NAIST-IS-DT9561027

博士論文

Zによるシステム仕様記述のための変換技術に関する基礎的研究

張 漢明

1999年2月5日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

張 漢明

審査委員：　福田 晃 教授
鳥居 宏次 教授
関 浩之 教授

Zによるシステム仕様記述のための変換技術に関する基礎的研究*

張 漢明

内容梗概

本研究の目的は，ソフトウェア開発におけるシステム仕様記述の作業を支援することである．ソフトウェア開発は複数の人間の協調作業であり，ソフトウェア開発における成功の鍵は，顧客と開発者の間，また開発者どうしの間において，開発するシステムが何であるかを正しく理解し共有することである，と言っても過言ではない．システムに対する理解を共有するための中心的な役割を果たすものがシステム仕様記述である．信頼性の高いシステム仕様記述を効率良く構築するためには，対象システムのモデル化，およびシステム記述の分析を支援するための技術の開発が必要である．

本研究では，システムの最適なモデルと仕様記述は試行錯誤の結果得られることに着目し，仕様記述の試行錯誤の過程を変換と捉え，その過程を形式仕様記述言語Zを用いて定式化することにより，システムのモデル化支援および記述の分析支援を図る．システム仕様記述は，基本的には，システムが保持すべき内部状態と，システムが必要な機能を記述することである．より良いシステム記述を構築するためには，システムの内部状態のモデル化と仕様記述の構造が重要である．また，システムの内部状態が保持すべき制約を明確に記述することが，システム分析において重要な役割を果たす．更に，モデル化の作業を支援するために，直観的で人間になじみやすい図式表現の導入が望まれる．

*奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文, NAIST-IS-DT9561027, 1999年2月5日.

本論文では，Z仕様記述における仕様記述変換手法を提示する．まず，システム内部状態のモデルとして2項関係に着目し，Zにおける2項関係モデルの中で最も特徴的なモデルである全域関数，部分関数および関係の間の仕様記述の変換手法を示す．次に，仕様記述の構造の変換として階層的な構造を持たないシステムの仕様記述から，階層的に機能分割された仕様記述に変換する手法を示す．システム状態不変条件の記述の変換では，操作の仕様記述に操作的に表現されているシステムの内部状態に対する制約を，システム状態不変条件として変換する手法を示す．最後に，図式表現と形式的な表現の間の変換として，CASE(Computer-Aided Software Engineering) ツールで一般的に用いられている構造化ダイアグラムからZ仕様記述に変換する手法を示す．

仕様記述変換技術は，仕様記述者が仕様記述を変更するときの過程を定式化することに相当し，仕様記述の内容を変えることなく仕様記述を系統的に変更する手法に対する基礎づけを与える．仕様記述者が勘と経験に頼って行っていた仕様記述の作業を仕様記述の変換の観点から整理し，仕様記述を変更する意義を明確にすることにより，変換技術が仕様記述を構築するときの指針となることが期待できる．系統的な変換手法の提示は，モデル化支援の観点において，仕様記述者に対してより良いモデルを得るためのヒントを与え，積極的な仕様記述の書き換えを促進する効果が期待できる．また，分析支援の観点からは，目的に応じたモデルで仕様記述を提示することにより，仕様記述の分析支援を図ることができる．変換技術の蓄積は，人間が仕様記述を構築する上で本質的なシステムの意味を考えるための開発環境の基礎づけを与える．

キーワード

システム仕様記述，仕様記述変換，モデル化，仕様記述分析，形式的手法，Z

A Fundamental Study of Transformation Techniques for System Specification with Z*

Han-Myung Chang

Abstract

One of the most important issues in software development is how to make a suitable system specification at the specification stage. Since software development is a cooperative work between customers and developers, the crucial factor in successful development projects is to build up their common understanding of a target application system. The system specification is a key document in order to improve a proper recognition of the system among them. It is necessary to develop supporting techniques for a modeling process and an analysis process to construct a reliable system specification in the specification stage.

In the modeling process, a specifier obtains the most suitable model after trying to specify an application system using various models repeatedly. In this research, we pay attention to the process to transform the system specification on converting a model, and aim to present it as specification transformation techniques in order to support the modeling process in the specification stage. The system specification is basically constructed by internal system states and functions. A modeling of internal system states of the system and a structure of the specification are important in order to make a good specification. And when analyzing system properties, it is significant to explicitly show system invariants of a target system, which are state conditions the system must keep for all time. And

*Doctor's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT9561027, February 5, 1999.

diagrammatic notations help intuitive understanding in the modeling process. Therefore, we pay attention to the following four transformations as fundamental studies of transformation techniques: (1)system internal state models, (2)structure of specification, (3)system invariants, and (4)diagrammatic notations and formal notations.

In this thesis, specification transformation methods are discussed with the above four points of view using the formal notation Z. First of all, transformation methods between binary relation models are shown such as total functions, partial functions and relations which are most characteristic of Z specification when modeling system internal states. Next, function partitioning methods for a layer model are shown as structure of specifications. And transformation methods of extracting system invariants from an operational specification are shown. System invariants are sometimes implicitly expressed as constraints in operational descriptions. Finally, transformation methods from structured diagrams into a Z specification are shown.

Specification transformation techniques are effective to support modeling and specification in software development. A specifier may freely select an appropriate representation in consequence of considerable examinations at various points of view through several models. The development of specification transformation methods means to clear and formalize the specifier's thinking process of modeling and building specifications. With specification transformation techniques, the specifier can select an appropriate representation suitable for a purpose. We expect that the reliability of the specification increases because of examining the specification with an appropriate model for the system.

Keywords:

System Specification, Specification Transformations, Modeling, Specification Analysis, Formal Methods, Z

目 次

図目次

表 目 次

1. はじめに

本章では，ソフトウェア開発における問題点と課題について概説し，本研究の目的と研究方針について述べた後，仕様記述変換技術の概要を説明する．

1.1 本研究の背景と目的

ソフトウェア開発とは「記述」することである，と言っても過言ではない．Michael Jackson は著書 [1] において，「記述こそがソフトウェア開発の中心である」と記している．ソフトウェア開発とは記述を通して機械を構築することであると捉えれば，ソフトウェア開発の最終成果物は機械の記述 (プログラム) であるが，機械の記述を行うためには，システムが何をするものなのかという記述，つまりシステム仕様記述が最も重要であることは言うまでもない．システムの仕様を記述するためには，対象とする世界とそこで解かれるべき問題が何であるか，つまり要求が何であるかを理解することが大切である．システム仕様記述は対象とする世界の中でシステムが何をすべきかを記述し，プログラムはシステムの仕様をいかにして実現するかを記述するものである．ところで，ソフトウェア開発は複数の人間の協調作業であり，ソフトウェア開発における成功の鍵は，顧客と開発者の間，また開発者どうしの間において，開発するシステムが何であるかを正しく理解し共有することである．システムに対する理解を共有するための中心的な役割を果たすものがシステム仕様記述である．

ソフトウェア開発の実際の現場では，ソフトウェア開発の生産性の向上が最も重要な課題として挙げられる．ソフトウェア開発の工程は，(1) 要求定義，(2) 仕様記述，(3) 設計，(4) プログラミング，の 4 つの工程に大きく分けて考えることができる．まず，対象となるシステムの分析を行いシステムに対する要求を定義する．次に要求を満足するためのシステムの仕様を記述する．システムの仕様に従って設計を行った後プログラミングを行う．このようなソフトウェア開発の過程において実際のソフトウェア開発で最も大きな問題は，

- 戻り作業の増大

である．戻り作業の増大の最も大きな要因は，顧客と開発者の間，もしくは開発

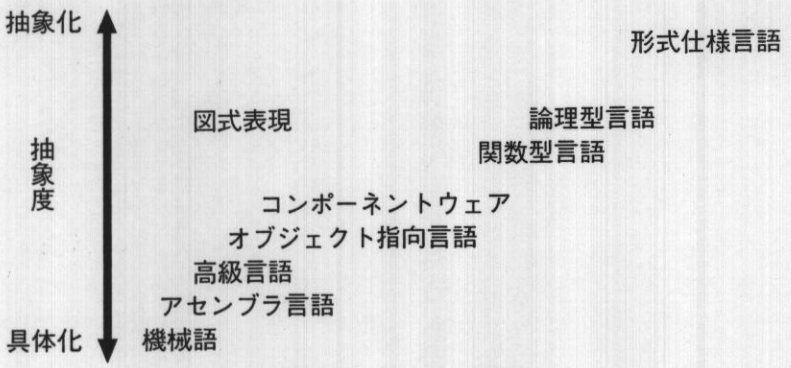


図 1.1 言語の観点からみたソフトウェア工学の歴史

者どうしの間で，システムに対する十分な共通認識が形成されていないことに起因する誤解である．システムに対する要求分析が十分に行われずに，システムの仕様記述が明確に記述されていなければ，開発に携わる人たちの間でシステムに対する理解にずれが生じる．このずれの発見が遅くなればなるほど戻り作業が増大すると考えられる．実際のソフトウェア開発では納期を守ることが重要視されがちなので，戻り作業の増大に伴う修正はプログラムの修正に注がれ，仕様記述まで十分に反映されずに，仕様記述に対する信頼性の低下につながる．また，度重なるプログラムの修正は，ソフトウェアの信頼性の低下の要因にもなりうる．さらに，仕様記述の信頼性の低下は保守作業の増大にも影響を与える．システム仕様記述とプログラムの間にギャップが大きくなり，開発者の間で「プログラムの記述が仕様である」ということが言われるようになれば，仕様記述の信頼性はなくなってしまう．プログラムからシステムの仕様を理解することは困難な作業であり，保守作業コストの増大の要因となる．ソフトウェア開発の生産性を向上させるためのには，仕様記述の信頼性を向上することが最も本質的かつ重要な課題である．

ソフトウェア開発の生産性を向上するための，ソフトウェア工学の歴史は，抽象化の歴史と言っても過言ではない．図 1.1は言語の観点からみたソフトウェア工学の歴史の概略を図示したものである．プログラミング言語は，機械語，アセンブラ言語，高級言語，オブジェクト指向言語，コンポーネントウェアへと機械

語からより抽象的な記述を可能にしている．プログラミング言語の抽象化は，(1) プログラムの可読性の向上，および (2) モジュール化によるプログラムの再利用性の向上，によりソフトウェア開発におけるプログラミングの生産性向上に貢献してきた．しかしながら，プログラミング言語の記述能力が向上したとはいえ，対象とするシステムが何であるかを簡潔に記述するための道具建てとしては，プログラミング言語は適当ではない．今後，社会システムから家電製品にいたるまで，世の中の隅々までシステムの情報化が浸透するにつれて，システムの要求分析およびシステム仕様記述をいかにして効率良く構築するか，というソフトウェア開発における本質的な課題の解決が求められる．

近年，システムの仕様記述と分析を支援するための道具建てとして，実際のソフトウェア開発において形式的手法の適用 [2][3] が盛んに試みられている．形式的手法とは，コンピュータシステムのモデル化，設計，解析を行うための数学を基にした技術であり，仕様記述のための形式的な言語として，Z[4]，VDM[5]，RAISE[6] 等数多く提案されている．また，ソフトウェア開発における形式的手法の有用性については，文献 [7] において明解に述べられている．数学に基づいた言語を用いてシステムの仕様を記述することにより，システムの性質や，詳細化した記述が元の仕様の性質を満足しているかを論理的に証明することを可能にする [5][8][9] [10][11]．形式的手法といえばこのような証明の側面が強調されがちであるが，実用的な形式的手法の効用は，形式的な言語を用いて仕様を記述することにより，記述者に対して抽象的かつ論理的な思考を促し，数学という共通概念を基にした対象システムの一側面の記述を通して，開発者の間における本質的な議論を促進することにある．最近の形式的手法の研究課題はシステムの最適なモデルをいかにして構築するかということに主眼が置かれている．

ソフトウェア開発における課題は，

1. ソフトウェアの信頼性向上，および

2. ソフトウェア開発の生産性向上

である．これまでソフトウェア工学は上記の課題を解決するために，特にプログラミングのレベルで大きな成果をあげてきた．ソフトウェア工学の成果のおかげ

でソフトウェア開発のための技術は向上してきたものの，実際のソフトウェア開発の現場では，未だ経験と勘に頼るところが大きく，ソフトウェア開発を芸術や職人の域から技術の域に向上させることが急務である．ソフトウェア開発を技術の域に向上させるためには，ソフトウェア開発の過程を分析し定式化することが重要である．定式化のためには形式的手法を基にした応用技術の開発が必要であり，特にソフトウェア開発の上流工程において，仕様記述の観点からの要求定義手法およびシステム仕様記述手法の確立が望まれる．

本研究では，上記の問題を解決するための課題として

- システム仕様記述の信頼性向上

に着目した．ソフトウェアの信頼性向上とソフトウェア開発の生産性向上を達成するための鍵はシステム仕様記述である．システム仕様記述の信頼性を向上させるためには，要求定義と仕様記述といった開発の上流工程における作業量は増加するものの，後のソフトウェア開発の見通しが良くなり，戻り作業の減少および保守作業の効率化につながり，ソフトウェア開発の生産性向上に寄与すると期待できる．良い仕様記述を得るためには対象システムのモデル化が最も重要である．良いモデルを発見し記述の信頼性を向上させるためには，記述に対する分析手法の開発が必要である．

以上の議論より，本研究の目的は，

- ソフトウェア開発におけるシステム仕様記述の作業を支援すること

である．本研究では，信頼性の高いシステム仕様記述を効率良く構築するために

1. 対象システムのモデル化，および
2. システム記述の分析

を支援するための技術の開発を目指す．

1.2 研究の方針

本研究の基本方針は，

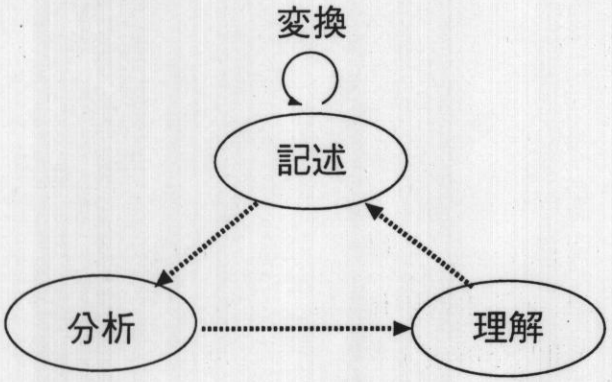


図 1.2 仕様記述における思考パターン

- システムの最適なモデルと仕様記述は試行錯誤の結果得られることに着目し，仕様記述の試行錯誤の過程を変換と捉えその過程を定式化する

ことである．システムの最適なモデルは最初から得られるのではなく，システムの記述が進むにつれて徐々にシステムに対する理解が形成され，より良いモデルを用いて仕様記述が書き換えられる．本研究では，この書き換えの過程を解明し変換手法として提示することにより，システム仕様記述におけるモデル化の支援および分析の支援を図る．仕様記述変換の過程を定式化するために形式的手法を導入する．仕様記述の変換を記号上の処理として形式的に扱うことができれば，仕様記述の変換に対する機械による支援が期待できる．

本研究では，システムの仕様記述の過程を，対象システムに対する理解，記述，分析の繰り返しであると捉えた．図 1.2は仕様記述における思考パターンを図示したものである．まず，仕様を記述するためには対象システムに対する理解が必要である．仕様記述者は自由な発想でシステムの仕様記述を試みる．仕様を記述する過程で，仕様記述者は無意識のうちにシステムの分析を行い，システムに対する理解を深めると考えられる．システムの分析を行い理解が深まった結果，より最適な仕様記述を得るために仕様記述を再構築する．仕様記述の過程は，このような理解，記述，分析の繰り返しとみなすことができる．仕様記述の過程を記述だけの観点からみれば，仕様記述の過程は記述が変換しているだけであるとみ

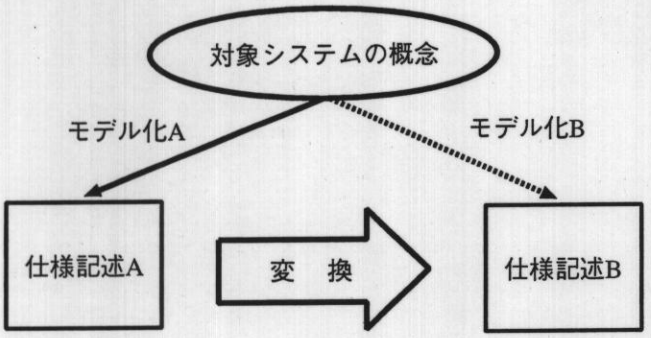


図 1.3 仕様記述変換の概念

なすことも可能であろう．本研究では，仕様記述の書き換えのパターンをモデルの変更として捉え，モデル間の対応関係を明確にすることにより，それぞれのモデル間の系統的な変換手法を提示することを目指している．

1.3 仕様記述変換技術

本論文では，形式仕様記述言語 Z[4] を用いて，対象システムのモデル化とシステム記述の分析を支援するための変換技術を提示する．

1.3.1 仕様記述変換の概念

仕様記述の作業とは「対象システムの概念をモデル化し，そのモデルを用いてシステムの性質を記述すること」である．対象システムのモデル化を行うためには，対象システムが何であるかを理解することが最も重要であり，対象システムの概念を完全に理解した後で，システムの仕様記述を行うことが理想的である．しかし，システムの概念を完全に理解することは困難な作業であり，システムに対する理解は仕様記述の過程を通して徐々に形成される場合が多いと考えられる．

図 1.3は仕様記述変換の概念を図示したものである．仕様記述者は，まず対象システムをモデル A に基づいて仕様記述 A を作成した（モデル化 A）．仕様記述者は，仕様記述 A を用いてシステムの分析を行いシステムに対する理解が深まり，より最適なモデル B に基づいて仕様記述を再構築し仕様記述 B を作成した

(モデル化B)．ところで，モデル化Bは対象システムの概念から直接モデル化されたのではなく，モデル化Aで得られた情報を基にして，モデルAからより最適なモデルBへと変換したとみなすことができる．モデル化Aの実線の矢印は，仕様記述Aが対象システムの概念から直接モデル化されたことを表し，モデル化Bの破線の矢印は，仕様記述Bは概念から直接モデル化されたのではなく，仕様記述Aで得られた情報から再構築されたことを表している．本研究では，この再構築の過程を仕様記述変換とみなし，モデルの変更に伴う仕様記述の変換手法を提示することにより，仕様記述におけるモデル化の支援を図る．

1.3.2 変換の対象

システム仕様記述は，基本的には，システムが保持すべき内部状態と，システムが必要な機能を記述することである．より良いシステム記述を構築するためには，システムの内部状態のモデル化と仕様記述の構造が重要である．また，システムの内部状態が保持すべき制約を明確に記述することが，システム分析において重要な役割を果たす．更に，モデル化の作業を支援するために，直観的で人間になじみやすい図式表現の導入が望まれる．そこで，本研究では変換の対象として，

1. システムの内部状態のモデルの変換，
2. 仕様記述の構造の変換，
3. システム状態不変条件の記述の変換，
4. 図式表現と形式的な表現の間の変換，

について着目した．

システム内部状態のモデルの変換

システムの内部状態のモデルとして最も基本的なモデルである2項関係について着目した．本論文では，Zにおける2項関係モデルの中で最も特徴的なモデルである全域関数，部分関数および関係の間の対応関係を明確にし，それぞれのモデル間の仕様記述の変換手法について議論する．

仕様記述の構造の変換

仕様記述の構造として階層的な機能分割について着目した．システムが大規模で複雑になれば，機能を階層的に分割することは有効な手段である．本論文では，Zにおけるスキーマ合成の表現を用いて，仕様記述の機能分割と階層化を行う変換手法について議論する．

システム状態不変条件の記述の変換

システムの内部状態の性質に影響を及ぼす操作の仕様記述について着目した．操作の仕様記述に操作的に表現されているシステムの内部状態に対する制約を，システム状態不変条件として変換する手法について議論する．

図式表現と形式的な表現の間の変換

直観的で人間になじみやすい図式表現と形式的な表現の間の関係について着目した．CASE(Computer-Aided Software Engineering) ツールで一般的に用いられている構造化ダイアグラムで表現された記述の分析を行うために，図式表現から形式的な表現への変換手法について議論する．図式表現を形式的な表現に変換することにより，ダイアグラム間の一貫性の検証を可能にする．

1.3.3 仕様記述変換の意義

本変換技術の特徴は,「仕様記述の内容を維持しつつモデルを変換する手法を与えること」である．変換の対象となるモデルが何であるかを明確にし，モデル間の対応関係を明示することにより，モデルを変換する手法を提示する．変換技術の意義として，

- モデル変換に伴う暗黙の関係の明示化
- 記述を書き換えるための形式的な作業と意味に関わる作業の分離

があげられる．仕様記述者が仕様記述を書き換える時，仕様記述には明示されない何らかの理由と意図があると考えられる．変換するモデル間の対応関係を明示

することは，仕様記述者が暗黙のうちに対応づけているモデル間の違いの一面を特徴づけることに相当する．モデル間の相違を特徴づけることにより，モデルを変更する仕様記述者の意図を明確にし，仕様記述として陽に表現されない暗黙の関係を明らかにする．また，モデルの対応関係には，仕様の意味に関係なく記述を記号処理として形式的に変換できる対応と，仕様記述者が仕様の意味を解釈してその意味を記述することにより変換を可能にする対応があると考えられる．変換手法の提示は，記号処理として形式的に行える作業と意味を解釈する必要がある作業の違いを明確に分離する．意味を解釈する必要がある作業を明示することは，仕様記述者が暗黙に想定している関係を明示することに相当する．暗黙に想定している関係を明示すれば，後は記号処理として形式的に仕様記述の変換が可能であろう．仕様記述変換は仕様記述者に対して，モデルを変換する時に何を考えればよいかを明確に提示する．ところで，記号処理として形式的に行うことができる変換は機械による支援が期待できる．変換技術の蓄積は，機械にできることは機械に任せ，人間が仕様記述を構築する上で本質的なシステムの意味を考えるための開発環境の基礎づけを与える．

1.4 本論文の概要

本論文は全8章で構成される．

第1章，すなわち本章では，ソフトウェア開発における問題点と課題について概説し，本研究の目的と研究方針，および仕様記述変換技術の概要について述べた．

第2章では，形式仕様記述言語Zの概要について説明する．Zは1970年代後半から英国のオックスフォード大学のPRG(Programming Research Group)を中心に開発された，集合と一階述語論理を基にしたモデル指向の形式仕様記述言語である．Zの特徴と記述スタイルについて説明し，本論文で例題として用いられる「予約管理システム」の仕様記述を示す．

第3章では，システムの内部状態のモデルの変換として，2項関係モデルにおける変換技術について述べる．Zにおける2項関係モデルの中で最も特徴的なモデルである全域関数，部分関数，関係の間の対応関係に着目し，(1) 全域関数と

部分関数の間の変換手法と，(2) 部分関数と関係の間の変換手法，を提示する．

第 4 章では，仕様記述の構造の変換として，階層的な機能分割を行う手法について述べる．階層的機能分割とは，階層的な構造を持たないシステムの仕様記述から，階層的に機能分割された仕様記述を得ること，である．階層的機能分割では，(1) 機能分割と，(2) 階層化，の変換手法を提示する．

第 5 章では，システム状態不変条件の記述の変換として，操作の仕様記述に操作的に表現されているシステムの内部状態に対する制約を，システム状態不変条件として変換する手法を提示する．

第 6 章では，図式表現と形式的な表現の間の変換として，構造化ダイアグラムから形式的な表現 (Z 仕様記述) に変換する手法について述べる．

第 7 章では，変換技術を基にしたソフトウェア開発環境について考察する．

最後に第 8 章では，本研究で得られた成果をまとめ，今後の課題について述べる．

2. 形式仕様記述言語Zの概要

本章では，Zの概要について述べる．まず，Zの特徴について説明し，その後，本論文を通して具体例として用いられる予約管理システムの仕様記述を行う．具体的な仕様記述例を通して，Zの仕様記述構成の概念を示す．

2.1 Zの特徴

Z [4][9][12][13][14] は，集合論と一階述語論理を基にしたモデル指向の形式的仕様記述言語である．Zは，1970年代後半から英国のオックスフォード大学のPRG（Programming Research Group）を中心に開発された．Zでは，関係や関数などの数学モデルを用いて対象システムの記述を行い，システムが満たすべき性質を簡潔かつ厳密に記述する．形式的な仕様記述を基に，数学を道具とした，論理的な議論を展開することが可能である．

Zでは，対象システムが満たすべき性質を記述することを目的としている．システムが「どのように」振舞うかということではなく，システムが「何」をすべきかについて着目して仕様記述を行う．ところで，Zでは仕様記述に対する実行可能性は考慮していない．しかし，Z仕様記述に対する実行可能性は本質的な問題ではなく，逆に実行可能性の概念を排除したために，表現力が豊かで簡潔な記述を可能にしている．仕様記述の妥当性を確認するためには，実際に動作させることも有効であり，文献[15]ではZのサブセットを定義し，Zの仕様記述の実行可能性についても議論している．

一般的に，Zではシステムを

- システム内部状態，と
- 操作

を定義することにより表現する．システムが保持すべきシステムの内部状態は，集合，関係，関数などの概念を用いてモデル化される．また，システムの動的振舞いを表現する操作は，入力，出力，および実行前後のシステム内部状態の間の関係を用いてモデル化される．Zにおいて最も特徴的なものとして，仕様記述の

モジュール化を促進するためのスキーマによる図式表現があげられる．システム内部状態と操作の仕様記述は，それぞれ状態スキーマ，および操作スキーマを用いて定義する．

以下では，Z仕様記述の構成について，簡単な例を用いて説明を行う．ある団体の会員の名前を保持するシステムについて考える．

Zでは，基本的なデータの型として，システムが定義している基本型（basic type) と，ユーザが定義する型（given set）がある．まず，人の名前を表す型について given set を用いて定義することにしよう．given set では，その内部構造については言及しない．

$[PERSON]$

PERSON は人の名前をからなる集合として定義される．

次に，会員の名前を保持するシステム内部状態の仕様を定義しよう．以下に，システム内部状態を表す状態スキーマを示す．

Member

$$member : \mathbb{P}\, PERSON$$

$$\#member < 100$$

上のスキーマで *Member* はスキーマの名前を表している．スキーマの表記は，宣言部（上半分）と述語部（下半分）に分かれている．宣言部ではスキーマで使用する変数の宣言を行い，述語部ではその変数の制約を記述する．ここでは，宣言部において変数 *member* の宣言を行っている．$\mathbb{P}$ は集合を表し，*member* の型は人の名前の集合である．また，述語部では会員の数が１００人未満であることを表現している．# はZで定義されている演算子で，集合の要素の数を表す．

最後に，会員を登録する操作の仕様を定義しよう．操作スキーマの構成は状態スキーマと同じである．操作スキーマでは，宣言部で入力，出力，および実行前後のシステム内部状態の変数を宣言し，それらの関係を述語部で定義することによりその振舞いを定義する．操作スキーマを集合の概念でとらえると，図 2.1に

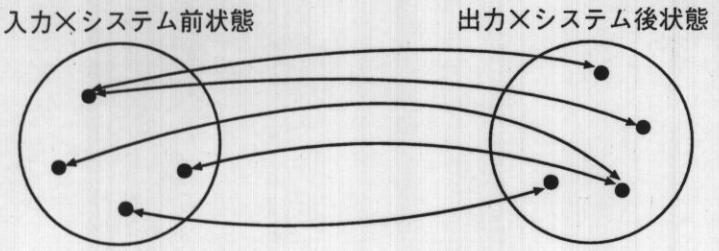


図 2.1 操作の仕様記述の概念

示すように，入力とシステム前状態の組と出力とシステム後状態の組との間の関係を定義することに相当する．

会員の登録を表す操作スキーマを以下に示す．

AddPerson

$member, member' : \mathbb{P}\, PERSON$

$p? : PERSON$

$member' = member \cup \{p?\}$

Zでは，慣用的に，変数名の最後に′がついた変数は操作の実行後の変数を表し，′がついていない変数は実行前の変数を表す．また，変数名の最後に？がついた変数は入力であることを表し，！がついた変数は出力であることを表す．上記例では，宣言部において，システムの前状態 *member*，後状態 *member′* および入力として登録する会員の名前 $p?$ を宣言している．この例では出力はないものとしている．述語部ではこれらの変数の間の関係を定義している．

2.2 予約管理システム

2.2.1 システム概要

ここで，予約管理システムとは「ある共有施設の予約管理を行うためのシステム」であるとする．本システムは，図 2.2 で示されるように，予約に関するデー

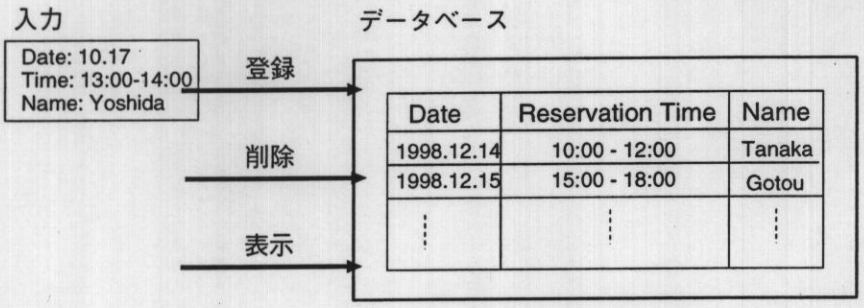


図 2.2 予約管理システム

タを記録するデータベースを保持し，システムの使用者に対して予約の登録，削除，表示などのサービスを提供する．

本システムで求められる要件を以下に示す．

1. システムが保持するデータベースは，日付と予約時間および予約者の名前を記録する．
2. システムは，予約時間が重なることがないことを保証する．(予約時間の非重複性)
3. システムは，予約の登録，予約の削除および予約状況の表示の操作を提供する．

2.2.2 Z仕様記述

ここでは，予約管理システムのZによる仕様記述を行う．まず，仕様記述で用いられるデータの定義を行い，その後，システム内部状態および操作の定義を行う．

データ

仕様記述で必要なデータを定義する．まず，日付と名前の定義を行う．日付と名前のデータついては，その内部構造については言及せず given set として定義する．

$$[DATE, NAME]$$

$DATE, NAME$ は，それぞれ日付および名前の集合を表す．

次に，時刻の定義を行う．時刻は時と分を用いて表現する．

$$Hour == 0\,..\,23;\ Min == 0\,..\,59$$

$$\begin{array}{|l} \textit{Time} \\ \hline h : Hour \\ m : Min \\ \hline \end{array}$$

時は0から２３までの整数，分は0から５９までの整数でモデル化し，時刻は時と分から構成される．$Hour$ は0から２３までの整数の集合，Min は0から５９までの整数の集合を表している．そして，$Time$ は 時を表す h と分を表す m から構成されている．

ここで，時刻の順序関係を表すために，時刻の間の二項関係 LT と LE を定義する．

$$\begin{array}{|l} \_\,\mathsf{LT}\,\_ : Time \leftrightarrow Time \\ \_\,\mathsf{LE}\,\_ : Time \leftrightarrow Time \\ \hline \forall x, y : Time \bullet \\ \quad x\ \mathsf{LT}\ y \Leftrightarrow x.h < y.h \vee (x.h = y.h \wedge x.m < y.m) \\ \forall x, y : Time \bullet x\ \mathsf{LE}\ y \Leftrightarrow x\ \mathsf{LT}\ y \vee x = y \end{array}$$

上の定義は，LT と LE を中置記法の２項演算子として定義している．x LT y は「x は y より早い時刻である」ことを表し，x LE y は「x は y より早い時刻，もしくは同じ時刻である」ことを表している．

次に，予約時間および１日の予約時間の集合を定義する．まず，予約時間を定義する．予約時間は開始時間と終了時間を用いて表現する．

$$RTime == \{st, et : Time \mid st\ \mathsf{LE}\ et\}$$

予約時間は開始時刻と終了時刻の二項組でモデル化する．$RTime$ は開始時刻と終了時刻の二項組で表される予約時間の集合を表している．集合定義の制約

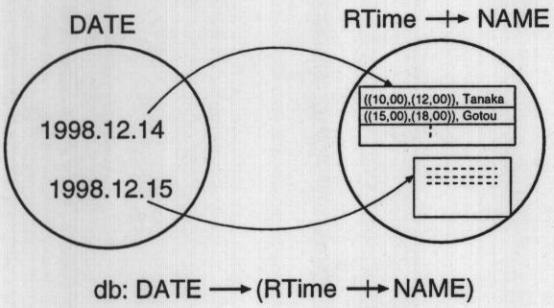


図 2.3 予約管理システムの内部状態のモデル化

部では,「予約時間の開始時刻は終了時刻より早い」という制約を記述している. *RTime* を用いて 1 日の予約時間の集合を定義する.

$$
\begin{aligned}
&RTimeSet == \\
&\quad \{r : \mathbb{P}\, RTime;\ xs, xe, ys, ye : Time \mid \\
&\qquad (xs, xe) \in r \land (ys, ye) \in r \land (xe \ \mathsf{LE}\ ys \lor ye \ \mathsf{LE}\ xs) \bullet r\}
\end{aligned}
$$

1 日の予約時間の集合では，予約時間の非重複性を保証する必要がある．予約時間の非重複性は，*RTimeSet* の集合定義の述語部で記述している．*RTimeSet* は，予約時間が重なることのない予約時間の集合の集合を表している．

最後に，操作の結果を表すデータを定義する．

$$REPORT ::= ok \mid invalid\_input \mid already\_reserved$$

REPORT は， *ok*, *invalid_input*, *already_reserved* のいずれかである．

内部状態

予約管理システムの内部状態を定義する．システムの内部状態は，予約に関するデータを保持するデータベースに相当する．予約状態を管理するデータベースのモデル化はいろいろ考えられるが，ここでは図 2.3に示すように．日付の集合を定義域，予約時間から名前への関数を値域とする，全域関数としてモデル化した．これをスキーマでは次のように表現する．

State

$$tf : DATE \longrightarrow (RTime \mathrel{+\!\!\!\rightarrow} NAME)$$

$$\forall\, d : \mathrm{dom}(tf) \bullet \mathrm{dom}(tf(d)) \in RTimeSet$$

スキーマ定義の述語部では，予約時間の非重複性について *RTimeSet* を用いて表現している．

ところで，内部状態の初期状態を定義する必要がある．内部状態の初期状態は，何も予約されていない状態とする．

InitState

State

$$\forall\, d : \mathrm{dom}(tf) \bullet tf(d) = \varnothing$$

操作

本システムでは 2.2.1項で述べたように，3 つの操作を定義しなければならないが，本論文では紙数の都合上，予約の登録についてのみ行う．

予約管理システムにおける予約登録の操作を定義する．予約の登録は，日付，予約の開始時刻と終了時刻，および予約者の名前を入力とし，結果を出力する．

Input

$d : DATE$

$st, et : Time$

$n : NAME$

Output

$r : REPORT$

Input? は入力の変数，*Output*! は出力の変数を表す．

予約の登録は，登録が行われる正常処理と，登録ができないエラー処理の操作を記述する．まず，正常処理の操作を定義する．

EntryOk

$$\Delta State;\ Input?;\ Output!$$

$$(st?, et?) \in RTime$$
$$(st?, et?) \notin \mathrm{dom}(tf(d?))$$
$$tf' = tf \oplus \{d? \mapsto (tf(d?) \cup \{(st?, et?) \mapsto n?\})\}$$
$$r! = ok$$

宣言部における $\Delta State$ は，操作を実行する時の *State* の前状態と後状態の変数の宣言を表す．データベースの登録は，

$$tf' = tf \oplus \{d? \mapsto (tf(d?) \cup \{(st?, et?) \mapsto n?\})\}$$

の述語で表現されている．

ところで，*EntryOk* の述語部は *State* の述語部も展開されるので，*EntyrOk* は以下のスキーマと同等である．

EntryOkAlt

$$tf, tf' : DATE \longrightarrow (RTime \nrightarrow NAME)$$
$$d? : DATE;\ st?, et? : Time;\ n? : NAME$$
$$r! : REPORT$$

$$\forall\, d : \mathrm{dom}(tf) \bullet \mathrm{dom}(tf(d)) \in RTimeSet$$
$$\forall\, d : \mathrm{dom}(tf') \bullet \mathrm{dom}(tf'(d)) \in RTimeSet$$
$$(st?, et?) \in RTime$$
$$(st?, et?) \notin \mathrm{dom}(tf(d?))$$
$$tf' = tf \oplus \{d? \mapsto (tf(d?) \cup \{(st?, et?) \mapsto n?\})\}$$
$$r! = ok$$

EntryOk の述語部では、操作実行後の予約時間の非重複性の保証は陽に示されていないが、非重複性は *EntryOkAlt* における述語部の 2 行目の述語の制約で示

されている．したがって，*EntryOk* の操作では「保持するデータベースの後状態 *State'* は予約時間の非重複性を満足する」という制約が暗に記述されている．

エラー処理の操作は次のように定義される．

InvalidInput

$$\Delta State;\ Input?;\ Output!$$

$$(st?, et?) \notin RTime$$
$$tf' = tf$$
$$r! = invalid\_input$$

AlreadyReserved

$$\Delta State;\ Input?;\ Output!$$

$$(st?, et?) \in RTime$$
$$(st?, et?) \in \mathrm{dom}(tf(d?)) \lor$$
$$\mathrm{dom}(tf(d?)) \cup \{(st?, et?)\} \notin RTimeSet$$
$$tf' = tf$$
$$r! = already\_reserved$$

InvalidInput は入力の時刻が予約時間として妥当でない時，また，*AlreadyReserved* はすでに予約が行われている時間に対して予約登録を行った時の操作を記述している．$\mathrm{dom}(tf(d?)) \cup \{(st?, et?)\} \notin RTimeSet$ は，予約する日に登録されいているデータの集合と入力のデータを足し合わせた集合が，1日の予約時間の集合として妥当な集合 (*RTimeSet*) に含まれないという式で，入力の予約時間が既に予約されているデータと重複することを表現している．これらの操作では，保持するデータベースは変化しない．

最後に，予約の登録の操作の全体を定義する．

$$Entry \triangleq EntryOk \lor InvalidInput \lor AlreadyReserved$$

上の定義は，「*Entry* は *EntryOk*，*InvalidInput*，もしくは *AlreadyReserved* のいずれかである」ことを表している．

3. 2項関係モデルにおける仕様記述変換

本章では，2項関係モデルにおける仕様記述変換の手法について述べる．本手法では，Zにおける2項関係として，全域関数，部分関数，および関係に着目した．全域関数は，定義域において未定義項を考慮する必要がないので，仕様記述が扱いやすく，対象システムの概念の本質を端的に表すことができる．しかし，2項関係の情報をシステムの内部状態として保持するためには，定義域として有限の情報を扱うために部分関数によるモデル化が現実的なモデルとして有効である場合が多い．また，関係による仕様記述では，リレーショナルデータベースにおける表に対して最も親和性の高い記述となっている．

そこで，全域関数，部分関数，および関係の間で仕様記述の内容を保持したまま仕様記述を変換することができれば，まず，全域関数のもとで未定義項の関数適用を特別扱いせずに考え易いモデルで仕様記述を検討した後，仕様記述変換を用いて部分関数化さらに関係化という過程をとることにより，プログラムに近い仕様記述を系統的に構築することができる．

また，仕様を修正する場合は，対象システムの概念の本質を端的に現している全域関数の仕様記述で仕様の修正を行うことが望ましい．仕様記述者は，仕様の修正を考えやすいモデルを用いて，仕様の修正に本質的な作業に集中することができる．仕様の修正に適したモデルで検討し，プログラムに近い仕様記述は形式的な変換で得ることにより，仕様の修正に伴うエラーの混入を防ぐことが期待できる [16][17].

3.1 概要

本変換技術の特徴は,「仕様記述の内容を維持しつつモデルを相互に変換する手法を与えること」である．変換の対象となるモデルが何であるかを明確にし，モデル間の対応関係を明示することにより，モデル間を相互に変換する手法を提示する．モデルの対応関係には，仕様の意味に関係なく記述を記号処理として形式的に変換できる対応と，仕様記述者が仕様の意味を解釈してその意味を記述することにより変換を可能にする対応があると考えられる．変換手法の提示は，記号

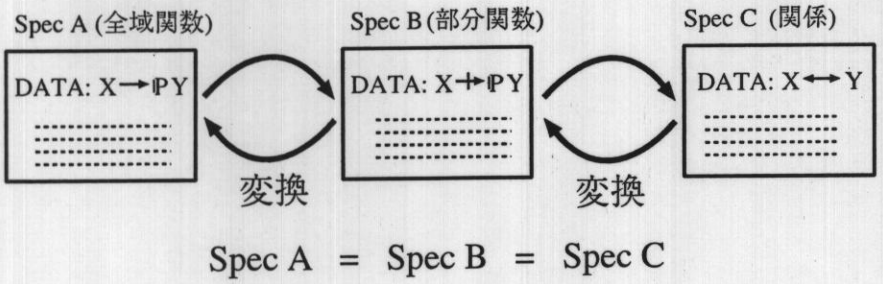


図 3.1　2 項関係モデルにおける変換

処理として形式的に行える作業と意味を解釈する必要がある作業の違いを明確に分離する．意味を解釈する必要がある作業を明示することは，仕様記述者が暗黙に想定している関係を明示することに相当する．暗黙に想定している関係を明示すれば，後は記号処理として形式的に仕様記述の変換が可能であろう．仕様記述変換は仕様記述者に対して，モデルを変換する時に何を考えればよいかを明確に提示する．

2 項関係モデルの変換手法として，図 3.1に示すように，

1. 全域関数と部分関数間の変換手法，および

2. 部分関数と関係間の変換手法

をそれぞれ提示する．2. については，集合 X から 集合 Y の巾集合への部分関数 $X \nrightarrow \mathbb{P}\, Y$ と関係 $X \leftrightarrow Y$ の間の変換について議論する．

全域関数と部分関数間の変換では，全域関数でモデル化されたデータと，部分関数でモデル化されたデータの間の変換について議論する．本手法では，部分関数による仕様記述と全域関数による仕様記述が有する情報を同じにするために，部分関数の定義域に属さない要素の対応を補助関数を導入して定義する．全域関数と部分関数間の変換は，全域関数，部分関数，および補助関数間の関係を明示することにより実現される．

部分関数と関係間の変換では，集合 X から 集合 Y の巾集合への部分関数 $X \nrightarrow \mathbb{P}\, Y$ でモデル化されたデータと，関係 $X \leftrightarrow Y$ でモデル化されたデータの間の変換について議論する．本手法では，$X \nrightarrow \mathbb{P}\, Y$ と $X \leftrightarrow Y$ を同一視する

概念を導入し，これらの間のデータ構造を変換する関数を定義する．$X \pfun \mathbb{P}\, Y$ と $X \rel Y$ 間の変換は，この変換関数を用いて実現される．

ところで，Zでは2項関係や関数は集合を基に定義されており，それらのデータを定義に遡って集合の表記を用いて書き換えることができる．例えば「集合 X と 集合 Y の関係」は，Zでは 「X と Y の直積の集合」として捉えられている．実際，Zにおける関係は以下のように定義されている．

$$X \rel Y == \mathbb{P}(X \times Y)$$

また，Zにおける関数は「関係の制約されたもの」として捉えられている．例えば，部分関数は以下のように定義されている．

$$X \pfun Y == \{f : X \rel Y \mid (\forall x : X, y_1, y_2 : Y \bullet \\ (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2)\}$$

このように，Zでは，2項関係や関数などは直積の集合で表現されており，これらの表記は集合による表記の省略形とみなすことができる．したがって，以下のスキーマは，それぞれ全く同じ内容を表現している．

PFun

$$f : X \pfun Y$$

Rel

$$f : X \rel Y$$

$$\forall x : X; y_1, y_2 : Y \bullet (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2$$

CartesianSet

$$f : \mathbb{P}(X \times Y)$$

$$\forall x : X; y_1, y_2 : Y \bullet (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2$$

それぞれのスキーマにおける変数 f は，データ構造は表記上は異なっているけれども，その意味を定義に遡って考えると全く同じものである．

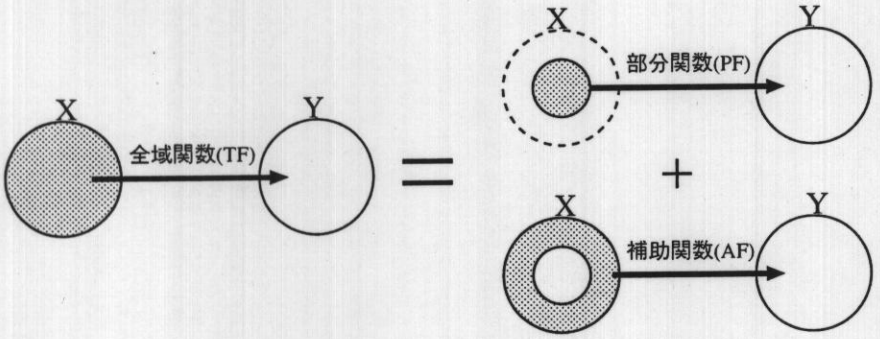


図 3.2 補助関数の導入による全域関数と部分関数の同一視

3.2 全域関数と部分関数間の変換

本節では，全域関数と部分関数間の変換について述べる．まず，変換の基本アイデアについて説明し，その後，全域関数と部分関数間の変換手法を示す．また，具体例として予約管理システムの仕様記述に対して変換を試みる．

3.2.1 基本アイデア

全域関数と部分関数間の変換の基本アイデアは，

- 部分関数の定義域に属さない要素に対する補助関数を導入して，全域関数が有する情報と部分関数が有する情報を同一視すること

である．図 3.2では，補助関数（AF）を導入して，全域関数（TF）と部分関数（PF）を同一視する概念を示している．AF，TF，および PF の間には，次の条件を満足する必要がある．

$TPACondition[X, Y]$

$$tf : X \longrightarrow Y$$
$$pf, af : X \rightarrowtail Y$$

$$pf = \mathrm{dom}(pf) \lhd tf$$
$$af = \mathrm{dom}(pf) \lhd\!\!\!\!- tf$$

上のスキーマ *TPACondition* は関数間の満たすべき条件を記述している．宣言部では全域関数 tf，部分関数 pf，および補助関数 af の宣言を行っている．述語部において，1行目の制約

$$pf = \mathrm{dom}(pf) \lhd tf$$

は「pf の対応関係は，pf の定義域に属している tf の対応関係と同じであること」を表現している．また，2行目の制約

$$af = \mathrm{dom}(pf) \lhd\!\!\!\!- tf$$

は「af の対応関係は，pf の定義域に属していない tf の対応関係と同じであること」を表現している．

以上のような関係を満足する部分関数と補助関数を定義すれば，全域関数が有する情報と部分関数が有する情報を同一視することができる．すると，全域関数と部分関数の間で，互いにデータのモデルを変換することが可能となる．ところで，補助関数を定義することは，部分関数のモデル化において，定義域に属さない要素の意味を明示することに相当する．

3.2.2 変換手法

本項では，全域関数と部分関数間の変換手法について述べる．まず，全域関数から部分関数への変換について考える．変換は次の手順で行う．

1. 補助関数の定義と，変換の対象となる全域関数，部分関数，および補助関数の関係をスキーマで記述する．
2. スキーマ合成を用いて，部分関数による仕様記述を得る．

手順の詳細は以下に示す．

関数間のスキーマの定義

補助関数の定義，および関数間の関係をスキーマを用いて記述する．

TPARelation

$$tf : X \longrightarrow Y$$
$$pf, af : X \pfun Y$$

$$tf = pf \cup af$$
Predicate

宣言部では全域関数 tf，部分関数 pf，および補助関数 af の宣言を行う．述語部では，関数間の関係と補助関数 af の定義を行う．1行目の制約

$$tf = pf \cup af$$

は，関数間の関係を表現している．そして，2行目の制約 *Predicate* において，関数間の条件 *TPACondition* を満足する補助関数 af の定義を行う．*Predicate* では，条件 *TPACondition* を満足するように af を定義する必要がある．

部分関数による仕様記述の獲得

全域関数で定義された仕様記述から，部分関数による仕様記述を得る方法について述べる．ここでは，全域関数による仕様記述 *TSpec* に対する変換について考える．*TSpec* は以下の通りである．

TSpec

$$tf : X \longrightarrow Y$$
Decl

Predicate

Decl は変換の対象となる変数 tf 以外の変数の宣言を表し，*Predicate* は述語を表す．部分関数による仕様記述 *PSpec* は以下の定義により得られる．

$$PSpec \mathrel{\widehat=} (TSpec \land TPARelation) \setminus (tf, af)$$

上の定義では，スキーマ *PSpec* は，

1. 全域関数によるスキーマ *TSpec* と関数間のスキーマ *TPARelation* を論理積によるスキーマ合成により結び付け，

2. 全域関数の変数 *tf* と補助関数の変数 *af* を変数を隠蔽する演算子（$\setminus$）を用いて宣言から取り除く

ことを意味している．したがって *PSpec* の宣言部は

PSpecDecl

$$pf : X \pfun Y$$
$$Decl$$

となり，*PSpec* は部分関数を用いた仕様記述となる．

以上で，全域関数から部分関数への変換の手法を示した．逆に，部分関数から全域関数への変換は，同様にして

$$TSpec \mathrel{\widehat=} (PSpec \land TPARelation) \setminus (pf, af)$$

により，全域関数による仕様記述 *TSpec* を，部分関数の仕様記述 *PSpec* から得ることができる．

3.2.3 適用例

本項では，具体例として，予約管理システムの仕様記述に対して全域関数から部分関数への変換の適用を試みる．2.2節では，システム内部状態を表すスキーマ *State* において，データベースの構造を全域関数

$$tf : DATE \fun (RTime \pfun NAME)$$

としてモデル化した．ここでは，これを部分関数

$$pf : DATE \pfun (RTime \pfun NAME)$$

としてモデル化した仕様記述に変換する．まず，全域関数および部分関数によるモデル化についての考察を行い，関数間のスキーマを定義する．その後，全域関数

による仕様記述から部分関数による仕様記述を得る．ここでは，*State*, *InitState*, *EntryOk* の仕様記述に対する変換を行う．

全域関数では，全ての日付に対応する予約情報をデータベースとして保持するという概念に基づいてモデル化されている．これを部分関数を用いてモデル化するためには，どのような補助関数を導入すればよいだろうか．ここでは，

- 部分関数で保持していない日付に対しては「予約がない」とみなす

というモデル化を考える.「予約がない」は，空集合を対応させることで表現することができる．すると，補助関数として以下の *af* を導入する．

$$af = \{d : DATE \mid d \notin \mathrm{dom}(pf) \bullet d \mapsto \varnothing\}$$

そこで，関数間のスキーマを以下のように定義する．

TPARel

$$State$$
$$pf, af : DATE \pfun (RTime \pfun NAME)$$

$$tf = pf \cup af$$
$$af = \{d : DATE \mid d \notin \mathrm{dom}(pf) \bullet d \mapsto \varnothing\}$$

上の定義では，*pf* と *af* の定義域は排他的である．つまり定義域の要素に重なりはないので，関数間の条件 *TPACondition* の制約

$$pf = \mathrm{dom}(pf) \lhd tf \land af = \mathrm{dom}(pf) \ndres tf$$

は確かに満足されている．

次に，*TPARel* を用いて部分関数による仕様記述を得る．全域関数による仕様記述 *State*, *InitState*, *EntryOk* に対して部分関数による仕様記述 *PState*, *PInitState*, *PEntryOk* を定義する．

$$PState \mathrel{\widehat=} (State \land TPARel) \setminus (tf, af)$$
$$PInitState \mathrel{\widehat=} (InitState \land TPARel) \setminus (tf, af)$$
$$PEntryOk \mathrel{\widehat=} (EntryOk \land TPARel) \setminus (tf, af)$$

上で定義したスキーマを展開し簡単化した仕様記述を以下に示す.

ExpandPState

$$pf : DATE \pfun (RTime \pfun NAME)$$

$$\forall d : \mathrm{dom}(pf) \bullet \mathrm{dom}(pf(d)) \in RTimeSet$$

ExpandPInitState

$$PState$$

$$pf = \emptyset$$

ExpandPEntryOk

$$\Delta PState;\ Input?;\ Output!$$

$$(st?, et?) \in RTime$$
$$d? \notin \mathrm{dom}(pf) \land$$
$$\quad pf' = pf \cup \{d? \mapsto \{(st?, et?) \mapsto n?\}\} \lor$$
$$d? \in \mathrm{dom}(pf) \land$$
$$\quad (st?, et?) \notin \mathrm{dom}(pf(d?)) \land$$
$$\quad pf' = pf \oplus \{d? \mapsto (pf(d?) \cup \{(st?, et?) \mapsto n?\})\}$$
$$r! = ok$$

以上のようにして，全域関数による仕様記述 *State*, *InitState*, *EntryOk* から部分関数による仕様記述 *PState*, *PInitState*, *PEntryOk* を変換により得ることができた．ここで述べたスキーマ以外の仕様記述に対しても同様にして，部分関数による仕様記述を得ることができる．

3.3 部分関数と関係間の変換

本節では，部分関数と関係間の変換について述べる．ここでは，集合 X から集合 Y の巾集合への部分関数 $X \pfun \mathbb{P}\, Y$ と関係 $X \rel Y$ の間の変換について考

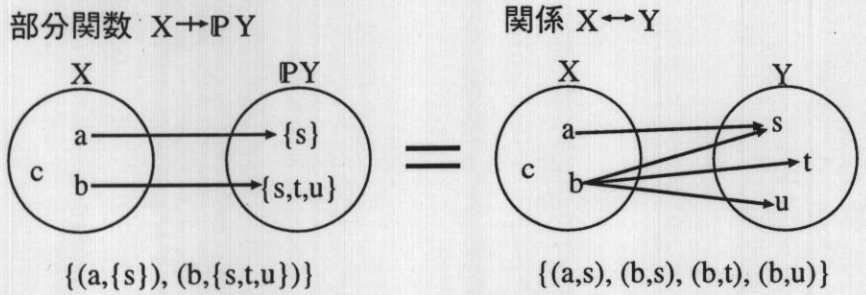


図 3.3 部分関数と関係の内容の同一視

察する．まず，変換の基本アイデアについて説明し，その後，部分関数 $X \nrightarrow \mathbb{P}\, Y$ と関係 $X \leftrightarrow Y$ 間の変換手法を示す．また，具体例として予約管理システムの仕様記述に対して変換を試みる．

3.3.1 基本アイデア

部分関数 $X \nrightarrow \mathbb{P}\, Y$ と関係 $X \leftrightarrow Y$ 間の変換の基本アイデアは，

- 部分関数 $X \nrightarrow \mathbb{P}\, Y$ と関係 $X \leftrightarrow Y$ の表記内容を同一視すること

である．これは，例えば集合 X, Y をそれぞれ

$$X = \{a, b, c\}, Y = \{s, t, u\}$$

としたとき，図3.3に示すように，部分関数 $X \nrightarrow \mathbb{P}\, Y$ の集合表記

$$\{(a, \{s\}), (b, \{s, t, u\})\}$$

と，関係 $X \leftrightarrow Y$ の集合表記

$$\{(a, s), (b, s), (b, t), (b, u)\}$$

の内容を同一視することである．

これは，部分関数の視点からみれば，「X から Y の集合への部分関数を X と Y の関係として解釈すること」に相当する．逆に，関係の視点からみれば，「X

と Y の関係を X から Y の集合への部分関数として解釈すること」に相当する．以上のような部分関数と関係の間の解釈は，以下のスキーマで表現することができる．

$$
\begin{array}{|l}
\hline
\mathit{IdentifyPR}[X, Y] \\
\hline
pf : X \pfun \mathbb{P}\, Y \\
rl : X \rel Y \\
\hline
pf = \{x : \mathrm{dom}(rl) \bullet x \mapsto rl(\!|\{x\}|\!)\} \\
rl = \bigcup\{x : \mathrm{dom}(pf) \bullet \{y : pf(x) \bullet x \mapsto y\}\} \\
\hline
\end{array}
$$

部分関数 pf は関係 rl を用いて表現することが可能で，逆に関係 rl も部分関数を用いて表現することが可能である．*IdentifyPR* の関係からデータ構造を変換する高階関数を定義することは可能である．そして，その変換関数を用いることにより，どちらのデータ構造にでも仕様記述を変換することができる．変換関数，および変換方法については，次項で述べる．ところで，部分関数 $X \pfun \mathbb{P}\, Y$ の型は $\mathbb{P}(X \times \mathbb{P}\, Y)$，関係 $X \rel Y$ の型は $\mathbb{P}(X \times Y)$ であり，違う型である．したがって，この変換はデータ構造の変換とみなすことができる．

3.3.2 変換手法

部分関数 $X \pfun \mathbb{P}\, Y$ と関係 $X \rel Y$ 間の変換手法について述べる．部分関数から関係への変換関数 $P2R$，および関係から部分関数への変換関数 $R2P$ を定義し，仕様記述の変換の方法を示す．

変換関数

変換関数 $P2R$ および $R2P$ を定義する．変換関数 $P2R$ は，部分関数 $X \pfun \mathbb{P}\, Y$ のデータ構造を関係 $X \rel Y$ のデータ構造に変換し，逆に，$R2P$ は，関係 $X \rel Y$ のデータ構造を部分関数 $X \pfun \mathbb{P}\, Y$ のデータ構造に変換する関数である．変換関数は，1対1の関数（単射）として以下のように定義される．

$$
\begin{array}{|l}
\hline
[X, Y] \\
P2R : (X \pfun \mathbb{P}\, Y) \rightarrowtail (X \rel Y) \\
R2P : (X \rel Y) \rightarrowtail (X \pfun \mathbb{P}\, Y) \\
\hline
\forall\, pf : X \pfun \mathbb{P}\, Y \bullet \\
\quad P2R(pf) = \bigcup \{x : \mathrm{dom}(pf) \bullet \{y : pf(x) \bullet x \mapsto y\}\} \\
\forall\, rl : X \rel Y \bullet \\
\quad R2P(rl) = \{x : \mathrm{dom}(rl) \bullet x \mapsto rl(\!|\{x\}|\!)\} \\
\hline
\end{array}
$$

図 3.3で示した表記例に対して，この変換関数を適用すれば，

$$
\begin{aligned}
&P2R(\{(a, \{s\}), (b, \{s, t, u\})\}) = \{(a, s), (b, s), (b, t), (b, u)\} \\
&R2P(\{(a, s), (b, s), (b, t), (b, u)\}) = \{(a, \{s\}), (b, \{s, t, u\})\}
\end{aligned}
$$

を得る．このようにして，変換関数を用いて部分関数と関係の間のデータ構造を自由に換えることができる．

仕様記述の変換

まず，部分関数から関係への仕様記述の変換について考える．この変換では，部分関数の「変数」のデータ構造を関係のデータ構造に変換する．もちろん，変換前の仕様記述の述語部では,「変数」は部分関数として記述されているので，変換後の述語部に変換前と同じ記述を使うことはできない．しかし，関係のデータの意味を部分関数と解釈して,「変数」を部分関数に変換して制約を記述することは可能である．これは，関係の「変数」に対して $R2P$ を適用することにより実現することができる．したがって，変換前と同じ仕様記述を得るには，$R2P$ を用いて変換前の述語部と同じ制約を記述すればよい．これは，変換前の述語部に対して，対象となる変数に $R2P$ を適用することにより得ることができる．

以上の議論より，部分関数 $X \pfun \mathbb{P}\, Y$ から関係 $X \rel Y$ の仕様記述は，以下の手順で得ることができる．

1. 変換の対象となる変数の型を $X \pfun \mathbb{P}\, Y$ から $X \rel Y$ に換える．

2. 変換の対象となる変数に $R2P$ を適用する.

以上で，部分関数 $X \pfun \mathbb{P}\, Y$ から関係 $X \rel Y$ への変換手法を示した．逆に，関係から部分関数の仕様記述は，以下の手順で得ることができる.

1. 変換の対象となる変数の型を $X \rel Y$ から $X \pfun \mathbb{P}\, Y$ に換える.

2. 変換の対象となる変数に $P2R$ を適用する，

3.3.3 適用例

本項では，具体例として，予約管理システムの仕様記述に対して部分関数から関係への変換の適用を試みる．変換は，部分関数による仕様記述 *PState*, *PInitState*, *PEntryOk* に対して行うことにする.

3.2.3項では，データベースのデータ構造を

$$pf : DATE \pfun (RTime \pfun NAME)$$

としてモデル化した．Zでは，3.1節で述べたように，関数は直積の集合として捉えているので，pf は，

$$pf : DATE \pfun \mathbb{P}(RTime \times NAME)$$

と記述することができる．ただし，pf には関数の制約

$$\forall r : \mathrm{ran}(pf) \bullet r \in RTime \pfun NAME$$

を付け加える必要がある．したがって，3.2.3項における，システムの内部状態を表すスキーマ *PState* は，以下のように展開することができる.

PStateAlt

$$pf : DATE \pfun \mathbb{P}(RTime \times NAME)$$

$$\forall r : \mathrm{ran}(pf) \bullet r \in RTime \pfun NAME$$
$$\forall d : \mathrm{dom}(pf) \bullet \mathrm{dom}(pf(d)) \in RTimeSet$$

上のスキーマ *PStateAlt* に対して変換を適用すれば，関係による内部状態のスキーマ *RState* を得る．

RState

$$rl : DATE \leftrightarrow (RTime \times NAME)$$

$$\forall r : \mathrm{ran}(R2P(rl)) \bullet r \in RTime \nrightarrow NAME$$
$$\forall d : \mathrm{dom}(R2P(rl)) \bullet \mathrm{dom}(R2P(rl)(d)) \in RTimeSet$$

RState は以下のように簡単化することができる．

ExpandRState

$$rl : DATE \leftrightarrow (RTime \times NAME)$$

$$\forall r : \{d : \mathrm{dom}(rl) \bullet rl(\!|\{d\}|\!)\} \bullet r \in RTime \nrightarrow NAME$$
$$\forall d : \mathrm{dom}(rl) \bullet \mathrm{dom}(rl(\!|\{d\}|\!)) \in RTimeSet$$

同様にして，*PInitState*, *PEntryOk* に対しても変換を適用すると，関係による仕様記述 *RInitState*, *REntryOk* を得る．

RInitState

$$RState$$

$$R2P(rl) = \varnothing$$

REntryOk

$\Delta RState$; $Input?$; $Output!$

$(st?, et?) \in RTime$

$d? \notin \mathrm{dom}(R2P(rl)) \land$

$\quad R2P(rl') = R2P(rl) \cup \{d? \mapsto \{(st?, et?) \mapsto n?\}\} \lor$

$d? \in \mathrm{dom}(R2P(rl)) \land$

$\quad (st?, et?) \notin \mathrm{dom}(R2P(rl)(d?)) \land$

$\quad R2P(rl') = R2P(rl) \oplus$

$\quad\quad \{d? \mapsto (R2P(rl)(d?) \cup \{(st?, et?) \mapsto n?\})\}$

$r! = ok$

変換により得られた *RInitState*, *REntryOk* は，以下のように簡単化することができる．

ExpandRInitState

$RState$

$rl = \varnothing$

ExpandREntryOk

$\Delta RState$; $Input?$; $Output!$

$(st?, et?) \in RTime$

$(st?, et?) \notin \mathrm{dom}(rl(\!|\{d?\}|\!))$

$rl' = rl \cup \{d? \mapsto ((st?, et?), n?)\}$

$r! = ok$

以上のようにして，部分関数による仕様記述 *PState*, *PInitState*, *PEntryOk* から関係による仕様記述 *RState*, *RInitState*, *REntryOk* を変換により得ることができた．ここで述べたスキーマ以外の仕様記述に対しても同様にして，関係による仕様記述を得ることができる．

3.4 評価

本節では，2項関係モデルにおける変換の特徴について述べた後，ソフトウェア開発における2項関係モデルの変換手法の意義について議論する．

3.4.1 2項関係モデルにおける変換の特徴

2項関係モデルはシステムの内部状態を表現するための最も基本的なモデルであり，全域関数、部分関数および関係は，Zにおける2項関係モデルの中で最も特徴的なモデルである．本論文では，2項関係モデルにおける変換手法として，(1) 全域関数と部分関数間の変換手法と，(2) 部分関数と関係間の変換手法を提示した．本変換技術の特徴は,「仕様記述の内容を維持しつつモデルを相互に変換する手法を与えること」である．以降では，それぞれの変換手法の特徴について述べる．

全域関数と部分関数間の変換手法の特徴

全域関数と部分関数間の変換手法では，部分関数の定義域に属さない要素に対する補助関数を定義することにより，全域関数と部分関数による仕様記述間の形式的な相互変換を可能にした．ここで，形式的な変換とは，仕様の意味を解釈することなく仕様記述を記号処理として変換することを言う．補助関数を定義することは，全域関数と部分関数が有する情報を同一視するために，仕様の意味を解釈して部分関数のモデル化における定義域に属さない要素に対する対応関係を明示したことを意味する．したがって，全域関数と部分関数間の変換では，補助関数を定義するために仕様の意味を解釈する必要がある．しかし逆に言えば，補助関数を定義さえすれば，全域関数と部分関数の間で相互に変換が可能である．

ところで，部分関数の観点から補助関数を定義することの意義について考えると，定義域に属さない要素の対応関係を定義することは，仕様記述には明示されない暗黙の関係を陽に記述することを仕様記述者に対して強制する．ソフトウェア開発では，仕様記述の段階における開発者間での誤解が後のソフトウェア開発に大きな影響を与える．開発者間における誤解の原因の一つとして，一見自明と思われる仕様の意味が明記されていないことが考えられる．開発者間で対象シス

テムに対する共通の認識を形成するためは，仕様はできるだけ明記されていることが望ましい．補助関数を定義することは，仕様の誤解を少なくするための暗黙の仕様を明示することに効果がある．

部分関数と関係間の変換手法の特徴

部分関数と関係間の変換手法では，集合 X から集合 Y の巾集合への部分関数 ($X \pfun \mathbb{P}\, Y$) と，集合 X と集合 Y の関係 ($X \rel Y$) に関して，部分関数と関係の間のデータ構造に関する変換関数を提示することにより，部分関数と関係による仕様記述間の形式的な相互変換を可能にした．したがって，部分関数と関係間の変換では，仕様の意味を解釈することなく，仕様記述を記号処理として変換することができる．

本変換における部分関数から関係への変換は，リレーショナルデータモデルにおける正規化 [18] に相当すると捉えることができる．ここでいう正規化とは，レコードのフィールド値として集合の値を許す非正規形リレーションを，フィールド値に単一の値をとる第一正規形 (first normal form) に直すことを言う．リレーショナルデータベースにおける正規化では表に関する変換を提示しているが，本変換では操作の仕様記述を含めた仕様記述全体の変換を可能にしている．ところで，関係の仕様記述の観点からデータ構造に関する変換関数の意義について考えると，データ構造に関する変換関数は，関係のデータをどのように解釈するかを明示することに相当すると考えられる．

3.4.2 ソフトウェア開発における変換手法の意義

ソフトウェア開発における 2 項関係モデルにおける変換手法の意義について，仕様記述の過程と仕様の修正作業の過程の観点から考察する．

仕様記述の過程

本論文の適用例で示した，全域関数，部分関数，関係への仕様記述の流れは，仕様記述における一つの指針となり得ると考えられる．全域関数によるモデル化では，定義域において未定義項を考慮する必要がないので，仕様記述が扱いやす

く，対象システムの概念の本質を端的に表しており，システムの本質を理解するには最も適したモデルであると考えられる．しかし，2項関係の情報をシステムの内部状態として保持するためには，定義域として有限の情報を扱うために部分関数によるモデル化が現実的なモデルとして有効である場合が多い．部分関数によるモデル化では，定義域における未定義項の関数適用を考慮する必要があるので仕様記述は複雑になる傾向がある．また，関係による仕様記述では，リレーショナルデータベースにおける表に対して最も親和性の高い記述となっている．

全域関数，部分関数，関係への仕様記述の流れは，仕様記述を構築するための一つの指針を示してはいるが，仕様記述者は常にこのような過程をとるとは限らない．本章で提示した変換技術は，全域関数と部分関数の間，および部分関数と関係の間で，仕様記述を系統的に書き換えることを可能にしている．変換技術の提示は，仕様記述者に対して，仕様記述者が自由に思い付いたモデルで仕様を記述した後で，より良い別のモデルに仕様記述を書き換えるためのヒントを与える．

仕様の修正作業の過程

システムの仕様を修正する場合は，システムの仕様の概念が最も明確に現れている仕様記述で検討することが望ましい．関係の仕様記述はプログラムに近い仕様記述であり，また，部分関数による仕様記述は先に述べたように定義域における未定義項の関数適用を考慮する必要がある．したがって，対象システムの概念の本質を端的に現している全域関数の仕様記述で仕様の修正を行い，その後，部分関数化さらに関係化という過程が有効な手段であると考えられる．

仕様の修正作業を行うときには，全域関数と部分関数の変換における補助関数は既に定義されているので，全域関数で仕様を修正した後の変換作業は，全て記号処理として変換を取り扱うことができる．仕様記述者は，仕様の修正を考えやすいモデルを用いて，仕様の修正に本質的な作業に集中することができる．仕様の修正に適したモデルで検討し，プログラムに近い仕様記述は形式的な変換で得ることにより，仕様の修正に伴うエラーの混入を防ぐことが期待できる．

4. 階層的機能分割

大規模なソフトウェア開発において機能分割は重要な課題であり，その良否は仕様記述からプログラミングコードにまで影響を与える．機能分割は仕様記述の過程において，十分な分析を行う必要がある．機能分割は最初から最適な分割が得られるわけではなく，いろいろな視点から機能分割を試み，仕様記述者による試行錯誤の結果，最終的な分割を得ることができる．仕様記述者が経験的に行っている機能分割の過程の分析を行い，それを形式化し手法として提示することにより機能分割の支援を図る [19][20]．本章では，操作の仕様記述モデルの変換として，仕様記述の階層的な機能分割を行う手法について述べる．

4.1 概要

階層的機能分割とは「階層的な構造を持たないシステムの仕様記述から，階層的に機能分割された仕様記述を得ること」である．本手法では，階層的機能分割は

1. 機能分割，と
2. 階層化

の2つの部分に分かれている．機能分割では，論理演算子を用いたスキーマ合成により定義された操作のスキーマに対して，形式的に機能分割の仕様記述を得る

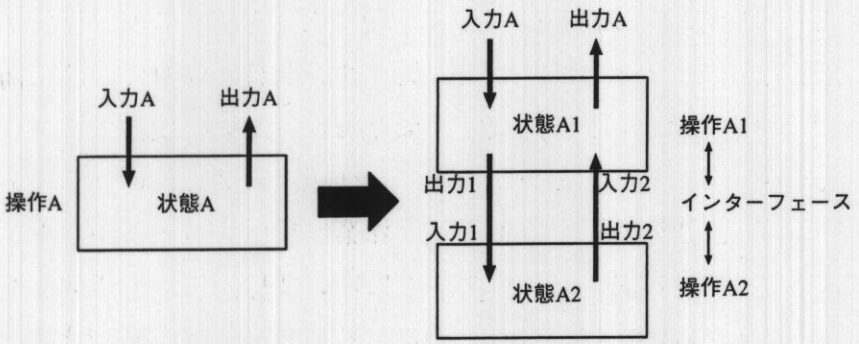


図 4.1 階層的機能分割の概念

手法を与える．また階層化では，論理演算子を用いたスキーマ合成の仕様記述に対してインタフェースを介した階層的な仕様記述に変換する手法を与える．

図4.1は階層的機能分割の概念を表している．図4.1では，操作Aの仕様記述を操作A1と操作A2に，階層的な分割を行っている．操作A1と操作A2は，出力1と入力1，および出力2と入力2の入出力インターフェースを介してデータを交換する．操作の間には従属関係があり，操作A1を上位操作，操作A2を下位操作とここでは呼ぶ．また，本手法では操作間のデータ交換は，インタフェースの入出力を介してのみ行い，操作間では変数を共有しないモデルを想定している．したがって，各操作は，他の操作の内部状態から独立しており，入力，出力，内部状態から構成されている．

4.2 機能分割

本節では機能分割の手法について述べる．まず，機能分割の基本アイデアについて説明し，その後，分割手法を示す．また，具体例として予約管理システムの仕様記述に対して機能分割を試みる．

4.2.1 基本アイデア

本手法では，操作の機能と操作スキーマの制約の記述を同一視し，記号上の操作により分割を行う．操作スキーマにおける「同じ制約の記述を1つのスキーマで記述すること」を基準として，スキーマの分割を行う．

Zでは，操作の機能を，操作の入力と出力，および操作実行前後の状態の間の関係を記述することにより表現する．操作スキーマでは，一般的に，入力と出力，および操作実行前後の状態を変数として宣言し，述語部において変数間の関係を制約として記述する．操作の機能を記述することは，操作スキーマの制約を記述することに相当する．したがって，ここでは操作の機能と操作スキーマの制約の記述を同一視することにする．

本手法では，論理演算子を用いたスキーマ合成により定義された操作スキーマに対する分割手法を提示する．操作の機能と操作スキーマの制約の記述を同一視することにより，機能分割を操作スキーマのスキーマ分割として捉える．スキー

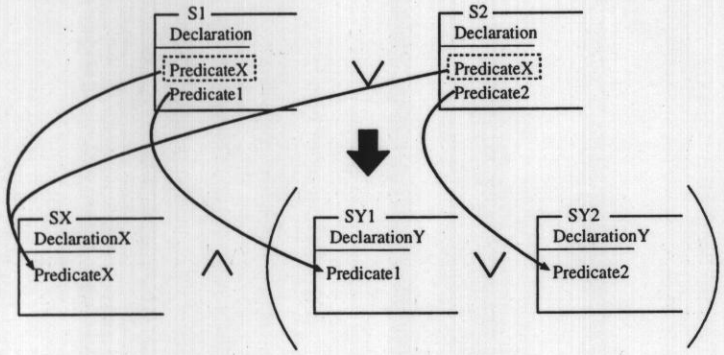


図 4.2 スキーマ分割のアイデア

マ分割は，図4.2で示すように，スキーマの述語に現れる同じ制約を1つのスキーマで記述することを基準としている．この分割は，Zにおける記号上の操作により形式的に行うことができる．

4.2.2 変換手法

本項では，機能分割の手法について述べる．分割は，スキーマ間の述語部において同じ制約を1つにまとめ，操作スキーマを再定義することにより実現する．

まず，分割の対象となるスキーマ S は，複数のスキーマの合成により以下のように定義されていると仮定する．

$$S \mathrel{\widehat{=}} S_1 op_1 S_2 op_2 \cdots op_{n-1} S_n \quad (n \geq 2)$$

とする．ただし，S_i はスキーマを表し，op_i はZで定義されているスキーマ合成の演算子を表す．また，S_i の間で述語部に同じ制約をもつスキーマを S_j とする．

分割は，各 S_j を再定義することにより実現する．S_j は，以下のように再定義する．

$$S_j \mathrel{\widehat{=}} SX \wedge SY_j$$

ここで，SX は，着目した同じ制約を1つのスキーマとして定義したものである．SX の宣言部では，述語部で用いられている変数が全て宣言されている必要があ

る．また，SY_j は S_j から着目した同じ制約，つまり SX の述語を削除したスキーマである．各 S_j に対して，このように再定義することによりスキーマの分割を行う．

例えば，図 4.2の例では，

$$S \triangleq S1 \vee S2$$

で定義されたスキーマ S に対して，分割は，

$$S \triangleq (SX \wedge SY1) \vee (SX \wedge SY2)$$

となる．これは，以下のように変形することができる．

$$S \triangleq SX \wedge (SY1 \vee SY2)$$

4.2.3 適用例

本項では，具体例として，予約管理システムの仕様記述に対して機能分割の適用を試みる．2.2節では，予約の登録を行う操作スキーマは

$$Entry \triangleq EntryOk \vee InvalidInput \vee AlreadyReserved$$

と定義した．$Entry$ を構成する各スキーマの述語部の間には，同じ制約が 2 つ存在する．

1. $tf' = tf$ （$InvalidInput$ と $AlreadyReserved$ の間）

2. $(st?, et?) \in RTime$ （$EntryOk$ と $AlreadyReserved$ の間）

ここでは，まず，1. の制約について分割を行う．次に，1. で分割した仕様記述に対して，2. の制約についての分割を行い，最終的に分割された仕様記述を得る．

分割 1

操作スキーマ $Entry$ に対して $InvalidInput$ と $AlreadyReserved$ の間にある制約 $tf' = tf$ に着目した分割を行う．この述語をスキーマとして記述すると，以下のようになる．

DbNoChange

$\Delta State$

$tf' = tf$

DbNoChange により抽出されるスキーマ *InvalidInput* と *AlreadyReserved* は以下のようになる．

InvalidInputY

$Input?;\ Output!$

$(st?, et?) \notin RTime$

$r! = invalid\_input$

AlreadyReservedY

$\Delta State;\ Input?;\ Output!$

$(st?, et?) \in RTime$

$(st?, et?) \in \mathrm{dom}(tf(d?)) \lor$

$\quad \mathrm{dom}(tf(d?)) \cup \{(st?, et?)\} \notin RTimeSet$

$r! = already\_reserved$

したがって，*Entry* に対して *DbNoChange* を用いて分割されたスキーマ *Entry*1 は

$$Entry1 \mathrel{\widehat=} EntryOk \lor DbNoChange \land InvalidInputY \lor DbNoChange \land AlreadyReservedY$$

となる．これは，以下のように変形することができる．

$$Entry1Alt \mathrel{\widehat=} EntryOk \lor DbNoChange \land (InvalidInputY \lor AlreadyReservedY)$$

DbNoChange に関する操作は $(InvalidInputY \lor AlreadyReservedY)$ であることが，*EntryAlt* の定義から一目で読みとることができる．したがって，これは *DbNoChange* に着目した仕様記述とみることができる．*Entry* では，操作は *EntryOk*，*InvlalidInput*，*AlreadyReserved* のいずれかという定義であり，*EntryAlt* は *Entry* から仕様記述の観点を変えた仕様記述であるということがわかる．

分割 2

分割 1 により得られた操作スキーマ *Entry*1*Alt* に対して *EntryOk* と *AlreadyReservedY* の間にある $(st?, et?) \in RTime$ に着目した分割を行う．この述語をスキーマとして記述すると，以下のようになる．

ValidInput

$Input?$

$(st?, et?) \in RTime$

ValidInput により抽出されるスキーマ *EntryOk* と *AlreadyReservedY* は以下のようになる．

EntryOkY

$\Delta State; Input?; Output!$

$(st?, et?) \notin \mathrm{dom}(tf(d?))$

$tf' = tf \oplus \{d? \mapsto (tf(d?) \cup \{(st?, et?) \mapsto n?\})\}$

$r! = ok$

AlreadyReservedZ

$\Delta State; Input?; Output!$

$(st?, et?) \in \mathrm{dom}(tf(d?)) \lor$

$\quad \mathrm{dom}(tf(d?)) \cup \{(st?, et?)\} \notin RTimeSet$

$r! = already\_reserved$

したがって，*Entry*1*Alt* に対して *ValidInput* により分割したスキーマ *Entry*2 は

$$\begin{aligned} Entry2 \triangleq\ & ValidInput \wedge EntryOkY \vee \\ & DbNoChange \wedge \\ & (InvalidInputY \vee ValidInput \wedge AlreadyReservedY) \end{aligned}$$

となる．これは，以下のように変形することができる．

$$\begin{aligned} Entry2Alt1 \triangleq\ & InvalidInputY \wedge DbNoChange \vee \\ & ValidInput \wedge \\ & (EntryOkY \vee AlreadyReservedZ \wedge DbNoChange) \end{aligned}$$

さらに，以下のようにも変形することができる．

$$\begin{aligned} Entry2Alt2 \triangleq\ & InvalidInputY \wedge EntryOkY \vee \\ & DbNoChange \wedge \\ & (InvalidInputY \vee ValidInput \wedge AlreadyReservedZ) \end{aligned}$$

*Entry*2*Alt*1 は *ValidInput* に着目した仕様記述，また *Entry*2*Alt*2 は *DbNoChange* に着目した仕様記述である．どちらも *Entry* と同じ仕様記述であるが，全く観点の違う仕様記述が得られる．これは，仕様記述の分析を行う場合，有効な手段となり得るであろう．

4.2.4 評価

本項では，機能分割の手法の評価を行う．

本手法では，機能と制約を同一視し，複数のスキーマの合成により定義されたスキーマを分割することで機能分割を実現した．分割は記号上の形式的な操作で処理することができる．しかし，仕様記述は複数のスキーマに分割されて定義されているという前提があるので，１つのスキーマから分割することはできない．スキーマ合成で表現されたスキーマも１つのスキーマとして展開することができるので，１つのスキーマから分割する手法を開発すれば一般的な手法を提示することができる．

ところで，本手法では分割を行う基準は，

- 複数のスキーマ間における同じ機能は1つのスキーマで表現すること

であった．本手法では,「同じ機能」の基準を「字面が同じである」とした．しかし，Zでは仕様記述の字面には現れない暗黙の制約の存在や，数学の概念を基にした記号が多く存在する．暗黙の制約とは，例えばスキーマ

$n : \mathbb{Z}$
$n < 5$

において，変数の宣言における制約，つまりここでは $n > 0$ という制約（constraint）をいう．制約記述の対象を

- 暗黙の制約を陽に示した記述や，
- 数学記号の意味を定義にたち戻って示した記述

にすることにより，字面からだけの基準ではなく，より有益な基準を提示できる可能性がある．

また，分割において，分割の記述を一意に求めることができるかという，一意性の問題がある．この問題は，制約記述の一意性に帰着される．制約記述の一意性を得るために，例えば，制約記述において，論理和標準形（選言標準形）といった標準形を定義する必要がある．この標準形の制約記述から，一意に分割を求めるアルゴリズムを開発すれば分割の一意性は解決することができる．

以上の議論より，一般的な機能分割の手法を提示するための今後の課題として，

- 同じ機能の基準の設定，
- 制約記述の標準形の定義，
- 標準形からの分割アルゴリズムの開発

が挙げられる．

4.3 階層化

本節では階層化の手法について述べる．まず，階層化の基本アイデアについて説明し，その後，階層化の手法を示す．また，具体例として予約管理システムの仕様記述に対して階層化を試みる．

4.3.1 基本アイデア

まず，階層構造の前提条件について述べる．本手法では，階層として図 4.3のような階層構造を想定している．このような階層を構成する前提として，階層間の内部状態，およびインターフェース間の依存関係について以下の条件を与える．

1. 内部状態Aと内部状態Bにおいて共有する変数は存在しない．

2. インターフェース間の入出力において，出力の集合は入力の集合の部分集合である．

1. の条件は階層間の内部状態についての依存関係で，操作間のデータ交換がインターフェースの入出力を介してのみ行われることを保証する．各操作は入力と出力，および内部状態を用いて定義するので，他の操作の内部状態を外部参照することはない．

2. の条件はインターフェース間についての依存関係で，インターフェース間で想定している出力に対する入力の操作が定義されていることを保証する．出力の

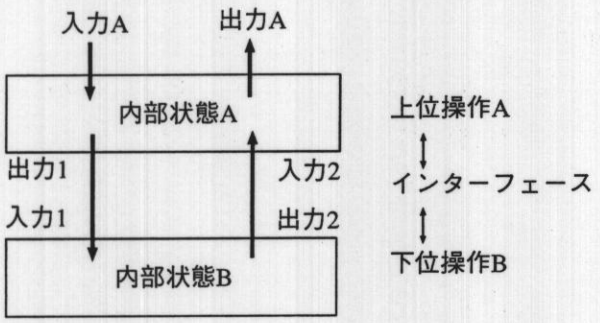


図 4.3 階層構造

集合とは，操作で想定している全ての出力の集合であり，入力の集合とは，操作で想定している全ての入力の集合である．この条件は，例えば，上位操作Aからの出力1に対して，下位操作Bの出力2が定義されていることを保証する．

本手法では，上で述べたような階層構造の仕様記述を，Zにおけるスキーマ合成の1つであるパイプ演算子（≫）を用いて表現する．例えば $S1 \gg S2$ という表現は，直観的には，スキーマ $S1$ の出力と スキーマ $S2$ の入力を結び付けることを意味する．したがって，ここでは $S1$ が上位操作，$S2$ が下位操作の仕様記述となる．

本手法では，階層的な仕様記述を得る手段として，論理積の演算子によるスキーマ合成による仕様記述からパイプ演算子による仕様記述に変換する手法を提示する．本手法により得られる階層構造では，図4.3における入力Aと出力1，および出力Aと入力2のデータは同じであるとする．つまり，入力Aのデータはデータ構造を変換せずに出力1に出力され，同様に，入力2のデータはデータ構造を変換せずに出力Aに出力される．次項では，上で示した前提条件を満足する階層構造を得るための条件を提示し，$S1 \wedge S2$ の仕様記述から $S1 \gg S2$ の仕様記述へ変換する方法を示す．

4.3.2 変換手法

本項では，階層化の手法について述べる．階層化は，論理積の演算子によるスキーマ合成からパイプ演算子によるスキーマ合成に変換することにより得られる．まず，階層構造を得るための変換条件を示し，その後，変換の方法を示す．

図4.4は変換の概要を表している．変換の対象となるスキーマは，論理演算子によるスキーマ合成 $Op1 \wedge Op2$ である．ここで，$Op1, Op2$ は以下のような構成とする．

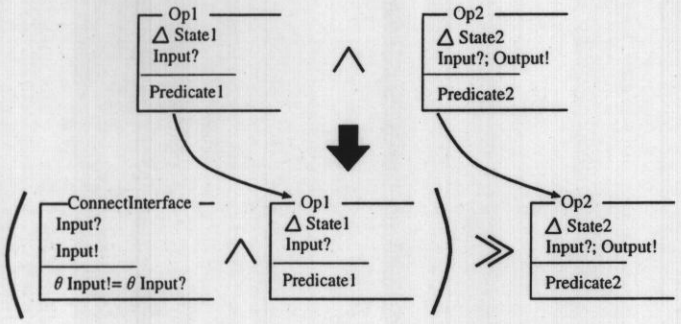
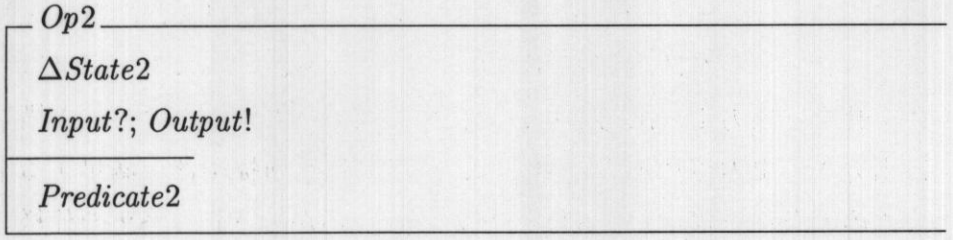


図 4.4 パイプ演算子を用いた階層化の記述



*State*1, *State*2 は内部状態，*Input*? は入力，*Output*! は出力の変数を表すスキーマである．また，*Predicate*1, *Predicate*2 は，それぞれの操作の制約を表すスキーマとする．

ここで，変換の条件を以下に示す．

1. *State*1 と *State*2 の間には変数の名前の衝突はない．
2. *Op*1 と *Op*2 の事前条件である pre *Op*1 と pre *Op*2 の間で，pre *Op*1 ⇒ pre *Op*2 が成立する．

これは，前項で示した変換の条件をZで表記したものである．変換を行うには，*Op*1 において入力とインタフェースへの出力を関係付けるスキーマを定義する必要がある．そのスキーマを以下に示す．

ConnectInterface

$Input?$

$Input!$

$\theta Input? = \theta Input!$

スキーマ *ConnectInterface* を用いてパイプ演算子によるスキーマ

$$(ConnectInterface \wedge Op1) \gg Op2$$

を得る．

4.3.3 適用例

本項では，具体例として，予約管理システムの仕様記述に対して階層化の適用を試みる．4.2.3項における $Entry2Alt1$ の定義のうち

$$ValidInput \wedge (EntryOkY \vee AlreadyReservedZ \wedge DbNoChange)$$

の部分について変換を試みる．これは，

$$Op2 \triangleq (EntryOkY \vee AlreadyReservedZ \wedge DbNoChange)$$

として

$$ValidInput \wedge Op2$$

を変換の対象とみなす．このスキーマの構成は，変換の対象となるスキーマ構成を満たしている．

ところで，変換条件について考える．まず，*ValidInput* に内部状態の変数は存在しないので，内部状態の変数の名前の衝突はない．また，*ValidInput*, $Op2$ の事前条件 pre *ValidInput*，pre $Op2$ は両方とも，

Precondition

$Input?$

$(st?, et?) \in RTime$

である．したがって，

$$\text{pre } ValidInputTime \Rightarrow \text{pre } Op2$$

を満たしている．したがって，変換条件は満足していることが示せた．

したがって，インタフェースを結び付けるスキーマ

ConnectInterface

$Input?$

$Input!$

$\theta Input? = \theta Input!$

を定義し，以下のスキーマを得る．

$$(ValidInput \wedge ConnectInterface) \gg (EntryOkY \vee AlreadyReservedZ \wedge DbNoChange)$$

したがって，$Entry2Alt1$ は，

$$EntryLayer \mathrel{\widehat=} InvalidInputY \wedge DbNoChange \vee (ValidInput \wedge ConnectInterface) \gg (EntryOkY \vee AlreadyReservedZ \wedge DbNoChange)$$

と変換することができる．*EntryLayer* の仕様の構造を図 4.5に示す．

4.3.4 評価

本項では，階層化の手法の評価を行う．

本手法では，論理積によるスキーマ合成 $S1 \wedge S2$ の表現からパイプ演算子によるスキーマ合成 $S1 \gg S2$ に変換する条件を明示して，変換の手順を示した．階層化は記号上の形式的な操作で処理することができる．また，階層化が可能な条件は，Z上の述語で表現されており，論理的に判断することができる．したがって，この条件はZ仕様記述において階層的な記述を行う基準となり得る．また，

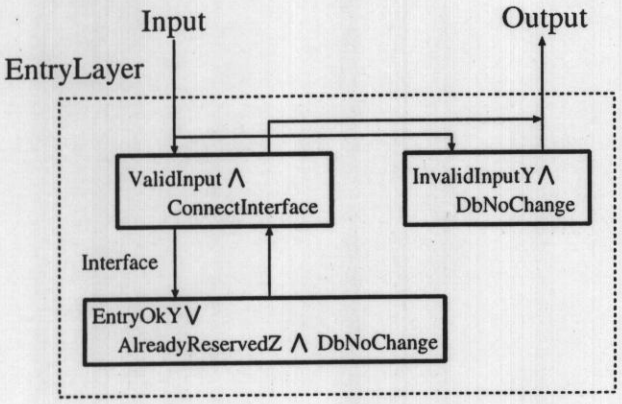


図 4.5 EntryLayer の階層構造

この基準は階層化を行う際の指針となる．機能分割された仕様記述から，階層化の条件を満足するように仕様記述を変形することにより，階層化の仕様記述を得ることができる．このようにして得られた仕様記述では，下位操作は，上位操作からの入力に対する下位操作からの出力を保証する．

5. システム状態不変条件の変換

ソフトウェア開発における仕様記述において，対象システムが保持しなければならないシステム状態不変条件を理解することは重要である．システム状態不変条件は，操作の仕様記述の制約として，暗黙に記述されていることがある．本章では，形式的仕様記述言語Zによる仕様記述において，操作の仕様記述からシステム状態不変条件を抽出する方法を，具体例を用いて示す．システム状態不変条件を明示することにより，対象システムに対する理解が深まると共に，システム状態の仕様記述と操作の仕様記述の間の一貫性を形式的に検証することを可能にする [21][22].

5.1 概要

本手法では，対象システムが保持しなければならない状態不変条件に着目し，操作の仕様記述からシステム状態不変条件を抽出する手法の確立を目指す．図5.1は，本手法の概念を表している．図に示すように，仕様記述は「システム状態」と「操作」から構成されているとする．システム状態の仕様記述は，見方を変えれば，システムの状態が常に満足しなければならない制約，つまり，システム状態不変条件を表現しているとみなすことができる．ところで，システム状態不変条件が，図5.1の仕様記述（A）のように，システム状態の仕様記述としてではなく，操作の仕様の制約として暗黙的に表現されていることがある．これは，操作の仕様記述を行った時，システムの状態が満足しなければならない条件を，仕様

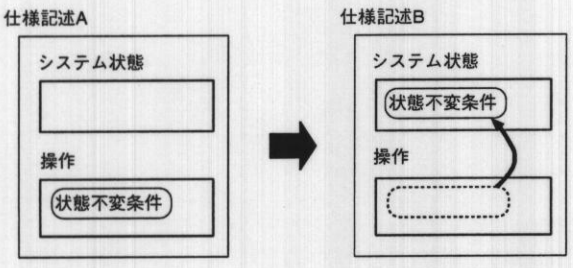


図 5.1 状態不変条件の抽出

記述者が暗黙のうちに想定し，その条件を操作における仕様として表現したと考えられる．本手法では，操作の仕様記述で表現されている暗黙のシステム状態不変条件（仕様記述 A）を，システム状態の仕様記述（仕様記述 B）に形式的に変換する手法を開発することを目標としている．

システム状態不変条件を明示的に記述することは重要である．システム状態不変条件が，操作の仕様記述として局所的に表現されていれば，それがシステム全体を支配する制約として認識されずに，別の操作の仕様記述において，その制約に矛盾する仕様を記述する可能性がある．システム状態不変条件が明示的に表現されていれば，操作の仕様記述が，システム状態不変条件を満足するという仕様記述間の一貫性を検証することができる．また，操作の仕様記述において，システム状態不変条件を満足する制約を導出することもできる．ところで，システム状態不変条件はどの様にして得られるのだろうか．これは，仕様記述の過程において，対象システムに対する理解が深まるにつれて，発見的に得られると考えられる．本手法は，操作の仕様記述において，システム状態不変条件を発見する過程の明示化および形式化に相当する．

5.2 基本アイデア

本節では，仕様記述の構成およびシステム状態における暗黙の制約の基本概念について述べる．

5.2.1 仕様記述の構成

先に述べたように，本論文では対象システムの仕様記述は「システム状態」と「操作」から構成されているとみなす．システム状態の仕様記述は，システムが保持すべき内部状態を表現し，操作の仕様記述はシステムの動的な振舞いを表現する．

システム状態を定義することは，システムが存在し得るシステム状態の空間を明示することに相当する．図 5.2はシステム状態の仕様記述の概念を表している．システムの状態は，操作を実行することにより，システム状態空間内を遷移する．ところで，システム状態の記述は，見方を変えれば，システムの状態が常に満足

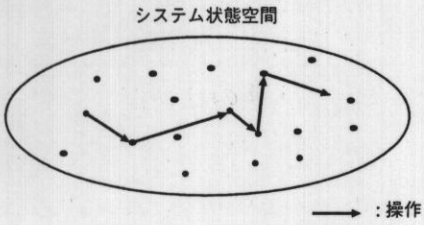


図 5.2 システム状態の仕様記述の概念

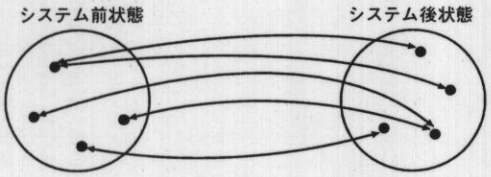
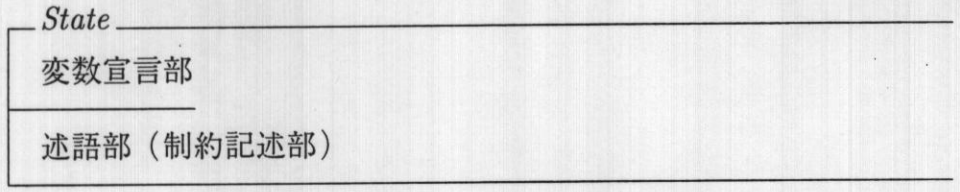


図 5.3 操作の仕様記述の概念

しなければならない制約，つまり「システム状態不変条件」を表現しているとみなすことができる．システムの状態は，システム状態不変条件を満足する範囲で，状態遷移を繰り返す．

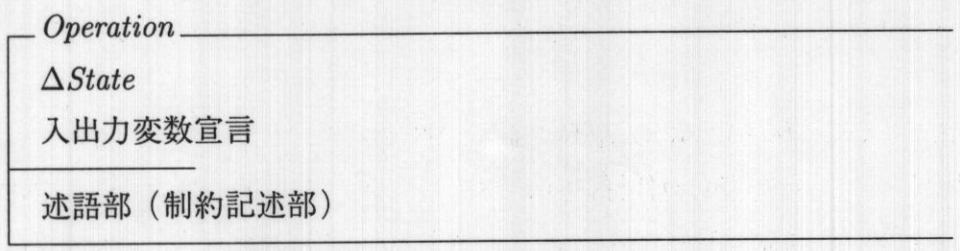
操作を定義することは，操作を実行する前のシステム状態（システム前状態）と，操作を実行した後のシステム状態（システム後状態）の関係を明示することに相当する．図 5.3は操作の仕様記述の概念を表している．操作の実行前後のシステム状態は，システム状態空間上にあるので，操作の定義におけるシステム前状態およびシステム後状態は，システム状態不変条件を満足しなければならない．

ここで，仕様記述の概念と Z 仕様記述との対応について考える．Z では，一般的に，システム状態を「状態スキーマ」，そして，操作の仕様記述を「操作スキーマ」を用いて表現する．スキーマは，Z において最も特徴的な図式表現である．以下に，状態スキーマの一般的な形を示す．



スキーマは変数宣言部と述語部から構成されており，*State* はスキーマの名前を表す．変数宣言部では，システムの内部状態を表現するための変数の宣言を行う．述語部では，変数宣言部で宣言した変数および変数間の制約を論理式で記述する．述語部の記述は，システムの内部状態の変数が常に保持しなければならない制約，つまり，システム状態不変条件の記述に相当する．したがって，状態スキーマは，変数宣言部で宣言した変数上のシステム状態空間を表現していると解釈することができる．

次に，操作スキーマの一般的な形を示す．



変数宣言部では，操作実行前と操作実行後の状態スキーマ（$\Delta State$）と入出力の変数の宣言を行っている．述語部では，操作実行前後のシステム状態と入出力の関係を記述する．状態スキーマの述語部の記述は，*Operation* の述語部に展開され，システム前状態とシステム後状態はシステム状態不変条件を満足することを表現している．仮に，*Operation* の述語部においてシステム状態不変条件と矛盾する制約を記述すれば，述語部の表現は偽（*false*）となる．

5.2.2 システム状態における暗黙の制約

システム状態における暗黙の制約とは,「システム状態の仕様記述には明示的に表現されていない制約」である．対象システムの仕様を記述する時，初めから対象システムの分析と理解が完全に行われることはない．仕様記述の過程および仕

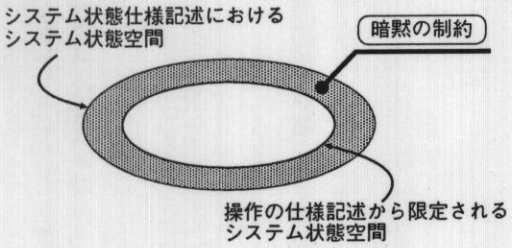


図 5.4 システム状態における暗黙の制約

様記述に対する議論を行う過程を通して，対象システムの理解が深まる．特に，システムの状態が常に保持しなければならないシステム状態不変条件を，明示的に記述することは困難な作業である．ところで，操作の仕様記述を行う時，仕様記述者は，システムの状態が満足しなければならない条件を暗黙のうちに想定し，その条件が操作における仕様として表現される場合がある．本論文では，暗黙の制約の中で,「操作の制約記述として表現されているシステム状態不変条件」に着目する．

図 5.4はシステム状態における暗黙の制約の概念を表している．図において，外側の楕円は，システム状態の仕様記述におけるシステム状態空間を表す．内側の楕円は，操作の仕様記述から限定されるシステム状態空間を表す．二つの楕円の間は，システムの最初の状態から操作を何回繰り返しても到達しないシステム状態の空間である．この空間を狭める制約が，暗黙の制約に相当する．

ところで，暗黙の制約による空間が，システム状態として到達しないだけで，システム状態としては妥当な空間であるか，それとも，システム状態として到達してはいけない空間であるかを判断することは重要である．もし，システム状態として到達してはいけない空間であれば，暗黙の制約がシステム状態不変条件として明示されることが望ましい．このような判断を記録することは，後の保守やシステム再構築の際のシステムの分析や理解に有益な情報となりうる．

Zによる仕様記述では，システム状態不変条件は状態スキーマの述語部の制約記述に相当する．したがって，操作の仕様記述からシステム状態における暗黙の制約を明示することは，操作スキーマの制約記述からシステム状態不変条件を抽

出し，その制約を状態スキーマの制約として記述することに相当する．

5.3 システム状態不変条件の抽出

本節では，具体例として「予約管理システム」の仕様記述を用いて，操作の仕様記述からシステム状態不変条件を抽出する手法を示す．

State

$$db : DATE \longrightarrow (Time \times Time \pfun NAME)$$

InitState

$$State$$

$$\forall d : \mathrm{dom}(db) \bullet db(d) = \varnothing$$

EntryInput

$$d? : DATE$$
$$st?, et? : Time$$
$$n? : NAME$$

Entry

$$\Delta State$$
$$EntryInput$$

$$st? \ \mathsf{LT} \ et?$$
$$\forall st, et : Time \mid (st, et) \in \mathrm{dom}(db(d?)) \bullet (et \ \mathsf{LE} \ st? \lor et? \ \mathsf{LE} \ st)$$
$$db' = db \oplus \{d? \mapsto (db(d?) \cup \{(st?, et?) \mapsto n?\})\}$$
$$(st?, et?) \notin \mathrm{dom}(db(d?))$$

5.3.1 操作仕様記述の分析

ここでは，先に定義した操作スキーマ *Entry* の述語部の制約記述について分析し，システム状態における暗黙の制約について考える．*Entry* の述語部は 4 つの述語から構成されており，スキーマ全体としては，それぞれの述語は連言でつながっている．以下，各述語の意味を分析し，その後，システム状態の制約との関係について検討する．

1. $st?$ LT $et?$
   入力の予約の開始時刻が終了時刻より小さい，つまり，入力の予約時間が妥当であることを表現している．

2. $\forall st, et : Time \mid (st, et) \in \mathrm{dom}(db(d?)) \bullet (et \text{ LE } st? \vee et? \text{ LE } st)$
   入力の予約時間が，既に予約されている予約時間と重ならないことを表現している．この制約記述は，1. の予約時間の制約（入力の予約の開始時刻が終了時刻より小さい）を前提としている．

3. $db' = db \oplus \{d? \mapsto (db(d?) \cup \{(st?, et?) \mapsto n?\})\}$
   入力の予約時間をシステム状態のデータベースに加えることを表現している．

4. $(st?, et?) \notin \mathrm{dom}(db(d?))$
   3. において，データベースに入力と同じ予約時間が，既に登録されていた場合に，予約の登録が行われてしまうという問題を防ぐための制約である．

ここで，システム状態の制約との関係について考える．システム状態の遷移を表現している述語は，3. の述語である．これは，システム前状態に予約データ $(st?, et?) \mapsto n?$ を加えることを表現している．本システムにおいて，予約時間の追加は *Entry* を通して行われるとすれば，*Entry* における予約データの制約は，システム状態の制約として捉えることができる．上の 1.,2. の述語の意味は，システム状態の制約として，次のように解釈することができる．

1. $st?$ LT $et?$
   システム状態で保持する予約時間の制約である．

2. $\forall st, et : Time \mid (st, et) \in \mathrm{dom}(db(d?)) \bullet (et \ \mathsf{LE}\ st? \lor et? \ \mathsf{LE}\ st)$
   システム状態で保持する予約時間の集合の制約（予約時間の非重複性）である．

ところで，仕様記述者は，*Entry* の仕様を記述した時，上で示したシステム状態の制約を暗黙のうちに考慮し，その制約を操作の制約として記述したと考えられる．操作の仕様の制約を，システム状態の仕様の制約として記述することは，仕様記述者が暗黙のうちに想定したシステム状態の制約を明示することに相当する．

5.3.2 システム状態不変条件の記述

ここでは，先に分析した *Entry* における「予約時間」と「予約時間の集合」に関する制約をシステム状態不変条件，つまり，システム状態の制約として記述する．

まず,「予約時間」の制約をシステム状態の制約として記述する．

StateAlt1

$$
\begin{array}{|l}
\hline
\mathit{State} \\
\hline
\forall d : \mathrm{dom}(db) \bullet \\
\quad \forall st, et : Time \mid (st, et) \in \mathrm{dom}(db(d)) \bullet st \ \mathsf{LT}\ et \\
\hline
\end{array}
$$

上のスキーマの述語部は，予約状態を管理するデータベースにおいて，予約時間には $st \ \mathsf{LT}\ et$ ，すなわち,「予約の開始時刻が終了時刻より小さい」という制約があることを表現している．

次に,「予約時間の集合」の制約をシステム状態の制約として記述する．

StateAlt2

$$
\begin{array}{|l}
\hline
\mathit{StateAlt1} \\
\hline
\forall d : \mathrm{dom}(db) \bullet \\
\quad \forall st1, et1, st2, et2 : Time \mid \\
\qquad (st1, et1) \in \mathrm{dom}(db(d)) \land \\
\qquad (st2, et2) \in \mathrm{dom}(db(d)) \bullet (et1 \ \mathsf{LE}\ st2 \lor et2 \ \mathsf{LE}\ st1) \\
\hline
\end{array}
$$

上のスキーマの述語部は，予約状態を管理するデータベースにおいて，予約時間の集合には

$$et1 \ \mathsf{LE} \ st2 \lor et2 \ \mathsf{LE} \ st1$$

すなわち,「予約時間が重なることがない（予約時間の非重複性）」という制約があることを表現している．

以上のようにして，*Entry* の操作の制約を状態スキーマ *StateAlt2* として表現することができた．そこで，*Entry* を *StateAlt2* を用いて再定義する．

EntryAlt

$\Delta StateAlt2$

$EntryInput$

$db' = db \oplus \{d? \mapsto (db(d?) \cup \{(st?, et?) \mapsto n?\})\}$

$(st?, et?) \notin \mathrm{dom}(db(d?))$

上のスキーマの述語部は，*Entry* の述語部から，予約時間の制約と予約時間の集合の制約を削除したものである．ところで，削除した制約は *EntryAlt* の事前条件

$$\mathrm{pre} \ EntryAlt$$

から導出することができる．

5.4 抽出手法に関する考察

操作の仕様記述からシステム状態不変条件を抽出する手法として，変換規則により形式的に与える方法について考察する．変換の対象となる仕様記述の前提を示し，変換規則を提示した後で変換規則の適用可能性について述べる．

5.4.1 前提

まず，変換規則を提示するにあたり，対象としているシステム状態の仕様記述，および操作の仕様記述を示す．

システム状態の仕様記述の前提

システム状態の仕様記述では，システムが保持する変数は1つで，その型はある型（X）の集合であるとする．具体的には以下のようなスタイルの仕様記述である．

State

$$S : \mathbb{P}\, X$$

$$\overline{P}$$

但し，$\overline{P}$ は述語の集まりを表している．また，システムの初期状態 (*InitState*) では，S は空集合とする．

InitState

State

$$S = \varnothing$$

操作の仕様記述の前提

操作の仕様記述では，システム状態 (*State*) で定義されている変数 S において，操作の仕様として $S' = S \cup \{T\}$ の形をした式が存在するものとする．具体的には以下のようなスタイルの仕様記述である．

Op

$$\Delta State$$

$$\overline{D}$$

$$\overline{P}$$

$$S' = S \cup \{T\}$$

但し，$\overline{D}$ は変数宣言の集まりを表し，T の型は X とする．

5.4.2 変換規則

操作の仕様記述において以下のパターンの述語について変換規則を与える.

1. $P(T, \overline{C})$
   T と定数項の集まり $\overline{C}$ で構成されている述語

2. $P(S', \overline{C})$
   システムの後状態 (S') と定数項の集まり $\overline{C}$ で構成されている述語

3. $P(S, \overline{C})$
   システムの前状態 (S) と定数項の集まり $\overline{C}$ で構成されている述語

変換規則を以下に示す.
[変換規則 (1)]

$$P(T, \overline{C}) \quad \rightsquigarrow \quad \forall e : S \bullet P(T, \overline{C})[e/T]$$

システム状態に付け加えられる要素の制約は，システム状態の全ての要素に反映されると考えられる.
[変換規則 (2)]

$$P(S', \overline{C}) \quad \rightsquigarrow \quad P(S', \overline{C})[S/S']$$

操作の仕様記述におけるシステムの後状態の制約は，そのままシステム状態の制約に反映されると考えられる.
[変換規則 (3)]

$$\#S < N \quad \rightsquigarrow \quad \#S < N + 1$$

システムの前状態の制約では，システム状態の変数の要素の数に対する制約について変換規則を示した.

5.4.3 適用例

以下の仕様記述に対して変換規則によりシステム状態不変条件の抽出を試みる．

State

$$nset : \mathbb{P}\,\mathbb{N}$$

Op

$$nset, nset' : \mathbb{P}\,\mathbb{N}$$
$$n? : \mathbb{N}$$

$$n? < 10$$
$$\#nset < 5$$
$$n? \notin nset$$
$$nset' = nset \cup \{n?\}$$

上記の仕様記述は，前提条件を満たしており，*Op* の述語部の 1 行目と 2 行目について，それぞれ変換規則 (1) と変換規則 (3) が適用可能である．変換規則を適用した結果，以下のシステム不変条件が抽出される．

$$\forall e : nset \bullet e < 10$$
$$\#nset < 6$$

上記の制約は，システム状態 *State* に対して操作 *Op* だけを適用することにより得られるシステム状態の性質である．この制約が本当にシステム状態として満たすべき性質であるかどうかは，仕様記述者が判断することである．

5.4.4 変換規則の拡張

前項では，操作の仕様記述において暗黙に表現されているシステム状態の制約を，変換規則により提示することが可能であることを示した．しかし，本変換規則だけでは予約管理システムの例で示した性質を抽出することはできない．

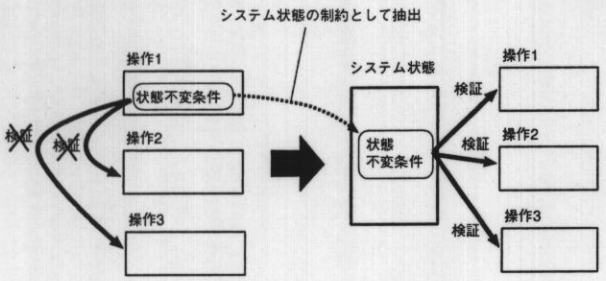


図 5.5 仕様記述間の一貫性の検証

変換規則の拡張として，システム状態の変数の適用パターン，および操作の仕様記述の述語のパターンの分析を行う必要がある．システム状態の変数として，

- 直積　　$S : \mathrm{P}(X \times Y)$
- 集合の集合　　$S : \mathbb{P}\,\mathbb{P}\,X$
- 複数の状態　　$S1 : X1; S2 : X2$

が考えられる．また，述語のパターンとしては，

- 述語 $P(T, S', S, \overline{V}, \overline{C})$ の分析

を行う方法を開発する必要がある．

5.5 評価

本節では，システム状態不変条件の抽出について，仕様記述変換の一貫性の検証，およびリバースエンジニアリングの観点からの考察を行う．

5.5.1 仕様記述間の一貫性の検証

仕様記述間の一貫性とは，システムが保持しなければならない状態不変条件を操作の仕様記述が満足していることをいう．図 5.5は，仕様記述間の一貫性の検証の概念を示している．もし，システム状態不変条件が操作の仕様記述の制約として表現されていれば，他の操作の仕様記述において，その制約の検証を行うこ

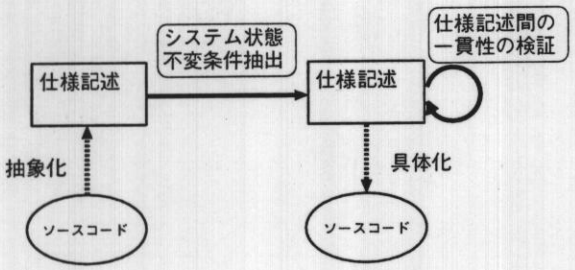


図 5.6 リバースエンジニアリング

とはできない．操作の仕様記述におけるシステム状態不変条件を，システム状態の制約として抽出すれば，全ての操作の仕様記述において一貫性の検証を行うことができる．

Zの仕様記述では，システム状態不変条件をシステム状態の制約として表現すれば，操作の仕様記述において，その制約を満たした事前条件および事後条件を導出することができる．したがって，システム状態と操作の仕様記述の間の一貫性の検証を形式的に行うことができる．このように，システム全体に関する制約が明示されていれば，仕様記述間の検証が可能となり，仕様記述の信頼性が向上する．

5.5.2 リバースエンジニアリング

システム状態不変条件の抽出技術のリバースエンジニアリングへの応用について考える．リバースエンジニアリングは，既存のシステムの仕様を保存した上でシステムの再構築を行う技術である．リバースエンジニアリングでは，ソースコードから仕様を抽出する技術と，仕様を分析する技術が重要である．本手法は，仕様を分析する際の有効な手段となり得ると考えられる．

ここでは，リバースエンジニアリングとして，図5.6に示されている過程を想定する．まず，既存のソースコードから抽象化された仕様記述を得る．抽象化された仕様記述を分析することにより，より最適な仕様記述を再構築した後，新たなソースコードを得る．ところで，ソースコードには，システム状態の不変条件が明示されている可能性は少なく，システム状態の制約を理解するには，ソース

コード全てを読む必要がある．一方，本手法を用いれば，操作の仕様記述から部分的に状態不変条件を抽出し，それらを合わせて分析することにより，システム状態不変条件を明示することができる．さらに，得られた仕様記述間の一貫性を検証することにより，信頼性の高い仕様記述を得ることが期待できる．

6. ダイアグラム間の一貫性検証

CASE ツールを使用したソフトウエア開発における重要な問題の一つに，構造化ダイアグラムを用いて記述した仕様について，意味的な一貫性の検証ができないということがある．仕様の一貫性について厳密な検証を行うために，CASE ツールに形式的手法を適用する．本章では，PAD (Process Action Diagram) が DFD (Data Flow Diagram) の詳細化 (refinement) であることの検証を例にとり，言語 Z を用いたダイアグラムの検証手法について述べる．Z を用いた形式的な仕様記述を適用すると，種類の異なるダイアグラム間の一貫性を検証することや，抽象度の異なるダイアグラムの正当性を保証することが可能になると考える．これらの検証や保証をすることは，ユーザー要求に対する仕様の正確さを向上させることにつながるものである [23][24][25][26].

6.1 はじめに

仕様の記述は，ソフトウエア開発における上流の工程の 1 つである．それゆえ，言語的な構造や規則の違反，曖昧・矛盾・不完全といった誤りが仕様の段階で含まれると，下流工程の開発に影響をおよぼし開発効率が低下する．仕様からこれらの違反や誤りを取り除くために,「構文」と「意味」といった 2 つの観点から仕様を検証する必要があると考える．ここで，構文の検証とは，仕様が記述する言語の構造や規則に違反していないことを確認することをいう．意味的な検証とは，仕様として記述したものに曖昧・矛盾・不完全といった誤りが存在せず，システムが満たすべき要件を満足していることを確認することをいう．

確認の方法として代表的なものに，デザインレビューやウォークスルー，コード・インスペクションなどがある．これらの確認方法は人の手によっているため，システム規模が増大するにつれて作業効率が低下する．これに対し，近年，確認を含めた開発作業全般をサポートするものとして CASE(Computer-Aided Software Engineering) ツールが開発された．

CASE ツールを使うと，仕様の構文についての検証が効率良く可能となる．CASE ツールを使ったソフトウエア開発では，仕様は構造化ダイアグラム（図）

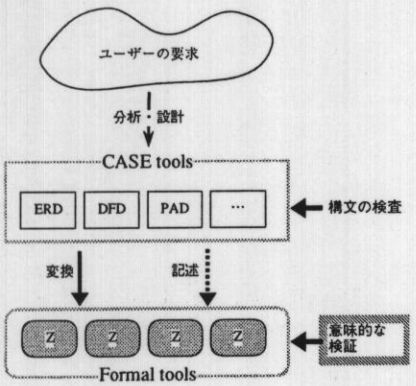


図 6.1 開発環境のモデル

などを用いて記述する．ダイアグラムを用いると，複雑な構造や処理を簡潔かつ明確に記述できる．ほとんどのツールは，ダイアグラムの構文を自動的に検証する機能を持つ．しかし，多くのツールではダイアグラムの意味についての検証をほとんど行うことができず，現在のシステム開発ではデザイン・レビューなどの方法で意味的な確認を行っている．これは，ツールに意味的な検証を支援する機能が欠けているためであると考える．すなわち，CASE ツールには構文を検証する機能は備わっているが，意味を厳密に記述する機能とその意味を検証する機能が欠けていると考える．

仕様について意味的な検証を行うには，形式的な仕様記述が有効であると考える．形式的な仕様記述言語を用いた仕様記述法は，意味について仕様を明確に記述でき，矛盾性や不完全性といった誤りについて厳密な論証ができる．

仕様を構文と意味の 2 つの観点から検証するために，本研究では，CASE ツールに形式的仕様記述法を適用した開発環境を提案する．図 6.1はこの環境をモデル化したものであり，CASE ツールと Formal ツールの 2 種類のツールから構成される．Formal ツールは，形式的な仕様記述言語を用いて仕様を記述し，仕様の誤りについて数学的に論証する機能を持つツールである．仕様の構文についての検証は CASE ツールを使って行い，意味的な検証は Formal ツールで行う．意味的な検証に必要な形式的な仕様は，CASE ツールのダイアグラムに機械的な変

換を行うことと，ダイアグラムを参照しながら手作業的な記述を行うことで獲得する．

本論文では，CASE ツールに形式的仕様を適用する際の様々な課題のうち，次の事柄に注目する．

- ダイアグラムから形式的な仕様記述への変換方法
- ダイアグラムを変換した形式的な仕様記述の検証方法

これらの課題を具体的に考察するために，ダイアグラムとしては，ERD (Entity Relationship Diagram) と DFD (Data Flow Diagram)，PAD (Process Action Diagram) を用いる．これら異なる種類のダイアグラム間でデータの定義を保持するため DD (Data Dictionary) を用いる．そして，形式的な仕様を記述するために，言語 Z[4] を用いる．

CASE ツールに形式的な手法を導入することにより，次の 2 つの効能が考えられる．1 つは，様々な種類のダイアグラムや様々な抽象度のダイアグラムの間で，形式的な検証が可能となることである．これによって，従来の CASE ツールのようにダイアグラムがツールによって限定されるのではなく，開発者の好みや対象システムの種類に応じた任意のダイアグラムの選択が可能になると考える．もう 1 つは，ダイアグラムを変換することにより，形式的な仕様を効率良く構築できることである．これは，実際のシステム開発への形式的な手法の導入を促進すると考える．

以下本章では，6.2 節で，CASE ツールに形式的な仕様記述を適用する方法について，その基本概念を具体的なダイアグラムを用いて説明する．6.3 節では，ダイアグラムから Z 仕様記述への変換・記述方法と，Z の仕様の検証方法について，具体例を用いて述べる．6.4 節では，CASE ツールに形式的な仕様記述を適用した開発環境における，ソフトウエア検証の新しい視点を考察する．

6.2 CASE ツールと Formal ツールによる開発

本章では，CASE ツールに形式的手法を適用する方法について，その基本概念を具体的なダイアグラムを使い説明する．まず最初に，本研究で想定する CASE

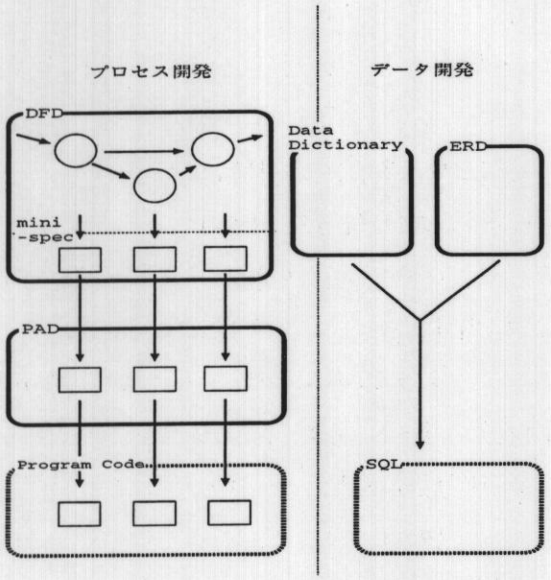


図 6.2 CASE ツールによる開発手順

ツールを使った開発手順について述べる．次に，本論文で検証する一貫性の定義を行い，検証の際の ERD と DFD，PAD 間の関係について述べる．最後に，ダイアグラムを形式化する手順について説明し，検証する方法について述べる．

6.2.1 CASE ツールによる開発

本論文では，顧客の要求を分析したり，ダイアグラムを記述したりするための CASE ツールとして，プロセス開発とデータ開発の 2 つの側面から構成される CASE ツールを想定する．図 6.2は，それぞれの開発手順を示す．このツールは，ERD や DFD といった構造化技法における代表的なダイアグラムを扱う．各ダイアグラムについては，[27] [28] [29] に詳しく述べられている．

プロセス開発においては，対象とするシステムの動的な振舞いを分析し，その結果をプロセス・ダイアグラムに記述することを目的とする．まず最初に，処理と処理の間を流れるデータについて着目して各処理の機能を明確にしてゆき，分析結果を DFD に描く．DFD を使った開発方法については，DeMarco [30] や Yourdon [31] に詳しく書かれている．そして，この分析を各プロセスが基本プロ

セスになるまで繰り返し行う．基本プロセスとは，システムの内部状態に関する制約を保持する最小の処理単位である．次に，各基本プロセスが何を行うのかを示すミニ仕様書 (mini-spec) を記述する．ミニ仕様書は，自然言語を使い簡明に記述する．最後に，ミニ仕様書を参照して PAD を記述する．PAD は，条件分岐やループといった処理構造や処理手続きを記述するものである．この，PAD を記述するということは，コンピュータシステム上で実現するために，仕様を処理手続きに詳細化することに相当する．

データ開発においては，システムの静的なデータ状態を分析し，分析結果を ERD に記述することを目的とする．ERD は，システムやプロセスにおける概念データモデルのデータ間の関係を表し，そのデータモデルから RDB(relational database) が生成される．上記のプロセス開発とデータ開発を通して定義されたデータ定義は，DD に記録される．DD は，ダイアグラムではなくデータ定義のリポジトリ（貯蔵庫）であり，データ定義を全開発を通して共通に保持するために存在する．DD に記述されたデータ定義は，データ開発を通じてその意味が吟味され統廃合されていく．

上記の 2 種類の開発が終了した後，PAD からプログラム・コードを生成し，ERD から SQL(Structured Query Language) コードを生成する．この生成作業は，CASE ツールで自動的に行う．

6.2.2 ダイアグラムの一貫性検証

本論文では，次の 2 つの観点からダイアグラムを検証する．1 つは,「PAD はミニ仕様書を詳細化したものである」ということである．これは，ミニ仕様書に書かれた前提条件のもとでの PAD の任意の振舞いが，ミニ仕様書に記述される仕様を満足することを意味する．詳細化の概念については Morgan [8] に詳しく書かれている．この検証は，ミニ仕様書と PAD の間の一貫性を保証するものである．もう 1 つは,「PAD の各プロセスは，その実行後に ERD に書かれたデータの制約を保存する」ということである．これは，システム開発者が共通認識としてもつ,「プログラムは，実行後いつも，RDB のもつデータの制約を満足しなければならない」という考えを，仕様として記述したものである．ここで，プロ

グラム・コードの仕様はPADであり，RDBの仕様はERDであり，上の検証によって，PADとERDの間の一貫性を保証するものである．本論文では，PADとERDの一貫性について，プログラムの実行によるテストではなく，仕様の段階での形式的な証明により検証することを目標とする．

ここで，上記の2つの検証が1つに統合できることを示す．前者のPADがミニ仕様書の詳細化であることの検証は，後者のPADの実行がERDの制約を保存することの検証を含むと考える．図6.3は，この概念を示す．ミニ仕様書は,「データ状態の仕様」と「処理操作の仕様」で構成される．データ状態の仕様はデータ状態の制約を記述し，その記述の中には，常に,「プロセスの実行後はデータ状態の制約を満足する」という仕様が，明示的もしくは暗黙的に存在する．ERDは，このデータ状態の記述の一部分を構成する．もし，PADがミニ仕様書を正しく詳細化したものならば，PADはその実行後に，ミニ仕様書に書かれた状態の制約を満足しなければならない．よって，PADがミニ仕様書の詳細化であることの検証は，PADの実行が，ERDの制約を保存することの検証を含む，と考えることができる．ゆえに，本論文では，ミニ仕様書とPAD間の詳細化の関係という観点で，ダイアグラムの検証を形式的な証明を用いて行う．

6.2.3 Formalツールによる開発

ここでは，Formalツールによる開発方法について具体的なダイアグラムを用いて説明する．図6.4は，その作業の流れを示す．本論文では，Formalツールに

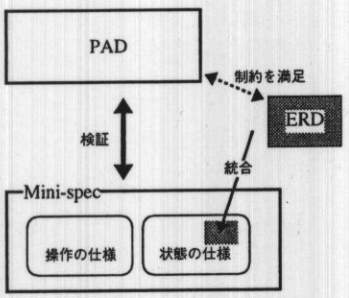


図 6.3 検証の統合

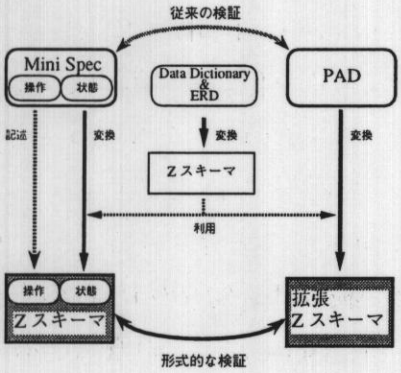


図 6.4 Formal ツールによる開発方法

よる開発方法において，次の 2 つの技術的な問題点に注目する．1 つは，ダイアグラムから Z 言語で書かれた形式的な仕様を得る方法である．もう 1 つは，PAD がミニ仕様書の詳細化であることを検証する方法である．

まず，ダイアグラムから形式的な仕様を得るために，次の 3 つの手順を踏む．

1. Z 言語で書かれたテンプレート・スキーマを用いて，DD と ERD を形式的な仕様に変換する．変換された仕様は，DD や ERD が記述しているものと意味的な内容は等しい．

2. ミニ仕様書の状態仕様を，上記で変換された DD や ERD を用いて形式的な仕様に変換する．その後，ミニ仕様書の処理操作の仕様を，変換前のミニ仕様書を参照しながら Z 言語で記述する．この記述するという作業は，曖昧な言語で書かれた仕様を明確に定義することだけでなく，ミニ仕様書では暗黙的に示されていた仕様を明示的に書き下すことを含むと考える．

3. テンプレート・スキーマを用いて PAD を形式的な仕様に変換する．Z 言語では，プログラムの制御構造を記述する表記法が用意されていないので，本手法では文献 [9] における Dijkstra のガード付言語風に拡張した表記法を使用する．

本論文では，これらの作業を通して，ダイアグラムから Z 言語へのテンプレートを用いた機械的な変換の手法を提示し，Z を用いた操作の仕様の記述についてガ

表 6.1 関連を表す記号

記号	関連
A ○< B	A は 0 もしくは多数の B と関連
A >< B	A は 1 つ以上の B と関連
A ○| B	A は 0 もしくは 1 つの B と関連
A >|| B	A は 1 つの B と関連

イドラインを提示する．

次に，ミニ仕様書と PAD の詳細化の関係を検証する方法を述べる．詳細化の概念は Morgan [8] に則り，検証の方法については Wordsworth [9] に述べてある方法を使用する．本論文では，6.3.3項において，その方法をダイアグラム間の一貫性検証に適用する方法について，具体例を用いて説明する．

6.3 Z を使った開発方法

本節では，Z を使った形式的なアプローチによる開発方法を説明する．そして，簡単な銀行システムを具体例として使い，ERD と DFD，PAD から Z の仕様を構築する方法を提示し，ミニ仕様書と PAD 間の詳細化の関係を検証する方法について述べる．

6.3.1 銀行システムの仕様

ここでは，この章で扱う銀行システムについてダイアグラムを使った仕様を示す．図 6.5は，銀行システムの ERD であり，システム内のエンティティーや関連を記述する．ERD の表記法は，Martin [29] の表記に従う．図 6.5において，エンティティーは四角形で表現し，関連 (relation) は表 6.1に示すような記号で表現する．

各エンティティーはいくつかの属性 (attribute) を持ち，その属性は CASE ツール内の DD に定義する．

図 6.5に示すように，この銀行システムは，5 つのエンティティーと 6 つの関

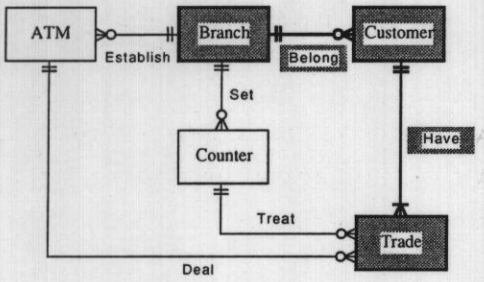


図 6.5 銀行システムの ERD

```
Branch   =  @BranchId + BranchName
Customer =  @(primary key of Branch + CustomerId) + CustomerName
Trade    =  @(primary key of Customer + TradeId)  + Balance
ATM      =  @(primary key of Branch + AtmId)      + AtmPlace
Counter  =  @(primary key of Branch + CounterId)  + CounterName
```

図 6.6 エンティティーを構成する属性

連から構成される．システムには，支店 (Branch) が存在する．各支店には顧客 (Customer) が存在し，顧客エンティティーは顧客名などの属性を持つ．各支店には現金自動預払機 (ATM) が設置してあり，受付窓口 (Counter) が存在する．顧客の取引 (Trade) は，ATM か受付窓口を通して行い，全てデータベースに記録する．各エンティティーには 1 つ以上の主キーを含む属性リストが存在する．この主キーは，各エンティティーの要素を一意に同定するものである．図 6.6は，各エンティティーを構成する属性の定義であり，キー属性は属性名の前に@をつけて区別している．例えば，図 6.6では，支店エンティティーが支店 ID(BranchId) と支店名 (BranchName) の 2 つの属性で構成され，支店 ID を主キーとして定義していることを示している．本論文では，図 6.5のうち影をつけた 3 つのエンティティーと 2 つの関連を扱い，6.3.2項で Z の仕様に変換する．

銀行システムには，利払い・預金・支払などの様々なトランザクションが発生するが，今回は特に「利払い処理」について取り扱う．図 6.7は，利払い処理についての DFD である．図において，プロセスはバブル（円）で示し，ファイルは 2 本の平行線で表す．データフローは矢印で表し，矢印の先がデータの流れる

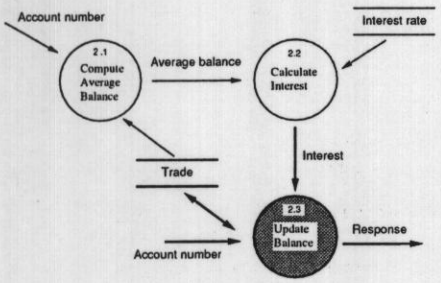


図 6.7 「利払い処理」の DFD

Account number	=	CustomerId
Average balance	=	Money
Interest	=	Money
Response	=	Message + NewBalance

図 6.8 データーフローのデータ定義

先を示す．プロセスからファイルへ向かうデータフローは，そのプロセスがファイルにデータを書き込み，ファイルの状態を変化させることを表す．ファイルからプロセスへ向かうデータフローは，そのプロセスがファイルからデータを読み出し，ファイルの状態は変化させないことを表す．プロセスとファイルの両方に向かう矢印を持つデータフローは，そのプロセスが書き込みと読み出しの両方の処理を行うことを表す．プロセスからプロセスに向かうデータフローは，システム内部でデータが移動することを表す．図 6.8は，図 6.7中のデータフローに対するデータ定義を記述する．

図 6.7の DFD には 3 つのプロセスが存在するが，本論文ではプロセス「2.3 Update Balance」に対してミニ仕様書や PAD を記述する．図 6.9はこのプロセスに対するミニ仕様書である．ミニ仕様書は,「状態の仕様」と「操作の仕様」で構成される．ミニ仕様書の上部に記述する状態の仕様は，ミニ仕様書のプロセスが関係するデータフローの入出力を定義し，プロセスが実行後に満足しなければならないデータの制約を記述する．ミニ仕様書の下部に記述する操作の仕様は，プロセス内で行うデータフローに対する操作を自然言語で簡明に記述する．

2.3 Update Balance

Input_File: Trade

Output_File: Trade

Input_Var: Interest
Account number

Output_Var: Response

If Interest >= 0, do the following processes,
else output "ng" as Message of Response.

Read the latest balance of Account number from Trade file.

Add the latest balance and Interest up to the new balance.

Update Trade file of Account number with the new balance.

(At this time, increase the statement number by one.)

Assign the new balance and "ok" to Response.

Output Response.

図 6.9 「2.3 Update Balance」のミニ仕様書

```
2.3 Update Balance
Input_File:  Trade
Output_File: Trade
Input_Var:   Interest
             Account
Output_Var:  Response
------------------------------------------------------
IF   Interest of Input_Var  >=  0  THEN
     READ  ONE RECORD  FROM  Trade of Input_File
           WITH  Account of Trade of Input_File  =  Account of Input_Var
           AND  DOWN_SEQUENCE  of  id of Trade of Input_File
     LET  Account of Trade of Output_File  =  Account of Input_Var
     LET  id of Trade of Output_File  =  id of Trade of Input_File  +  1
     LET  balance of Trade of Output_File  =  balance of Input_File
                                           +  Interest of Input_Var
     LET  newbalance of Response of  Output_Var  =  balance of Input_File
                                                 +  Interest of Input_Var
     LET  message of Response of Output_Var  =  "ok"
     WRITE  Trade of Output_File
ELSE
     LET  message of Response of Output_Var  =  "ng"
```

図 6.10 「2.3 Update Balance」の PAD

図 6.10は,「2.3 Update Balance」に対する PAD である．PAD は,「状態の記述」と「処理手続きの記述」で構成される．PAD の上部に記述する状態の記述では，ミニ仕様書と同様に，PAD のプロセスが関係するデーターフローの入出力を記述する．PAD の下部に記述する処理手続きの記述では，プロセスの処理構造や処理手続きを，第 4 世代言語のようにプログラム・コードが生成可能な言語で記述する．

6.3.2 ダイアグラムの形式化

ここでは，ダイアグラムから形式的な仕様を構築する方法について，次の 2 点に注目する．1 つは，ダイアグラムを Z の仕様に変換する際に使用するテンプレート・スキーマである．もう 1 つは，ミニ仕様書の操作の仕様を形式化する時の指針である．以下では，上記の 2 点について実際にいくつかの事例を示しながら，ダイアグラムから形式的な仕様を構築する方法について述べる．

データ・ディクショナリと ERD の変換

ここでは，DD と ERD から Z を使った形式的な仕様に変換する方法について述べる．変換された仕様は，DD や ERD が表現するものと意味的に等しくなる．DD と ERD を Z の記述に変換するために，以下のことを行う．

- 属性やエンティティー，関連に対してテンプレート・スキーマを与える．
- 上記のテンプレート・スキーマを使用して，DD や ERD から形式的な仕様を得る．

次に，属性やエンティティー，関連を変換する方法について述べる．

[データ・ディクショナリの変換]

以下は，属性に対するテンプレート・スキーマである．

$$AttrName \mathrel{\widehat=} [name : AttrType]$$
$$CompAttrName \mathrel{\widehat=} [AttrName1; AttrName2]$$

AttrName は，単一の属性に対するテンプレート・スキーマであり，1つの属性とそのデータ型を表す．*CompAttrName* は，複合属性に対するテンプレート・スキーマであり，2つの属性が含まれることを表す．

ここで，図 6.8における利子（Interest ）と応答（Response）に対する変換の適用例を示す．利子は単一の属性でデータ型は Money なので，テンプレート・スキーマ *AttrName* を使い，以下のように変換する．

$$Interest \mathrel{\widehat=} [interest : Money]$$

応答は複合属性なので，テンプレート・スキーマ *CompAttrName* を使い，以下のように変換する．

$$Response \mathrel{\widehat=} [Message; NewBalance]$$

Message と *NewBalance* に対する定義は付録で与える．

[エンティティーの変換]

以下は，エンティティーに対するテンプレート・スキーマである．

$$\begin{array}{|l}
\hline
Record[PKeyAttr, Attr] \\
\hline
p : PKeyAttr \\
a : Attr \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
Entity[PKeyAttr, Attr] \\
\hline
entity : \mathbb{P}\, Record[PKeyAttr, Attr] \\
\hline
\forall r1, r2 : entity \bullet r1.p = r2.p \Rightarrow r1 = r2 \\
\hline
\end{array}$$

Record は，エンティティーの要素の構造を表すテンプレート・スキーマであり，エンティティーが主キー属性とその他の属性からなることを表す．主キー属性の構造は *PKeyAttr* で定義し，その他の属性は *Attr* で定義する．

Entity は，エンティティー全体の構造を示すテンプレート・スキーマであり，*Record* の集合として表す．*PKeyAttr* と *Attr* は，上記同様 *Record* の構成要素である．主キーの特性は *Entity* スキーマの述語部に記述する．

図 6.5における Branch エンティティーをスキーマに変換する適用例を示す．Branch エンティティーは，主キーである BranchId と，支店名を表す BranchName の 2 つの属性を持つ．それら属性は，図 6.6にある DD で定義されている．以下のスキーマでは，*BraP* と *BraA* を,「Branch の主キー」 ならびに「Branch の属性」として使用する．よって，*BraP* は BranchId を表し，*BraA* は BranchName を表すことになる．

$$BraP \mathrel{\widehat=} [id : \mathbb{N}]$$
$$BraA \mathrel{\widehat=} [name : STRING]$$
$$BraR \mathrel{\widehat=} Record[BraP, BraA]$$
$$BraE \mathrel{\widehat=} Entity[BraP, BraA][braE/entity]$$

BraP は，主キーである id の型が自然数であることを示し，*BraA* は，その他の属性である name の型が *STRING* であることを示す．ここで，型 *STRING* は所与の集合 (Given Set) として定義されている．Branch エンティティー *BraE* は，*Entity* スキーマの *entity* を *braE* と名称変更したスキーマであることを示す．Customer エンティティーや Trade エンティティーについても同様に変換でき，それらについては付録で示す．

[関連の変換]

本論文では，表 6.1にある記号で ERD におけるエンティティー間の関連を表す．表 6.1に示す記号は 4 種類であるから，関連の組合せは 16 通りになる．これら関連の組合せは,「多対多」と「多対一（一対多)」,「一対一」の 3 つに分類できる．多対一と一対多は，対称性を考慮して同様なものとして取り扱う．対称性を考慮すると，関連の組合せは 10 通りになる．本論文では，多対多関連のテンプレート・スキーマと多対一関連の変換例を示す．

以下に，多対多関連についてのテンプレート・スキーマを示す．多対多関連は，もっとも基本的な関連である．

$M2M[E1, E2]$

$$rel : \mathbb{P}(E1 \times E2)$$

$E1$ と $E2$ はエンティティーを表し，関連は，それらエンティティーの直積の集合としてモデル化する．その他の関連に対するテンプレート・スキーマは付録で提示する．

ここでは，図 6.5で示す Trade エンティティー と Customer エンティティー，Branch エンティティー間の関連について，変換の適用例を示す．

$Have$

$$EM2EOne[TraR, CusR][relTCu/rel]$$

$$TraE; CusE$$

$$\mathrm{dom}(relTCu) = traE \land \mathrm{ran}(relTCu) = cusE$$

$$\forall t : traE \bullet t.p.cusp \ \mathsf{FKey} \ cusE$$

Have スキーマは，Trade エンティティーと Customer エンティティー間の関連を表す．スキーマの宣言部では，この関連がテンプレート・スキーマ *M2EOne* の *rel* を *relTCu* と名称変換したものであり，Trade エンティティー (*TraE*) と Customer エンティティー (*CusE*) 間の関連であることを宣言する．スキーマの述語部では，エンティティーや関連についての制約を述べる．FKey は外部キーについての制約を示し，その定義は付録に記述する．Customer エンティティーと Branch エンティティー間の関連 Belong についても，同様にして変換可能である．

以上のように，ここでは DD や ERD における属性やエンティティー，関連に対して一般的な変換方法を示した．

ミニ仕様書の変換と記述

ここでは，ミニ仕様書から Z の仕様を得るための変換と記述の方法を示す．この作業を始める前に，システム内の全てのデータは DD 内に定義してあり，DD

と ERD 内の定義は全て Z の記述に変換しているものとする．ミニ仕様書から Z の仕様を得るための変換と記述を行うために，以下のことを行う．

- ミニ仕様書に対してテンプレート・スキーマを与える．
- 上記のテンプレート・スキーマを使用して，ミニ仕様書の状態の仕様を Z の記述に変換する．
- 上記の変換したスキーマに，ミニ仕様書の操作の仕様を言語 Z を用いて書き加える．この記述という作業は，処理操作の前提条件や，入力と出力の関係，状態データの変化などを定義することに相当すると考える．そして，この作業を機械的に行うことは不可能であると考える．

以下はミニ仕様書に対するテンプレート・スキーマである．

$$
\begin{aligned}
&ProcessName \mathrel{\widehat=} \\
&\quad [\Xi UnaffectDS;\ \Delta AffectDS;\ InData?; \\
&\qquad\qquad\qquad\qquad OutData!]
\end{aligned}
$$

UnaffectDS は，プロセスで状態変化をしないファイルを表す．状態変化しないファイルは，ミニ仕様書の中では Input_File でのみ定義される．*AffectDS* は，プロセス内で状態変化するファイルを表し，そのファイルはミニ仕様書内の Output_File で定義される．Input_File と Output_File の両方で定義されているファイルは，*AffectDS* に変換する．ミニ仕様書内の入力データは *InData*? に変換し，出力データは *OutData*! に変換する．

ここで，図 6.9の「2.3 Update Balance」に対する変換の適用例を示す．

$$
\begin{aligned}
&UpdateBalanceTrans \mathrel{\widehat=} \\
&\quad [\Delta TraE;\ Interest?;\ AccountNumber?; \\
&\qquad\qquad\qquad\qquad Response!]
\end{aligned}
$$

ファイル Trade は，Input_File と Output_File の両方で定義されているので，*TraE* に対して記号 Δ が使用される．*Interest*? と *AccountNumber*?，*Response*! は，入出力データを表すスキーマ である．

次に，入出力データに対する操作や関係を，上記で求めたスキーマに対して記述する．図 6.9 のミニ仕様書は条件分岐を含んでいるので，以下では仕様を条件値により場合分けして記述する．ミニ仕様書の正常処理を *UpBalOk* に記述し，エラー処理は *UpBalNg* に記述する．*UpBalNg* の記述は，付録で示す．

UpBalOk

$\Delta TraE$

$Interest?$

$AccountNumber?$

$Response!$

$interest? \geq 0$

$traE' = traE \cup$

$\{(\mu\, x : TraR \mid x.p.cusp = cusp? \wedge$

$x.p.id = NextId(traE, cusp?) \wedge$

$x.a.balance = recentB(traE, cusp?) + interest?)\}$

$message! = ok$

$newbalance! = recentB(traE, cusp?) + interest?$

UpBalOk の述語部において，関数 *recentB* は，対象顧客の最新の残高を戻り値とする関数であり，関数 *NextId* は，顧客の最新取引通番の次番号を戻り値とする関数である．これら関数の定義は付録で示し，スキーマの述語部では隠蔽してある．

$$UpdateBalance \mathrel{\widehat{=}} UpBalOk \vee UpBalNg$$

UpdateBalance はミニ仕様書全体の仕様である．

PAD の変換

ここでは，PAD の仕様から拡張 Z の記述への変換の方法を示す．変換した記述は PAD が表現するものと意味的には等しくなる．ここで用いる拡張した Z

言語は，Z 言語に Dijkstra のガード付言語を付加したものであり，その表記は，Wordsworth[9] の表記に従う．Wordsworth の表記法では，入力変数は変数名に ? を付けずに変数名のままで表し，出力変数は！ではなくダッシュ(') を付けて表す．

PAD を拡張 Z の記述に変換するために以下のことを行う．

- PAD の制御構造に対してテンプレートを与える．
- PAD のコマンド（命令）にテンプレート・スキーマを与える．
- 上記のテンプレートを使用して，PAD を拡張 Z の記述に変換する．この時，ミニ仕様書に与えたテンプレート・スキーマも使用する．なぜなら，PAD の状態の記述はミニ仕様書の状態の仕様と表記が同じだからである．

これから PAD の変換方法を示す．

まず最初に，PAD の制御構造に対してテンプレートを与える．拡張 Z 言語は 3 種類の制御構造で構成される．それは,「逐次合成」と「条件分岐 (if-then-else)」,「ループ (do-while)」である．PAD もこの 3 つの構造で構成されるので，PAD の制御構造を変換するのに拡張 Z 言語の表記をそのまま利用する．以下は，PAD の制御構造に対するテンプレートである．

$$Sequence : A_1; A_2$$
$$Alternation : \textbf{if}\ B \rightarrow A_1 \ \Box\ \neg B \rightarrow A_2\ \textbf{fi}$$
$$Iteration : \textbf{do}\ B \rightarrow A\ \textbf{od}$$

A と A_1 ， A_2 はスキーマやコマンドを表し，B はガードを表す．

次に，PAD のコマンドに対してテンプレート・スキーマを与える．PAD に出現するコマンドは様々な種類のものが存在するが，本論文では，RDB を操作する種類のコマンドについて述べる．読み込み（read）や書き出し（write）は RDB を操作する典型的なコマンドであり，以下では，それら読み書きのコマンドに対するテンプレート・スキーマを示す．ただし，読み書きの対象となるエンティティーの例として *SmplE* を用いている．*SmplE* の定義と書き出しのテンプレート・スキーマは付録で示す．

SmplRead

$\Xi SmplE$

$key : KEY$

$OutRec' : SmplR$

$OutRec' \in smplE$

$OutRec' \in \{(\mu\, x : SmplR \mid x.p.key = key)\}$

SmplRead は読み込みコマンドに対するスキーマである．ただし，このスキーマをもとにデータベース読み込みのスキーマを作成することができるので，*SmplRead* はテンプレート・スキーマの役割を果たしていると見なして，ここに掲げている．

key は，目的とする要素を示す主キーであり，*OutRec* は出力要素である．ここで，RDB を操作するコマンド全てに対してテンプレート・スキーマを用意することが可能であると考える．なぜなら，Z 言語と RDB の両方が集合論を基礎としており，RDB を操作するコマンドも集合論の表記で記述可能であるからだ [18] [32].

最後に，図 6.10 の PAD「2.3 Update Balance」に対する変換例を示す．

$UpdateBalancePAD \mathrel{\widehat=}$

if $interest \geq 0 \rightarrow UpBalOkPrg$

$\Box$ $interest < 0 \rightarrow UpBalNgPrg$

fi

UpdateBalancePAD をテンプレート・スキーマを使って変換した．*UpBalOkPrg* は PAD の THEN 部を変換したスキーマであり，*UpBalNgPrg* は PAD の ELSE 部を変換したスキーマである．それらについての詳細は付録で述べる．

さらに,「2.3 Update Balance」の正常処理が,「読み込み (read)」と「代入 (assign)」,「書き出し (write)」の 3 つの部分処理から構成されるとみなす．PAD の THEN 部を以下のように逐次合成された処理として詳細化する．

$UpBalOkPrg \sqsubseteq Read; Assign; Write$

記号 '$\sqsubseteq$' は詳細化の関係を表現し，この詳細化の関係については 6.3.3項で説明する．

Read

$$
\begin{array}{l}
\Xi TraE \\
AccountNumber;\ Interest \\
Response' \\
ReadRec' : TraR \\
Interest' \\
\hline
ReadRec' \in traE \\
ReadRec' \in \{(\mu\, x : TraR \mid \forall\, y : TraR \bullet \\
\qquad\qquad (y \in traE \land \\
\qquad\qquad x.p.cusp = cusp \land \\
\qquad\qquad y.p.cusp = cusp) \longrightarrow \\
\qquad\qquad x.p.id \geq y.p.id)\} \\
Interest' = Interest
\end{array}
$$

スキーマ *Read* は，上記の *Read* 処理をテンプレート・スキーマを使用してZの表記に変換したものである．*ReadRec* と *NewRec* はローカル変数であり，3つの処理間で値を伝達するものである．*Interest* は次の処理 *Assign* で参照されているので，*Read* スキーマに対しては，変数 *Interest'* と，「$Interest = Interest'$」という記述を加えた．*Read* 以外の操作についても同様にして変換でき，それらについては付録で示す．

6.3.3 ミニ仕様書と PAD の詳細化の検証

ここでは，ミニ仕様書と PAD の詳細化について検証する方法を示す．PAD がミニ仕様書の正しい詳細化であるとは，ミニ仕様書に書かれた前提条件のもとでの PAD の任意の振舞いが，ミニ仕様書に記述される仕様を満足する時である．この詳細化の概念は，Morgan[8] に詳しく述べられている．Z の仕様と拡張 Z の記述について詳細化の関係を検証する方法は，Wordsworth[9] の方法に従う．

文献 [9] では，プログラムが仕様に関して正しいとは，仕様が定める事前条件の下で，そのプログラムの全ての可能な振舞いが仕様によって許容されることを言う．このような関係を詳細化の関係と呼ぶ．詳細化の関係を保証する条件として以下の 2 つの検証条件がある．

- safety 条件

  $$\mathrm{pre}\ \mathit{Spec} \vdash \mathrm{pre}\ \mathit{Ref}$$

  Spec に適用可能などんな環境も *Ref* で適用可能でなければならない．

- liveness 条件

  $$(\mathrm{pre}\ \mathit{Spec}) \wedge \mathit{Ref} \vdash \mathit{Spec}$$

  Spec に適用可能な環境において，*Ref* の振舞いは *Spec* によって許容されなければならない．

上記の 2 つの検証条件を満足する関係を以下のように示す．

$$\mathit{Spec} \sqsubseteq \mathit{Ref}$$

ここで，$\sqsubseteq$ は詳細化の関係を表す記号である．

検証を始める前に，仕様中の入力変数名から記号？を取り除く作業を行う．そして，出力変数名から記号！を取り除き，代りにダッシュ(') を付ける．これら作業は，Wordsworth の方法で提唱されており，この作業の詳細は付録で述べる．この章では，*UpBalOk* と *UpBalNg* をこれらの作業を施したスキーマとして扱う．

プロセス「2.3 Update Balance」を表現するミニ仕様書と PAD について詳細化の関係を検証するために，以下の 3 つの手順を踏む．

1. *UpdateBalance* を *UpBalOkPrg*，*UpBalNgPrg* の条件分岐構造に詳細化したことを検証する．

   $$\mathit{UpdateBalance} \sqsubseteq \mathit{UpdateBalancePAD}$$

2. *UpBalOkPrg* が *UpBalOkIntr* に詳細化されたことを検証する.

$$UpBalOkPrg \sqsubseteq UpBalOkIntr$$

UpBalOkIntr は, *UpBalOkPrg* にローカル変数を導入したスキーマであり, その定義は付録で示す.

3. *UpBalOkIntr* が, *Read* と *Assign* , *Write* 処理の逐次合成に詳細化されていることの検証を行う.

$$UpBalOkIntr \sqsubseteq Read; Assign; Write$$

本論文では, 紙面の都合上, 3番めの検証とその検証条件について述べる.

以下に, *UpBalOkIntr* が, *Read*, *Assign* , *Write* 処理の逐次合成に詳細化されたことを検証するための検証条件を示す.

- safety 条件

$$\text{pre } UpBalOkIntr \vdash \text{pre } Read$$
$$\text{pre } UpBalOkIntr \wedge Read \vdash (\text{pre } Assign)'$$
$$\text{pre } UpBalOkIntr \wedge Read \wedge Assign' \vdash (\text{pre } Write)''$$

- liveness 条件

$$\text{pre } UpBalOkIntr \wedge Read \wedge Assign' \wedge Write'' \vdash UpBalOkIntr[\_'''/\_']$$

本論文では, 上の liveness 条件 を示す検証条件について証明を行う. まず最初に, 上記の検証条件を証明するために, 各スキーマの述語部を展開する. 以下は, 検証条件の左側を展開したものである.

$$ReadRec' \in traE$$
$$ReadRec' \in \{(\mu x : TraR \mid \forall y : TraR \bullet$$
$$(y \in traE$$
$$x.p.cusp = cusp \wedge$$
$$y.p.cusp = cusp) \longrightarrow$$
$$x.p.id \geq y.p.id)\}$$
$$interest' = interest$$

$$
\begin{aligned}
&traE' = traE \\
&NewRec'' \in \{(\mu\, x : TraR \mid \\
&\quad x.p.cusp = ReadRec'.p.cusp \wedge \\
&\quad x.p.id = ReadRec'.p.id + 1 \wedge \\
&\quad x.a.balance = ReadRec'.a.balance + interest')\} \\
&traE'' = traE' \\
&traE''' = traE'' \cup \{NewRec''\} \\
&newbalance''' = NewRec''.a.balance \\
&message''' = ok
\end{aligned}
$$

検証条件の右側を展開すると，以下のようになる．

$$
\begin{aligned}
&traE''' = traE \cup \\
&\quad \{(\mu\, x : TraR \mid x.p.cusp = cusp \wedge \\
&\quad x.p.id = NextId(traE, cusp) \wedge \\
&\quad x.a.balance = recentB(traE, cusp) + interest)\} \\
&newbalance''' = recentB(traE, cusp) + interest \\
&message''' = ok
\end{aligned}
$$

あとは，述語論理を基にした導出によって，証明を行うだけであるが，ここでは省略する．

6.4 評価

本論文では，仕様を構文と意味の２つの観点から検証するために，CASE ツールに形式的手法を適用した新しい開発環境を提案し，その適用の際に問題となる，ダイアグラムを形式的な仕様記述へ変換する方法と，変換した仕様記述の検証方法について，具体的なダイアグラムを用いて述べた．これらの変換と検証の方法は，形式的な手法を CASE ツールに適用する際の一番核になる技術であると考える．

ここでは，新しい開発環境において可能になる新しい視点からのダイアグラム

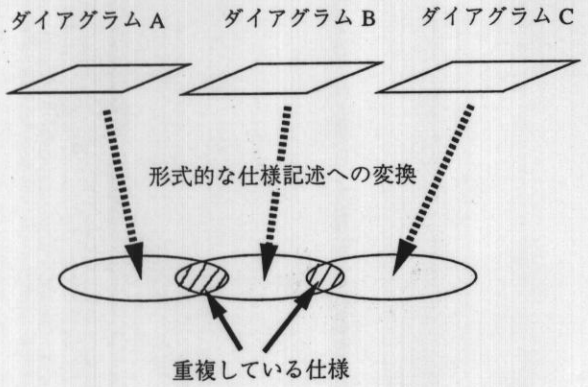


図 6.11 異なる種類のダイアグラム

の検証方法について述べ，仕様記述の検証におけるツールによる支援について考察する．

6.4.1 ダイアグラムの新しい検証方法

形式的手法を CASE ツールに適用した新しい開発環境では，開発者は，複数のダイアグラムについて一貫性の検証を行うことが可能となる．ここでは，検証における２つの主要な視点について論じる．

１つは，種類の異なるダイアグラムについて一貫性を検証することができることであり，図 6.11は，この概念を表す．通常，開発者は，ユーザーの要求を様々な角度から分析するために，色々な種類のダイアグラムを使用する．もし，異なるダイアグラムの間で同じ仕様についての記述があれば，その仕様についてはダイアグラム間で一貫性を保つ必要がある．例えば，6.2節で述べたように，ミニ仕様書と ERD は両方ともデータの状態制約をダイアグラムの中で記述する．新しい環境では，ダイアグラムを形式的な仕様記述に変換することにより，異なる種類のダイアグラムに関する正当性の検証を行うことができる．このことは，ユーザーの要求を分析する時や要求の構造を定義する際に，開発者の好みに応じて任意のダイアグラムを選択することを可能にする．

もう１つは，異なる抽象度のダイアグラム間で一貫性の検証ができることであ

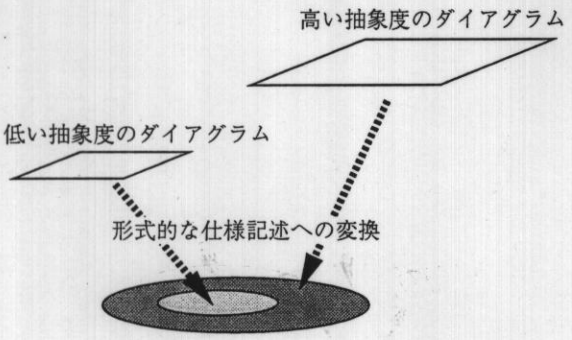


図 6.12 異なる抽象度のダイアグラム

る．図6.12は，この概念を表す．システム開発者は，ユーザーの要求を分析したり要求の構造を定義する際に，高い抽象度から低い抽象度へと抽象度に沿って作業を行う．この時に，抽象度の低いダイアグラムは，抽象度の高いダイアグラムの制約を満たさなければならない，と考える．これは，異なる抽象度のダイアグラムについて一貫性の検証を行うことを必要とする．新しい環境では，変換した形式的な仕様記述を使い，そのような一貫性の検証を行うことができる．

6.4.2 ツールによる検証の支援

仕様記述の検証を行うには構文と意味の検証を行う必要があり，検証を正しく効率良く行うにはツールによる検証の支援が重要となる．

Z 言語では，構文チェックおよび型チェックを行うツールとして fuzz 等の多数のツールが存在する．このようなツールを用いて仕様記述に対する構文的な過ちを検出し，構文的な正しさを保証することは，仕様記述の作業効率を向上させるとともに，仕様記述に対する信頼性を高める．

仕様記述の意味を検証するために，

- 論理式の書き換え
- 事前条件の導出
- 検証条件の導出

を支援するツールの開発が望まれる．

Z 言語では，仕様記述を集合と一階述語の論理式を用いて表現するが，一つの仕様を表現するための記述は一通りではなく複数存在する．最適な仕様記述は，仕様記述者の試行錯誤の結果得られるものであり，その過程では論理式の書き換えが頻繁に行われる．また，着目する仕様記述の目的に応じて，違った観点から仕様記述の表現を書き換えた方が有効な場合がある．仕様記述の過程では論理式を書き換える作業は重要であり，正しく効率良く論理式の書き換えを支援するツールが必要である．

また，Z 言語では，操作の仕様記述において，その操作を適用する場合の前提条件，つまり事前条件が明確に表現されず，暗黙に表現されている場合がある．事前条件は操作の仕様記述の適用可能性を判断するための重要な事項であり，Z 言語の仕様には，事前条件の定義は示されてはいるものの，その導出は容易であるとは限らない．事前条件を正しく効率良く導出するためのツールの開発が望まれる．

仕様記述を段階的に詳細化する過程では，6.3.3項で述べたような詳細化の検証を行う必要がある．詳細化の検証を支援するために，検証条件の導出およびその証明を支援するツールの開発が必要である．検証条件の定義は文献 [9] に示されており，その導出は可能であるが，証明の自動化は困難であると考えられる．証明の支援については，本項で述べた論理式の書き換えによるプルーフチェッカやプルーバによる証明支援ツールの作成が考えられ，今後の研究課題の一つである．

6.4.3 今後の研究課題

今後の研究課題として，同一のダイアグラムで異なる抽象度の仕様について一貫性の検証を行う予定である．例えば，DFD (Data Flow Diagram) は構造化技法の中でも最もよく使用されるダイアグラムであり，高い抽象度から低い抽象度までユーザーの要求を分析するのに使用される．DFD では，高い抽象度についてはコンテキスト・ダイアグラムを，低い抽象度ではミニ仕様書を描く．仕様の意味的な観点からみると，高い抽象度のダイアグラムは，低い抽象度のダイアグラムを包含していると考える．形式的手法を用いて，この意味的な包含関係につ

いての検証を，今後行う予定である．

7. 開発支援環境

本章では，仕様記述の過程を分析し，変換を基にした仕様記述のための支援技術，および開発支援環境について議論する．

7.1 形式的手法による従来の開発手法との比較

Zは現存している形式仕様記述言語の中で最も利用されている言語の一つであり，Zを用いた様々な事例報告は文献[14]において多数紹介されている．Zを用いたソフトウェア開発手法として，抽象度の高い仕様記述からプログラムコードと同等のレベルまで，段階的詳細化によるトップダウンの開発手法が提案されている[9][33]．また，表記はZを用い，仕様記述のレベル間の保証はRefinement Calculus[8]を用いた統合的な手法[10][11]も提案されている．これらの手法は，最も抽象度の高い記述から，データ詳細化およびアルゴリズム詳細化といった技術をもとに，系統的にインプリメントに近づける際に有効な手法である．しかし，仕様記述のもとになる最も抽象度の高い記述をどのようにして得たかについては，そこには述べられていない．仕様記述の過程では，対象システムをどのようにモデル化するかが最も重要な作業であり，このモデル化の作業をいかにして支援するかがソフトウェア開発における重要な課題である．

本研究では，対象システムの最適なモデルは仕様記述の過程において試行錯誤の結果確定することに着目し，仕様記述の過程を変換と捉えその過程を定式化することにより，仕様記述におけるモデル化作業の支援を図った．システムの最適なモデルは最初から得られるのではなく，システムの記述が進むにつれて徐々にシステムに対する理解が形成され，より良いモデルを用いて仕様記述が書き換えられる．本研究の基本的なアイデアは，この書き換えの過程を解明し変換手法として提示することにより，モデルの変更に伴う記述の変換を系統的に行う手段を与えることである．仕様記述段階での最適なモデルを系統的に模索する方法の提示は，従来，仕様記述者がアドホックに行っていたモデル化の模索を改善する意義があると考えられる．

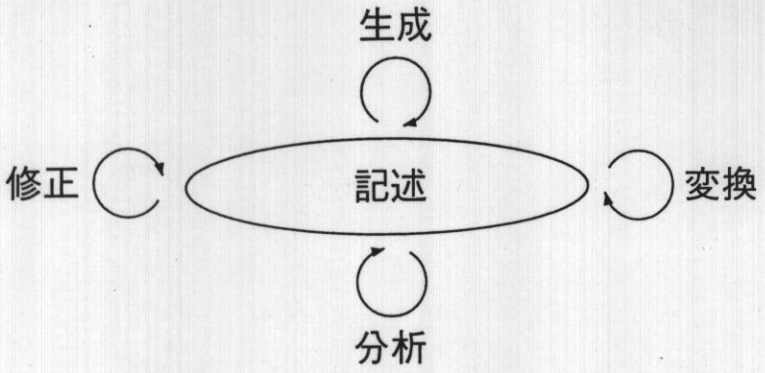


図 7.1 仕様記述の過程

7.2 仕様記述支援技術

ソフトウェア開発の工程は，(1) 要求定義，(2) 仕様記述，(3) 設計，(4) プログラミング，の 4 つの工程に大きく分けて考えることができる．まず，対象となるシステムの分析を行いシステムに対する要求を定義する．次に要求を満足するためのシステムの仕様を記述する．システムの仕様に従って設計を行った後プログラミングを行う．

ここでは，仕様記述の工程に着目して仕様記述の過程について考えてみる．第 1 章においては，図 1.2に示したように，仕様記述における思考パターンを理解，記述，分析の繰り返しであると捉えた．記述の種類には，生成，修正，変換の 3 種類が考えられる．図 7.1は記述を中心にした仕様記述の過程を図示している．以降，生成，修正の観点から求められる支援記述について考察する．

7.2.1 記述の生成

記述の生成は新しく仕様を記述することである．一番最初の記述はもちろん記述の生成となる．新しい仕様記述を考える場合，まず正常処理の機能を記述した後で，エラーの処理を記述することが一般的に行われる．このようなエラーの記述を追加する過程も記述の生成と考える．

記述の生成を支援するための技術について考える．

図式表現と形式的な表現の間の変換 第7章では構造化ダイアグラムから形式的な表現への変換について議論した．仕様記述の初期の段階では，直観的な可読性が高い図式表現を用いることは有効な手段である．しかし，図式表現だけでは仕様を表現するには記述能力が十分ではない．図式表現から形式的な表現への変換技術は図式表現の情報を最大限に利用するために有効な手段であると考えられる．

事前条件の抽出 エラーの記述を行うときは，エラーの場合分けを検討することが最も重要な問題である．エラーの条件を検討する時には，事前条件が明示されていなければならない．Z仕様記述では事前条件が陽に示されていないので，操作の仕様記述の事前条件を抽出する技術が必要である．

エラー条件の抽出 エラー処理を検討する時には，エラーの条件を厳密に把握する必要がある．機能が大きく複雑になると，エラーの条件を正しく理解することは困難な作業である．そこで，正常の処理からエラー条件を抽出する技術の開発が望まれる．

エラー条件の分析 一般的にエラーの処理は場合分けを考慮する必要がある．エラー処理の場合分けがどのようになっているかを把握することは困難である．分析技術として，(1) エラー処理の網羅性の検証技術，および (2) エラー処理の重複性の検出技術，の開発が必要である．

7.2.2 記述の修正

仕様記述の過程では，記述の間違いや仕様の変更にともなう，記述の修正が発生する．記述の修正は意味の変更を含むので，機械的に修正を行うことができない．記述の修正のための支援技術としては，変更にともなう波及効果の分析と，変更後のシステムの一貫性の検証が挙げられる．

波及効果の分析 仕様記述の変更を行う場合は，その波及効果を分析することが最も重要である．波及効果を分析するために，システムの内部状態のデータと操作仕様記述の間の依存関係を明示する．依存関係は，操作の仕様記

述における論理式のレベルではなく，内部状態のデータと操作の間の関係で捉える．したがって，あるシステム内部状態のデータとそのデータに依存している操作との対応関係を考える．Z 仕様記述では，システムの内部状態と操作の関係は陽に示されているので，内部状態のデータと操作の間の依存関係の解析は可能であると考えられる．

一貫性の検証 仕様記述を変更したあとで，システムの一貫性が保たれているかは重要な問題である．しかし，システムの一貫性が保たれているかを検証するには，一貫性とは何かが明示されていなければならない．また，システムの一貫性の基準とは何であろうか？ここでは，システムの一貫性とはシステム内部状態の制約，つまりシステム不変条件として考える．システムの一貫性の検証とは，操作の仕様記述がシステム不変条件を満たしているかを検証することである．システムの一貫性の検証を効率良く行うために，操作の仕様記述からシステム状態不変条件を抽出する技術が有効であろう．

7.3 システム仕様記述支援環境

本節では，システム仕様記述を支援するための開発環境について考える．これまで，変換技術をシステム仕様記述を支援するための技術について議論してきた．システム仕様記述の中心は記述であることは，これまでにも論じてきた．そこで，記述を中心としたシステム仕様記述の支援環境を考える．図 7.2はシステム仕様記述支援環境の構成を表している．支援環境は，

- 仕様記述リポジトリ
- ダイアグラムエディタ
- 形式仕様記述エディタ
- 変換エンジン

から構成されている．

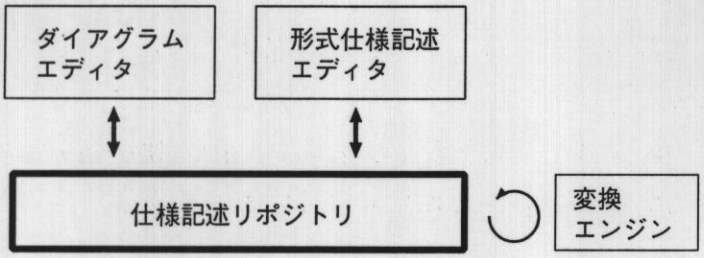


図 7.2 システム仕様記述支援環境

仕様記述リポジトリ 形式的な仕様記述を保存するためのリポジトリを用意する．保存する形式は言語に依存しない一般的な表現が望ましいが，そのような一般的な表記法は現存しない．既存の形式言語の中では，RSL[6] が最も記述能力が高い言語の一つであると考えられる．RSL は形式的な表現の一般的な記述として期待できる．RSL と Z の間の関係，もしくは RSL と他の形式言語の関係を明示することができれば，RSL リポジトリとして編集は他の言語を使用することができる．

ダイアグラムエディタ ダイアグラムに形式的な意味づけを与え，ダイアグラムと形式的な表現の対応を明確にすることにより，ダイアグラムの表現を形式的な仕様記述レベルで議論することを可能にする．システム仕様記述の段階では，システムの内部状態として必要なデータと，データ間の関連を記述することが重要である．データとデータの関連を表現するには，基本的には E-R ダイアグラムで十分であると考えられる．しかし，データ間の制約を表現するためには，E-R ダイアグラムでは不十分である．詳細な制約は形式的な表現で記述すれば良いが，形式的な記述を扱いやすくするために，ダイアグラムのレベルで簡略化した表記法の開発が望まれる．

形式仕様記述エディタ 形式仕様記述エディタは，仕様記述リポジトリとのインターフェースを提供する．仕様記述の変換は，仕様記述リポジトリの表現を基に行う．仕様記述リポジトリの形式との対応関係を明確にすることにより，仕様記述リポジトリとエディタの間の記述の変換を可能にする．

変換エンジン 本論文で提案した，(1) システムの内部状態のモデルの変換，(2) 仕様記述の構造の変換，(3) システム状態不変条件の記述の変換，の機能を実現する．ところで，論理式の表現には，同じ制約の記述でも様々な表現方法がある．形式的な仕様記述を作成するときには，論理式の書き換えが頻繁に行われる．そこで，変換エンジンの機能として，論理式の同値関係の変換を支援する機能が望まれる．また，論理式に特有の表現方法をライブラリとして提供する機能があれば，仕様記述の助けとなる．

8. おわりに

本章では，本研究で得られた成果をまとめ，今後の課題について述べる．

8.1 本研究で得られた成果

本論文では，システム仕様記述の過程を分析し，システム仕様記述を支援するための変換技術について議論した．ソフトウェア開発における本質的な課題は，

- 信頼性の高いシステム仕様記述をいかにして効率良く構築するか，

ということである．

本研究の基本的なアイデアは，

- システムの最適なモデルと仕様記述は試行錯誤の結果得られることに着目し，仕様記述の試行錯誤の過程を変換と捉えその過程を定式化する

ことである．システムの仕様記述は記述と分析を繰り返して徐々に形成されるものである．したがって，仕様記述の書き換えは，仕様記述の工程において本質的な作業である．

実際の開発現場では，仕様記述の構築は，仕様記述者の経験と勘に頼って行われているのが現状である．仕様記述の書き換えを行うときは，より良い記述へと必ず何らかの意図があるはずである．この書き換えの作業を定式化することができれば，仕様記述を構築するときの明確な指針となり得る．

書き換え作業の明確化は，書き換えの作業が

- 構文レベル，つまり記号処理として扱うことができるか，
- 意味を考慮する必要性があるか，

を明示する．記号処理として扱う変換は機械による支援が期待できる．書き換えが記号処理のレベルで書き換えを行っているのか，意味の書き換えを行っているのかを明確にすることにより，書き換えを行うときに何を考えればいいのかを明確にする．

記号処理として書き換えができる場合は，変更によるエラーの混入の可能性は低いので，より良い記述へと仕様記述を積極的に変更する効果がある．また，仕様を分析する場合に目的に応じた最適なモデルで仕様を検討することにより，仕様の分析支援を図ることができる．

変換技術の具体的な成果を以下に述べる．

システム内部状態のモデルの変換

システム内部状態のモデルとして最も基本的なモデルである2項関係について着目した．本論文では，Zにおける2項関係モデルの中で最も特徴的なモデルである全域関数，部分関数および関係の間の対応関係を明確にし，それぞれのモデル間の仕様記述の変換手法について提示した．

仕様記述の構造の変換

仕様記述の構造として階層的な機能分割について着目した．システムが大規模で複雑になれば，機能を階層的に分割することは有効な手段である．本論文では，Zにおけるスキーマ合成の表現を用いて，仕様記述の機能分割と階層化を行う変換手法について提示した．

システム状態不変条件の記述の変換

システムの内部状態の性質に影響を及ぼす操作の仕様記述について着目した．操作の仕様記述に操作的に表現されているシステムの内部状態に対する制約を，システム状態不変条件として変換する手法を提示した．

図式表現と形式的な表現の間の変換

直観的で人間になじみやすい図式表現と形式的な表現の間の関係について着目した．CASE(Computer-Aided Software Engineering) ツールで一般的に用いられている構造化ダイアグラムで表現された記述の分析を行うために，図式表現から形式的な表現への変換手法を提示した．図式表現を形式的な表現に変換することにより，ダイアグラム間の一貫性の検証を可能にする．

8.2 今後の課題

本研究において今後検討すべき課題は次の通りである．

変換技術の検討

仕様記述の信頼性を向上するためには，仕様記述の一貫性を示すための明確な記述が必要である．一貫性を検証するための要因として

- 操作の仕様記述の網羅性と重複性

があげられる．

操作の仕様記述の網羅性とは，操作の仕様記述がすべての場合を網羅していることを言う．また，操作の仕様記述の重複性とは，操作の仕様記述の間で事前条件に重なりがあることを言う．操作の仕様記述の網羅性と重複性が仕様記述者の意図していることかどうかの妥当性を検証することが重要である．操作の仕様記述の網羅性と重複性を検証するために以下の変換技術の開発が望まれる．

事前条件の変換 (抽出) 操作の仕様記述の網羅性と重複性を検証するためには，事前条件の抽出が重要である．Z 仕様記述では事前条件が陽に示されていないので，操作の仕様記述の事前条件を抽出する技術が必要である．

エラー条件の変換 (抽出) 操作の仕様記述の網羅性を検討する時には，エラーの条件を厳密に把握する必要がある．通常，エラー処理の検討は正常処理の記述の後に行われる．エラー処理の条件を厳密に把握するために，正常の処理からエラー条件を抽出する技術の開発が望まれる．

システム仕様記述支援環境の開発

本論文で提案した変換技術を実際に有効に利用するためには，支援環境の開発が不可欠である．変換の過程では論理式の書き換えが頻繁に行われる．この論理式の書き換えの効率を向上するために機械による支援が求められる．

仕様記述リポジトリ 形式的な仕様記述を保存するためのリポジトリの形式は，言語に依存しない一般的な表現が望ましい．既存の形式言語の中では，RSL[6]が最も記述能力が高い言語の一つであると考えられるので，RSL は形式的な表現の一般的な記述として期待できる．変換技術をより一般的な形で提供することにより変換の汎用性を高めることが重要である．本論文では形式仕様言語 Z を用いて変換手法を提示した．RSL と Z の間の関係を明確にすることにより，RSL のサブセット上で同じ変換手法を提示することは可能であろう．

論理式の書き換え 論理式の表現には，同じ制約の記述でも様々な表現方法がある．形式的な仕様記述を作成するときには，論理式の書き換えが頻繁に行われる．そこで，変換エンジンの機能として，論理式の同値関係の変換を支援する機能が望まれる．また，論理式に特有の表現方法をライブラリとして提供する機能があれば，仕様記述の助けとなる．

謝辞

本研究を進めるにあたり，終始温かい御指導を賜わりました奈良先端科学技術大学院大学情報科学研究科の福田晃教授に心より感謝の意を表します．また，本論文をまとめるにあたり，奈良先端科学技術大学院大学情報科学研究科の鳥居宏次教授ならびに関浩之教授から貴重なる御助言と御指導を賜りましたことをここに述べ，両教授に深く感謝の意を表します．

九州大学大学院システム情報科学研究科の荒木啓二郎教授には，著者在学の間，福田教授とともに御指導を賜り，また多くの対外発表の機会とその際の御支援を頂きました．荒木教授には，研究に対する基本的な心構えや，ソフトウェア工学全般に渡る広い見識からの御意見，本研究の細部に渡る議論まで幅広い御助言により，筆者の知見を広めることができましたことをここに述べ，心より感謝の意を表します．

財団法人九州システム情報技術研究所の長田正研究所長には，筆者に研究所において研究の機会と環境および多くの方たちとの交流，議論の場を提供して頂きましたことをここに述べ，感謝の意を表します．

株式会社野村総合研究所の河野勝利氏には，実際の開発現場からの観点から，形式的手法の適用可能性や有効性の議論を通して，ソフトウェア開発における問題や課題に対する筆者の認識が深まり，本研究を進める上での大きな推進力となりましたことをここに述べ，心から感謝致します．

九州大学大学院システム情報科学研究科の田口研治助手には，形式的手法に関する多岐にわたる議論を通して，多くの御助言が得られ，筆者の知見を広めることができましたことをここに述べ，深く感謝致します．

株式会社 SRA の岸田孝一氏，林香氏，佐原伸氏，および小田朋宏氏，有限会社デザイナーズデンの酒匂寛氏，ならびに株式会社ニルソフトウェアの伊藤昌夫氏には，実際のソフトウェア開発の現場と広い知見からの議論と御助言から筆者の見識が広まり，本研究に対する多大なヒントが得られたことをここに述べ，深く感謝の意を表します．

九州大学大学院システム情報科学研究科荒木研究室の諸先生方および学生の皆様に感謝致します．特に，田中俊行氏には形式的手法に関する議論にいつもつき

合って頂いて，貴重な意見を伺えたことに深く感謝致します．さらに，研究に関する討議に参加して頂いた，財団法人九州システム情報技術研究所第2研究室の研究員ならびに研究助手の方々，および研究所のスタッフの方々に心より感謝の意を表します．

最後に，大学院での研究生活に多大な協力と理解を頂いた両親および家族に心から感謝の意を表します．

参考文献

[1] Jackson, M.: Software Requirements & Specifications a lexicon of practice, principles and prejudices, Addison-Wesley, 1995. 邦訳: 玉井哲雄，酒匂寛: ソフトウェア博物誌 世界と機械の記述, トッパン, 1997.

[2] Craigen, D., Gerhart, S., and Ralston, T.: An International Survey of Industrial Applications of Formal Methods, Volume 1 Study Methodology, Tech. Report PB93-178556/AS, National Technical Information Service, 1993.

[3] Craigen, D., Gerhart, S., and Ralston, T.: An International Survey of Industrial Applications of Formal Methods, Volume 2 Case Studies, Tech. Report PB93-178564/AS, National Technical Information Service, 1993.

[4] Spivey, J. M.: The Z Notation - A Reference Manual 2nd ed., Prentice Hall, 1992.

[5] Jones, C. B.: Systematic Software Development using VDM 2nd ed., Prentice Hall, 1990.

[6] The RAISE Language Group: The RAISE Specification Language, Prentice Hall, 1992.

[7] Hall, A.: Seven Myths of Formal Method, IEEE Software, Vol 7, No. 5, pp. 11-19, 1990.

[8] Morgan, C.: Programming from Specifications 2nd ed., Prentice Hall, 1994.

[9] Wordsworth, J. B.: Software Development with Z: A Practical Approach to Formal Methods in Software Engineering, Addison-Wesley, 1992.

[10] King, S.: Z and the Refinement Calculus, Lecture Notes in Computer Science, Vol. 428, pp. 164-188, Springer-Verlag, 1990.

[11] Wood, K. R.: A Practical Approach to Software Engineering using Z and the Refinement Calculus, ACM Software Engineering Notes, Vol. 18, No. 5, pp. 79-88, 1993.

[12] Potter, B., Sinclair, J., and Till, D.: An Introduction to Formal Specification and Z, Prentice Hall, 1991.

[13] Diller, A.: Z: An Introduction to Formal Methods, John Wiley & Sons, 1990.

[14] Hayes, I. J.(editor): Specification Case Studies 2nd ed., Prentice Hall, 1993.

[15] Valentine, S. H.: Z−− an Executable Subset of Z, Z User Workshop York 1991, Springer-Verlag, pp. 157-187, 1992.

[16] Chang, H.-M., Furuki, R., and Araki, K.: Formal Approach to Modeling and Requirement Analysis with Z, Proc. of Kunming International CASE Symposium'94, pp. 5B.1-5B.14, 1994.

[17] Chang, H.-M. and Araki, K.: Modeling Support by Specification Transformation, Proc. of a Workshop on Algebraic and Object-Oriented Approaches to Software Science, pp. 133-140, 1995.

[18] 増永良文: リレーショナルデータベース入門, サイエンス社, 1991.

[19] 張漢明, 荒木啓二郎: 形式的手法による機能分割手法, 第1回ソフトウェア工学の基礎ワークショップ FOSE'94 論文集, pp. 129-134, 1994.

[20] 張漢明, 荒木啓二郎: Z仕様記述における階層的機能分割, ソフトウェア・シンポジウム'95 論文集, pp. 15-24, 1995.

[21] Chang, H.-M. and Araki, K.: Extracting System Invariants from Operation Specifications, Proc. of Changsha International CASE Symposium'95, pp. 16-22, 1995.

[22] 張漢明, 荒木啓二郎: 操作仕様記述におけるシステム状態不変条件の抽出, ソフトウェア工学の基礎II, レクチャーノート／ソフトウェア学15, pp. 183-188, 近代科学社, 1996.

[23] 河野勝利, 張漢明, 荒木啓二郎: 形式的手法に基づいた構造化ダイアグラムの一貫性検証について, コンピュータソフトウェア, Vol. 15, No. 3, pp. 2-16, 1998.

[24] Kouno, S., Chang, H.-M., and Araki, K.: Consistency Checking between Data and Process Diagrams based on Formal Methods, Proc. of COMPSAC96, pp. 261-269, 1996.

[25] 河野勝利, 張漢明, 荒木啓二郎: 形式的手法を用いた構造化ダイアグラムの一貫性検証, ソフトウェア工学の基礎II, レクチャーノート／ソフトウェア学15, pp. 81-90, 近代科学社, 1996.

[26] 河野勝利, 張漢明, 荒木啓二郎: 構造化ダイアグラムの一貫性検証に対する形式的手法の適用, ソフトウェア・シンポジウム'96論文集, pp. 99-111, 1996.

[27] Fisher, A. S.: CASE - Using Software Development Tools, John Wiley & Sons, 1988.

[28] Martin, J. and McClure, C.: Diagramming Techniques for Analysis and Programmers, Prentice Hall Agency, 1985.

[29] Martin, J.: Information Engineering Book 1, 2, 3, Prentice Hall Agency, 1989.

[30] DeMarco, T.: Structured Analysis and System Specification, Prentice Hall, 1979.

[31] Yourdon, E.: Managing the Structured Techniques 3rd ed., Prentice Hall, 1985.

[32] Korth, H. F. and Silberschatz, A.: Database System Concepts 2nd ed., McGraw-Hill, 1991.

[33] King, S. and Sørensen, I. H.: From Specification, Through Design to Code:A Case Study in Refinement, Formal Methods - Theory and Prctice, pp. 103-137, Blackwell Scientific Publications, 1989.

[34] The RAISE Language Group: The RAISE Development Method, Prentice Hall, 1995.

[35] Tamai, T.: How Domain Analysis Methods Affect Architectural Design, Proc. of Kunming International CASE Symposium'94, pp. 6B.1-6B.7, 1994.

付録

A. 銀行システムの仕様とテンプレート・スキーマ

ここでは，本文中に掲載することができなかった，変換の際に使用するテンプレート・スキーマと銀行システムの形式的な仕様を示す．

[データ・ディクショナリー]

データの型 *STRING* と *MessageType* ， *Money* を定義する．

$$[STRING]$$
$$MessageType ::= ok \mid ng$$
$$Money == -999999999 \,..\, 999999999$$

以下は，図 6.8の属性を変換したものである．

$$Message \mathrel{\widehat=} [message : MessageType]$$
$$NewBalance \mathrel{\widehat=} [newbalance : Money]$$
$$AccountNumber \mathrel{\widehat=} [cusp : CusP]$$
$$AverageBalance \mathrel{\widehat=} [averagebalance : Money]$$

[エンティティー]

以下は，図 6.5で影をつけたエンティティーを変換したスキーマである．

Customer

$$CusP \mathrel{\widehat=} [bp : BraP; id : \mathbb{N}]$$
$$CusA \mathrel{\widehat=} [name : STRING]$$
$$CusR \mathrel{\widehat=} Record[CusP, CusA]$$
$$CusE \mathrel{\widehat=} Entity[CusP, CusA][cusE/entity]$$

Trade

$$TraP \triangleq [cusp : CusP; id : \mathbb{N}]$$
$$TraA \triangleq [balance : Money]$$
$$TraR \triangleq Record[TraP, TraA]$$
$$TraE \triangleq Entity[TraP, TraA][traE/entity]$$

[関連]

関連のテンプレート・スキーマを示す.

$M2M[E1, E2]$

$$rel : \mathbb{P}(E1 \times E2)$$

$EM2M[E1, E2]$

$$M2M[E1, E2]$$

$$\forall r : \mathrm{ran}(rel) \bullet \exists d : \mathrm{dom}(rel) \bullet d \mapsto r \in rel$$

$EM2EM[E1, E2]$

$$M2M[E1, E2]$$

$$\forall r : \mathrm{ran}(rel) \bullet \exists d : \mathrm{dom}(rel) \bullet d \mapsto r \in rel$$
$$\forall d : \mathrm{dom}(rel) \bullet \exists r : \mathrm{ran}(rel) \bullet d \mapsto r \in rel$$

$M2One[E1, E2]$

$$M2M[E1, E2]$$

$$rel \in E1 \nrightarrow E2$$

$EM2One[E1, E2]$

$M2One[E1, E2]$

$rel \in E1 \nrightarrow\!\!\!\!\!\rightarrow E2$

$M2EOne[E1, E2]$

$M2One[E1, E2]$

$rel \in E1 \longrightarrow E2$

$EM2EOne[E1, E2]$

$M2One[E1, E2]$

$rel \in E1 \longrightarrow\!\!\!\!\!\!\rightarrow E2$

$One2One[E1, E2]$

$M2M[E1, E2]$

$rel \in E1 \rightarrowtail\!\!\!\!\!\!\rightarrow E2$

$EOne2One[E1, E2]$

$One2One[E1, E2]$

$rel \in E1 \nrightarrow\!\!\!\!\!\rightarrow E2$

$EOne2EOne[E1, E2]$

$One2One[E1, E2]$

$rel \in E1 \rightarrowtail\!\!\!\!\!\!\rightarrow E2$

上記に加え，関連には外部キーについての制約が必要である．外部キーについては，以下に示す FKey を使って表す．

$$
\begin{array}{|l}
\hline
[PKey, Attr] \\
\_\ \mathsf{FKey}\ \_ : PKey \leftrightarrow \mathbb{F}\ Record[PKey, Attr] \\
\hline
\forall\, a : PKey;\ e : \mathbb{F}\ Record[PKey, Attr] \bullet \\
\qquad a\ \mathsf{FKey}\ e \Leftrightarrow a \in \{r : e \bullet r.p\} \\
\hline
\end{array}
$$

関連 Belong を変換したスキーマを示す．

$$
\begin{array}{|l}
\hline
Belong \\
M2EOne[CusR, BraR][belong/rel] \\
CusE;\ BraE \\
\hline
\mathrm{dom}(belong) = cusE \land \mathrm{ran}(belong) = braE \\
\forall\, r : cusE \bullet r.p.bp\ \mathsf{FKey}\ braE \\
\hline
\end{array}
$$

[ミニ仕様書]

以下は，ミニ仕様書のエラー処理である．

$$
\begin{array}{|l}
\hline
UpBalNg \\
\Delta TraE \\
Interest? \\
AccountNumber? \\
Response! \\
\hline
interest? < 0 \\
traE' = traE \\
message! = ng \\
\hline
\end{array}
$$

ミニ仕様書の記述の際に使用した関数の定義は以下である．

$$NextId : \mathbb{P}\, TraR \times CusP \longrightarrow \mathbb{N}$$
$$recentB : \mathbb{P}\, TraR \times CusP \longrightarrow Money$$

[PAD]

PADを変換する前に，ミニ仕様書のスキーマ *UpBalOk* と *UpBalNg* に対して2つの事前処理を行う必要がある．まず，入出力変数から記号！と？を取り除き，以下のスキーマ *Remove* を使ってそれら入出力変数をダッシュなし と ダッシュ付の変数名に変換する．

Remove

$$Interest?; Interest$$
$$AccountNumber?; AccountNumber$$
$$Response'; Response!$$

$$interest = interest?$$
$$cusp = cusp?$$
$$message' = message!$$
$$newbalance' = newbalance!$$

$$(UpBalOk \land Remove) \setminus (interest?, cusp?, message!, newbalance!)$$
$$(UpBalNg \land Remove) \setminus (interest?, cusp?, message!, newbalance!)$$

以下では， *UpBalOk* と *UpBalNg* を変換後のスキーマとして扱う．以下は，PADの正常処理 *UpBalOkPrg* とエラー処理 *UpBalNgPrg* である．

UpBalOkPrg

$\Delta TraE$
$Interest$
$AccountNumber$
$Response'$

$traE' = traE \cup$
$\quad \{(\mu\, x : TraR \mid x.p.cusp = cusp \wedge$
$\quad x.p.id = NextId(traE, cusp) \wedge$
$\quad x.a.balance = recentB(traE, cusp) + interest)\}$
$message' = ok$
$newbalance' = recentB(traE, cusp) + interest$

UpBalNgPrg

$\Delta TraE$
$Interest$
$AccountNumber$
$Response'$

$traE' = traE$
$message' = ng$

コマンドの変換例のためのサンプル・スキーマを示す.

$[KEY, ATR]$
$SmplP \triangleq [key : KEY]$
$SmplA \triangleq [atr : ATR]$
$SmplR \triangleq Record[SmplP, SmplA]$
$SmplE \triangleq Entity[SmplP, SmplA][smplE/entity]$

以下は，書き込みのコマンドに対するテンプレート・スキーマである.

SmplWrite

$\Delta SmplE$

$InRec : SmplR$

$smplE' = smplE \cup \{InRec\}$

InRec は RDB に追加する入力要素である.

Assign と *Write* の 2 つの処理を変換したスキーマを示す.

Assign

$\Xi TraE$

$AccountNumber; Interest; Response'$

$ReadRec, NewRec' : TraR$

$NewRec' \in \{(\mu x : TraR \mid$
$x.p.cusp = ReadRec.p.cusp \land$
$x.p.id = ReadRec.p.id + 1 \land$
$x.a.balance = ReadRec.a.balance + interest)\}$

Write

$\Delta TraE$

$AccountNumber; Interest; Response'$

$ReadRec, NewRec : TraR$

$traE' = traE \cup \{NewRec\}$

$message' = ok$

$newbalance' = ReadRec.a.balance + interest$

[検証条件]

6.3.3項で省略した検証条件 (1 と 2) を示す.

1. 条件分岐の検証

$$UpdateBalance \sqsubseteq UpdateBalancePAD$$

上の検証条件を以下に示す.

- safety 条件

$$\text{pre } UpdateBalance \vdash (interest \geq 0) \vee (interest < 0)$$

- liveness 条件

$$UpdateBalance \wedge (interest \geq 0) \sqsubseteq UpBalOkPrg$$
$$UpdateBalance \wedge (interest < 0) \sqsubseteq UpBalNgPrg$$

上の liveness 条件は詳細化の関係を含んでいる. これら詳細化の関係を示すには以下の検証条件を証明する必要がある.

- safety 条件

$$\text{pre } (UpdateBalance \wedge (interest \geq 0)) \vdash \text{pre } UpBalOkPrg$$
$$\text{pre } (UpdateBalance \wedge (interest < 0)) \vdash \text{pre } UpBalNgPrg$$

- liveness 条件

$$\text{pre } (UpdateBalance \wedge (interest \geq 0)) \wedge UpBalOkPrg \vdash$$
$$UpdateBalance \wedge (interest \geq 0)$$
$$\text{pre } (UpdateBalance \wedge (interest < 0)) \wedge UpBalNgPrg \vdash$$
$$UpdateBalance \wedge (interest < 0)$$

2. ローカル変数の導入に対する検証

$$UpBalOkPrg \sqsubseteq UpBalOkIntr$$

上の *UpBalOkIntr* は, *UpBalOkPrg* にローカル変数を導入したスキーマである.

UpBalOkIntr

$ReadRec, ReadRec' : TraR$

$NewRec, NewRec' : TraR$

UpBalOkPrg

検証条件を以下に示す.

- safety 条件

  pre *UpBalOkPrg* $\vdash$ pre *UpBalOkPrgIntr*

- liveness 条件

  (pre *UpBalOkPrg*) $\wedge$ *UpBalOkPrgIntr* $\vdash$ *UpBalOkPrg*

B. 著者研究業績

本論文に関連する研究業績

学術論文

[1] 河野勝利，張漢明，荒木啓二郎: 形式的手法に基づいた構造化ダイアグラムの一貫性検証について，コンピュータソフトウェア，Vol.15，No.3，pp. 2-16，1998. 本論文第 6 章に関連する内容.

[2] 張漢明，荒木啓二郎： モデル形成支援のための仕様記述変換技術，情報処理学会論文誌『数理モデル化と問題解決応用』，採録決定，1999. 本論文第 3 章に関連する内容.

国際会議

[1] Chang, H.-M., Furuki, R., and Araki, K.: Formal Approach to Modeling and Requirement Analysis with Z, Proc. of Kunming International CASE Symposium'94, pp. 5B.1-5B.14, 1994. 本論文第 3 章に関連する内容.

[2] Chang, H.-M. and Araki, K.: Modeling Support by Specification Transformation, Proc. of a Workshop on Algebraic and Object-Oriented Approaches to Software Science, pp. 133-140, 1995. 本論文第 3 章に関連する内容.

[3] Chang, H.-M. and Araki, K.: Extracting System Invariants from Operation Specifications, Proc. of Changsha International CASE Symposium'95, pp. 16-22, 1995. 本論文第 5 章に関連する内容.

[4] Kouno, S., Chang, H.-M., and Araki, K.: Consistency Checking between Data and Process Diagrams based on Formal Methods, Proc. of COMPSAC96, pp. 261-269, 1996. 本論文第 6 章に関連する内容.

シンポジウム（査読付）

[1] 張漢明，荒木啓二郎: 形式的手法による機能分割手法，第1回ソフトウェア工学の基礎ワークショップ FOSE'94 論文集，pp. 129-134，1994. 本論文第4章に関連する内容.

[2] 張漢明，荒木啓二郎: Z 仕様記述における階層的機能分割，ソフトウェア・シンポジウム '95 論文集，pp. 15-24，1995. 本論文第4章に関連する内容.

[3] 張漢明，荒木啓二郎: 操作仕様記述におけるシステム状態不変条件の抽出，ソフトウェア工学の基礎 II，レクチャーノート／ソフトウェア学 15，pp. 183-188，近代科学社，1996. 本論文第5章に関連する内容.

[4] 河野勝利，張漢明，荒木啓二郎: 形式的手法を用いた構造化ダイアグラムの一貫性検証，ソフトウェア工学の基礎 II，レクチャーノート／ソフトウェア学 15，pp. 81-90，近代科学社，1996. 本論文第6章に関連する内容.

[5] 河野勝利，張漢明，荒木啓二郎: 構造化ダイアグラムの一貫性検証に対する形式的手法の適用，ソフトウェア・シンポジウム '96 論文集，pp. 99-111，1996. 本論文第6章に関連する内容.

研究会

[1] 張漢明，荒木啓二郎: Z における仕様記述変換手法，情報処理学会研究報告，94-SE-99，Vol.94，No.55，pp. 9-16，1994. 本論文第3章に関連する内容.

その他の研究業績

国際会議

[1] Kawamura, A., Oda, S., Chang, H.-M., and Araki, K.: Software Development with Common Formal System Descriptions, Proc. of International Symposium on Future Software Technology '96, pp. 146-153, 1996.

[2] Tanaka, T., Chang, H.-M., Taguchi, K., and Araki, K.: Formal Specification and Verification of Security Protocol in RSL, Proc. of International Symposium on Future Software Technology '97, pp. 143-150, 1997.

[3] Chang, H.-M., Sakoh, H., and Araki, K.: A Proposal of 4W Diagram Notation for Requirements Definition Processes, Proc. of International Workshop on Principles of Software Evolution, pp. 188-191, 1998.

研究会

[1] 長尾周司，片山徹郎，張漢明，最所圭三，福田晃: OSの自動生成に向けて，情報処理学会研究報告，96-OS-73，pp. 103-108，1996.

[2] 張漢明，田中俊行，荒木啓二郎: Z, ML, Smalltalkによるソフトウェア開発の試み，ソフトウェア・シンポジウム'97論文集，pp. 148-155，1997.

[3] 田中俊行，張漢明，田口研治，荒木啓二郎: RSLによるセキュリティプロトコルと攻撃の仕様記述，平成9年度電気関係学会九州支部連合会大会論文集，p.198，1997.

[4] 張漢明，酒匂寛，荒木啓二郎: システム要求定義における4Wダイアグラムの提案，ソフトウェア・シンポジウム'98論文集，pp. 190-197，1998.