Modeling in Event-B: System and Software Engineering
イベント B でモデリング: システムとソフトウェア工学

Jean-Raymond Abrial

2011 年 9 月 20 日

目次

第 1 章

序論

1　本書の動機

この本の目的は、読者に***モデリング*** *(modeling)* と***形式的な推論*** *(formal reasoning)* についての理解を与えることである。これらの活動はコンピューター・システムの実際のコーディングに取り掛かる***前に*** *(before)* 行い、そのため、そのシステムは***構成的に正しい*** *(correct by construction)* ものになる。

この本では、どのようにプログラム——さらに一般的に言えば、離散システム——のモデルを作るかを学ぶことができる。ただし、これは読者の***頭の中で実習*** *(practice in mind)* してもらうことにする。私たちは、多様なコンピューター・システムの開発に取材した多数の例——逐次実行プログラム、並列プログラム、分散プログラム、電子回路、化学反応システムなど——から学ぶ。

読者はプログラムのモデルはプログラムそのものとはいささか異なることを理解する。そして、モデルについて***推論する*** *(reason)* のはプログラムについて推論するよりもはるかに易しいことを学ぶ。さらに、読者はとても重要な概念である***抽象化*** *(abstruction)* と***精錬化*** *(refinement)* を意識するだろう。これは、目的のプログラムに向かって徐々に正確になっていくモデルの系列の、最後のモデル (最終モデル) だけが実行可能なプログラムであるというアイディアである。(建築家の青写真を想像して欲しい。)

私たちはモデルを***推論する*** *(reasoning)* とは何かを明確にできる。モデルの推論は単純な数学的手続きで可能である。必要な数学的手続きは、まずいくつかの例を用いて、次いで古典論理 (命題論理と述語論理) を復習することで示す。非常に厳格に証明を行うことの必要性も理解していただけると思う。

また、証明に失敗したという事実からモデルの矛盾の存在を検出できることも理解できると思う。証明の失敗は、モデルの誤りと不足が何であるかを知る重要な手がかりを与えてくれる。

この本で使用する形式は Event-B という名前である。Event-B は B 形式 [1] の拡張であると同時に単純化でもある。B は 10 年前に開発され、すでにいくつかの大きな産業プロジェクト [4, 3] で使われている。Event-B のような形式のコンセプトは目新しいものではない。Event-B の原型となる多数の形式——Action Systems [6]、TLA$^+$ [2]、VDM [15]、UNITY [5] など——が長い時間をかけて提案されてきた。

この本は例を中心に構成されている。それぞれの章は新しい例 (複数であるときもある) と、その例で使う数学的な概念を理解するために必要な形式を併せて含む。もちろん、これらの説明はただ繰り返すのではなく、より詳細にしていく。それぞれの例はほぼ独立した説明である。それぞれの例で行われる証明はオープンソースのツール Rodin Platform [7] でも可能である (ウェブサイト event-b.org も見るとよい)。

この本は 1 章を 1 回ないし複数回の授業で解説することで教科書として使用できる。次の節で各章の概要を説明したあと、導入コースまたは上級コースとして使うための方法も提案する。

2　各章の説明

各章のリストと簡単な概要を示す。

2.1　第 1 章: イントロダクション

この最初の (技術的でない) 章の目的は***形式手法*** *(formal method)* という概念を導入することである。また、***モデリング*** *(modeling)* という用語の意味を明確にすることも目的とする。私たちはモデリングのための体系的な***約束事*** *(convention)* を説明する。ただし、システムの要求が何であるか知らずにモデリングに乗り出すことは意味がないことを注意しなくてはならない。そのため、***要求文書*** *(requirement document)* をどのように書くべきかを学ぶ。

2.2　第 2 章: 橋上の自動車の制御

この章の目的は小さなシステム開発の完結した例を示すことである。私たちは島と本土をつなぐ片側通行の橋の上の自動車を制御するシステムのモデルを作る。さらに、島に入れる自動車の数は制限されている。物理的な設備は信号機と自動車センサーから構成される。

開発の過程で、読者は私たちの体系的な手法を意識するだろう: それは、段階的に正確になるモデルの系列を作るというものである。それぞれのモデルは高級言語によるプログラミングを表現するのではないことを注意しておく。むしろモデルはシステムの***外部観察者*** *(external observer)* からどのように見えるかを定式化する。

それぞれの段階のモデルは解析および証明されることでいくつかの基準に対して***正しい*** *(correct)* という確証が得られる。その結果、最後のモデル (最終モデル) が証明されたとき、このモデルは***構成的に正しい*** *(correct by construction)* と言うことができる。その上、このモデルは最終的な実装にとても近く、本物のプログラムに容易に変換できる。

前述の正しさの基準は、***証明義務規則*** *(proof obligation rule)* をモデルに適用することで、完全に明確かつ体系的にすることができる。この規則を適用すると、私たちはいくつかの言明[*1]を形式的に証明しなければならなくなる。この目的のために、私たちは古典的な***シークエント計算の推論規則*** *(rules of inference of the sequent calculas)* を思い出す必要がある。このような導出規則は命題論理 (propositional logic)、等式 (equality)、初等算術 (basic arithmetic) を含む。明らかに、このような証明は自動定理証明器でも可能だが、それを使う前に手動での証明を練習しておくことをお勧めする。ちなみに、自動証明器の動作はここで説明するものとは異なることを注意しておく。ほとんどの自動証明器は、人間がする証明とは異なる方法で動作する。それによって、モデルがいくつかの基準に対して正しいということを保証できる。その結果、最終モデルが完成したとき、私たちはこのモデルが構成的に正しいと言うことができる。さらに、この最終モデルは簡単に本物のプログラムに変換できる。

[*1] 訳注:「言明」は原文では statement。論理学で証明の対象となる文を statement と言い、sentence とは言わない。

2.3 第 3 章: 機械プレスのコントローラー

この章では私たちは再び完結したシステムのコントローラーを作る。今回は機械プレスのコントローラーである。この章の目的は、正しいコードを得る体系的な方法を見せることである。私たちは、いつものように、まずこのシステムの要求文書を示す。続いて、2 種類の普遍化された***デザインパターン*** *(design pattern)* を開発する。これらのパターンの開発には、不変式 (invariant) と限定条件 (guard) を発見するために証明を用いる。最後に、機械プレスの本流の開発を行う。

この章では、定式化されたデザインパターンで正しい開発を助ける方法を示す。

2.4 第 4 章: 簡単なファイル転送プロトコル

この章で示す例は以前のものとは少し異なる。以前の 2 個の例はいずれも外的な状況 (橋上の自動車または機械プレス) を制御することを前提としていた。この章ではコンピューターネットワークで 2 者のエージェントによって使われるプロトコルを示す。これはとても古典的な 2 フェーズのハンドシェークプロトコルである。この例のとても素晴らしい解説は Lamport の本 [2] にある。

この例では私たちの数学言語を拡張して、部分写像、全域写像、定義域、値域、制限などの構造を使えるようにする。私たちはまた、論理言語と導出規則に***全称論理式*** *(universally quantified formula)* とそれに対応する推論規則を導入する。

2.5 第 5 章: Event-B の記法と証明義務規則

これまでの章では、Event-B の記法と、それに対応する多様な証明義務規則を、体系的な導入無しで使用してきた。そのかわり、例の中で、必要に応じて示してきた。これは簡単な例では十分である。なぜなら、これまでの例では記法と規則の一部分しか使わないからである。しかし、後の章でもっと複雑な例を解説するようになると、このやり方では不十分である。

この章の目的はこの状態を正すことである。まず Event-B の記法の全てを、滅多に使わないものも含めて示し、また全ての証明義務規則を示す。これは簡単な例とともに示すことにする。証明義務規則の数学的な正当性は 14 章で示されることを注意しておく。

2.6 第 6 章: 境界有り再送信プロトコル

この章では 4 章のファイル転送プロトコルを拡張する。この章では 2 個の拠点を繋いでいるチャネルが***信頼できない*** *(unreliable)* と仮定する。すると、境界有り再送信プロトコルを実行した結果は、連続したファイルの***部分的な*** *(partially)* コピーに過ぎなくなる。この例の目的は、誤り許容を用いることでこの種の問題のもとでファイルをコピーする方法を学ぶこと、それを形式的に推論することである。この例は文献 [8] に関連する多くの論文で研究されている。

この章では以前の章ほどは証明を行わないことを注意しておく。私たちはヒントを示し、形式的な証明は読者に委ねる。

2.7 第 7 章: 並列プログラムの開発

これまでの章では***逐次実行*** *(sequential)* プログラムと***分散*** *(distributed)* プログラムの開発の例を見てきた (15 章で逐次実行プログラムに戻ってくる)。この章では***並列*** *(concurrent)* プログラムの開発を見ていく。並列プログラムは分散プログラムとは異なる。分散プログラムはさまざまなプロセスが異なるコンピューターで実行され、特定の目標のために***協調*** *(cooperate)* して (明確に定義されたメッセージを交換することで) 動作する。これは 4 章と 6 章の例で顕著であり、10, 11, 12, 13 章もそれにあたる。

並列プログラムでは、複数のプロセスが存在することは同じだが、それらのプロセスは同じコンピューターで実行され、共有資源へのアクセスのために***競合*** *(compete)* している。並列プログラムはメッセージを交換せず (プロセスは互いに無視する)、かわりに他のプロセスに対してランダムに割り込む。私たちは、シンプソンの「4 スロット完全非同期機構」(4-slot Fully Asynchronous mechanism) [14] として知られている並列プログラムを開発することで、この手法を示す。

2.8 第 8 章: 電子回路の開発

この章では電子回路をシステマティックに開発する方法論を示す。これによって、Event-B のアプローチが別の実行パラダイムにも適用できることが分かる。この章で用いられるアプローチは、15 章で使われる逐次実行プログラムの開発に用いられるものと似ている。すなわち、回路はまず単一のイベントによって定義される。ここで単一のイベントとは、「一気に」事がなされるものをいう。そして次に、最初の極端に抽象的な遷移は、精義化されて複数の遷移に分割される。この分割は、構文規則を適用することで、これらの遷移が統合され、単一の回路になるまで行われる。

2.9 第 9 章: 数学言語

この章は例を含まない。かわりに、この本で使う数学言語の形式的な定義を含む。この説明は 4 個の部分——命題論理、述語論理、集合論、算術——から成る。それぞれの言語は以前のものの拡張として導入する。

しかしながら、これらの言語を導入する前に、まずはシークエント計算の簡単なまとめを説明する。これは証明のための多数の概念を強調するためである。

この章の終わりには、古典的ではあるが「高度な」多数の概念——推移閉包 (transitive closure)、さまざまなグラフ特性 (graph properties) (特に強連結)、リスト (lists)、木 (trees)、整礎関係 (well-founded relations)——を定式化する方法を示す。これらの概念は後の章で使われる。

2.10 第 10 章: 環状ネットワークでのリーダー選出プロトコル

この章では、分散コンピューティングの別の興味深い問題を勉強する。この例では非常に多数である可能性のある (ただし有限である) エージェントを扱う。これは 4 章、6 章のファイル転送プロトコルでエージェントが 2 者であるのとは対照的である。それぞれのエージェントは異なる拠点におり、それぞれの拠点は環状の双方向チャネルで繋がれている。個別のプログラムを分散して実行した結果として、***ただ一つのエージェント*** *(unique agent)* を「リーダーに選出」しなければならない。この例は G. Le Lann が 70 年代に書いた論文 [9] をもとにしている。

この章の目的はモデリングについて、特に非決定的な分野を学ぶことである。私たちはまた、関係における集合の像 (image of a set under a relation)、関係の上書き演算子 (relational overriding operator)、関係の合成演算子 (relational composition operator) などの数学的な約束事を使用する。これらの数学的な約束事はこの章で導入する。最後に、いくつかの興味深いデータ構造——環状および線形のリスト——について学ぶ。これらもこの章で導入される。

2.11 第 11 章: 木構造ネットワークでのプロセスの同期

この章で示す例は、前の章で扱った環状のネットワークよりも少し複雑な構造、木構造のネットワークを用いる。それぞれのノードではそれぞれのプロセスが同じ仕事をしている (この仕事の正確な性質はここでは重要でない)。外から見てこれらのプロセスが満たさなければならない制約は、プロセスが同期していることである。さらに、それぞれのプロセスはすぐ隣のノードとしか通信できない。この例は多数の研究者に研究されている [10, 11]。

この章では、私たちは木という興味深い構造に出会う。私たちはこのようなデータ構造を定式化する方法を学び、帰納法 (induction rule) を用いて効果的に論証する方法を見ていく。木構造は既に 9 章で導入されていることを注意しておく。

2.12 第 12 章: モバイルエージェントのルーティングアルゴリズム

この章で開発する例の目的は、携帯電話にメッセージを送る興味深いルーティングアルゴリズムを示すことである。この例では、前の章で触れた木構造に再び出会う。しかし今回は木構造は動的に変化する。また、今回の例ではクロックが本質的な役割を果たす (これは 6 章の境界有り再送信プロトコルと同じである)。この例は文献 [12] から取った。

2.13 第 13 章: IEEE 1394 プロトコル: 連結グラフでのリーダー選出

この章の例は 10 章と同じリーダー選出プロトコルであるが、この章ではネットワークは単なる環状よりも複雑である。詳しく言えば、IEEE 1394 プロトコル [13] の目標は、通信チャネルで接続された有限のノードから成るネットワークにおいて、有限時間で特定のノード (リーダー) を選ぶことである。この選出は分散かつ非決定的に行われる。

ネットワークはある特殊な性質を持っている。数学的構造としては、これは***自由木*** *(free tree)* と呼ばれている。自由木は、対称、非反射、連結、無閉路な有限グラフである。私たちは、このような、9 章で示した複雑な構造をどのように扱い、推論するかを学ぶ。

2.14 第 14 章: 証明義務規則の数学的モデル

この章では、5 章で導入した証明義務規則の、数学的な正当性を示す。これは Event-B の開発の軌跡意味論 (trace semantics) をもとにした集合論的なモデルを構築することで行われる。この章では、この本で使用している証明義務規則が Event-B の数学的モデルが要求するものと同値であることを示す。

2.15　第 15 章: 逐次実行プログラムの開発

この章は全て逐次実行プログラムの開発に費やす。私たちはまず逐次実行プログラムの構造について学ぶ。逐次実行プログラムは、複数の代入文を演算子——逐次結合 (sequential composition)、条件、ループ——でつないだものである。私たちは、これをどのように単純な遷移でモデル化するかを見ていくことにする。単純な遷移は Event-B の形式の本質である。ここで見ていくのは、一度、多段階の精義化を行うことで単純な遷移を開発して、その後、単純な遷移が多数の統合規則を用いて結合されることである。この結合規則の本質は完全に構文論的である。

これらは全て、単純な配列と数値のプログラムから、より複雑なポインタを用いたプログラムに至る多数の例で示される。

2.16　第 16 章: 位置アクセスコントローラー

この章の目的は 2 章 (橋上の自動車の制御) と 3 章 (機械プレスの制御) で扱ったような完結したシステムの別の例を示すことである。私たちは特定の人たちが「ワークスペース」内の別の場所に進入することを制御するシステムを構築する。ワークスペースとは、例えば、大学の校舎、工場、軍事施設、ショッピングモールのようなものである。

この章で学ぶシステムはこれまでのものより少し複雑である。特に、数学的構造はより高度なものを使う。モデルの推論の過程で、要求文書の重要な抜けを見つけることも、この章の目的とする。

2.17　第 17 章: 鉄道システム

この章の目的は完結したコンピューター化されたシステムの仕様作成と構築である。例として***鉄道システム*** *(train system)* を挙げる。このシステムは、実際には、***鉄道管制官*** *(train agent)* によって管理されている。鉄道管制官の役割は、線路網の監視下にある部分を通過する列車をコントロールすることである。私たちが作ろうとしているコンピューター化されたシステムは、鉄道管制官がするこの仕事を助けるものである。

この例は複雑なデータ構造 (線路網) の興味深い事例を示す。このデータ構造は数学的性質を注意深く定義しなければならない。私たちはそれが可能であることを示す。

この例はまた、最終的な製品の信頼性を絶対的に不可欠とする非常に興味深い事例である。ソフトウェア製品の完全に自動的な指示に従って、複数の列車が線路網を安全に通過しなければならない。この理由により、起こりうる悪い事態を研究し、完全に避けるかまたは安全に管理しなければならない。

このソフトウェアは、注意深くコントロールしなければならない環境に関わっている。というのも、この章で提示する形式的なモデリングは、私たちが作ろうとしているソフトウェアだけではなく、環境の詳細なモデルを含んでいるからである。私たちの究極の目標は外部の設備と完全に同期して動くソフトウェアである。外部の設備とは、具体的には線路回路、ポイント、信号機、そして運転手である。私たちは、列車が信号 (ソフトウェアが設定する) を守り、線路を周回し、その線路のポイントが動いていて (これもソフトウェアが設定する)、全ての列車が完全に安全であることを証明する。

2.18 第 18 章: 練習問題

この最後の章は読者が挑戦すべき問題のみを集めている。練習とプロジェクトをそれぞれの章に分散させるよりは、一つの章に集めた方が良いと判断した。

全ての問題は Rodin プラットフォームで実行できる。Rodin はウェブサイト “event-b.org” からダウンロードできる。

練習 (比較的簡単であることを前提としている) とプロジェクト (練習よりも大きく難しいことを前提としている) に併せて、Rodin プラットフォームで証明できる数学的な開発も提案している。

2.19 この本の使い方

この本で提供する素材はさまざまなコースを教えるために使用できる。そのコースは本質的には導入コースと上級コースの 2 種類である。これら 2 種類のコースを以下に示す。

導入コース

導入コースでの危険は素材を多く解説し過ぎてしまうことである。この悪影響は聴講生が完全に圧倒されることである。解説してもかまわないものは以下の通り:

- 1 章 (イントロダクション)
- 2 章 (橋上の自動車)
- 3 章 (機械プレス)
- 4 章 (簡単なファイル転送)
- 5 章 (Event-B 記法)
- 9 章 (数学言語)
- 15 章 (逐次実行プログラムの開発)

このコースの要点は、複雑な概念に出会うことを避けて、簡単な数学的構造——命題論理、算術——と簡単な集合論的構造のみに触れることである。

2 章 (橋上の自動車) は重要である。なぜなら、この例はとても理解しやすく、なおかつ Event-B と古典論理の基本的な概念が導入されるからである。しかしながら、教師はこの章をとてもゆっくり説明するように注意しなくてはならない。特に証明は気をつけなければならない。なぜなら、学生たちはこの種の素材に最初に触れるときとても混乱するからである。この例でのデータ構造——数と真偽値——はとても単純である。

3 章 (機械プレス) は再び完結した開発を見せる。これは形式的デザインパターンの使用例になっている。形式的デザインパターンはコントローラーを体系的に開発するのに役に立つ。

4 章 (単純なファイル転送) はとても単純な分散プログラムを示す。学生たちは、とてもよく知られた分散プロトコルを得るために仕様を作成し、精義化する方法を学ぶ。学生たちは、このようなプロトコルは非常に抽象的な (まだ分散ではない) 仕様から始め、段階的に複数の (この場合には 2 個の) プロセスに分散させていく方法で構築できることを理解しなくてはならない。この例は前の章よりも進歩したデータ構造——時間、関数、制約——を含んでいる。

5 章 (Event-B 記法) は Event-B 記法と証明義務規則のまとめである。矛盾なく定義された、それでいて簡潔な記法、なおかつそれが証明義務規則を通じて数学的に翻訳できることを学生たちに見せることが重要である。ただし、導入コースでは細部に深く立ち入る必要はない。

9 章 (数学言語) は例から離れる。そして、コースの中ほどで、数学的概念を再確認する。重要な点はここで学生たちが集合論的概念による証明に慣れることである。学生たちには集合論的構造を述語計算に変換する練習問題を多く与えるべきである。この章の全てを網羅する必要はない。

15 章 (逐次実行プログラム開発) は、学生たちがプログラムを書くことに慣れているならば、部分的に説明するとよい。プログラムを体系的に構築できることを理解させなくてはならない。また、ここで、形式的なプログラム構築 (ここでの手法) とプログラム検証 (開発が終わってからプログラムを「証明」する) の違いを理解しなくてはならない。一部の例は避けるべきである。具体的には、ポインターを扱う例。これは難しすぎる。

コースの終わりには、学生たちは抽象化と精義化という概念に慣れているはずである。また、簡単な数学的命題に対しては形式的証明を恐れないようになっていないといけない。最後に、学生たちは動くプログラムを開発することが可能であると確信しているはずである。

学生たちは Rodin プラットフォーム [7] を意識するかもしれない。しかし、私たちは、Rodin が何をしているか理解させるために、まず手動で証明をさせることにする。

上級コース

ここでは学生たちが導入コースを受けていることを前提とする。この場合、2 章と 3 章を繰り返す必要はない。しかしながら、これを読み返すことを勧めるとよい。このコースはその他の章をすべて含む。

同じ Event-B 手法を多様な実行パラダイム——逐次 (sequential)、分散 (distributed)、並列 (concurrent)、平行 (parallel)——のモデルシステムに使用できることを学生たちが理解することが重要である。

学生たちは複雑なデータ構造——リスト、木、DAG、任意のグラフ——の推論に慣れていなければならない。複雑なデータ構造を作るための集合論を理解しなければならない。このため、11 章 (木の同期プロセス)、12 章 (モバイルエージェント)、13 章 (IEEE プロトコル)、17 章 (鉄道システム) の例は全て重要である。

このコースでは、学生たちは手動の証明はしなくてもよい。かわりに Rodin プラットフォームなどのツール [9] を使わなければならない。

3 形式手法

形式手法という言葉は今日では***非常に大きな混乱*** *(great confusion)* を引き起こしている。なぜなら形式手法は多様な活動によって拡張されてきたからである。典型的な疑問は以下のようなものである: なぜ形式手法を使うのか？ 何に対して形式手法を使うのか？ どの時点で形式手法を使う必要があるのか？ UML は形式手法なのか？ オブジェクト指向プログラミングをする際に形式手法は必要なのか？ そして、形式手法をどのように定義するか？

私はこれらの質問に少しずつ答えよう。形式手法は (内部の) ***プログラム開発プロセス*** *(program development process)* が不適切であると認識している人のためにある。不適切さにはいくつかの理由——誤り、費用、リスク——がある。

形式手法を選ぶのは簡単なことではない。その理由は部分的には形式手法の販社が多数あることである。さらに正確に言えば、「形式」という言葉は何も意味しない。以下は読者が販社に尋ねるかもしれない質問である: あなたの形式手法の背景に理論がありますか？ あなたの形式手法はどんな言語を使いますか？ あなたの

形式手法には精義化メカニズムがありますか？ あなたの形式手法では推論の方法は何ですか？ あなたの形式手法ではあらゆるものを証明することができますか？

形式手法には本質的な困難があるから、それを使うことは不可能であると主張する人がいるかもしれない。以下は主張された困難のうちの一部である。数学者にならなくてはならない。提案されている形式手法は習得するには難しすぎる。視覚的でない (箱と矢印がない)。多くの人は証明ができない。

私はこれらの主張の大部分に賛成しないが、いくつかの本物の困難があることを認める。私の考えでは、その困難は以下である。

1. 決式手法を使うと、コーディングの前にたくさんのことを考える必要がある。現状はそうではない。
2. 形式手法を使うには、何らかの開発プロセスに組み入れる必要がある。これは簡単ではない。製造業では、製品を開発するとき、とても詳細なガイドラインのもとで、それにきわめて注意深く従う必要がある。たいていは、ガイドラインの導入が技術者に受け入れられ活用されるには長い時間がかかる。形式手法を組み入れるためにガイドラインを変更することは管理者にとっては気が進まない。なぜなら、プロセスの変更にかかる時間と費用が心配だからである。
3. モデルの構築は単純な行為ではない。というのも、これは読者がこの本で学んでいくことだからである。モデリングとプログラミングを混同しないよう注意が必要である。モデリングのかわりに擬似プログラミング (pseudo-programming) をしてしまうことは多い。詳しく言うと、初期モデルはプログラムが満たすべき性質を記述する。これはプログラムに含まれるアルゴリズムを記述することではなく、むしろ、完成したプログラムが正しいかどうかを最終的に判断するためのものである。例えば、ファイルソートプログラムの初期モデルはどのようにソートするかを説明しない。そのかわり、ソート後のファイルの性質と、ソート前とソート後のファイルの関係を説明する。
4. モデリングは推論を伴う必要がある。言い換えれば、プログラムのモデルは単なる文字の集まりではなく、何らかの形式に従って書かれる必要がある。さらに、その文字の集まりに関係した証明も含まれる。長年、形式手法は単に、私たちが作ろうとしているプログラムの抽象的な記述を得るために使われてきた。繰り返すが、記述だけでは十分ではない。私たちは、無矛盾性を証明することで、私たちの書くものを正当化しなくてはならない。ここで問題は、ソフトウェアの実践者はこのような証明を構築することに慣れておらず、他方では、他の技術的な分野の人たちははるかに証明に慣れていることである。さらに、ソフトウェア技術者の日常の実践に証明を加えることの難しさの一つは、証明を助ける良い証明ツールがないこと、特に大規模な問題に使えるものがないことである。
5. 最後に、モデリングの際に直面する重要な困難は、私たちが行おうとしているプログラミングの仕事に関連付けられた良い要求文書が、ほとんどの場合に存在しないことである。ほとんどの場合、産業分野で見ることができる要求文書は、存在しないに等しいか、あるいは、とんでもなく冗長すぎる。私の主張は、ほとんどの場合、あらゆるモデリングの前に要求文書を完全に書き直すことが絶対不可欠ということである。この点については後ほどまた触れる。

4 ちょっと寄り道: 青写真

大規模で複雑な計算機システムの開発に携わる人たちは、全ての成熟した工学分野で共有されている視点、具体的には***対象システムについて、その構築中に推論するために、ある成果物を利用する*** *(using an artifact to reason about their future system during its construction)* という視点を取り入れるべきである、というのが私の

信条である。成熟した工学分野では、広義の***青写真*** *(blue-print)* が使われている。青写真は、まさに構築プロセスで形式的に推論することを可能にする。ここに多数の成熟した工学分野を挙げる: 航空工学、土木工学、機械工学、鉄道制御システム、造船、など。これらの分野の人々は青写真を使い、また青写真を工学活動のとても重要な部分とみなしている。

ここでちょっと青写真が何であるか分析してみよう。青写真は、対象システムの、ある種の代替表現である。しかしながら、青写真は実物大模型 (mock-up) ではない、なぜなら現実のものではないからである: 自動車の青写真を運転することはできない！ 青写真は、作ろうとしている対象システムついて、構築プロセスで推論することを可能にする。

対象システムについて推論するとは、システムの動作と制約を定義することと計算することを意味する。また、建造物を段階的に構築することを可能にする。推論は、いくつかの専門的な基礎理論——素材の強度、流体数学、重力など——に基づいている。

私たちは多数の『青写真』技法を使うことができる。青写真を作るときには未定義の約束事が使われる。それは、推論を助けるだけでなく、大きな集団で青写真を共有することを可能にする。青写真は、たいていの場合、段階的に正確になっていく版の連続として計画される。同様に、青写真は、読みやすさのために、小さい部分に分割することができる。さらに、早期の青写真では決定せず、(より先の青写真で) 後ほど精義化する未決定の選択肢として残しておくこともできる。最後に、技術者が過去の仕事を再利用するために目を通すことができる古い青写真のライブラリーはとても興味深い。これらの全て (精義化、分割、再利用) のために、青写真を注意深く使うことが明確に必要である。そのためには、青写真の開発の全体が首尾一貫しなければならない。例えば、より詳細な青写真は以前の詳細でない青写真と矛盾してはならない。

ほとんどの場合、私たちのソフトウェア構築という工学分野は、青写真のような成果物は使われていない。その結果、最終製品のテスト工程はとても重くなり、よく知られているように、それでは遅すぎることが多々ある。私たちの分野で青写真を描くことは、対象システムの***モデルを作ること*** *(building models)* に相当する。プログラムのモデルはプログラムそのものでは決してない。しかし、プログラム——より一般的に言えば複雑なコンピューターシステム——のモデルは実行可能ではないけれども、対象システムの属性を確定し、その属性がモデルの中に存在することを証明できる。

5 要求文書

前節で簡単に説明した青写真は、開発プロセスの最初の工程ではない。その前に***要求文書*** *(requirement document)* を書く工程がある。ほとんどの場合、要求文書は欠陥があるか極めて下手に書かれている。この理由から、私たちはしばらくこの疑問について深く考察して、満足できる解答を得ることを試みる。

5.1 ライフサイクル

まず、私たちはこの活動 (activity)——具体的には要求文書を書くこと——がプログラム開発のライフサイクルの中でどのような位置を占めるかを思い出しておく。以下はライフサイクルにおける多様な工程のおおまかなリストである: システム分析 (system analysis)、***要求文書*** *(requirement document)*、技術仕様 (technical specification)、設計 (design)、実装 (implementation)、テスト (tests)、保守 (maintenance)。

それぞれの工程の簡単なまとめをしておく。システム分析工程は、構築しようとするシステムの、先行する実現可能性の調査である。要求文書工程は、システムの機能と制約を明確に言明する。ほとんどの場合、要求

文書は自然言語で書かれる。技術仕様は、前工程の文書を、何らかのモデル化技法を用いて構造化および定式化したものを含む。設計工程は、何を用いて前工程の仕様を実装するかを決定し、それを正当化する、ならびに、対象システムのアーキテクチャーを定義する。実装工程は、前工程の成果物をハードウェアおよびソフトウェアの部品に変換する。テスト工程は、最終的なシステムの実験による妥当性確認を含む。保守工程はシステムのアップグレードを含む。

前述のように、要求文書工程はこのライフサイクルの中でとても頻繁に**とてつもないウィークポイント** *(very weak point)* になっている。このことは後続の工程で多大な困難を引き起こす。特に、設計工程で発生する、仕様変更が不可避になるというよく知られた症候群は、要求文書の欠陥に起因する。要求文書が上手に書けていれば、この種の困難は消滅するはずである。この工程をどうやって改良するかを示すことがとても重要である理由はここにある。

5.2 要求文書の困難

良い要求文書を書くことは難しい仕事である。要求文書を読むのは次の工程——具体的には技術仕様と設計——を処理する人である。その人たちが要求文書を利用するのは、たいていの場合、とても難しい。なぜなら、何を、どのような順序で考えに入れるか識別することができないからである。

さらに、とても頻繁に、要求文書の何らかの重要な点が抜け落ちていることがある。私は航空機の警報システムの巨大な要求文書を見たことがある。そして、このシステムは誤報を配信してはならないという単純な (simple) 事実が単純に (simply) 抜け落ちていた。この要求文書の作者がこの欠陥について質問されたとき、その回答はさらに驚くべきものだった。そのような細かいことを要求文書に盛り込む必要はない、なぜなら、「システムが誤報を配信するべきではないことなど、当然、誰もが知っている」とのことである。対照的に、要求文書は多数の無関係な細かい項目で過剰仕様になっていることもある。

要求文書の読者は、テキストのどの部分が***説明*** *(explanation)* で、どの部分が本当の***要求*** *(requirement)* なのかを正確に区別することが難しい。説明は読者が最初に対象システムを理解するために必要である。しかし、読者がシステムの目的に精通すると、説明は重要ではなくなる。そのとき、頼りにするものは、構築しようとしているシステムの中の何を考えに入れるべきかを正確に知るために、本当の要求が何であるかを思い出すことである。

5.3 有益な比較

説明と、(ある意味では) 要求を含む別の文書 (もしくは本) を考えてみる。それは数学の本である。ここで「要求」とは定義と定理である。これらの要素は、ほとんどの場合、簡単に確認できる。なぜなら、機能 (定義、補題、定理) に従ってラベル付けされ、体系的に採番されており、また、本文とは別の書体で書かれている。例えば、

定理 1（カントール＝ベルンシュタインの定理） $a \preceq b$ **かつ** $b \preceq a$ **ならば** a **と** b **は濃度が等しい。**
この定理は 1895 年にカントールによって最初に予想され、1898 年にベルンシュタインによって証明された。
証明 $b \preceq a$ のとき a は $b \approx c$ を満たす部分集合 c を持つ。...

...

□

この例は数学の本から取った。私たちは最初の行に示されている「要求」——定理番号、定理の名前、定理の言明 (書体を変えて書かれている) ——をはっきりと見ることができる。関連する「説明」——歴史についてのコメントと証明——がこれに続く。

この区別は究極的に興味深く、読者にとって役に立つ。この文章に最初に出会うときには、説明は不可欠である。その後は、正確な言明が一目で分かることに興味を持つかもしれないが、歴史についてのコメントや証明には興味が薄れる。いくつかの数学の本は、「要求」すなわち定義と定理を付録としてまとめて本の後ろに付けている。これは調べ物に便利である。

5.4 要求文書の構造化

このような数学の本からの類推 (訳注: 5.3 節より続く) に従うと、要求文書を互いに埋め込まれた 2 種類のテキストで構成するという考えが得られる。2 種類のテキストとは、***説明テキスト*** *(explanatory text)* と***参照テキスト*** *(reference text)* である。これらの 2 種類のテキストは、参照テキストだけを独立して要約できるように、分離可能にするべきである。

ほとんどの場合、参照テキストは、自然言語で書かれた、***ラベルと番号を付けた短い言明*** *(labeled and numbered short statements)* である。こうすれば、参照テキストを説明テキストから独立して読むことがとても簡単である。この目的のため、私たちは参照テキストに特別の書体を用いる。これらの個々の項目 (参照テキスト) は説明なしで自己完結していなければならない。説明テキストは単に、文書を初めて読む人を補助するためのコメントである。2 回目以降は参照テキストのみに頼ればよい。

要求の個々の項目のラベルはとても重要である。ラベルはシステムによって違ってもよい。ただし共通のラベルは以下のようなものである:

FUN: 機能要求 (functional requirements)
ENV: 環境要求 (environment requirements)
SAF: 安全属性 (safety properties)
DEG: 限定モード要求 (degraded mode requirements)
DEL: 遅延についての要求 (requirements concerned with delays)
など

要求文書を書く前にすべき重要な活動は、使用すべきラベルを注意深く定義することである。また、要求に番号を付けることは、開発の後の段階になって要求を参照するために重要である。これは***トレーサビリティ*** *(traceability)* と呼ばれる。すなわち、このラベルと番号は後の段階 (技術仕様、設計、実装においても) で使われる。これにより、それぞれの要求が、システムの構築の過程で、あるいは最終的な稼動するバージョンで、本当に考えに入れられているかを簡単に確認できる。

ほとんどの場合、要求の個々の項目は短い言明で書かれる。しかし、データ定義表、遷移ダイアグラム、数式、物理ユニット表、図などの別の方法を使うこともある。

ただし、要求文書全体の構造はこの段階では重要でない。開発の後の工程になって注意すればよい。

6 「形式手法」という用語のこの本での定義

形式手法は私たちの分野に適応した青写真を作る技法である。そのような青写真は**形式モデル** *(formal model)* と呼ばれる。

本物の青写真と同じように、モデルを書く際に私たちはいくつかの**未定義の約束事** *(pre-defined conventions)* を用いる。新しい言語を発明する必要はない。私たちは**古典論理** *(classical logic)* と**集合論** *(set theory)* の言語を用いる。これらの決まりはモデルを他の人とやりとりすることを容易にする。なぜなら、何らかの数学的バックグラウンドを持っている人たちはこれらの言語を知っているからである。これらの数学的言語を使うことで、普段と同じように、数学的証明の形式で推論することが可能になる。

再び注意しておくが、青写真と同じように、モデルは現実のものではない。**モデルは一般に実行することができない** *(our model will thus not in general be executable)*。

私たちが開発の対象とするシステムの種類は**複雑** *(complex)* で**離散的** *(discrete)* なものである。この二つの概念をしばらく分析しよう。

6.1 複雑システム

ここでは以下のような疑問から始める。電子回路、ファイル転送プロトコル、航空機の座席予約システム、ソートプログラム、PC オペレーティングシステム、ネットワークルーティングプログラム、原子力発電所制御システム、スマートカード電子財布、ロケットの飛行制御装置、などに共通するものは何か？ それぞれ非常に違った規模と目的の**システム** *(system)* の、要求、仕様作成、設計、実装に関して深い分析と形式的な証明に関して統一されたアプローチは存在するか？

私たちはこの時点ではごく一般的な回答にとどめる。ほとんど全てのこのようなシステムは**複雑** *(complex)* である。すなわち、多数の部品からできている。また、変化が激しく、時には敵対的でもある環境と相互に影響し合う。複数の並列実行エージェントを伴うことも多い。また、高度に正しさが要求される。最後に、これらのうちのほとんどは技術者と専門家から成る大規模かつ有能なチームによる数年にわたる構築プロセスの結果である。

6.2 離散システム

これらのシステムの振る舞いは常に究極的に連続するにもかかわらず、前節で挙げたシステムは、ほとんどの場合、**離散的** *(discrete fashion)* に動作する。すなわち、変化のない状態と、突然の変化をもたらす飛躍とが交互に並んだモデルによって、これらのシステムの振る舞いは紛れもなく**抽象化** *(abstract)* できる。もちろん、可能な変化の数は莫大であり、さらに、思考不可能なほど高頻度に同時発生する。しかし、この数と頻度は問題の本質を変えることではない。すなわち、これらのシステムは本質的に離散的である。これらは**遷移系** *(transition system)* という総称名にまとめることができる。こう言ったからといって方法論が得られるわけではないが、少なくとも**共通の出発点** *(a common point of departure)* が得られる。

これまでに考えた例のうちのいくつかはプログラムそのものである。言い換えれば、それらの例での遷移は本質的に**単一の媒体** *(one medium)* に集中している。電子回路とソートプログラムは明らかにこの区分にあたる。しかしながら、他のほとんどの例は単にプログラムそのものよりもはるかに複雑である。なぜなら、それらの例は多数の異なる実行主体と関わり、さらに環境と深く影響し合うからである。これは同時に動作する異

種の実体が遷移を引き起こすことを意味する。しかし、繰り返すが、このことは問題の離散的な本質を変えるものではなく、単に問題を複雑にするだけのことである。

6.3 テスト推論 VS モデル (青写真) 推論

このような複雑離散システムの構築を考えるとき、とても重要な——少なくとも時間とお金に関して——活動は、最終的な実装が、言ってみれば、***正しく*** *(correct)* 動作していることを検証することである。昨今はほとんどの場合、この活動は、「研究所実行」(laboratory execution) とも呼ぶべき、とても重いテスト工程として実現されている。

このような「研究所実行」による離散システムの妥当性を確認することは、仮に不可能でないとしたならば、複数媒体の場合には単数媒体の場合よりもはるかに複雑である。そして、私たちが既に知っているようにプログラムのテストは極めて不完全なプロセスである。すなわち、プログラムのテストはほとんど全てのプログラム製作のプロジェクトで妥当性を確認するためのプロセスとして行われているが、これは極めて不完全である。実行される場合分けを全て網羅することは不可能であるが、実際のところ、それは本当の原因ではない。不完全性はむしろ、私たちにとって、ほとんどの場合、***ご託宣の欠落*** *(lack of oracles)* の結果である。「ご託宣」があれば、テストの段階で得られるはずの期待される結果が***事前に*** *(beforehand)* かつテスト対象とは独立に得られる。

にもかかわらず、現在、優秀な人々から成るとても小さな設計チームが、複雑システムの構築の中心である。その際、その小さな設計チームが、実装担当者という軍隊を管理しており、結局は長く重いテスト工程が構築プロセスに含まれてしまうというのが依然として実情である。そして、よく知られたことであるが、テストの費用は純粋な開発成果の場合の少なくとも 2 倍である。これは現在でも合理的なやり方なのだろうか？ 私たちの意見は、このような手法を用いる技術というのは、いまだに幼児期にあるということである。これは、いくつかの技術にとっては前世紀の初めには問題であったことではあるが、現在ではもっと成熟した段階に達している (例: 航空工学)。

この短い説明で対象としている技術は***複雑離散システム*** *(complex discrete system)* に関するものである。妥当性確認の主要な方法がテストである限り、私たちはこの技術は発展途上の状態にあるとみなす。テストはいかなる種類のよく考えられた推論も含んでいない。むしろ、仕様作成工程と設計工程を通じて***深く考えることを常に先送りする*** *(always postponing any serious thinking)* ことから成っている。システムの構築は、常に、テストの結果によって再適応、再形成される (これを試行錯誤という)。しかし、知られているように、ほとんどの場合これでは遅すぎる。

結論として、テストは、構築中のシステムの直近の操作上の状況を教えてくれる。すなわちそれは稼動状況である。別の技術分野では、再び航空工学を例に取ると、人々は最終的に構築中のもののテストをするけれども、テストは練られた設計プロセスの不可欠な工程であるというよりも、むしろ単に***通常の確認*** *(routine confirmation)* であるに過ぎない。実際のところ、ほとんどの推論は最終的な対象を構築する***前に*** *(before)* 行われている。この推論は多様な青写真に、明確に定義された実践的な理論を適用することで行われる。

この本の目的は、このような「青写真」手法を複雑離散システムの設計に取り入れることである。また、このような青写真の何らかの***証明された推論*** *(proved reasoning)* を作り上げるのを容易にする理論を示すことも目的とする。このような推論は、したがって、最終的な構築物よりもはるか以前に位置する。これまでの文脈では、「青写真」は***離散モデル*** *(discrete model)* である。私たちは、ここで、離散モデルという概念の非公式の要約を示すべきである。

7　離散モデルの非公式の要約

この節では私たちは離散モデルを非公式に述べる。離散モデルは一つの状態と複数の遷移から成る。理解のため、私たちは離散モデルの操作的な解釈を示す。続いて、私たちが表現したい形式的な推論の種類を示す。最後に、私たちは 3 個の概念――精義化 (refinement)、分割 (decomposition)、汎用的開発 (generic development)――を用いてモデルの複雑さを制御するための問題を簡単に示す。

7.1　状態と遷移

おおまかに言って、離散モデルは***状態*** *(state)* から作られる。状態は、いくつかの定数と変数で表現される。定数と変数は、調査対象となる実際のシステムの、特定のレベルの抽象化で現れるものである。このとき、変数は、自然のシステムを研究する応用科学――物理学、生物学、オペレーショナル・リサーチ――で使われているものと同じである。これらの科学でもモデルが使われている。モデルは、何らかの推論をモデルに適用することで、現実の法則を推論することを助ける。

状態とならんで、モデルは、特定の状況で発生する複数の***遷移*** *(transition)* を含む。このような遷移をここでは「イベント」(event) と呼ぶ。それぞれのイベントは、まず、***限定条件*** *(guard)* からできている。限定条件は状態の定数と変数についての述語を用いている。限定条件はイベントが発生する***必要条件*** *(necessary condition)* を表現する。それぞれのイベントは、また、***動作*** *(action)* からできている。動作は、特定の状態変数がイベントの発生の結果として変更される方法を記述する。

7.2　操作的な解釈

見て分かるように、離散動的モデルはある種の状態遷移機械を構成する。私たちはこの機械に極めて単純な***操作的な解釈*** *(operational interpretation)* を与える。このような解釈は私たちのモデルに対して操作的意味論 (operational semantics) を与えるためではなく (操作的意味論は後に証明系を用いて与えられる)、単に***非公式の理解*** *(informal understanding)* を助けるために与えられることを注意すべきである。

まず第一に、イベントの実行には***時間がかからない*** *(no time)* と考える。ここで、イベントの実行とは、観測可能な状態変数の遷移である。次に、2 個のイベントが同時に起こることはない。続いて、実行は以下のようになる。

- 限定条件が真になるイベントがなければ、モデルの実行は停止する。***これはデッドロックと呼ばれる*** *(it is said to have deadlocked)*。
- 限定条件が真になるイベントがあれば、それらのイベントのうちいずれかが必ず発生し、それによって状態が変更され、続いて、再び限定条件がチェックされ、以下続く。

この振る舞いは複数の限定条件が同時に真であるとき非決定性を示している。この非決定性は特に外的非決定性と呼ばれる。限定条件が真であるイベントのうち、どのイベントが実際に実行されるか、私たちは***前提を持たない*** *(no assumption)*。いつでも 1 個だけの限定条件が真であるとき、モデルは決定的であると言える。

モデルがいつか停止することは***必須ではない*** *(not at all mandatory)* ことを注意しておく。実際のところ、私たちが学ぶほとんどのシステムはデッドロックしない。つまり、永久に動き続ける。

7.3 形式的な推論

前節で説明したごく基本的な遷移機械は、素朴なものではあるが、興味深い形式的な推論を可能にするだけの十分に凝ったものである。以下、2 種類の離散モデルの性質を示す。

私たちがモデルについて証明しようとしている、すなわち究極的には実際のシステムについて証明しようとしている第 1 種の性質は、言ってみれば、***不変の性質*** *(invariant properties)* である。不変式は、状態変数に対する、常に保たれなくてはならない条件である。これを実現するためには、問題の不変式と、各イベントの限定条件のもとで、そのイベントに関連付けられている動作によって変更された後も不変式が保たれていることを***証明*** *(prove)* すればよい。

私たちは、不変式とは対照的に、常に保たれるわけではない条件に関わる、もっと複雑な形の推論についても考えなくてはならない。対応する言明は***様相*** *(modalities)* と呼ばれる。私たちの手法では、***到達性*** *(reachability)* と呼ばれる極めて特殊な形の様相について考える。私たちが証明したいのは、現時点で限定条件が真でないことがあり得るイベントは、特定の有限時間の間に確かに発生するということである。

7.4 閉じたモデルの複雑さの管理

私たちが作るモデルは、私たちが意図するシステムの制御部のみを記述するのではないことを注意しておく。モデルは環境の代替表現を含む。ここで環境とは、私たちが作るシステムが動作すると想定される場所である。実際のところ、私たちは多くの場合、本質的に、特定の環境と対応するコントローラーとの動作と応答を表現できる***閉じたモデル*** *(closed model)* を作る。コントローラーは分散されている場合もある。

閉じたモデルを作るにあたって、コントローラーのモデルを抽象化された環境の中に挿入できるようにするべきである。ここで、環境は、別のモデルであるかのように定式化されている。このような閉じたシステムの状態は、従って、環境の状態を記述する物理変数と、コントローラーの状態を記述する論理変数を含む。同じように、遷移は、環境に関わるものとコントローラーに関わるものの 2 種類のグループに分けられる。私たちはまた、2 個のものがやりとりする方法をモデルに組み入れなければならない。

しかし、先に注意したように、実際のシステムの遷移の数は間違いなく巨大である。そして、言うまでもないが、そのシステムの状態を記述する変数の数も、とんでもなく大きい。このような複雑さに現実的に対処するにはどうしたらよいか？ この問いへの答えは 3 個の概念——***精義化*** *(refinement)* (7.5 節)、***分割*** *(decomposition)* (7.6 節)、そして***汎用的なものを具体化すること*** *(generic instantiation)* (7.7 節)——の中にある。ここでの重要な注意点として、これらの 3 個の概念は互いに関連していることを挙げておく。実際のところ、私たちはモデルを精義化し、それは後ほど分割するためである。そして、より重要なことは、モデルを分割するのは、後に、より自由に精義化するためである。そして、最後に、汎用的なモデルを開発することにより、後で具体的なシステムを作ることができるようになる。これによって、同じような証明を繰り返すことを防ぐ。

7.5 精義化

精義化はモデルを***徐々に*** *(gradually)*、すなわち段々に精密にかつ現実に近くなっていくように作ることを可能にする。言い換えると、私たちは、現実を平板に表現する一つのモデルを作るのではない。平板な方法は、状態の大きさと遷移の数のために、明らかに不可能である。さらに、結果として得られるモデルは、ただ読む

ためのものでない限り理解するのはとても難しい。私たちは、組込みモデルの順序のある系列を作る。この系列のそれぞれのモデルは、系列の前方のモデルの精義化である。すなわち、精義化された、より具体的なモデルは、抽象的なモデルよりも多くの変数を持つ。このような新しい変数は、私たちのシステムをより細かい解像度で見ることで見えるようになる。

ここでは顕微鏡を使う科学者のことが有益な類推である。顕微鏡を使うとき、実物は同じであり、顕微鏡が実物を変化させることはなく、***見た目が精細になる*** *(our look at it is only more accurate)* だけである。それまで見えなかったものが顕微鏡によって見えるようになる。より強力な顕微鏡を使えば、よりたくさんの部品が見えるようになる。精義化されたモデルは、このようにして、以前の抽象モデルよりも空間的に大きい。

この***空間的広がり*** *(spatial extension)* との類推で言えば、対応する***時間的広がり*** *(temporal extension)* が存在する。新しい変数が変更されるのは、以前の抽象モデルには存在しなかったイベントのみである。なぜなら、単に、対応する変数が以前のモデルには存在しなかったからである。実際には、このことは、新しい変数のみと関係する***新しいイベント*** *(new events)* を用いて実現される。このような新しいイベントは、抽象モデルでの暗黙の何もしないイベントを精義化したものである。このようにして、精義化は、私たちの現実を***より時間的な粒度を細かくして*** *(finer time granularity)*、離散的に観測した結果である。

精義化はまた、何らかのプログラミング言語を用いてコンピューター上に実装できるように状態を作り直すためにも使われる。この精義化の 2 番目の使用法は***データ精義化*** *(data-refinement)* と呼ばれる。これは、全ての重要な性質がモデル化された後で、2 番目の技術として使われる。

7.6 分割

精義化は複雑さの問題を完全に解決するわけではない。モデルが精義化されていくに従って、状態変数と遷移の数は増大し、全部を取り扱うのが不可能になる。この理由から、私たちの 1 個の精義化されたモデルを、複数のほぼ独立した部分に切り分ける必要がある。

分割はまさに 1 個のモデルを複数のコンポーネントモデルに分解するシステマティックなプロセスである。このようにして、私たちはそれぞれのコンポーネントをその他とは独立して研究し、精義化する。それにより、私たちは全体の複雑さを減少させる。このような分割の定義は以下のことを意味する。まず、それぞれのコンポーネントモデルを独立して精義化したモデルは、常に、1 個のモデルの形に再び寄せ集めることができる。しかも、それはもとの 1 個のモデルを精義化したものと等しいことが保証されている。この分割プロセスはさらにコンポーネントに対して適用でき、それを繰り返すことができる。コンポーネントモデルは既に存在し開発されていてもよく、それによって、トップダウン手法とボトムアップ手法を組み合わせることが可能になることを注意しておく。

7.7 汎用的開発

精義化と分割の適用によるいずれのモデル開発も、複数の属性によって定義されるキャリアセット (carrier sets) と定数でパラメータ化される。

このような汎用的なモデルは、数学の理論と同じ方法で、別の開発に具体化できる。例えば集合論は、より特殊な数学理論に具体化できる。もし抽象理論の公理が第 2 の理論の定理に過ぎないことを証明できたならば、これは可能である。

この汎用的な具体化の手法の興味深い点は、抽象的な開発で既になされている証明を繰り返すことを避けら

れることである。

参考文献

[1] J.R. Abrial. *The B-book: Assigning Programs to Meanings.* CUP 1996

[2] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley 1999.

[3] F. Badeau. *Using B as a high level programming language in an industrial project: Roissy val.* In Proceedings of ZB ’05, 2005.

[4] P. Behm. *Meteor: A successful application of B in a large project.* In Proceedings of FM ’99, 1999.

[5] K.M. Chandy and J. Misra. *Parallel Program Design, a Foundation.* Addison-Wesley, 1988.

[6] R.J. Back and R. Kurki-Suonio. *Distributed Cooperation with Action Systems. ACM Transaction on Program-ming Languages and Systems.* 10(4): 513-554, 1988.

[7] Rodin. *European Project Rodin.* `http://rodin.cs.ncl.ac.uk.`

[8] J.F. Groote and J.C. Van de Pol *A bounded retransmission protocol for large data packets - a case study in computer checked algebraic verification.* Algebraic Methodology and Software Technology, 5th International Conference AMAST ’96, Munich. Lecture Notes in Computer Science 1101.

[9] G. Le Lann. *Distributed systems - towards a formal approach.* In B Gilchrist, editor *Information Processing 77* North-Holland 1977.

[10] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers 1996.

[11] W.H.J. Feijen and A.J.M. van Gasteren. *On a Method of Multi-programming* Springer 1999.

[12] L. Moreau. *Distributed Directory Service and Message Routers for Mobile Agent.* Science of Computer Programming 39(2-3):249-272, 2001.

[13] IEEE *Standard for a High Performance Serial Bus.* Std 1394-1995, August 1995.

[14] H.R. Simpson. *Four-slot Fully Asynchronous Communication Mechanism.* Computer and Digital Techniques. IEE Proceedings. Vol 137 (1) (Jan 1990).

[15] C.B. Jones. *Systematic Software Development using VDM.* Prentice Hall International (1990).