ものづくりと数学—Symbolic Approaches

益岡　竜介 [1,2]　　　穴井　宏和 [1,3]

[1] (株)富士通研究所，[2] 国際公共政策研究センター
[3] 九州大学マス・フォア・インダストリ研究所

1　はじめに

本稿ではものづくりと数学、その中でも、ものづくりへの symbolic approaches の適用を見ていく。

ハードウェアでもソフトウェアでも、それらのものづくりにおいて数値解析や最適化アルゴリズムをはじめとして数学はいろいろな場面で使われている。ものづくりの各過程にあわせて数学を適用する課題を切り出すと次のようになろう。

(1) モデル化 : どのように対象のシステムをモデル化すべきか?

(2) 解析 : どのような解析をそのシステムで行うべきか?

(3) 設計 :
- 問題をどのように定式化し、何を設計目的とするか?
- どのように設計を行う（設計問題を解く）か?

(4) 検証 :
- システムの設計は正しいか、不具合はないか?
- システムが望ましくない状態に陥らないか?

例えばこれを本稿の第 2 章で取り上げている HDD（ハードディスクドライブ）のヘッドの場合に当てはめると以下のようになる。

(1) モデル化 : 制御をするための機構の数理モデルを作る。具体的にはアクチュエータにどんな入力を入れたらどの位置に行くか（出力）の入出力モデルを作る。

(2) 解析 : コントローラでの動作をシミュレーションし、解析を行う。具体的にはモデルでパラメータを変え、どのように動くかのシミュレーションを行う。

(3) 設計 : ヘッドの浮上量や角度などのあるべき状態を最適値とする目的関数を作り、その目的関数を最適化する設計パラメータを決定する。

(4) 検証 : 実際に (1), (2) を使い、(3) で設計したものが条件を満たすか（与えられた仕様内に入っているか）どうかを検証する。

ものづくりの中でも、本稿では特に symbolic approaches の適用を見ていく。ここでいう symbolic approaches とは変数を持った数式を数式のまま処理して解を導くものである。それはどういうもので、数値解析や最適化アルゴリズムなどの numeric approaches とどう違うのか?

例えば設計の都合上、どんな x の値に対しても $x^2 + bx + c > 0$ となっていないといけないとしよう。例えば x はアームが動く範囲、HDD ヘッドの浮上量が $x^2 + bx + a$ と表され、その

浮上量が d より大きい必要があるとすると、その条件は $c = a - d$ とすれば上記と同値になる。そのとき適切な設計パラメータ b, c を決定するという問題を考えよう。Numeric approaches であれば、b, c, x に具体的な数字をランダムに入れたり、漸次的に変更していったりして適切な値を見つけるというものになる。それはそれで式の形や要素に依存しないで解けるのでいいのであるが、一方で（悪い言い方をすると）行き当たりばったりで、また得られた解がどの程度いいものなのかを判断することは難しい。

Symbolic approaches では b と c の範囲は $b^2 - 4c < 0$ として厳密に与えられる。この範囲は二次元の座標系で簡単にプロットすることができ（二次関数で区切られた一つの領域となる)、その中から b と c を選べばよい。 そのとき重要なのは設計パラメータの解 (b, c) の空間の見通しが非常によくなることである。これは設計パラメータを与えても必ず誤差がでてしまうものづくりにとって非常に重要なことである。b, c を領域内の点だが（二次関数で与えられる）境界に近くにとってしまうと、製造誤差で仕様を満たさなくなる可能性が高い。Numeric approaches では得られた解が解集合全体の中でどのようなところにあるのか分からないため、そのような製造誤差への対処が難しい。 一方 symbolic approaches だと解の領域が分かっているため、解を領域内の十分余裕がある点とし、それを設計パラメータと与えることにより、製造誤差を吸収して歩留まりを高めることができる。

あるいは Web サイトの安全性を高めるため Web サイトのプログラムが SQL Injection 攻撃の可能性がないことをテストしたいとしよう。SQL Injection 攻撃は Web サイトの入力フィールド (やアドレスバーなど) に特別な文字列を入力し、それがプログラムで処理され、Web サイトのデータベースに渡す SQL コマンド文字列に変換されたときに、データベースに意図されていない行動を起こさせる攻撃である。 例えば SQL コマンド文字列のどこかに “;SHUTDOWN;” 部分文字列を仕込めれば、データベースが意図しないときに終了する。

今まではこういった攻撃の可能性がないことを検証するためには、(numeric approaches ではないが、具体的な値を使うという意味で numeric approaches に近い手段で）人が Web サイトの入力フィールドに実際にいろいろな文字列を入れて、意図しない危険な SQL コマンド文字列が生成されないことを確かめる。 これは人でなく、コンピュータで Web サイトの入力フィールドへの文字列を生成、自動入力することにより多少効率的にはなるが、本質は変わらない。すなわち入力テキストには無限の可能性があり、その全てを試すことはできない。Hacker の人たちは非常に創造的でどんどん新しい手段を使ってくるので[1] 、事前に分かっている文字列を試すだけでは Web サイトの安全性はあまり高まらない。

Symbolic approaches では入力フィールドへの入力 s を具体的な文字列ではなく、変数のままプログラムを実行する。入力が変数のままでは条件分岐を一つに決定できないが、その条件を式に付け加え、各条件をそれぞれたどって行く。最後に SQL コマンド文字列を作るところでは、その SQL コマンド文字列が入力の変数を含む式 $\mathrm{SQL}(s)$ で表現される。そこで $\mathrm{SQL}(s)$ が “;SHUTDOWN;” を含むという式をたて、その式に解があるかどうかを決定し、もし解がある場合は具体的な解を導出する。もしその式に解がなければ、そのような攻撃ができないということができる。もし解 s_0 があれば、その s_0 を入力として与えれば、データベースが予期

[1]例えば SQL コマンドを HEX で与え、それを SQL コマンド文字列生成の段階で通常の文字列に直す攻撃などもある。

せず終了する可能性があるので、入力に s_0 を排除するコードを付け加えればより安全な Web サイトとなる。

最初の方法では、無限にある入力の空間を有限の（それもかなり限られた）点でカバーしようとしているのに対し、symbolic approaches では、無限にある入力の空間の中から解（= 攻撃）を見つけてくる。もし解がないのなら、そのような攻撃はありえないということを言うことができる。

このように symbolic approaches は非常に強力である。ただもちろん万能ではない。まず対象をモデル化をする部分に課題がある。現実を操作可能な式に落とす際にはどうしても失われるものがある。現実は数多くの要素からなり、非常に複雑であり、完全に数式であらわせると考えるのは naive である。ハードウェアの場合であれば、全ての物理現象を式に落とし込むというのは不可能である。 またソフトウェアの場合であっても、非常に小さなソフトウェアであれば完全に解くことも不可能ではないが、現在の 10 万行、100 万行といった巨大なプログラムになると、必要な部分以外をある程度抽象化しないと、現実的に解くことはできない。

すなわちものづくりでの symbolic approaches の適用ではいかに現実を、あまり関係ない部分を抽象化あるいは省略し、symbolic approaches に乗るようにするかが肝要である。またもう一つは、より高性能なコンピュータあるいは cloud computing による分散コンピューティングを使い、symbolic approaches の適用範囲を広げることである。

以下では、第 2 章ではハードウェアである HDD ヘッドの設計を例に、第 3 章ではソフトウェアの検証を例に、具体的にいかに現実を symbolic approaches に乗せ、また symbolic approaches の適用範囲を広げて実際の問題に適用しているかを記述している。最後の第 4 章でまとめと将来への展望を記述する。

2　ハードウェア設計

さまざまな「ものづくり」における設計過程の効率化・コスト削減、さらに、設計結果の高付加価値化を実現していく上で、最適化技術の発展が 1 つの鍵である。現在、最適化技術の発展は、numeric approaches を前提とした各種アルゴリズムにより支えられている。これらの技術も普及してきたが、より実用的で重要な問題に適用しようとすると numeric approaches だけでは本質的な解決が難しいことも明らかになってきた。例えば、機械系システム設計において、機構系と制御系を同時に最適設計しようとすれば、非凸最適化問題を解くことが必要となるが、numeric approaches では大域的最適解を導くことは容易ではない。また、各種設計における最適な設計パラメータの決定は一般に、さまざまなパラメータ値に対してシミュレーションを繰り返すことが求められ、よりよい解を見通しよく求めること、複数の要求仕様を同時に満たす解を求めることは非常に手間のかかる作業となっている。

ここでは symbolic approaches を用いた最適化手法を紹介する。Symbolic approaches は不定元やパラメータなどの記号が入った式を多項式としてそのまま扱うことを特徴としており、一般には処理時間がかかり扱える問題規模に制約があるが、numeric approaches では処理が困難である課題に対して有効な方法論を提供することができる。具体的には、非線形あるいは非凸な制約問題も正確に解くことが可能となり、また、パラメータをそのまま扱うこと（パラ

メトリック最適化）が可能となる。Symbolic approaches を活用することで、設計仕様を満たす設計パラメータの値をパラメータ空間内の可能領域として正確に求めることができるため、対象となる設計問題の特性を深く理解できよりよい解（よりロバストな解など）の探索や複数仕様設計問題に対しても系統的な設計フローを提供することが可能となる。

最近では、symbolic approaches による最適化手法をさまざまな分野の設計・検証問題に適用する研究が盛んになってきた [1]。特に、不等式制約問題の代数的算法である限量記号消去 (quantifier elimination: QE) を用いてさまざまな問題を解く試みがなされており、ある程度実用的な問題へ適用できるレベルになってきている。次項では、QE とはどういうアルゴリズムか説明し、QE による最適化を導入する（QE について詳細は [1] を参照されたい)。その後で、QE による最適化手法の適用事例として HDD の形状設計の事例を紹介する。

2.1 Symbolic Approaches に基づく最適化

Symbolic approaches によるパラメトリック最適化を実現するための基本技術が QE である。QE は、多項式等式、不等式、限量記号 ($\forall, \exists$)、そしてブール演算（$\wedge, \vee, \Rightarrow, \neg$ 等）からなる一階述語論理式に対し、等価で限量記号を含まない式を導く算法である。出力される式は入力式が真であるための限量記号のない変数の可能な領域を示す。例えば、式 $\forall x(x^2 + bx + c > 0)$ に対し QE によって等価な式 $b^2 - 4c < 0$ が導かれる。QE を用いて最適化問題をどのように解くか説明する。

$$\begin{aligned} &\text{目的関数：} f(x_1, \ldots, x_n) \to \text{最小} \\ &\text{制約条件：} g_1(x_1, \ldots, x_n)\, \rho_1\, 0,\ \ldots,\ g_k(x_1, \ldots, x_n)\, \rho_k\, 0 \end{aligned} \tag{1}$$

ここで、$\rho_i \in \{\neq, <, >, \leq, \geq\}$ である。最適化問題 (1) を QE 問題として解くには、新たな変数（ここでは k）を導入して、$k - f(x_1, \ldots, x_n) = 0$ という式を作り、制約条件と合わせて以下の一階述語論理式を構成する。

$$\exists x_1 \cdots \exists x_n (k - f(x_1, \ldots, x_n) = 0 \wedge \varphi(x_1, \ldots, x_n)) \tag{2}$$

ここで $\varphi(x_1, \ldots, x_n) \equiv \bigwedge_i (g_i(x_1, \ldots, x_n)\, \rho_i\, 0)$ である。一階述語論理式 (2) に QE を適用すると k の満たす論理式 $\psi_1(k)$ が得られる。この式を満たすような k の値の中での最小値を求めるとそれが目的関数 f の最小値になる。得られた結果が示すのは目的関数のすべての正確な実行可能領域である。よって、大域的な最適値がもとまることになり、上限値や下限値があるかどうか不明な問題の場合にも有効である。全変数に限量記号が付いているときには、QE は入力式の真／偽を判定する。まとめると、制約問題や最適化問題の手法として QE は

- 全ての実行可能解をパラメータ空間内の領域として正確に求めることができる
- 非凸な最適化問題も正確に解くことができる
- 実行可能解が存在しない場合も正確に判定できる

という特長を持っている。QE を巧く用いることにより、さまざまな解析・設計問題から得られる制約・最適化問題をパラメトリックに正確に解くことができるようになり、設計の効率化・高度化を図るために有効である。

例　次式で与えられる最適化問題を QE で解いてみる。

$$
\begin{aligned}
&\text{目的関数：} 2x_1 + x_2 \to \text{最小} \\
&\text{制約条件：} 4x_1 + x_2 \leq 9,\ x_1 + 2x_2 \geq 4,\ 2x_1 - 3x_2 \geq -6
\end{aligned}
\tag{3}
$$

この場合、次の一階述語論理式に対して QE を適用する。

$$
\exists x_1 \exists x_2 (k - (2x_1 + x_2) = 0 \land 4x_1 + x_2 \leq 9 \land x_1 + 2x_2 \geq 4 \land 2x_1 - 3x_2 \geq -6) \tag{4}
$$

QE を適用すると、k の実行可能領域 $2 \leq k \leq 6$ が得られる。これより k すなわち目的関数 $2x_1 + x_2$ の最大値が 6 で最小値が 2 であることがわかる。

　次に、以下の多目的最適化問題を考える。多目的最適化は、複数の目的関数を同時に最小化する問題である。通常考える場合には目的関数の間にトレードオフの関係があるため、目的関数の空間における実行可能領域の中で最小化可能なぎりぎりのトレードオフの境界（パレートフロント）を求めるのが目指すところである。多目的最適化についての詳しい解説は [2] を参照されたい。

$$
\begin{aligned}
&\text{目的関数：} f_1(x_1, x_2),\ f_2(x_1, x_2) \to \text{最小} \\
&\qquad f_1 = x_1^2 + x_2^2,\ f_2 = 5 + x_2^2 - x_1 \\
&\text{制約条件：} -5 \leq x_1 \leq 5,\ -5 \leq x_2 \leq 5.
\end{aligned}
\tag{5}
$$

目的空間における f_1, f_2 の実行可能領域を求めるには以下の一階述語論理式を QE で解く。

$$
\exists x_1 \exists x_2 (y_1 = x_1^2 + x_2^2 \land y_2 = 5 + x_2^2 - x_1 \land -5 \leq x_1 \leq 5 \land -5 \leq x_2 \leq 5) \tag{6}
$$

その結果、次式の y_1, y_2 の実行可能領域、すなわち f_1-f_2 空間の実行可能領域（図 1 (a) のグレーの領域）を得る。

$$
\begin{aligned}
&(y_2 - y_1 + 25 \geq 0 \ \land\ y_2^2 - 60y_2 - y_1 + 925 \geq 0 \ \land \\
&\ y_2 \leq 30 \ \land\ y_1 \geq 25) \lor \\
&(4y_2 - 4y_1 - 21 \leq 0 \ \land\ y_2 \geq 30 \ \land\ 4y_1 \leq 101) \lor \\
&(y_2 - y_1 + 15 \geq 0 \ \land\ y_2^2 - 60y_2 - y_1 + 925 \leq 0) \lor \\
&(y_2 - y_1 + 25 \geq 0 \ \land\ y_2^2 - 10y_2 - y_1 + 25 \leq 0) \lor \\
&(4y_2 - 4y_1 - 21 \leq 0 \ \land\ y_2 \geq 5 \ \land\ y_1 \leq 25 \ \land \\
&\ 4y_1 \geq 1).
\end{aligned}
$$

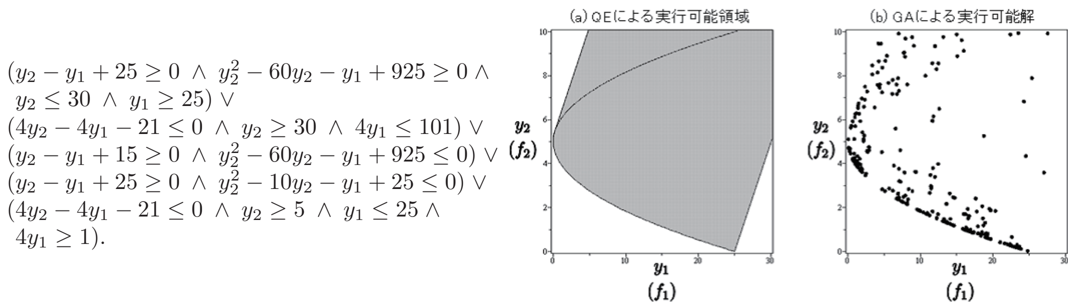

図 1：多目的最適化問題 (5) に対する目的関数の実行可能領域

　この図から、正確なパレートフロントが求まっていることがわかる。同じ問題を数値的な多目的最適化手法である遺伝アルゴリズム (genetic algorithm; GA) によって解いた結果が図 1 (b) である。GA などの numeric approaches ではくり返し計算が進むにつれてパレート付近の実行可能解が増えていく方法なので実行可能領域全体を推測するのは容易ではない。

2.2 適用事例—HDD のスライダ形状設計

ものづくりへの適用例として、HDD のスライダ部分の形状（図 2）の最適設計への応用について紹介する。

スライダは、先端の磁気ヘッドで情報の読み取り・書き込みを行う役割を担っており、ディスクに近いほど読み取り・書き込みエラーが少なくなるが、一方で、ディスクに接触するとクラッシュの原因になる。そのため、スライダとディスク面を適度な距離になるように動作させることが求められる。スライダは、ディスクの回転で生じる空気の流れで浮上しており、浮上量は高度（気圧）などの環境変化によっても変化する。また、浮上時の角度も、アームの位置により空気の流れが変わることや、スライダに縦・横方向への回転が生じることなどで変化する。スライダの浮上量や姿勢は、先端にある ABS（Air Bearing Surface、図 2 の一番右）の形状を工夫することによって調整がなされている。したがって、HDD の設計では、適切なスライダの浮上量や安定な位置・姿勢を実現するための ABS の形状設計が重要になる。

図 2：HDD のスライダと ABS

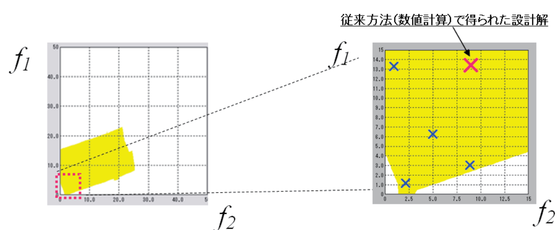

図 3：目的関数 f_1, f_2 の実行可能領域

このような ABS の最適な形状設計問題を多目的最適化問題として捉える。いくつかの高度下での浮上量や姿勢が目標の状態で安定していることを目的関数として数式で表現し、それらの複数の目的関数を最小化する多目的最適化問題として定式化し、前項で紹介した QE による多目的最適化手法を適用する。その結果、ある 2 つの目的関数 f_1, f_2 について、QE により目的関数空間での実行可能領域を求め可視化したのが図 3 である。

数値最適化の手法で得られた設計解に比べ、効率的かつ正確にパレート解を構成することが可能となり、また、数値最適化の手法で得られた設計解よりも、どちらの目的関数に重点をおいてどのぐらいまで最適化できそうかといった知見も得ることが可能になった。その結果、実際の設計工数の大幅な削減が実現できた（ある設計工程では、14 日間かかっていたところを 1 日に短縮できた）。

3 ソフトウェア検証

3.1 背景—なぜ社会にとって重要か

ソフトウェアが社会のいろいろな側面に入り込んでいる。個人が使う PC から、スマートフォン、タブレットなどではソフトウェアが大きな役割を果たしているのが明確であるが、比

較的一般から見えない部分でもソフトウェアの存在感が日に日に増している。金融システム、交通システムなどの社会を支える巨大なソフトウェアから、家電などもコンピュータチップと共にソフトウェアがコントロールしている。最近の車には200以上のコンピュータチップが入り、ソフトウェアが大きな役目を果たしている。Smart Grid（電力網を賢くする取り組み）、Smart City、Smart Homeなどが提案され、ソフトウェアがより重要となる方向は加速こそすれ、とどまることはない。

それらのソフトウェアのバグは金銭的な被害だけではなく、社会や人命にも関わる重大な帰結をもたらしうる。例えば放射線治療の装置のソフトウエアのバグで放射線を大量に浴びた少なくとも6人が亡くなったことがあった。その他にも[3]の記事はソフトウェアバグがどのようなことを引き起こしうるかを示すもので、是非一読ありたい。

そしてソフトウェアのバグは残念だがなくならないであろう。それにはいくつかの要因がある。 一つは現在のソフトウェアあるいはシステムは巨大で、複雑になり、人がその全てを把握することはできなくなってきている。巨大なシステムはまず分割して、それぞれのモジュールが「何」をするものか（仕様書等）を書くのだが、全体が見えない中、また多くの関係するモジュールの関係の中、それぞれのモジュールの「何」を間違えなく書けるかという問題がある。また通常「何」は自然言語で記述されているので、曖昧さも排除できない。そういった面からバグが入り込む余地がある。

また「何」が完璧にかけたとしても、プログラム（すなわちソフトウェア）を作るということはその「何」[2]を「どう」[3]やって実現するのかの手順を記述することである。この「何」とそれを「どう」実現するかの間にギャップが起き得てそこに間違いが入り込む可能性がある。

もちろんソフトウェアは実際に使われる前に意図したとおりに動くかどうか検証（テスト）が行われるのだが、巨大かつ複雑なソフトウェアの全ての状態を検証することは不可能である。それでもできるだけ適切な検証を行おうとするのだが、ソフトウェアが大きくなればなるほど、ソフトウェアを書くためにかかるリソースに比較してその検証を行うために必要なリソースの方が飛躍的に伸びていく。さらには現在の人手による労働集約的な検証[4]には膨大な金額・時間が必要なのだが、世の中はよりコストに厳しい方向に向かっており、検証にかけられるリソースも限られて、バグが見つけられず残ってしまう可能性が高まる。

そのテストの工程を一部でも自動化し、よりコスト効率的に行う、あるいはソフトウェアのより広い部分をカバーする検証を可能にするのがソフトウェア検証である[5]。

3.2 ソフトウェア検証の歴史

ソフトウェア検証の試みは1960年頃から始まった。当初はプログラムが行う「何」を厳密に書き、それと実際に書かれたプログラムが正確に一致しているかを厳密に「証明」すると

[2]例えば「与えられた直径の円の面積を求める」としよう。

[3]例えば先の簡単な円の面積の例に対しても「直径を2で割った値をrに記録し、$3.1415 \times r \times r$を計算して返す」とか、「与えられた直径dを使って$d^2 \times 0.7854$を計算して返す」など「どう」は無限にありうる。

[4]例えばWebサイトのシステムでは人がWebブラウザーからいろいろなデータを入力し、想定通りの動作をするかを検証する。

[5]ソフトウェア検証についての一般的な紹介については[4]や[5]を参照されたい。**??**で紹介している米国富士通研究所で研究開発されているSymbolic実行については[6]を参照されたい。

いったことを行っていた。

しかしそれでは「何」を書くために全く別の書式を学ばないとならず、また「何」を書くだけで別にプログラムを書く程度の労力が必要となる。そのため小さなプログラムにしか適用できず、また広く使われることにはならなかった。つまり当初のアプローチは大規模なソフトウェアに対しては無力であった。

その潮目が変わったのは 1998 年頃で、それまでの正確性を証明などによって保証するという立場から、厳密性・完全性は犠牲になるかもしれないができるだけ多くのバグを見つけようという立場に変わっていった。以前のアプローチは無駄ではなかったが、やはり象牙の塔の中の研究であったことは否めず、産業界からのそれが何の役に立つの？ という問いかけに対して、実際のプログラムを直接検証できるようにしようという方向に変わった。それからは CMU、Stanford 大学、Microsoft、NASA、UC Berkeley、NEC、そして富士通などがソフトウェア検証をより大きなソフトウェアに適用可能にする努力を続けてきた。

3.3 Symbolic 実行

ソフトウェア検証の symbolic 実行についてその手法を簡単な例をもって以下に説明し、そのメリット、限界、今後を議論する。

```
foo(a, b, c) {
  int a,b,c;
  c = a + b;
  if (c > 0) {
    c++;
  }
  return c;
}
```

図 4：プログラム

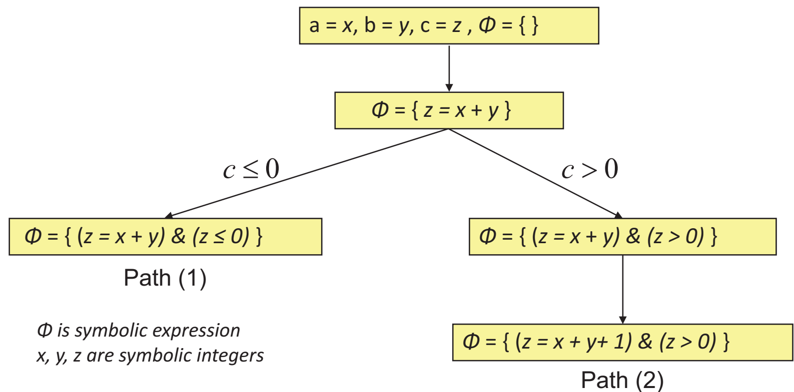

図 5：Symbolic 実行の流れ

プログラムの例として次のような簡単な関数（図 4）を考える。この関数は手続きとして書かれているが、その関数の役割から

$$(a > 1) \wedge (b > 0) \rightarrow (c > 4) \tag{7}$$

という条件を満たさないといけないとしよう。このプログラムが本当にこの条件を満たすものになっているかどうかを確認するためには以下のようにする。

まず条件 (7) の結論を否定したもの

$$(a > 1) \wedge (b > 0) \rightarrow (c \leq 4) \tag{8}$$

を用意する。関数の実行の最後でこの結論を否定した条件 (8) を満たす a, b, c がもし存在しなければ、このプログラムはもともと与えられた条件 (7) を満たし、もし存在すればこのプログラムは条件 (7) を満たさ**ない**と結論付けることができる。

そのために関数（図 4）を図 5 のように変数 a, b, c に symbolic な変数 x, y, z を代入してそのまま実行する。代入文 ($c = a + b$) があれば、それを対応する条件 ($z = x + y$) を symbolic 表現に付け加え、条件文 (if ($c > 0$)) のところでは、その真偽を決定できないので、条件文の条件式（$z \leq 0$ と $z > 0$）をそれぞれに加えて、両方の分岐を別々にたどっていく。それら二つの経路、Path (1) も Path (2) もやがて関数の終わりにたどり着き、それぞれの経路をたどる条件がそれぞれの Φ に蓄えられる。

Equations at the End of Path (1)

x > 1
y > 0 } Preconditions
z = x + y
z <= 0
z <= 4 } Post condition

Solve using ILP
- No solutions
- Property holds

Equations at the End of Path (2)

x > 1
y > 0 } Preconditions
z = x + y + 1
z > 0
z <= 4 } Post condition

Solve using ILP
- **SOLUTION FOUND !!**
- Counter example: x = 2, y = 1, z = 4

図 6：各経路の実行終了時の式

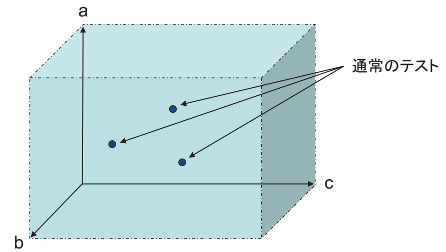

図 7：通常テストと Symbolic 実行

図 6 のように、Φ のそれぞれに条件 (8)（図の中では Precondition と Post Condition とに分けている）を組み合わせ、それぞれの式のセットを例えば ILP (Integer Linear Programming) を使って解く。Path (1) の場合には解がなく条件 (7) を満たし、Path (2) の場合には解（例えば $x = 2, y = 1, z = 4$）が存在し、条件 (7) を満たさ**ない**ことが分かる。Path (2) の場合に得られた解は実際に（与えられた条件 (7) を満たさないという意味で）エラーを引き起こす入力として関数にテストデータとして与え、バグの存在確認にも使うことができる。

図 7 で模式的に表されるように、通常のテストでは変数空間内の限られた点でしか与えられた条件を満たすかどうか検証できないのに対して、この symbolic 実行では変数空間全体で一気にその条件を満たすかを検証することができる。

これは一般にも有効であるが、terminal condition[6]での検証や、通常のプログラムの実行ではまれにしか通らない経路での検証[7]などに特に力を発揮する。

このように symbolic 実行は強力ではあるが、もちろんその限界がある。条件分岐があれば経路がどんどん増えていき、探索すべき状態空間が爆発し、計算機の処理能力の限界を超してしまう。またより多くの変数を symbolic として扱えば、計算量的負担はさらに増える。Symbolic 実行はだいぶ実用のレベルに近づいてきたが、実際のより巨大なプログラムに symbolic にする変数の数などの制約がより少なく、実用的に適用できるようにするためにさらに工夫を重ねる必要がある。

[6] c が実数で $c > 545$ といった条件。$c = 546, 545.1, 545.01, \ldots$ とどこまで 545 に近い値を入れても完全には検証できない。

[7] 具体的な値を必要とする通常のテストではその経路を通るようにすることが難しい。Symbolic 実行では全ての経路を同じようにたどり検証することができる。

その更なる実用化には二つの方向から取り組まれている。一つはプログラムの適切な抽象化を行い、状態空間の爆発を抑え込むものである。例えば、注目するモジュールだけ厳密にsymbolic 実行を行い、それ以外の関係するモジュールは簡単化したり、注目している部分に関係しないコードを削除してしまったり、数値変数を整数や実数ではなく正負とゼロの場合だけにしてしまうことなどが行われており、さらにいろいろな新しいアイデアも導入されている。

もう一つの方向は cloud computing など分散コンピューティング技術を使って経路探索を複数の計算ノードに分散して、扱える状態空間のサイズを大きくするといったものである。

4　おわりに

本稿では、数学を活用した「ものづくり」の効率化・高度化のための方法論として symbolic approaches を説明し、ハードとソフトの開発での適用事例を紹介した。

ものづくりの現場では、開発対象の複雑化や開発期間の短縮化が進むにつれ、効率化・コスト削減・高付加価値化の実現のためモデルベース設計が注目されている。その高度化には、本稿で紹介した symbolic approaches をはじめとした各手順を支える数理的手法の継続的な革新が必須である。さらに、数理的手法を積極的に用いた設計手法を、広く有効活用してもらうにはわかりやすさや使いやすさを考慮したツールとして提供することも大切な点であり、将来的には、数学問題をコンピュータが自動で解くような研究 [7] も活用されるであろう。

最後に、ものづくりの分野で蓄積されてきた数理的な分析・最適化・検証手法の重要性は、ものづくりに留まらず社会システム、エネルギーマネージメントなど広範な領域においても今後ますます高まっていくと思われる。

参考文献

[1] 穴井宏和，横山和弘，『QE の計算アルゴリズムとその応用—数式処理による最適化』東京大学出版会，2011.

[2] 中山弘隆，岡部達哉，荒川雅生，尹禮分，『多目的最適化と工学設計—しなやかシステム工学アプローチ』現代図書，2008.

[3] “Epic failures: 11 infamous software bugs,” ComputerWorld, Sep. 9, 2010, `http://www.computerworld.com/s/article/9183580`.

[4] Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C., Sen, K., Tillmann, N., “Symbolic Execution for Software Testing in Practice: Preliminary Assessment,” ICSE’11.

[5] D’Silva, V., Kroening, D., Weissenbacher, G., “A Survey of Automated Techniques for Formal Software Verification,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), July 2008.

[6] Li, G., Ghosh, I., Rajan, S. P., “KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs,” CAV 2011.

[7] 相澤彰子，松崎拓也，穴井宏和，“自然言語処理と計算代数の接合による数学問題へのアプローチ”人工知能学会誌 27 (5), pp. 483–491, 2012.