とともに, 扱える状態数が実用的なものになってきてはいるが, 状態数が, 検証ツールを動かしているコンピュータのメモリに収まり, 検証ツールが現実的な時間で計算を終えるように, 検査対象を抽象化し, 適切な規模の状態数としなければならない.

本論文では, FeliCa IC チップ開発の設計工程において, ハードウェア設計とソフトウェア設計をラベル付き遷移システムで表現することで, 設計上の矛盾が起きないことをモデル検査を用いて検証した適用事例 [中津川+06] について述べる. 特に, モデル検査の段階的な導入の方法と, 検査対象の状態数の削減方法について述べる.

1.5　本研究の基となるフォーマルメソッドの適用事例

本節では, 本研究の基となる筆者らの FeliCa IC チップ開発への形式仕様記述手法の適用事例を紹介する. まず, FeliCa IC チップの特徴について述べ, 形式仕様記述手法の適用に至る背景を説明する. 次に適用における効果を統計データから論じ, 最後に適用により得られた課題を述べる. 3 章以降では, これらの課題について議論を進め, 課題に対

する解決策を提案する.

形式仕様記述言語 VDM++ により実行可能仕様を記述し, 記述した仕様を仕様アニメーションにより動作検証を行った. 下流工程では仕様に基づくテストに形式仕様記述を活用した. 仕様記述の構文検査, 型検査, 仕様アニメーションには VDMTools を用いた.

1.5.1 FeliCa IC チップの特徴

FeliCa とはソニー株式会社が開発した非接触 IC カードの技術方式である [松尾 07]. FeliCa IC チップは非接触 IC カードや,「おサイフケータイ」として知られる携帯電話に組み込まれており, 電子マネーや公共交通機関の乗車券・定期券, クレジットカード, ドアの鍵, 身分証などのサービスに応用されている. 1 つの IC チップに複数のアプリケーションの搭載が可能なため, 複数の事業者が 1 つの IC チップを共用できる. 本研究では, これらの電子マネーや乗車券, クレジットカードなどの用途としての機能を, FeliCa IC チップのカード機能と呼ぶこととする. FeliCa IC チップのカード機能は, ファイルシステム機能, セキュリティ機能, コマンド機能の 3 つの機能からなる. ファイルシステム機能は, 複数のサービス事業者が IC チップとの相互認証に使用する各事業者の鍵データや, 電子マネーの残高などのユーザデータを保持・管理する機能である. セキュリティ機能は, リーダ・ライタと相互認証および暗号通信を行う機能である. コマンド機能は, リーダ・ライタからのコマンドを解釈する機能である. リーダ・ライタとは, 店舗のレジ端末にあるような読み書きを行う装置である.

特に「おサイフケータイ」として知られる携帯電話に搭載された FeliCa IC チップをモバイル FeliCa IC チップと呼ぶ [杉山+07]. モバイル FeliCa IC チップには, 前述のカード機能に加え, リーダ・ライタ機能が搭載されている. リーダ・ライタ機能により, 店舗のレジ端末にあるような読み書き装置として, モバイル FeliCa IC チップを使用することができる. また, モバイル FeliCa IC チップは, リーダ・ライタからモバイル FeliCa IC チップにアクセスすることができる無線通信インタフェース機能と, 携帯端末側からモバイル FeliCa IC チップにアクセスすることができる有線通信インタフェース機能を備

えている. 本論文では, モバイル FeliCa IC チップを含めて FeliCa IC チップと呼ぶ.

以下に, FeliCa IC チップ開発の特徴と形式仕様記述手法の適用に至る背景を示す.

- FeliCa IC チップは広く社会基盤の一部として利用され, 製品開発における品質確保が重要な課題である.
- FeliCa IC チップは同一の仕様を複数のハードウェアプラットフォームに実装しており, 製品間の相互接続性を確保するためには, 仕様の厳密性が求められる.
- FeliCa IC チップは様々な耐タンパ機能を搭載しており, セキュリティ製品として全ての入力に対して正常系, 準正常系の振る舞いを厳密に定義する必要がある.

これらの背景から, 仕様を簡潔かつ厳密に記述することができる形式仕様記述手法に着目した.

1.5.2　開発プロジェクトの概要

筆者らは, 次に示す 2 つの製品開発プロジェクトにおいて形式仕様記述手法を適用した. 1 つ目は, モバイル FeliCa IC チップ開発プロジェクトである. これは, フェリカネットワークス株式会社にて, 2004 年から 2007 年の期間で行った開発プロジェクトである. この開発プロジェクトに形式仕様記述手法を用いた [栗田+06, KCN08, KN09, 栗田+09]. 2 つ目は, カード開発プロジェクトである. これは, ソニー株式会社にて, 2007 年中頃から約 4 年間の期間で行った開発プロジェクトである. いずれの開発プロジェクトにおいても, 上流工程の仕様記述に形式仕様記述言語 VDM++ を用い, 記述した実行可能仕様を用いて仕様アニメーションにより動作検証を行い, 下流工程では仕様に基づくテストに形式仕様記述を活用した.

本研究では, 1 度目のモバイル FeliCa IC チップ開発における形式仕様記述手法の適用を第 1 世代の適用と呼び, 2 度目のカード開発における適用を第 2 世代の適用と呼ぶこととする. 第 1 世代の適用を通して, 仕様記述手法を用いることにより得られる効果に対して確信を持ち, また適用における課題が明確になった. 第 2 世代では, 第 1 世代において得られた課題を考察し, 課題を解決する形式仕様を記述することができた.

表 1.2 第 1 世代と第 2 世代の形式仕様記述の規模

モジュール名	有効行数 (行)	
	第 1 世代	第 2 世代
コマンド機能	26,370	11,460
ファイルシステム機能	5,097	1,716
セキュリティ機能	821	1,548
記述フレームワーク	3,011	425
汎用ライブラリ	1,305	648
合計	36,604	15,797

本章では, 第 1 世代の適用を通して得られた成果と課題について述べる. 以降の章において, 第 2 世代の適用を通して得られた, 第 1 世代の課題に対する解決方法を提案する [中津川+09, 中津川+10, NKA10] . また, 第 1 世代において, 仕様策定工程に形式仕様記述手法を用いたが, 形式仕様記述手法では取扱いが容易でない動的な振る舞いに関して, 設計工程においてモデル検査を適用した. この動的な振る舞いに関するモデル検査による設計検証についても, 本章以降の章において述べる [中津川+06].

1.5.3 形式仕様記述手法の適用対象と規模

形式仕様記述手法の適用対象は FeliCa IC チップの仕様書である. 第 1 世代と第 2 世代の形式仕様記述の規模を **表 1.2** に示す. 第 1 世代の対象製品は, リーダ・ライタ機能などの機能を搭載しているモバイル FeliCa IC チップであり, 第 2 世代の対象製品は, カード機能のみを搭載している FeliCa IC チップである. 形式仕様記述の有効行数は, 第 1 世代が約 37K であり第 2 世代は約 16K である. ここでいう有効行数とは, 総行数からコメント行と空行を除いた行数である. 第 2 世代に比べ第 1 世代の方が製品に搭載されている機能が多いため仕様の規模が大きくなっている. 共にコマンド機能が約 7 割を占め, 主要な仕様記述モジュールとなっている.

表 1.3　摘出工程別の仕様不具合件数

不具合発見工程	不具合件数 (件)
自然言語で仕様記述する過程で見つけた不具合	93
形式仕様記述言語で仕様を記述する過程で見つけた不具合	162
仕様アニメーションを実施して見つけた不具合	116
ドメインエンジニアと形式仕様記述者の コミュニケーションにより見つけた不具合	69
合計	440

1.5.4　適用における効果

第 1 世代の形式仕様記述手法の適用により様々な効果が明らかになった. ここでは, 本論文の背景となる 2 つの効果について述べる.

1 つは, 形式仕様記述手法を適用しなければ摘出が困難な不具合を早期に摘出し, 後工程における手戻りを減らし, 開発スケジュールの遅延, コストの超過を防ぐことができた点である. 実行可能仕様を動かして仕様アニメーションによる検証を行うことで, 上流工程において早期に仕様不具合を見つけることができた. 上流工程における不具合を後工程で修正した場合の修正コストは, 上流工程で修正した場合の数 10 倍から数 100 倍と言われている [Lef97, SBB+02]. そのため, 早期に不具合を摘出することは, 後工程における手戻りを減らし, 品質の高いソフトウェアを開発する上で重要な活動である.

表 1.3 に上流工程において摘出した仕様の不具合件数を示す. 形式仕様記述言語で仕様を記述する過程で見つけた不具合 162 件, 仕様アニメーションを実施して見つけた不具合 116 件は形式仕様記述手法を導入しなければ見つけることが難しい不具合であり, 後工程において手戻りとなる可能性があった. 形式仕様記述手法を導入することにより, 設計・テストが開始する前の上流工程において多くの不具合を見つけることができた. これらの経験から, 第 2 世代においても実行可能仕様を用いる方針とした.

2 つ目の適用における効果は, 仕様策定, 設計, 実装, テストと作業分担を行い複数人で

開発するプロジェクトにおいて, 厳密な仕様をコミュニケーションツールとして用いることができた点である.

仕様策定工程において, 自然言語と形式仕様記述言語をそれぞれ用いて, FeliCa IC チップの仕様書を作成した. これら 2 つの仕様書は, それぞれで個別に完結した仕様書となっている. 自然言語で記述した仕様を入力として, 形式仕様記述言語を用いてより詳細な仕様を記述した. 形式仕様記述言語を用いることで, 曖昧な仕様を見つけることができた. これにより, 自然言語の曖昧な記述を修正することができた.

コミュニケーションの 1 つの側面として, これらの自然言語の仕様書と形式仕様記述言語の仕様書への質問内容を分析した. 質問内容の分析は, 両者の質問を分類し, 傾向を比較することにより行った. **図 1.2** に自然言語の仕様書に対する質問の分類結果を, **図 1.3** に形式仕様記述言語の仕様書に対する質問の分類結果を示す. 質問の分類方法を以下に示す. まず質問を「理解」, 「意図」, 「誤り」の 3 つに分類した. 質問者が「理解したい」という考えを持った質問を「理解」とし, 理解はしているが「理解以外の意図」を持った質問を「意図」とし, 単純に誤りを指摘した質問を「誤り」と分類した. さらに「理解」を「記載済み」, 「記載なし」, 「背景」に分類し, 「意図」を「明確化依頼」, 「精査依頼」, 「設計確認依頼」, 「実装要望」に分類した. 「理解」の「記載済み」と「記載なし」は質問に関する回答が仕様書に記載済みか, 記載なしかにより分類し, 「なぜこのような仕様になっているか」という質問を「背景」と分類した. 「意図」に分類した質問について, 仕様は明確ではあるが記載内容を明確にして欲しいという「明確化依頼」, 仕様の間違いではないが統一性がないなどの仕様を精査して欲しいという「精査依頼」, 設計者からの設計が仕様を満たしていることを確認して欲しいという「設計確認依頼」, 設計・実装情報を仕様にも反映して欲しいという「実装要望」に分類した.

図 1.2, 図 1.3 を比較すると, コミュニケーションの傾向が分かる. 自然言語の仕様書に対しては, 「明確化依頼」が占める割合が最も多く, 形式仕様記述言語を用いた仕様書に対しては, 「記載済み」の内容に関する「理解」に分類した質問の割合が多かった. 自然言語と形式仕様記述言語で書かれた仕様は同じ「理解」に分類される質問が多いが, コミュニケーションの質が異なる. 占める割合が最も多い自然言語の仕様書に対する「明

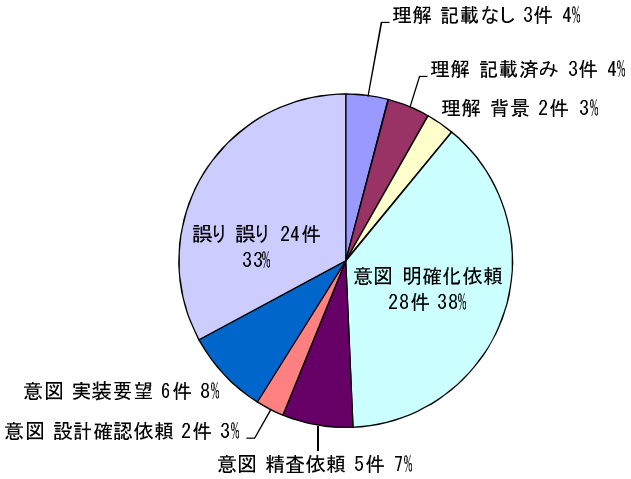

図 1.2　自然言語の仕様書に対する質問の分析

確化依頼」の質問は,「書かれていない」,「明確になっていない」ことを「記載してもらう」,「明確にしてもらう」ための質問である. 一方, 形式仕様記述言語の仕様書に対しては,「記載済み」の内容, 背景に関する質問が占める割合が多いことから, 読み手が仕様をより深く理解するための質問であることが分かる. また, 形式仕様記述の場合, コミュニケーションにより理解したことが既に「記載済み」の内容として厳密に定義されているということは, 開発者に安心感を与える効果があった.

1.5.5　適用における課題

形式仕様記述言語 VDM++ により実行可能仕様を記述し, 仕様アニメーションにより動作検証を行った. 下流工程では仕様に基づくテストに形式仕様記述を活用した. これらの形式仕様記述言語を用いた開発において次の 3 つの課題が明らかになった. 本節ではこれらの課題について考察する.

(1)　「伝えること」と「動かすこと」を目的とした実行可能仕様の課題

(2)　仕様と設計の分離に関する課題

(3)　レビューとテストの用途として形式仕様記述を利用する場合の課題

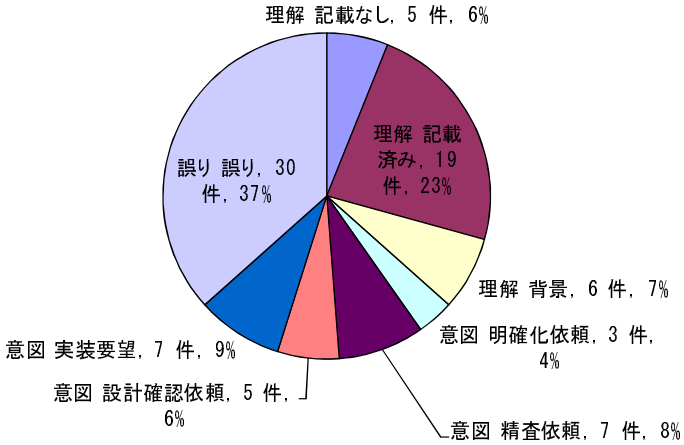

図 1.3 形式仕様記述言語の仕様書に対する質問の分析

(1) と (2) は, 形式仕様記述の読みやすさに関する課題である. 図 1.3 の「記載済み」の内容に対する質問の占める割合が多い点から, 読みやすい仕様を書くことが課題であることが分かる.

まず, (1) の課題について述べる. 上流工程において記述した形式仕様を, 仕様アニメーションにより動作検証を行った. そのため, 仕様として「伝えること」を目的とした記述と「動かすこと」を目的とした記述が混在したため, 仕様の読みやすさが課題となったと考えた. したがって, 1 つ目の課題として, 「伝えること」と「動かすこと」を目的とした実行可能仕様について考察することとした. 詳細は, 2.1.1 節で述べる.

次に (2) の課題について述べる. 仕様と設計の境界が曖昧であったため, 仕様の読みやすさが課題となったと考えた. これは, 第 1 世代の開発中において, 仕様策定者と設計者の間で議論になったことである. 議論の内容は, 形式仕様記述から仕様として規定している範囲と, 形式仕様記述には記述してはあるが, 設計者が自由に規定できる範囲を, 設計者が読み取ることが難しいということであった. したがって, 2 つ目の課題として仕様と設計の分離に関する課題について考察することとした. 詳細は, 2.1.2 節で述べる.

最後に (3) の課題について述べる. 第 1 世代では, 仕様に基づくテストにおいて, 製品に対して実行したテストスクリプトを VDM++ のソースコードに変換することにより, 形式仕様記述に対しても実行することができるようにした. これにより, 製品に対して実

行したテストスクリプトの形式仕様記述に対する網羅性を確認した．テスト設計, テスト項目作成においては, 形式仕様記述を参照するにとどまった．第 2 世代では, 第 1 世代よりも広く形式仕様記述を使うことを目標とした．そこで, 仕様に基づくテストとレビューに着目し, 形式仕様記述の利用方法をあらかじめ考慮して, 形式仕様記述フレームワークを設計していく方針とした．したがって, レビューとテストの用途として形式仕様記述を利用する場合の課題について考察した．詳細は, 2.2 節で述べる．

このように, 第 1 世代の適用において, 前記の 3 つの課題が明らかになり, 第 2 世代において解決すべき目標を設定することができた．

1.6　本論文の構成

本論文の構成は, 以下のとおりである．まず 2 章では, フォーマルメソッドを適用する上で課題となった, 本研究において扱う次の 3 つの課題を分析する．1 つ目の課題は, 1.5.5 節において述べた (1) と (2) の課題に関連する実行可能仕様における仕様と設計の分離に関する課題である．2 つ目の課題は, 1.5.5 節において述べた (3) の課題に関連する, レビューとテストの用途として, 形式仕様記述を利用する場合の課題である．3 つ目の課題は, 形式仕様記述手法では取扱いが容易でない動的な振る舞いに関する課題である．次に, 3 章では, 関連研究として, 産業界へのフォーマルメソッドの適用事例について概観し, 筆者らのフォーマルメソッドの適用事例について考察する．また, 1 つ目の課題である, 実行可能仕様における仕様と設計の分離に関する関連研究から, 本研究で扱う課題をさらに考察する．続いて, 4 章から 6 章では, 2 章において述べた 3 つの課題について, 本研究において得られた成果を述べる．4 章では, 1 つ目の課題である実行可能仕様における仕様と設計の分離に関する課題を解決する, 実行可能仕様の設計・構成法を提案する．5 章では, 2 つ目の課題であるレビューとテストの用途に, 形式仕様記述手法を活用する場合の課題を解決する, 仕様記述フレームワークの設計・構成法を提案する．また, 仕様に基づいたテストへの, 形式仕様記述の活用方法を提案することで, 本論文で提案する仕様記述フレームワークがテストの用途を考慮していることを示す．6 章にお

いては, 形式仕様記述手法では取扱いが容易でない動的な振る舞いに関して, モデル検査を用いた設計検証の適用事例について述べる. 7 章では, それまでの議論をまとめ, 今後の展望を示す.

第2章

本研究において扱うフォーマルメソッドの適用における課題

1章では, 組織的に開発を進める必要があるソフトウェア開発現場において, 上流工程の文書を中心に多くの問題を抱えており, 品質の高いシステムを組織的に効率よく開発することができる手法として, フォーマルメソッドが注目されていることを述べた. フォーマルメソッドの3段階の適用レベルにおいて, 本研究が扱うのは, 形式仕様記述を行うレベル0である. レベル0は, 証明までは行わないが, 数学的な記法を用いて厳密な仕様を記述し, プログラムを開発していく段階である. このレベル0の段階において, 筆者らはモバイル FeliCa IC チップ開発を通して適用の効果を得ることができた. しかし, 産業界は多くの問題を抱えているにもかかわらず, フォーマルメソッドが, ありふれた「当然の技術」として産業界に浸透していないという Parnas の指摘を紹介し, 「当然の技術」として浸透するための課題を分析した. 分析において, 適用レベルをレベル0からレベル0-aとレベル0-bの2つに分けた. レベル0-aは, 形式仕様の記述者が, 形式仕様記述手法を用いて仕様を厳密に記述し, 早期に仕様の曖昧さに起因する不具合を見つけることができるという段階とした. レベル0-bは, 形式仕様の記述者以外である, ドメインエンジニアや設計者, 実装者, 評価者が, 形式仕様記述を参照し, 形式仕様記述に基づいて設計と実装とテストを行う適用の段階とした. 開発プロジェクトの中で形式仕様記述を活

用する人は, レベル 0-a においては, 形式仕様の記述者のみの限られた人であるが, レベル 0-b においては, ドメインエンジニア, 設計者, 実装者, 評価者といったほぼ全ての工程の人たちとなる. これらの人たちが形式仕様記述手法をあらかじめ習得していることは希であり, 形式仕様記述手法の専門家ではない人が, 形式仕様記述を読み, 理解し, 活用していく必要がある. つまり, 形式仕様記述手法の専門家ではない人が, 読みやすいと感じ, 形式仕様記述手法を利用することにメリットを感じるような, 形式仕様を記述するための技術と, 活用するための技術が必要である. この課題を解決することで, 形式仕様記述手法がありふれた「当然の技術」として, 開発プロジェクトにおいて広く活用されるようになり, さらには, 産業界においても広く活用されることにつながると考える.

本研究では, 前述のレベル 0-b の段階に到達するために必要な, 形式仕様を記述するための技術と, 活用するための技術を研究課題として設定した. 記述するための技術として, ドメインエンジニア, 設計者, 実装者, 評価者といった様々な役割を担った開発者にとって, 理解容易性に優れた仕様を記述する技術が必要である. 活用するための技術としては, 従来からある既に開発現場に浸透しているレビューやテスト手法といった検証技術を主体として, その中で形式仕様記述手法を活用していく技術が必要である. 本章では, これらの課題を具体化する. 本研究において扱う課題は次の 3 つである.

(1)　実行可能仕様における仕様と設計の分離に関する課題

(2)　レビューとテストの用途として, 形式仕様記述を利用する場合の課題

(3)　形式仕様記述手法では取扱いが容易でない動的な振る舞いに関する課題

(1), (2) は, レベル 0-b の適用段階において, 形式仕様を記述するための課題と, 活用するための課題である. (3) は, レベル 0-b の議論とは別の, 仕様策定後の設計工程における課題である. 以降の節において, それぞれについて議論する.

2.1 実行可能仕様における仕様と設計の分離に関する課題

2.1.1 「伝えること」と「動かすこと」を目的とした実行可能仕様の課題

レベル 0-b の段階において, 形式仕様記述の読者は, 形式仕様記述手法の専門家ではない, ドメインエンジニア, 設計者, 実装者, 評価者といったほぼ全ての工程の人たちとなる. 開発工程の上流工程である仕様策定工程において, 品質の高い仕様書を作成することで, その後の設計, 実装, テスト, 運用, 保守を正確かつ組織的に行うことができるため, 開発効率を高めることができる.

仕様書には何を作るべきかという「解決すべき問題」を定義することで, 仕様策定工程以降の設計, 実装, テスト, 運用・保守工程では, 仕様書を参照し, 各工程において実施すべきことを決める. 設計工程では仕様書を参照して「問題の解決方法」を検討し, 設計書を作成する. つまり, 仕様書の役割は, 何を作るべきかという「解決すべき問題」を各工程に明確に「伝えること」である.

形式仕様記述言語を用いて仕様を厳密かつ簡潔に記述することにより, 仕様書の理解容易性を高めることが期待できる. また, 形式仕様記述言語を用いて記述した場合, 記述の意味が厳密に定義されているため, 言語処理系を構築し, 仕様アニメーションにより仕様を「動かすこと」で検証を行うことができる. 実行可能仕様を記述することにより, 上流工程において仕様を検証し, 早期に不具合を摘出し, 後工程における手戻りを減らすことができる. また, 仕様アニメーションによる検証は, 開発現場に既に浸透している検証手法であるテストの技術を流用することができるため, 証明などと比べると導入が容易である.

しかし, 実行可能仕様には, 注意すべき問題があることが知られている. 仕様書の役割は, 何を作るべきかという「解決すべき問題」を各工程に明確に「伝えること」である. 実行可能仕様により, この「伝えること」を目的とした記述と, 仕様アニメーションにより「動かすこと」を目的にした記述が混在することになる. 2 つの記述が混在することにより, 本来は「解決すべき問題」を規定する仕様に, 「問題の解決方法」としての記述が多く含まれてしまう. これにより, 設計者を過剰に制約し, 実装の選択肢を狭めてしま

う可能性がある．また,「問題の解決方法」の記述が多く含まれることにより, 仕様の記述が複雑になってしまう傾向がある．

理解容易性に優れた仕様を記述するためには,「伝えること」を目的とした記述を明確にする必要がある．そのため, これを妨げる「動かすこと」を目的とした記述が課題となる．一方, 従来からある既に開発現場に浸透しているテスト手法を用いて, 仕様アニメーションにより実行可能仕様を活用するためには,「動かすこと」を目的とした記述が必要である．これらの課題を解決するために, 本研究では「伝えること」を目的とした記述と,「動かすこと」を目的とした記述を明確に分離する方法を提案する．3.2 節において, 関連する Ian J. Hayes と Cliff B. Jones の「実行可能性の影響」に関する研究 [HJ89] から課題をさらに考察する．4.1 節において提案方法の詳細を述べる．

2.1.2　仕様と設計の分離に関する課題

一般に, 仕様策定工程では, 何を作るべきかという「解決すべき問題」を検討し, 設計工程では,「問題の解決方法」を検討することで, 仕様策定者と設計者が分業して組織的に開発を行うことができると言われている．この「解決すべき問題」と「問題の解決方法」の対比を,「何を (What) 」と「どのように (How) 」の対比として捉えることで, ソフトウェア開発では「 What と How を分けて考えるとよい」と言われている．これは, 仕様策定工程では What を検討することに集中し, 設計工程では How に集中するということである．本論文では, この「解決すべき問題」である What と「問題の解決方法」である How を分けて考えることを, 仕様と設計の分離と呼ぶこととする．

仕様と設計を分離することは, ソフトウェア開発において広く言われていることではあるが, 実際に, 仕様策定者が仕様と設計を分離して仕様を検討し, 仕様書に「解決すべき問題」のみを簡潔に記述することは容易ではない．仕様と設計の境界が曖昧な仕様書が設計者に渡された場合, 設計者は, 仕様が規定する範囲と設計が規定すべき範囲を仕様書から読み取ることができない．仕様書内の実現手段を含んだ記述について, 設計の自由度がどこまであるのかを設計者は仕様策定者に尋ねなければならない．仕様策定工程

において, この仕様と設計の分離について十分に議論されていることは少ないため, 仕様策定者は設計者の質問に答えられない, あるいは, 仕様と設計の分離に関する議論が仕様書の広範囲に渡るため, 個別に答えることが難しい場合が多い. この場合, 結局のところ, 仕様と設計の分離に関する検討は, 仕様の読み手である設計者任せとなることが多いため, 仕様策定者が意図した仕様とは異なる製品が設計されてしまう場合や, 設計者が仕様の制約を過剰に解釈し複雑な設計となる場合がある. その結果, 開発効率が低下してしまう問題や, 製品のパフォーマンスに影響を与えてしまう問題, 不具合や市場トラブルの問題を引き起こす可能性がある.

以上のことから, 仕様策定工程において, 仕様が規定する範囲と設計が規定する範囲を設計者が容易に読み取ることができ, 理解容易性に優れた形式仕様を記述する必要がある. 本研究では, 開発対象の内部データ構造に着目することで, 仕様と設計を分離する方法を提案する. 3.3 節において, 関連する Michael Jackson の「実装の影響」に関する研究 [Jac95] から課題をさらに考察し, 4.2 節において解決方法を提案する.

2.2 レビューとテストの用途として形式仕様記述を利用する場合の課題

2.2.1 レビューとテストにおける一般的な課題

現在, ソフトウェア開発においてレビューとテストが産業界において「当然の技術」として広く用いられている. しかし, レビューとテストには様々な課題があることが知られている. 主な課題を以下に示す.

- 一般にテストは設計と実装の後に行われるため, テストにより仕様の不具合が見つかった場合の手戻りが, 上流工程で不具合を見つけた場合と比較して大きくなる. 仕様に関する不具合が下流工程で実施するテストにより見つかることで, 手戻りが発生することが課題である.

- ソフトウェア開発において正しさは相対的であり [Mey00], ソフトウェアが正しいこ

とを確認するには基準が必要である. 仕様は, テストとレビューにおいて正しさの基準となる. そのため, 仕様が曖昧であると, レビューとテストにより不具合を見つけることが難しくなることが課題である.

- レビューは品質確保の主要な手段であり, ソフトウェア開発において品質確保に効果があることが知られている [MDL87, MST11]. しかし, 繰り返し機械的に実施できないため, 変更が入った場合のレビューの運用と, レビューによる検証の網羅性が課題となる.

これらの課題に対して, 形式仕様記述手法を適用することにより, 以下の効果を期待することができる.

- 上流工程で形式仕様記述言語を用いて仕様を記述することにより, テストによる検証手段を仕様アニメーションにより用いて, 早期に不具合を見つけることができる.
- 形式仕様記述言語を用いて厳密に仕様を記述することにより, 正しさの基準としての仕様を明確に定義することで, テスト工程を担当する評価者に正しさの記述を明確に伝えることができる.
- 形式仕様記述言語を用いて仕様を記述することにより, 仕様の構文検査, 型検査, 単体テストなどの機械的な検証手段を用いることができる.

これらの広く知られた形式仕様記述手法の効果は, FeliCa IC チップ開発の第 1 世代の適用において経験することができた. 以降の各節において述べる課題は, 上記のテストとレビューにおける一般的な課題とは異なる. 以降で述べる課題は, 第 1 世代の形式仕様記述手法の適用において明らかになったことにより, 第 2 世代において取り組んできたものである.

2.2.2　レビューの用途から見た仕様記述の理解容易性に関する課題

開発プロジェクトにおいて, ドメインエンジニア・仕様策定者・設計者・実装者・評価者など, 様々な役割を担った読者が仕様書を参照する. また, 仕様を読む目的によって,

これらの読者が必要とする仕様の詳細度は異なる. 仕様が要求を満たしていることをドメインエンジニアがレビューする場合に必要となる仕様の詳細度と, 設計者と実装者が, 仕様と実装コードをレビューする場合の詳細度は異なる. ドメインエンジニアが必要とする仕様の詳細度と比べ, 設計者と実装者の方が, より細かい詳細度の仕様が必要となることが多い. 評価者が, 仕様とテスト項目表をレビューする場合も, ドメインエンジニアよりも細かい仕様の詳細度を必要とすることが多い.

様々なレビューの目的を持った仕様書の読者にとって, 読みやすい仕様書とは, 次の 3 つの課題を解決したものであると考える.

- 概要と詳細を階層化して明確に区別して記述してあり, 概要のみを読むことで, 仕様の全体像を理解することができる仕様記述.
- 概要から詳細へと即座にたどることができる仕様記述.
- 形式仕様記述手法の専門家ではない開発者であっても, 仕様を読み進めることができるように, 特に概要の記述は簡潔な構造を持つ仕様記述.

このような仕様であれば, ドメインエンジニアが要求を満たしていることを確認するために, まず仕様の概要を読み進め, 必要に応じて概要から詳細を読むことができる. 実装者と設計者と評価者は, 概要を把握した上で, 概要と詳細な条件式を往き来しながら, 仕様を読み進めることができる.

本研究では, レビューの用途を考慮した, 前述の 3 つの課題を解決する仕様記述フレームワークの設計・構成法を提案する. 提案手法の詳細を 5.1 節において述べる.

2.2.3 仕様に基づくテストの用途から見た課題

本節では, 形式仕様記述を仕様に基づくテストの工程において, 形式仕様記述手法を活用するための課題について考察する.

一般に, 要求から考えた利用シーケンスをテストスクリプトとして記述し, 仕様アニメーションによりテストスクリプトを形式仕様に対して実行することで, 要求通りに仕

様が記述されていることを確認することができる. これにより, テストスクリプトとして記述した要求を, 仕様が満たしていることを確認することができる. 本研究において述べる仕様に基づくテストとは, 前述の仕様が要求を満たしていることを確認することではなく, 開発対象の実装が, 仕様を満たしていることを確認するテストである. 一般に, 実装が仕様を満たしていることを確認するために, 仕様書を基にテスト項目とテストスクリプトを作成し, 開発対象の実装に対してテストスクリプトを実行する. 本研究では, この方法を形式仕様記述に応用する. まず, 形式仕様記述を基にテスト項目とテストスクリプトを作成する. 次に, 作成したスクリプトを形式仕様記述に対して仕様アニメーションにより実行して, テストの実行ログを得る. 最後に, 形式仕様記述に対して実行したテストスクリプトを, 開発対象の実装に対しても実行し, 得られたテストの実行ログと仕様アニメーションにより得られた実行ログを比較することにより, 開発対象の実装が, 仕様を満たしていることを確認する. ここで述べたテストの実行ログとは, 開発対象への入力データや開発対象の出力データ, デバッグのために開発対象から出力した内部状態のデータなどである.

仕様に基づくテストの工程において, 形式仕様記述手法を活用することで, メリットを感じることができるように, 本研究では次の 2 つの研究課題を設定した.

- まず, 仕様に基づくテストにおいて前提とする品質目標を決める. その上で, 品質目標を満たす体系的なテスト項目の作成方法を研究課題とした. これは, 体系的にテスト項目を作成することができるような, 形式仕様記述の設計を行わなければならないことでもある.

- テスト項目とテストスクリプトが品質目標を満たしていることを, 形式仕様記述を用いた機械的な処理により確認する方法を研究課題とした.

5 章において, 本研究において定めた仕様に基づくテストの品質目標と, 体系的にテスト項目を作成するためのテスト設計および, そのための仕様記述フレームワークを提案する. そして, テスト項目とテストスクリプトが, 品質目標を満たしていることを確認するための, ログ出力による機械的な検証方法を提案する.

2.3 形式仕様記述手法では取扱いが容易でない動的な振る舞いに関する課題

形式仕様記述手法を用いることで, 仕様策定工程において早期に仕様の検証を行うことができた. 形式仕様記述手法の検証対象は, 入力データに対する内部状態の遷移と出力するデータの仕様を記述した, 静的な機能仕様である. 形式仕様記述手法を用いることで, 早期にこれらの機能仕様を検証することができた. しかし, 同時並行に動作する動的な振る舞いについて, 筆者らが用いてきた形式仕様記述手法では取扱いが容易ではなかった.

そこで, 第 1 世代の FeliCa IC チップ開発における仕様策定後の設計工程において, 同時並行に動作する動的な振る舞いに関する検証方法を検討する必要があった. 検証方法としてモデル検査に着目した. 本研究では, モデル検査の段階的な導入方法と, モデル検査を実施する上で課題となった, 検査対象の状態数の削減方法について議論する. 詳細を 6 章において述べる.

第3章

関連研究

本章では, 関連研究として産業界へのフォーマルメソッドの適用事例について概観し, 筆者らのフォーマルメソッドの適用事例について考察する. また, 2 章において述べた本研究で扱う 3 つの課題のうち, 次の 2 つの課題に関する関連研究について述べる. 1 つは, 2.1.1 節において述べた「伝えること」と「動かすこと」を目的とした実行可能仕様の課題であり, 2 つ目は, 2.1.2 節において述べた仕様と設計の分離の課題である. これらの関連研究から, 本研究において扱う課題を考察し, 関連研究と本研究の提案手法を比較する.

3.1 フォーマルメソッドの産業界への適用事例

本節では, フォーマルメソッドの適用事例を紹介する. フォーマルメソッドの適用事例として, CICS [HK91], CDIS [Hal96], パリの地下鉄とシャルル・ド・ゴール国際空港のシャトル [Abr06] などが有名である. 本節では, これらの事例について述べた後に, 1.5 節で述べた筆者らの FeliCa IC チップ開発への形式仕様記述手法の適用事例と他の適応事例を比較する.

3.1.1　CICS

1980 年代に英国 IBM ハーズレー研究所とオックスフォード大学は, IBM 社のオンライントランザクション管理システム CICS (Customer Information Control System) のシステム改版時に Z 記法を仕様記述と設計に用いた [HK91]. システムの改版では, 全体として 800K LOC (Line of Code) のうち, 268K LOC を新規作成もしくは変更した. このうち, Z 記法を用いて仕様策定と設計を行った範囲はおよそ 37K LOC である.

Z 記法を用いることにより, 品質改善と開発コストの削減を達成することができたとしている. Z 記法を用いたコードと, 用いなかったコードの開発ライフサイクルごとの不具合検出率を比較したところ, Z 記法を用いたコードの方が早期にバグを見つけることができた. 不具合を修正するのに費やした日数を基に開発コストを IBM が見積もったところ, Z 記法を用いた箇所は Z 記法を用いていない箇所と比較して開発コストを 9% 削減することができたとしている.

その後, CICS の成功によりオックスフォード大学と IBM 社は 1992 年に英国の賞「 The Queen's Award for Technological Achievement 1992 」を受賞した.

3.1.2　CDIS

英国の Praxis 社は, ロンドン国際空港の航空管制システムの一部である CDIS (CCF Display Information System) の開発にフォーマルメソッドを用いた [Hal96]. CCF とは, 中央管理機能 (Central Control Function) の略称である. CDIS は, 航空機の発着, 空港の気象条件や設備の状態, データ入力スタッフが提供するサポート情報を表示する装置であり, フォールトトレラントシステムである. 1992 年に Praxis 社から UK Civil Aviation Authority に納入された. CDIS の規模は 197K LOC であり, 総工数は 15,500 人日であった.

機能要求の記述には VDM を用い, システム仕様の記述には VVSL[Mid89] を用いた. VVSL は VDM から言語仕様を拡張し, モジュール構造をより明確に記述できるようにした仕様記述言語である. システム仕様は, 「フォーマルコア仕様」, 「ユーザーインタ

フェース定義」,「同時並行仕様」の 3 つの記述からなる.「フォーマルコア仕様」は, CDIS によって管理されているデータと, それぞれの操作に関する仕様である. それぞれの操作は, 入力と出力, 状態への影響を意味定義レベルで定義している. これらの操作は,「ユーザインタフェース定義」で記述している操作と結びついている.「ユーザインタフェース定義」は, ユーザと装置間における CDIS の操作に必要なやりとり, 必要なキー操作やマウス操作, スクリーンの表示に関して記述している.「同時並行仕様」は, CDIS のユーザ入力装置や外部システムおよび故障と復旧を監視しているハードウェア装置などの同時並行に実行されているプロセスの仕様であり, CSP[Hoa78] と VVSL を用いて記述した.

全体の生産性は, 同等規模の類似する開発プロジェクトと比較して同程度かもしくは良い結果となった. 品質の面では, 出荷後 20 ヶ月で 150 件の不具合が報告され, 不具合密度は約 0.75 件/KLOC であった. これは, 同程度のプロジェクトと比較するときわめて少なかった.

3.1.3 パリの地下鉄とシャルル・ド・ゴール国際空港のシャトル

Jean-Raymond Abrial らはパリの地下鉄とシャルル・ド・ゴール国際空港のシャトルの開発に B メソッドを用いて自動運行制御システムを開発した [Abr06]. B メソッドは, 抽象的な形式仕様を段階的に具体化して, 実行可能なプログラムを作成する段階的詳細化の開発法である. この適用事例において, 仕様から段階的詳細化により Ada プログラムを導き出している. **表 3.1** に概要を示す.

3.1.4 本適用事例との比較

本研究において述べる筆者らのフォーマルメソッドの適用事例および, 設計・構成法に関する提案は, Abrial らの B メソッド適用事例のように, 抽象的な形式仕様を段階的に具体化して, 実行可能なプログラムを作成する段階的詳細化による開発法とは異なる. 産業界には, レビューやテストといった従来からある実践的な検証技術が既に浸透して

表 3.1　パリの地下鉄とシャルル・ド・ゴール国際空港の適用事例

項目	パリの地下鉄	シャルル・ド・ゴール国際空港のシャトル
Ada の行数	86,000	158,000
証明の数	27,800	43,610
対話的な証明の割合	8.1%	3.3%
対話的証明の工数	7.1 人月	4.6 人月

J.R. Abrial 「 Formal methods in industry 」[Abr06] より引用

いる. 筆者らは, これらの実践的検証技術を主体として, その中にフォーマルメソッドを組み合わせた検証手法を提案する. フォーマルメソッドを用いたソフトウェア開発の最終的な目標は, 要求, 仕様, 設計, 実装と各ステップにおいて正しさを保証し, 正しさを保証しながら開発のステップを進めていく, Correctness by construction の考え方である. しかしながら, 現状は誰もがすぐに Correctness by construction の考え方に従って開発を行える環境ではない. 各々の開発現場においては, 既に蓄積してきたレビューやテスト手法などの検証技術があり, その検証技術を主体としてフォーマルメソッドを用いた開発法を議論することで, 開発現場の多くの課題を解決することができると考える. 本研究では, 従来からある検証技術の中にフォーマルメソッドを適用した事例について述べ, 適用に必要な設計・構成法の提案を行う.

3.2　実行可能仕様における「実行可能性の影響」

2 章において, 本研究において扱う課題について考察した. 本節では, これらの課題のうち, 2.1.1 節において述べた, 「伝えること」と「動かすこと」を目的とした実行可能仕様の課題について述べる. この, 仕様の実行可能性が仕様記述に与える影響に関する関連研究から, 課題を考察し, 関連研究と本研究において提案する手法とを比較する.

実行可能仕様には, 何を作るべきかという「解決すべき問題」を各工程に明確に「伝

えること」と, 仕様アニメーションにより「動かすこと」を目的にした記述が混在することになり, 課題があることが知られている. Hayes と Jones は, 1990 年に発表した「Specifications are not (necessarily) executable」[HJ89] の中で次の 2 つの課題を述べた.

- 仕様は, 「解決すべき問題」 What を規定したものである. 実行可能仕様により, 「問題の解決方法」 How を多く含んだ記述となり, 概して要求されていることよりも多くのことを規定することになる. それにより, 実装の選択肢を狭めてしまう.
- どのように結果を計算するかというアルゴリズムに関する詳細な記述により, 複雑な記述になる傾向がある.

Hayes と Jones はこれらの課題を解決するために, 次のような提案をした. 実行可能仕様は, 形式仕様記述手法ではなく, ラピッドプロトタイピングとして分類した方がよく, 「動かすこと」を目的としたプロトタイプとしての用途と, 「伝えること」を目的とした用途は分けるべきだと主張した. また, Hayes と Jones は, 仕様を動かすことは考えずに, 開発対象に求められている性質 (property) を記述することに集中し, 仕様の検証には証明を使うべきだと述べている.

本研究では, 証明ではなく, レビューやテストといった従来からある実践的な検証技術を, 形式仕様記述手法と組み合わせて用いた. テストの検証技術を用いて, 実行可能仕様を仕様アニメーションにより検証した. Hayes と Jones が指摘した, 「動かすこと」を目的とした記述と「伝えること」を目的とした記述が混在する問題について, 2 つの記述を分離する手法を提案する. 詳細は 4.1 節で議論する.

また, Hayes と Jones は, 簡潔性と正確性の側面に着目し, How を表現した記述が What の記述に与える影響について述べた. 本研究では, Hayes と Jones が指摘したこの側面に加え, 開発工程全体で形式仕様を活用するために必要な, レビューの用途を考慮した仕様記述を研究課題とした. ドメインエンジニアや仕様策定者, 設計者, 実装者, 評価者といった様々な役割を担った読者が, 形式仕様記述をレビューに用いる点に着目した. 2.2.2 節において, 理解容易性に関して優れた仕様記述について次のように考察した.

- 概要と詳細を階層化して明確に区別して記述してあり, 概要のみを読むことで, 仕

様の全体像を理解することができる仕様記述.

- 概要から詳細へと即座にたどることができる仕様記述.
- 形式仕様記述手法の専門家ではない開発者であっても, 仕様を読み進めることができるように, 特に概要の記述は簡潔な構造を持つ仕様記述.

本研究では, この課題を解決する形式仕様記述の設計・構成法を提案する. 詳細は, 5.1 節において議論する.

3.3　仕様と設計の分離に関する「実装の影響」の研究

2 章において, 本研究において扱う課題について考察した. 本節では, これらの課題のうち, 2.1.2 節において述べた, 仕様と設計の分離に関する関連研究について述べる. この関連研究から本研究において提案する手法を考察する.

Michael Jackson は 1995 年に発表した「 Software Requirements & Specifications 」の「実装の影響」という章において, 仕様と設計の分離について考察をしている [Jac95]. また, Pamela Zave と Jackson が 1997 年に発表した「 Four dark corners of requirements engineering 」の中でも同様に仕様と設計の分離について述べている [ZJ97]. 仕様は「実装の影響」を受けるべきではないという理由は, 3.2 節と同様に What と How を分けることで, いかに作るかということに惑わされることなく, 何を作るのかに集中することができるためである. その結果, 仕様を簡潔に記述し, 設計者に設計の自由度を持たせることができる.

以下では, 「 Software Requirements & Specifications 」を基に, まず Jackson が示した要求と仕様, プログラムを区別する考え方と, 適用領域の考え方について述べる. 次に, これらの概念を使って Jackson が述べた「実装の影響」について考察する. 最後に, Jackson の主張する方法と, 本研究において提案する手法とを比較する.

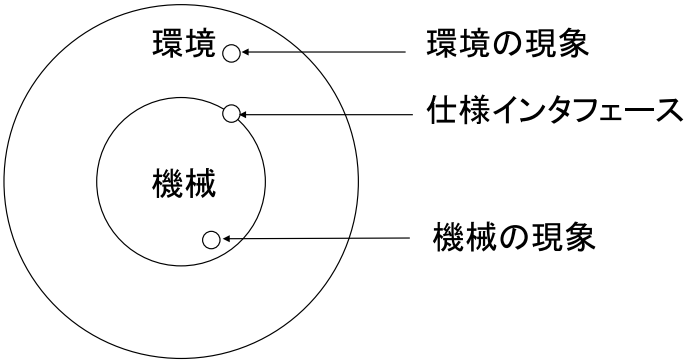

図 3.1 要求と仕様とプログラム

Jackson 「 Software Requirements & Specifications 」[Jac95] より引用

3.3.1 「要求と仕様とプログラム」と「適用領域」

Jackson は次のような要求と仕様とプログラムを区別する考え方を示した. **図 3.1** に Jackson が用いた図を引用する. この図が表していることは, 要求はシステムの外にある環境の現象について述べたものであり, 機械と表記しているプログラムはシステム内の現象について述べたものであり, 仕様はシステムの内と外の境界上にある現象について述べたものであることということである. Jackson はシステムの内側のことを機械と呼び, 環境と機械の境界上を仕様インタフェース呼んでいる. つまり, 環境と機械の間で共有された事象や状態からなるインタフェースを仕様と定義した.

さらに Jackson は, 要求や仕様をより正確に議論する上で, 環境という言葉の代わりに適用領域という用語を用いている. その理由を次のように述べている [Jac95].

> 環境という言葉の代わりに適用領域という用語を使うのには理由がある. 少なくとも私にとって, 環境という言葉は機械を取り巻く単なる物理的な世界をイメージさせる. これに対して適用領域には実際には手にふれることのできないさまざまな対象 (図形描画のためのイメージや, 給与支払システムのための表や合意事項そして従業員規定, ワードプロセッサのためのフォント, 航空管制システムのための安全航行規定, コンパイラのための原始コード) などが含まれる. こういったものは機械を取り巻いていないし, 逆に機械の中

に取り込まれていると考えたほうがよい場合が多い.

3.3.2　「実装の影響」

Jackson は, 要求・仕様・プログラムを区別する考え方と適用領域の考え方から「実装の影響」について次のように考察している.

仕様は, 機械の外的な振る舞い, すなわち連続した外部操作に対する反応を記述したものである. 図書館の管理システムにおける外部操作の例として StartNewLibrary や AcquireBook (b), LendBook (b), RetrunBook (b) などがある. 外部操作に対する反応を記述するためには, 図書館の管理システムはどの本が貸出中であり, どの本が図書館のなかに残っているのかを知っていなければならない. 内部状態として, 本のレコードを配列を用いて管理するのか, ハッシュを用いて管理するのかを考えなければならない. Jackson は, 機械の内部状態を用いて外部操作を規定している限りは, 仕様は「実装の影響」を受けると主張する. Jaskson はその理由を「システムが何かを覚えておくという考えかたそのものが, 実装固有の考えだからである」と述べている.

仕様が「実装の影響」を受けないために, Jackson は次の提案をした. その提案は, 「もし仕様のなかでなんらかの状態を記述しなければならないとすると, それは, 機械の状態ではなく, すべて [適用領域] の状態でなければならない.」[Jac95] というものである. Jackson が用いた具体例を次に示す [Jac95].

> 例えば PublishedOn (b,d) が, 本 b が日付 d に出版されたときに観察される現象だとしよう. そして EalierDate (d,e) が日付 d が日付 e よりも早いことを表す領域の現象であるとする. このとき本 b が本 c より順序が早いということを表す EalierInBookSequence (b,c) は, 次のように観察可能な領域の現象だけを用いて定義すれば実装の影響をのがれることができる.
>
> $EalierInBookSequence(b, c) \triangleq$
>
> $\exists d, e \bullet PublishedOn(b, d) \wedge PublishedOn(c, e) \wedge EalierDate(d, e)$

3.3.3 本提案手法との比較

確かに, Jackson の提案する方法を用いることで, 「実装の影響」を回避することができる. しかし, 本研究では, 仕様策定工程において, 機械の内部状態のデータ構造をある程度, 限定することができる場合を前提とし, 開発効率の観点から Jackson の提案する手法とは別の方法を提案する. その理由を以下において論じる.

Jackson が述べた適用領域の現象のみを用いて仕様を記述することで, 機械の内部状態を考慮する必要はなくなるが, 何らかのデータ構造を定義する必要がある. 例えば, 前述の図書館の管理システムにおいて, lendOut (lib, b, d) が, 図書館 lib から本 b が日付 d に貸し出されたときに観測される現象だとする. lendOut 関数は, 前述の PublishedOn 関数と同様に TRUE または FALSE を戻り値とする. TRUE を返す場合は, 図書館 lib の本 b の貸出日と日付 d が一致する場合であり, 一致しない場合は FALSE を返す. この関数の仕様を記述するためには, 図書館 lib の本 d の貸出日を管理するデータ構造が必要となる. つまり, 仕様策定者は, 仕様を記述するために, 適用領域の現象から見た上記の貸出日を管理するようなデータ構造を定義する必要がある. Jackson の主張は, 仕様策定工程で定義した適用領域のデータ構造は, 機械のデータ構造ではないから, 「実装の影響」を受けないということである. そのため, 適用領域から見たデータ構造の定義と, 機械の内部状態に関するデータ構造の定義の間に, 関連を持たせることは難しい.

仕様策定工程において, 機械の内部状態のデータ構造をある程度は限定できる場合, 仕様策定と設計と実装とテストの開発効率から Jackson の主張を考察する. 設計者は, まず, 仕様を読むために, 適用領域の現象から見たデータ構造を理解し, 設計を行う中で機械の内部状態に関するデータ構造を新たに定義していく. 実装者や評価者は, 仕様書と設計書を参照するため, 適用領域の現象から見たデータ構造と, 機械の内部状態に関するデータ構造の 2 つのデータ構造を理解する必要がある. Jackson の主張から, 両者のデータ構造に関連があることを期待することは難しい. 仕様策定, 設計, 実装, テストにおける開発効率は, 関連性のない 2 つのデータ構造を定義するよりは, 両者の間の関連性を許容した方が向上する. なぜなら, 仕様策定工程において定義したデータ構造を, 設計工程においても流用することができるからである. これにより, 設計者が新たにデータ構造を

検討して設計書に記述する工数や, 設計者や実装者や評価者が 2 つのデータ構造を理解するために必要な工数を削減することができる. また, データ構造を流用することで, 仕様に基づいたテストと設計に基づいたテストの観点が近くなる. したがって, 仕様に基づいたテストによって, 設計に基づいたテストの一部を省略できる可能性がある. これにより, 設計に基づいたテストに費やす工数を削減することができる.

本研究では, 仕様策定工程において, 機械の内部状態のデータ構造をある程度, 限定することができる場合を前提とし, 仕様で定義するデータ構造と, 設計で定義するデータ構造の間に関連性を許容した. この上で, 「実装の影響」を回避する方法を提案する. 提案手法の概要を次に述べる. まず, データ構造を定義するための型を, 仕様において規定する型と, 仕様においては規定しない型に分類する. 次に, 仕様において規定しない型の定義・演算子を, 関数を定義することにより隠蔽する. つまり, 仕様において規定する型と, 仕様において規定しない型を明確に区別することで, 「実装の影響」を回避する. 仕様において規定しない型を隠蔽することで, 仕様書の読み手には, 仕様において規定する型のみを提示することができる. Jackson が述べた, 適用領域に関するデータ構造と, 機械の内部データ構造とが同じ場合は, 仕様において規定する型となり, 異なる場合は, 仕様において規定しない型となる傾向がある. これらの型の分類は, 仕様策定者と設計者が協議をすることにより行った.

型の隠蔽は, ソフトウェア開発で用いられている情報隠蔽 [Par72, Par79, PCW85] の技術であるので, 現在は「当然の技術」としてソフトウェア開発において用いられている. 本論文で提案する手法は, Jackson と Zave が述べた「実装の影響」に関する考え方と, 開発現場において「当然の技術」として用いられている情報隠蔽の考え方を組み合わせたものである. これにより, 実際の開発現場において浸透しやすい形式仕様記述の設計・構成法に関する提案であると考える. 具体的な設計・構成法は 4.2 節で述べる.

第4章

仕様と設計を分離する実行可能仕様の設計・構成法の提案

本章では, 2 章と 3 章において述べた研究課題のうち, 2 つの課題に対する解決方法を提案し, 提案方法の評価を行う. 1 つ目の課題は, 「伝えること」と「動かすこと」を目的とした実行可能仕様の課題である. この課題は, 2.1.1 節において考察し, 3.2 節において Hayes と Jones の指摘する「実行可能性の影響」の関連研究から議論した. 2 つ目の課題は, 仕様と設計の分離に関する課題である. この課題は, 2.1.2 節において考察し, 3.3 節において Jackson の「実装の影響」の関連研究から議論した.

以降の各節では, まず, それぞれの課題を解決する方法を提案し, 次に, 提案手法を用いた具体例を示すことで, それぞれの解決方法を評価する. 最後に, FeliCa IC チップ開発へ本提案手法を適用した結果を考察する.

1 つ目の「実行可能性の影響」に関する課題について, この課題を解決する仕様記述スタイルを提案する. まず, この課題をより具体的に議論するために, 操作的な記述と宣言的な記述の観点と, 仕様アニメーションによる動作検証の作業工数という 2 つの観点を加えて議論を進める. そして, これらの課題を解決する仕様記述スタイルを提案する.

2 つ目の仕様と設計の分離に関する課題について, 仕様において定義するデータ構造と設計において定義するデータ構造に着目し, この課題を解決する形式仕様記述の設計・

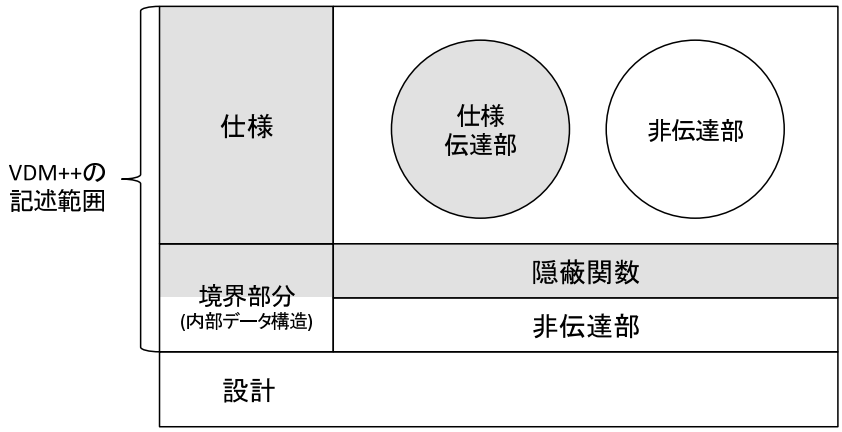

図 4.1　提案する形式仕様記述の設計・構成法の概略図

構成法を提案する.

本章において提案する形式仕様記述の設計・構成法の概略図を**図 4.1** に図示する. 仕様と設計の記述を次の 3 層に分けた. 上から, 仕様の記述, 内部データ構造に着目した仕様と設計の境界部分の記述, 設計の記述である. VDM++ の記述範囲は, 設計より上の層になる. VDM++ の記述範囲に対して, 「伝えること」を目的とした記述範囲をグレーで塗りつぶした. VDM++ の記述範囲に対して, グレーで塗りつぶしていない範囲は「動かすこと」を目的とした記述範囲となる.

本章で扱う 1 つ目の課題である「実行可能性の影響」に関する課題を解決するために, 図 4.1 に示した設計より上の層をさらに, 仕様伝達部と非伝達部に分けた. 仕様伝達部と非伝達部については, 4.1.2 節において後述する. 本章で扱う 2 つ目の課題である仕様と設計の分離に関する課題を解決するために, 隠蔽関数により設計上の内部データ構造を隠蔽した. 隠蔽関数については, 4.2 節において述べる.

まず, 次節において 1 つ目の課題である「実行可能性の影響」を解決する仕様記述スタイルについて議論する.

4.1 「実行可能性の影響」を解決する仕様記述スタイルの提案と考察

2.1.1 節において, 「伝えること」と「動かすこと」を目的とした実行可能仕様の課題について考察した. この課題は, 次のようなものであった. 従来からある既に開発現場に浸透しているテスト手法を用いて, 仕様を検証するには, 実行可能仕様を記述して, 仕様アニメーションにより動作検証を行う必要があった. 一方で, 理解容易性に優れた仕様を記述するためには, 「解決すべき問題」である What を仕様の読み手に明確に伝える必要があった. ここでの課題は, 「動かすこと」を目的とした「問題の解決方法」である How を含んだ記述が, 「伝えること」を目的とした仕様記述の理解容易性に影響を与えるということであった. 結果, 「問題の解決方法」である How の要素を含んだ記述が, 設計者を過剰に制約し, 実装の選択肢を狭めてしまう場合や, 仕様の記述が複雑になる場合がある. これが「実行可能性の影響」に関する課題であった.

本研究では, この課題を解決する仕様記述スタイルを提案する. 提案の記述スタイルを用いることにより, 宣言的な記述を用いて「解決すべき問題」である What を記述することができるようになる. さらに, この What の記述と, 実行可能仕様として「動かすこと」を目的とした How の記述を明確に分離することができるようになるため, How の記述が What の記述の理解容易性を妨げることを避けることができる. その結果, What を記述した読みやすい仕様と, 従来からあるテスト手法を用いて検証することができる仕様を記述することができるようになる.

本節では, まず, 「実行可能性の影響」について, 操作的な記述と宣言的な記述という観点と, 仕様アニメーションによる動作検証の作業工数という観点を加えて, より具体的に課題を考察する. 次に, これらの課題を解決する仕様記述スタイルを提案する. 最後に, VDM++ を用いて提案の仕様記述スタイルを記述する場合の設計・構成法について議論する.

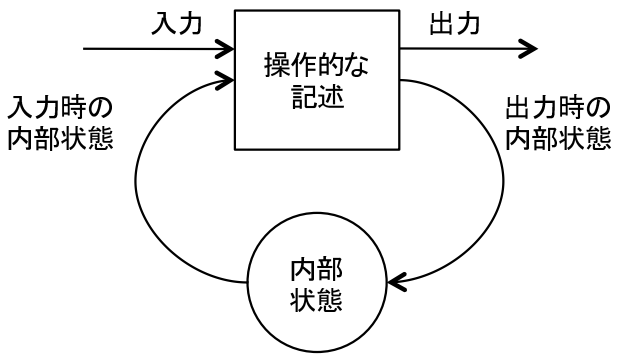

図 4.2　模式的に表した操作的な記述

4.1.1　操作的な記述と宣言的な記述から見た実行可能仕様の課題

本節では, 操作的な記述と宣言的な記述という側面から, 「実行可能性の影響」の課題をより具体的に考察する. 本論文では, 操作的な記述と宣言的な記述を次のように定義することとする. **図 4.2** において操作的な記述を, **図 4.3** において宣言的な記述を模式的に表した. 操作的な記述とは「入力」と「入力時の内部状態」を使用して, 「出力」と「出力時の内部状態」を作成する過程を記述したものとする. 一方, 宣言的な記述とは「入力」と「入力時の内部状態」, 「出力」と「出力時の内部状態」を使用して, 入出力の関係を記述したものとする. 4.1.3 節においても述べるが, 平方根を求める場合, 操作的な記述は「入力」からニュートン法などにより「出力」を求める過程を記述する. 宣言的な記述の場合, 「入力」と「出力」の関係を次のように記述する.

```
abs ( res**2 - x ) <= EPS
```

ここで, 「 abs 」は絶対値を求める関数, 「 res 」は 実数型の「出力」を表す変数, 「 ** 」はべき乗の演算子, 「 x 」は「入力」を表す実数型の変数, 「 <= 」は左辺より右辺の方が大きいか等しいときに true となる演算子, 「 EPS 」は許容誤差を表す実数型の定数である. この記述は, 「出力」を 2 乗した値と「入力」との差の絶対値は, 許容誤差以内であることと読むことができる. このように, 宣言的な記述は「入力」と「出力」を用いて入出力の関係を記述したものとする.

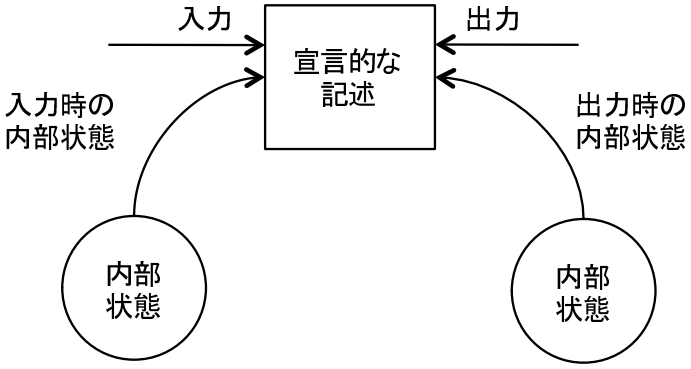

図 4.3 模式的に表した宣言的な記述

操作的な記述の場合, 「解決すべき問題」である What よりも「問題の解決方法」である How を多く含んだ記述となり, 仕様が過剰に設計者を制約し, さらに, 記述が複雑になる傾向がある. 一方, 宣言的な記述の場合, 記述方法にもよるが, 入出力の関係を条件式を用いて記述することで, 「実行可能性の影響」を回避することができる.

しかし, 宣言的な記述を用いて, 実行可能仕様を記述した場合, 実行するために必要な評価者の作業工数が課題となる場合がある. この作業工数の課題について考察する. 操作的な記述の場合, 図 4.2 に示すように, 「入力」と「入力時の内部状態」から, 仕様アニメーションにより仕様を動かし, 「出力」と「出力時の内部状態」を得ることができる. 一方, 宣言的な記述の場合, 図 4.3 に示すように, 仕様アニメーションにより動かすためには, 「入力」と「入力時の内部状態」に加え, 「出力」と「出力時の内部状態」が必要になる. つまり, 宣言的な記述を仕様アニメーションにより動かすためには, あかかじめ「出力」と「出力時の内部状態」を作成しておく必要がある. これは, 「出力」と「出力時の内部状態」を容易に作成することができる場合は課題とはならないが, 複雑な内部状態を持つ仕様の場合, 評価者の作業工数が課題となる場合が多い.

作業工数の点で考察すると, 操作的な記述の場合においても, あらかじめ「出力」と「出力時の内部状態」の一部を作成しておく必要がある. 仕様アニメーションを行う場合, 実行結果とあらかじめ作成しておいた期待値とを比較することで, 実行結果が正しいことを確認する. そのため, この期待値としての「出力」と「出力時の内部状態」をあらかじめ作成しておく必要がある. しかし, このとき, 評価者は期待値として作成する「出力

時の内部状態」を, 実行結果が変化した範囲だけとするのか, 変化しない範囲も含めた全てとするのかを選択することができる. この選択は, 検証対象としている関数の重要度や複雑度, 内部状態を作成するのに必要な作業工数など, 評価者が費用対効果を考慮して決めることができる. 一方, 宣言的な記述の場合, 常に全ての「出力」と「出力時の内部状態」を作成しなければならないため, 評価者の作業工数が課題となることが多い.

つまり, 操作的な記述の場合は, 「実行可能性の影響」を受けることが課題となり, 宣言的な記述の場合, 「実行可能性の影響」を回避することができるが, 仕様アニメーションにより動かすときに, 評価者の作業工数が課題となる. この課題を解決する仕様記述スタイルを次の 4.1.2 節において提案する.

4.1.2　「実行可能性の影響」を解決する仕様記述スタイルの提案

操作的な記述は, 「出力」と「出力時の内部状態」を生成することができるが, 「実行可能性の影響」を受ける. 一方, 宣言的な記述は, 「実行可能性の影響」を回避することができるが, 仕様アニメーションにより動かすときに, あらかじめ, 「出力」と「出力時の内部状態」を生成しておく必要があるため, 評価者の作業工数が課題であった. 「実行可能性の影響」を回避して, さらに, 評価者の作業工数の課題を解決するために, **図 4.4** に示す仕様記述スタイルを提案する. この図は, 操作的な記述を模式的に表した図 4.2 と, 宣言的な記述を模式的に表した図 4.3 を組み合わせたものである. この記述スタイルは, 宣言的な記述の箇所に「伝えること」を目的とした仕様記述を配置して, 操作的な記述の箇所に「動かすこと」を目的とした記述を配置したものである. 本論文では, 図 4.4 の宣言的な記述箇所を仕様伝達部と呼び, 操作的な記述箇所を非伝達部と呼ぶ. 図 4.4 の操作的な記述の箇所において, 「入力」と「入力時の内部状態」を使用して, 「出力」と「出力時の内部状態」を作成する. 図 4.4 中の (1) において, 操作的な記述を仕様アニメーションにより動かすことで作成した「出力」と「出力時の内部状態」が出力される. この出力が, (2) において宣言的な記述への入力となっていることが分かる. つまり, 「入力」と「入力時の内部状態」を操作的な記述に与えることで, 宣言的な記述を動かすこ

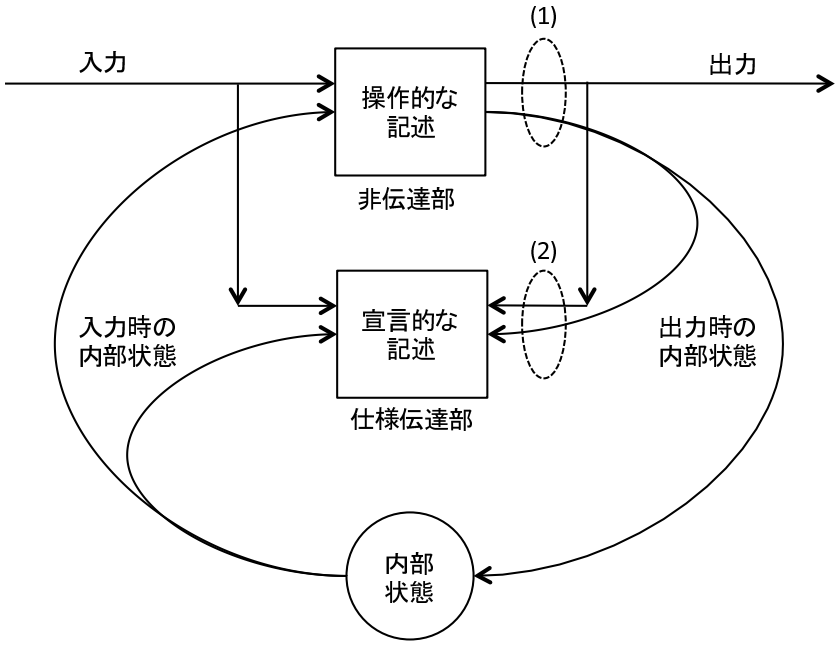

図 4.4 模式的に表した「実行可能性の影響」を解決する仕様記述スタイル

とができるようになる. これにより, 図 4.3 の宣言的な記述を, 仕様アニメーションにより動かすときに課題であった, 評価者の作業工数の課題を解決することができる.

以上により, 「伝えること」と「動かすこと」を目的とした実行可能仕様の課題を解決することができる. つまり, 宣言的な記述を用いて「解決すべき問題」である What を記述することができるようになる. さらに, この What の記述と実行可能仕様として「動かすこと」を目的とした How の記述を明確に分離することができるため, How の記述が What の記述の理解容易性を妨げることを避けることができる. また, 仕様アニメーションによる仕様の動作検証における, 評価者の作業工数の課題を解決することができる.

本節において定義した, 非伝達部に関して注意する点として, 操作的な記述箇所が非伝達部とはならないことがある. 例えば, 図 4.2 おいて, 操作的な記述箇所に, 仕様策定者が操作的な内容を「伝えること」を目的として記述した場合は, その記述は, 操作的な記述ではあるが, 仕様伝達部となり, 非伝達部とはならない. ただし, このとき仕様策定者は, 「問題の解決方法」である How の記述を行っていることを認識するか, あるいは, 「解決すべき問題」である What の記述を行っているのであれば, 「実行可能性の影響」を考慮しなければならない.

以降の各節では, 図 4.2 と図 4.3 と図 4.4 に示した記述スタイルを VDM++ を用いて記述する場合の設計・構成法ついて議論する. まず, 図 4.3 で示した宣言的な記述に対応する設計・構成法として, 陰関数定義について考察し, 次に, 図 4.2 で示した操作的な記述に対応する設計・構成法として, 陽関数定義について考察する. 最後に, 本研究において提案する図 4.4 の記述スタイルに対応する設計・構成法として, 拡張陽関数定義について考察する. **図 4.5** は, これらの関数定義と記述スタイルの関係を模式的に表したものである. 図は, 最上段に図 4.2 と図 4.3 と図 4.4 に示した記述スタイル名を示した. 次の段に, 各記述スタイルに対応する VDM++ の言語仕様に示された関数定義名を示した. その下に, 仕様アニメーションによる動作検証において, 評価者が入力として用意するテストデータを示した. 次に, その入力データについて, 仕様伝達部と非伝達部におけるデータの流れを示して, 出力をその次の段に示した. ここでは, 図 4.1 と同様に「伝えること」を目的とした仕様伝達部をグレーで塗りつぶしている. そして次の段に, この出力が正しいことを確認するために, 期待値として用意するテストデータを示した. 最後の行に, 記述スタイルの考察をまとめた. この図を用いて, 4.1.1 節と本節において分析した, 「実行可能性性の影響」の課題と, 実行するために必要な評価者の作業工数について, VDM++ の関数定義から議論する.

4.1.3　陰関数定義に関する考察

陰関数定義を用いることで, 図 4.3 に示した宣言的な記述を行うことができる. 陰関数定義は, 事前条件 (pre) と事後条件 (post) から構成される. VDM++ の言語仕様として, 陰関数定義の構文例を以下に示す.

```
f (a1:T1, a2:T2, ..., an:Tn) res : R
pre preexpr (a1, a2, ..., an)
post postexpr (a1, a2, ..., an, res)
```

ここで, f は関数名, T1,...,Tn はパラメータの型, a1,...,an はパラメータ名, R は戻り値の型であり, res は戻り値の識別子である. preexpr は入力の前提条件を定義した事前条

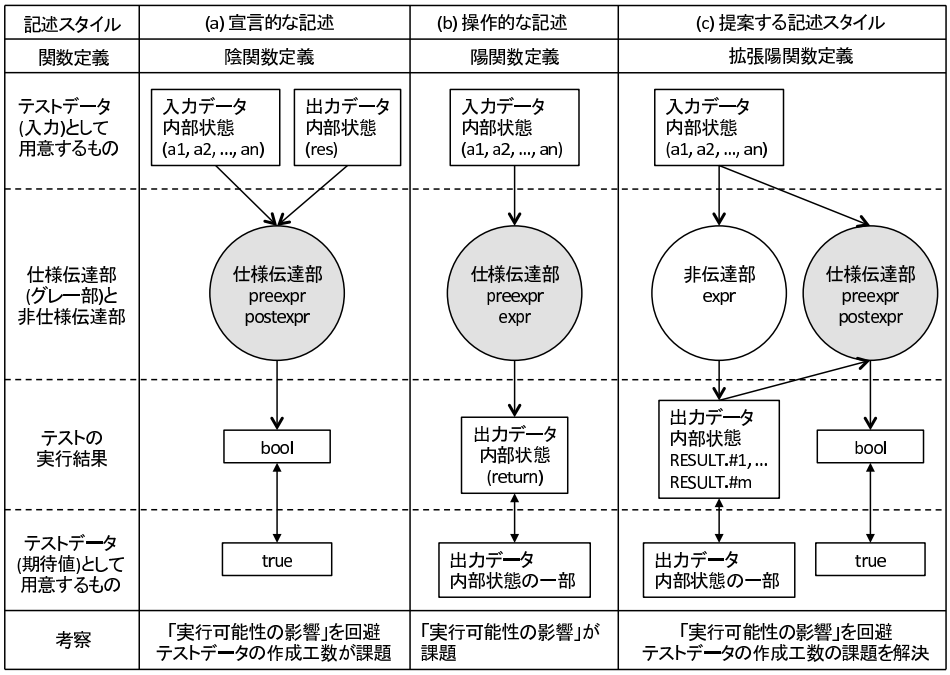

図 4.5 仕様記述スタイル

件であり, postexpr は関数 f 実行後に成立すべき条件を定義した事後条件である. 事後条件 postexpr の中で引数 a1,...,an と戻り値 res を用いて, 関数実行前後の関係を記述する. 陰関数定義を用いて平方根を求める仕様記述の具体例を次に示す.

```
public sqrt (x : real) res : real
pre x >= 0
post abs (res**2 - x ) <= EPS; -- EPS は許容誤差
```

sqrt 関数は実数型 (real) のパラメータ (x) を受け取り, 実数型 (real) の結果 (res) を返す関数として定義している. 事前条件 (pre) では, 関数実行前にはパラメータ (x) が 0 以上でなければならないことを記述している. 事後条件 (post) では, 結果 (res) を二乗したものとパラメータ (x) の差の絶対値 (abs) が, 許容誤差 (EPS) 以下でなければならないことを記述している.

この仕様は VDMTools のインタープリタ機能を使用して, 仕様アニメーションにより実行することができる. 実行をするためには, 関数名に「 pre_」を付けて, pre_sqrt (x) とすることで sqrt 関数の事前条件式を呼び出す. また, 関数名に「 post_」を付けて,

post_sqrt (x,res) とすることで sqrt 関数の事後条件式を呼び出す．VDMTools のインタープリタ機能を用いて事前条件と事後条件を評価するには, 以下のように行う．パラメータ (x) を 4 とし, 結果 (res) を 2 とした例を次に示す.

```
> print pre_sqrt(4)
true
> print post_sqrt(4, 2)
true
```

「 print 」 は 実行結果の出力を指示する VDMTools のコマンドである．実行結果は bool 型であり, 期待値とするテストデータは true である．実行結果が期待値と異なり false であった場合は, 入力のテストデータとして用意したパラメータ (x) の値もしくは結果 (res) の値が誤っているか, 実行した pre_sqrt, post_sqrt の記述が誤っているかのいずれかであるため, デバッグを行って誤りを修正する必要がある.

図 4.5 (a) に, 4.1.1 節において述べた宣言的な記述に対応する模式図を示す. 宣言的な記述において, 「伝えること」を目的とした仕様伝達部は, 次に示す, 事前条件 (構文例の preexpr) と事後条件 (構文例の postexpr) である.

```
pre x >= 0
post abs (res**2 - x ) <= EPS;
```

評価者が, 仕様アニメーションにおいて入力のテストデータとして用意するものは, 入力データであるパラメータ (x) と出力データである結果 (res) となる．図 4.5(a) の図中の preexpr と postexpr の実行結果は bool 型である．期待値とするテストデータは実行結果の true となる.

宣言的な記述を模式的に表した図 4.3 において「入力」と「入力時の内部状態」はパラメータ (x) に対応し, 「出力」と「出力時の内部状態」は結果 (res) に対応する．つまり, テストデータとして「入力」と「入力時の内部状態」, 「出力」と「出力時の内部状態」を用意する必要があることがこの例から分かる．そのため, 4.1.1 節において述べたように, 内部状態が複雑な仕様の場合, 評価者の作業工数が課題となることがある.

陰関数定義を用いることで, 入出力の関係を宣言的に記述することができるため, 「解決すべき問題」である What を記述することができる. したがって, この宣言的な記述箇所において, 操作的な記述を行わなければ, 「実行可能性の影響」を回避することができる. さらに, 前述の VDMTools のインタープリタ機能を用いて, 仕様アニメーションにより動作検証を行うことができる. しかし, 評価者は, 「入力時の内部状態」と「出力時の内部状態」をあらかじめ作成しておく必要があるため, 仕様アニメーションにより動かすときに, 評価者の作業工数が課題となる.

4.1.4 陽関数定義に関する考察

陽関数定義を用いることで, 図 4.2 に示した操作的な記述を行うことができる. 陽関数定義は, 事前条件 (pre) と, 関数本体から構成される. VDM++ の言語仕様として, 陽関数定義の構文例を以下に示す.

```
f : T1 * T2 * ... * Tn -> R1 * R2 * ... * Rm
f (a1, a2, ..., an) ==
  expr
pre preexpr (a1, a2, ..., an)
```

ここで, f は関数名, T1,...,Tn はパラメータの型, R1,...,Rm は戻り値の型, a1,...,an はパラメータ名であり, expr は関数本体を表している. preexpr は入力の前提条件を定義した事前条件である. ニュートン法を用いて平方根の求め方を記述した例を次に示す.

```
public sqrt : real -> real
sqrt (x) ==
  newton (x, x)
pre x >= 0;


private newton : real * real -> real
newton (x, t) ==
```

```
if abs (t ** 2 - x) <= EPS then
  t
else
  newton (x, (t + x/t)/2);
```

sqrt 関数は, 実数型 (real) のパラメータ (x) から実数型 (real) の結果を算出する. 算出方法を関数本体で記述している. 事前条件 (pre) は陰関数と同様である.

この仕様も VDMTools のインタープリタ機能を使用することで, 仕様アニメーションにより実行することができる. sqrt (x) を呼び出すことで, 関数の戻り値として結果を得ることができる. VDMTools のインタープリタ機能を用いて sqrt 関数を評価するには, 以下のように行う. 許容誤差である EPS を 0.001 としてパラメータ (x) を 4 とした例を次に示す.

```
> print exp_sqrt (4)
2.000000092922295
```

図 4.5 (b) に, 4.1.1 節において述べた操作的な記述に対応する模式図を示す. 「伝えること」を目的とした仕様伝達部は, 次の事前条件 (構文例の preexpr) である.

```
pre x >= 0
```

また, 次の関数本体 (構文例の expr) も仕様伝達部となる.

```
newton (x, x)
```

評価者が, 仕様アニメーションにおいて入力のテストデータとして用意するものは, 入力データであるパラメータ (x) となる. 図 4.5(b) の図中の preexpr と postexpr の実行結果は, 出力データとなる.

操作的な記述を模式的に表した図 4.2 において「入力」と「入力時の内部状態」はパラメータ (x) に対応し, 「出力」と「出力時の内部状態」は関数の戻り値に対応する. つまり, テストデータとして「入力」と「入力時の内部状態」を用意すればよい. 「出力」

と「出力時の内部状態」のうち, 期待値と比較して確認する範囲は, 評価対象の関数の重要度や複雑度, 内部状態を作成する作業工数など, 費用対効果を考慮して評価者が決めることができる.

陽関数定義を用いることにより, 4.1.1 節において述べた, 評価者の作業工数の課題を解決することができる. しかし, 図 4.2 の操作的な記述箇所に, 「解決すべき問題」である What を記述しようとしたときに, 入力から出力を作成する過程を記述した操作的な記述となる. このため, 「問題の解決策」である How を含んだ記述が, 「解決すべき問題」である What の記述の理解容易性を妨げるという, 「実行可能性の影響」が課題となる.

4.1.5 拡張陽関数定義に関する考察

拡張陽関数定義を用いることで, 図 4.4 に示した本研究において提案する記述スタイルを用いることができる. VDM++ の言語仕様として, 拡張陽関数定義の構文例を以下に示す.

```
f : T1 * T2 * ... * Tn -> R1 * R2 * ... * Rm
f (a1, a2, ..., an) ==
  expr
pre  preexpr (a1, a2, ..., an)
post postexpr (a1, a2, ..., an, RESULT.#1, RESULT.#2, ..., RESULT.#m)
```

拡張陽関数定義には, 陽関数定義に陰関数の postexpr が追加され, 戻り値の識別子として `RESULT.#1, RESULT.#2, ..., RESULT.#m` を用いていることが分かる. 戻り値が 1 つの場合は, `.#1` は不要である. 平方根の例を用いた具体例を次に示す.

```
public sqrt : real -> real
sqrt (x) ==
  newton (x, x)
pre x >= 0
```

```
post abs (RESULT**2 - x ) <= EPS;
```

VDMTools のインタープリタ機能を使用することで, 構文例において expr と示した関数本体と, 事前条件と事後条件を実行することができる. sqrt (x) を呼び出すことで, 関数の戻り値として結果を得ることができる. 許容誤差である EPS を 0.001 としてパラメータ (x) を 4 とした例を次に示す.

```
> print exp_sqrt (4)
2.000000092922295
```

図 4.5 (c) に, 4.1.2 節において述べた提案の記述スタイルに対応する模式図を示す. 「伝えること」を目的とした仕様伝達部は, 次に示す事前条件 (構文例の preexpr) と事後条件 (構文例の postexpr) である

```
pre x >= 0
post abs (res**2 - x ) <= EPS;
```

「伝えること」ではなく「動かすこと」を目的とした非伝達部は, 関数本体 (構文例の expr) となり, 次の式である.

```
newton (x, x)
```

拡張陽関数定義が, 陰関数定義や陽関数定義と異なる点は, 関数本体の実行結果である戻り値が RESULT に入り, 「 post_sqrt (4, RESULT) 」として, 事後条件を評価することができる点である. このため, 評価者が, 仕様アニメーションにおいて入力のテストデータとして用意するものは, 陽関数と同様に入力データであるパラメータ (x) となり, 陰関数の実行もできるようになる. 図 4.4 に表した, 本研究において提案する記述方法において「入力」と「入力時の内部状態」はパラメータ (x) に対応し, 「出力」と「出力時の内部状態」は関数の戻り値に対応する. つまり, 陽関数定義と同様に, テストデータとして「入力」と「入力時の内部状態」を用意すればよく, 4.1.1 節において述べた評価者の作業工数の課題を解決することができる.

拡張陽関数定義を用いることで, 仕様伝達部と非伝達部を分離して, 仕様伝達部において入出力の関係を宣言的に記述することができるため, 「解決すべき問題」である What を記述することができる. したがって, この宣言的な記述箇所において, 操作的な記述を行わなければ, 「実行可能性の影響」を回避することができる.

次の 4.2 節では, 2 つ目の研究課題である, 仕様と設計の分離の課題を解決する形式仕様記述の設計・構成法を提案する. その次の 4.3 節において, 本節において述べた, 記述スタイルの提案手法について, 具体的な仕様記述例を用いて評価する.

4.2 データ構造の隠蔽による仕様と設計の分離方法の提案

本節では, 本章において扱う 2 つ目の課題である, 仕様と設計の分離に関する課題を解決する形式仕様記述の設計・構成法を提案する.

2.1.2 節で述べたように, 仕様と設計の分離とは, 「解決すべき問題」を定義した仕様と, 「問題の解決方法」を定義した設計を分けて考えるということである. 一般に, 仕様策定工程では, 何を作るべきかという「解決すべき問題」を検討し, 設計工程では, 「問題の解決方法」を検討することで, 仕様策定者と設計者が分業して組織的に開発を行うことができると言われている. 仕様と設計の分離は広く言われていることではあるが, 実際に, 仕様と設計を分離することは容易ではない. 仕様と設計の境界が曖昧であるために, 形式仕様記述から仕様として規定している範囲と, 形式仕様記述には記述してはあるが, 設計者が自由に規定できる範囲を, 設計者が読み取ることが難しいという課題がある.

本研究では, 仕様と設計の分離を行うために, 開発対象の内部状態を表す内部データ構造に着目した. 形式仕様記述言語には, 集合や写像などの内部データ構造を抽象化するための型が用意されている. しかし, 集合や写像を用いて内部データ構造を表現したとしても, 型の特性から設計者に仕様と設計の責任範囲を伝えることはできない. 以下に, この課題に関する具体的な例を示す. 例えば, 名前 (NAME 型) とアドレス (ADDRESS 型) からなるアドレス帳 (BOOK 型) について考える. 名前 (NAME 型) とアドレス (ADDRESS 型) を次のように定義する.

```
NAME = seq of char;
ADDRESS = seq of char;
```

ここで, 名前 (NAME 型) とアドレス (ADDRESS 型) は仕様策定者が仕様として規定している型とし, アドレス帳 (BOOK 型) の保持方法は設計者が規定すべきことと仮定する. つまり, 仕様と設計が分離された仕様記述であるためには, 上記の, 仕様として規定している型と, 設計者が規定すべき型の仮定が, 仕様記述から読み取れなければならない. 実行可能仕様を記述するためには, アドレス帳 (BOOK 型) の型を決める必要がある. BOOK 型を, 内部データ構造を抽象化するために用意された集合と写像の型を用いて規定した例を示す. 写像を用いた場合を BOOK1 型とし, 集合を用いた場合を BOOK2 型とすると, 次のような記述になる.

```
BOOK1= map NAME to ADDRESS;
BOOK2= set of  CARD;
```

ここで, CARD 型はレコード型である. レコード型は複数の型をまとめて管理することができる. CARD 型の型定義を次に示す.

```
CARD ::
  Name : NAME
  Address : ADDRESS;
```

この型の値は、レコード型の構成子を用いて次の表に定義する。

```
mk_CARD ("name", "name@a.co.jp")
```

このようにして, Name に “name” という文字列を保持し, Address に “name@a.co.jp” という文字列を保持した CARD 型の値を定義することができる. ここでは, CARD 型も BOOK 型と同様に, 形式仕様記述には記述してあるが, 設計者が自由に決めることができる型とする.

次に, 名前とアドレスをアドレス帳 (BOOK 型) へ新規に登録する仕様を考える. 前述の BOOK1 型であっても BOOK2 型であっても, 新規に名前とアドレスを登録する仕様

を記述することができる. 事後条件は, 登録後のアドレス帳 (BOOK1 型, BOOK2 型) に指定した登録対象の名前とアドレスが存在していることとした. 事前条件として, 既に同一の名前 (NAME 型) がアドレス帳に登録されていないこととした.

写像を用いて定義した BOOK1 型を用いると次のような記述になる.

```
public 新規カード登録 1 : BOOK1 * NAME * ADDRESS -> BOOK1
新規カード登録 1 (bk, name, addr) ==
  bk munion {name |-> addr}
pre name not in set dom bk
post RESULT = bk munion {name |-> addr};
```

次に, CARD 型の集合を用いた BOOK2 型を用いた場合, 次のような記述となる.

```
public 新規カード登録 2 : BOOK2 * NAME * ADDRESS -> BOOK2
新規カード登録 2 (bk, name, addr) ==
  bk union {mk_CARD (name, addr)}
pre forall c in set bk & c.Name <> name
post mk_CARD (name, addr) in set RESULT;
```

ここでは, いずれの型定義であっても, 仕様が記述可能であることを示すまでにとどめておいて, 詳細な VDM++ の言語仕様の説明は省略する. BOOK1 型を用いた場合も, BOOK2 型を用いた場合も, 事前条件と事後条件において記述している内容は同じである. しかし, 設計者はどちらの表現であっても, NAME 型と ADDRESS 型は仕様において規定する型であることを読み取ることができない. 同様に, CARD 型と BOOK 型は設計者が自由に定義することが許された型であることを読み取ることができない.

この課題を解決するために, 本論文では, 仕様と設計の境界部分として内部データ構造に着目し, 仕様として規定しない型を隠蔽することで, 仕様策定者が仕様として規定する型と, 仕様策定者が仕様としては規定せずに, 設計者の責任範囲として規定すべき型の違いを明示する方法を提案する. 型を隠蔽した場合と, 隠蔽しない場合を比較し, 可読性の違いを比較する.

型を隠蔽するために, 以下のような型の仕様記述のルールを定めた. まず, 型を 2 つに分類した. 仕様で定義する「仕様の型」と, 仕様では定義しない「隠蔽対象の型」である. 「仕様の型」とは, 工程が仕様策定と設計工程とに分かれる中で, 仕様策定者が仕様として規定する型である. そのため, 設計者はこの型の定義に従うべき型である. この型は, 仕様策定者の責任範囲とする型である. 一方, 「隠蔽対象の型」とは, 設計に自由度を持たせるために, 仕様策定者が仕様策定工程では規定しない型である. この型は, 設計工程で設計者の責任範囲として規定すべき型である. 次に, 「隠蔽対象の型」の定義・演算子を隠蔽するための関数を導入した. この関数を「隠蔽関数」と呼ぶこととする. 図 4.4 の仕様伝達部から「隠蔽対象の型」の値を使用する場合は, 必ず「隠蔽関数」を経由するルールとした. このルールにより, 「隠蔽対象の型」の型の定義と型演算子を仕様伝達部から直接参照する記述を排除することができた. また, 「仕様の型」と「隠蔽対象の型」を区別する型の命名ルールを定めることで, 設計者をはじめとする仕様記述の読み手が「仕様の型」と「隠蔽対象の型」を区別できるようにした. 4.3 節において, 具体的な記述例を用いて説明し, 提案手法の評価を行う.

4.3　本提案手法を用いた具体的な記述例による評価

名前とメールアドレスからなるカードを管理するアドレス帳の具体例を用いて, これまで述べた 2 つの手法を評価する.

1 つ目の評価は, 4.1 節において述べた, 「実行可能性の影響」の課題を解決する仕様記述スタイルの提案に関する評価である. 図 4.4 において示した, 提案の仕様記述スタイルを用いて, 具体的な仕様記述例を示す. まず, この記述例を用いて, 宣言的な記述により「実行可能性の影響」の課題を解決できることを示す. 次に, 仕様伝達部と非伝達部を分離することで, 理解容易性に優れた仕様を記述することができることを示す. 最後に, VDM++ によるテストスクリプトを具体的に示す. この具体例を用いて, 4.1.1 節において述べた, 仕様アニメーションによる動作検証時に課題となる, 評価者の作業工数の課題を具体的に示す. そして, 提案する記述スタイルを用いて, この課題が解決できるこ

と示す.

2 つ目の評価は, 4.2 節において述べた, 仕様と設計の分離に関する課題を解決する形式仕様記述の設計・構成法の提案に関する評価である. 評価において, 提案の手法を用いた場合と, 用いなかった場合の記述例を比較し, 本提案手法の有効性を示す.

以降の本節では, 具体例として, **図 4.6** に示したアドレス帳の例を用いる. この記述例の詳細は, 各節の中で述べる.

4.3.1 「実行可能性の影響」を解決する仕様記述スタイルの評価

まず, 図 4.6 に示した具体例について, 「新規カード登録」関数を例に, 「実行可能性の影響」の課題を解決する仕様記述スタイルについて議論する. 「新規カード登録」関数を次に示す.

```
public 新規カード登録 : BOOK * CARD -> BOOK
新規カード登録 (bk, cd) ==
  impl_新規カード登録 (bk, cd)
pre cd.Name not in set 名前集合 (bk)
post カード集合 (RESULT) = カード集合 (bk) union {cd};
```

「新規カード登録」関数は, 引数としてアドレス帳を表す BOOK 型, および名前とアドレスからなる登録対象の CARD 型を受け取り, 戻り値として登録後の BOOK 型を返す. 「新規カード登録」関数の事前条件 (pre) は次の条件式である.

```
pre cd.Name not in set 名前集合 (bk)
```

これは, 登録対象のカードが登録済みではないことを示している. 以降では, この事前条件を「 pre_新規カード登録」と表記する. 事後条件 (post) は次の条件式である.

```
post カード集合 (RESULT) = カード集合 (bk) union {cd};
```

これは, 次のように読むことができる. 登録後のアドレス帳のカード集合は, 登録前のアドレス帳のカード集合に, 登録対象のカードを加えたものとなっていること, という条件

```
types
  public NAME = seq of char;            -- 仕様の型
  public ADDRESS = seq of char;         -- 仕様の型
  public CARD ::                        -- 仕様の型
    Name : NAME
    Address : ADDRESS;
  public BOOK = BOOK_IMPL;              -- 隠蔽対象の型
  public BOOK_IMPL = map NAME to CARD;  -- 非伝達部の型

functions
  -- 拡張陽関数定義による機能仕様の記述
  public 新規カード登録 : BOOK * CARD -> BOOK
  新規カード登録 (bk, cd) ==
    impl_新規カード登録 (bk, cd)        -- 非伝達部
  pre cd.Name not in set 名前集合 (bk)  -- 仕様伝達部
  post カード集合 (RESULT)
          = カード集合 (bk) union {cd}; -- 仕様伝達部

  -- 「隠蔽対象の型」BOOK の「隠蔽関数」
  public カード集合 : BOOK -> set of CARD
  カード集合 (bk) ==
    rng bk;
  public 名前集合 : BOOK -> set of NAME
  名前集合 (bk) ==
    dom bk;

  -- 新規カード登録の非伝達部
  private impl_新規カード登録 : BOOK * CARD -> BOOK
  impl_新規カード登録 (bk, cd) ==
    bk munion {cd.Name |-> cd}
```

図 4.6　本研究において提案する方法を用いた仕様記述の例

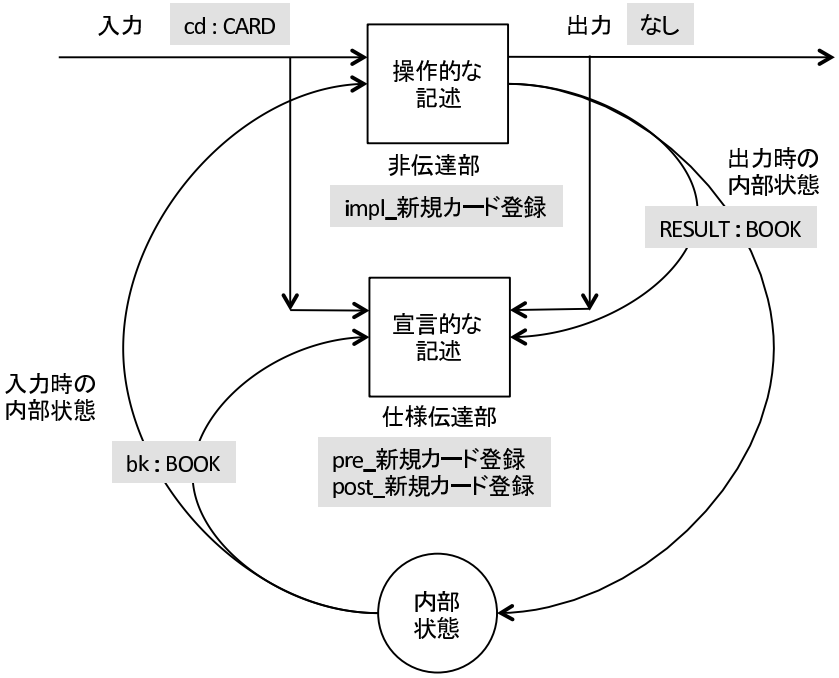

図 4.7 「実行可能性の影響」を解決する仕様記述スタイルと仕様記述例との対応関係

式である. 以降では, この事後条件を「 post_新規カード登録」と表記する. 「新規カード登録」関数の関数本体は次の記述である.

```
impl_新規カード登録（bk, cd）
```

以降では, この関数本体を「 impl_新規カード登録」と表記する.

「新規カード登録」関数は, 本研究において提案した図 4.4 の記述スタイルを用いた. 提案の記述スタイルと「新規カード登録」関数との対応関係を**図 4.7** に示す. 「入力」は, 「新規カード登録」関数の引数である「 cd : CARD 」である. 図中では, 関数内の該当箇所を指し示すために「変数名 : 型」という表記を用いることとする. 「入力時の内部状態」も関数の引数である「 bk : BOOK 」となる. 非伝達部は「 impl_新規カード登録」となる. 非伝達部であることを明確にするために, implementation を表す「 impl_」を関数の先頭に付ける命名ルールとした. 仕様伝達部は, 事前条件と事後条件の「 pre_新規カード登録」と「 post_新規カード登録」となる. 仕様伝達部では, 「出力」と「出力時の内部状態」を使用して, 入出力の関係を宣言的に記述することができる.

以上から,「実行可能性の影響」を回避することができる.「実行可能性の影響」とは「解決すべき問題」である What を記述するべき仕様に,「問題の解決方法」である How の要素を多く含んだ記述が影響を及ぼし, 設計者を過剰に制約し, 仕様が複雑になるというものであった. 宣言的な記述により,「問題の解決方法」である How の要素を含まない仕様を記述することができる. また,「動かすこと」を目的とした記述を, 図 4.7 に示した非伝達部にのみ記述し, 非伝達部のみから呼び出す関数には, implementation を表す「 impl_」を関数名の先頭に付ける命名ルールとした. これにより,「動かすこと」を目的にした記述と,「伝えること」を目的にした記述を区別し, 理解容易性に優れた仕様を記述することができることを示した.

次に, 仕様アニメーションによる動作検証時に課題となる, 評価者の作業工数の課題について, 図 4.4 において提案した記述スタイルを用いて, この課題が解決できること示す. まず, 図 4.3 の記述スタイルの場合に, VDM++ を用いたテストスクリプトの記述から, 評価者の作業工数が課題となる具体例を示す. 次に, 既に示した「新規カード登録」関数と図 4.7 に関するテストスクリプトの記述から, この課題が解決できることを示す.

評価者の作業工数が課題となるケースは, 図 4.3 に示した宣言的な記述を行った場合であった. この宣言的な記述スタイルは, 前述の「新規カード登録」関数の事後条件「 post_新規カード登録」の記述に対応している. 対応関係を**図 4.8** に示す. 4.1.3 節において述べたように, 関数の事後条件は VDMTools のインタープリタ機能を使用して, 次のように呼び出すことができる.

```
post_新規カード登録（bk, cd, result_bk）
```

ここで, bk は実行前のアドレス帳を表す BOOK 型, cd は登録対象のカードを表す CARD 型, result_bk は「新規カード登録」関数の戻り値の BOOK 型である. この「 post_新規カード登録」関数に対するテストスクリプトの記述から, 評価者の作業工数が課題となる具体例を示す.

VDM++ を用いて記述したテストスクリプトを示す. まず,「 post_新規カード登録」関数のテストを行うために, テスト対象のアドレス帳である BOOK 型を作成するための

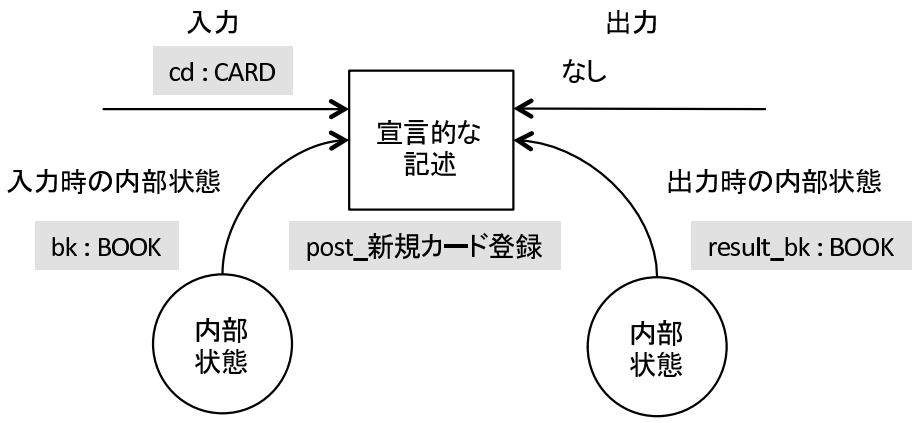

図 4.8 宣言的な記述スタイルと仕様記述例との対応関係

関数は次のようになる.

```
public MakeAddressBook : () -> BOOK
  MakeAddressBook () ==
    let
      a = mk_CARD ("a", "a@a.a"),
      b = mk_CARD ("b", "b@a.a"),
      c = mk_CARD ("c", "c@a.a"),
      d = mk_CARD ("d", "d@a.a"),
      e = mk_CARD ("e", "e@a.a"),
      f = mk_CARD ("f", "f@a.a")
    in{
      a.Name |-> a,
      b.Name |-> b,
      c.Name |-> c,
      d.Name |-> d,
      e.Name |-> e,
      f.Name |-> f
    };
```

MakeAddressBook 関数では, 名前が a から f で, アドレスが a@a.a から f@a.a の 6 人のカードを保持したアドレス帳を作成している. 次に, テストスクリプトの一部を取り出すと以下のようになる.

```
let
  address_book  = MakeAddressBook (),
  new_card      = mk_CARD ("new_name", "new@address.test"),
  expected_book = address_book munion {new_card.Name |-> new_card}
in
  post_新規カード登録 (address_book, new_card, expected_book) = true;
```

address_book は, 前述の MakeAddressBook 関数により作成されたテスト対象とするアドレス帳の値を保持し, new_card は, 新規に登録するカードの値を保持している. expected_book は, 写像の munion (併合) 演算子を用いて, テスト実行前の address_book に new_card を登録した値を保持している. let 式中で作成した変数を用いて, 「post_新規カード登録」関数のテストを in 式中で実施している. このように, 図 4.3 に示した宣言的な記述を評価するためには, 新規カード登録後のアドレス帳のテストデータ (expected_book) が必要である. このデータを作るために, 「新規カード登録」関数で実施すべきことと同様の処理をテストスクリプト側で実施している. つまり, あらかじめ登録後のアドレス帳を作成しておかなければ, 宣言的な記述である事後条件を評価することができない. このため, 宣言的な記述は, 評価者の作業工数が課題となることが多い. 内部状態の構造として, BOOK 型は簡単な例であったが, 内部状態が複雑な仕様において特に課題となることが多い.

次に, 既に示した「新規カード登録」関数と図 4.7 に関する, テストスクリプトの記述から, この課題が解決できることを示す.

テストスクリプトの一部を取り出すと以下のようになる.

```
let
  address_book       = MakeAddressBook (),
```

```
    new_card           = mk_CARD ("new_name", "new@address.test"),
    actual_result_book = 新規カード登録 (address_book, new_card)
  in
    actual_result_book (new_card.Name) = new_card;
```

let 式中の address_book と new_card は前述の例と同様である. actual_result_book にはテスト対象の「新規カード登録」関数を実行した結果のアドレス帳の値が入る. in 式内では, 次のように実行結果と期待値を比較している. 「新規カード登録」関数の実行結果を格納した actual_result_book から, 登録時に指定した名前 (new_card.Name) をキーにしてカードを取得している. そして, 取得したカードと期待値である new_card が一致することを確認している. 期待値の確認範囲として, 変化した範囲のみを確認する場合は, 期待値として作成するテスト用のデータは new_card のみでよい. 一方, 全ての変化しない範囲も含めて確認する場合は, 次に示すように, 期待値 (expected_book) を作成し, in 式内で実行結果 (actual_result_book) と期待値 (expected_book) の比較を行うことができる.

```
  let
    address_book       = MakeAddressBook (),
    new_card           = mk_CARD ("new_name", "new@address.test"),
    actual_result_book = 新規カード登録 (address_book, new_card)
    expected_book = address_book munion {new_card.Name |-> new_card}
  in
    actual_result_book = expected_book;
```

ここで示した expected_book は, 前述の宣言的な記述のテストスクリプトにおいて示した方法と同様である. 図 4.3 の記述スタイルを用いた宣言的な記述スタイルの場合は, 常に新規カード登録後の実行結果であるアドレス帳のデータ (expected_book) が必要であった.

このように, 提案の記述スタイルを用いた場合は, 期待値として作成する実行結果について, 変化した範囲だけとするべきか, 変化しない範囲も含めた全てとするべきかを評価者が選択することができる. この選択は, 検証対象としている関数の重要度や複雑度, 内部状態を作成する作業工数など, 評価者が費用対効果を考慮して決めることができる. 一方, 図 4.3 に示した宣言的な記述の場合は, 常に変化しない範囲も含めた全ての実行結果の期待値が必要となり, 評価者が費用対効果を考慮して決めることができない点において課題となる.

簡単な記述例ではあるが, 図 4.4 において提案した記述スタイルを拡張陽関数定義に適用した具体例を示した. まず, 具体例の中で, 事前条件と事後条件を仕様伝達部として, 宣言的な記述を行うことで「実行可能性の影響」を回避した. また, 「 impl_」のような命名ルールを定めることで, 仕様伝達部と非伝達部を明確に分離できることを示した. 次に, 仕様アニメーションによる動作検証時に課題となる, 評価者の作業工数の課題を, テストスクリプトから具体的に示した. これにより, 提案した記述スタイルを用いることで, 評価者の作業工数の課題が解決できることを示した.

4.3.2　仕様と設計の分離に関する課題を解決する形式仕様記述の評価

本節において, 4.2 節で述べた, 仕様と設計の分離に関する課題を解決する仕様記述の設計・構成法を評価する. 評価において, 提案の手法を用いた場合と, 用いなかった場合の記述例を比較し, 本提案手法の有効性を示す.

図 4.6 のアドレス帳の仕様記述例において, 次の 5 つの型を定義した.

```
public NAME = seq of char;
public ADDRESS = seq of char;
public CARD ::
  Name : NAME
  Address : ADDRESS;
public BOOK = BOOK_IMPL;
```

```
public BOOK_IMPL = map NAME to CARD;
```

まず, 仕様記述に用いる型を「仕様の型」と「隠蔽対象の型」とに分類する. ここでは, NAME 型, ADDRESS 型, CARD 型を仕様において規定する「仕様の型」とした. BOOK 型を仕様では規定しない内部データ構造を表す「隠蔽対象の型」と仮定する. BOOK 型を BOOK_IMPL 型として再定義する型の定義ルールによって, BOOK 型は「隠蔽対象の型」であることを明示した. 「_IMPL 」を型名の後ろに付けることで, 再定義した BOOK 型は「隠蔽対象の型」であることを示した. これにより, 形式仕様には記述してあるが, 設計に自由度を持たせるために仕様策定者が仕様策定工程では規定しない型であるため, 設計工程で設計者の責任範囲として規定すべき型であることを示すことができた.

次に, 「隠蔽対象の型」を「隠蔽関数」を定義することにより, 型の定義, 演算子を隠蔽する. BOOK 型を隠蔽する「隠蔽関数」を次に示す.

```
public カード集合 : BOOK -> set of CARD
カード集合 (bk) ==
  rng bk;
public 名前集合 : BOOK -> set of NAME
名前集合 (bk) ==
  dom bk;
```

このように, 「カード集合」「名前集合」といった BOOK 型を操作する「隠蔽関数」を用意した. 仕様伝達部では「隠蔽関数」を用いて BOOK 型の値を使用することで, BOOK 型の型定義と型演算子を隠蔽した. このため, 形式仕様の読者が仕様伝達部を読んでいる限りは, 「隠蔽対象の型」の型定義, 型演算子を参照することはない.

最後に, 仕様の理解容易性を評価する. 以下に, 型の隠蔽を行わない記述と, 型の隠蔽を行った記述例を示す.

```
-- 型の隠蔽を行わない記述
public 新規カード登録 : BOOK * CARD -> BOOK
新規カード登録 (bk, cd) ==
```

```
  impl_新規カード登録 (bk, cd)
pre cd.Name not in set dom bk
post rng RESULT = rng bk union {cd};

-- 型の隠蔽を行った記述
public 新規カード登録 : BOOK * CARD -> BOOK
新規カード登録 (bk, cd) ==
  impl_新規カード登録 (bk, cd)
pre cd.Name not in set 名前集合 (bk)
post カード集合 (RESULT) = カード集合 (bk) union {cd};
```

型の隠蔽を行わない記述では, 事前条件 (pre) と事後条件 (post) の仕様伝達部において, 写像の型演算子を用いて仕様を記述した. この型の隠蔽を行わない記述と, 行った記述を比較することで, 内部データ構造の隠蔽が, 仕様と設計の規定範囲の観点で理解容易性を向上させる効果があることが分かる. また, 3.3 節において述べた「実装の影響」の観点で考察する. 型を隠蔽する隠蔽関数名として, 「カード集合」や「名前集合」といった, Jackson が述べた機械の内部の状態ではなく, 適用領域の用語を用いることができた. このため, ドメインエンジニアや仕様策定者から見ても, 読みやすい仕様となっていることが分かる.

4.4　本提案手法の FeliCa IC チップ開発への適用の結果と考察

4.3 節で述べた具体例は, 説明のための簡単な例であるため, 実際の開発現場における有効性を示すには十分ではない. 筆者らは, 本論文で提案した手法を FeliCa IC チップ開発プロジェクトにおいて適用した. FeliCa IC チップ開発プロジェクトのうち, 次に示す 2 つの製品開発プロジェクトにおいて形式仕様記述手法を適用した. 2004 年から 2007

```
functions
  public コマンド実行 : PACKET_DATA * FILE_SYSTEM * STATE
                        -> PACKET_DATA * FILE_SYSTEM * STATE
  コマンド実行 (cmnd_pkt, fs, state) ==
    is subclass responsibility
  pre
  post
    コマンド仕様 (cmnd_pkt, fs, state, RESULT.#1, RESULT.#2, RESULT.#3);

  public コマンド仕様 : PACKET_DATA * FILE_SYSTEM * STATE *
                        PACKET_DATA * FILE_SYSTEM * STATE -> bool
  コマンド仕様(cmnd_pkt, old_fs, old_state, res_pkt, new_fs, new_state) ==
    is subclass responsibility;
```

図 4.9 各コマンドの仕様を記述するテンプレートの記述例

年までフェリカネットワークス株式会社にてモバイル FeliCa IC チップ開発プロジェクトに形式仕様記述手法を用い, その後, 2007 年中頃から約 4 年間, ソニー株式会社にて FeliCa IC チップ開発に形式仕様記述手法を適用した. 前者を第 1 世代の適用, 後者を第 2 世代の適用と呼ぶ. 本提案手法は, 第 1 世代の適用において課題となったため, 第 1 世代の適用が終了した後に提案手法の検討を行ったものである. そして, 第 2 世代の適用において実際に用いた.

第 2 世代の適用において, 提案手法を用いた記述例として, コマンド機能の仕様記述を示す. コマンド機能は, 仕様記述全体の約 7 割を占め, 主要な仕様記述モジュールとなっている. コマンド機能は数十のコマンド群からなり, 各コマンドの仕様を記述するために, テンプレートが必要であった. この各コマンドの仕様を記述するためのテンプレートを**図 4.9** に示す. テンプレートは, 「コマンド実行」関数と「コマンド仕様」関数からなり, 図 4.4 に示した本章において提案した記述スタイルを用いている. また, 図 4.5 (c) に示した拡

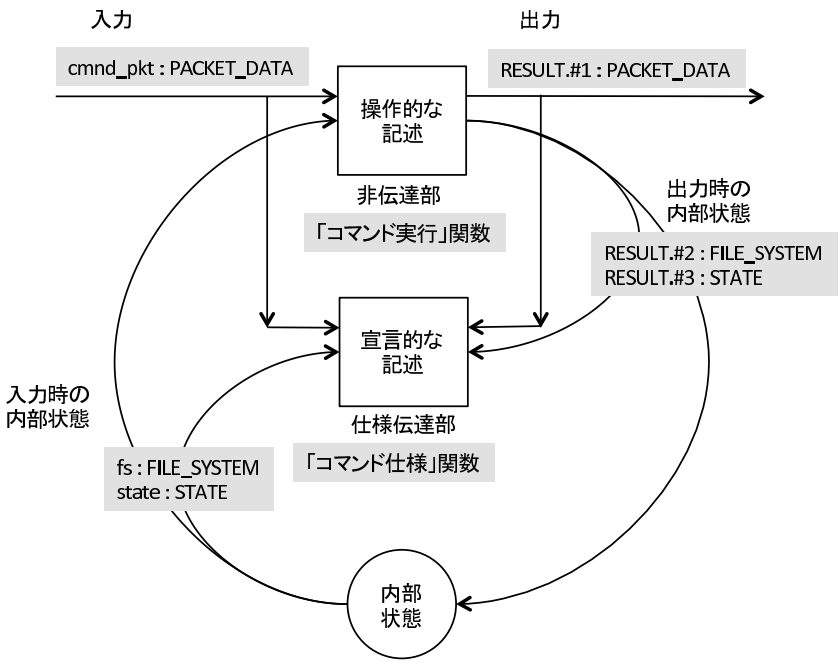

図 4.10　提案の仕様記述スタイルと仕様記述例との対応関係

張陽関数定義を使用している．「コマンド実行」関数の is subclass responsibility となっている箇所の内容が非伝達部である．「コマンド仕様」関数の is subclass responsibility の箇所の内容が仕様伝達部となる．これらの is subclass responsibility の内容は，このクラスを継承した子クラスにおいて記述した．

図 4.9 のテンプレートの記述例と，図 4.4 において示した提案手法との対応関係を，**図 4.10** に示す．対応関係について，図 4.9 と図 4.10 を用いて説明する．図 4.10 に示すように，「コマンド実行」関数は非伝達部となり，「コマンド仕様」関数は仕様伝達部となる．「コマンド実行」関数の引数は，FeliCa IC チップへの「入力」である PACKET_DATA 型と，「入力時の内部状態」を定義した FILE_SYSTEM 型と STATE 型からなる．「コマンド実行」関数の戻り値は，FeliCa IC チップの「出力」である PACKET_DATA 型と，「出力時の内部状態」を定義した FILE_SYSTEM 型と STATE 型からなる．「コマンド仕様」関数の引数は，「コマンド実行」関数の引数と戻り値 (RESULT) とした．RESULT.#1 は 1 つ目の「コマンド実行」関数の戻り値，つまり PACKET_DATA 型の戻り値である．同様に RESULT.#2 と RESULT.#3 は 2 つ目と 3 つ目の「コマンド実行」関数の戻

り値として示した FILE_SYSTEM 型と STATE 型である．また, FILE_SYSTEM 型と STATE 型は, 4.2 節で述べた「隠蔽対象の型」とした.

このように,「コマンド仕様」関数と「コマンド実行」関数を用いて, 仕様伝達部と非伝達部を分離し, 仕様伝達部において, 入出力の関係を宣言的に記述することができた. 宣言的な記述により,「解決すべき問題」である What を記述することができ,「実行可能性の影響」を回避することができた．さらに, 仕様アニメーションにより動作検証を行う場合, 非伝達部において「出力」と「出力時の内部状態」を作成することで, 評価者の作業工数の課題を解決することができた．FILE_SYSTEM 型と STATE 型を「隠蔽対象の型」として, 隠蔽関数を用い, 設計において定義するデータ構造を隠蔽した．これにより, 仕様において定義する「仕様の型」と区別し, 仕様と設計を分離することができた.

表 4.1 に, VDM++ で記述したソースコードの有効行数をモジュール単位で示す．ここでいう有効行数とは, 総行数からコメント行と空行を除いた行数である．コマンド機能は, 記述例として示したテンプレートの中身を記述したものである．これは, 全体の約 7 割を占める主要なモジュールである．ファイルシステム機能と状態管理機能は, 内部データ構造を含む機能仕様である．フレームワークとは仕様を記述する枠組みを提供し, 図 4.9 に示したコマンド仕様を記述するテンプレートもフレームワークの一部である．汎用ライブラリは, ビット演算などの汎用的なライブラリである.

フレームワークと汎用ライブラリを除く全ての機能仕様は, 図 4.5(c) の拡張陽関数定義を用い, 図 4.4 に示した本研究において提案した記述スタイルを用いた．フレームワークと汎用ライブラリは, 図 4.5(b) の陽関数定義を用い, 図 4.7 に示した操作的な記述スタイルを用いた.

図 4.4 に示した本研究において提案した記述スタイルの課題は, 宣言的な記述と操作的な記述という 2 つの記述が必要な点である．例えば, 図 4.7 に示した「新規カード登録」の例を用いて説明する．操作的な記述箇所である「 impl_新規カード登録」では, アドレス帳にカードを「登録する」というアドレス帳への操作を記述し, 宣言的な記述箇所である「 post_新規カード登録」では, アドレス帳にカードが「存在する」という条件を記述している．これは, 仕様を代入として記述するか, 条件式として記述するかの違いであ

表 4.1　適用対象の規模 (有効行数)

モジュール名	有効行数
コマンド機能	11,460
ファイルシステム機能	1,716
状態管理機能	428
セキュリティ機能	1,120
フレームワーク	425
汎用ライブラリ	648
合計	15,797

る. 仕様記述の観点では, この違いの重要性を考慮する必要がある. なぜなら, この違いが仕様と設計の違いであるからである. つまり, 「解決すべき問題」である What の記述と, 「問題の解決方法」である How の記述の違いであるからである. 一方で, 実際に提案手法を適用するときにおいて, 仕様記述の観点では重要な違いかもしれないが, 「同じような」記述を 2 箇所において記述する必要があるため, 作業効率の観点から課題となった. 図 4.2 の操作的な記述と, 図 4.3 の宣言的な記述と, 図 4.6 の提案手法の記述を比較すると, 提案手法の記述量が最も多くなる. 提案手法において, 操作的な記述箇所と宣言的な記述箇所について, 記述の共通化を行わなかった場合, 提案手法の記述量は, 操作的な記述量と宣言的な記述量の総和となる. 筆者らの適用において, 操作的な記述箇所と宣言的な記述箇所の共通部分を抜き出して関数化することで, 記述量の増加を抑えた. **表 4.2** にコマンド機能モジュールの仕様伝達部と非伝達部の構成比を示す. 共通部が, 操作的な記述箇所と宣言的な記述箇所において, 共通部分を関数化した部分である. この表において示すように, 67%のソースコードを共通化した.

実際に適用するときの課題として, 仕様記述者が陰関数を用いて宣言的に仕様を記述するには, 陰関数の記述経験が必要である点がある. 拡張陽関数定義の記述スタイルであれば, 陰関数と陽関数を同時に書きながらテストを行い, 宣言的な仕様を完成させることができる. 最初に陽関数の記述を行い, 開発対象の仕様と, 形式仕様記述手法に対する理

表 4.2 仕様伝達部と非伝達部の構成 (有効行数)

コマンド機能モジュールの構成	有効行数	比率
共通部	7,680	67%
仕様伝達部	1,691	15%
非伝達部	2,089	18%
合計	11,460	

解を深め, その後, 仕様アニメーションにより仕様を動作させることで, 動作検証を行いながら陰関数を記述することができる. 本提案手法の適用において, 仕様記述の経験を積みながら徐々に記述スタイルを改善していくことが重要である.

4.5 仕様と設計を分離する実行可能仕様のまとめ

本章では, 2 章と 3 章において述べた研究課題のうち 2 つの課題に対する解決方法を提案し, 提案方法の評価を行った. 1 つ目の課題は, 「伝えること」と「動かすこと」を目的とした実行可能仕様の課題であった. この課題は, 2.1.1 節において考察し, 3.2 節において Hayes と Jones の指摘する「実行可能性の影響」の関連研究から議論したものである. 2 つ目の課題は, 仕様と設計の分離に関する課題であった. この課題は, 2.1.2 節において考察し, 3.3 節において Jackson の「実装の影響」の関連研究から議論したものである.

1 つ目の「実行可能性の影響」に関する課題について, この課題を解決する仕様記述スタイルを提案した. まず, 次の 2 つの観点から仕様記述スタイルを議論した. 操作的な記述と宣言的な記述の観点と, 仕様アニメーションによる動作検証の作業工数の観点である. この中で, 仕様伝達部と非伝達部からなる仕様記述スタイルを提案した. 次に, 具体的な仕様記述例を用いて, 提案の仕様記述スタイルを評価し, 次の 3 つのことを示した.

- 「解決すべき問題」である What を記述するために, 仕様を宣言的な記述箇所に記述することができること.

- 仕様伝達部と非伝達部を分離することで, 理解容易性に優れた仕様を記述することができること.
- 仕様アニメーションによる動作検証時に課題となる, 評価工数の課題が解決できること.

2 つ目の仕様と設計の分離に関する課題について, 仕様において定義するデータ構造と設計において定義するデータ構造に着目し, この課題を解決する形式仕様記述の設計・構成法を提案した. 仕様策定者が決める「仕様の型」と, 設計者が決める, 「隠蔽対象の型」に分類し, 「隠蔽対象の型」を「隠蔽関数」により隠蔽する方法を提案した. この手法の評価において, 提案の手法を用いた場合と, 用いなかった場合の記述例を比較し, 本提案手法の有効性を示した.

最後に, 提案した手法を FeliCa IC チップ開発プロジェクトにおいて適用した結果について述べ, 適用における知見を考察した.

第5章

レビューとテストの用途を考慮した仕様記述フレームワークの提案

1章において, Parnas が指摘した, フォーマルメソッドがありふれた「当然の技術」として産業界に浸透していないという課題を述べた. 本研究が扱うフォーマルメソッドの適用の段階は, 証明などは行わないで, 形式仕様記述を行うレベル 0 の段階である. このレベル 0 の段階を, レベル 0-a とレベル 0-b に分けることで, Parnas が指摘した課題を分析した. レベル 0-a は, 形式仕様の記述者が, 形式仕様記述手法を用いて仕様を厳密に記述し, 早期に仕様の曖昧さに起因する不具合を見つけることができる適用の段階とした. レベル 0-b は, 形式仕様の記述者以外である, ドメインエンジニアや設計者, 実装者, 評価者が, 形式仕様記述を参照し, 形式仕様記述に基づいて設計と実装とテストを行う適用の段階とした. 形式仕様を参照する人は, レベル 0-a では形式仕様の記述者に限定されるが, レベル 0-b ではドメインエンジニア, 設計者, 実装者, 評価者といった, ほぼ全ての工程の人となる. これらの人たちが, 形式仕様記述手法をあらかじめ習得していることは少なく, 形式仕様記述手法の専門家ではない人が, 形式仕様記述を読み, 理解し, 活用していく必要がある. つまり, これらの人たちが, 読みやすいと感じ, 形式仕様記述手法を利用することにメリットを感じることが重要であり, そのような, 形式仕様を記述するための技術と, 活用するための技術が必要である. この課題を解決することで, 形式仕様

記述手法がありふれた「当然の技術」として, 開発プロジェクトにおいて広く活用されるようになり, さらには, 産業界においても広く活用されることにつながると考える.

本研究では, 前述のレベル 0-b の段階に到達するために必要な, 形式仕様を記述するための技術と, 活用するための技術を研究課題として設定した. 記述するための技術として, ドメインエンジニア, 設計者, 実装者, 評価者といった様々な役割を担った開発者にとって, 理解容易性に優れた仕様を記述する技術が必要である. 活用するための技術としては, 従来からある既に開発現場に浸透しているレビューやテスト手法といった検証技術を主体として, その中で形式仕様記述手法を活用していく技術が必要である.

以上のような課題認識から, 2 章では, 本研究において扱う次の 3 つの課題について述べた.

- 実行可能仕様における仕様と設計の分離に関する課題
- レビューとテストの用途として形式仕様記述を利用する場合の課題
- 形式仕様記述手法では取扱いが容易でない動的な振る舞いに関する課題

1 つ目の課題の解決方法として, 4 章では, 「実行可能性の影響」を回避する形式仕様の記述スタイルと, 内部データ構造に着目した仕様と設計の分離方法を提案した.

本章では, 2 つ目の課題を解決するレビューとテストの用途を考慮した仕様記述フレームワークを提案する. また, 実際に, 形式仕様記述から体系的にテスト項目を作成し, テスト項目とテストスクリプトの網羅性を機械的に検証する方法を提案する. レビューとテストの用途として形式仕様記述を利用する場合の課題は, 2.2 節において考察した.

レビューにおける課題は, 次のような課題であった. 形式仕様記述手法の専門家ではない, ドメインエンジニア, 仕様策定者, 設計者, 実装者, 評価者といった, 様々な読者が形式仕様をレビューに用いる場合を考える. この時, レビューの目的によって, 読者が必要とする仕様の詳細度が異なる. 例えば, ドメインエンジニアが, 仕様が要求を満たしていることを確認するレビューと, 実装者が, 実装コードが仕様を満たしていることを確認するレビューとでは, 必要となる仕様の詳細度は異なる. ドメインエンジニアが必要とする仕様の詳細度に比べ, 設計者と実装者と評価者はより詳細度の細かい仕様を必要とする

ことが多い. このような仕様の詳細度の分析から, 様々なレビューの目的を持った読者にとって, 読みやすい仕様書を次のように考えた.

- 概要と詳細を階層化して明確に区別して記述してあり, 概要のみを読むことで, 仕様の全体像を理解することができる仕様記述.
- 概要から詳細へと即座にたどることができる仕様記述.
- 形式仕様記述手法の専門家ではない人であっても, 仕様を読み進めることができるように, 特に概要の記述は簡潔な構造を持つ仕様記述.

このような仕様であれば, ドメインエンジニアが仕様をレビューする場合は, まず概要を読み進め, 必要に応じて概要から詳細を読むことができる. 設計者と実装者と評価者は, 概要を把握した上で, 概要と詳細を往き来しながら仕様を読み進めることができる. 本章では, 以上のような課題を解決する仕様記述フレームワークを提案する.

仕様に基づくテストにおける課題は, 次のような課題であった. 仕様に基づくテストとは, 実装のコードが仕様を満たしていることを確認するテストのことであった. 仕様に基づくテストの工程において, 形式仕様記述手法を活用することで, メリットを感じることができるように, 本研究では次の 2 つの課題を設定した.

- まず, 仕様に基づくテストにおいて前提とする品質目標を決める. その上で, 品質目標を満たす体系的なテスト項目の作成方法を研究課題とした. これは, 体系的にテスト項目を作成することができるような, 形式仕様記述の設計を行わなければならないことでもある.
- テスト項目とテストスクリプトが品質目標を満たしていることを, 形式仕様記述を用いた機械的な処理により確認する方法を研究課題とした.

以上の課題設定から, 本研究において定めた仕様に基づくテストの品質目標と, 体系的にテスト項目を作成するためのテスト設計および, そのための仕様記述フレームワークを提案する. そして, テスト項目とテストスクリプトが, 品質目標を満たしていることを確認するために, ログ出力による機械的な検証方法を提案する.

5.1 節では, まず, レビューとテストの用途を考慮した仕様記述フレームワークを提案する. ここでは, ディシジョンテーブルとラベル付き条件がフレームワークの中心的な概念となる. これらの概念については, 5.1 節において述べる. 次に, レビューの用途から, 仕様記述フレームワークの理解容易性を考察することにより, 前述のレビューにおける課題への解決方法を示す. 5.2 節では, 5.1 節において述べた仕様記述フレームワークを, 仕様に基づくテストに活用する方法を提案することにより, 前述のテストにおける課題への解決方法を示す.

まず, 次節において, 本章において提案する仕様記述フレームワークについて述べる.

5.1　ラベル付き条件による仕様記述フレームワークの提案

本章において提案する仕様記述フレームワークは, 4.1 節において述べた宣言的な記述を行うためのフレームワークである. つまり, 図 4.3 または, 図 4.4 に示した模式的な図において, 宣言的な記述と示した箇所に提案のフレームワークを用いることができる. 本章では, 図 4.4 の記述スタイルを用いて説明する. この記述スタイルを, VDM++ を用いて記述した例を **図 5.1** に示す. この例は, 図 4.9 に示した, コマンド仕様を記述するためのテンプレートの例を簡略化したものである. この VDM++ の記述例を用いて仕様記述フレームワークを説明していく. この VDM++ の記述例と, 図 4.4 の対応関係を **図 5.2** に示す. 図 5.1 に示した RunSpecification 関数は実行可能仕様の最上位の関数であり, RunSpecification 関数を実行することで仕様の動作検証を行う. RunSpecification 関数の関数本体に記述した「 is not yet specified 」は, 図 5.2 に示す操作的な記述箇所に対応するため非伝達部となる. Specification 関数の内容は, 図 5.2 の宣言的な記述箇所に対応するため仕様伝達部となる. 4.1.2 節で述べたように, 非伝達部と仕様伝達部は, 「実行可能性の影響」を回避するために, 「動かすこと」を目的とした記述と「伝えること」を目的とした記述を分離するための仕組みであった. 図 5.2 に示す「入力」と「入力時の内部状態」は, 図 5.1 においては, それぞれ「 in : INPUT 」と「 state_in : STATE 」である. ここでは, 「変数名:型名」という表記を用いることとする. 「出力」と「出力時

```
RunSpecification : INPUT * STATE -> OUT * STATE
RunSpecification (in, state_in) ==
  is not yet specified
post
Specification (in, state_in, RESULT.#1, RESULT.#2);
```

図 5.1 図 4.9 を簡略化した図 4.4 に示した提案の記述スタイルの記述例

の内部状態」は,「 RESULT.#1 : OUT 」と「 RESULT.#2 : STATE 」である.

以降では, 図 5.1 に示した Specification 関数の内部について議論する. まず次節において, 仕様記述フレームワークを議論する上で, 重要な概念であるディシジョンテーブルについて述べる.

5.1.1 ディシジョンテーブルの構成を用いた仕様記述フレームワーク

本節では, ディシジョンテーブルに対応づけた仕様記述フレームワークについて述べる. ディシジョンテーブルとは, 条件の組み合わせとアクションの関係を表現したものである. 一般に, ディシジョンテーブルにおいて, 処理の内容をアクションと呼び, 条件の組み合わせとアクションの関係をルールと呼ぶことがある. ディシジョンテーブルは, 仕様や設計やテストなどにおいて広く用いられている. ATM の仕様を表したディシジョンテーブルの例を**表 5.1** に示す. これは, 現金を引き出す場合の一部の仕様をディシジョンテーブルを用いて表したものである. ディシジョンテーブルは, 大きく条件の組み合わせとアクションから構成される. 読み方は, ディシジョンテーブルのルールの列を番号ごとに縦に読んでいく. ここで,「 Y 」は YES を表し,「 N 」は NO を, ダッシュ記号「–」は, 条件が判定の対象外であることを表している. 以下に, 番号が 1 のルールを用いてディシジョンテーブルについて説明する.「ユーザが有効なカードを挿入」という条件が「 N 」の場合は,「ユーザが有効な暗証番号を入力」などの条件が「–」となっており, 暗証番号の照合や, 残高確認は行わないことを表している. これらの条件の組み

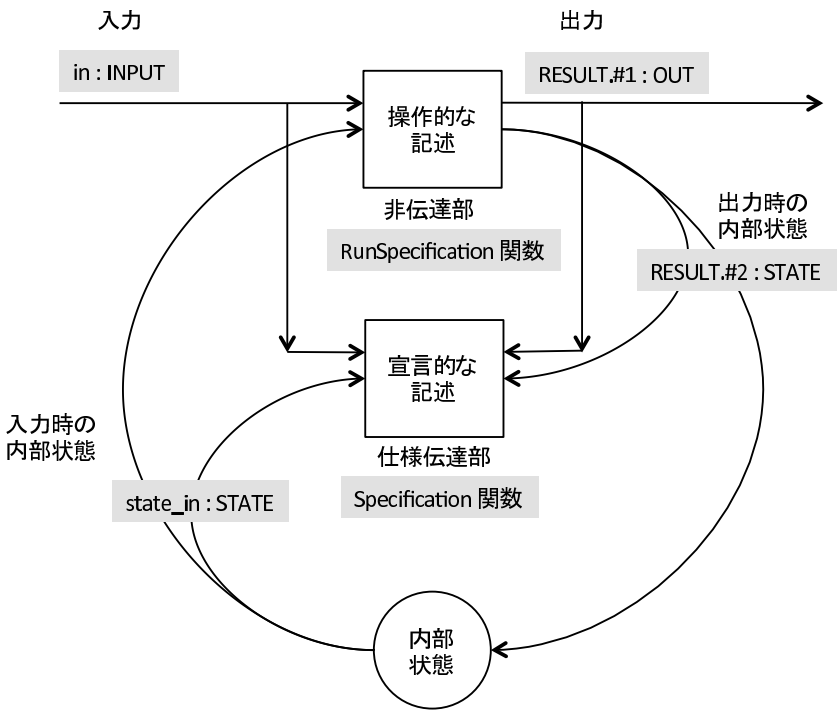

図 5.2　提案の仕様記述スタイルと図 5.1 の仕様記述例との対応関係

合わせの結果, 「カードの受け付けを拒否する」というアクションが選択されることを表している.

ここで, ディシジョンテーブルを作成する上で, 次の 2 点に注意する必要がある.

- アクションの選択は, 条件が評価される順番に依存しないようにする必要がある. つまり, ある条件判定によって, 内部状態が遷移したり, 他の条件判定に使用する変数を更新するようなことがないようにする必要がある.
- アクションの選択はディシジョンテーブルの条件によって決定し, 記載されていない入力条件には依存しないようにする必要がある. つまり, アクションを選択するために必要な, 全ての条件がディシジョンテーブルに記載されていなければならない.

以上の 2 つの注意点のうち, 1 つ目の注意点である条件間の依存関係は, 仕様を記述する上で必要なことが多い. この点について 5.1.3 節において解決方法を議論する.

本研究において提案する仕様記述フレームワークの構成は, このディシジョンテーブ

表 5.1 ATM の仕様を表したディシジョンテーブルの例

	ルール				
条件	1	2	3	4	5
ユーザが有効なカードを挿入	N	Y	Y	Y	Y
ユーザが有効な暗証番号を入力	-	N	N	Y	Y
無効な暗証番号を 3 回入力	-	N	Y	N	N
引き出し額が残高以内	-	-	-	N	Y
アクション					
カードの受け付けを拒否する	Y	N	N	N	N
暗証番号の再入力を促す	N	Y	N	N	N
カードを差し押さえる	N	N	Y	N	N
要求額の再入力を促す	N	N	N	Y	N
要求額の現金を払い出す	N	N	N	N	Y

ルの構成を参考にしている. その理由は次の 2 つである.

- ディシジョンテーブルは, 仕様を簡潔な構造で表現することがきる. また, 一覧性にも優れている. そのため, この構造を仕様記述フレームワークに適用することで, フレームワークもまた簡潔になると考えたためである.

- ディシジョンテーブルは, テストとの親和性が良い. 5.2 節において述べる, 形式仕様記述を利用した仕様に基づくテストのテスト設計に, ディシジョンテーブルを用いている. 仕様記述フレームワークの構成を, テストにおいて用いるディシジョンテーブルの構成と合わせておくことで, 仕様とテスト項目の対応関係が明確になると考えた. これにより, 仕様から体系的にテスト項目を抽出することができるようになるため, 仕様とテスト項目のレビューも容易になると考えたためである.

次に, 仕様記述フレームワークの構成を議論する. 表 5.1 に示したように, ディシジョンテーブルは, 大きく条件の組み合わせとアクションに分かれている. 本研究では, ディシジョンテーブルとの対応関係を明確にするために, 仕様記述を, 条件の組み合わせの仕様に関して記述した箇所と, アクションの仕様を記述した箇所に分けた. 前者を仕様記述フレームワークの Action Selection 部 と呼び, 後者を仕様記述フレームワークの Action

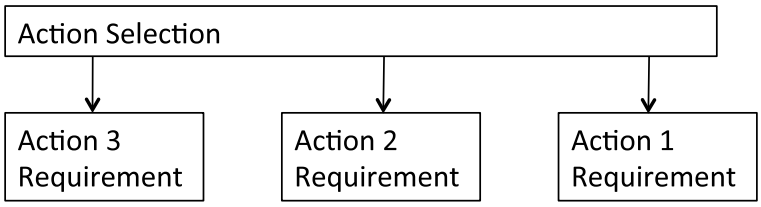

図 5.3　Action Selection 部と Action Requirement 部の構成例

Requirement 部と呼ぶこととする.

次の 5.1.2 節では仕様記述フレームワークの基本型について述べる. 仕様記述フレームワークの基本型とは, Action Selection 部内では条件の判定のみを行い, 条件判定中に入力データや内部状態を変更する必要がない簡単な仕様記述の構造である. その次の 5.1.3 節では Action Selection 部内で入力データや内部状態を変更しながら条件判定を行う必要がある仕様記述フレームワークの拡張型を提案する. これは, 5.1.1 節のディシジョンテーブルを作成する上での注意事項として述べた, 条件間に依存関係がある場合に用いる仕様記述フレームワークである. 例えば, 暗号化したデータを入力とする場合, 暗号化データの復号に関する Action Selection 部の条件記述と, 入力データを復号処理した後に, 復号した入力データに関する条件を記述した Action Selection 部が必要になる. このように, Action Selection 部内で処理が必要な仕組みを仕様記述フレームワークの拡張型と呼ぶこととする.

5.1.2　仕様記述フレームワークの基本型

以降では, 図 5.1 に示した Specification 関数の内部について議論する. **図 5.3** は, Specification 関数内部における, 前述の Action Selection 部と Action Requirement 部の構成例を示している. Action Selection 部の条件判定により, 3 つのアクション (Action1 Requirement, Action2 Requirement, Action3 Requirement) のうちいずれかが選択されることを表している. Action Selection 部では, 図 5.2 に示した「入力」と「入力時の内部状態」からアクションを選択する仕様を記述し, Action Requirement 部では, 「入力」と「入力時の内部状態」, 「出力」と「出力時の内部状態」から, アクションを実行した

表 5.2 図 5.4 の仕様記述をディシジョンテーブルで表した例

	ルール			
条件	1	2	3	4
ConditionA	N	Y	Y	Y
ConditionB	-	N	Y	Y
ConditionC	-	-	N	Y
アクション				
Act1Requirement	Y	N	N	N
Act2Requirement	N	Y	Y	N
Act3Requirement	N	N	N	Y

結果の事後条件を記述する.

図 5.3 の例を VDM++ により記述した例を**図 5.4** に示す. Specification 関数は図 5.2 において示した, 宣言的な記述箇所に対応する仕様伝達部である. すなわち, 「入力」と「入力時の内部状態」である, 「 in : INPUT 」と「 state_in : STATE 」, 並びに, 「出力」と「出力時の内部状態」である, 「 out : OUT 」と「 state_out : STATE 」を用いて仕様を宣言的に記述することができる.

Specification 関数内の cases 式において, ActionSelection 関数を呼び出し, この関数の結果から, Act1Requirement 関数, Act2Requirement 関数, Act3Requirement 関数を選択する. Specification 関数は, ディシジョンテーブルの条件の組み合わせとアクションの関係を記述したものであり, ActionSelection 関数は, ディシジョンテーブルの条件の組み合わせを記述したものである. 図 5.4 に対応するディシジョンテーブルを**表 5.2** に示す. 例えば, ディシジョンテーブルのルール 1 から, ConditionA 関数が「 N 」つまり false である場合, Act1Requirement 関数が選択されることが分かる.

図 5.4 に示した ActionSelection 関数の内部は, cases 式と条件群からなる簡潔な構造となっている. 簡潔性を保つために以下のようなコーディングガイドラインを定めている. まず, VDM++ の言語仕様から cases 式の構文例を**図 5.5** に示す.

「 e 」 は 1 つの任意の式であり, 「 pij 」は 1 つ 1 つが式 e にマッチするパターンである. VDM++ の言語仕様上, 「 pij 」のパターンの列をパターンリストと呼ぶ. 上

```
Specification : INPUT * STATE * OUT * STATE -> bool
Specification (in, state_in, out, state_out) ==
  cases ActionSelection (in, state_in) :
  <ACTION1> ->
    Act1Requirement (in, state_in, out, state_out),
  <ACTION2> ->
    Act2Requirement (in, state_in, out, state_out),
  <ACTION3> ->
    Act3Requirement (in, state_in, out, state_out),
  others ->
    false
  end;

ActionSelection : INPUT * STATE -> SELECTION_RESULT
ActionSelection (in, state_in) ==
  cases false :
    (ConditionA (in, state_in)) ->
      <ACTION1>,
    (ConditionB (in, state_in)),
    (ConditionC (in, state_in)) ->
      <ACTION2>,
  others ->
      <ACTION3>
  end;
```

図 5.4　基本型の VDM++ による記述例

```
cases e :
  p11, p12, ..., p1k  -> e1,
  p21, p22, ..., p2l  -> e2,
  ...
  pn1, pn2, ..., pnm -> en,
  others             -> enplus1
end;
```

図 5.5 cases 式の構文例

記の例において, i は 1 から n となり, j は 1 から k, l, m を用いて表した範囲の値である. k, l, m, n は 1 以上の整数である. 「 ei 」で表すのは任意の式である. 「 pij 」で表すパターンが e にマッチした場合「 ei 」が呼ばれる. 「 others 」とそれに対応する式「 enplus1 」はオプションである. 「 others 」は, いずれのパターンも e にマッチしなかった場合, 「 enplus1 」が呼ばれることを表している. 次に, 本研究において定めたコーディングガイドラインを以下に示す.

- 「 cases e :」の e は false 値とする. つまり, 「 cases false :」とする. パターンリスト中のいずれかのパターンが false であった場合に「 ei 」が呼び出される. パターンリスト中の全てパターンが true であった場合は「 others 」となり「 enplus1 」が呼び出される.

- 「 pij 」は true または false を返す関数呼び出しにより構成することとする. 例えば, 図5.4 の ConditionA 関数, ConditionB 関数, ConditionC 関数のように, 「 pij 」は関数呼び出しとする.

この構成により, ディシジョンテーブルで表した条件の組み合わせとアクションの関係を, 簡潔な構造を用いて VDM++ により記述することができた. 本研究では, この ConditionA 関数といった ture または false を返す「 pij 」のパターンリスト内の関数を

ラベル付き条件と呼び, パターンリストに列挙したラベル付き条件の集合をラベル付き条件群と呼ぶこととする.

以下では, この cases 式を用いた記述について考察する. cases 式により, ディシジョンテーブルに対応する仕様を, 簡潔な構造で表現することができた. しかし, VDM++ の言語仕様上, パターンリスト中の「 pij 」の評価順序は規定されていないことに注意する必要がある. そのため, パターンリスト中の「 pij 」の中で複数の「 pij 」が false となった場合, どの「 ei 」が選択されるのかは未定義となる. 本来, 「 cases e :」を false とした場合, 「 pij 」中で常に 1 つだけが false となるように, 「 pij 」を互いに素の関係にすることが一般的である. しかし, 本フレームワークにおいて, 互いに素の関係となるように厳密に記述した場合, 記述が複雑になってしまうという課題があった. 例えば, 表 5.2 のディシジョンテーブルのルール 1 において, ConditionA 関数が「 N 」つまり false の場合, ConditionB 関数と ConditionC 関数は「–」として, 条件判定の対象外であることを表している. つまり, ConditionA 関数が false であった場合, ConditionB 関数と ConditionC 関数の判定結果にかかわらず, ConditionA 関数が false の場合のアクションを選択することを表している. しかし, 図 5.4 の VDM++ の記述例では, 言語仕様上は, ConditionA 関数と ConditionB 関数と ConditionC 関数の評価順序は規定していない. これは, 上記のとおり cases 式は「 pij 」の評価順序を規定していないからである. そのため, ConditionA 関数が false であったとしても, ConditionB 関数と ConditionC 関数の結果が false であった場合, ConditionA 関数が false の場合のアクションが選択されるとは限らないこととなる. 現時点は, VDMTools のインタープリタ機能が, パターンリストの先頭から順に評価を行っているため, 「 pij 」の中で先に false となった「 ei 」を呼び出している. VDM++ の言語仕様上は, 本フレームワークの使い方は問題ではあるが, 現在は, このインタープリタ機能の判定順序を仕様として, このディシジョンテーブルの「–」の仕様を表現している. これは, VDM++ の構文を使って, 「–」の仕様を簡潔に表現することは難しいためである. VDM++ の言語仕様の検討も含めて今後の課題としたい.

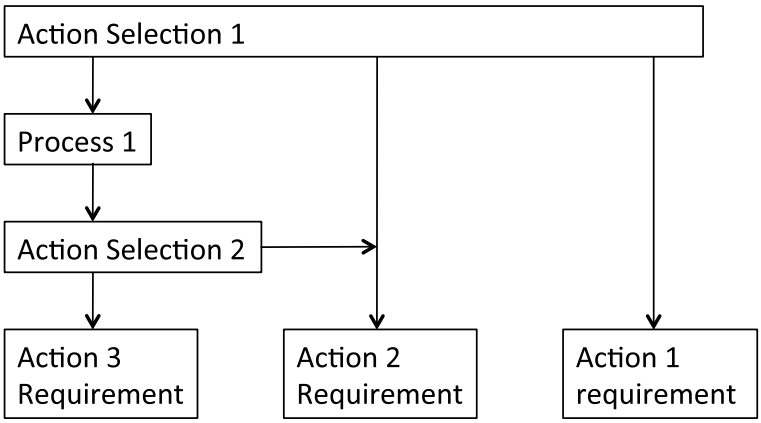

図 5.6 拡張型のフレームワークの例

5.1.3 仕様記述フレームワークの拡張型

本節では, Action Selection 部の条件間に処理が入る場合について, 基本型を拡張したフレームワークについて述べる. **図 5.6** に仕様記述フレームワークの拡張型の例を示す. Action Selection1 と Action Selection2 が Action Selection 部であり, Action Requirement 部として Action1 Requirement, Action2 Requirement, Action3 Requirement がある. ここで Process1 は, Action Selection1 の判定を行った後に, 「入力」と「入力時の内部状態」に対して何らかの処理を行う箇所である. 前述の暗号化データが入力された場合の例において, 復号処理を行う箇所にあたる. この構成により, Action Selection 部内で何らかしらの処理を行って, 条件判定を行う必要がある場合に対応することができる.

以下では, これらの処理を細かく見ていく. Action Selection1 は, Process1 か, Action2 Requirement, Action1 Requirement のいずれかを選択するラベル付き条件群からなる. Action Selection1 の条件判定により Process1 が選択された場合, Process1 において入力データや, 内部状態の変更を行い, Action Selection2 内のラベル付き条件群による条件判定を行う. Action Selection2 のラベル付き条件の判定結果により, この例では, Action3 Requirement, もしくは Action2 Requirement が選択される.

この構成を VDM++ を用いて記述した例を, **図 5.7** に示す. Specification 関数と ActionSelection1 関数, ActionSelection2 関数は, 基本型の仕様記述フレームワークと一部異なるところがあるが, 基本的には同じ構成である. 大きな違いは, 複数の ActionSelec-

```
Specification : INPUT * STATE * OUT * STATE -> bool
Specification (in, state_in, out, state_out) ==
  cases ListedActionSelection (in, state_in) :
  <ACTION1> -> Act1Requirement (in, state_in, out, state_out),
  <ACTION2> -> Act2Requirement (in, state_in, out, state_out),
  <ACTION3> -> Act3Requirement (in, state_in, out, state_out),
  others    -> false
  end;

ListedActionSelection : INPUT * STATE -> SELECTION_RESULT
ListedActionSelection (in, state_in) ==
  let funcs = [ActionSelection1, Process1, ActionSelection2]
  in EvalActionSelection (funcs, in, state_in);

ActionSelection1 : INPUT * STATE -> SELECTION_RESULT |
                                    SELECTION_RESULT * INPUT * STATE
ActionSelection1 (in, state_in) ==
  cases false :
    (ConditionA (in, state_in)) -> <ACTION1>,
    (ConditionB (in, state_in)),
    (ConditionC (in, state_in)) -> <ACTION2>,
  others ->
      mk_(<NEXT>, in, state_in)
  end;

Process1 : INPUT * STATE -> SELECTION_RESULT * INPUT * STATE
Process1 (in, state_in) ==
  mk_(<NEXT>, Update (in), state_in);

ActionSelection2 : INPUT * STATE -> SELECTION_RESULT
ActionSelection2 (in, state_in) ==
  cases false :
    (ConditionD (in, state_in)),
    (ConditionE (in, state_in)) -> <ACTION2>,
  others -> <ACTION3>
  end;
```

図 5.7　拡張型の VDM++ による記述例

tion 関数と Process1 関数を管理することができる ListedActionSelection 関数である. ListedActionSelection 関数では, 次に示すように複数の ActionSelection 関数と Process 関数を列型で管理し, これらの各関数を列の先頭から順に EvalActionSelection 関数内で評価する.

```
let funcs = [ActionSelection1, Process1, ActionSelection2]
in EvalActionSelection (funcs, in, state_in);
```

この例では, EvalActionSelection 関数内において, ActionSelection1 関数が評価され, 次に Process1 関数が評価され, 最後に ActionSelection2 関数が評価される. 順々に各関数が評価される中で, 前の関数の戻り値が次の関数の引数に渡される. 例えば, Process1 関数が評価された後に ActionSelection2 が評価される場合について述べる. 図 5.7 の Process1 関数の関数定義を次に示す.

```
Process1 : INPUT * STATE -> SELECTION_RESULT * INPUT * STATE
```

同一の図中の ActionSelection2 関数の関数定義を次に示す.

```
ActionSelection2 : INPUT * STATE -> SELECTION_RESULT
```

Process1 関数の戻り値である INPUT 型と STATE 型の値が, 次に実行される ActionSelection2 関数の引数である INPUT 型と STATE 型に渡される. つまり, ActionSelection1 関数を評価した後に, Process1 関数により, INPUT 型の入力データと STATE 型の状態を更新し, 次の ActionSelection2 関数の評価を行うことができる. このようにして, 処理が条件判定の間に入る仕様を記述することができる.

以上が, 仕様を読むために読み手が必要となる情報である. この ListedActionSelection 関数と EvalActionSelection 関数の役割を形式仕様記述の補足資料として形式仕様記述の読み方ガイドラインに記述し, 読み手がこれらの複雑な関数を読む必要がないようにした.

以下では, この ListedActionSelection 関数と EvalActionSelection 関数の仕組みについて述べる. これは, 形式仕様記述の書き手には必要な情報ではあるが, 読み手には不要な情報である.

ActionSelection1 関数が次に実行する関数を選択する流れを説明する．ActionSelection1 関数は図5.6 に示すとおり, 次に実行する関数の候補は, Process1 関数と, Act1Requirement 関数, Act2Requirement 関数である．まず, Process1 関数を選択する場合について述べ, 次に Act1Requirement 関数と Act2Requirement 関数を選択する場合について述べる．

Process1 関数を選択した場合は, 次に示す ActionSelection1 関数内の全てのラベル付き条件群 (ConditionA 関数, ConditionB 関数, ConditionC 関数) が true の場合である．

```
cases false :
  (ConditionA (in, state_in)) -> <ACTION1>,
  (ConditionB (in, state_in)),
  (ConditionC (in, state_in)) -> <ACTION2>,
others ->
    mk_(<NEXT>, in, state_in)
end;
```

ActionSelection1 関数の戻り値は, 上記 others の式の評価結果であり, 次のようになる．

```
mk_(<NEXT>, in, state_in)
```

これは SELECTION_RESULT 型と, INPUT 型, STATE 型の組型である．VDM++ は言語仕様として, 複数の型をまとめて一つの型として定義することができる．これを組型と呼ぶ．例えば, 整数が 3 つの組型は int * int * int と定義することができ, この型の値は「 mk_」という構成子を用いて mk_(1, 2, 3) と定義する．「 mk_」は組構成子と呼ばれる．ActionSelection1 関数の戻り値である INPUT 型と, STATE 型の組型の値が, 次に実行される Process1 関数の INPUT 型と STATE 型の引数に与えられ, 順次評価が実行される．

次に, ActionSelection1 関数が次に実行する関数として Act1Requirement 関数を選択した場合の評価の流れを述べる．次に示す ActionSelection1 関数内のラベル付き条件である ConditionA 関数が false の場合, Act1Requirement 関数が選択される流れを説明する．

```
cases false :
  (ConditionA (in, state_in)) -> <ACTION1>,
  (ConditionB (in, state_in)),
  (ConditionC (in, state_in)) -> <ACTION2>,
others ->
    mk_(<NEXT>, in, state_in)
end;
```

このとき, ActionSelection1 関数の戻り値は, SELECTION_RESULT 型の <ACTION1> となる. ListedActionSelection 関数を再度以下に示す.

```
let funcs = [ActionSelection1, Process1, ActionSelection2]
in EvalActionSelection (funcs, in, state_in);
```

ActionSelection1 関数の戻り値として SELECTION_RESULT 型が選択された場合, 上記の EvalActionSelection 関数は, funcs 変数に格納された残りの関数の評価を行わず, 評価対象の funcs 内の関数が戻り値として返した SELECTION_RESULT 型の値 (この場合は <ACTION1>) を戻り値として終了する.

EvalActionSelection 関数が返した戻り値がそのまま ListedActionSelection 関数の戻り値となる. そのため, 次に示す Specification 関数内の cases 式の結果は, <ACTION1> となり Act1Requirement 関数が呼ばれる.

```
  cases ListedActionSelection (in, state_in) :
  <ACTION1> -> Act1Requirement (in, state_in, out, state_out),
  <ACTION2> -> Act2Requirement (in, state_in, out, state_out),
  <ACTION3> -> Act3Requirement (in, state_in, out, state_out),
  others    -> false
  end;
```

以上が, ActionSelection1 関数が次に実行する関数として Act1Requirement 関数を選択

した場合であるが, ActionSelection1 関数が Act2Requirement 関数を選択する場合も同様の流れである.

以上より, ActionSelection1 関数の次に Process1 関数を評価する場合と, Act1Requirement 関数または Act2Requirement 関数を評価する場合で, ActionSelection1 関数の戻り値の型が異なることが分かる. Process1 関数を選択した場合, ActionSelection1 関数の戻り値の型は SELECTION_RESULT 型と INPUT 型と STATE 型の組型であった. Act1Requirement 関数を次に実行する関数として選択した場合, ActionSelection1 関数の戻り値の型は SELECTION_RESULT 型であった. VDM++ は言語仕様として, 異なる型を戻り値として返すことができる. 図 5.7 の ActionSelection1 関数の関数定義を次に示す.

```
ActionSelection1 : INPUT * STATE
  -> SELECTION_RESULT | SELECTION_RESULT * INPUT * STATE
```

戻り値の型が「|」により二つに区切られている. 「|」を用いることにより合併型と呼ばれる型を定義することができる. 合併型は, 「|」により区切られた構成要素の全てを含む型である.

この戻り値の型の違いにより, EvalActionSelection 関数では選択する関数を振り分けている. **図 5.8** に EvalActionSelection 関数の記述を示す. EvalActionSelection 関数は, 他の関数と比較すると複雑な記述であるため, ドメインエンジニアや設計者や評価者などの形式仕様記述手法の専門家ではない読み手が混乱する可能性がある. しかし, 前述の通り, 読み方のガイドラインに EvalActionSelection 関数の役割を明記することにより, この関数は, 各 Action Selection 関数と Process1 関数を処理する動かすための仕組みであることを伝えた.

5.1.4　レビューの用途から見た仕様の理解容易性の評価と考察

本節では, まず, 本章において提案した仕様記述フレームワークが, レビュー用途における課題を解決していることを示す. 次に, ラベル付き条件を用いた場合の記述と, 用い

```
EvalActionSelection :
  seq of (INPUT * STATE -> SELECTION_RESULT | SELECTION_RESULT * INPUT * STATE)
           * INPUT * STATE -> SELECTION_RESULT
EvalActionSelection (funcs_in, in, state_in) ==
  cases funcs_in :
  [func] ^ funcs ->
    cases func (in, state_in) :
    mk_(<NEXT>, next_input, next_state_in) ->
      EvalActionSelection (funcs, next_input, next_state_in),
    selection_result ->
      selection_result
    end
  end;
```

図 5.8 EvalActionSelection 関数の記述

なかった場合の記述を比較し, 仕様の理解容易性と保守性について評価する.

レビューにおける課題は, 次のような課題であった. 形式仕様記述手法の専門家ではない, ドメインエンジニアや仕様策定者, 設計者, 実装者, 評価者といった, 様々な読者が形式仕様をレビューに用いる場合を考えたとき, レビューの目的によって, 読者が必要とする仕様の詳細度が異なるということであった. ドメインエンジニアが必要とする仕様の詳細度に比べ, 設計者や, 実装者, 評価者はより詳細度の細かい仕様を必要とすることが多い. このような仕様の詳細度の分析から, 様々なレビューの目的を持った読者にとって, 読みやすい仕様書を次のように考えた.

- 概要と詳細を階層化して明確に区別して記述してあり, 概要のみを読むことで, 仕様の全体像を理解することができる仕様記述.
- 概要から詳細へと即座にたどることができる仕様記述.
- 形式仕様記述手法の専門家ではない人であっても, 仕様を読み進めることができるように, 特に概要の記述は簡潔な構造を持つ仕様記述.

このような仕様であれば, ドメインエンジニアが仕様をレビューする場合は, まず概要を

読み進め, 必要に応じて概要から詳細を読むことができる. 設計者や実装者, 評価者は, 概要を把握した上で, 概要と詳細を往き来しながら仕様を読み進めることができる.

本章において提案した, 仕様記述フレームワークはディシジョンテーブルと対応づけたものである. したがって, 仕様記述フレームワークは, ディシジョンテーブルの簡潔な構造と, 一覧性に優れた表現形式となっている. まず, Specification 関数内の, Action Selection 部と Action Requirement 部から, 仕様記述の全体構成を把握することができる. 次に, Action Selection 部のラベル付き条件群は, cases 式を用いた簡潔な構造を用いて記述してあるため, 形式仕様記述手法の専門家ではない人であっても, 形式仕様を読み進めることができる. また, ラベル付き条件のラベル名は, ラベル付き条件の中で記述した詳細な条件式に関する概要が分かるように命名している. このため, ラベル付き条件のラベル名を参照していくことで, 形式仕様記述手法の専門家ではない人であっても, 仕様の概要を把握することができる. 読者が, ラベル付き条件の関数内部を参照することで, 詳細な仕様を知ることができる. つまり, ラベル付き条件により, 概要と詳細を階層化して管理し, ラベル付き条件のラベル名を参照していくことで, 仕様の概要を読み進めることができる. 詳細が知りたいときは, ラベル付き条件の関数内部を参照することで, 概要から詳細へと即座にたどることができる. また, 仕様の概要を伝えるために, ラベル付き条件のラベル名の付け方を工夫した. 3.3 節において述べた Jackson が考察した適用領域の用語をラベル名として用いるようにした. これにより, ドメインエンジニアが理解している適用領域の用語であれば, ラベル付き条件のラベル名から記述の内容を理解することができる.

ドメインエンジニアなどが仕様の概要のレビューを行うときは, ラベル付き条件のラベル名を読み進めることでレビューを行い, 設計者と評価者が, 詳細な仕様のレビューを行うときには, まずドメインエンジニアと同様に概要を把握し, 次に個々のラベル付き条件の関数内部のレビューを行った. このようにして, 上に述べた形式仕様記述をレビューの用途で用いるときの課題を解決した.

次に, ラベル付き条件を用いた場合と, 用いなかった場合の記述例を比較することで, 本章において提案した仕様記述フレームワークのレビューにおける理解容易性と, 運用・

保守工程における保守性を評価する．まず, FeliCa IC チップの仕様記述のラベル付き条件の具体例を **図 5.9** に示す．その後, ラベル付き条件を用いない具体例を示す．

まず, ラベル付き条件を用いた場合である．電子マネーの残高情報などを取得する命令をパケットとして FeliCa IC チップが受信したときに, このパケットの正当性を確認する Action Selection 部の仕様記述であり, 次の 2 つのラベル付き条件からなる．

```
(パケット長十分? ("サービス数", cmd_pkt)),
(サービス数範囲内? (cmd_pkt))                -> <ERROR_NO_RESPONSE>,
```

ラベル付き条件の関数名は, FeliCa IC チップにおける適用領域の用語を用いて定義した．「パケット長十分? 」という関数は, 受信したパケットの長さがサービス数まで存在する十分なパケットであるのかを判定する．「サービス数範囲内? 」という関数は, パケットにおいて指定されたサービス数が規定の範囲内にあるのかを判定する．いずれかのラベル付き条件が false となった場合, <ERROR_NO_RESPONSE> を選択する．全てのラベル付き条件が true の場合は <SUCCESS> となる．<ERROR_NO_RESPONSE> とは, FeliCa IC チップが受信した命令に対してレスポンスのパケットを返さない, 無応答となることを表している．この後, <ERROR_NO_RESPONSE> の場合は, 無応答時の Action Requiremet 部の評価に進み, <SUCCESS> の場合は, 正常終了時の Action Requiremet 部の評価に進む．

次に, ラベル付き条件を用いない仕様の記述例を **図 5.10** に示す．形式仕様記述言語を用いたソースコード行の上に, 自然言語を用いてコメントを記載した例である．

まず最初に, 仕様の概要に関する仕様記述の読みやすさについて考察する．図 5.10 の場合, レビューワは概要を把握するためにコメント行を読み, 細かい条件式に関する詳細なレビューを行う場合, 形式仕様記述言語を用いたソースコード行を読むこととなる．ドメインエンジニア等のレビューワは, コメント行とソースコード行からなる記述から, コメント行のみを選択しながら読む必要があり, 図 5.9 の記述例と比較してレビューワの負担が大きいことが分かる．構文解析によりコメント行を抽出したり, ソースコード行とコメント行を色分けすることで, レビューワの負担を軽減できると考えられる．しかし,

```
cases false :
  (パケット長十分? ("サービス数" cmd_pkt)),
  (サービス数範囲内? (cmd_pkt)) -> <ERROR_NO_RESPONSE>,
others ->
  <SUCCESS>
end;
```

図 5.9　ラベル付き条件を用いた記述例

```
-- パケットの長さがサービス数まで存在する十分なパケットであること
packet_len_of_service_num >= 9 and
-- パケットにおいて指定されたサービス数が規定の範囲内であること
1 <= service_num and service_num <= 32
```

図 5.10　ラベル付き条件を用いない場合の記述例

コメント行の色づけや, 抽出を行ったとしても, 条件判定の流れや条件式間の関係性がコメント行に十分に記載されていない場合, ソースコード行とコメント行を相互に参照しなければならない. コメント行において, 条件判定の流れや, 条件間の関係性を記載するには, 各条件のコメントにラベルを付ける必要がある. コメントにラベルを付けて, 条件間の関係性を記載したとしても, 記述量が増えて, 簡潔な記述とはならないことが多い. また, その関係性や処理の流れはテストが行われていないので, 間違いを発見することは容易でない. そのため, 不具合の原因となる可能性がある.

次に, 設計者と実装者と評価者がレビューする場合における, 詳細な条件式のレビューにおける仕様記述の読みやすさついて考察する. 図 5.10 はコメント行の下に条件式があり, 図 5.9 はラベル付き条件の関数内部に条件式を定義している. ラベル付き条件の場合, 設計者と実装者と評価者は, 詳細な条件式をレビューするために, ラベル付き条件の関数名から関数定義の記述箇所に視点を移動する必要があり, 図 5.10 の記述例と比較してレビューワの負担が大きいと考えられる. そこで, 筆者らは VDM++ のソースコード

を HTML に変換し, 関数名をクリックすることで関数定義に移動するツールを開発し, レビューワの負担を軽減した. 形式仕様記述言語は, 構文が厳密に定義されているため, このようなツールを用いるのに適している. これにより, ラベル付き関数の視点の移動の負荷は問題とはならない.

簡単な例ではあるが, 本論文で提案する仕様記述フレームワークを用いることで, レビューワに応じて提示する仕様の詳細度を制御し, 簡潔で読みやすい仕様を記述できることを示した.

最後に, 保守性について考察する. 図 5.10 においてコメントとして記述した内容を, 図 5.9 では関数名として表現している. 筆者らの 第 1 世代の FeliCa IC チップ開発の形式仕様記述は, 図 5.10 に近い記述スタイルであり, ソースコード行のみが更新され, コメント行が更新されないなど, コメント行の保守が課題となった. 第 2 世代の開発では, 本章において述べたラベル付き条件を用いて, 第 1 世代ではコメント行に記載していた内容を, ラベル名などのソースコード行に記載する方針とした. コメント行を少なくすることで, 保守対象が少なくなり作業効率が向上した. また, ソースコード行の記載は, VDMTools を用いて構文検査や, 型検査, 仕様アニメーションによるテストを行うことができ, 繰り返し, 機械的な検証を行うことができるようになった. また, 仕様変更や記述の一貫性の観点から用語の変更を行う際に, リファクタリングに相当する関数名の正確な一括変換など, コメント行に対しては困難であった正確で効率的な変更ができるようになり保守性が向上した.

5.2 仕様に基づくテストへの形式仕様記述の活用方法の提案

本節では, 仕様に基づくテストへの形式仕様記述の活用方法を提案する. まず, 本研究において想定する品質目標について述べる. その後, この品質目標を達成するためのテスト設計と, ログ出力を利用したテスト項目とテストスクリプトの網羅性の検証方法について議論する.

5.2.1　本研究において定めた品質目標

プロジェクト管理者や評価者は, 予算, 製品の特性など, それぞれの開発プロジェクトの状況に応じて品質目標を定める. 以下に, 本研究において定めた品質目標を述べる.

まず, 品質目標として使用している用語として, ブランチカバレッジ, 同値分割, 境界値分析について簡単に説明する. ブランチカバレッジとは, 全ての条件分岐 (ブランチ) が取り得る true と false の結果に対して, 実施したテストの割合である. つまり, 全ての条件分岐が true の場合と false の場合を, それぞれ少なくとも 1 回は実行するとブランチカバレッジが 100% となる.

次に, 同値分割, 境界値分析について説明する. これらは, テストの網羅性を保ちつつ, テスト件数を減らしていく手法として有名である. 同値分割は, 起こりうるすべての事象をいくつかのグループに分け, 各グループから代表値を選ぶ方法である. 一般に, このグループを同値クラスと呼ぶ. 例えば, 年齢が 20 歳未満と 20 歳以上を判別する実装コードがあった場合, 同値分割を用いると, 「 20 歳未満」と「 20 歳以上」の 2 つの同値クラスに分け, その中から代表値を選択してテストを行うこととなる. 入力として, 0 以下の値や数値以外の値を取り得る場合は, 「正数以外」, 「 0 以上 20 歳未満」, 「 20 歳以上」といった同値分割を行う. 同値分割の分割方法は, 上記の例以外にも様々なものが考えられる. 境界値分析とは, 各同値クラスから境界値を選択してテストを行う手法である. 例えば, 上記の例では境界値として 0, 19, 20 をテストに用いる値として使用する.

以上の用語をふまえて, 本研究において定めた品質目標を見ていく.

(1) 仕様の条件分岐についてブランチカバレッジ を 100% とすること.

(2) 同値分割と境界値分析を行うこと.

(3) 少なくとも 1 度は境界値をテスト項目として使うこと.

(4) 各ラベル付き条件は戻り値が true と false である. 最も粗い粒度で同値分割したとき, この 2 つの true と false が同値クラスとなる. ラベル付き条件間の同値分割の組み合わせのパターンとして, ある SELECTION_RESULT 型のアクショ

ンを選択する複数の同値クラスが存在する場合は, 1 つのラベル付き条件の要因によってアクションが選択されるパターンになるようにすることとした. 例えば, 図 5.4 の ActionSelection 関数において, ACTION2 を選択する要因は ConditionB と ConditionC のいずれかが false の同値クラスを選択した場合である. 1 つの同値クラスの要因によって ACTION2 が選択されるために, (ConditionB, Condition C) が (true, false), (false, true) の 2 つの組み合わせをテストケースとして選択する必要がある. なぜなら, (false, false) のみの 1 パターンを選択してしまうと, ACTION2 に遷移する ConditionB と ConditionC の 2 つの要因が存在するため, ConditionB と ConditionC のそれぞれの単体テストとはならないためである.

5.2.2 ディシジョンテーブルを用いたテスト設計手順

テスト設計には, ディシジョンテーブルを用いた. テスト設計に用いたディシジョンテーブルを**表 5.3** に示す.

表 5.3 は 図 5.4 を例に, 上記で定めた品質目標を満たすテスト項目を表したディシジョンテーブルである. 第 1 列目にはラベル付き条件名とアクション名を記載する. 第 2 列目は同値分割の結果からなる. 第 3 列目は境界値分析の結果を記載する構成である. TC1 から TC6 は各テスト項目の番号を表す. 各テスト項目は, 各ラベル付き条件の同値分割の組み合わせと, 結果の組み合わせから構成される.

以下に, テスト項目を作成する手順の概要を示す.

(1) Action Selection 部から抽出した全てのラベル付き条件をディシジョンテーブルの第 1 列目に記載する. ラベル付き条件の最も粒度の粗い同値分割の結果は true と false となる.

(2) Action Requirement 部から, 結果を記述した関数を抽出する. 例えば, 図 5.4 の Specification 関数において結果を記述した関数は, Act1Requirement 関数などである. これらの関数名をディシジョンテーブルの結果の同値分割欄に記載する.

(3) もし境界値分析が可能な条件式であった場合は, 境界値分析欄に同値クラスごと

に境界値を記載する．例えば, ConditionA が以下の仕様であった場合の境界値を表 5.3 に示す.

```
ConditionA : INPUT * STATE -> bool
ConditionA (in, state_in) ==
  if 1 <= in and in <= 16 then
    true
  else
   false;
```

これは, in の入力値が 1 から 16 の範囲の場合は, ConditionA の結果が true となり, 1 より小さい場合と, 16 より大きい場合は, ConditionA の結果が false となる仕様である．したがって, 同値クラスは true と false の 2 つに分割することができる．同値クラスが true の場合の境界値は 1 と 16 であるから, 表 5.3 の境界値の列に, 1 から 16 の範囲が true であることを示すために「 1..16 」と記載する．「..」は開始端と終端を含む範囲を表すものとして使用する．同値クラスが false の場合は 0 と 17 が境界値であるから, 境界値の列に「 0, 17 」と記載する.

(4) 5.2.1 節において述べた品質目標に従った組み合わせとなるように同値クラスを選択する．また, 5.2.1 節で述べた品質目標の (2) を満たすように境界値を割り付けていく．表 5.3 の「$\surd$」は指定した行の同値クラスを選択したことを表す．境界値が存在する同値クラスの場合は,「$\surd$」の代わりに選択した境界値の値を記載する．もし, 同値クラスの組み合わせによるテスト数では, 境界値を全て割り付けることができない場合は, テストケースを増やして境界値を割り付けていく.

このように, これまで述べた仕様記述のフレームワークを使い, 仕様記述から体系的に品質目標を満たすテスト項目を作成することができる．また, 仕様からもれなく抽出できていることをレビューや機械処理により確認することが可能である.

表 5.3 図 5.4 からディシジョンテーブルによるテスト項目の作成例

	同値分割	境界値	TC1	TC2	TC3	TC4	TC5	TC6
ConditionA	true	1..16	1	16			1	16
	false	0, 17			0	17		
ConditionB	true		√	√	√	√		√
	false						√	
ConditionC	true		√	√	√	√	√	
	false							√
結果	ACTION1				√	√		
	ACTION2						√	√
	ACTION3		√	√				

5.2.3 ログ出力によるテスト項目とテストスクリプトの網羅性の検証方法

本節では, テスト項目が品質目標で定めた網羅性を満たしていることを確認する検証方法を提案する. 品質目標で定めた網羅性を満たしているとは, (1) 仕様記述内から全てのラベル付き条件が抽出され, (2) 同値クラスは品質目標に従って組み合わされ, (3) 境界値は少なくとも 1 回は使用されていることである. この中で, (1) と (2) の条件を満たすテスト項目と, テストスクリプトであることを機械的に確認する検証方法を提案する.

仕様に基づくテストは, 実装コードが仕様を満たしていることを, 仕様書から作成したテスト項目を用いて検証することであった. つまり, 仕様を正しさの基準として実装コードをテストする. 本節において述べるテスト項目とテストスクリプトの検証は, 実装コードを検証する前の準備段階である. 準備として, テストスクリプトを作成するためのテストドライバとして実行可能仕様を用いる. そして, 実行可能仕様によりテストスクリプトの網羅性と仕様記述の正しさを同時に確認する. ここでいうテストドライバとは, テストスクリプトを作成するために必要な動作環境である. ここで見つけることができるバグとして, 実行可能仕様のバグとテストスクリプトのバグの 2 つが考えられる.

まず, 実行可能仕様にバグが存在した場合の課題について述べる. 例えば, 表 5.3 の TC6 のテストは, ConditionB が true, ConditionC が false となり結果として Action2 が

選ばれるテストである．ここで問題は, ConditionB が false であっても Action2 が選ばれることである．TC6 のテスト項目は, ConditionC が false となることを確認するテストである．しかし, ConditionB の実行可能仕様にバグがあり, true であるべきところで, false となってしまった場合, 実行可能仕様の ConditionC が評価されていないにもかかわらず, 結果として期待値通りの Action2 が実行される．そのため, 実行可能仕様にバグがあるにもかかわらず, テストが成功したように見えてしまう．ConditionB の実行可能仕様のバグを見つけることはできない上に, TC6 では ConditionC も実行されていないため, ConditionC のバグが存在したとしても見つけることができない．このテストスクリプトを用いた実装コードのテスト段階においても同様に, ConditionB と ConditionC のバグを見つけることができない．

次に, テストスクリプトにバグが存在した場合の課題を考える．例えば, 表 5.3 の TC6 において, 本来は ConditionC を false とするテストにおいて, 誤って, ConditionB の条件が false となるようなテストスクリプトを記述してしまった場合を考える．前述の実行可能仕様にバグが存在した場合と同様に, ConditionC の条件式が評価されずに Action2 が実行され, テストが成功したように見える．このため, テストスクリプト中の ConditionB の記述誤りを見つけることはできない．また, ConditionC を評価するテストスクリプトが存在しないため, ConditionC に関するバグを見つけることができない．このテストスクリプトを用いた実装コードのテスト段階においても同様に, ConditionC にバグが存在した場合, そのバグを見つけることができない．

これらの課題に対して, 実行可能仕様の各ラベル付き条件に, ログを出力させる機構を組み込み, 網羅性を検証する手法を提案する．**図 5.11** にログ出力の機構を示す．TestLog 関数を各ラベル付き条件の前に挿入することによりログ出力を行うことができるようになる．例えば, 表 5.3 の TC6 を実行した場合のログ出力結果を以下に示す．

```
CondA, TRUE
CondB, TRUE
CondC, FALSE
```

もし, TC6 のテストスクリプトに誤りがあり, ConditionB を誤って false となるようなテストを実施した場合のログは以下のようになる.

```
CondA, TRUE
CondB, FALSE
```

簡単な仕組みであるが, ログから ConditionB が true とはならずに false となり ConditionC が評価されていないことが明らかである.

ログを解析してディシジョンテーブルと比較することは, 機械処理により可能である. 全てのラベル付き条件がテスト項目として抽出されていることは, ディシジョンテーブルから機械的に抽出したラベル付き条件の一覧と, ログの出力結果を比較することにより明らかになる.

1 つ目の品質目標であるブランチカバレッジが 100% であることを確認するためには, 全てのラベル付き条件について, ログ上に TRUE の場合と FALSE の場合が存在することを確認することで, 検証を行うことができる.

4 つ目の品質目標である同値分割の組み合わせに関しては, 先ほどの実行可能仕様のバグとテストスクリプトのバグの考察から, 検証を行うことができる. つまり, (1) 仕様記述内から全てのラベル付き条件が抽出され, (2) 同値クラスは品質目標に従って組み合わされていることを機械的に検証することができた.

これらの機械処理が可能となったのは, テストの基となる仕様が, 形式仕様記述言語の厳密な構文を用いて実行可能仕様として記述してあるためである. さらに, テスト項目が, ディシジョンテーブルを用いて機械処理可能な構造化された形式で記述してあるためである. また, 図 5.11 のログ出力の機構として, 重要なポイントは, 以下に述べるような機械的な TestLog 関数の取り外しができることである. 仕様を実行しテストを行うときにはこの機構は必要ではあるが, 仕様のレビュー, 仕様を参照するときには不要である. そのため機械的な取り外しが行えるような構造にしている.

```
values
private io = new IO ()

functions
ActionSelection : INPUT * STATE -> SELECTION_RESULT
ActionSelection (in, state_in) ==
  cases false :
    (TestLog ("CondA") (ConditionA (in, state_in))) -> <ACTION1>,
    (TestLog ("CondB") (ConditionB (in, state_in))),
    (TestLog ("CondC") (ConditionC (in, state_in))) -> <ACTION2>,
  others -> <ACTION3>
  end;

TestLog : seq of char -> bool -> bool
TestLog (message) (condition) ==
  if condition then
    let - = io.echo (message ^ ", TRUE\n") in true
  else
    let - = io.echo (message ^ ", FALSE\n") in false;
```

図 5.11　ログ出力の機構

5.3 レビューとテストの用途を考慮した仕様記述フレームワークのまとめ

本章では, レビュー用途とテストの用途から, 形式仕様記述を記述するための課題と, 活用するための課題について考察し, これらの課題を解決する仕様記述フレームワークとテスト設計を提案した.

レビューの用途として, 様々なレビューの目的を持った読者にとって, 読みやすい仕様書を次のように考えた.

- 概要と詳細を階層化して明確に区別して記述してあり, 概要のみを読むことで, 仕様の全体像を理解することができる仕様記述.
- 概要から詳細へと即座にたどることができる仕様記述.
- 形式仕様記述手法の専門家ではない人であっても, 仕様を読み進めることができるように, 特に概要の記述は簡潔な構造を持つ仕様記述.

これらの課題に対して, ディシジョンテーブルと仕様記述を対応づけた仕様記述フレームワークを用いることで, 概要の仕様を簡潔な構造で表現し, 形式仕様記述手法の専門家ではない人に対して, 理解容易性に優れた仕様記述を提供することができた. また, ラベル付き条件を用いることで, 概要と詳細を階層化し, 概要から詳細へと即座にたどることができる仕様記述を提供することができた. これらの効果を評価するために, ラベル付き条件を用いた場合と, 用いない場合の仕様記述を比較し, レビュー用途における理解容易性と運用・保守工程における保守性を評価した.

テストの用途として, まず, 前提とする品質目標を示した. そして, この品質目標を満たすテスト項目を仕様記述から体系的に抽出することができることを示した. また, ログ出力の機構を仕様記述フレームワークに組み込むことで, テスト項目とテストスクリプトの網羅性を機械的に検証する方法を提案した. これにより, テスト実施者が形式仕様記述を用いることにメリットを感じることができるような, 形式仕様記述の活用方法を提案した.

第6章

動的な振る舞いに関するモデル検査の適用

これまで述べてきた形式仕様記述手法に関する設計・構成法の提案は, 第 1 世代の開発において明らかになった課題を, 第 2 世代の開発において解決していった内容であった. 第 1 世代の開発とは, 2004 年から 2007 年において, フェリカネットワークス株式会社において行ったモバイル FeliCa IC チップ開発のことであり, 第 2 世代の開発とは, 2007 年中頃から約 4 年間, ソニー株式会社にて行った FeliCa IC チップ開発のことであった. 形式仕様記述手法の適用対象は, 図 4.4 に示した「入力」と「入力時の内部状態」, 「出力」と「出力時の内部状態」の関係を記述した仕様である. これは, 一般に, 静的な仕様であり, 機能仕様と呼ばれるものである. 第 1 世代の開発において, 形式仕様記述手法を用いることにより, この静的な仕様である機能仕様を早期に検証し, 不具合を摘出することができた. 一方で, 同時並行に動作する動的な振る舞いに関する検証方法が, 仕様策定後の設計工程において課題となった. これは, 別の見方をすると, 静的な仕様である機能仕様の品質を早期に確保していたため, 動的な振る舞いに関する検証方法を課題とすることができたとも言える. そこで, 第 1 世代のモバイル FeliCa IC チップ開発の設計工程において, 同時並行に動作する動的な振る舞いに関する検証方法を検討した [中津川+06].

本研究では, 動的な振る舞いの検証にモデル検査を用いた. モデル検査とは, システム

の仕様を表現したモデルがシステムに要求される性質を満たすかどうかを自動的に検証する手法である．特に並行性を含むシステムを対象に, モデルをラベル付き遷移システム (labelled transition system) で記述し, 性質を時相論理 (temporal logic) で表すものをモデル検査と呼ぶことが多い．モデル検査ツールとして, LTSA (Labelled Transition System Analyser) [MK06, 藤倉 06] を用いた．次の 2 つの理由により, LTSA を選定した．1 つ目は, 時相論理式を書かなくとも検証できるため, 初学者が取り組み易い点である．2 つ目は, 抽象度の高いモデリングや形式仕様記述と連携して用いることにも適しているためである.

本章では, 導入の目的と検査内容, 検査結果について述べ, その効果と課題を考察する．まず, 6.1 節において導入の目的と, 段階的な導入方法について述べる．ここで, 導入の段階を 3 つに分けて定義する．3 つの段階のうち, 本研究では, 第 1 と第 2 の段階について議論する．これらの段階を, ステップ 1 とステップ 2 と呼ぶこととする．次に, 6.2 節においてモデル検査ツール LTSA について説明する．その後, 6.3 節において, ステップ 1 とステップ 2 の検証範囲について述べ, 6.4 節と 6.5 節おいて, それぞれのステップの実施内容を説明し, 実施結果を考察する．最後に, 6.6 節において, 本章の結論を述べる.

6.1　導入目的と段階的導入

本節では, モデル検査を導入するにあたり, 設定した導入の目的について述べる.

6.1.1　導入目的

本研究では, 次に示す 6 つの目的を設定した.

目的 1.　モデル検査導入の検討, ツールの選定, 導入, 学習, 有用性検証を行う.

目的 2.　既存の仕様の評価を行う．同時に開発プロジェクト内で手法の展開を行う.

目的 3.　実装コードが動作する環境の (主にハードウェアの制約) 条件を満たしていることを検査する.

目的 4. 実装コードを評価するための評価仕様の策定を行う.

目的 5. モデル検査を用いて精密に形式仕様を策定・評価できる方法を検討する.

目的 6. プロセスへのフィードバックを行い, 次世代の開発においては, 仕様検討段階からモデル検査を活用できるようにする.

6.1.2 段階的導入

上記 6 つの目的のために, 下記の 3 段階の導入を行う. 本研究は, ステップ 2 の途上の段階である.

ステップ 1 既存のコンパクトな仕様を対象として, 目的 1 および目的 2 を達成する.

ステップ 2 ステップ 1 よりも複雑な仕様を対象として目的 2 を達成し, 目的 3, 目的 4 に着手する.

ステップ 3 目的 6 までを達成し, 開発プロジェクト全体でモデル検査を使いこなす.

6.2 モデル検査ツール LTSA の概要

動的な振る舞いの検証を行うために, モデル検査ツール LTSA を用いた. LTSA は, 検査対象のシステムを記述した「モデル」 (M) と, 満たすべき性質を定式化した「プロパティ」 (P) を記述することで, 「モデル」が「プロパティ」を満たしていることを自動的に検査することができる. 「モデル」は, 有限状態機械 (finite state machine) により記述する. 有限状態機械とは, 有限個の状態と, 状態の遷移を表す動作であるアクションの組み合わせからなる, 数学的に抽象化された記述である.

検査対象のシステムが満たすべき性質として安全性 (safety) と活性 (liveness) がある. 安全性とは, 「検査対象が決して悪い状態には到達しない」という性質である. 安全性の例としては, 次の遷移が存在しないデッドロックや, 資源の共有違反などがある. 活性とは, 「検査対象がいつか良い状態に到達する」という性質である. LTSA はこれらの性質

を検査する機能を備えており, 特にあるアクションがいずれ起こることが常に成り立つ性質をプログレス性 (progress) と呼び, 時相論理式を用いるものと区別している. 本論文では, LTSA のプログレスチェック機能を使用したプログレス性の検査をプログレスチェックと呼び, プログレス性に違反していることをプログレスエラーと呼ぶこととする.

LTSA の入力言語は FSP (finite state process) と呼ばれる. FPS の言語仕様として, CardMode, PowerOff のような大文字から始まる文字列はプロセス名を表し, powerOn, resetMode のような小文字から始まる文字列はアクション名を表す. 状態の遷移は「->」を用いて次のように表現する.

```
(a -> P)
```

これは, 最初にアクション a を実行し, 次にプロセス P の振る舞いとなることを表している. あるプロセスから複数のプロセスへ遷移するアクションは, 選択演算子「|」を用いて表現する. 例えば, 次の PowerOn プロセスの FSP 記述を用いて説明する.

```
PowerOn = (resetMode->InitMode0
  | powerOff->PowerOff),
```

PowerOn プロセスから InitMode0 プロセスと PowerOff プロセスを実行することができることを表している. それぞれのアクション名は resetMode と powerOff である. resetMode を選択した場合は InitMode0 プロセスを実行し, powerOff を選択した場合は PowerOff プロセスを実行する.

LTSA の描画機能を使用して, FSP 記述を LTS (labelled transition system) に変換することができる. LTS とは, 各状態と遷移にラベルを付けた**図 6.1** の図に示したような表現形式である. ここでは, FSP と LTS の概念の説明にとどめ, 図 6.1 において図示した各アクションと各状態の詳細については述べない. LTS において, 番号が割り振られた円は状態を表し, 番号 0 の状態は初期状態であり, アクション名のラベルが割り振られた矢印付きの線により状態の遷移を表現している. 図 6.1 において, 例えば, 前述の PowerOn プロセスは番号 1 のプロセスであり, resetMode アクションにより番号 2 の InitMode0

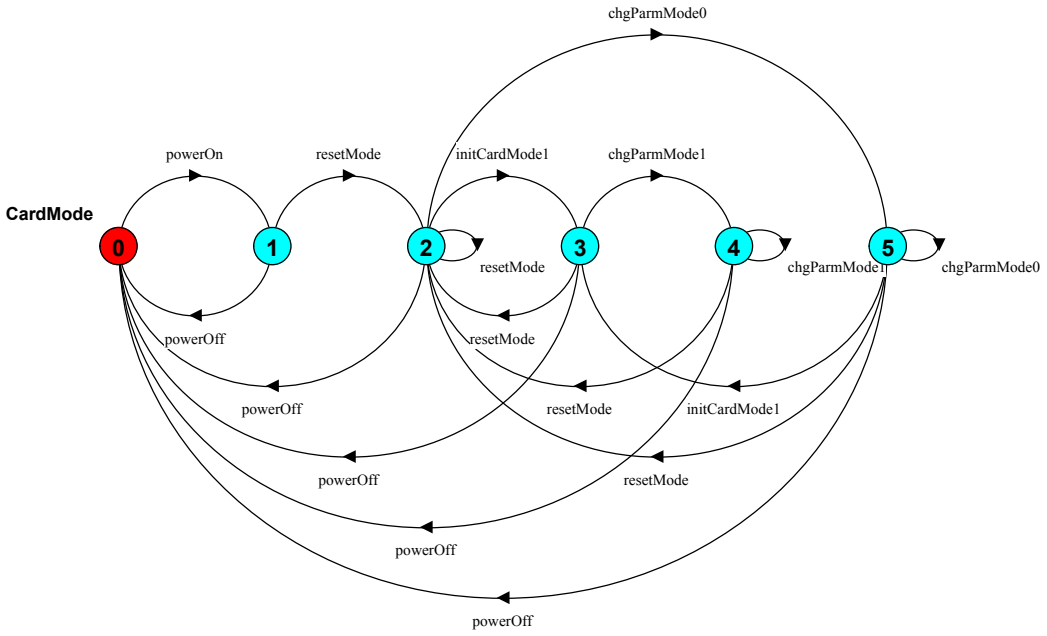

図 6.1 簡易化したカード機能の LTS

プロセスに遷移し, powerOff アクションにより番号 0 の PowerOff プロセスに遷移している.

6.3 ステップ 1 とステップ 2 の検証範囲

本節では, ステップ 1 とステップ 2 の検証範囲について, 検査対象の静的な構造を表現することができる「構造図 (structure diagrams) 」[MK06] を用いて説明する.

モバイル FeliCa IC チップの検査対象の構造図を**図 6.2** に示す. 構造図は, 複数のプロセスとプロセス間で共有するアクションを図示したものである. プロセスとは, 一つの逐次プログラムの実行のことであり, 有限状態機械を用いて表現することができる. 有限状態機械は, 状態とアクションから構成される. ここでは, 一つのアクションが実行中に他のアクションが割り込むことはできないものとする. 構造図において, 各プロセスは長方形で表し, 境界線上の円は公開しているアクションを図示している. 公開しているアクション同士を接続している線はプロセス間でアクションを共有していることを示す.

図 6.2 において図示している内容は, 検査対象のモバイル FeliCa IC チップと, 外部環

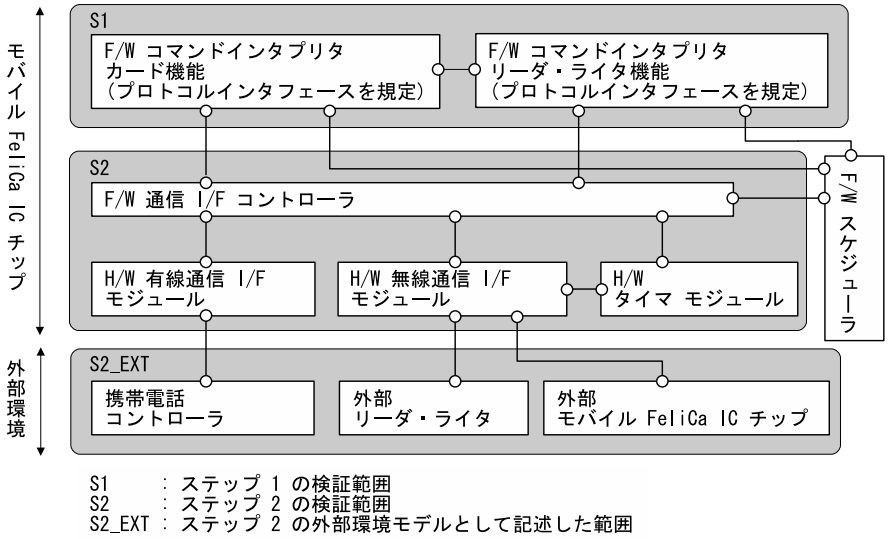

図 6.2　モバイル FeliCa IC チップの構造図

境に分類することができる．外部環境とは, 物理的に検査対象の外側に位置しているものである．図の S1 で囲った範囲はステップ 1 の検証対象を表し, S2, S2_EXT で囲った範囲はステップ 2 の検証対象を表している．S1 と S2 は検査対象のモバイル FeliCa IC チップのプロセスであり, S2_EXT は外部環境のプロセスである．EXT は external の略称として用いた．

モバイル FeliCa IC チップは, 電子マネーの残高情報を保持するようなカード機能と, 店舗のレジ端末にあるリーダ・ライタのようなリーダ・ライタ機能を備えている．また, リーダ・ライタからモバイル FeliCa IC チップにアクセスすることができる無線通信インタフェース機能に加え, 携帯端末側からモバイル FeliCa IC チップにアクセスすることができる有線通信インタフェース機能を備えている．これらの機能を実現するために, モバイル FeliCa IC チップは 図 6.2 に示した構造を持っている．以下では, 図 6.2 に示した各プロセスについて説明する．記載を簡略にするためにハードウェアを H/W と, ファームウェアを F/W, 通信インタフェースを通信 I/F という略称で記載することにする．

まず, S1 の範囲にあるモバイル FeliCa IC チップのプロセスは次のようになる．

F/W コマンドインタプリタ カード機能

カード機能のコマンドを解釈するプロセスである.

F/W コマンドインタプリタ リーダ・ライタ機能

リーダ・ライタ機能のコマンドを解釈するプロセスである.

次に, S2_EXT の範囲にある外部環境のプロセスの説明を次に示す.

携帯電話コントローラ

携帯端末からモバイル FeliCa IC チップの有線通信インタフェース機能を使用してモバイル FeliCa IC チップ内のデータを読み書きするプロセスである.

外部リーダ・ライタ

店舗のレジ端末にあるようなリーダ・ライタのプロセスである. モバイル FeliCa IC チップの無線通信インタフェース機能を使用してモバイル FeliCa IC チップ内のデータを読み書きするプロセスである.

外部モバイル FeliCa IC チップ

検査対象とは別のモバイル FeliCa IC チップのプロセスである. 検査対象のモバイル FeliCa IC チップがリーダ・ライタとして機能する場合に, 読み書き対象とするカード機能のプロセスである.

モバイル FeliCa IC チップと外部環境において共有するアクションの例として, 携帯端末の画面にモバイル FeliCa IC チップ内に保持している電子マネーの残高情報を表示する場合を説明する. 「携帯電話コントローラ」はモバイル FeliCa IC チップに電子マネーの残高確認を行うアクションを送信し, モバイル FeliCa IC チップは残高情報を「携帯電話コントローラ」に送信することにより実現する.

最後に, S2 で囲んだプロセスについて説明する. S2 で囲んだプロセスは, モバイル FeliCa IC チップの H/W モジュールと F/W モジュールの構造を表している. 以下に S2 の各プロセスについて説明する.

H/W 有線通信 I/F モジュール

有線通信 I/F 機能を実現する H/W モジュールのプロセスである. 受信開始や受

信中, 受信完了割り込み, 送信開始, 送信中, 送信完了割り込みなどのアクションを他のプロセスと共有する.

H/W 無線通信 I/F モジュール

無線通信 I/F 機能を実現する H/W モジュールのプロセスである. H/W 有線通信 I/F モジュールと同様に受信開始や受信中, 受信完了, 送信開始, 送信中, 送信完了割り込みなどのアクションを他のプロセスと共有する.

H/W タイマモジュール

時間を計測する H/W モジュールのプロセスである. タイムアウト値設定, タイマ開始, タイムアウト割り込みなどのアクションを他のプロセスと共有する.

F/W 通信 I/F コントローラ

「 F/W コマンドインタプリタ カード機能」および「 F/W コマンドインタプリタ リーダ・ライタ機能」からの指示により,「 H/W 有線通信 I/F モジュール」と「 H/W 無線通信 I/F モジュール」の排他制御などの管理を行うプロセスである. 例えば,「 H/W 有線通信 I/F モジュール」においてデータを受信中に,「 H/W 無線通信 I/F モジュール」からデータを受信しないような制御を行っている.

以上が, 検査対象とする, モバイル FeliCa IC チップと外部環境の構成である.

ステップ 1 の検証対象は, S1 で示したコマンドインタプリタである. コマンドインタプリタには, カード機能とリーダ・ライタ機能があり, 各機能は並列に動作する. ステップ 1 では, 記述範囲を仕様書が規定する両機能の状態遷移とした. ステップ 2 の検証対象は, S2 で示した 2 つの通信 I/F を制御している H/W モジュールと,「 F/W 通信 I/F コントローラ」である. ステップ 2 の検証において, S2_EXT で示したプロセスを外部環境モデルとして記述した.

上記の通り, モバイル FeliCa IC チップは, 外部環境も含め同時並行に動作するプロセスを制御する特徴があり, 仕様・設計ともに, 複数のプロセスに関する厳密なモデルが必要である. 以下, ステップ 1 とステップ 2 におけるモデルやプロパティの作成方法, 安全性と活性の検証結果, 考察について述べる.

表 6.1 ステップ 1 の検査項目と結果

検査項目	目的	入力	出力
1. 仕様モデルの作成	状態遷移仕様モデルの作成	(1) 状態遷移図 (UML 状態遷移図) (2) 状態遷移仕様 (形式仕様)	(1) 仕様モデル (FSP) (2) 状態数 (本文参照) (3) 遷移数 (本文参照) (4) プロセス数: 2 (5) LTSA 適用可能性: 可能
2. 仕様モデルの検査	状態遷移仕様の検査	仕様モデル (FSP)	(1) デッドロック: なし (2) プログレスエラー: なし (3) 状態の不正干渉: なし
3. プロパティの作成と検査	共有変数仕様の一部を検査 (初期化前に参照されないことの検査)	(1) 仕様モデル (FSP) (2) 共有変数仕様 (UML 状態遷移図) (3) 共有変数仕様 (形式仕様)	(1) 共有変数のプロパティ (FSP) (2) プロパティ違反: なし
4. シナリオの検査	一部コマンド仕様の検査	(1) 仕様モデル (FSP) (2) コマンド仕様 (形式仕様)	(1) コマンド仕様から作成したシナリオの検査モデル (FSP) (2) コマンド仕様 (FSP, LTS)

6.4 ステップ 1 の実施内容と結果・考察

ステップ 1 の検証対象は図 6.2 に示したカード機能とリーダ・ライタ機能の状態遷移である. ステップ 1 では特に, 各機能の並列動作時に, 各々の状態が不正に干渉し合わないことを検証目的とした. 具体的には, カード機能とリーダ・ライタ機能の個々の振る舞いが, 両者が並列に動作した時に制約を受けて, 個々の振る舞いとは異なる振る舞いになっていないことを検証した. 検査項目と結果を**表 6.1** に示す. 以下, 表 6.1 の各検査項目について, 実施内容と結果・考察を述べる.

6.4.1 仕様モデルの作成

UML で記述された状態遷移図を, LTSA の入力言語である FSP (finite state process) の記述に変換した. 具体的には, カード機能とリーダ・ライタ機能でそれぞれのモードを保持しており, コマンドによりモードが切り替わる遷移を FSP を用いて記述した.

6.4.2　仕様モデルの検査

6.4.1 節で作成した仕様モデルに対して LTSA のプログレスチェック機能を使用して, 各機能の状態が不正に干渉し合わないことを検査した. 各機能とは, カード機能とリーダ・ライタ機能である.

これは, LTSA の並列合成という機能を使用することで, 検査を行うことができる. 並列合成とは, プロセス間で共有している同一名称のアクションを同期して動かし, 共有していないアクションはインタリーブさせて動作させる. FSP の言語仕様として, 「$\|$」は並列合成を行う並列合成演算子である. この演算子を用いると P と Q のプロセスを並列合成する FSP 記述は, $(P\|Q)$ と記述できる. 例えば, 以下に示す P と Q のプロセスを定義する.

```
P = (a -> STOP).
Q = (b -> c -> STOP).
```

この 2 つのプロセスを並列合成したプロセス R を得るには, 次のように記述する.

```
|| R = (P || Q).
```

プロセス P と プロセス Q と プロセス R を LTS として出力すると**図 6.3** のようになる. 図 6.3 のプロセス R においてプロセス P の a のアクションが, プロセス Q のアクション b の実行前と実行後, アクション c の実行後の 3 箇所にインタリーブしていることが分かる.

例えば, カード機能のプロセスを CardMode とし, リーダ・ライタ機能のプロセスを RWMode とすると, 両者が同時に動作した場合のプロセスは, CardMode プロセスと RWMode プロセスを並列合成することにより得られる. このプロセスを CardAndRW とすると, FSP を用いて次のように定義できる.

```
|| CardAndRW = (CardMode || RWMode).
```

説明のために, 実際とは異なる簡易化したカード機能とリーダ・ライタ機能を用いて, 並列合成した LTS の結果を **図 6.4** に示す. 図の詳細はここでは述べない. 図から分かる

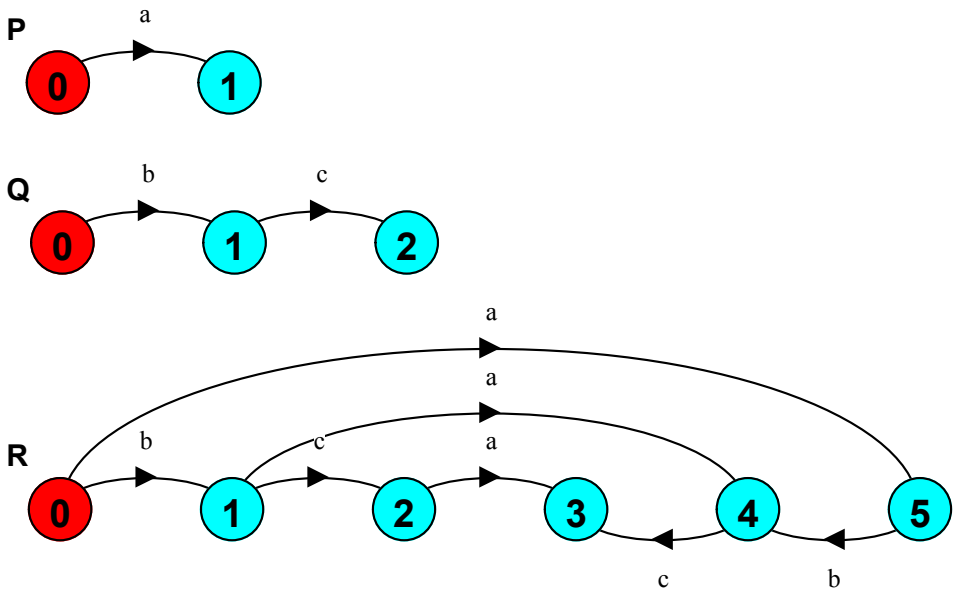

図 6.3 並列合成の例

ように, 簡易化した仕様であっても, カード機能とリーダ・ライタ機能を並列合成したプロセス CardAndRW を LTSA の並列合成機能を用いないで作成することは難しい. 作成することができたとしても, 並列合成後の状態遷移において, デッドロックおよびプログレスエラーが起きていないことを目視により確認することは困難である. LTSA を使用することにより, 機械的に並列合成後の FSP 記述を作成し, その全ての状態について, デッドロックおよびプログレスエラーの確認を行うことができる.

以上より, カード機能とリーダ・ライタ機能が共有するアクションによって, 各機能が相互に干渉し, デッドロックおよびプログレスエラーが起きていないことを検査した.

6.4.3 プロパティの作成と検査

仕様モデルについて, 満たすべき性質 (プロパティ) を定式化することで, その性質を仕様モデルが満たしていることを網羅的に検査することができる. 本研究では, プロパティとして各機能が共有して使用している変数が, 初期化前の未定義の状態で参照されていないことを定式化した.

例えば, カード機能のみの振る舞いでは, 正しく変数の初期化を行い, 初期化後の変数

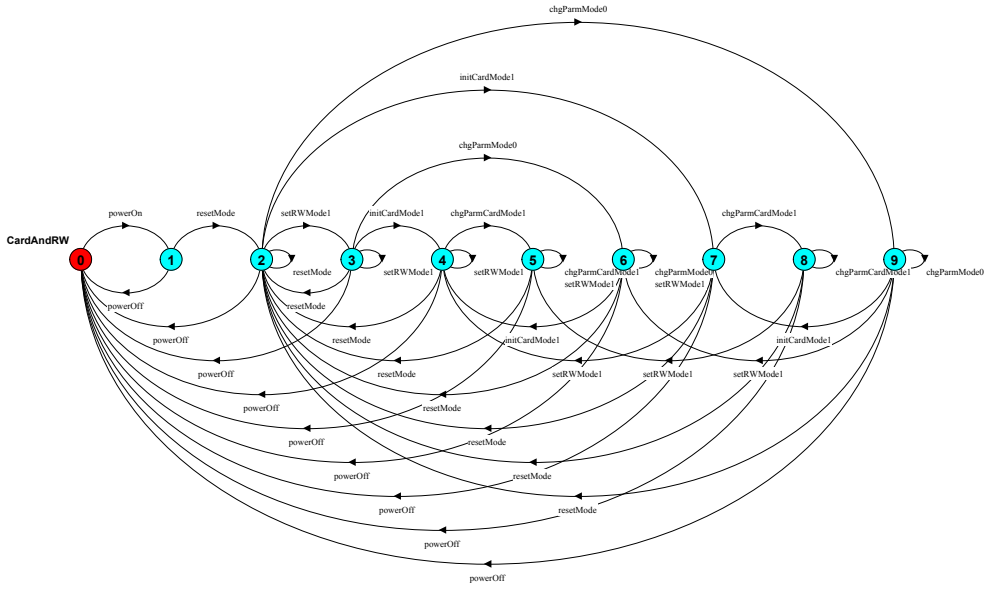

図 6.4　簡易化したカード機能とリーダ・ライタ機能を並列合成した LTS

を参照していた動作が, リーダ・ライタ機能により, 初期化したはずの変数が未初期化の状態に遷移してしまい, カード機能から参照するときに, 未初期化状態になっていないことを確認することができた.

6.4.4　シナリオの検査

仕様モデルのアクションを用いて, 想定する動作をシナリオとして FSP を用いて記述することができる. 仕様モデルとこのシナリオのモデルを並列合成し, このモデルの安全性検証とプログレスチェックを実施することで, シナリオが実行可能であることを確認することができる. シナリオに記載した順序でアクションを実行することができることを確認することができる. また, シナリオを実行することにより, デッドロックおよびプログレスエラーとなるような不正な状態にならないことを確認することができる.

本研究では, カード機能の状態が遷移した後に, リーダ・ライタ機能の状態も遷移させ, カード機能とリーダ・ライタ機能を同時に使用するようなシナリオを記述し, このシナリオを実行することにより, デッドロックおよびプログレスエラーとならないことを確認することができた.

6.4.5 ステップ 1 の結果・考察

以上の検査過程で, カード機能の状態数は 9, 遷移数は 27, リーダ・ライタ機能の状態数は 6, 遷移数は 19 となった. 両者を並列合成すると, 状態数が 28, 遷移数は 121 となった. 検査の結果, 要件に対する過剰仕様を発見することができた. これは, 図 4.9 において示したコマンド仕様について, 複数のコマンド仕様が共有している変数の初期化に関するものである. この共有変数が, 未初期化状態において参照されることがないように, 複数のコマンド仕様において初期化を行っていた. 過剰であるという認識はあったが, 初期化を行わなくてもよいという確証がなかった. しかし, 6.4.3 節で述べた共有変数の検査を行うことで, 全ての場合において, 過剰仕様であることが明らかになった.

上記の規模の状態数や遷移数であれば, モデル検査を実施しなくても, 並列合成前の各機能の状態を把握することは容易である. しかし, 各機能が並列に動作する図 6.4 に示すような状態を人が網羅的に把握することは難しい. モデル検査ツールを利用すれば, 並列合成後のモデルを得ることができる. この並列合成後のモデルが意図したモデルであるかを確認するデバッグの過程で, 並列合成前と合成後のモデルの間違いが見つかることがある. モデル検査の最終的な目的は, モデルがプロパティを満たすことを確認することであるが, それ以前のデバッグ作業により, 検査対象について深く考察することができる.

また, 上記検査手順では, モデルの作成後にプロパティを定式化した. プロパティはモデルのアクションを用いて記述する. 検査したい内容が記述できない場合は, モデルを見直さなければならない. したがって, モデルを作成する前にプロパティの内容を十分に精査する必要がある.

モデル検査は「ボタン一発の技術」というイメージがあるが, 現実には, 通常のソフトウェア開発と同様, デバッグを行いながらツールを実行し, 正しく検査できていることを確認しなければならない. 例えば, 仕様モデルが誤っているために, プロパティの検査やシナリオの検査を実施してもデッドロックやプログレスエラーとならないことがある. したがって, 意図的に誤ったプロパティやシナリオを作成して検査を実行する. そして, この結果がデッドロックやプログレスエラーとなることを確認する. これにより, 問題を検出することができる正しい仕様モデルであると確認を行いながら, 仕様モデルを作成し

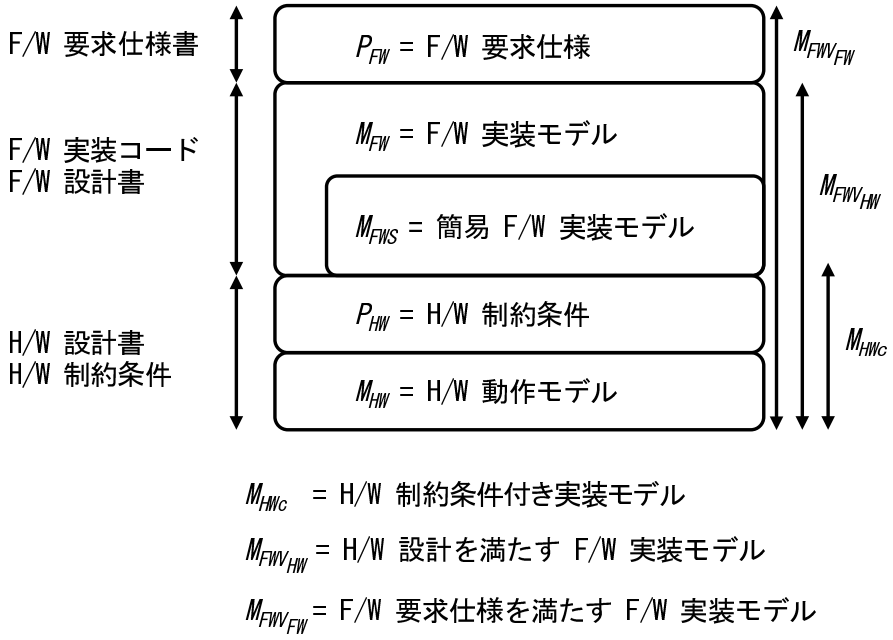

図 6.5　ステップ 2 の 各モデルの関係

ていかなければならない. つまり, 検査は「ボタン一発」で実施することができるが, その検査結果が正しいことを検査するには, 検査対象のモデルのデバッグやテストが必要である.

6.5　ステップ 2 の実施内容と結果・考察

ステップ 2 の検証対象は, 図 6.2 の S2 で示したモバイル FeliCa IC チップの H/W モジュールと F/W モジュールと, S2_EXT で示した外部環境モデルである. 特に, ステップ 2 では通信 I/F 機能を検証対象とした. 対象となる入力は, 通信 I/F の仕様が記載された F/W の設計書と, 実装コードである. ステップ 2 では特に, F/W 実装コードが, F/W 要求仕様および H/W 設計を満たしていることを検証することを目的とした. 検査項目と結果を**表 6.2** に示す. 各検査項目で作成するモデルの関係を**図 6.5** に示した. 図 6.5 の左にモデルを作成する上で入力となる文書名を記載した. 各検査項目の実施内容と各モデルの説明は各節で述べる.

表 6.2 ステップ 2 の検査項目と結果

検査項目	目的	入力	出力
1. H/W 実装モデルの作成と検査	F/W 実装モデルのプラットフォームである H/W 実装モデルの作成	H/W 設計書	(1) M_{HW} = H/W 実装モデル (2) デッドロック: なし (3) プログレスエラー: なし (4) P_S = 注意事項確認用のシナリオ
2. H/W 制約条件のプロパティの作成と検査	簡易 F/W 実装モデルを用いた H/W 制約条件の検査	(1) H/W 実装モデル (FSP) (2) H/W 制約条件一覧 (3) F/W 設計書 (4) F/W 実装コード	(1) P_{HW} = H/W 制約条件, M_{HWC} = H/W 制約条件付き実装モデル = $M_{HW} \| P_{HW}$ (2) M_{FWS} = 簡易 F/W 実装モデル (3) H/W 制約条件: 検査可能
3. F/W 実装モデルの作成と検査	F/W 実装コードが H/W 設計を満たしていることの確認	(1) H/W 制約条件付き実装モデル (FSP) (2) F/W 設計書 (3) F/W 実装コード (4) 注意事項確認用のシナリオ	M_{FW} = F/W 実装モデル, $M_{FWV_{HW}}$ = H/W 設計を満たす F/W 実装モデル = $M_{FW} \| M_{HW} \| P_{HW}$
4. F/W 要求仕様のプロパティ作成と検査	F/W 実装コードが F/W 要求仕様を満たしていることの確認	(1) H/W 設計を満たす検査済み F/W 実装モデル (FSP) (2) F/W 要求仕様書	P_{FW} = F/W 要求仕様, $M_{FWV_{FW}}$ = F/W 要求仕様に対する検査済み F/W 実装モデル = $M_{FW} \| M_{HW} \| P_{HW} \| P_{FW}$ = $M_{FWV_{HW}} \| P_{FW}$

6.5.1　H/W 実装モデルの作成と検査

H/W 設計からレジスタ単位でプロセスを作成し, 各レジスタを制御するプロセスと並列合成することで, H/W 実装モデルを作成・検査した. 表 6.2 および 図 6.5 において, H/W 実装モデルを記号 M_{HW} で表す.

H/W 実装モデルの作成には, 図 6.2 の S2_EXT で示した外部環境モデルのアクションも用いた. また, H/W 設計書には, F/W 実装者に対する注意喚起事項が記載されている. 例えば「割り込み要求発生タイミングと割り込み要因クリアが競合した場合, 割り込み要求が発生するが要因はクリアされている」などである. 注意事項確認用シナリオをモデル化し, H/W 実装モデル上で活性検証することで, H/W 実装モデルが注意喚起事項も含めて正しく作られていることを確認した. 注意事項確認用のシナリオは 表 6.2 において, P_S という記号で表した.

6.5.2　H/W 制約条件のプロパティの作成と検査

6.5.1 節では, H/W 実装モデルを作成し, H/W 設計書に記載された F/W 実装者向けの注意喚起事項が正しく H/W 実装モデルの中に含まれていることを確認した. 次の 6.5.3 節では, F/W 実装モデルを作成し 6.5.1 節において作成した H/W 実装モデルと並列合成を行い, H/W 実装モデルと F/W 実装モデルを組み合わせたときに正しく動いていることを検査する. F/W 実装モデルが, F/W 実装者向けの注意喚起事項を考慮して設計されていることを確認することは, 今回の検査の目的の一つである. 例えば, 6.5.1 節で用いた例で説明すると, 「割り込み要求発生タイミングと割り込み要因クリアが競合した場合, 割り込み要求が発生するが要因はクリアされている」において, F/W 実装者がこの注意喚起事項を考慮して, 割り込み要因発生タイミングと割り込み要因クリアが競合しない設計となっていることを H/W 実装モデルと F/W 実装モデルを並列合成することにより確認する.

この検査を行うために, 注意喚起事項に対する F/W 実装の対策を F/W 実装モデルが満たすべきプロパティとして定式化する. 今回の割り込み要因がクリアされてしまう例

では, F/W 実装者と H/W 実装者で協議し, この注意喚起事項に対する対策として,「割り込み要因が発生したタイミングでのみ, 割り込み要因をクリアする」こととした. これにより, 割り込みが発生するタイミングにおいて, F/W 実装が割り込み要因をクリアすることはないため, 問題となる競合は発生しない. 本研究では, このような F/W 実装者が守るべき, H/W 使用上の制約条件を H/W 制約条件と呼ぶ.

H/W 制約条件を H/W 実装モデルに組み込むには, 次のような手順で行った. 意図的に H/W 制約条件に違反させた簡易 F/W 実装モデルを作成して, H/W 制約条件付き実装モデルと簡易 F/W 実装モデルを並列合成した. この結果が, デッドロックとなることを確認することで, H/W 制約条件が正しくモデル化できていることを確認する. 表 6.2 および 図 6.5 において, H/W 制約条件を表したプロパティのモデルを P_{HW} の記号を用いて表し, H/W 実装モデル (M_{HW}) と並列合成を行った H/W 制約条件付き実装モデルを M_{HWC} と表す. これを並列合成の演算子 ($||$) を用いて次のように表すことができる.

$$M_{HWC} = M_{HW}||P_{HW}$$

H/W 制約条件付き実装モデル M_{HWC} が正しいことを確認するために, 意図的に H/W 制約条件に違反させた簡易 F/W 実装モデル M_{FWS} を作成した. この簡易 F/W 実装モデル M_{FWS} と, H/W 制約条件付き実装モデル M_{HWC} を次のように並列合成することで, H/W 制約条件付き実装モデルが正しく作られていることを確認した.

$$M_{HWC}||M_{FWS}$$

6.5.3 F/W 実装モデルの作成と検査

F/W 設計書と実装コードから抽出した通信 I/F の F/W 実装モデルと, H/W 制約条件付き実装モデルとを並列合成した. そして, 並列合成後のモデルに対して安全性検証と活性検証を行った. これにより, F/W 実装コードが H/W の制約条件を満たしていることを確認した.

表 6.2 および 図 6.5 において, F/W 実装モデルは M_{FW} として表した. また, H/W 制約条件付き実装モデルは M_{HWC} と表した. 6.5.2 節において述べたように, H/W 制約条

件付き実装モデル M_{HWC} は H/W 実装モデル (M_{HW}) と H/W 制約条件のプロパティ (P_{HW}) を並列合成したプロセスである. H/W 制約条件付き実装モデル M_{HWC} を次に示す.

$$M_{HWC} = M_{HW} \| P_{HW}$$

ここで, 本節で述べる F/W 実装モデルを M_{FW} という記号を用いて表す. F/W 実装モデルと H/W 制約条件付き実装モデルとを並列合成した結果は, H/W 設計を満たす F/W 実装モデルである. これを $M_{FWV_{HW}}$ という記号を用いて表す. $M_{FWV_{HW}}$ は次のように表すことができる.

$$M_{FWV_{HW}} = M_{FW} \| M_{HW} \| P_{HW}$$

H/W 制約条件付き H/W 実装モデルと F/W 実装モデルの並列合成は, 状態数が増えたため行うことができなかった. そこで, 状態数を減らすために, 各 H/W モジュールと F/W 実装モデルで共有していないアクションを内部アクションとすることで, LTSA の最小化機能を使用した.

最小化機能は弱双模倣等価の概念を適用しており, これにより仕様間の無矛盾性を検証しながら状態数を減らすことができる [MK06] . 弱双模倣等価を含む観測等価とは, 外部観測上区別不可能なプロセスを同一とみなすことである.

LTSA の最小化機能を使用するには, 隠蔽演算子 (\) および, インタフェース演算子 (@) を用いる. 例えば, 次に示す P1 プロセスと P2 プロセス について, P1 は a アクションと c アクションを隠蔽して b アクションと d アクションのみを公開したプロセスであり, P2 は b アクションと d アクションのみをインタフェースとして用いることを示している. その結果, P1 プロセスも P2 プロセスも同じ状態遷移となる.

```
P1 = (a -> b -> c -> d -> P1)
\{a, c}.

P2 = (a -> b -> c -> d -> P2)
@{b, d}.
```

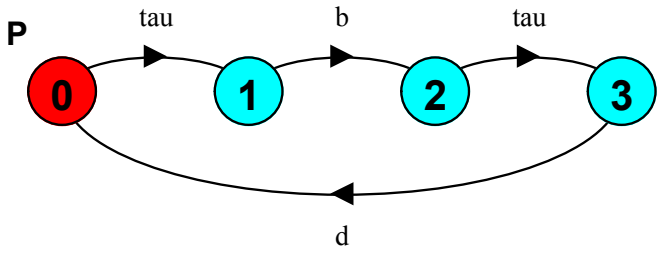

図 6.6 FSP の隠蔽演算子およびインタフェース演算子を用いた LTS

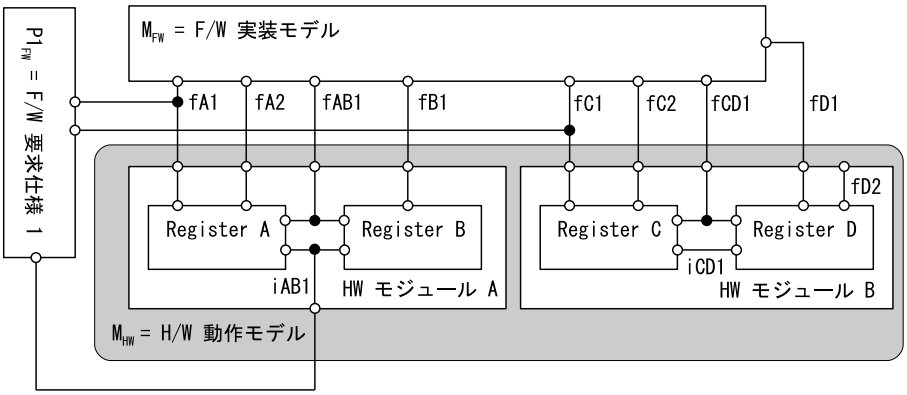

図 6.7 モデルのコンポーネント化と階層構造

P1 プロセス, P2 プロセスの LTS を**図 6.6** に示す. tau は隠蔽したアクションであり, 他のプロセスと共有しないことを示している.

この最小化機能を用いて, H/W モジュール内部でのみ共有している, F/W には非公開のアクションを隠蔽することで, 状態数を削減した. 以下では, LTSA の最小化機能を用いる際に, 検査内容に従って隠蔽するアクションを選択する方法について述べる. **図 6.7** はステップ 2 のモデルを簡略化した構造図である. H/W 実装モデルは複数のレジスタのプロセスを並列合成したものである. 各 H/W 実装モデルは, F/W 実装モデルと共有しているアクションおよび, H/W モジュール内のレジスタ間でのみ共有している内部アクションを持つ.

H/W 実装モデルと F/W 実装モデルとを並列合成する場合には, 内部アクションである iAB1 アクションと iCD1 アクションを隠蔽することで状態数を減らすことができる. iAB1 アクションは, H/W モジュールの内部で共有しているため internal を表す i を接

頭辞とし, また Register A と Register B とで共有しているため iAB と命名した. 他の H/W モジュール内部で共有しているアクションも同様である. Register A のプロセスを P_Reg_A とする. 他のレジスタも同様の命名規則とした. F/W 実装モデルを M_FW とすると, この例における H/W 実装モデルと F/W 実装モデルの並列合成において, 最小化機能を利用するための FSP 記述は次のようになる.

```
|| MIN_HW_FW_Example = (
  P_Reg_A || P_Reg_B || P_Reg_C || P_Reg_D || M_FW
)\{iAB1, iCD1}.
```

また, fD2 のような F/W 実装モデルからは使用していない H/W 実装モデルの公開アクションは, 実行不可にしておく必要がある.

このように, 複数のプロセスをコンポーネントとしてまとめ, 階層構造をとることで, 最小化機能を有効的に利用することができる.

6.5.4　F/W 要求仕様のプロパティ作成と検査

F/W 要求仕様のモデルと, F/W 実装モデル, H/W 制約条件付き H/W 実装モデルとを並列合成し, 安全性検証と活性検証を行った. これにより, F/W 設計が F/W 要求仕様を満たしていることを確認することができた.

表 6.2 および 図 6.5 において, F/W 要求仕様のモデルは P_{FW} として表した. F/W 要求仕様に対する検査済み F/W 実装モデルを, $M_{FWV_{FW}}$ とすると, 並列合成の演算子を用いて次のように表せる.

$$M_{FWV_{FW}} = M_{FW} \| M_{HW} \| P_{HW} \| P_{FW} = M_{FWV_{HW}} \| P_{FW}$$

6.5.3 節で述べた状態数を削減する方法と同様に, LTSA の最小化機能を利用する. F/W 要求仕様から H/W モジュールの内部アクションを参照している場合は, 参照している内部アクションを F/W 要求仕様と共有する必要がある. 図 6.7 では, 内部アクション iAB1 を参照する F/W 要求仕様 1 の検査例を示す. ここで, F/W 要求仕様 1 とは, 複数ある

要求仕様のうちの一つという意味である. それぞれの F/W 要求仕様の内容によって, 共有するアクションは異なる. そのため, 最小化機能を利用するには, 要求仕様ごとに隠蔽対象のアクションを選択しなければならない. この F/W 要求仕様 1 の場合は, iCD のみを内部アクションとして最小化することで, iAB1 も含めた要求仕様が検査できる.

具体的には, Register A のプロセスを P_Reg_A とし, 他のレジスタも同様の命名規則とする. F/W 実装モデルを M_FW とし, F/W 要求仕様 1 のプロパティを P_FW とすると, この例における H/W 実装モデルと F/W 実装モデル, F/W 要求仕様モデルの並列合成において, 最小化機能を利用するための FSP 記述は次のようになる.

```
|| MIN_HW_FW_Example = (
  P_Reg_A || P_Reg_B || P_Reg_C || P_Reg_D || M_FW || P_FW
)\{iAB1}.
```

6.5.5 ステップ 2 の結果・考察

モデル検査を始めるにあたり, モデルの精度・抽象度が問題となった. 検査項目に対して十分精度の高い H/W 実装モデルであれば問題とはならないが, 精度の高さは状態数・複雑度の増加とトレードオフの関係にある.

抽象化は検査の網羅性に影響を与えるためモデリング以前の判断が難しい. 本適用では, システムの基盤となる, 最も精度が高い H/W 設計から作成したレジスタモデルを基本とした. このときに, 最初のアプローチでは, レジスタ値の取りうる組み合わせを洗い出すことで H/W モジュールの状態遷移を作成した. しかし, 一度に記述する範囲が広範囲なため, 作成する状態数が多くなり, 状態遷移の正当性・網羅性を判断することができなかった. そこで, 次のアプローチとして, 最も粒度の小さいレジスタ仕様を基に, 状態数が 2 〜 6 の小規模なプロセスを作成し, これらを並列合成することで仕様全体を記述した. これにより, 正当性・網羅性の判断が可能な範囲でプロセスを作成することができた.

以上の検査過程で, 通信 I/F を構成する 4 つのモジュールの状態数と遷移数は以下の

とおりとなった．2 つのタイマモジュール (それぞれのプロセス数は 4) において, 各々の状態数は約 30, 各遷移数は約 150 となり, 2 つの通信 I/F モジュール (それぞれのプロセス数は 15) において, 各々の状態数は約 10,000, 遷移数は 370,000 となった．各モジュールのモデル化の過程で, 要件に対する過剰なレジスタ仕様を発見することができた．しかし, 全モジュールを合成することはできなかった．これは, 状態数が, 検証ツールを動かしているコンピュータのメモリに収まらないために, 検証ツールが現実的な時間で計算を終えることができなかったためである．

そこで, 6.5.3 節と 6.5.4 節に示した LTSA の最小化機能を利用し, 要求仕様の検査が可能となるまで状態数を削減した．対象のアクションが異なるため, 要求仕様ごとに状態数を減らす必要がある．モデルに不具合がある状態では, 反例を用いたデバッグ作業が必要となり, 検査工数が課題となった．

モデル作成・デバッグ工数の多くを, 検査対象である F/W 実装モデルではなく, H/W 実装モデルに費やした．H/W のアーキテクチャ検討段階から, F/W 設計者と H/W 設計者でモデルを共有することで, 開発プロセス全体でバランス良く利用可能となる．

検査した要求仕様は数十項目であり, 十分とはいえない．状態数が増加した場合のデバッグ手法や, コンポーネント化による, システマティックな状態数の削減に取り組み, 検査工数の課題を解決し, 開発プロセス全体での活用を目指したい．

6.6　動的な振る舞いに関するモデル検査の適用のまとめ

6.6.1　ステップ 1

ステップ 1 では, 検査対象のモデル化, 要求項目の抽出・検査, シナリオの確認などをコンパクトな仕様を対象として行った．専門家の助けを借りれば, モデル検査ツールの導入・学習と, 既存のコンパクトな仕様のモデル検査は容易に行うことができる．

たとえ簡素な仕様といえども, 機械支援なしに網羅的検証は容易でない．本プロジェクトでの適用においては, 要件に対する過剰仕様を発見することができた．これは, 図 4.9 において示したコマンド仕様について, 複数のコマンド仕様が共有している変数の初期

化に関するものである．この共有変数が, 未初期化状態において参照されることがないように, 複数のコマンド仕様において初期化を行っていた．過剰であるという認識はあったが, 初期化を行わなくてもよいという確証がなかった．しかし, 6.4.3 節で述べた共有変数の検査を行うことで, 全ての場合において, 過剰仕様であることが明らかになった．また, 上記モデル検査の全体像と有用性を, 開発プロジェクト内において認識することができた.

6.6.2 ステップ 2

ステップ 2 では, ステップ 1 よりも複雑な仕様を対象としてモデル検査を行った.

ステップ 1 におけるモデル検査の導入・学習は, ステップ 2 の開始において有効であった．一方で, ステップ 2 では状態数の爆発に対する工夫が必要となった．モデルをコンポーネント化し, 階層構造を導入することで, LTSA の最小化機能を有効的に利用した．これにより, 状態数をある程度減らすことができた.

仕様を小さなモデルに分割し, 個別にモデル化・可視化を行うことができるので, ステップ 1 同様有効であった．更に, 人の認識能力を超える小さなモデルを合成した仕様全体の把握は, ツールの可視化機能の助けを借りても困難であるが, 複雑な状態遷移の様々な断面の可視化により, 仕様に関する新たな知見が得られる効果がある.

6.6.3 まとめ

モデル検査は, 従来の頭の中での検討や, 自然言語や UML などの記法を用いた仕様記述では困難な, 仕様の検査を行うことができ有用である.

また, 導入には専門家の助言が必要である．特に, 仕様をモデル化する上では, 状態を爆発させないことがポイントとなるが, このためには様々なノウハウが必要であり, 独学では難しい.

本研究は, 6.1 節において定めた段階的導入のステップ 2 の途中である．ステップ 1 において, 既存の仕様の評価を行うことで, ツールの有用性を確認することができた．つま

り, 6.1 節において述べた目的 1 と目的 2 を達成することができた. ステップ 2 において, 目的 3 として掲げた, 実装コードと実装コードが動作する環境の制約条件を満たしていることは, 6.5.3 節において述べた方法により, 一部確認することができた. また, 目的 4 の評価仕様の一部を 6.5.4 節において述べた方法により確認することができた. 目的 3 と目的 4 に関して, それぞれの一部にとどまっている理由は, 6.5.3 節と 6.5.4 節において述べたように, 検査対象のモデルの状態数が増えたため, 現実的な時間内にツールが検査を終えることができなくなったためである. そのため, 検査する制約条件や評価仕様ごとに, LTSA の最小化機能を利用して状態数を減らした. しかし, それぞれの制約条件や評価仕様ごとに最小化機能を用いて状態数を削減したため, 1 つの検査にかかる作業量が多くなり, 検査工数が課題となった. その結果, 目的 3 と目的 4 の全ての検査を終えることができなかった. ステップ 2 の後半以降である, 状態を爆発させないノウハウの蓄積や, モデル検査と形式仕様記述の組み合わせは今後の課題である.

第7章

結論

本論文では, 組織的に開発を進める必要があるソフトウェア開発現場において, 上流工程の文書を中心に多くの問題を抱えており, 品質の高いシステムを組織的に効率よく開発することができる手法として, フォーマルメソッドが注目されていることを述べた. フォーマルメソッドの 3 段階の適用レベルにおいて, 本研究が扱うのは, 形式仕様記述を行うレベル 0 である. レベル 0 の段階は, フォーマルメソッドの利用において, 証明までは行わないが, 数学的な記法を用いて厳密な仕様を記述する段階である. このレベル 0 の段階において, 筆者らはモバイル FeliCa IC チップ開発を通して形式仕様記述手法の適用の効果を得ることができた. しかし, 産業界は多くの問題を抱えているにもかかわらず, フォーマルメソッドが, ありふれた「当然の技術」として産業界に浸透していないという Parnas の指摘を紹介した. そこで, 本研究では「当然の技術」として産業界に浸透するための課題を分析した. 分析において, 適用レベルをレベル 0 からレベル 0-a とレベル 0-b の 2 つに分けた. レベル 0-a は, 形式仕様の記述者が, 形式仕様記述手法を用いて仕様を厳密に記述し, 早期に仕様の曖昧さに起因する不具合を見つけることができるという段階とした. レベル 0-b は, 形式仕様の記述者以外である, ドメインエンジニアや設計者, 実装者, 評価者が, 形式仕様記述を参照し, 形式仕様記述に基づいて設計と実装とテストを行う適用の段階とした. 開発プロジェクトの中で形式仕様記述を活用する人は, レベル 0-a においては, 形式仕様の記述者のみの限られた人であるが, レベル 0-b に

おいては, ドメインエンジニア, 設計者, 実装者, 評価者といったほぼ全ての工程の人たちとなる. これらの人たちが形式仕様記述手法をあらかじめ習得していることは希であり, 形式仕様記述手法の専門家ではない人が, 形式仕様記述を読み, 理解し, 活用していく必要がある. つまり, 形式仕様記述手法の専門家ではない人が, 読みやすいと感じ, 形式仕様記述手法を利用することにメリットを感じるような, 形式仕様を記述するための技術と, 活用するための技術が必要である. この課題を解決することで, 形式仕様記述手法がありふれた「当然の技術」として, 開発プロジェクトにおいて広く活用されるようになり, さらには, 産業界においても広く活用されることにつながると考える.

本研究では, 前述のレベル 0-b の段階に到達するために必要な, 形式仕様を記述するための技術と, 活用するための技術を研究課題として設定した. 記述するための技術として, ドメインエンジニア, 設計者, 実装者, 評価者といった様々な役割を担った開発者にとって, 理解容易性に優れた仕様を記述する技術が必要である. 活用するための技術としては, 従来からある既に開発現場に浸透しているレビューやテスト手法といった検証技術を主体として, その中で形式仕様記述手法を活用していく技術が必要である.

本研究では, FeliCa IC チップ開発への形式仕様記述手法の適用事例を通して得られた課題と, 形式仕様を記述するための技術, 形式仕様を活用するための技術を示し, 形式仕様記述手法が実践的手法として産業界において広く利用されるような「当然の技術」たる所以を示した. 開発ドメイン, 適用レベル, 使用した記述言語は限定されたものである. また, 本論文で述べた課題は一部ではある. しかし, 「当然の技術」として形式仕様記述手法を適用する際に必要不可欠な適用技術の設計・構成法に関する知識不足を本論文が補うものであると考える.

本研究においては, 次の 3 つの課題を扱った.

(1)　実行可能仕様における仕様と設計の分離に関する課題

(2)　レビューとテストの用途として, 形式仕様記述を利用する場合の課題

(3)　形式仕様記述手法では取扱いが容易でない動的な振る舞いに関する課題

(1), (2) は, レベル 0-b の適用段階において, 形式仕様を記述するための課題と, 活用す

るための課題である. (3) は, レベル 0-b の議論とは別の, 仕様策定後の設計工程において課題となった, 形式仕様記述手法では取扱いが容易でない動的な振る舞いに関する課題である.

(1) に関する関連研究として, Hayes と Joens の 「実行可能性の影響」と Jackson の「実装の影響」について考察した. Hayes と Jones の「実行可能性の影響」と Jackson の「実装の影響」は, 仕様記述が複雑になる点と, 設計者を過剰に制約する点が課題である.

Hayes と Joens の「実行可能性の影響」に関する課題は, 次のようなものであった. 仕様書の役割は, 何を作るべきかという「解決すべき問題」を各工程に明確に「伝えること」である. 実行可能仕様により, この「伝えること」を目的とした記述と, 仕様アニメーションにより「動かすこと」を目的にした記述が混在することになる. 2 つの記述が混在することにより, 本来は「解決すべき問題」として What を規定する仕様に, 「問題の解決方法」として How の記述が多く含まれてしまう. これにより, 設計者を過剰に制約し, 実装の選択肢を狭めてしまう可能性がある. また, How の記述が多く含まれることにより, 仕様の記述が複雑になってしまう傾向がある.

理解容易性に優れた仕様を記述するためには, 「伝えること」を目的とした記述を明確にする必要があり, それを妨げる「動かすこと」を目的とした記述が課題となる. 一方, 従来からある既に開発現場に浸透しているテスト手法を用いて, 仕様アニメーションにより実行可能仕様を活用するためには, 「動かすこと」を目的とした記述が必要である.

本研究では, この課題を解決する仕様記述スタイルを提案した. まず, 次の 2 つの観点から仕様記述スタイルを議論した. 操作的な記述と宣言的な記述の観点と, 仕様アニメーションによる動作検証の作業工数の観点である. この中で, 仕様伝達部と非伝達部からなる仕様記述スタイルを提案した. 次に, 具体的な仕様記述例を用いて, 提案の仕様記述スタイルを評価し, 次の 3 つのことを示した.

- 「解決すべき問題」である What を記述するために, 仕様を宣言的な記述箇所に記述することができること.
- 仕様伝達部と非伝達部を分離することで, 理解容易性に優れた仕様を記述すること

ができること.

- 仕様アニメーションによる動作検証時に課題となる, 評価工数の課題が解決できること.

次に, Jackson の「実装の影響」として, 仕様と設計の分離に関する課題について議論した. 仕様と設計の分離とは, 「解決すべき問題」を定義した仕様と, 「問題の解決方法」を定義した設計を分けて考えるということである. 一般に, 仕様策定工程では, 何を作るべきかという「解決すべき問題」を検討し, 設計工程では, 「問題の解決方法」を検討することで, 仕様策定者と設計者が分業して組織的に開発を行うことができると言われている. 仕様と設計の分離は広く言われていることではあるが, 実際に, 仕様と設計を分離することは容易ではない. 仕様と設計の境界が曖昧であるために, 形式仕様記述から仕様として規定している範囲と, 形式仕様記述には記述してはあるが, 設計者が自由に規定できる範囲を, 設計者が読み取ることが難しいという課題がある. 仕様が「実装の影響」を受けないために, Jackson の提案は, 「もし仕様のなかでなんらかの状態を記述しなければならないとすると, それは, 機械の状態ではなく, すべて [適用領域] の状態でなければならない.」というものであった. しかし, 本研究では, 仕様策定工程において, 機械の内部状態のデータ構造をある程度, 限定することができる場合を前提とし, 開発効率の観点から Jackson の提案する手法とは別の方法を提案した. 仕様策定工程において, 機械の内部状態のデータ構造をある程度, 限定することができる場合は, 仕様で定義するデータ構造と, 設計で定義するデータ構造の間に関連性を許容した方が開発効率は向上するためである. この上で, 「実装の影響」を回避する方法を提案した.

本研究では, 仕様において定義するデータ構造と設計において定義するデータ構造に着目し, 「実装の影響」の課題を解決する形式仕様記述の設計・構成法を提案した. それは, 仕様策定者が決める「仕様の型」と, 設計者が決める, 「隠蔽対象の型」に分類することで, 「隠蔽対象の型」を「隠蔽関数」により隠蔽する方法である. この手法の評価において, 提案の手法を用いた場合と, 用いなかった場合の記述例を比較し, 本提案手法の有効性を示した.

(1) に掲げた課題に対する考察として, 提案した手法を FeliCa IC チップ開発プロジェクトにおいて適用した結果について述べた. 操作的な記述箇所と宣言的な記述箇所の共通部分を抜き出して関数化することで, 記述量の増加を抑える方法など, 適用における知見を考察した.

(2) のレビューとテストの用途として, 形式仕様記述を利用する場合の課題は次のようなことであった. まず, レビューにおける課題は, 次のような課題であった. 形式仕様記述手法の専門家ではない, ドメインエンジニアや仕様策定者, 設計者, 実装者, 評価者といった, 様々な読者が形式仕様をレビューに用いる場合を考えた. この時, レビューの目的によって, 読者が必要とする仕様の詳細度が異なる. 例えば, ドメインエンジニアが, 仕様が要求を満たしていることを確認するレビューと, 実装者が, 実装コードが仕様を満たしていることを確認するレビューとでは, 必要となる仕様の詳細度は異なる. ドメインエンジニアが必要とする仕様の詳細度に比べ, 設計者と実装者と評価者はより詳細度の細かい仕様を必要とすることが多い. このような仕様の詳細度の分析から, 様々なレビューの目的を持った読者にとって, 読みやすい仕様書を次のように考えた.

- 概要と詳細を階層化して明確に区別して記述してあり, 概要のみを読むことで, 仕様の全体像を理解することができる仕様記述.
- 概要から詳細へと即座にたどることができる仕様記述.
- 形式仕様記述手法の専門家ではない人であっても, 仕様を読み進めることができるように, 特に概要の記述は簡潔な構造を持つ仕様記述.

このような仕様であれば, ドメインエンジニアが仕様をレビューする場合は, まず概要を読み進め, 必要に応じて概要から詳細を読むことができる. 設計者と実装者と評価者は, 概要を把握した上で, 概要と詳細を往き来しながら仕様を読み進めることができる. 本研究では, 以上のような課題を解決する仕様記述フレームワークを提案した.

ディシジョンテーブルと仕様記述を対応づけた仕様記述フレームワークを用いることで, 概要の仕様を簡潔な構造で表現し, 形式仕様記述手法の専門家ではない人に対して, 理解容易性に優れた仕様記述を提供することができた. また, ラベル付き条件を用いるこ

とで, 概要と詳細を階層化し, 概要から詳細へと即座にたどることができる仕様記述を提供することができた. これらの効果を評価するために, ラベル付き条件を用いた場合と, 用いない場合の仕様記述を比較し, レビュー用途における理解容易性と運用・保守工程における保守性を評価した.

次に, 仕様に基づくテストにおける課題は, 次のような課題であった. 仕様に基づくテストとは, 実装コードが仕様を満たしていることを確認するテストのことであった. 仕様に基づくテストの工程において, 形式仕様記述手法を活用することで, メリットを感じることができるように, 本研究では次の 2 つの課題を設定した.

- まず, 仕様に基づくテストにおいて前提とする品質目標を決めた. その上で, 品質目標を満たす体系的なテスト項目の作成方法を研究課題とした. これは, 体系的にテスト項目を作成することができるような, 形式仕様記述の設計を行わなければならないことでもあった.

- テスト項目とテストスクリプトが品質目標を満たしていることを, 形式仕様記述を用いた機械的な処理により確認する方法を研究課題とした.

本研究において定めた仕様に基づくテストの品質目標と, 体系的にテスト項目を作成するためのテスト設計および, そのための仕様記述フレームワークを提案した. そして, テスト項目とテストスクリプトが, 品質目標を満たしていることを確認するためのログ出力による機械処理の方法を提案した.

これまで述べてきた形式仕様記述手法に関する設計・構成法の提案は, 第 1 世代の開発において明らかになった課題を, 第 2 世代の開発において解決していった内容であった. 第 1 世代の開発とは, 2004 年から 2007 年において, フェリカネットワークス株式会社において行ったモバイル FeliCa IC チップ開発のことであり, 第 2 世代の開発とは, 2007 年中頃から約 4 年間, ソニー株式会社にて行った FeliCa IC チップ開発のことであった. 形式仕様記述手法の適用対象は, 図 4.4 に示した「入力」と「入力時の内部状態」, 「出力」と「出力時の内部状態」の関係を記述した仕様である. これは, 一般に, 静的な仕様である機能仕様と呼ばれるものである. 第 1 世代の開発において, 形式仕様記述手法を

用いることにより, この静的な仕様である機能仕様を早期に検証し, 不具合を摘出することができた. 一方で, 同時並行に動作する動的な振る舞いに関する検証方法が, 仕様策定後の設計工程において課題となった. これは, 別の見方をすると, 静的な仕様である機能仕様の品質を早期に確保していたため, 動的な振る舞いに関する検証方法を課題とすることができたとも言える. そこで, (3) に掲げた形式仕様記述手法では取扱いが容易でない動的な振る舞いに関する課題を解決するために, 第 1 世代のモバイル FeliCa IC チップ開発の設計工程において, 同時並行に動作する動的な振る舞いに関する検証方法を検討した.

フォーマルメソッドの 1 つであるモデル検査を設計検証に導入した. モデル検査ツールとしては, LTSA を用いた. 2 つのステップに分けて導入を行い, ステップ 1 では, コンパクトな仕様を対象にモデル検査の学習と有用性の確認を行い, ステップ 2 では, ハードウェアのレジスタ仕様とソフトウェア仕様からなるステップ 1 よりも複雑な仕様を対象にモデル検査を適用した. ステップ 1 におけるモデル検査の導入・学習は, ステップ 2 の開始において有効であった. 一方で, ステップ 2 では状態数の爆発に対する工夫が必要となった. モデルをコンポーネント化し, 階層構造を導入することで, LTSA の最小化機能を有効的に利用した. これにより, 状態数をある程度減らすことができた. 仕様を小さなモデルに分割し, 個別にモデル化・可視化を行うことができるので, モデル検査はステップ 1 同様有効であった. 更に, 人の認識能力を超える小さなモデルを合成した仕様全体の把握は, ツールの可視化機能の助けを借りても困難であるが, 複雑な状態遷移の様々な断面の可視化により, 仕様に関する新たな知見が得られる効果が分かった.

フォーマルメソッドの最終的な目標の 1 つは, Correctness by construction の考え方であり, 要求, 仕様, 設計, 実装と各ステップにおいて正しさを保証し, 正しさを保証しながら開発のステップを進んでいくことである. しかしながら, 現状は誰もがすぐに Correctness by construction の考え方に従って開発を行える環境ではない. 各々の開発現場においては, 既に蓄積してきた検証技術があり, その検証技術を主体としてフォーマルメソッドを用いた開発法を議論するすることで, 開発現場の問題が解決し, 工学が発展していくと考える. 本論文では, 一部ではあるが, 従来からある検証技術の中にフォーマルメソッドを

適用した事例について述べ, 適用に必要不可欠な設計・構成法の提案を行い議論した. ソフトウェア開発において, フォーマルメソッドも銀の弾丸ではないが, 開発現場における品質問題を論理的に解決する「当然の技術」として適用可能な手法である. 今後, さらにフォーマルメソッドの設計・構成法や活用方法に関する議論が活発に行われることを期待する.

謝辞

本論文の執筆を勧めて下さり終始熱心にご指導くださり, またフォーマルメソッドの導入に際して私達を常に導いて下さった九州大学大学院システム情報科学研究院情報知能工学部門 荒木啓二郎教授に深く感謝致します.

本論文の調査委員として貴重なご助言とご指導を賜りました, 九州大学大学院システム情報科学研究院情報知能工学部門の鵜林尚靖教授, 九州大学大学院システム情報科学研究院情報知能工学部門の日下部茂准教授に, 心から御礼申し上げます.

形式仕様記述手法, モデル検査の導入にあたり, 法政大学コンピュータ科学科情報科学研究科の劉少英教授, 国立情報学研究所アーキテクチャ科学研究系の中島震教授, SCSK 株式会社の佐原伸氏, 有限会社デザイナーズ・デンの酒匂寛氏, 株式会社エクスモーションの藤倉俊幸氏, 独立行政法人 情報処理推進機構ソフトウェア・エンジニアリング・センターの新谷勝利氏, 九州大学大学院システム情報科学研究院情報知能工学部門の大森洋一助教に, 多くの有益なご助言を頂きました. 深謝いたします.

ソニー株式会社の業務執行役員 SVP 大塚博正氏, FeliCa 事業部長 川西泉氏, FeliCa 事業部プラットフォーム開発部統括部長 吉田宏氏, FeliCa 事業部プロダクト＆サービス部統括部長 坂本和之氏は, 研究を深くご理解くださり, 本論文の作成を支援してくださいました. 深く感謝致します.

共同研究者でもあるフェリカネットワークス株式会社コアテクノロジー開発部開発 2 課統括課長 栗田太郎博士は, 形式仕様記述手法をはじめる貴重なきっかけを与えてくださり, 有益なご討論とご助言を頂きました. 感謝いたします.

本研究は, 開発プロジェクトの全ての人たちの努力と協力がなければ, 実現できません

でした. 感謝いたします.

最後に, 本研究の遂行と論文の執筆にあたり, 様々な面で協力し, また励ましてくれた妻典子に感謝します.

参考文献

[Abr96] J.R. Abrial. *The B-book: Assigning Programs to Maanings* . Cambridge University Press, 1996.

[Abr06] J.R. Abrial. Formal methods in industry: Achievements, problems, future. *Proceedings of the 28th international conference on Software engineering*, pp. 761–768, 2006.

[BH95] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, Vol. 28, No. 4, pp. 56 –63, 1995.

[BH06] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods... ten years later. *IEEE Computer*, Vol. 39, No. 1, pp. 40 – 48, 2006.

[BJ78] D. Bjørner and C.B. Jones. *The Vienna Development Method: The Meta-Language*, Vol. 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[CWA+96] E.M. Clarke, J.M. Wing, R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, B. Steven, P. Wolper, J. Woodcock, and P. Zave.

Formal methods: State of the art and future directions. *ACM Computing Surveys*, Vol. 28, pp. 626–643, 1996.

[FLM+05] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-oriented Systems*. Springer-Verlag, 2005.

[Flo67] R.W. Floyd. Assigning meanings to programs. Proc. *Symposia in Applied Mathematics*, Vol. 19, pp. 19–31. American Mathematical Society, 1967.

[GHJ+93] J. V. Guttag, J. J. Horning, S. J. Garland K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and tools for formal specification*. Springer-Verlag, 1993.

[Gog79] Joseph A. Goguen. Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs. Proc. *Mathematical Studies of information processing*, Vol. 75 of *Lecture Notes in Computer Science*, pp. 425–473. Springer-Verlag, 1979.

[Hal96] A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, Vol. 13, No. 2, pp. 66–76, 1996.

[HJ89] I. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, Vol. 4, pp. 330–338, 1989.

[HK91] I. Houston and S. King. CICS project report experiences and results from the use of Z in IBM. Proc. *VDM'91 Formal Software Development Methods*, Vol. 551 of *Lecture Notes in Computer Science*, pp. 588–596. Springer-Verlag, 1991.

[HO09] D. Hoover and A. Oshineye. *Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman*. O'Reilly Media, Inc., 2009.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Vol. 12, pp. 576–580, 1969.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, Vol. 21, No. 8, pp. 666–677, 1978.

[IEC10] IEC TC/SC 65A. *IEC 61508 – Functional safety of electrical / electronic / programmable electronic safety-related systems (7 Parts)*, second edition. 2010. Available online at http://www.iec.ch/.

[ISO96] ISO/IEC JTC 1/SC 22/WG 19. *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language.* 1996.

[ISO09a] ISO/IEC JTC 1/SC 27. *Information technology – Security techniques – Evaluation criteria for IT security – Part 1:Introduction and general model.* 2009.

[ISO09b] ISO/IEC JTC 1/SC 27. *Information technology – Security techniques – Evaluation criteria for IT security – Part 2: Security functional components.* 2009.

[ISO09c] ISO/IEC JTC 1/SC 27. *Information technology – Security techniques – Evaluation criteria for IT security – Part 3:Security assurance components.* 2009.

[Jac95] M. Jackson. *Software Requirements & Specifications.* ACM Press, 1995.

[Jon90] C.B. Jones. *Systematic Software Development using VDM*, second edition. Prentice Hall, 1990.

[KCN08] T. Kurita, M. Chiba, and Y. Nakatsugawa. Application of a Formal Specification Language in the Development of the "Mobile FeliCa" IC Chip Firmware for Embedding in Mobile Phone. Proc. *FM 2008 : Formal Methods*, Vol. 5014 of *Lecture Notes in Computer Science*, pp. 425–429. Springer-Verlag, 2008.

[KN09] T. Kurita and Y. Nakatsugawa. The Application of VDM to the Industrial Development of Firmware for a Smart Card IC Chip. *International Journal of Software and Informatics*, Vol. 3, No. 2–3, pp. 343–355, 2009.

[Lef97] D. Leffingwell. *Calculating your return on investment from more effective requirements management.* Rational Software Corporation, 1997. Available online at http://www.ibm.com/developerworks/rational/library/347.html.

[MDL87] H.D. Mills, M. Dyer, and R.C. Linger. Cleanroom software engineering. *IEEE Software*, Vol. 4, No. 5, pp. 19 –25, 1987.

[Mey85] B. Meyer. On Formalism in Specifications. *IEEE Software*, Vol. 2, No. 1, pp. 6 –26, 1985.

[Mey00] B. Meyer. *Object-Oriented Software Construction*, second edition. Prentice Hall, 2000.

[Mid89] C.A. Middelburg. VVSL: A language for structured VDM specifications. *Formal Aspects of Computing*, Vol. 1, No. 1, pp. 115–135, 1989.

[MK06] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*, second edition. Wiley, 2006.

[MST11] G.J. Myers, C Sandler, and T.M Thomas. *The Art of Software Testing*, third edition. Wiley, 2011.

[NKA10] Y. Nakatsugawa, T. Kurita, and K. Araki. A Framework for Formal Specification Considering Review and Specification-Based Testing. *Proc. TENCON 2010 - 2010 IEEE Region 10 Conference*, pp. 2444 –2448, 2010.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, Vol. 15, No. 12, pp. 1053–1058, 1972.

[Par79] D.L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, pp. 128 – 138, 1979.

[Par10] D.L Parnas. Really Rethinking 'Formal Methods'. *IEEE Computer*, Vol. 43, No. 1, pp. 28–34, 2010.

[PCW85] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 259 – 266, 1985.

[Roy70] W.W. Royce. Managing the development of large software systems: Concepts and techniques. Proc. *WESCON*, August 1970. reprinted in the Proceedings 9th International Conference on Software Engineering, 1987.

[SBB+02] F. Shull, V. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. *Proceedings of the 8th IEEE Symposium on Software Metrics, 2002*, pp. 249–258, 2002.

[SCS] SCSK 株式会社. VDM information web site. http://www.vdmtools.jp/.

[SCS10] SCSK 株式会社. VDM++言語マニュアル ver.1.0. SCSK 株式会社, 2010.

[Wik] Wikipedia. Formal methods. http://en.wikipedia.org/wiki/Formal_methods.

[ZJ97] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 1, pp. 1–30, 1997.

[荒木 08] 荒木啓二郎. フォーマルメソッドの過去・現在・未来 ー適用の実践に 向けてー. 情報処理, Vol. 49, No. 5, pp. 493–498, 2008.

[栗田+06] 栗田太郎, 中津川泰正, 太田豊一. 携帯電話組み込み用 “モバイル FeliCa IC チップ” 開発における形式仕様記述手法の適用とその効果. ソフトウェアシンポジウム 2006 論文集, pp. 49–66, 2006.

[栗田+09] 栗田太郎, 中津川泰正, 荒木啓二郎. 形式手法適用の実際と教訓 ー「形式手法の十戒」に照ら し合わせてー. ソフトウェア工学の基礎 XVI, 日本ソフトウェア科学会 FOSE2009, pp. 25–35. 近代科学社, 2009.

[佐原+07] 佐原伸, 荒木啓二郎. オブジェクト指向形式仕様記述言語 VDM++支援ツール VDMTools. コンピュータソフトウェア, Vol. 24, No. 2, pp. 14–20, 2007.

[杉山+07] 杉山寛和, 栗田太郎. 携帯電話と FeliCa を融合したモバイル FeliCa 技術. 情報処理, Vol. 48, No. 6, pp. 561–566, 2007.

[中島 07] 中島震. ソフトウェア工学の道具としての形式手法. ソフトウェアエンジニアリング最前線, ソフトウェアエンジニアリングシンポジウム 2007 (SES2007), pp. 27–48. 近代科学社, 2007.

[中津川+06] 中津川泰正, 栗田太郎, 米田篤生, 谷川正和, 守屋繁. 携帯電話組み込み用 “モバイル FeliCa IC チップ” 開発におけるモデル検証手法の導入と課題. 組込みシステムシンポジウム 2006 論文集, pp. 58–62, 2006.

[中津川+09] 中津川泰正, 栗田太郎, 荒木啓二郎. FeliCa IC チップ開発における仕様記述フレームワークの 構築. ソフトウェア工学の基礎 XVI, 日本ソフトウェア科学会 FOSE2009, pp. 13–24. 近代科学社, 2009.

[中津川+10] 中津川泰正, 栗田太郎, 荒木啓二郎. 実行可能性と可読性を考慮した形式仕様記述スタイル. コンピュータソフトウェア, Vol. 27, No. 2, pp. 130–135, 2010.

[藤倉 06] 藤倉俊幸. 組み込みソフトウェアの設計 & 検証 – 割り込み動作から RTOS を使った設計，ツールによる動作検証まで. CQ 出版, 2006.

[松尾 07] 松尾隆史. 非接触 IC カード技術 FeliCa. 情報処理, Vol. 48, No. 6, pp. 556–560, 2007.

索引