ISSN 1346-5597

National Institute of Informatics

NII Technical Report

ソフトウェア工学の道具としての形式手法
Formal Methods as Software Engineering Tools

中島　震
Shin NAKAJIMA

NII-2007-007J
July 2007

URL: http://www.nii.ac.jp/　Tokyo 101-8430, JAPAN　+81-3-4212-2000　FAX: +81-3-3556-1916

ソフトウェア工学の道具としての形式手法
– 彷徨える形式手法 –

中　島　　震[†,††]

高い信頼性を達成するための技術として形式手法に注目が集まっている。産業界での急激な関心の高まりの反面、実際のところ、地道な積み重ねのある欧米から周回遅れという感を否めない。本稿では形式手法の技術全般について筆者の経験を中心に解説する。

Formal Methods as Software Engineering Tools
– An Exile in FM Wonderland –

Shin NAKAJIMA[†,††]

Formal methods, whilst actually not, are expected to be a panacea for attaining to highly reliable software systems. In spite of such fever, we are more than one lap behind Western, where a long term steady effort has been made. This paper tries to illustrate what happens in the world of formal methods, which is mostly based on the author's personal experience.

1. はじめに

ソフトウェアの信頼性や安全性に対する関心が高まり、形式手法 (Formal Methods) が注目を集めている。産業機器を対象とする機能安全に関する IEC-61508、自動車向けの ISO-26262、あるいはシステム・セキュリティに関する Common Criteria など、欧米主導の国際規格で、形式手法ならびに支援ツールを利用することを推奨している。また、2006 年には経済産業省が発表した「情報システムの信頼性向上に関するガイドライン」でも形式手法に言及している。

同時に技術者個人レベルでも形式手法に関心を持つ方々が急速に増えている。以前であれば、形式手法の否定的な面を追認するために調査する程度であったのが、利用の仕方を知りたいという肯定的な面から積極的に関心を持つ方が増えている。実際、2006 年 12 月に国立情報学研究所で開催した「形式手法ワークショップ」[☆]には約 80 名が来場された。「最近、業務の中で」形式手法を知ったという方が半数以上であり、ちょっとした驚きだった。

一方、関心が高まっている反面、形式手法がどのような技術であるのか、正確に理解されているか不安にも感じる。その理由としては、以下を考えることができよう。

- 入手しやすい情報の絶対量が少ない
- 長い歴史の中で形式手法そのものが変化している
- 多様な要素技術の総称である

さらに、形式手法に関わる情報の多くは、論文はもとより、入門的な解説に至るまで、「(特定の) 形式手法の仕組み」について述べるものが主流である。仕組み、あるいは、理論的な基礎が大切であることは言うまでもない。しかし、「形式手法を道具として使う」という観点を蔑ろにして良いというものではない。工学的な技術は、現実に使われてはじめて価値がある。

本稿では、歴史的な発展の流れを加味しながら、現在の視点から形式手法の概要を解説する。上にも述べたように、形式手法はある種の技術の総称である。とても一人がカバーできるような領域ではない。そこで、個人的な経験を中心に整理することとした。多少のバイアスがかかることは容赦願いたい。

2. 形式手法の現在像

2.1 総合技術としての形式手法

「形式手法は何であって、何ができるの?」という

† 国立情報学研究所
National Institute of Informatics
†† 総合研究大学院大学
SOKENDAI
☆ 情処学会 SIGSE 組込み WG、NII、IPA/SEC の合同企画として実施。

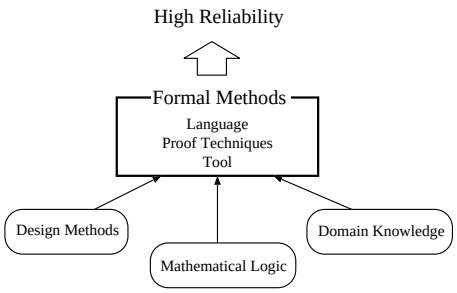

図 1　総合技術としての形式手法

素朴な質問への標準的な回答例は次のようなものであろう。

> 形式手法は、システム、特に、ソフトウェアの開発法であって、数理論理学に基づく科学的な裏付けを持つ。明確で厳密な意味を持つ言語を用いて設計対象を表現することにより、設計記述の正しさを系統的に示すことが可能になる。したがって、開発したシステム、あるいは、ソフトウェアが高い信頼性を持つことを保証することができる。

ここで強調されることは、「数理論理学」、「言語」、「正しさ」である。すなわち、

> 数理論理学に基づく言語を用いて設計対象を表現し正しいことを保証する

のである。結局のところ、

> ある言語を用いた記述を作成する

ことが第一歩である。これは、プログラミングと何ら変わらない。「では、何が異なるの?」と聞かれて、

> 数理論理学だよ

と、「ニヤリ」とするようでは、形式手法の工学的な発展はない。何しろ、「数学が得意だった」という人にお目にかかることは少ないのだから。「数学が苦手でもプログラミングは得意だよ」、「数学が大嫌いだから文科系にいったのに、就職して SE になったら、数学なの?」、というのが大方の反応でしょう。たまに、「数学はまぁまぁだったけど、ソフトウェア開発と関係するの?」という人もいるかもしれない。心情的にも、「現実にも」、

> ソフトウェア開発 ≠ 数学

である。この辺りについては、後に改めて考えたい。

さて、プログラミングが簡単かというと、そうでもない。

> プログラミング言語 (の定義) を知っていてもプログラムは作れない

開発対象に関する知識 (ドメインの知識)、設計法に関する知識、典型的な解法に関する知識☆、などなど、が必要になる。さらに、いわゆる「デバッグ法」に習熟する必要がある。「コンパイル一発、テスト一発」、でない限り、使っているプログラミング言語ならびに開発支援環境特有の「デバッグ法」を知らないと、自作プログラムを目の前にして途方に暮れるばかりである。

形式手法もプログラミングに似ている。異なるところは、プログラミングを含む「設計からの開発手法」を取り扱う技術であるという点、「高い信頼性」を達成するための手段を提供することに重きがあるという点、である。「言語」を用いて開発対象を記述するという観点では、プログラミングと同様な難しさがある。

冒頭の問いの前半、「形式手法は何?」に対する本稿の答えを図 1 に整理した。すなわち、形式手法は、

> 数理論理学に基づく言語ならびに証明技法などからなる基礎理論を持つことに加えて、設計手法、ドメイン知識、などが関係する総合的な技術

である。

2.2　形式手法の使い方

次に、先の問いの後半、「形式手法は何ができるの?」を考えるに際して、12 月のワークショップ参加者アンケートから得た情報を使わせて頂く。形式手法には次のような 4 つの使い方があることが参加者に知られていた。

- 正しいシステムだけを系統的に開発
- 記述に不具合がないことを数学的に証明し保証
- 厳密な言語を用いることで仕様を明確化
- 記述中に隠れている不具合を開発早期に発見

以下、4 つの使い方について説明する。

2.2.1　究極の目標

第 1 の使い方は、形式手法の究極の目標であり、夢

☆ 「データ構造とアルゴリズム」といった科目がある、(あった)。最近では「デザインパターン」でしょうか。

と言っても良い。当該の形式手法が示す手順にしたがって、開発対象システムの仕様を形式仕様言語で書き表す。記述内容が正しいことを数学的に証明することができ、誤っている箇所はたちどころにわかる。抽象度の高い記述から徐々に具体的なプログラムによる実行形式に近い記述に変換する。変換は正しいことが保証されているので、正しい記述から正しさを保証された変換方法によって得た最終的なプログラムも正しい。したがって、この「段階的な詳細化」に基づくリファインメントの方法によって、正しいシステムだけを系統的に開発することができる。システム開発の全工程が形式手法の対象となる。

この考え方を、Correctness by Constrution (CxC) と呼ぶ。形式手法に関する基礎的な研究がはじまった 1970 年代の当初の「目標＝夢」であり、現在も「夢」である。さらに、近い将来も「夢」のままかもしれない。

2.2.2 形 式 検 証

第 2 の使い方、形式検証は、上記の説明の中でも現れている。形式検証は、当初の夢を実現するための最も大きな技術的な関心事であった。一般に、システムの信頼性を向上させる技術としては、プログラム・テスティング技法が知られている。テスティングは、特定の入力データに対してプログラム実行を行うことで不具合を発見する方法である。入力データの選び方に失敗すると、不具合が発見できない。

当初、形式手法は、不具合を発見する技法であるテスティングに対して、不具合がないことを数学的に証明する技術の確立を目指した。この頃に、形式手法の技術に触れた方々は、形式手法に対して、否定的な見方を持つことが多い。

プログラムを定理とみたて、「定理が成り立つことを系統的に証明」する。そのためには、「プログラム」を表現し、その「性質」を証明するための論理体系が必要である。そのためには、「プログラムの性質」とは何であるかを考えなければならない。プログラムとして表現したい内容は多岐にわたるため、非常に表現力の豊かな論理系が必要になる。論理系は表現力が大きくなると自動的に論理式の真偽値判定ができない。たとえば、述語論理は決定不能である。したがって、人間が証明しなければならない。まさに、「数学」の演習問題になってしまう。これをプログラマが行う、というのであるから否定的にならざるを得ない。

このプログラム検証については、誰もが知っている Tony Hoare や E.Dijkstra たちによる基礎研究がある。基礎的な理論なので情報学科の講義で取り上げられるであろう。しかし、プログラミングに役立つという実感はない。

2.2.3 テスティングと形式仕様

話を次に進める前に、プログラム・テスティングが持つ問題点を考えてみたい。テスティングは産業界で日常的に使われて役立っているので、ここでは問題点だけを考えるのである。

まず、テスティングの対象はプログラムであるということに気がつくべきである。いろいろなところの集計によって、開発上流工程に原因のある不具合混入が多いことがわかっている。これを、テスティングによって発見するということは、2 つの問題を残すことと同じである。すなわち、

- テスティングまで不具合を放ってある
  「何とまぁ無責任な!」
- 改修には工程の後戻りがつきまとう
  「完成間近で製品出荷の見通したたず?」

である。テスティングしか手段がなければ、それもしかたない。でも、我々には、形式手法がある。形式手法を使う可能性が残されている。というわけで、第 3 の使い方が登場する。

この使い方では、形式手法の中核となる言語を用いて、上流工程で作成する開発成果物であるデザインを書く。特に、システムに求められる要件を厳密に書くことを特徴とする。

究極の目標 (Correctness by Construction) に近付くためには、開発の全工程を形式手法の観点から見直す必要があった。しかし、日常の開発方法を全面的に変えることは現実的なアプローチではない。そこで、All-or-Nothing ではなく、「使えるところから使い始める」Formal Methods Light の考え方に移ってきた。特に、システム要件やデザインを曖昧さなく厳密に作成することで、開発上流工程での不具合混入を防ぐ。プログラムを対象とする信頼性向上のテスティングと相補う技術の確立を目標とする。

実際、長期間、多人数が関わる大規模プロジェクトでは、仕様の解釈のちょっとした違いが頻繁に起こることが容易に想像できる。多義的な解釈が可能であれば、解釈の不一致がプログラム作成工程に伝播し、ようやく統合試験になって異なる理解のもとに作業していたことに気がつく。

さらに、将来にわたって、多数の技術者が参照する相互運用に関わる国際標準文書の記載内容など、曖昧さをなくすことの大切さを説く例はいくらでもある。

さて、どのような観点から、デザインを考えるのであろうか。形式手法の黎明期では、ソフトウェアのデザインといえば、プログラム設計に関わる決定事項で

あった。データ構造、アルゴリズム、操作あるいは関数の機能仕様、などが関心事であった。ソフトウェア工学の分野でいえば、構造化設計といえる。

ご存知の通り、その後、オブジェクト指向モデリングの考え方が登場した。また、システムの大局的な論理構造を示すアーキテクチャの重要性も広く知られるに至った。オブジェクト指向モデリングの中心に UML があるといっても、UML は単一のデザイン記法ではない。記法ファミリーであり、クラス図に代表される静的な情報構造からステート図のような動的な振舞いまで、多様な側面を記述する道具を提供する。

形式手法を用いることの利点は、厳密な言語を使うことによって、記述内容に関する Reasoning ができることである。すなわち、記述内容が意図通りであるかを調べる Validation、記述内容に論理的な矛盾など何らかの観点からの不整合があるか否かを確認する Verification などを、系統的に行うことができる。

この長所は短所でもある。厳密なデザイン記述を作成しなければならないため、手間の問題が大きい。UML 等のダイアグラム記法ではスケッチのような記述であっても、それなりに、熟練した技術者によるレビューが可能であろう。一方、形式手法の場合、スケッチだけで終らせることは、デザインを厳密に書き表すというそもそもの目的に合致しない。したがって、「工数」がかかることは当たり前のことである ☆。

さらに、厳密な記述を得ることは想像以上に難しい。ある規模のプログラムを作成する場合、正しく作動するかの以前に、一回でコンパイラが通ることはめったにない。ちょっとした入力ミス、構文の誤り、変数宣言忘れ、タイプ不一致、など、細心の注意を払ってもどこかでコンパイルエラーが出てくる。逆に、コンパイラに不具合箇所を指摘させて直せば良い。形式手法の場合も同様であって、たとえ高度な Reasoning ができなくても何らかのツール支援が必須である。

2.2.4 設計デバッグ

第 4 の使い方は、正しさの証明よりは、開発上流工程での不具合検出に焦点を当てる。形式手法を用いる場合、ある種の形式仕様言語を用いて厳密なデザイン記述を作成することになるため、何らかのツール支援が必須である。プログラミング言語のコンパイラが行うような静的検査だけではなく、Validation、Verification のための Reasoning もツール支援で行いたい。しかし、完全に自動化することは原理的に不可能である。また、形式手法ごとに基礎とする数理論理学の性質が異なる。

そこで、表現力あるいは解析能力のどちらか、あるいは両方を、限定することで、どこまで検証を自動化できるかという方向に話が進む。表現力を犠牲にすることが自動検証を達成する第一歩である。

多様なデザインのある側面だけを表現する形式仕様言語を導入し、その代わりとして、その側面に特化した自動解析の方法を確立する。このような自動検証ツールを前提とした形式手法を Light-weight Formal Methods (LFM) と呼ぶ。すなわち、LFM は Formal Methods Light の考え方を具体化したひとつの方法であって、特に、自動検証ツールを用いることを前提とするものである。

もうひとつの方向は Validation に特化しようという考え方である。Validation の有力な方法は具体的なテストデータの下での仕様実行である。実行結果、あるいは実行経過を見ることで、設計者は意図通りのデザインになっているかを確認することができる。プログラム・テスティングの代わりにデザイン・テスティングを行うのである。プログラムの実行と区別するために、仕様アニメーション (Specification Animation) と呼ぶことが多い☆☆。

これを良く表すのが、Validation through Animation という言葉である。ただし、この考え方を採用した言語では、言語要素が実行意味を持つものに限定されるので、表現力を制限している。さらに、与えたテストデータに対する確認しか行っていないことに注意しなければならない。プログラム・テスティングとの違いは、テスト対象が開発上流工程の成果物であるデザイン記述という点である。

さて、形式仕様言語を用いてデザインあるいは仕様を書くのであるが、形式仕様記述を作成する目的を明確に意識しなければならない点が、プログラミングの場合と大きく異なる。プログラムの場合、作り方の上手下手を考えない場合、とにかく作動して意図通りの計算結果が得られれば良い。あるいは、実行性能の向上に重きをおき、高速化のテクニックを駆使する。その結果、可読性が低下する場合も多く、後の保守が難しくなることもある。

形式仕様の場合、作成する目的の明確化が重要である。通常は、方式あるいはアルゴリズムが正しいことを確認あるいは証明するために形式仕様を作成する。

☆ オブジェクト指向設計が導入され始めた頃、上流工程の「工数がかかる」という理由で、導入に否定的なプロジェクトリーダが多数いたことを思い出す。

☆☆ Animation というのは「動かす」ことであって「虫プロ」が持つ技術のことではない。

一方、設計段階では、どのような場合に、どのような不具合が生じるのかを調べておきたいことがある。不具合が起こることの確認を目的とする使い方といってもよい。これによって記述対象の理解を深めることができる。すなわち、不具合モデルを作成することも有用なことがある。デザイン作成の目的が異なるので、形式仕様の書き方や注意点が違う。

2.3 技術の成熟さ

形式手法は開発対象システムに厳密な記述を与える手段である。その目的は、開発組織、開発プロジェクトごとに決めなければならない。また、目的に応じて、記述する内容が異なる。さらに、後の節でみるように、数多くの形式手法があるが、各々得意とする場面が異なる。目的に合わせて、適材適所の形式手法を選択する技術力を持つことが必要である。

今後、形式手法は究極の目標に向かった「夢」を実現するための研究と工学的な観点からの実用技術の 2 つに分化していくようである。技術が成熟してきた証であろう。

さて、20 世紀末、1999 年は、アカデミズムを中心とする形式手法の分野で、少し変わった動きをみた年であった。南仏ツールーズで

> World Congress on Formal Methods in the Development of Computing Systems

と題する大規模な国際会議 (略称 FM'99) が開催された。どちらかというと、Federated Conferences の形をとり、主コンファレンスの他、12 のワークショップあるいはユーザグループ・ミーティングが開催された[10]。いわば、形式手法の見本市のような要素のある国際会議で、

> 形式手法の研究者間の綱引きに労を費やすよりは、システム開発に広く役立つことをアピールしたい

というのが全体の論調である。何度目かの「形式手法、冬の時代」の頃であった。

このような流れがあったためか、世紀が改まって、欧州では形式手法の「応用」を目指す EU プロジェクトが相次いだ。IC カード、Java カード、等の安全性と同時にセキュリティが重要となるシステムを対象とする研究が行われた。

北米では、NASA やマイクロソフトといった安定した巨大機関が自己の研究所で研究プロジェクトを行った。その後、英国 IEE グランドチャレンジの中心的な研究課題として、形式手法に関わる技術が取り上げられている。英国発であるが、欧州ならびにアングロ・アメリカン、さらに、残りの世界 (the rest of the world) も巻き込む勢いである。

「冬の次に必ず来る春」が (英国式の) 形式手法にも到来したといえる。この春が以前と異なる点は、冬の時代の熟成期間に成功事例を持った、ということであろう。その自信を胸に、再び、夢の目標に戻ろうとする動きも見られる。

3. 形式手法を鳥瞰

代表的な形式手法を鳥瞰する。同時に、各々の形式手法に対して、筆者の経験を併記する。

3.1 分　　類

数多ある形式手法を、ソフトウェア・デザインのどのような側面を表現し解析することに向いているか、という観点からいくつかに分類することから始める。ただし、ひとつの形式手法がひとつの分類項目に入るというものではない。複数の使い方が可能な形式手法は、当然のことながら、複数の分類項目に入る。

(1) 構造化設計
伝統的なプログラム設計に強く関連する情報、データ構造の定義、操作あるいは関数の機能仕様、などに対応

(2) 静的な情報構造
UML のクラス図、オブジェクト図、あるいはデータベース設計で用いる ER 図など、情報間の静的な関係の表現に対応

(3) 振舞い仕様
並行システムなど制御の流れが複雑になるためにデッドロック等の不具合が発生しやすい側面で UML ではステート図が対応

(4) リアルタイム性
実行のタイミング、通信遅延、など、時間的な特性が振舞いに影響を与えるソフトウェア

(5) プログラム検査あるいは検証
特定のプログラミング言語で書かれたプログラムが与えられた要求性質を満たすか否かを検査する、あるいは、検証する

(6) 検証エンジン
特定の論理系に基づく表現ならびに推論機構を提供するもので、他の形式手法ツールのバックエンドとして利用されることが多い。エンジンが提供する言語を用いて、直接、ソフトウェアのデザインを記述し解析してもよい。

(7) 理論あるいは計算モデル

特定の基礎理論あるいは厳密に表現された計算モデル (抽象的な実行の仕組み) が提供する概念を直接表現するための言語と解析方法を提供する。

使い方によっては、検証エンジンとしての利用が可能な形式手法も多いので注意すべきである。

なお、過去に、複数の形式手法を紹介する総合的な解説 11)15)16) を公表した。また、文献 4)5)17)6) の 4 編からなる「先端ソフトウェア・ツール紹介 (小特集)」を企画した。続編として、文献 19) を含む紹介記事を企画中である。参考にされたし。また、膨大な量になるため、本稿ではオリジナルの文献を引用していない。末尾に参考文献として示した文献から「孫引き」して頂ければ幸いである。

3.2 構造化設計

「構造化設計」と整理した形式手法は長い歴史がある。歴史的な発展の中で、モデル規範型 (model-oriented) の形式仕様と呼ばれる VDM、Z 記法、B メソッド、の 3 つが代表である。モデル規範という考え方では、形式仕様言語の言語要素ひとつひとつが、それに対応する数学的な対象を「表示」する。

共通する基本的な表現の道具は 2 つあって、データ構造に関わる不変量 (Invariant) と操作あるいは関数の機能を表現する事前・事後条件 (Pre-/Post-Conditions) である。また、上位の機能仕様が、どのようにプログラミング言語が提供する概念に近い具体的な仕様に詳細化されるかを系統的に取り扱うリファインメントの規則を持つものが多い。Correctness by Construction を目指していた。

さらに、2 つの共通点がある。産業界への適用に熱心な点、ならびに、複雑な数学を使わないことである。集合論や 1 階述語論理、関係や関数についての入門的な知識があれば、これらの形式手法にしたがって仕様を作成することに困難はない。馴染みのある数学的な対象に限定するということは、産業界への技術移転と表裏一体のものであると考えられる。

3.2.1 VDM

形式手法の発端は VDM であると言われている。1960 年代に IBM ウィーン研究所で始まった PL/I コンパイラの正しさを形式検証するプロジェクトがきっかけである。Tony Hoare などが中心的な役割を果たし、PL/I 言語定義のメタ言語として VDL (Vienna Definition Language) が考案された。また、言語仕様が整理されて VDM (Vienna Development Method) となった。その後、研究の中心は、デンマークと英国の 2 拠点に移る。

デンマークでは、D. Bjorner が中心となって、プログラミング言語の定義やコンパイラの正しさの検証、あるいは、一般のシステム仕様への適用など、VDM 流形式仕様の適用を通して技術の洗練を行った。新たな形式仕様言語 RAISE の提案、要求工学と関連が深いドメインモデリングへの適用などを行った。

一方、英国では、C. Jones らのグループが、VDM の意味論や証明技法などの基礎的な研究を進めた。部分関数を表現する論理系として、true と false に undefined を追加した 3 値論理を採用したことが特徴である。リファインメント規則の整理、段階的詳細化に基づくアルゴリズム導出法、証明支援ツール Mural の開発を行った。同時に、高等教育や産業界への適用を通して、Formal Methods Light の考えが認識された。

産業界への技術移転に関しては、デンマークでも活動があり、VDM 技術のコンサルタントを中心業務とする IFAD 社が設立された。VDMTools と呼ばれる開発支援環境も開発された。このような流れの中で、VDM-SL が ISO 国際標準となった。

1990 年代前半には、ESPRIT の Afrodite プロジェクトにおいて、Validation through Animation を標語とする実行可能デザイン言語として、オブジェクト指向拡張された VDM++が開発された。その後、リアルタイム性と並行性に関するモデル化機構を追加した VICE (VDM++ In a Constrained Environment) が、組み込みシステムへの適用を念頭において開発されている。現在、日本の CSK システムズが、VDM オープンソース・プロジェクト Overture と協力して、VDMTools の保守を行っている。

VDM-SL の特徴は、Implicit Specification と、Explicit Specification と呼ぶ 2 つのスタイルを使い分けることができる点である。Implicit Specification は、操作などの機能仕様を、事前条件を満たす状態と事後条件を満たす状態の間に成立する関係として宣言的に書き表す。VDMTools では、構文チェック、タイプチェックを行うと同時に、検証条件を生成する。検証条件を確認するのは人間であり、ツールを用いる場合であっても対話的な証明になる。

Explicit Specification は、実行可能デザインのことである。VDMTools ではインタプリタによって VDM 記述の実行を行って、意図通りの振舞いを示すかの確認を行うことができる。実行可能にするために、Implicit Specification よりは、表現力が小さい。すなわち、言語仕様を制限している。

もともと、Implicit スタイルでは、数学的な関係として機能仕様を書き表すため、通常のプログラムと

して実現可能かどうかがわからない。表現力が高いということの裏返しである。そのため、宣言的に書き表すことが好ましい上位の仕様をリファインメントする際に、詳細化過程のどこかで、実行可能であることを確認しなければならない。実行可能な形が存在することを、Explicit スタイルで構成的に示す、と考えてもよい。

宣言的な仕様を書くこと、さらに、正しさの検証を対話的に行うこと、などは、技術者にとって大きな負担となる。実行可能なデザインという考え方のほうが馴染みやすい。したがって、現在、VDM++や VICE では、VDM-SL の Explicit スタイルの考え方を発展させた Validation through Animation に重きを置いている。

筆者らは VDM-SL (Implicit)、VDM++ (Explicit) による記述実験を行ったことがある。ソフトウェア工学や形式手法の教科書では、上流工程の成果物は宣言的に記述するのが良い、といわれる。ひとつには、手続き的な記述に比べて、宣言的な記述の抽象度が高いので、問題の本質に注力することができ、これによって、一般性のある再利用の容易な記述を得ることができるからである。すなわち、プログラムが持つ実行イメージに近い手続き的な記述を行おうとすると、実行させるための細かな点に目がいき、その結果、問題の本質がぼやけてしまう。

一方、デザインの記述を得る際には、2 つの異なる作業がある。ひとつめは、たとえば、UML のダイアグラムなどを用いて表現したアイデアのスケッチをもとに、記述を詳細化、厳密化して、内容の理解を深める。あるいは、複数の方式を比較検討するような試行錯誤的な作業がある。ふたつめは、記述対象の機能や構造がはっきりしてきた段階で、後工程に残す設計情報として、あるいはリファインメントを行う目的で、厳密に宣言的な仕様を書き下す作業である。

これらの 2 つの作業は、いずれも設計における重要な側面であり、言語やツールに求められる性質も異なる。前者のような試行錯誤の段階では、実行可能デザインによるプロトタイピング的な方法のほうが技術者にとって馴染みやすいであろう。いずれにしても、形式手法に関する従来の説明は、最終的な結果としての「安定した」形式仕様が持つべき性質に力点を置くものであって、実際の設計作業とは少々趣が異なる。

また、筆者らは VDM-SL を用いたデザイン記述法を改善するひとつの試みとして、アスペクト指向設計の考え方を導入した AspectVDM を提案した。複数の操作定義の事前・事後条件に対して新しい機能を表現する述語を一括して追加するためにアスペクト☆を用いる、という考え方である。VDM-SL に対してオブジェクト指向概念を導入した VDM++があるように、今後も、形式手法と独立に提案される新しい設計法を取り入れることが大切と考えている。

最後に、VDM に関しては良い文献が入手しやすい。文献 2) は VDM-SL を用いた形式仕様記述の教科書である。文献 3) は VDM++を産業界で実適用した経験の報告である。文献 6) は VDM++を中心とした最近の動向を紹介している。

3.2.2 Z 記 法

Jean-Raymond Abrial らによる「集合論を仕様記述の道具に使う」という考え方に影響を受け、Tony Hoare 率いるオクスフォード大学のグループを中心に開発された。公理的な集合論 (Zermelo 集合) と 1 階述語論理にタイプ概念を導入した体系に基礎をおく。これらの数学的な対象に対応する表記法を提供するということから、形式仕様言語と呼ばずに、記法と呼ぶ習慣がある。J.M.Spivey による初期の意味論は、この「記法」としての考え方に基づく。

Z 記法を用いる場合、VDM と同様に、操作あるいは関数の機能を表現する事前・事後条件の組で表す。分解すると述語論理の式であるが、機能仕様としては強く関連する論理式の集まりを記述の単位としたい。そのために、スキーマと呼ぶ考え方を導入した。データ型に対応し状態を定義する状態スキーマ、初期状態を定義する初期スキーマ、状態の変化、すなわち、事前・事後条件の組をまとめて表す操作スキーマ、等に分類できる。

Z 記法の発展は、産業界との交流、あるいは適用の歴史でもある。実際、スキーマという考え方と表記法、数学ツールキットと呼ばれる一種の標準ライブラリ定義、などは、具体的な適用の経験から追加されたものである。この頃の経緯に詳しい向きには、「Z 記法は妥協の産物」という印象を持つ方もいらっしゃると聞いている。また、Z 記法が基礎とする論理系は表現力が高いため、記述の正しさの自動検証は困難である。当時、fuzz という支援ツールがあったが、構文チェックと部分的なタイプチェック、文書清書機能くらいであった。

さて、Z 記法が注目される理由は、産業界での実績によるところが大きい。代表的な事例は、オクスフォード大学と英国 IBM が行った CICS の仕様記述がある。常に取り上げられることであるが、1992 年に英国の

☆ Heterogeneous Aspect である。

Queen's Award for Technological Achievement を受賞した。CICS プロジェクトでは、Z 記法を用いて仕様書を書き表すことによって、自然言語やダイアグラムを用いた従来仕様書の問題点を顕在化させることができ、システム改版時の設計にフィードバックすることができた。さらに、曖昧さがないため、長期間にわたる保守のための拠り所の文書としても有用であったと報告されている。

多くの技術者が参照する標準勧告文書でも曖昧さをなくすことが大切であるため、Z 記法が用いられた。たとえば、ITU-T の ODP トレーダ、ANSI の RBAC などで、データ定義や操作の機能仕様記述に Z 記法が使われている。ところが、このような標準化文書の策定活動に関わる形式手法の専門家が少ないことが原因であろうか、Z 記法の記述部分に誤りが散見されることは、公式の標準勧告文書としては残念なことである。逆に、Z 記法に精通していれば容易に誤りがわかるのである。逆説的であるが、Z 記法を用いた効果ともいえる。

このような標準勧告文書への適用を試みる過程で、Z 記法自身も ISO 標準になった。標準勧告に使われる形式仕様言語は標準化されていなければならない、という説明である。J.M. Spivey の方法から離れて、厳密な意味論が整理されている。同時に、Z 記法による仕様記述の性質を系統的に議論する推論規則やリファインメントに基づく形式証明技法が整理された。最近では、Z の意味論を高階論理で表現し、Z による仕様記述を解析するツール、ProofPower、Isabelle/HOL-Z が開発されている。

整理すると、Z 記法は、当初、厳密な仕様記述のための記法として提案されたが、現在では、形式仕様記述言語として整備されるに至り、対話型の証明支援環境が開発されている。

余談であるが、Z 記法はオクスフォード大学が中心になっていたこともあり、関係する研究者は英国における形式手法の中心をなす。英国 IEE グランドチャレンジでは、形式手法の研究を推進するために、異なる形式手法の比較のために標準例題を準備している。現在、使われている例題の Mondex については、Z 記法を用いた形式仕様記述とリファインメントの文書が公開されている。他の形式手法の研究者は、Z 記法による Mondex 仕様を参照することが期待されているのである。

Z 記法は筆者が最初に触れた形式手法のひとつである。Knuth-Bendix の完備化手続きと呼ばれる方法があるが、これを勉強している時、Z 記法で書いたものをみつけた。出典は忘れたが、とても簡単に書かれていた。宣言的に書かれていたので分かり易かっただけといえば、その通りなのであるが、「Z 記法も良いねぇ」と思った次第である。その後、Z 記法へのオブジェクト指向概念の導入などの検討を行った。しかし、よいツールがないので、Z 記法を用いた大規模な仕様を書く気にはなれなかった。単に気分の問題である。

最後に、Z 記法に関しても入手しやすい文献がある。文献 2) は Z 記法にも言及している。文献 5) には最近の Z 記法周辺の話題である Isabelle/HOL-Z の紹介がある。文献 9) には Z 記法を用いた形式仕様の例題が紹介されている。

3.2.3 B メソッド

さて、3 番目の B メソッドに話を移す。VDM、Z 記法と同様に、産業界への適用に熱心であり多くの成功事例が報告されている。一方、他の 2 つが ISO 標準になっているのに対して、B メソッドは、考案者 Jean-Raymond Abrial の「目の届く」範囲で使われている。その特殊性が良い方向に作用したのであろうか、産業界における開発での利用という観点から Correctness by Construction の考え方が成功した唯一の形式手法といえる。B メソッドの特徴は、リファインメントを中心とする段階的な詳細化の技法によって、仕様からプログラム導出までをカバーする点である。

リファインメントの考え方は素直であり実用的な観点から魅力的である。正しいとされる初期仕様から出発して、段階的な詳細化に基づく開発方法にしたがって、プログラムを導く。詳細化の各段階での変換が正しいことを B ツールを用いて検証する。変換の正しさは半自動的に検証することができるので、詳細化の過程で誤りが混入することはない。

B ツールは詳細化が正しいことを証明するために、検証条件を表す論理式を生成する。ツールが持つ推論規則を用いて論理式が真になることを自動的に証明する。論理式が複雑であったり適切な推論規則がない等の理由によって、自動証明に失敗したりツールが応答しなくなることがある。適当な打ち切り時間を設定してツールを使用し、証明ができなかったことがわかると技術者が対話的に証明することになる。

実際、この方法によって開発した有名な事例として、パリ地下鉄の自動運行プログラム (Ada で 86,000 行)、シャルル・ドゴール空港シャトルの自動運行プログラム (Ada で 158,000 行) が知られている。詳細化の正しさを示す検証条件については、各々、92%ならびに 97%を自動証明でき、また、ユースケース相当の統合テストは行ったが、Ada プログラムモジュールの単体

テストは省略したという。

B メソッドが完全に自動検証できないとしても、リファインメントの方法が産業界で利用できるだけの成熟レベルに達したのか、少々不思議である。一方、VDM や Z 記法などでも、リファインメントは研究されているが、研究者が使うようなレベルなのである ☆。

その理由は、B メソッドのリファインメント規則にある。すなわち、リファインメントをデータの詳細化、非決定性の除去、の 2 種類とする。新規オペレーション追加も許さない。さらに、いずれの場合も、リファインメント前後で外部からみた振舞いが同じであることを要請する。特に、対象オペレーションのシグネチャを変えてはならない、という制限をおく。

B メソッドを用いる場合の難しさは、そもそも、最初に与える初期仕様の正しさをどのようにして確認するかということにある。誤りを含む仕様に対して正しいリファインメントを行っても無駄と言うものである。

もうひとつの難しさは、証明が完全に自動化できないことにある。B ツールは利用者が推論規則を追加していくことで、自動検証の率を向上させる仕組みになっている。自動証明に失敗した検証条件があっても、これを検証する手順が分かれば、その手順を整理して推論規則の形でツールに追加すればよい。ただし、推論規則を追加することは容易ではない。ある規則の追加によって、既存の体系が壊れるかもしれない。すなわち、推論規則体系の整合性を確認しなければならず、当該 B ツールに習熟した専門技術者の役割が大きい。

リファインメントは開発過程全体に影響を与えること、上に述べた 2 つの問題は本質的な難しさを持つこと、等を考えると、B メソッドが他手法と異なり、限られた世界での大きな成功を示していることが十分に納得できる。すなわち、B メソッド適用に際して、専門家によるコンサルティングが大きな比重を占めると思われる。

先に述べたように、B メソッドの言語仕様は Abrial が自身のコンサルタント会社をベースに保守している。最近になって RODIN プロジェクトが B メソッドを拡張した Event-B を公開した。Abrial が中心的な存在であることには変わりがないが、英国の大学等も巻き込んだプロジェクトであるため、B メソッドよりはオープンであると考えられる。

B メソッドに関しては、自分で何かを書いたという経験はない。来間さんが B メソッドで仕様作成する際に一緒に考えたということである。いつも議論になったことは、初期仕様の正しさをどのように保証するかであった。何をもって、正しいと呼ぶかも難しい問題である。

なお、B メソッドに関しては、国内で良い文献が見当たらない。文献 4) が唯一のものであろう。

3.2.4 OCL と Alloy

今までに説明した 3 つの形式手法と密接な関係にある OCL と Alloy を紹介する。

UML では OCL というテキストベースの言語が提供されている。Syntropy と呼ぶオブジェクト指向開発法で提案されたもので、現在 UML に含まれているのは OCL 2.0 である。Syntropy は OMT のようなダイアグラムを基本とするオブジェクト指向モデリングの考え方と Z 記法や VDM といった形式手法を統合する目的で提案された。OCL は、現在、MDA に必須の言語として重要性を増している。

OCL を用いると、メソッドの機能仕様を事前・事後条件の組で表現することができる。しかし、OCL の厳密な言語仕様定義に関しては、オブジェクト指向プログラミング言語にも共通するタイプ・システムの難しさ、1 階述語論理式と見做した場合の全称限量子の取り扱い、など、幾つかの問題点が知られている。

上に述べた OCL2.0 の問題点は、OCL を宣言的な機能仕様の表現のための形式仕様言語ではなく、実行可能な言語として用いたいと考えるところから来ている。インタプリタ実行させるだけであれば大きな問題とならない。実際、インタプリタ機能を提供するツール USE が開発されている。

本稿では、OCL を形式仕様言語の仲間とは考えないことにする。ただし、OCL の曖昧さを修正して形式性を高めることで、HOL を用いた形式検証の支援環境が開発されるプロジェクトがいくつかある。また、MDA 向けに OCL2.0 サブセット言語を整理した LOCA (Logic of Objects, Constraints and Associations) が提案されている。

さて、OCL に関連した最近の形式手法では、Alloy が面白い。MIT の D. Jackson が開発した形式仕様言語であって、Z 記法の基本的な部分に OCL が持つオブジェクト指向概念を追加した新しい言語として考案された。ただし、言語仕様を単純化すること、形式検証を自動化すること、という 2 つの設計方針をたてた。

Alloy では、Z 記法と同様なスキーマ概念に似た言語要素を提供し、論理式から構成される操作や関数の機能仕様ならびにデータ定義に対する不変量を簡便に表現する。さらに、スコープを指定する有界モデル発

☆ Z 記法の Mondex 仕様記述文書ではリファインメントも行っている。噂では「とても大変だった」とのことである。

見という手法を用い、また、バックエンドの検証エンジンとしてSATソルバーを利用することで、Alloy記述の自動検証を可能にしている。

自動検証ツールを前提とする形式手法であるLightweight Formal Methodsは、Alloy開発に関連してD. JacksonとJ. Wingが整理した考え方である。Alloyは考えようによっては少々エキゾティックな数学的な枠組を用いている。そのため、残念なことに、慣れるのに時間がかかるという問題点があるが、自動検証できるので、とても面白い形式手法である。

筆者は、Javaセキュリティの基本的な仕組みであるスタック・インスペクションとこれを利用したJAASアクセス検査法モデルをAlloyで作成して解析した経験がある。また、筆者らが行った最近の適用事例を次節で紹介する。

3.3 静的な情報構造

UMLの代表的なダイアグラムにクラス図やオブジェクト図がある。クラスのような情報単位の関係 (継承、集約など) を表現する静的な構造は、ソフトウェアのデザインとして重要である。このような静的な情報構造は一種のグラフとして表現できる。ダイアグラムあるいは情報構造ごとに、ノードとエッジの意味が異なるので、各々に特化した解析方法が必要となる。逆に、静的な情報構造の表現と解析に特化した形式手法があるというわけではなく、既存の形式手法への埋め込みによる実現方法が多い。

3.3.1 UML記法の厳密な定義

UML 1.3あるいは1.4の検討が進んでいた頃、UMLダイアグラムの曖昧さをなくすために、形式手法を用いてクラス図等の意味を明確に定義することを試みたpreciseUML (pUML) という研究活動があった。先に紹介したZ記法やBメソッド、あるいは、高階論理の証明系HOLなどに、クラス図の要素を翻訳することによって、UMLの意味を明確にする。先に述べたOCLに厳密な定義を与える研究も含んでいた。

形式化を行う拠り所はOMGが標準化した文書である。一般に、標準化の文書は、設計書ではないので、全てを決定して記述しているわけではない。標準に準拠する製品やシステムに対して、最小を決めるだけで、多くの自由度を残す。さらに、UMLは数多くのダイアグラム記法を持つファミリー言語である。1つの開発プロジェクトで全てのダイアグラムを使用することはないだろう。プロジェクトに必要なダイアグラムだけを取捨選択して使う。目的に応じてダイアグラムの使い方を工夫し、ダイアグラムが互いにどのように関係するかも決める。すなわち、UMLを使いこなすということは、自分たちの目的に応じて、多様なUMLダイアグラムの使い方をカスタマイズしていくことである。

このように捉えると、形式手法の技術を用いて、UMLダイアグラムの意味を厳密に決めることは無理があるように思える。一歩進んで考えると、利用の方法に応じて、UMLサブセットを定義する能力が大切である。さらに、利用の目的に応じて形式手法を応用する能力が求められるということである。

3.3.2 フィーチャー・ダイアグラム

ところで、静的な情報構造は、UMLが規定するものだけでなない。必要に応じて、新しい記法も提案されている。そのようなもののひとつにフィーチャー・ダイアグラムがある。このダイアグラムを例として、筆者らによる静的な情報構造の形式化に関連する経験を紹介する。

フィーチャー・ダイアグラムはSPLE (ソフトウェア・プロダクトライン工学) で脚光を浴びている記法である。ダイアグラムであるために、記法や意味が明確に定められていない。一方、特定のフィーチャー・ダイアグラム記述に対して、何らかの解析を行いたい場合がある。複雑なフィーチャー・ダイアグラムでは

木を見て森をみず

になる。2つのフィーチャーの直接的な関係は自明であっても、多数が絡み合うと、全体として意図通りになっているか、意図しない排他的な論理関係が誤って混入していないか、等がわかりにくくなる。そこで、筆者らは、フィーチャー・ダイアグラムの自動検査ツールを作成した。

詳しくは別稿にゆずるが、フィーチャー・ダイアグラムの意味を定義し、Alloyに埋め込んで解析する方法を採用することで、非常に小さな手間で実現できた。「フィーチャー・ダイアグラムのための形式手法」というような大げさなものではなく、Alloyを「デザイン電卓」として使う感覚である。

この作業の基本は、フィーチャー・ダイアグラムで書き表したい内容と解析の目的を整理し、これをロジックを用いて表現することである。Alloy記述に至るまでの「モデリング」が大切であった。

静的な情報構造が重要な役割を果たすソフトウェアのデザインは多数ある。各々についてもモデリングあるいは問題設定が明確にできれば、問題に合わせて最適な既存の形式手法を選べば良い。

ちなみに、「デザイン電卓」というのは、PVSの開

発者である J. Rushby が PVS を中心とするツール群の使い方に関連して用いた言葉で、ソフトウェアのデザインを対象とした「プログラマブル電卓」というイメージである。すなわち、少しコーディングすれば、電卓、この場合は Alloy、のすべての機能を活用して、デザインを、ちょこちょこっと解析することができる。

3.4 振舞い仕様

ここ数年、産業界でも利用の機会が増えているモデル検査ツールが、この分類項目の代表である。歴史的には、CMU の SMV が重要であるが、現在での代表的なモデル検査ツールというと、SPIN、NuSMV、FDR、LTSA である。

3.4.1 モデル検査法の成功

実をいうと、これほどまでに、国内の産業界でモデル検査法への関心が高まっていることには驚いている。特に、状態遷移の考え方がデザイン表現として相性が良い装置制御ソフトウェアなどの組込みシステムへの適用についての関心が高い。

筆者はモデル検査ツールとしては主に SPIN を使っているのであるが、SPIN の研究コミュニティと最初に接したのは FM'99[10] の時である。その後、EJB コンポーネント基盤の振舞い検証についての研究[12] を行った。SPIN の開発者 G. Holzmann は長くいたベル研から NASA に移った。教科書を刊行[13] し、利用者を広げる活動も行っている。

モデル検査法は別稿 15) の解説でも述べたように、CMU の E.M. Clarke たちが考案した時相論理の検証アルゴリズムにはじまる。昨年 (2006 年) は最初の論文からちょうど四半世紀にあたる年であった。これを記念して、シアトルで開催された CAV で、Symposium 25 Years of Model Checking (25MC) が開かれた。並行システムの仕様を表現するために時相論理を提案した A. Pnueli が E. Clarke と E. Emerson によるモデル検査アルゴリズムを暗黙に意識していたか、からはじまり、最近の SAT ベースの方法や CEGAR まで、研究の当事者から面白い話を聞くことができた。シンポジウムは大盛況であった。

ACM Kanellakis Award が 1998 年と 2005 年の 2 回、合計 8 名のモデル検査法に関する研究者に贈られるなど、実用的な観点から、現在、もっとも評価の高い形式手法の技術である。

このような高い評価と広範な関心の理由は、工学的な観点から有用な自動検証ツールになったからと考えられる。モデル検査法の基本的な方法は、有限の状態空間を網羅的に調べて、別途与えられた性質が成り立つか否かを検査することである。成り立たないことがわかると、その理由となる反例を具体的に求める。反例を解析することで不具合の原因をさぐることができる。状態空間を効率良く表現し探索する方法の確立が鍵である。

CMU グループによる初期の成功事例は、SMV を用いての、ハードウェア、すなわち、論理回路の不具合検出であった。また、IEEE 標準として公開されていた Future+バスプロトコルに不具合があることを発見したことが大きなインパクトであった。

余談であるが、Alloy を考案した D. Jackson は、当時、CMU にいて、SMV が設計デバッグのツールとして有用であることをみていた。Z 記法が対象とする構造的な情報の表現と解析に自動検証ツールを持ち込むという目的で Alloy を開発したことは先に述べた通りである。すなわち、SMV の成功が Light-weight Formal Methods という考え方の動機となった。

SPIN の代表的な応用は当初、通信プロトコルの検証であった。いかにも、ベル研らしい。その後、交換機ソフトウェアの高度サービス機能を検証するツールへと向かった。交換機ソフトウェアは C 言語で記述されているため、プログラムを対象とするモデル検査である Software Model-Checking と呼ぶ技術が重要になってきた。

3.4.2 抽 象 化

モデル検査法を説明する言葉として、「ボタン一発の検証法」ということがある。これは、たとえば、B メソッドで避けることができない対話的な検証ではなくて、完全に自動的な検証を行うということを意味する。しかし、対象システムを状態遷移の考え方で表現すること、さらに、その状態空間が有限であること、という 2 つの条件がある。

第 1 の点については、検証したいデザインに依存する。たとえば、UML のステート図は、状態遷移ダイアグラムの拡張であるため、そのデザイン記述から状態遷移システムを抽出することができそうだと想像がつく。また、複数のコンポーネントが同時並行的に作動する分散システムの制御アーキテクチャでは、そのデザインとして状態遷移の考え方を適用することは自然であり本質をついている。手続き型で書かれている交換機プログラムから状態遷移システムを抽出するには何らかの工夫が必要となる。

第 2 の問題に関連して、対象の有限性を確保するため、また、計算機による取り扱いが可能な規模の状態空間におさえるために、設計者が対象を適切に抽象化する必要がある。しかし、抽象化によっては検証したい性質が消えてしまうことがあり、モデル検査ツール

が不具合を発見できないという問題が発生することがある。逆に、抽象化によって非決定性が増えて、本来はない不具合が出てくるという false alarm の状況もある。注意しなければならない。

Software Model-Checking は、Modern Model-Checking とも呼ばれる。本方式では、入力したプログラムを自動的に抽象化する方法を用いる。抽象化を自動化するために、バックエンドとして定理証明系を用いる。さて、抽象化が正しくない場合には増えた非決定性が悪さをし、モデル検査ツールは誤った反例出力を生成する。誤りであることを確認するためには、抽象化前のもとの記述に対する記号実行が必要となる。誤りであることがわかると、失敗の原因となった非決定性を除去するような詳細化を含む新たな抽象化をやり直す。

この「抽象化・モデル検査・確認」の作業を繰り返す方法を CEGAR (Counter Example Guided Abstraction Refinement) と呼ぶ。マイクロソフト研究所が開発した SLAM は Windows のデバイスドライバ C 言語プログラムを対象とした自動チェッカで、CEGAR の考え方を実用化した最初のツールである。

Software Model-Checking では、CEGAR が重要な技術であり、モデル検査ツールはバックエンドの検証エンジンの役割まで後退したのである。種々の抽象化法、CEGAR でも基本として用いられている述語抽象の方法などについては文献 8) が良い。

3.4.3 3 つの方式

モデル検査ツールの実現技術からみると、CMU の SMV は BDD というデータ構造を用いた記号モデル検査法が特徴である。ここで、「記号」と呼ぶのは、状態探索アルゴリズムが個々の状態を区別するのではなく、ある性質を満たす状態の集合を記号的に表現する方法を採用しているからである。さらに、「集合」を特性関数と呼ぶ方法で論理式として表現すると、モデル検査の方法は論理式の書き換えとして実現できる。論理式の表現と書き換えを効率よく実現するために考案されたデータ構造が BDD である。

一方、SPIN は性質検証のための状態探索を行いながら、検証に必要な状態空間だけを展開するという on-the-fly 法を用いた。この方法だと、有限の状態空間探索で不具合がみつかって停止するような場合、状態空間が無限となるシステムも取り扱えるという利点がある。他にも、検証する性質の特徴に依存するが、Partial Order Reduction 等の方法を導入して実用的なツールになっている。

なお、検証する性質表現のための時相論理は、SMV が CTL、SPIN が LTL、であって、互いに表現可能な性質が少し異なる。SMV や SPIN といったモデル検査ツールの基本的な仕組みについては文献 21) にわかりやすい解説がある。

さて、取り扱う状態空間が大きくなるにつれて、BDD の大きさが膨大になってきた。そこで、CMU グループは探索範囲の状態空間を限定する有界モデル検査法 BMC を考案した。多くの場合、不具合に相当する状態は初期状態からそんなに遠くない場所に存在するという経験に基づく。探索する深さを利用者が指定して、そこまでの範囲で不具合が見つかれば良いとする。不具合が見つからない場合、システムは検証性質を満たすと結論づけることはできない。探索範囲を広げることで不具合がみつかる可能性がある。BMC は記号モデル検査法の一種であって、状態空間を論理式で表現するというアイデアに基づき、バックエンドの SAT ソルバーで論理式の真偽判定を自動化する。

現在、記号モデル検査ツールの主流は、SAT ソルバーを用いる方式に移っている。SAT ソルバーの技術革新が進み、膨大な規模の命題論理式を取り扱えるようになったからである。当初は、CTL を対象として安全性検証を行う BMC であった。Software Model-Checking にも使われており、たとえば、C 言語を対象とする有界モデル検査ツールとして CBMC や VARVEL がある。

また、検証対象として、ハードウェアとソフトウェアの区別が小さくなっている。従来のハードウェア検証では回路図相当のネットリストを対象としていた。一方、最近は、ハードウェア記述の抽象レベルがあがり Verilog 等の高位設計言語を用いる。ビット単位のレジスタではなく、プログラミング言語と同様に変数を用いてデータ値を表す。Software Model-Checking で考案された方法を適用することが可能であり、たとえば、CMU では CEGAR ベースの統合検証ツール VCEGAR を開発している。VCEGAR 等のツールでは、先に述べたような「抽象化・モデル検査・確認」は自動化されており、利用者は抽象化に絡む難しさを知らなくても良い。ここ数年で急激に技術発展しているツールである。

最近のモデル検査ツールの代表格としては、SPIN の他に、NuSMV、FDR、LTSA の名前をあげた。SMV が発展した NuSMV では、CTL と LTL の双方を利用可能であり、さらに、有界モデル検査法も提供している。また、FDR と LTSA は共に、Hoare の CSP 系のプロセス代数[20] に基づく仕様言語を入力とする。FDR は CSP に忠実なツールで、CSP の意味論にし

たがった検証方法を採用する。一方、LTSA は有限プロセス表現に特殊化した FSP を入力とするツールであって、SPIN などと同様なラベルつき状態遷移システムによる形式化を採用している。

最後に、最近の SAT ベースのモデル検査ツールでは、LTL を対象とする方法を導入することで、検証性質を進行性まで広げることができるようになった。(有界でない) 通常のモデル検査を行う方法も研究され、表現力の高い SMT ソルバーを用いるツールも開発されている。今後の主流になると思われる。モデル検査法の重要性の高まりと共に、ここ数年で大きく発展している。

3.4.4 適用の実際例

筆者は先に述べたように専ら SPIN を使っている。モデル検査したい対象が分散システムであることが多いというのが理由であろうか。SPIN は通信オートマトン間のメッセージ通信機構として、同期ハンドシェイク型ならびに非同期バッファ通信などを用いることができる。したがって、分散システムの表現と解析に向いている。また、一度使いはじめるとノウハウが蓄積し、「デバッグ法」も習得できるので、さらに使い易くなる。

EJB コンポーネント基盤の振舞い検証についての研究で行ったことを整理すると次のようなことである。すなわち、EJB ドキュメントを頼りに SPIN の形式仕様言語である Promela を用いて EJB サーバを作成し、その上で、ドキュメント記載の満たすべき性質を LTL で表現し検証した。もとのドキュメントに不備があることを見つけた。

その後は、他言語あるいは設計表現を Promela に変換してモデル検査するという使い方をした。たとえば、Web サービス連携記述言語である WS-BPEL で書かれた一種の分散協調システム記述を Promela に変換した。WS-BPEL プログラムの検証であるため、Software Model-Checking で用いられていた抽象化の技法を導入する必要がある。OASIS 仕様書記載の例題をすべて検証できることを示した。

また、状態遷移ダイアグラムの操作的な意味を実験するために SPIN を用いたこともある。たとえば、UML ステート図を用いる場合であっても、その詳細な操作的な意味が UML 文書で細部に至るまで厳密に決まっているわけではない。また、記述対象によっては標準の規則に少し変更を加えて記述の実験を行いたい場合がある。このような場合、自前で状態遷移システムの特殊なインタプリタを作成するよりも、Promela に変換し SPIN 上で振舞いを解析するほうが小さな手間で実現可能である。いわば、SPIN を状態遷移系デザイン記法のラピッド・プロトタイピング基盤に利用するという方法である。

アスペクト概念を導入したステート図の実験なども容易に行うことができた。もちろん、実行だけでなく、モデル検査法を用いたデザイン検証も行った。

モデル検査法については、ここ数年、国内で急激に関心が非常に高まった。2001 年 10 月に最初のチュートリアルを行って以来、学会主催のもの[14] を数回担当した。SPIN を中心に、モデル検査法の概要と、どのような応用事例、適用事例があるかを紹介した。特に、モデル検査ツールが何も神秘的なものではないことを伝えたかった。

我々は、コンパイラを、その中身を知らなくても使っている。構文解析、フロー解析、最適化、レジスタ割り付け、コード生成、など、コンパイラの大まかな動きは理解しているが、その詳細な仕組みは知らない。ブラックボックスとして使う。

モデル検査ツールについても、状態空間の生成、状態空間の網羅的な探索、など、最小限のイメージを頭に浮かべることで、ブラックボックス化して使うことができる。ブラックボックス化して使って差し支えないのである。

3.5 リアルタイム性

モデル検査法の適用分野として組込みソフトウェアをあげることが多い。しかし、組込みソフトウェアのようなリアルタイム性を持つソフトウェアでは、設計段階での時間特性に関する考察が大切である。先に述べたモデル検査の方法ではリアルタイム性を扱うことができない。論理的な時間関係、すなわち、イベント等の発生順序の前後関係だけが扱える。

3.5.1 時　　間

リアルタイム性という場合には、数値的な時間、メトリック時間を扱う。たとえば、タスクのスケジューリングに関係してデッドラインを満たすか否か、メッセージの通信遅延がタスクの振舞いにどのような影響を与えるか、など、設計段階で確認しておきたいことが多い。

一般に、「時間の概念」は多様である。どのように考えれば良いのであろう。

> 「時間とは何か。と人に訊かれなければ、私はそれを知っている。しかし人に訊かれれば、私はそれを知っていない。」(聖アウグスチヌス)

リアルタイム・ソフトウェアの設計検証という目的に

とって、どのように時間を理解すれば良いかを考える必要がある。特に、設計者である人間が感じる「心理学的時間」とシステムが作動する環境の「ニュートン式物理学的時間」がある[22)]。

心理学的時間とは、事象の前後関係の議論に関わり、「過去」から「未来」へ一方向に流れる。特別な事象として、暦、カレンダーなどがあるが、メトリック時間とは限らない。また、一様に流れるわけではない。先に述べた時相論理は処理の進行順序に関係する性質を議論する体系であり、心理学的時間と関係が深い。

一方、ニュートン式物理学的時間、あるいは「力学的時間」は、決定論的、絶対的、可逆的、である。すべての対象に共通に、連続的かつ一様に流れる。「一様」という性質を表現するために、本質的にメトリック時間である。力学的時間を基礎として運動を表現するという意味で、ニュートン運動方程式が記述する運動の「運び手」である。

システムの振舞いは日常の生活空間であるニュートン世界で説明できるとするので、力学的時間によってリアルタイム・ソフトウェアの設計検証を考えたい。アインシュタインの相対論的な時間も、ミクロの世界を対象とする量子論的な時間も、ここでの目的に合わない。

心理学的時間と物理学的時間の向きは一致すると仮定する[22)]。すなわち、リアルタイム性と時相論理の性質を、どこかで統合して議論することができるとする。もってまわった言い方であるが、前節の振舞い仕様で論じることが可能な排他的な共有資源の制御とアクセス側タスクのリアルタイム・スケジューリングや通信遅延によるタイミングのずれなどの問題を同時に取り扱うということである。

3.5.2 リアルタイム・ソフトウェア

リアルタイム性に関しては、個別の問題を解決するための技術が開発されてきた。便宜上、代表的な技術として、リアルタイム・スケジューリング、性能解析、リアルタイム振舞い検証、に分けることにする。

リアルタイム・スケジューリングは、リアルタイム制約を守るためのタスク・スケジューリングに関する技術である。特に、組込みシステムのような無停止連続実行タスクが対象であり、タスクは周期実行されるとすることが多い。

タスクは、周期、最悪実行時間、デッドライン時間、等からなる時間特性によって定義されている。スケジューリングの問題とは、タスク群がデッドライン違反を起こさないような実行順序計画を立案することである。同時実行要求するタスクが複数ある場合の優先度の与え方によって、RM、EDF 等のスケジューリング・ポリシーが知られている。

スケジューリングの基本的な方法では、タスクは、CPU という資源を取り合うが、独立に作動すると考える。一方、実際には、タスク間に実行順序関係や共有資源への排他制御などが関係する。このような論理的な制約関係を考慮する場合であっても、時間特性を詳しく定義することで古典的なスケジューリングアルゴリズムを拡張する方法を採用する。

リアルタイム・スケジューリングを取り扱う場合、時間は連続である必要がない。タスク切替え時点に注目した離散時間を扱うだけで解析可能である。

2 番目の性能解析の問題は QoS に関係する。一定時間内にどのくらいの外部発生イベントを処理できるか、等を扱う。歴史的には、待ち行列を用いた解析や統計的な入力データに対するシミュレーションの技術が用いられてきた。重要な観点であるが、現在のところ、形式手法との関連は薄い。

なお、リアルタイム・ソフトウェアのモデリング向けに提案された UML/SPT プロファイル (UML Proflie for Schedulability, Performance and Time Specification) では、時間に関連した資源モデルと 2 つの解析法を明確に分離している。このことからも、スケジューリングならびに性能解析の問題に関しては、既知の手法を用いるというアプローチが一般的であることがわかる。

3.5.3 リアルタイム振舞い検証

リアルタイム性の解析に形式手法が関係するのは、3 つめのリアルタイム振舞い検証の問題である。1990 年頃から、メトリック時間を考慮した時相論理や時間オートマトンの理論が研究されてきた。先に述べたように力学的時間に基づくため、実数で表現される連続時間を取り扱うことが基本である。

メトリック時間を入れた時相論理では、未来のある時点において生起するイベントが具体的なクロックの値として何時起こるかを指定することができる。これによって、決められたデッドラインまでにイベントが起こるか否か等を表現し解析することを目標とする。基礎とする時相論理によって、Timed CTL、Timed LTL などがある。

また、検証対象システムの表現に関してはオートマトンの考え方に基づく理論が考案されている。その中で、現在、標準的なものは、Alur-Dill 時間オートマトンであろう。特別なクロック変数と呼ぶ概念を用いて、(力学的) 時間の経過を表す。システムの時間的な特性としては、クロック変数値を用いた時間制約式に

よって、状態滞留時間幅ならびに遷移ガード条件などを表す。これによって、計算処理にかかる時間や、タイムアウトなどの状態遷移を起こす時間条件を表現することが可能になる。

素朴に考えて、リアルタイム性が関係すると、モデル検査が難しくなることがわかる。モデル検査の基本的な考え方は、状態空間を網羅的に探索することであった。一方、連続時間に基づくリアルタイム性が入ると、微小時間だけ経過した時点が異なる「状態」に対応することになり、したがって、無限の状態空間を対象にすることになる。微小時間の経過という問題は力学的時間の本質である。

そこで、Alur-Dill 時間オートマトンでは、リージョンと呼ぶ方法を用いて、無時間化した (untimed) オートマトン に帰着させる方法を考案した。リージョンとはクロック変数の時間区間を組み合わせて表現した多次元時間区間のことである。ここで、次元はクロック数に一致し、クロック数と区間数が多いとオートマトンは大規模化する。

ある時間オートマトンが与えられたとき、以降の振舞いが同等となるような、すなわち区別できない「時間状態」からリージョンを構成する。リージョンを状態、リージョンが表す多次元時間区間からの出入りを遷移、とするような新たなオートマトンをつくる。このリージョン・オートマトンを対象としてモデル検査を行う。リージョンが有限個であれば、通常の有限オートマトンに対するモデル検査法を適用することが可能になる。

リアルタイム・モデル検査ツールについてはいくつかの研究があったが、現在も整備されているのは UPPAAL であろう。UPPAAL は Alur-Dill 理論をもとに、進行性に関する操作的な意味の取り扱いを簡素化した Timed Safety Automata を採用する。また、リージョンをクロック制約で記号的に表現したゾーングラフを考案し、ゾーン計算を効率良く行う DBM の導入により実用的なツールになった。

一方、検証性質の表現に際して、Timed CTL のサブセットしか使えないこと、監視オートマトンを導入しなければならない場合があること、などいくつかの方法を組み合わせる工夫が必要になる。また、時間オートマトン間の通信機構はハンドシェイク型だけであるため、通信遅延、割り込み等を表現したい場合、そのような機構を表現した時間オートマトンを新たに定義、導入するなどの繁雑さがある。

リアルタイム振舞い検証では、時間経過と同時に、通常の振舞い仕様でも取り扱う論理的なイベントの両方を考える必要がある。UPPAAL の場合であれば、通信機構を用いたインタラクションが論理的なイベントに相当する。この時、時間進行に関係しないインタラクションが発火可能な場合、常にインタラクションが瞬時進行する Maximal Progress を仮定する。

ところが、この瞬時イベントが連鎖的に無限に発生すると、時間進行をみることができない Zeno 性を示すことになる。リアルタイム性を考えない振舞い仕様であれば、瞬時規則による無限列は飢餓状態であるかもしれないし、あるいは、意図通りの処理進行であるかもしれない。時間特性を考慮する場合は、システム記述の不具合であるとしてよい。

VDM の項で触れた VICE は、Validation through Animation の考え方にたつ実行可能設計言語である。時間特性の解析を行う際、テストデータを与えた実行履歴を収集しておき、履歴情報を対象とした解析ツールによって評価する。種々のスケジューラが提供されること、基本的な式の実行時間見積りについてデフォールト値を用いれば仕様作成者が与えなくても良いこと、など、手軽に利用するための便利な機能が提供されている。しかし、あくまでも設計記述のテスト実行による妥当性確認であることに注意すべきである。

3.5.4 同期型言語

力学的時間を用いないが、リアルタイム性と関連の深い方法も提案されている。すなわち、メトリック時間を扱わない体系である。自然数で表現する離散時間に基礎をおくといってもよい。たとえば、クロック同期式順序回路のように、基軸となる信号の変化が離散的であるようなシステムの表現と解析では有用である。ただし、順序回路の解析といっても、ある組み合わせ論理回路の遅延時間がサイクル設計値で収まるか否かといったタイミング検証の問題については、当然であるが、連続時間の体系が必要となる。

順序回路のクロック同期という考え方を一般化したのが、同期型言語 (Synchronous Language) である。同期型言語は、組込みシステムのような外部との因果関係の強い「リアクティブ・システム」を想定したもので、必ずしも「リアルタイム性」を表現し解析する道具ではない。

代表的な言語としては、Esterel と Lustre がある。1980 年代からフランスを中心に研究、産業界への適用が進められている ☆。Esterel は手続き型言語であり、Lustre は同期式データフロー計算モデルに基づ

☆ モデル規範型の VDM と Z 記法は英国が中心である。モデル検査法は北米が強い。ドイツ語圏はロジックに強く基礎的な検証ツールに面白いものが見られる。

く。現在では、これら 2 つを含む幾つかの同期型言語を統合して SCADE と呼ぶ開発支援環境になっている。特に、D. Harel の Statecharts から派生した階層型オートマトン SyncChart と階層型データフローを統合したデザイン言語を提供している。なお、UML ステート図も操作的な意味は同期型言語であり、そのために、UML ステート図は単に階層型のオートマトンというわけではない[15]。

同期型言語に関わるツールは、デザイン検証よりも、プログラムの自動生成に重きを置いている。Lustre を中心とする SCADE 開発環境や、UML ステート図関連の代表的なツール Rhapsody は、C 言語あるいは Java のプログラムを自動生成する。実は、UML ステート図のもとになった Statecharts でも、実行概念を表す操作的な意味は定義されているが、モデル検査に用いるような検証のための規則は整理されていない。一方、SCADE は、SAT ベース有界モデル検査のエンジン ☆をバックエンドに持つことで、形式検証の技術を取り込んでいる。

先の順序回路の場合もそうであるが、一般に、同期型言語では、同時性仮説 (Synchrony Hypothesis) を採用する。何らかの論理的な処理、順序回路であれば組み合わせ論理回路による処理、Maximal Progress の仮定では瞬時インタラクションによる処理、の実行中、システムがおかれた外部の状況は変化しないとする仮定である。

同期クロックをきっかけとして開始した処理は、その時点で外部から得られた情報をもとに処理を進める。したがって、当該処理中に、外部の状況が大幅に変わると、処理結果が意味をなさない。このようなことが起きないということは仮定でしかない。しかし、外部の変化周期に対して、CPU の処理速度が数桁高速であれば、十分に満たされる仮定である。すなわち、適用対象システムによっては、同時性仮説は工学的に妥当なものとされている。

3.5.5 Right-weightness

本節では、リアルタイム性の取り扱いについては、考えたい問題に応じて、時間の考え方が異なることを強調した。時間オートマトンを用いてタスクのスケジュール可能性解析を行う研究もあるが、リアルタイム・スケジューリングの問題は、離散時間で十分に扱うことができる。筆者は、同期型言語の操作的な意味をベースにして離散時間を表現する中間表現を導入し、この操作的な規則を時間概念を持たない SPIN 上で作成することで、排他的利用の共有資源があるタスク群のスケジュール可能性判定を行った。

☆ ハードウェア検証向けに開発された検証エンジンである。

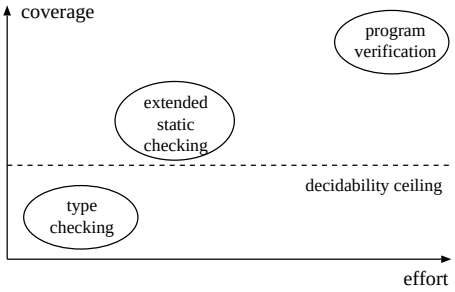

図 2 プログラムの拡張静的検査

リアルタイム振舞い検証を行うためには、時間オートマトンのモデル検査ツール UPPAAL が有力であり、いろいろな問題を取り扱うことが可能なことが知られている。一方、時間オートマトンは表現力が大きいと同時に、モデル検査の解析コストが大きい。「牛刀割鶏」にならないように、問題にあった適切な方法を考えるべきであろう。

筆者は、リアルタイム振舞い検証に関連するいくつかの実験を行ったことがある。何度か述べたように、リアルタイム振舞い検証は本質的に難しい。正しさの確認手段として使うよりも、設計段階での時間特性に関する検討手段として用いることが実際的であると感じている。形式手法の技術を設計デバッグに用いるという Light-weight Formal Methods の考え方である。しかし、リアルタイム振舞い検証の場合、問題が難しいだけに自動化も難しく、したがって、種々の手法を組み合わせる必要もあるだろう。Right-weight Formal Methods (RFM) というべきかもしれない。興味深い問題が山積みになっている研究テーマの宝庫ともいうべき領域である。

3.6 プログラム検査あるいは検証

形式手法の研究初期に、プログラム検証が重要なテーマであり、Hoare-Dijkstra 理論など活発な研究があった。長い間、これらの研究は理論的な面に終始し、他の形式手法のような実適用に発展してこなかった。最近になって、Hoare-Dijkstra 理論に基盤をおくが、より軽い使い方を目指したツールが開発されるようになった。以下に、Java や C 言語を対象とするプログラム検査あるいは検証のツールを紹介する。

3.6.1 拡張静的チェッカ

図 2 はプログラム検査あるいは検証ツールの位置付けを説明する。縦軸に検出できる不具合の広さ、横軸に検査のためにかかるコストをとった。左下のタ

イプチェックと右上のプログラム検証の中間に位置するツールとして拡張静的チェッカ (Extended Static Checker) がある。

タイプチェックの利点は、 コンパイラの一部機能として、自動的に行うことにある。利用者は何らタイプシステムに関わる理論的な詳細を知る必要がない。他方、プログラム検証では基礎とする論理体系に応じて多様な性質の検証を行うことができる。その反面、自動検証は困難であり、基礎となる論理体系を熟知して、対話的な証明による検証を行うことになる。

一方、タイプチェックに通っても実行時に発生する代表的な不具合として Null 参照、配列の添字範囲エラー、などがある。これらの不具合は静的な解析だけでは一般に決定不能であることが知られている。しかし、理論的な限界が影響するようなプログラムはよっぽどの事がない限り書かない。「病的な」プログラムといってもよい。一般のソフトウェア技術者が作成する通常のプログラムに対しては実用的な性能で自動チェック可能なツールを開発できると期待される。

拡張静的チェッカはコンパイラと同様な使い方でありながら、タイプチェックよりも幅広い不具合を検出可能な高度なデバッグ支援ツールの実現可能性を確認するために研究された。

3.6.2 ESC/Java2

ESC/Java2 は Java 言語を対象とする拡張静的チェッカである。もともと、DEC/SRC での ESC/Modula-3 の研究からはじまり、1990 年代半ばに K.R.M.Leino が Ph.D. 研究として ESC/Java を開発した。同時に、外部の研究者にも公開され, いくつかの先導的な研究プロジェクトでの実適用が試みられた。その後, DEC/SRC の閉鎖などにより、ESC/Java の研究主体が移り変わり、現在、ESC/Java2 は UCD の J. Kiniry が中心になっている。ESC/Java2 では、プログラムの仕様を表現するための注釈言語として JML(のサブセット) を用いるので、JML 周辺ツールのひとつとして数えられる。実は、JML 自身、ESC/Java で考案された注釈言語から影響を受けて設計されたものであり、互いに影響を与えながら成長している。

ESC/Java は、Design by Contract の考え方にしたがって、メソッドの機能仕様を事前・事後条件の組として、JML の式で表現する。検証の基本は、E.Dijkstra の最弱事前条件に基づく方法による。ESC/Java2 内部で Java 言語、実際は内部表現であるガードコマンド言語、の公理的な意味を持ち、事前条件の仮定下、Java メソッド本体が、事後条件を満たすことを検証する。検証条件を表した 1 階述語論理の式をバックエンドの定理証明ツール Simplify に与えて自動検証を試みる。ひとつの Java メソッドは大きな規模にならないので多くの場合の検証処理が終了する。

ESC/Java の面白い点は、Java プログラムの実行意味を与える背景述語を用いること、必要な検査表明を適宜プログラム中に挿入することである。たとえば、配列境界違反を検出するような検査表明を配列アクセス箇所に埋め込む。配列に関する一般的な知識を表現した背景述語を仮定の一部として持つ検証条件を生成するので、検証時に、埋め込んだ検査表明のチェックを行うことができる。このように、事前・事後条件を全く与えない場合でも、基本的な検査を行うことができることが面白い。

ESC/Java2 は、クラス継承によるサブタイプ関係や、例外機構もサポートしている。形式手法に関する知識が全くなくても気軽に使うことができる。一方、Design by Contract の考え方を前提としてメソッド単位で検査するため、不必要な警告 (false alarm) も多い。Light-weight Formal Methods のツールである。文献 17) に解説がある。

3.6.3 C 言語向けの Caduceus

ESC/Java と同様な狙いで C 言語を対象として開発されているツールに Caduceus がある。詳しくは、文献 19) の解説を参照されたい。

Caduceus も、ESC/Java と同様に、Hoare 論理の枠組に基づいたプログラム検証を行う。JML に基づいて C 言語向けに設計された注釈言語を用いて、関数の仕様を事前・事後条件で与える。また、ループに対しては、適切なループ不変量と変量を与える。ANSI C の広い範囲の言語仕様を取り扱うことが可能であり、C 言語に特有なポインタ演算を含むプログラムも対応可能である。サポートしていない機能としては、goto、関数ポインタ、ポインタのキャスト、などである。また、事前・事後条件で与えられた関数の仕様に関わる検査の他にも、不正な操作 (配列の添字エラー、ゼロ割算、Null ポインタ参照、等) の検査ができることも、ESC/Java と同じである。

Caduceus が、ESC/Java と異なるのは、対話的なプログラム検証を併用することで、検査結果の健全性を保証できる点にある。Caduceus が生成する検証条件はバックエンドの定理証明ツールを用いて検証する。簡単なものであれば、Simplify によって自動証明させることができる。一方、注釈言語によって表現できない仕様の場合は、Caduceus が生成した検証条件をバックエンドの定理証明ツールを用いて、利用者が対話的に証明する。必要に応じて、仕様を書き表すための基

本的な述語の定義を、ホストの定理証明ツール上で与える。ホストの定理証明ツールにもいろいろな特徴があるので、Simplify、Coq、Isabelle/HOL といった複数の定理証明系と連動可能になっている。

3.6.4 The Verifying Compiler

ESC/Java2、Caduceus 以外にも、Hoare-Dijkstra 流のプログラム検証の枠組を基礎とするプログラム検査および検証ツールが開発されている。研究的要素の強い発展中のツールが多いが、産業界での適用事例も増えてきている。使い方を工夫すれば、実用的なプログラムに対して適用可能な技術レベルに達しているといってよい。

ESC/Java2 はプログラム検証ツールではないため不具合がないことを保証することはできないが、デバッグ支援ツールとしては面白い。学部学生の授業で ESC/Java2 を使ったことがあるが、ツールを使うことで背景にある理論的な側面に興味が向いた学生もいた。形式手法の入門ツールとしても有用であろう。

また、マイクロソフト研究所では、ESC/Java2 と Caduceus とは少し異なるアプローチによって、C♯ を拡張した Spec♯ を研究している。ESC/Java2 と同様に自動検査ツールを目指すが、すべてを静的な検査に頼るわけではない。Design by Contract の考え方だけでは取り扱いが難しい「大域的な不変量」の計算などを実行時に行う動的検査を併用する。

さらに、個別の性質ごとに実用的な自動解析ツールを開発する研究も増えている。Windows 系 OS を対象としたもので、バッファオーバーフロー等の不具合解析に特化した SAL や、決定不能問題の代表として教科書に載っている「プログラムの停止性判定」を行う TERMINATOR などがある。いずれにしても、Hoare 卿が目指す IEE グランドチャレンジの目標である The Verifying Compiler は、ここで紹介した技術の将来像であるかもしれない。

3.7 その他の観点

形式手法の研究を分類するにあたって、「モデル規範型」、「性質規範型」、「インタラクション型」の 3 つを考えることがある。「モデル規範型」については既に解説した。「インタラクション型」は、Tony Hoare の CSP、R. Milner の CCS やπ計算、などのプロセス代数が入る。本稿では、プロセス代数は理論的な枠組であって、形式手法の基礎を与えるものと考えて説明を省いた。モデル検査法に関係する検証技術と関係が深い。文献 20) にプロセス代数の解説がある。ここでは、残った「性質規範型」である代数仕様言語について少し考える。

今までの解説から、多くの形式検証ツールが、バックエンドとして、高階論理、1 階述語論理、あるいは命題論理の証明系を用いていることがわかる。形式仕様言語の記述を論理系に翻訳して扱うからである☆。高階論理の証明系として Isabelle/HOL、Coq、PVS、などがある。PVS については文献 7) に簡潔な解説がある。1 階述語論理の自動証明系として Simplify が、命題論理に関しては SAT ソルバーが使われることが多い。また、B ツールのように、自身の中に専用の 1 階述語論理の自動証明系を持つものもある。

いずれにしても、形式手法を利用する技術者からみると、バックエンドの証明ツールを目にすることはあまりない ☆☆。本稿では、代数仕様言語も、上記のような証明系と同様に、他の形式手法ツールのためのインフラと考える。

3.7.1 代数仕様の考え方

代数仕様の性質規範という考え方は次のようなものである。記述対象を特徴付けるために必要な記号を導入し、その記号の間に成り立つ関係を定義することで、記号の意味を与える。記号と記号の間の関係を「等式仕様」と呼ぶ方法で表現した。

筆者が利用経験のある代数仕様言語は J. Goguen の OBJ 系である。代数仕様言語は、後に述べる抽象データ型の考え方と密接な関係にある。リストや木構造の抽象データ型を帰納的に定義し、等式仕様によって解釈を与える。等式仕様を左辺から右辺への書き換え規則と見做すことで「実行可能」仕様を得る。

その後、隠蔽代数と書き換え論理という 2 つの方向に研究が発展した。隠蔽代数に基づく形式仕様言語の代表は CafeOBJ であり、書き換え論理では Maude がある。CafeOBJ は書き換え論理も含む体系を実現している。詳しくは開発者の二木他による解説 18) を参照されたい。

Maude は J.Meseguer が考案したメンバシップ等式論理と書き換え論理に基づく代数仕様言語である。OBJ の等式論理を拡張したメンバシップ等式論理を用いて抽象データ型を表現できると同時に、書き換え論理によって状態遷移システムを表すことができる。さらに、Maude は、幅優先探索に基づく到達性解析法や LTL のモデル検査ツールを提供し、書き換え規則が生成する状態空間を解析することを可能にしている。

3.7.2 代数仕様の重要性

代数仕様のこうした特徴によって、ソフトウェアが

☆ まさに「数理論理学」である。
☆☆ 使うだけであれば「数理論理学」の素養な最低限でよい。

処理する多様な対象や概念を表現しやすい。ある操作の特性を表現するために、適宜、新たな記号を導入して抽象的に表すことができる。そのため、単に手続きや関数、あるいはデータ構造といった言語要素を用いる以上の柔軟さがある。特に、「抽象データ型」ならびに「モジュール・インタフェース」といったソフトウェアの基本的な重要概念を生み出した。

抽象データ型は、データ構造の具体的な実現方法に立ち入ることなく、そのデータ構造に対して操作可能な処理によって抽象的にデータを定義する考え方である。

抽象データ型は、内部の実現詳細と外部からの使い方であるインタフェースを明確に分離する重要な考え方である。インタフェースによって情報隠蔽の考え方が明確になる。特に、代数仕様の方法では、インタフェースに登場する記号の間の関係を等式仕様で表現できるので、単に手続きや関数のシグネチャを規定する以上の表現力がある。

情報隠蔽が重要であることは、古くからあるモジュール設計法やオブジェクト指向プログラミングでも指摘されている。一時期、「オブジェクト指向プログラミングは抽象データ型に継承概念を導入しただけである」、という説明がされた時代もあったくらい、代数仕様言語とオブジェクト指向プログラミング言語の関係は深い。実は、オブジェクト指向プログラミングは抽象データ型の考え方だけではとらえきれない側面を持つ。この辺りの事情については別稿 11) を参照して頂きたい。

3.7.3 適用の経験

筆者は、代数仕様言語を用いて、あるシステムの仕様を直接作成することはなく、中間的なデザイン記法を考案し、その意味を代数仕様言語で与えるという方法を採用した。すなわち、代数仕様言語を他デザイン記法向けツール構築基盤として使ったのである。

かなり昔になるが、オブジェクト指向デザインを形式手法によって厳密に表現し解析することに関心があった。特に、オブジェクト指向フレームワークの作り方が、多相性、情報隠蔽、クラス継承などをうまく利用した方法であり、最も、「オブジェクト指向」らしいと感じていた。そこで、その本質的な側面である「集団的な振舞い」を書き表す記法 GILO (Generic Interaction Language for Objects) を整理し、そのデザイン記述を CafeOBJ 上で実行させて Validation に用いた。

最近、「制約オートマトン」と呼ぶ考え方を Maude 上で構築した。モデル検査法が対象とする状態遷移の考え方は複雑な制御の流れに着目した振舞い仕様の解析に向いている。一方、データ値の計算や静的な構造の関係などを取り扱うことが不得意である。そこで、データ間の構造的な関係を記号的な制約として表現することで、既存モデル検査法の短所を補う方法を考案した。Maude の機能を活用することで、制約オートマトンを簡便に実現することができた。

以上、筆者の経験を述べたが、CafeOBJ を用いた検証事例[18] でも、いわば中間的な設計表現である OTS を導入することで複雑なプロトコルの表現と検証を行っている。代数仕様言語を用いる場合、取り扱う問題の分析を行って、問題を簡便に表現する言語を導入する方法、「言語パラダイムによるシステム検証」の方法が好ましいと感じている。

本稿で多くの形式手法を、筆者の経験を補いながら紹介した。実は、代数仕様言語が最も「好きな」形式手法である。取り扱う問題に合わせて柔軟に表現できる、という点がうれしい。

4. 形式手法の導入

次に、形式手法をソフトウェア開発に導入する際の注意点を考えてみたい。組織への導入、個人の技術習得について整理し、最後に、筆者の所属機関の活動を紹介させて頂く。

4.1 組織への導入

元来、形式手法はソフトウェア開発の方法を大きく変える。すなわち、開発パラダイムの変革を伴う技術である。究極の目標である Correctness by Construction はソフトウェア開発の全工程を形式手法で置き換えようとする。この方法が現実的でなかったため、Formal Methods Light の立場で、少しずつ使っていく試みが成功を収めた。しかし、単に開発上流工程でのデザインを厳密に作成する、という言い方では、具体的な形式手法の使い方に言及していない。実際、文献 3) に述べられているように

開発プロセス全体の見直し

が成功の鍵である。形式手法をどのように使えば最も効果的か、は、個々の開発プロジェクトごとに考えていかなければならない。したがって、形式手法の使い方に経験を有するコンサルタントの存在なしに、形式手法導入を考えることは大きな賭けとしか言えない。

「すこしの事も先達はあらまほしきことなり」
（徒然草）

である。

さて、開発パラダイムの変革を必要とすることは、何も、形式手法に固有の特殊な事柄ではない。実際、オブジェクト指向技術の導入が始まった 15 年くらい前にも同様な議論があった。

ご存知の通り、オブジェクト指向技術の本質は、どのようにシステムを理解するかである。そのため、構造化設計や機能分割の方法とは全くことなる「オブジェクト」の考え方を理解しなければならない。単に、オブジェクト指向プログラミング言語を使えば良いという問題ではない。さらに、モデリング品質を向上させるために、開発上流工程に工数がかかる。その結果、従来の開発経験から予測できないほどの工数が必要となり、プロジェクト進捗の見通しが得にくいという意見が多く出た。オブジェクト指向技術のひとつの長所とされた「再利用性」のご利益を享受するためには、単独プロジェクトでの結論はありえない。関連する複数のプロジェクトまで考えた全体的な戦略を持つことが大切であった。

プロジェクト・マネージャからみると、オブジェクト指向技術という新しい方法を採用することは、ご利益を享受するよりは、新たなリスク要因を抱え込むだけであり、まさに避けるべきことが正しかった。実際、単独プロジェクトを超えた範囲で、コストやリスクを考えるトップレベルの意志決定が必要であった。形式手法の導入も、当時のオブジェクト指向技術と同様に、戦略的な意志決定が必要である。

形式手法の採用が常に戦略的な意志決定を必要とするかというと、必ずしもそうではない。開発に携わる技術者に訴えかけるという方法がある。形式手法初期に目指した Correctness by Construction でなくてよい。Formal Methods Light や Light-weight Formal Methods の考え方で、形式手法ツールを用いて、少しでも品質が向上すればよいではないか。「デザイン電卓」としての気軽な利用、完全な検証はできなくてもプログラム検査ツールを用いてコードレビューやテスティングの手間をおぎなえばよい。手持ちの仕事が楽になれば、これにこしたことはない。

逆に、形式手法を採用したくない場合、開発の責任を担っているプロジェクト・マネージャに相談するのが良い。先にも述べたように、新しい技術の採用は新たなリスク要因を抱えることと同じである。

「形式手法を採用しない 100 の理由」をたちどころに報告してくれるであろう。

4.2 技術習得

形式手法の技術を習得しようとする場合、「数理論理学さ」が禍すると言われる。しかし、図 1 に示したように数理論理学的な面 (言語ならびに証明技法) に加えて、設計手法、ドメイン知識、などが関係する総合的な技術である。すなわち、「数理論理学さ」は重要な要素であるが全てではない。

形式手法を使ってみて、うまくいかない時、それは、「数理論理学さ」ではなく、対象を理解していないという技術者側の問題であることが多い。

UML 等のダイアグラムを用いる場合、スケッチで十分であるため顕在化しなかった理解不足が露呈するのである。また、小規模なトイ問題の場合、取り扱う問題が明確で単純であるため、理解不足の問題が出てこない。実用的な問題にアタックする時に問題点が露呈する。これをもって、「形式手法は実用規模の大規模問題に不向き」と結論つけるのは問題のすり替えである。逆に、形式手法の中心である形式仕様言語によって、対象を書き下す際に、理解不足を認識することができる。これを改善する努力をすることによって、対象システムをきちんと理解することが可能になる。

実際、先の形式手法ワークショップのパネル討論では、形式手法の研究者でない限り、利用者の立場で関わるのであれば、初歩的な「集合、論理、オートマトン」の知識で十分であるという意見が多かった。特段の数学的な素養が必要とされるわけではない。理科系の課程を終了した技術者では学習ずみである。

一方、多くの場合、授業時間数の制約から、「集合、論理、オートマトン」の基礎は教えられていても、ソフトウェア開発の、特に、形式手法の技術と、どのように関わっているか、までは示されないのではないか、ということも話題になった。実際、一部の専門性の高いカリキュラムを除くと、「集合、論理、オートマトン」については、通り一辺の入門授業で終らざるを得ないのが実情であろう。たとえば、「物理数学」という科目があるように、「ソフトウェア工学向け数理論理学」が必要なのかもしれない。しかも、「物理数学」のように演習つきが好ましい。

ところで、ある形式手法を使うために、当該手法の形式仕様言語によって何かの記述を作成する場合、その手法あるいはツール特有の「デバッグ法」を会得する必要がある。「デバッグ法」を知らないと、仕様記述とエラー出力を目の前にして途方に暮れるばかりである。残念なことに、プログラミングの場合と異なり、「デバッグ法」まで載っている解説はめったにない。実

際のところ、これが最も大きな障壁のひとつである。上記、徒然草からの引用を思い返せ。

さて、図 1 に現れてない重要な事柄が残っている。それは、「対象の本質的な側面を抽象的に把握する能力」、抽象化の能力である。形式手法ワークショップのパネル討論でも話題になった。

パネリストの大方の経験では、プログラミング能力の高い人、わかりやすいきれいなプログラムを作成できる人は、形式手法の理解が早いことが多い。問題の本質を見極めて仕様として整理し、問題解決の方法をプログラムの形で具体化できるからであろう。

関連して、抽象化の能力を訓練するための良い方法があるかについても話題になった。学生時代の物理や化学で行った実験が、対象の現象を整理し抽象的に把握するための訓練になっている。経験をつむことによって、対象の本質を把握する能力が高まる、等が話題になった。

また、抽象化能力の訓練に LFM 等の検証ツールを用いるという J. Kramer の話もある。記述内容に対してツールのフィードバックを得て、これによって記述を改良していく。この「記述–検証–改善」サイクルを通して、対象を適切な抽象レベルで表現する訓練ができる、というものである。

さて、最近、プロジェクト管理の重要性が高まると共に、プロジェクト管理に関する資格試験が準備されている。素朴には、プロジェクト管理こそ、座学や筆記試験による能力判定が難しい分野である。同様に、ソフトウェア技術者に要求される「抽象化の能力」を調べることも難しい。しかし、形式手法の技術からの試験を行うことで、技術者の将来能力、技術ポテンシャルを調べることができるかもしれない。少なくとも、他の手段よりは、「マシ」と思える。残業時間ではなくて、形式手法の理解度によって、給料に差がつくようになったら、どうする?

4.3 ブリッジング

既に、過去、現在にわたる幾つかの研究について、筆者らの適用経験に触れた。筆者の立場は形式手法の世界では非常に特殊である。通常は、特定の形式手法を信じ、その適用可能性を広げること、あるいは、適用限界を調べること、を通して、当該形式手法の技術を確立していく。

一方、筆者の方法は、まさに、「彷徨える」ところにある。対象とするソフトウェアの性質、重要となるデザインの観点、は個々の開発対象によって異なる。このような多様な対象、すべてに適切な解を与える手法は存在しない。適用の局面を誤らなければ、極論すると、どの手法も役立つのである。ここでは、単に「手法」と呼び「形式手法」と言っていないことに注意されたい。すなわち、適材適所で使うということは、形式手法だけに必要なことではない。ソフトウェア開発手法一般のことを繰り返したにすぎない。

形式手法の世界、特に国内では、長い間、産業界と研究の世界とが遊離してきた。そのため、40 年近い膨大な研究の蓄積があるにも関わらず、いや、膨大な蓄積が逆に禍して、形式手法の技術を正確に理解し判断することを難しくしている。形式手法について産業界の関心が高まっている現在、必要なことは、2 つの世界の「ブリッジング」☆である。産業界と研究の世界のブリッジングであり、また、数理論理学とソフトウェア工学のブリッジングである。

筆者が所属する国立情報学研究所 (NII) は、大学共同利用機関法人という聞き慣れない種類の組織である。情報学に関する文部科学省関連の唯一の研究所として、ソフトウェアの研究、特に、形式手法に関する研究を精力的に行っている。「共同利用機関」であるため、NII では、他大学との共同研究ならびに産業界からの委託研究の制度を活用して研究を進めることが多い。また、NII の研究職員は、総合研究大学院大学 (SOKENDAI) の情報学専攻を担当している。そこでは、博士課程の学生を受け入れている。社会人学生の比率が高いことが特徴のひとつである。

我々は、ブリッジングの観点から、多種多様な形式手法の「使い所」を求めて彷徨っているのである。「邪道」かもしれないが、形式手法を使うことが目的ではなく、高い信頼性を達成するための手段である限り、ブリッジングは必要なことと考えている。特に、現在、筆者らのグループでは、国立情報学研究所のソフトウェアに関する戦略的な研究の一部として、

- 形式手法を用いたソフトウェア・プロダクトラインの研究
- システムセキュリティに関わる形式検証の研究
- リアルタイム性やハイブリッド性を持つ組込みシステムに関する設計と形式検証の研究

などに強い関心を持っている。

NII の本務は研究開発であって実務コンサルティングではない。しかし、形式手法の場合、研究と実務とが密接に関わる。気軽にコンタクトして頂ければ幸いである。

☆ 株式会社一 (いち) の大槻氏の言葉。

5. おわりに

書棚の奥から書き込み一杯の文献 1) が出てきたのをみると、20 年ほど前に形式手法の世界に迷い込んだようである。冬の時代というほどではないが、当時、形式手法は活発な分野というわけではなかった。その後、取り巻く環境が変わり、形式手法自身が変化してきた。筆者は今も形式手法の世界を彷徨っている。

高い信頼性を達成するために、形式手法のような道具を使ってシステムをきちんと系統的に開発し、さらに、システムが正しく作動することを客観的に示すことは大変重要なことである。一方、「早い、安い、動けば良い」、という価値観からは後向きの技術であり、そのため、形式手法は、学生に不人気である。システム開発に携わる技術者としては「正しい」手法を体得して最善を尽くすべきであろう。Tony Hoare のチューリング賞受賞講演からの言葉

> There are two ways of constructing a software design: One way is to make it so simple there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.

にもあるように、「不具合がわからないほど複雑にする」ような愚かな逃げは、技術者がすべきことではない。

謝　辞

筆者が形式手法の世界に迷い込んで以来、多くの先生方にお世話になった。九州大学 荒木啓二郎教授、北陸先端科学技術大学院大学 片山卓也教授、東京大学 玉井哲雄教授、北陸先端科学技術大学院大学 二木厚吉教授、の諸先生（五十音順）に感謝する。また、日立製作所 来間啓伸博士、九州工業大学 鵜林尚靖准教授との議論は形式手法適用に関する理解を深める上で有益であった。深く感謝する。

参 考 文 献

1) 情報処理学会. 「ソフトウェア工学の現状と動向」講習会資料. 1986.12.11-12.
2) 荒木啓二郎, 張漢明. プログラム仕様記述論. オーム社 2002.
3) 栗田太郎. 仕様書の記述力を鍛える. 日経エレクトロニクス, 2007.2.12 号, pages 133–152, 2007.
4) 来間啓伸. B メソッドと支援ツール. コンピュータ・ソフトウェア, Vol. 24, No.2, pages 8–13, 2007.
5) 来間啓伸, B. Wolff, D.Basin, 中島震. 仕様記述言語 Z と証明環境 Isabelle/HOL-Z. コンピュータ・ソフトウェア, Vol. 24, No.2, pages 21–26, 2007.
6) 佐原伸, 荒木啓二郎. オブジェクト指向形式仕様記述言語 VDM++支援ツール VDMTools. コンピュータ・ソフトウェア, Vol. 24, No.2, pages 14–20, 2007.
7) 高木理, 渡邊宏, 武山誠. 対話型証明支援ツール PVS の紹介. コンピュータ・ソフトウェア, Vol. 22, No.3, pages 37–57, 2005.
8) 田辺良則, 高井利憲, 高橋孝一. 抽象化を用いた検証ツール. コンピュータ・ソフトウェア, Vol. 22, No.1, pages 2–44, 2005.
9) 玉井哲雄. ソフトウェア工学の基礎. 岩波書店 2004.
10) 中島震. FM'99 参加報告. コンピュータ・ソフトウェア, Vol. 17, No.2, pages 55–60, 2000.
11) 中島震. オブジェクト指向デザインと形式手法. コンピュータ・ソフトウェア, Vol. 18, No.5, pages 17–46, 2001.
12) 中島震. SPIN 2001 参加報告. コンピュータ・ソフトウェア, Vol. 18, No.5, pages 55–59, 2001.
13) 中島震. 書評 – The SPIN Model Checker. コンピュータ・ソフトウェア, Vol. 21, No.2, pages 61–69, 2004.
14) 中島震. モデル検査を用いたソフトウェアの形式検証. 日本ソフトウェア科学会チュートリアル, 2004/2005.
15) 中島震. モデル検査法のソフトウェアデザイン検証への応用. コンピュータ・ソフトウェア, Vol. 23, No.2, pages 72–86, 2006.
16) 中島震. 形式手法の実像を知る. 日経エレクトロニクス, 2006.8.26 号, pages 123–142, 2006.
17) 中島震. プログラム簡易検証ツール ESC/Java2. コンピュータ・ソフトウェア, Vol. 24, No.2, pages 2–7, 2007.
18) 二木厚吉, 緒方和博, 中村正樹. CafeOBJ 入門. コンピュータ・ソフトウェア (掲載予定).
19) 南出靖彦. C プログラムの検証ツール Caduceus. コンピュータ・ソフトウェア, Vol. 24, No.3 (掲載予定).
20) 結縁祥治. 通信プロセスモデルと形式意味論に基づくソフトウェアのモデル化. コンピュータ・ソフトウェア, Vol. 22, No.2, pages 22–43, 2005.
21) 米田友洋, 梶原誠司, 土屋達弘. ディペンダブルシステム. 共立出版 2005.
22) 渡邊慧. 時間の歴史. 東京図書 1973.