Vol.2017-EMB-45 No.1
2017/6/30

形式手法通論カリキュラム：立案と実施の経験

中島 震[1,2,a)]

概要： 大学院修士課程相当向け形式手法コースのカリキュラム立案と講義実施経験を報告する．形式手法の全体像として何を学ぶべきかは標準的な内容が整理されていない．そこで，試行錯誤で教材を開発した．一方，2009 年に最初の講義を行って以来，形式手法を取り巻く環境も大きく変化している．研究者との議論や受講生の意見を参考に，本コースのカリキュラムを改訂してきた．特徴的な教材の紹介を含めてコースの概要を紹介し，同様なカリキュラム立案の参考とすべく，当初の内容から最新版（2016 年）への変更点と理由を説明する．

キーワード：モデル規範，状態ベース仕様，リファインメント，モデル検査，Alloy，enPit.

1. はじめに

少し時代を遡る．大学に入学して出会う学問分野には，「通論」と付された厳めしい教科書があった．ある専門領域を全般にわたって論じた書．通論を英訳すると an introduction to らしい．「入門」というよりは「導入」というニュアンス．「通論」で良いだろう．

形式手法（Formal Methods）は，ソフトウェア・ディペンダビリティ達成への重要な技術アプローチ．ソフトウェア工学の中で確固とした位置を占める．形式手法自身，1970 年代初頭からの歴史があり，半世紀に届く年月を重ねて来た．そんな分野であるからには，通論があってしかるべきだろう．そう思って探してはみるが，ピッタリ感がない．まだ通論を書けるほど成熟していないのだろうか．ところが，学会では，産業界での実用的な適用事例とか，あるいは，新しい特徴を持つソフトウェア [10] への対応といった発表が増えている．対象が広がれば，形式手法も変わっていくのは事実だろう．一方で，形式手法の基本技術は既に安定した段階に達したようにも思える．

形式手法の分野を覗くと，多くの手法が乱立していることに気づく [2]．欧米では，各々が独立した学派（Schools）を形成しているよう．学会でも，学派ごとに棲み分けがみられた．最近は，異なる手法の統合化や交流が進み，分類学すら難しくなっている．他方，仲間内の小規模なワークショップが新たに誕生する．切磋琢磨といえば聞こえは良いが，相変わらずの乱立状態が続く．呉越同舟に近いか．特定手法の「入門」教科書はあっても，「形式手法通論」は見当たらない．

そんな折り，大学院で形式手法の授業を担当することになった．1 学期 14 あるいは 15 回で，形式手法の大まかな像を描きたい．産業界との共同作業の経験などから，「此れだけは押さえておきたい」範囲を考えることになった．本報告は，そんな事情から作成したカリキュラムに関する．通論の作成，実際は一筋縄ではいかない．計画初期の 2009 年時点と現在では，３分の１くらいは変わってしまった．逆に，残りは安定した内容とも言える．あるいは，当初の内容が不備で，現在の形が，あるべき姿かもしれない．本稿では，当初の計画，現在の形，変更になった理由などをまとめる．同様なカリキュラムを立案する際に参考になると幸いである．

2. 背景

2.1 ソフトウェア開発と形式手法

欠陥のないプログラムを開発する技術が必要なことは言うまでもない．1960 年代にソフトウェア工学（Software Engineering）が明確に意識され，ソフトウェア開発に関わるさまざまな技術開発が進められた．以下，形式手法の発端・発展の経緯 [2][3][8] を簡単にまとめる．

形式手法（Formal Methods）は，プログラム構築の科学的な基礎として数理論理（Mathematical Logic）を採用する方法の総称である．プログラムを記述するプログラミング言語を厳密に定義することからはじまった．次に，段階的な詳細化（Step-wise Refinement）の考え方にしたがっ

[1] 国立情報学研究所
NII, Hitotsubashi, Chiyoda-Ku, Tokyo 101–8430, Japan
[2] 総合研究大学院大学
SOKENDAI
[a)] nkjm@nii.ac.jp

て，仕様からプログラムを作成する過程に着目する．詳細化の各ステップが正しいことを数理論理学の方法で証明すると共に，対象ソフトウェア（Software Artifacts）を具体化してプログラムを構築する．

この「構築からの正しさ（Correct by Construction）」と呼ばれる考え方は，E.W. Dijkstra が 1972 年に行ったチューリング賞受賞講演で述べた構想にはじまる．プログラムを対象とするテスティング（Software Testing）では欠陥がないことを示すことができない．また，開発済みの膨大なプログラムを検査する「事後検査（A Posteriori Checking）」は不可能に近い．証明（Proof）とプログラムの同時作成によって欠陥の混入を防ぐという系統的な開発法の研究が必要なことを述べた．

その後，C.A.R. Hoare は 1981 年のチューリング賞受賞講演で，「設計を単純にして，あきらかな不具合をなくす．あるいは，複雑さを放置して，不具合があきらかでないようにする」と述べた．ソフトウェア開発上流工程で作成する「設計（Design）」の重要性を強調したもの．このように，形式手法は，開発上流工程で作成する設計記述を対象として欠陥混入を防ぐことで，開発プログラムの信頼性を向上する技術といえる．

同じ頃，1980 年代の初め，E.M. Clarke たちは，並行システムの設計仕様を対象とした自動検証の方法を考案した．ロジック・モデル検査（Logic Model-Checking）のはじまり．VLSI 回路，通信プロトコル，といった設計対象の欠陥除去に役立つことが示された．この成功は，検査対象の数理論理的な表現力を制限することで自動検証アルゴリズムを得たことにある．検査が万能というわけではない．しかし，形式手法を実践する際の大きな障害となっていた「正しさの証明」を自動化できる．その後，モデル検査は産業界にも大きな影響を与えてきた．E.M. Clarke を含む 3 名の研究者は，モデル検査法に関わる貢献が認められて，2007 年にチューリング賞を受賞した．

20 世紀，ソフトウェア工学は，オブジェクト指向概念を基盤とするソフトウェア技術によって，開発の生産性向上に大きく寄与した．21 世紀になると，ソフトウェアなくして社会基盤を構築することができない時代が到来している．従来にも増して，プログラムの欠陥に起因する不具合の社会的な影響が大きい．形式手法は，高い信頼性を達成する技術として，ソフトウェア・ディペンダビリティ（Dependability of Software）の基本技術になっている．

2011 年に，M. Shaw は「ソフトウェア工学教育の行く末（Whither Software Engineering Education?）」という講演の中で，「ソフトウェア工学はコンピューティング科学に基づき，ソフトウェア開発のさまざまな側面を取り扱う体系」であり，「ソフトウェアは複雑さに支配される」と述べた．形式手法は，プログラム構築に対する数理論理的な方法で，設計の複雑さを克服する技術体系といえる．ソフトウェア工学の中で，どのようなカリキュラムにすべきか，重要なテーマである．

2.2 教材の指針

形式手法に関わる既存教材[*1]を調べると，次のように，2 つに大別できることがわかる [7].

第 1 に，数理論理からのアプローチ．プログラム構築に関わる数理論理を学ぶことを目標とする．論理の基本的な方法，さまざまな論理系の基礎，を学び，その最後に，「応用として」形式手法に触れる．代表例として，集合論と 1 階述語論理によって設計物の表記法を与える Z 記法（Z Notation）が説明される．理論コンピュータ科学に重きをおく方法であるが，ソフトウェア工学の道具としての形式手法の教材として幾つかの欠点がある．紙とペン（Paper and Pencil）であり，学ぶ際に，理解したか否かのフィードバックを得にくい．数理論理という形式科学と形式手法が対象とするソフトウェアの違いが曖昧になる．

第 2 に，特定の形式手法を学習するアプローチ．数多ある形式手法から最善と思われるものを選び，その形式手法を具体的に説明する．当該手法だけを学ぶには適しているが，その他の方法（Alternative Methods）までカバーすることが難しい．たとえば，形式仕様作成を目的とする手法の教材では，自動検証の方法であるモデル検査に触れることは少ない．特定手法の実作業から学ぶ（Learning by Doing）スタイルの教材によって，実践経験を得る方法が効果的であるとも言われている．

素朴には，上記の 2 つを組み合わせれば良いかもしれない．つまり，数理論理アプローチの後に，特定の形式手法を学習する．この組合せ法は，学習時間が長くなるということ以外に，その他の方法が相変わらずカバーできないという欠点を持つ．

ソフトウェア開発方法論は一般論を論じるもの．特定の開発プロジェクトで実践する際には，一般的な開発方法論をカスタマイズして必要な技術を活用する能力が必要とされる．たとえば，UML は主な設計図式だけでも 7 種類を含む．実適用に際しては，開発対象が持つ性質，開発プロジェクトの条件などから，使用する設計図式を選択する．これを可能とするには，プロジェクトの実情に加えて，各々の設計図式の特徴に習熟していることが望ましい．仮に，知っているのがクラス図だけだとすると，他の図式を活用することを思いつかない．つまり，開発方法論の実践には，幅広く関連技術に精通し，カスタマイズする能力が必須である．

形式手法は開発上流工程で活用される技術であるという点で，ソフトウェア開発方法論の実践に必要とされる知識・能力と共通性がある．表現力が高く整合性のある記述を得

[*1] 多くは大学院修士課程が対象と思われる．

表 1 Curriculum Early Years

1	導入・形式手法の発展	1 回
2	Alloy 入門	2 回
3	状態ベース仕様	2 回
4	リファインメント	3 回
5	オブジェクト指向設計	4 回
6	プログラム検証	2 回

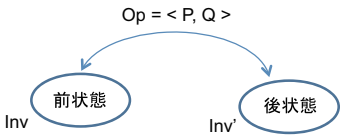

図 1 状態ベース仕様

ること，ひとつの側面に限定するが自動検証により振舞いを確認すること，段階的な詳細化でプログラム導出を行うこと，などの目的に応じて，特徴の異なる複数の形式手法を使い分ける．つまり，さまざまな手法を知っていることが大切である．

以上から，カリキュラム作成に関して，次のような方針をたてた [5]．(1) 個々の違いよりは，共通する考え方に注目する．個々の形式手法を，基本的な考え方の変奏 (Variations) と理解．(2) 概念を説明する「メタ言語 (Meta-Language)」として軽量形式手法 (Light-weight Formal Methods) の Alloy [1] を採用する．ツールによるフィードバックを通して概念を理解．(3) 少数の共通例題を用いて特徴の異なる形式手法を説明する．形式仕様記述と自動検証で同一の例題を用いる等．次節で，内容を紹介する．

3. カリキュラム

3.1 初期の内容

表 1 に初期のカリキュラムを示した．この頃は，モデル規範 (Model-Oriented) と呼ばれる形式手法 (VDM，Z 記法，B メソッド，Event-B，Alloy，等) およびオブジェクト指向デザインやプログラム検証を Alloy で説明することが中心だった．

項番 1 は形式手法の発展を概観するもので，年度によって細かな違いがあるものの，「お話」が中心．項番 2 は Alloy の簡単な紹介．説明の例題として，デザインパターンの代表例である `Composite Pattern`[*2]，モデル規範形式手法の入門で取り上げられる「誕生日帳 (`Birthday Book`)」，抽象データ型の代表例 `LIFO Stack` を用いた．`Alloy` 言語仕様やツール使用法の詳しい説明は省く．文献 [1] を参照すればわかる．しかし，`Alloy` が用いる有限スコープ解析の方法ならびに限界[*3]には言及する．特に，有限探索の限界から，見かけの不具合 (`Spurious Alarms`) が発生することを説明する．

[*2] 性質表現に transient closure を用いる．
[*3] 文献 [1] の第 5 章．

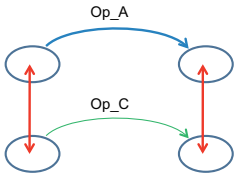

図 2 模倣関係

項番 3 はモデル規範形式手法が採用している仕様の書き方である「状態ベース仕様 (`State-based Specification`)」を紹介する．プログラミング言語の手続きをオペレーション (`Operations`) とし，これを前状態 (`Pre-states`) と後状態 (`Post-states`) をつなぐ関係 (`Relations`) として論理式で書き表す (図 1)．状態は複合的なデータ構造であり，不変量 (`Invariants`) を伴う．オペレーションが規定する機能仕様の正しさを判定する基準を証明条件 (`Proof Obligation`) として与える．形式手法ごとに証明条件の考え方が異なることを学習する．とりわけ，前状態に関わる論理式が事前条件 (`Pre-Conditions`) であるかガード条件[*4] (`Guard Conditions`) であるかに注意する．

項番 4 は状態ベース仕様を対象とするリファインメント (`Refinement`) を導入する．素朴には，段階的な詳細化の数理論理的な形式化．模倣関係 (`Simulation Relations`) で定義する (図 2)．理論的には，いくつかの種類の模倣関係が知られており，いずれも健全 (`Sound`) であるが完全 (`Complete`) ではない．つまり，ある模倣関係を使う場合，正しい模倣関係にも関わらず，正しいと証明できないことがある．その中で，前向き模倣と後向き模倣の 2 つがあれば良いことが知られている．

リファインメントを基本機構として持つ形式手法の `VDM`，`B` メソッド，`Event-B` は，前向き模倣に基づく．`Z` 記法や `Alloy` では，自身の記述の中で証明条件を表現することで，さまざまな模倣関係を調べることができる．項番 3 と同様，リファインメントの検査を行う証明条件は，形式手法の特徴を与える重要な要素である．

リファインメントの基本は段階的な詳細化．抽象的な仕様記述からプログラミング言語要素に具体化していくので，垂直リファインメント (`Vertical Refinement`) と言われる．一方，同じ抽象レベルのままで，新たな機能振舞いを追加する過程を模倣関係で定義することもできる．先の垂直との対比で，水平リファインメント (`Horizontal Refinement`) と呼ぶ．また，追加的であることを強調して上書きリファインメント (`Superposition Refinement`) とも言う．なお，この水平リファインメントはガード条件を採用している形式手法 (たとえば `Event-B`) で使える技法である．この違いは，リファインメントの証明条件として現れる．

状態ベース仕様のリファインメントには，上記の開発方

[*4] 発火可能条件 (Enabling Conditions) とも呼ぶ．

法と異なるもうひとつの見方がある．一般に，機能検証は，対象仕様と要求性質の２つの記述の突き合わせ検査で行う．一方，状態ベース仕様は検査対象であることから，要求仕様を表現する何らかの仕組みが別途必要．要求仕様が，すべての状態で成り立つ不変量（Invariants）として表現できれば，不変量保存（Invariant Preservation）の証明条件を用いて検査できる．ところが，不変量で表すことが難しい要求性質もある．このような場合，要求性質を上位仕様記述とし下位仕様記述を検査対象記述とみなす．リファインメントの証明条件を用いて，検査対象が要求仕様を「模倣」することを確認する．つまり，状態ベース仕様では，形式検証の方法として，リファインメントを切り離して考えることはできない．

項番 5 はオブジェクト指向設計で用いる図式を厳密に書き表す方法を学習する．図式表現の曖昧さあるいは多義性が古くから指摘され，形式手法を導入する研究が多数あった．たとえば，1990 年代の pUML（precise UML）という研究活動など．項番 5 では，これらの研究を踏まえて，UML と形式手法の関わりを２つに絞って論じた．ひとつめは，クラス図と OCL の組合せ．OCL はテキスト形式の設計言語であり，あるクラスに属するインスタンス・オブジェクト全体を参照する全称記号などを提供する．また，OCL オペレーションは事前・事後条件の組でメソッドの機能仕様を表す．このように OCL は状態ベース仕様スタイルに基づく形式手法と似た表現形式を持つ．一方，OCL は，操作的な実行意味定義を採用する．実際，最近のツールでは，OCL を変換した後の Java プログラムが意味を与えるという方法が主流になっている．したがって，OCL は形式手法ではない．余談であるが，Alloy は，UML の標準化活動に，OCL に代わる形式仕様言語として提案された．結果として，Alloy を採用しなかった．

UML の中で，ステート図は，操作的な意味が標準化文書の中で与えられているという点で特徴的な図式．そもそも，OMT が D．Harel の Statechart をオブジェクト指向分析設計図式として持ち込んだ．Statechart は Statemate 意味論と呼ばれる操作的な意味が与えられていたが，その後，多数の変奏が提案された．UML ステート図は，SynchCharts 意味論に基づくとされ，並行性と同報通信を特徴とする．項番 5 の教材では，P．Zave と M．Jackson の方法によって，ステート図の表層的な定義を論理式で，つまり Alloy で書き表すことで，整合性の検査が可能なことを説明する．また，状態遷移マシンの振舞い仕様を表現する命題時相論理が，1 階述語論理の部分系になっていることを学ぶ．これは，動的モデルを静的な数理論理の枠内で書き表す方法の例になる．

項番 6 は手続き的なプログラミング言語の意味を数理論理の枠組みで表す方法を学習する．ホーア論理ならびにダイクストラの最弱事前条件の方法を簡単な while 言語の例

表 2 Curriculum y2016

1	導入・形式手法の発展	1 回
2	Alloy 入門	2 回
3	状態ベース仕様	4 回
4	ロジック・モデル検査	4 回
5	プログラム検証とテスト自動生成	2 回
6	モデリング	2 回

を説明する．B メソッドの証明条件は，最弱事前条件の方法をもとにしている．この方法によって，VDM や Event-B に比べて，B メソッドの証明条件の論理式が簡単になる．その他，契約としての設計（Design by Contract）に基づく検証ツール ESC/Java もダイクストラの方法を採用している．

教科書 [6] では，項番 5 と項番 6 を再考した．項番 5 に関しては，モデル検査ツールとして SPIN を用いた例を紹介する一方．有界モデル検査（Bounded Model Checking，BMC）の仕組みを Alloy で示した．項番 6 に関しては，プログラム検証についてホーア論理と最弱事前条件の方法に加えて，手続き型プログラミング言語の有界モデル検査を説明した．また，テスト入力データ自動生成の話題を含める．特に，有界モデル検査と仕様に基づくテスティング（Specification-based Testing，SBT）は，SAT ツールの興味深い２つの使い方であることを説明した．Alloy 自身も SAT ソルバーを用いており，BMC も SBT も Alloy 上で簡単に実験することができる．

2012 年より始めた東工大での授業[*5]は，教科書 [6] に準じた内容が基本だったが，2013 年からは表 1 の項番 3，項番 4，項番 5 を再編して，表 2 の項番 3，項番 4，ならびに項番 6 の一部に置き換えた．

3.2 2016 年のカリキュラム

表 2 は 2015 年総研大および東工大講義での改訂をベースに 2016 年東工大で実施した最終形である．表 1 と比較すると，項番 1 と項番 2 は，ほぼ同じ，項番 3 は状態ベース仕様の中でリファインメントを説明するように変更した．リファインメントについては，後向き模倣の説明を省いた．VDM，B メソッド，Event-B といった具体的な形式手法が後向き模倣に基づくリファインメントを想定していないことが省いた理由である．限られた時間の中では，垂直リファインメントと水平リファインメントといった使い方の違いに言及するほうが有用と判断した．

項番 5 は教科書 [6] の内容に合わせた．前節の終わりに述べたように，BMC も SBT も SAT の興味深い応用になっていることの学習が有用であると判断したことによる．SAT/SMT ツールが容易に利用可能な状況になってきていることを考えると，ソフトウェアを対象とする検査問題を SAT/SMT で扱えることは知っておくべきことである．以

[*5] enPit（http://www.enpit.jp）の一貫．

下，大きく変更した項番 4 と項番 6 について説明する．

項番 4 のロジック・モデル検査は 4 回で説明する．既に 2013 年からモデル検査ツールとして SPIN を紹介していたが，表 2 では SPIN/Promela の教材も用いた．

SPIN の簡単な例を用いて，状態空間探索による自動検証の方法としてのモデル検査アルゴリズムを説明する．次に，状態爆発を避ける抽象化の方法を導入する．述語抽象で用いる抽象遷移の計算を Alloy で行う．第 3 に，有界モデル検査の方法を Alloy で行う方法を説明する．第 4 に，時間オートマトンのモデル検査法を説明する．リージョン・オートマトンを構成して，通常の離散遷移にする方法と時間オートマトン遷移の論理式表現を与える有界モデル検査の方法を説明する．Alloy は実数を持たないことから，有理数を取り扱う簡単な自作ライブラリを用いた．

項番 6 で取り扱うモデリング（Modeling）とは何かを整理しておこう．ソフトウェア工学では，プログラミング（Programming）に対比してモデリングという言葉を使うことがある．前者がプログラムを作成することであるから，後者はモデルを作成すること．UML は図式をモデル表記法とするモデリング言語である．

一方，ここでは，対象の特徴を抽出するという素朴な意味で，モデリングという言葉を使う．つまり，自然言語や設計図式を用いる従来の仕様書から，形式仕様を得る作業のことを指す．

表 2 のモデリングでは，2 つの方法を試みた．ひとつは，従前，形式手法からみた OCL として説明した内容を，OCL によるモデリング記述から Alloy を得る手順とした．ふたつめは，文献 [9] の方法を新たに付け加えた．アンドロイド・アプリに関する非形式的で説明的な技術資料から，形式仕様記述を得る試行錯誤・繰り返しの過程を扱うもの．この学習項目は，形式手法研究者との議論の結果[*6]，導入した．限られた時間の中で，「実作業から学ぶ」スタイルを試みるものといえる．

最後に，表 2 のカリキュラム全体を振り返る．当初のカリキュラムでは，Alloy をメタ言語の論理式の代わりに用いるという方針だった．標準的なモデル検査の説明として SPIN を用いることが最も大きな違いになる．SPIN はモデル検査が自動検証ツールとして，よく用いられることから，カリキュラムに入れることが妥当と考えたことによる．同時に，モデル検査に関わる他の教材では，高度な話題として説明が省かれる抽象化について説明したことが特徴だろう．

SPIN が採用しているオートマトン・ベースの方法では，モデル検査とは，状態遷移グラフの網羅的な探索である．一般に言われることであるが，モデル検査では状態爆発が問題となる．これを解決するには，抽象化による方法と振舞い等価を利用する方法がある．

[*6] 文献 [7] の発表ならびに ICFEM2014 で Liu 教授に依頼されて行ったパネル討論での発表．

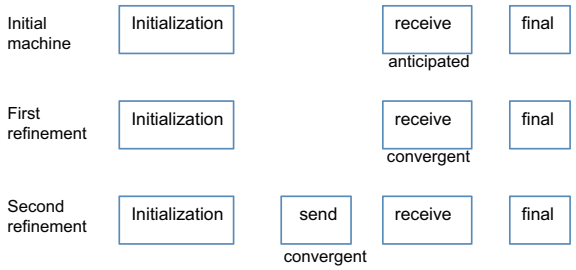

図 3 リファインメント・ステップ

述語抽象（Predicate Abstractions）の応用として，検査対象状態空間の大きさを規定するパラメータが未知の場合のモデル検査の例を考える．通常の方法では，パラメータを具体的な値に確定しなければならない．そのような例題として，Event-B による仕様記述を検査する．安全性（Safety）は述語抽象で取り扱えるが，進行性（Progress）を検査すると見かけの不具合が生じる．これを解決する方法として，ランキング抽象（Ranking Abstractions）を合わせて紹介した．

また，振舞い等価を利用する方法が有用な例として，時間オートマトンを取り扱う．時間を表すクロック変数は実数値をとり，無限の大きさの状態空間となる．そこで，振舞い等価な有限の離散状態遷移系であるリージョン・オートマトンを構成する方法である．

以上によって，モデル検査の基本は素朴な方法である一方，興味深いソフトウェア・デザインの自動検証には，さまざまな技法を併用する必要があることを学べる．

4. 教材の例

モデル規範による状態ベース仕様で最も重要なリファインメントに関連する話題を，Event-B 仕様の例を用いて説明する教材を紹介する．

4.1 振舞い仕様の検証

表 2 の項番 3 および項番 4 に関連する．ファイル転送（File Transfer，FT）の簡単な例を用いて，リファインメント関係の検証，および，抽象化支援モデル検査による検証を Alloy で説明する．

4.1.1 リファインメント関係の検証

図 3 に．3 段階の仕様記述，つまり 2 段階のリファインメントからなる Event-B 記述の全体像を示す．ファイル f の中身を g に転送するもので，第 1 段階は次の 2 イベントで構成される．

$$\texttt{recv} \triangleq \textbf{when}\ (\texttt{g} \neq \texttt{f})\ \textbf{then}\ (\texttt{g} :\in \texttt{N} \leftrightarrow D)$$
$$\texttt{final} \triangleq \textbf{when}\ (\texttt{g} = \texttt{f})\ \ \textbf{then}\ \textbf{skip}$$

上記のイベント仕様では，転送が完了していない（$g \neq f$）時，何らかのデータを g に格納することを表している．あ

る時点で転送が完了するという進行性を表す．

Alloy で表した Event-B の証明条件の例を次に示す．recv イベントに対する FIS（後状態の存在性）と INV（不変量の保存）である．

```
assert receiveFIS {
  all x : State0 | some y : State0 |
     not(x.g = x.f) implies (some y.g)
}
pred inv0 ( x :  State0 ) { x.g in Natural -> D }
assert receiveINV {
  all x, y : State0 |
      not(x.g = x.f) and inv0[x] and (some y.g)
    implies inv0[y]
}
```

次に 2 段階めを考える．ファイル f を n 個のチャンクに分割し，変数 r によって最後に受信したチャンクの位置を表す．また，転送終了を表すブール値変数 b を導入．論理式 $b = TRUE \Rightarrow g = h$ は垂直リファインメントで必要な条件[*7]を表す．

```
recv  ≙ when (r ≤ n)
          then (h := h ∪ {r ↦ f(r)}), (r := r+1)
final ≙ when (r = n+1),(b = FALSE) then (b := TRUE)
```

先の第 1 段階の場合と同様に，FIS と INV をイベントごとに証明する．

```
assert receiveFIS {
  all x : State1 | some y : State1 |
    inv1[x] and lt[x.r, x.n]
       implies (y.g = x.g + (x.r -> (x.f)[x.r]) and
                y.r = add[x.r, One])
}
assert receiveINV {
  all x, y : State1 |
      inv1[x] and lt[x.r, x.n] and y.r = add[x.r, One]
                and (y.g = x.g + (x.r -> (x.f)[x.r])
    implies inv1[y]
}
```

次に，いつか final イベントが実行されるという進行性を証明する．これには VAR と DLKF の 2 つの証明条件が関わる．

証明条件 VAR（変量の減少）は当該イベントだけの無限回発火状況が起きないことを保証する．Event-B では，発火可能なガード条件を持つイベントがない状況をデッドロックと呼ぶ．逆に，少なくともひとつのイベントのガード条件が発火可能なことがデッドロック・フリー．証明条件 DLKF（デッドロック・フリー）が示せれば，処理の進行を保証できる．この例は 2 つのイベントしかないので，recv イベントが VAR 条件を満たして，かつ DLKF であることがわかれば，いつか final イベントが実行されると結論

*7 糊付け（glue）条件．

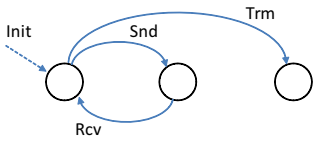

図 4 振舞い仕様

つけることができる．

VAR の証明では，イベント発火と共に減少するが無限降下しない変量（Variants）を導入する．$n - r$ は，たしかに転送回数が進むにつれて単調減少する．

```
assert receiveVAR {
  all x, y : State1 |
      inv1[x] and lt[x.r, x.n] and y.r = add[x.r, One])
              and (y.g = x.g + (x.r -> (x.f)[x.r])
    implies lt[(sub[y.n, y.r]), (sub[x.n, x.r])]
}
assert DLKF {
  all x : State1 | x.r in (x.range2 + x.n)
          implies (lt[x.r, x.n] or x.r = x.n)
}
```

さらに，第 1 段階の記述のリファインメントになっていることを，証明条件 GRD（ガード条件の強化），SIM（模倣関係）で示す．

```
assert receiveGRD {
  all c : State1,  a : State0 |
     inv0[a] and abs1[a,c] and inv1[c] and (lt[c.r, c.n])
  implies not(a.g = a.f)
}
assert receiveSIM {
  all c1, c2 : State1, a1 : State0 | some a2 : State0 |
    inv0[a1] and abs1[a1,c1] and inv1[c1]
             and (lt[c1.r, c1.n])
             and ( c2.g = c1.g + (c1.r -> (c1.f)[c1.r])
                  and c2.r = add[c1.r, One] )
  implies abs1[a2,c2] and (some a2.g)
}
```

図 4 は 3 段階め相当の振舞い仕様である．チャンク単位の転送（send と recv）を繰り返す．

```
send  ≙ when (s = r), (r ≠ n+1)
        then (d := f(s)), (s := s+1)
recv  ≙ when (s = r+1)
        then (g := g ∪ {r ↦ d}), (r := r+1)
final ≙ when (s = n+1) then skip
```

第 1 段階の場合と同様に，FIS と INV をイベントごとに証明する．さらに，第 2 段階の記述のリファインメントになっていることを，証明条件 GRD（ガード条件の強化），SIM（模倣関係），DLKF（デッドロック・フリー）で示す．

```
assert receiveGRD {
  all c : State2,  a : State1 |inv1[a] and abs2[a,c] and
      inv2[c] and c.s = add[c.r, One]
```

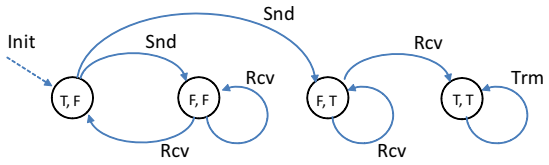

図 5 抽象化した遷移システム

```
    implies (lt[a.r, a.n])
}
assert receiveSIM {
  all c1, c2 : State2, a1 : State1 | some a2 : State1 |
    inv1[a1] and abs2[a1,c1] and inv2[c1]
             and c1.s = add[c1.r, One]
    and ( c2.g = c1.g ++ (c1.r -> c1.d)
        and c2.r = add[c1.r, One]
        and c2.s = c1.s and c2.d = c1.d )
implies abs2[a2,c2] and
       ( a2.g = a1.g + (a1.r -> (a1.f)[a1.r])
         and a2.r = add[a1.r, One] )
}
assert DLKF {
  all c : State2, a : State1 | inv1[a] and abs2[a,c] and
       inv2[c] and (lt[a.r, a.n] or (a.r = a.n))
    implies ( (c.s = c.r and not(c.r = c.n))
           or (c.s = add[c.r, One]) or (c.r = c.n) )
}
```

Event-B の標準ツール RODIN では，証明条件の生成と検証作業を自動的に行う．一方，本教材では，証明条件を Alloy で具体的に書き表し実験することで証明条件の成り立ちを理解することができる．

4.1.2 抽象化支援モデル検査

FT の例は，最終的にファイル転送が完了する．このような進行性は振舞い仕様を調べれば良い．素朴にモデル検査を行う場合，転送回数 n を具体的（たとえば 3）に与える必要がある[*8]．以下の Promela 記述例では，無限長の実行列を生成するようにした．

```
#define n 3
int s = 1; int r = 1;
active proctype System () {
  do
   ::  (s == r) && !(s == n + 1) -> s = s + 1    /* Snd */
   :: !(s == r) -> r = r + 1                     /* Rcv */
   ::  (s == r) &&  (s == n + 1) -> s = 1; r = 1 /* Trm */
  od
}
```

モデル検査を行うと，安全性（例，[](s <= n+1)）に加えて，「いつか転送終了する」という進行性（[]<>trm）が成り立つことを確認できる．

次に，任意の定数値 n に対するモデル検査を行う．命題値の組 $\langle (s=r), (s=n+1) \rangle$ で状態を表す．対応する状態遷移を図 5 に示した．

[*8] Event-B/RODIN のモデル検査ツール ProB でも具体的な数値を与える．

```
bool b1 = true;   /* (s = r) */
bool b2 = false;  /* (s = n + 1) */
active proctype AbsSystem () {
  do
   ::  b1 && !b2 -> b1 = false;
                    if :: b2 = true :: b2 = false fi
   :: !b1 -> if :: b1 = true :: b1 = false fi
   ::  b1 && b2  -> b1 = true; b2 = false
  od
}
```

この遷移系は，元の記述から抽象状態間の遷移（抽象遷移）を導くことで構築できる．抽象状態に対応する具体状態は，述語の連言で定義できる．述語抽象の方法で，具体状態を構成する述語から抽象化した命題の連言を求める．前状態で成り立つ述語論理式，ガード条件，アクション，イベント実行後に成り立つか否かを調べる述語から作られる論理式が充足する時，対応する抽象状態の間で遷移が存在すると判断する．Event-B のイベントの場合，この計算は簡素化することができる．初期状態からはじめて，発火可能なイベントを選択して得られる論理式の充足判定を行う．図 4 を参考にして，選択するイベントの順序を決めれば良い．

抽象遷移計算の具体例を示す．初期状態からイベント send 実行後に，述語 $(s=r)$ と $(s=n+1)$ が成り立つかを調べている．

```
fact { all a, a' : State | a.n = a'.n }
fact { all a : State | a.n1 = add[a.n,1] }
pred s1 (a, a' : State) {
  (a.s = a.r) and not(a.s = a.n1) and (a'.s = add[a.s,1])
              and (a.r = a'.r) and (a'.s = a'.r)
}
pred s2 (a, a' : State) {
  (a.s = a.r) and not(a.s = a.n1) and (a'.s = add[a.s,1])
              and (a.r = a'.r) and not(a'.s = a'.r)
}
```

s1 が充足する時 $(s = r)$ が成り立ち，s2 が充足する時 $!(s=r)$ が成り立つ．抽象状態を定義する命題変数で考えると，前者は b1 が成り立つ場合，後者は!b1 が成り立つ場合を示す．Alloy で検査すると，s1 は充足しないが，s2 は充足する．同様な計算を $(s=n+1)$ に対して行うと，両方とも充足する．図 5 の初期状態からイベント send による非決定的な 2 つの遷移が存在することがわかる．

上記で得た抽象遷移系に対して，LTL 式 []<>trm のモデル検査を行うと，反例が生成される．この反例は，send と recv を繰り返す無限ループの存在を表すもので，見かけの不具合である．そこで，ランキング抽象の方法によって，この進行性を検査すれば良い．証明条件 VAR で変量を導入したように，無限降下しない単調減少列を生成するランキング関数を適切に選ぶ必要がある．たとえば，$2n-(s+r)$

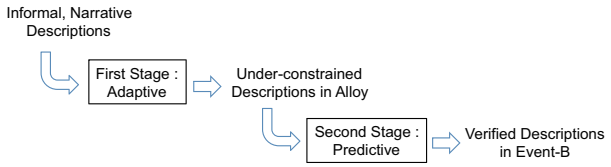

図 6 2 段階過程

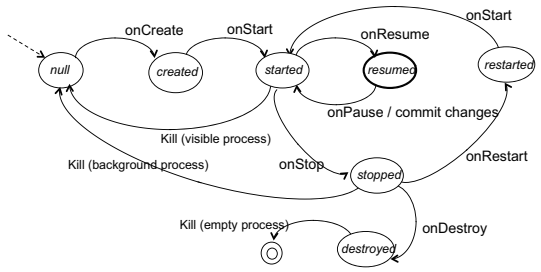

図 7 ライフサイクルの状態遷移マシン

を用いる．

4.2 仕様形成の繰り返し過程

Event-B では，リファインメントを用いて構築からの正しさの方法に従い，形式仕様を段階的に作成する．試行錯誤で行うものではなく，予め作成したリファインメント計画 [4] に従って行う．

リファインメント計画は初期仕様から目的記述を具体化していく手順の概略を記したもの．これを作成するには，初期仕様と共に目的記述に関する情報が必要になる．記述対象について非形式的な説明の文書が入力になるとしても，予め目的記述が存在することはあり得ない．何らかの方法で，目的記述のスケッチを得る必要がある．

この教材では，Alloy を用いて入力文書を調査・分析する繰り返し作業過程を導入する．この作業の結果は，制約条件不足（Under-constrained）かもしれないが，曖昧さのない Alloy 記述である．この「厳密性のあるスケッチ」は，Event-B のリファインメントによって得る目的記述と等価になる．これをもとにリファインメント計画を作成し，Event-B の仕様構築・検証を行えばよい（図 6）．以下，教材の概要を紹介する．詳細は文献 [9] を参照のこと．

対象はアンドロイド・アプリで用いるアクティビティの振舞いであって，Android フレームワークの標準の開発者向け文書（Developers Document）が入力．アクティビティのライフサイクルは，基本的にはコールバックメソッド起動を遷移エッジに持つ状態遷移（図 7）で表せる．コールバックメソッド実行終了後に，状態として表された「安定ステージ」に移行する．

状態遷移マシンは対象の振舞いを表す整理された記述であるが，標準の文書は，図 7 のような情報を明示していない．この図は，文書に散らばっている断片的な記述から本質的な振舞いと見做した情報を寄せ集めて作った．この記述が妥当であるか否かは，同じ文書中にある動作シナリオを再現することが可能かによって判断する．一方で，図式表現は，非形式的であって曖昧さが残ることから，動作シナリオとの整合性検査を行うことが難しい．教材では Alloy を用いるので，状態遷移マシンの解析が可能である．

5. おわりに

本カリキュラムは総合研究大学院大学で 2009 年より隔年，東京工業大学で 2012 年より毎年，実施した．年によって若干のバラツキはあるが，12 回程度の授業を行った．2016 年東工大では，15 回フルに授業を実施することができた．講義スライドのコピーと Alloy ならびに Promela の記述サンプルを配布し，受講生が自身で，記述例をカスタマイズ，実験できるように配慮した．

記述サンプルの多くは Alloy によるもの．見方によっては，Alloy の記述例となっている．文献 [1] にはアプリケーションの仕様記述例が数多くある．本教材の記述例は形式手法の「仕組み」を理解する事例集といえる．ツールで作動させて理解を深めるという利点がある．一方で，有界スコープ解析に起因する見かけの反例生成の問題が残る．ツールの出力結果に細心の注意を払わなくてはならない．これは，欠点であると同時に，自動解析ツールの出力結果を鵜呑みにできないことから，常に結果を吟味するという姿勢が必要になる．逆に，教育的な観点では良いかもしれないと考えている．

謝辞 東工大での講義実施にあたり，お世話になった米崎直樹名誉教授ならびに西崎真也教授に感謝する．

参考文献

[1] D. Jackson ：抽象によるソフトウェア設計：Alloy ではじめる形式手法，中島震（監訳），オーム社 2011.
[2] 中島震：ソフトウェア工学の道具としての形式手法，NII テクニカルレポート，2007-007J，2007.
[3] 中島震：SPIN モデル検査：検証モデリング技法，近代科学社 2008.
[4] S. Nakajima : A Refinement Planning Sheet, *Rodin User and Developer Workshop 2010*, Dusseldorf, 2010.
[5] 中島震：軽量形式手法ツールで学ぶ形式手法の基本，In *Proc. DSW & DSS 2011*，2011.
[6] 中島震：形式手法入門：ロジックによるソフトウェア設計，オーム社，2012.
[7] S. Nakajima：Teaching Formal Methods with Alloy, In *Proc. SOFL + MSVL 2014*, pp.97 - 110, 2015.
[8] 中島震，來間啓伸：Event-B：リファインメント・モデリングに基づく形式手法，近代科学社 2015.
[9] 中島震：Alloy と Event-B を用いる 2 段階モデリング手法，電子情報通信学会ソフトウェア・サイエンス研究会，札幌，2015.
[10] 中島震：共通理解フレームワークとしてのサイバー・フィジカル・システムズ，情報処理学会第 190 回ソフトウェア工学研究会，博多，2015.