CT COMPUTER TECHNOLOGY

見本

形式手法によるソフトウェアの仕様記述と検証

組み込みソフトへの数理的アプローチ

藤倉 俊幸 著

CQ出版社

第1章

数理的手法を使ったソフトウェアの検証
ソフトウェアの正しさをどう証明するか

1.1 ソフトウェアの正しさ

ソフトウェアの正しさには，配列オーバーフローしない，ゼロで割り算をしないなどのプログラム・コードとしての正しさと，そのプログラムが上位要求を満たしているかという二通りがある．上位要求は普通，文章などのプログラム・コード以外で記述されるのでプログラムと直接比較することはできず，正しいかどうか頭の中で考える必要がある．したがって，ソフトウェア開発では考えなければならないことが至る所にある．プログラムの正しさを示すことは，この考え方の正しさを示すこととほぼ等しいと言える．そのために，命題論理を使った検証手法を利用できる．

考え方の筋道を通して，モレやヌケがないことを確認するために命題論理を利用できるのである．考える過程を論理の世界では，推論という．推論というと堅苦しいが，要するに理由を考えたり，何かを説明したりするということである．

推論の前提となることや結論を論理的に表現するだけでなく，推論の過程も論理的に表現することができる．そして論理的な表現から真理表を作る．もし，考え方にモレやヌケがあると真理表で偽になるところがあるので検出できる．例えば，方程式$x^2=1$の解について考えてみる．

「$x=1$ならば$x^2=1$，よって$x=1$である」

この推論は**図1.1**に示した様に間違っていることが真理表から分かる．推論に含まれる論理構造を表現してその論理構造自身の真理表を作るのである．論理構造に間違いがあればその部分は偽になるので検出することができる．

では，間違いが発見されたらどうしたら良いであろうか．正しい推論とするための不足している条件を

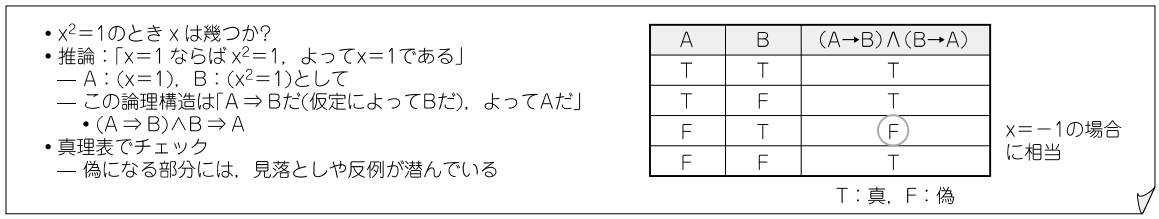

図1.1　間違った推論の例

探してそれを前提に加えてやればよい．**図1.1**の間違った部分は，Aが偽でBが真の場合である．したがって，推論の真理表全てを真にするためには，前提に¬A∧Bの否定を加えてやればよい．すなわち，

$$\neg(\neg A \wedge B) \wedge (A \Rightarrow B) \wedge B \Rightarrow A$$

とする．この真理表を作成すると全ての場合で真になる．

ここで追加した¬(¬A∧B)は変形するとB⇒Aになる．つまり，BからAを結論づけるにはA⇒BではなくB⇒Aが必要になる．元々の問題がBを前提としていたのにいつの間にかBを結論として話を進めてしまったことが間違いの原因である．

開発やデバッグの際に少しややこしいことを考えた時には，考えた過程を命題によって表現し，結論にいたる根拠を論理構造で表現し，真理表を作成すればモレを見つけることができる．そして，モレが見つかったらその条件の組み合せ(Aが偽でBが真)を真にする条件(B⇒A)を追加する．それが，見落としていた条件になる．

具体的な問題で$x=-1$に対応する具体的な条件をどのように見つければよいのか，そもそも$x=1$をどうやって見つけたのかは新しい技術を使うか，思い付くしかない．思い付きや気付きが重要である．「B⇒Aが抜けています」ということが分かっても，振り出しに戻っただけである．$x=-1$が抜けていることを思い付いても，まだほかにもヌケがあるかどうかは分からないのである．ヌケがないことを保証するには，因数分解や根の公式のような演繹(えんえき)によって証明された手法を使うしかない．そしてこれらの手法自身も最初は思い付きによって作られたのである．

数理的手法は演繹をベースにした手法であり，思い付きを肩代わりしてくれる手法ではない．思い付きこそが設計のような新しいものを作り出す作業の本質であり，演繹はそれをサポートしていると言っても良い．この思い付きの過程をアブダクションと呼ぶ．数理的手法は，思い付きが正しいことを演繹によって証明してくれる．演繹は新しいことは生み出さない，すでにあるものの形を変えて見せてくれるだけである．新しいものを生み出せるのはアブダクションだけである．しかも，その新しいことは最初は古い論理系では間違いに分類されるのである．形式手法に関する誤解はいろいろあるが(**図1.2**，**図1.3**)，これも誤解の一つだろう．

1.2　開発過程と推論

呼び方は本によっていろいろあるが，推論のタイプにはアブダクションと帰納，演繹がある(**図1.4**)．もの作りの本質はアブダクションであるが，アブダクションには間違いが混入する可能性があるので，検

- 形式手法はソフトウェアが完全であることを保証できる
- 形式手法とはすべからくプログラムの証明である
- 形式手法はセーフティ・クリティカル・システムにのみ有効である
- 形式手法は高度に訓練された数学者を必要とする
- 形式手法は開発コストを増加させる
- 形式手法はユーザには受け入れられない
- 形式手法は現実の大規模ソフトウェアには使われない

図1.2　形式手法に関する七つの誤解　Hall版[1.1]

- 形式手法は開発過程を遅らせる
- 形式手法にはツールがない
- 形式手法は従来の設計手法に置き換わる
- 形式手法はソフトウェアにのみ適用できる
- 形式手法は必要ない
- 形式手法は支持されない
- 形式手法の人達は常に形式手法を用いる

図1.3　形式手法に関するもう七つの誤解　Bowen&Hinchey版[1.2]

仮説形成またはアブダクション(abduction)
　— 仕様決め，設計，実装における推論
帰納(induction)
　— テストや実験，シミュレーション
演繹(deduction)
　— 形式検証

図1.4　推論のタイプ

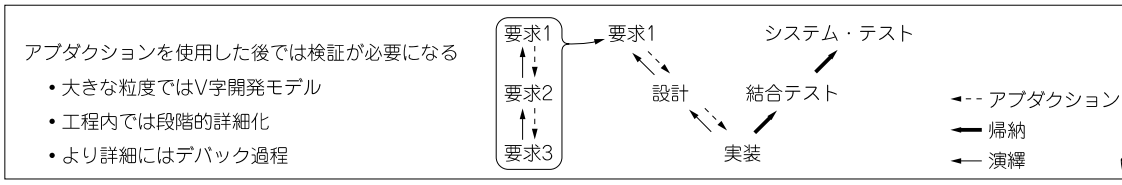

図1.5　推論タイプの開発プロセスへのマップ

証が必要になる(**図1.5**，**図1.6**)．検証の手段として帰納による場合と演繹による場合がある．テストは帰納に属する検証法で網羅性に不安がある．演繹証明は難しいことが多い．網羅的なアプローチであるモデル検査は状態爆発が起こる．

テストも演繹証明もモデル検査も検証側の技術であって，アブダクションのような新しいモノを作り出す技術ではない．数理的なアプローチは，推論過程の中のアブダクション部分を検出してリスク箇所を明

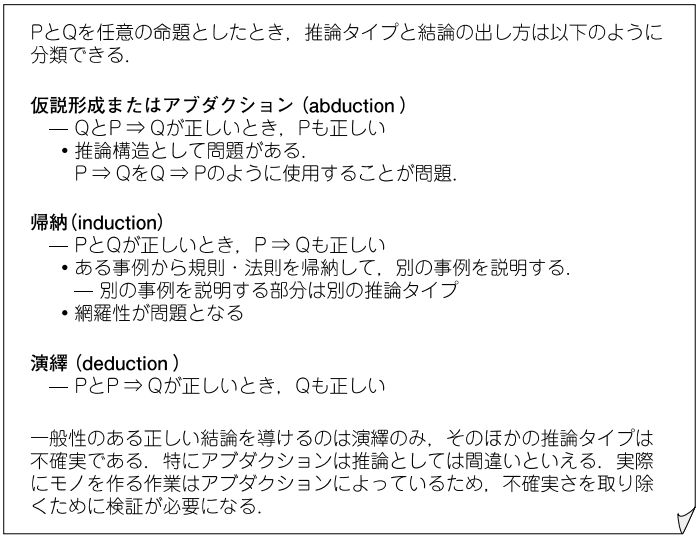

図1.6
推論タイプによる前提と結論の違い

確にすることと，開発工程の中に演繹による部分を増やして品質を確保することと，自動化を進めることを目的としている．また，テスト対象部分の全体空間を明らかにして実施したテストのカバレッジなどの管理に利用することもできる．

演繹に依った部分（数理的手法を使った部分）は基本的に間違いは無い筈なので，演繹できなかった部分にレビューやテストを集中させることができる．演繹は，新しいことを生み出さないが自動化できるのも特徴の一つである．演繹する技術レパートリーを拡大していくことが重要である．

第2章

真理表を使いこなす（1）
仕様の形式化のはじめの一歩

2.1　命題論理から始めよう

● モデル検査は市民権を得たが，形式仕様記述はまだ

近年，モデルを使った「モデル検査」という言葉が市民権を得たような感がある．モデル検査を使うと「何ができて，何ができないのか」ということもだいぶ明確になってきたように思う．

しかし，本書で扱う形式仕様記述[注1]については，まだこれからではないかと思う．ここでは形式仕様記述をどのように使うのかという点にフォーカスして解説する．

● 中学で習った命題論理からはじめよう

形式仕様記述というとVDM++[注2]が思い浮かぶが，いきなり使おうとしてもちょっと難しい．何が難しいかというと，仕様を形式化するために述語論理という馴染みのないものを使うところである．

そこで，まず一般的な命題論理からはじめて，命題論理が使いづらいと感じるようになったところで述語論理に移るのがよいのではないかと思う．命題論理は，中学や高校の数学でも出てくるし，C言語ならandやorやnotで表現できる．最初は，命題論理式に馴染むことだ．また，命題論理もいろいろと便利な使い方があり，それを知っておいて損はない．

ここでは，命題論理を中心にして仕様を形式化するということを考える．

注1：数学的な手法を用いて仕様を記述すること．専用の形式使用言語などを用いる．
注2：形式検証ツール．国内では`http://www.vdmtools.jp/`が詳しい．

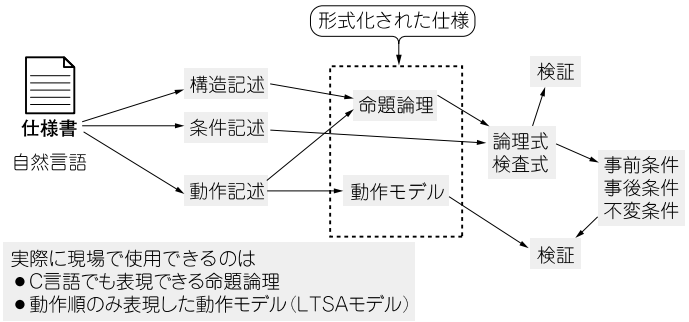

図2.1　仕様の形式化

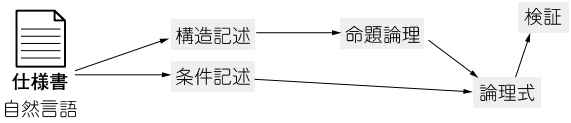

図2.2　当面の目的

一般に，仕様書の中には，「構造に関する記述」，「条件に関する記述」，「動作に関する記述」が含まれている（**図2.1**）．組み込みソフトウェアの仕様書では，動作や操作に関する記述が重要だ．そこで，動作に関する記述については論理式ではなく，LTSA[注3]などを利用して動作モデルによって形式化した方がよい．そのため，モデル検査が先行して使われるが，モデル検査を行うためには検査式が必要になる．検査式というのは時相論理式のことであり，最終的には論理式の扱いに慣れておく必要がある．論理式には命題論理と述語論理，時相論理の3種類があり，この順に難しくなる．つまり，命題論理をやっておくとモデル検査でも役に立つのである（**図2.2**）．

● 仕様書を論理式で書き直す

当面の目的は，仕様書に書いてあることを命題論理式に書き直すことだ．仕様書の何パーセントが命題論理式に変換できるのかというと，おそらく10%から20%ではないかと思う．これは，「かつ」，「または」，「ならば」，「〜でない」だけで表現できる部分に対応する．少ないと思うかもしれないが，これでも御利益はある．

2.2　形式化すると何が嬉しいのか

● $x^2=1$ の解を例に考えてみよう

仕様書を命題論理式で形式化すると，何が嬉しいのかということが重要である．まずは，そのことを実感してみよう．

注3：形式検証ツール．http://www.doc.ic.ac.uk/ltsa/

第15章

形式手法導入でバグはどれくらい減らせたのか
形式仕様記述は役に立つのか

15.1　早い段階でバグを取り除く

本書では，プログラム仕様を正確に記述すること，そして検証を行うための技術を扱っている．この技術を使うと開発の初期の段階でバグを取り除くことが可能になる．形式手法は，数学や論理学を使うので慣れるまでは大変だが，この壁を乗り越えれば効率的にバグを減らすことができると信じている．…が，使う人が何時までたっても増えてこない．確かに早い段階でバグを取り除くことはできるが，手間が増えるだけでバグが減らなかったという開発シミュレーションの報告が参考文献(15.1)にある．どんな報告か読んでみよう．

その報告では，形式仕様記述ツールのVDM(The Vienna Development Method)を使って効果を定量的に調べる実験を行っている．実験といっても，あくまでも思考実験であり実際に何かを開発したわけではないらしい．でも，その割にはリアルだ．ソフトウェア開発というのは人に強く依存する．開発者が変われば状況も変わるので比較できないし，同じ開発者による実験でも1回目と2回目では学習してしまうので比較できない．だから思考実験によって調べるしかないのかもしれない．

15.2　VDMの特徴

● 海外では多く使われているVDMだが…

報告で使用しているVDMとは，形式手法を使って要求を厳密に記述して設計品質を向上させるためのツールであり，形式仕様記述の老舗で，形式手法の事例では必ず出てくる．この連載でこだわっている命

題論理ではなく，やや難易度の高い述語論理をベースにしている．VDMは2005年から日本のCSKシステムズが所有していて，研究会[15.2]などを立ち上げて普及に努めている．ISOにもなっているフリーのツールだが日本では使う人が少ない．しかし，世界に目を向ければ20年も前からコンスタントに使われている有名なツールだ[15.3]．

最近IPA(独立行政法人 情報処理推進機構)から発表された参考文献(15.3)に掲載されている形式手法関連プロジェクト数の推移を**図15.1**に示す．多いとはいえないが2000年ぐらいから機能安全などの規制が入る関係で増えてきている．この中の11.8%がVDMを使用している．それなのに日本では皆無といってもよい状態になっていて，CSKもなかなかビジネスにならず持て余し気味と聞いている．その謎が明らかになるかもしれない．

実験は形式手法を使わなければならなくなった学生を想定して行われている．したがって，初心者が段階的にVDMを利用していく設定になっている．導入のステップは**表15.1**のようになっている．VDMにはVDM++とVDM-SLの2種類がある．今回使用するのはVDM-SLである．

● Step-1：外部仕様記述を利用する

表15.1の中のデータ型記述は，整数とか実数とかプログラム言語における型定義と同じような方法で行われる．幾つかのフィールドを持ったレコード型も定義できる．いきなりデータ型が出てくる開発とはいかがなものかという気もするが，仕様が与えられた課題なのと，今回は設計工程が対象なのでデータ型の定義から始まっても不思議はない．VDM++の場合はVDM-SLにオブジェクト指向を追加したものなので，もっと上流から使いやすくなっている．VDM-SLでも，集合型や写像型，直積型などの概念を記述するための型も用意されているので設計工程以前から使用することができる．実際，これらの概念を記述するための型を使わなければ本当のVDMの効果は測定できない．

例えば，**リスト15.1**のように型を定義する．温度によって角度を変える制御が必要であれば**リスト15.2**のようなデータ型として定義できる．この抽象度が重要でVDMらしさである．テーブルのようなデータとして実装されるかもしれないし，関数として実装されるかもしれないが，そのような詳細はとりあえず保留しておくことができる．

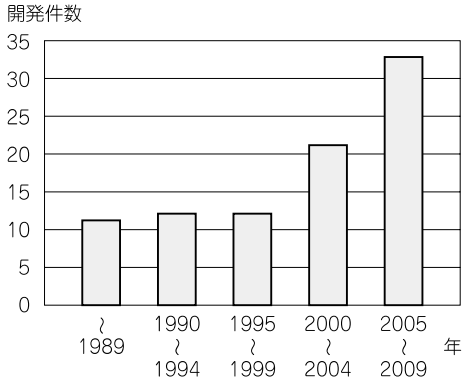

図15.1　形式手法関連プロジェクト[57]

表15.1　VDM導入ステップ

Step 1	外部仕様記述を利用する	データ型記述，データ不変条件記述，陰関数記述，事前条件・事後条件の記述
Step 2	内部仕様記述を利用する	陽関数記述を加える．完全にVDM-SLで設計をする段階
Step 3	検証を行う	Validation(仕様実行)を行う

データ不変条件記述というのは，例えば温度であれば－273.15℃以上，角度であれば0°以上360°未満のような，そのデータが常に満たさなければならない付加的制約のことである．**リスト15.3**のように指定する．

データ不変条件記述は検証する際に重要になってくる．当たり前の性質を書くことが多いが無駄ではない．当たり前の記述を積み上げて，システム全体の本質的な性質が構成されるのである．また，自然言語の仕様書では暗黙知のような形で埋もれてしまう制約を明記することで，あるドメインでは常識的だが別のドメインではそうでない制約を取りこぼすことがなくなる．

陰関数記述というのは，関数の入出力だけ記述して中身は考えない関数定義である．引数の型と数，戻り値の型を定義する．そして，具体的な入力値ではなく入力の条件を記述するのが事前条件で，出力の条件を記述するのが事後条件である．例えばRTOSのスケジューラを考えるときに，具体的なスケジューリング・アルゴリズムではなく，「実行するタスクはレディ・キューの中に存在している，選択したタスクはレディ・キューから外される」，というような記述をする．第12章で紹介したCBMCを使ったプログラム検証で`assume()`で記述するのが事前条件，`assert()`で記述するのが事後条件に対応する．`assume`と`assert`だけ記述することでCBMCでも陰関数定義をすることができる．

開発を進める際にはまず関数についての陰関数定義を行って，インターフェースや性質を決めてから内部の詳細に入っていくのが普通であるが，初心者には結構難しい．次に述べる陽関数定義の方が簡単である．Step-1に定義されている項目はVDMのキモの部分で，導入の第1ステップとしてはハードルが高い．

例えば，参考文献(14.2)の住所を登録する関数の陰関数定義は，**リスト15.4**のように，事前条件として「入力は未だ登録されてない名前である」こと，登録後は「bookの中に名前とアドレスが対応付けられて入っている」こと，を比較的簡単に指定できる．preの行が事前条件，postの行が事後条件指定である．一般の関数を陰関数定義するのは容易ではない．10進2進変換関数の場合は，**リスト15.5**のようになり関数そのものを書くのと変わらなくなる．そして，述語論理(例えばforall)などの基礎的な教育を受けないと読むのも難しい．魔方陣を生成するプログラムではpostの部分がかなり長くなってしまう．

リスト15.1　温度と角度の型定義

```
Temperature=real
Angle=real
```

リスト15.2　温度によって角度を変える制御の定義

```
Control=map Temperature to Angle
```

リスト15.3　付加的制約の定義

```
Temperature=real
inv t==t> -273.15

Angle=real
inv a==a>=0 and a<360
```

リスト15.4　住所を登録する関数の陰関数定義

```
function IAddAddress(name: Name,address: Address,book: AddressBook)
r: AddressBook
pre name not in set dom(book)
post r=book munion{name|->address}
```

リスト15.5
10進2進変換関数
の陰関数定義

```
function dec2bin(X : nat)b: seq of nat
pre X>=1
post forall i in set inds b&b(i)=1 or b(i)=0 and b(1)=1 and X=NatSeqSum
( [b(i)*2**(len b-i) |i in set inds b] )
```

リスト15.6
住所を登録する関
数の陽関数定義

```
function AddAddress(name: Name,address: Address,book: AddressBook)r: AddressBook
AddAddress(name,address,book) == book munion{name|->address}pre name not in set
dom(book)
```

● Step-2：内部仕様記述を利用する

陽関数定義は，事後条件の代わりに実際に関数の中を書くので普通のプログラミングと同様である．住所を登録する関数の場合，**リスト15.6**となる．この場合，bookに新しいnameとaddressの対応関係を追加する処理としてmunionを使っている．結果的に事後条件と同じになる．陽関数定義を行うことでシミュレーションが可能になる．

● Step-3：検証を行う

検証にはいろいろなタイプがあるが，この場合の検証は，陽関数定義を使ったシミュレーションによるテストのことである．関数単体のテストと操作シナリオを使った結合テストに対応するテストを実行した．テストなのでテスト・データが必要になる．関数の事前条件・事後条件，データの不変条件からテスト・ケースを作ってテストした．扱いとしては完全にテストである．ただし，これはコーディング前の出来事である．

15.3　個人対象の開発プロセスPSPの特徴

PSPとはThe Personal Software Process[15.4]のことで，個人対象の開発プロセスで，レビュー時間やバグの数などの開発中のデータを細かく取って開発効率を計測する手法である．PSPには教育コースという側面もあり，10題程度のプログラミング課題があって，それを指示に従ってメトリックスを取りながらこなしていくと自分の生産性などがわかって見積もり精度が向上する．さらに，コーディング前にバグを取り除けるようになって品質も向上する．

作業時間などのメトリックスを取って集計するのは結構手間が掛かるので，Process Dashboard[15.5], [15.6]というフリーのJavaで書かれたツールも存在している(**図15.2**)．また米国Carnegie Mellon大学のWebサイトから教育用素材も無料でダウンロードすることができる[15.7]．しかし，教育を受けて現場に戻っても継続的に使う人は意外に少なくて，効果が続かない．この状況は形式手法の置かれた状況と似ている．

PSPには，プロジェクトとして実施するために上位にTSP(Team Software Process)がある．そして，さらに組織として実施するためのCMMI(Capability Maturity Model Integration；能力成熟度モデル統合)などにつながっていく[15.8]．CMMIの認定を受けたら最終的には各開発者が何をすべきかというところまで落とさなければ現実味は出てこないが，PSPは開発者レベルまで落とす枠組みを提供している．CMMIの後，PSPまで落として使っているところは，どの程度あるのだろうか．

見本

ISBN978-4-7898-3808-5

C3055 ¥3200E

CQ出版社

定価：本体3,200円（税別）

組み込みソフトへの数理的アプローチ

■CONTENTS

第1部 プログラムの基本，論理編
【第１章】 ソフトウェアの正しさをどう証明するか
【第２章】 仕様の形式化のはじめの一歩
【第３章】 推論の正しさの検証
【第４章】 前提が正しいとするとどんな結論を得られるのか―充足問題
【第５章】 真理表から場合分け表へ
【第６章】 AND-OR木を形式化する
【第７章】 原因結果グラフ
【第８章】 組み合わせ問題とLTSA
【第９章】 組み合わせ問題で写像に親しむ
【第10章】 Alloyによるモデリング
【第11章】 要求のトレーサビリティとモデリング
【第12章】 プログラム検証とテスト
【第13章】 TDD(テスト駆動開発)におけるテスト
【第14章】 検証しながらプログラムを作るPDD
【第15章】 形式仕様記述は役に立つのか
第2部 組み込みの基本，ふるまい編
【第16章】 時間仕様の扱い
【第17章】 数理的アプローチによる開発
【第18章】 論理式のテスト
【第19章】 論理式の実験場を作る，全数チェックへの道
【第20章】 様相論理で変化を扱う
【第21章】 時相論理の導入：CTLとLTL
【第22章】 到達可能性と安全性を検証する
【第23章】 オーバーラップ制御用ステート・マシンの設計と検証