GSLetterNeo vol.130

2019 年 5 月

形式手法コトハジメ –TLA$^+$ Toolbox を使って-

熊澤 努 kumazawa @ sra.co.jp

はじめに

2013 年の GSLetterNeo Vol.56 に形式検証ツール LTSA を紹介する記事を寄稿しましたが、その後かなり間が空いてしまいました。今回は LTSA とは異なるツール TLA$^+$ Toolbox[1]を紹介します。TLA$^+$ Toolbox は Microsoft Research の Leslie Lamport 等により開発されている形式的仕様記述・検証ツールです。Lamport は並行システムの解析における世界的な権威で、その業績により 2013 年に ACM A.M.チューリング賞を受賞しています。学術論文を書く方には、Latex の開発者としてお馴染みかもしれません。

TLA$^+$ Toolbox はオープンソースソフトウェアとして Java で開発されており、無償で使うことができます。チュートリアルも充実していますので、本稿を読んで興味を持たれた読者は、公式サイトからダウンロードして使っていただくことをお勧めします。

本稿の目的は、TLA$^+$ Toolbox でどのようなことができるのか、そのイメージを読者に持っていただくことにあります。したがって、詳細な使用方法には立ち入りません。形式手法は、理論が先行していて非常に難しく、現実のソフトウェア開発への適用は難しい、と考えられがちですが、その敷居の高さを少しでも緩和できましたら幸いです。

---

[1] より詳しい情報は公式サイトで知ることができます。
https://lamport.azurewebsites.net/tla/tla.html

形式手法について

形式手法とは、開発対象システムの仕様を形式的に記述することで、仕様を正確に把握し、不具合をなくしていく開発技術を指します。形式的とは数学的ということです。数学というと敬遠したくなる読者もいると思いますが、数学的な記述はコンピュータとの相性がよいという利点があります。また、近年は便利なツールが数多く開発されていて、高度な専門知識がなくとも形式手法を手軽に利用できるようになってきました。形式手法の研究開発には大きく分けて次の二つの流れがあります。

1. 形式的仕様記述言語とその記述環境の研究開発
2. 与えられた仕様の正しさを検証するための技法と検証ツールの研究開発

第一の仕様記述言語とは、ユーザが仕様を書く際に使う言語です。抽象度の高い記述をするための機構が備わっている言語が多いですが、今はプログラミング言語に近いものとイメージしていただくのがよいと思います。実際、プログラムを最も粒度の細かい形式仕様とみなす考え方もあります。なお、ここでの「仕様」という言葉には注意が必要です。開発現場で一般的な仕様書は、システムの挙動や構成といった様々な側面を記述する文書を指すことが多いですが、形式手法でいう仕様は、特定の側面を精緻に記述したものです。例えば、TLA$^+$ Toolbox はシステムの挙動の記述のみを取り扱っています。

第二の検証は、テスティングや仕様のシミュレーションなど、仕様の正しさを確認する工程を指します。TLA$^+$ Toolbox は形式的自動検証を標準機能として提供しています。形式的自動検証は、ツールが自動的に数学でいう証明をする技術です。TLA$^+$ Toolbox は形式的な仕様記述と形式的な検証の両方をサポートする強力なツールです。

仕様を記述する

TLA$^+$ Toolbox では PlusCal と TLA$^+$という二つの言語を使って仕様を記述することができます。例を見てみましょう。次に挙げるのは、PlusCal で書いた食事をする哲学者の問題の仕様記述の例です。食事をする哲学者の問題は、自律動作する複数のプロセスでデッドロッ

```
------------------------- MODULE AsymmetricDiningPhilosophers -------------------------
EXTENDS Integers, Sequences, TLC, FiniteSets
CONSTANTS NumPhilosophers, NULL
ASSUME NumPhilosophers > 0

(* --algorithm asymmetric_dining_philosophers
variables forks = [fork \in 1..NumPhilosophers |-> NULL]
define
LeftFork(p) == p
RightFork(p) == IF p = NumPhilosophers THEN 1 ELSE p + 1
HeldForks(p) == { x \in {LeftFork(p), RightFork(p)}: forks[x] = p}
LeftAvailable(p) == forks[LeftFork(p)] = NULL
RightAvailable(p) == forks[RightFork(p)] = NULL
end define;

macro take_left() begin
 await LeftAvailable(self);
 forks[LeftFork(self)] := self;
end macro;

macro take_right() begin
 await RightAvailable(self);
 forks[RightFork(self)] := self;
end macro;

process philosopher \in 1..NumPhilosophers
variables hungry = TRUE;
begin Proc:
  while hungry do
    if self = NumPhilosophers then
     RightNP: take_right();
     LeftNP: take_left();
    else
     Left: take_left();
     Right: take_right();
    end if;
    Eat:
     if Cardinality(HeldForks(self)) = 2 then
       hungry := FALSE;
       forks[LeftFork(self)] := NULL ||
       forks[RightFork(self)] := NULL;
     end if;
  end while;
end process;
end algorithm; *)
```

クが発生する問題としてよく知られています。上記の例は、チュートリアル[2]をもとに筆者がデッドロックを解消する仕様を作成したもので、チュートリアルの簡単な別解となっています。

PlusCalでの仕様記述は`(* --algorithm`から`end algorithm; *)`の範囲です。いかがでしょうか。プログラミングの経験のある読者であれば、`variables`が変数の宣言であること

[2] https://learntla.com/introduction/ ． 食事をする哲学者の問題については https://learntla.com/concurrency/processes/ に掲載されています。本稿では食事をする哲学者の問題の詳細には立ち入りませんので、興味のある読者は参照してください。

や、`while`が繰り返し処理であること、`if`が分岐を表すことはなんとなく見て取れるのではないでしょうか。また、この仕様で記述されているシステムは、複数の自律的なプロセスから成っています(`process`から`end process;`までがプロセスを表します)。このように、PlusCalは手続き型プログラミング言語に近い記法で複数のプロセスから成る複雑な仕様を書くことができるのです。

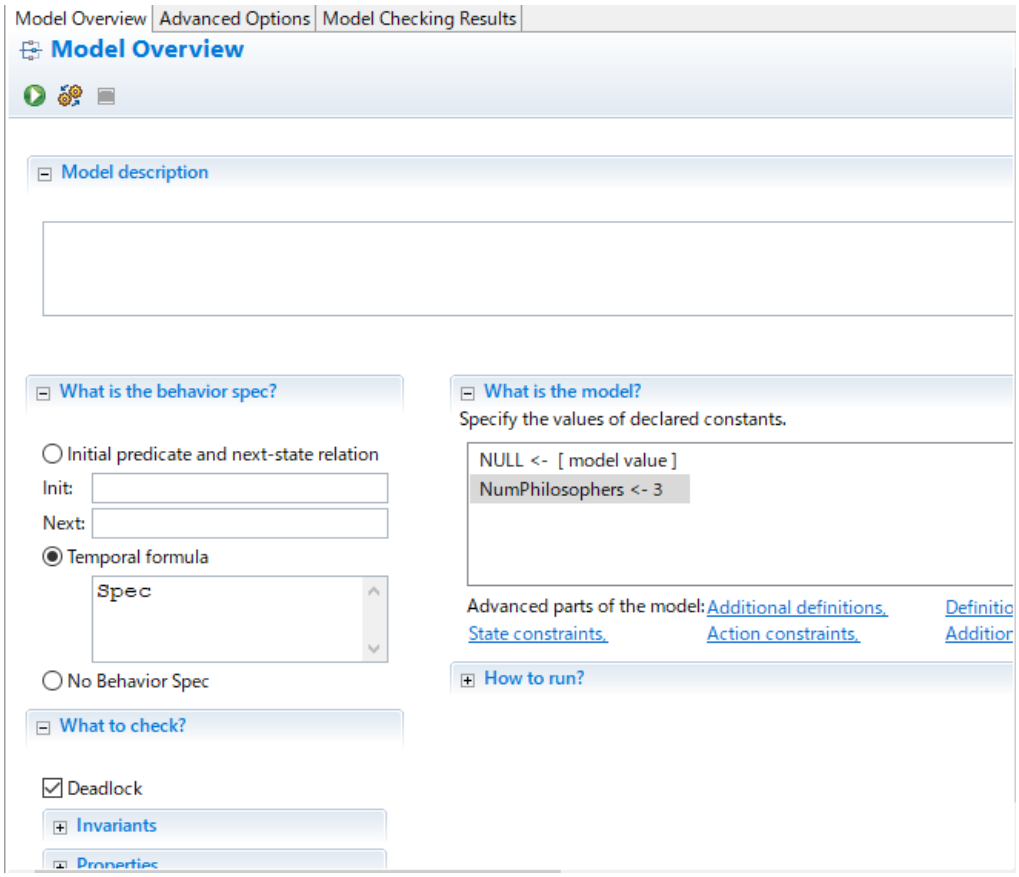
自動的に仕様を検証する

上で作成した仕様を検証してみましょう。残念ながら TLA$^+$ Toolbox は TLA$^+$で書かれた仕様は自動で検証できますが、PlusCal で書かれた仕様を検証することはできません。そこで、まず PlusCal の仕様を TLA$^+$に変換します。幸いなことに、TLA$^+$ Toolbox にはこの変換を自動で行う機構が備わっています(変換結果は省略します)。なお、TLA$^+$は論理式で仕様を書く言語です。論理式を扱うのが得意な読者は、PlusCal を使わずに TLA$^+$で仕様を書いてもかまいません。

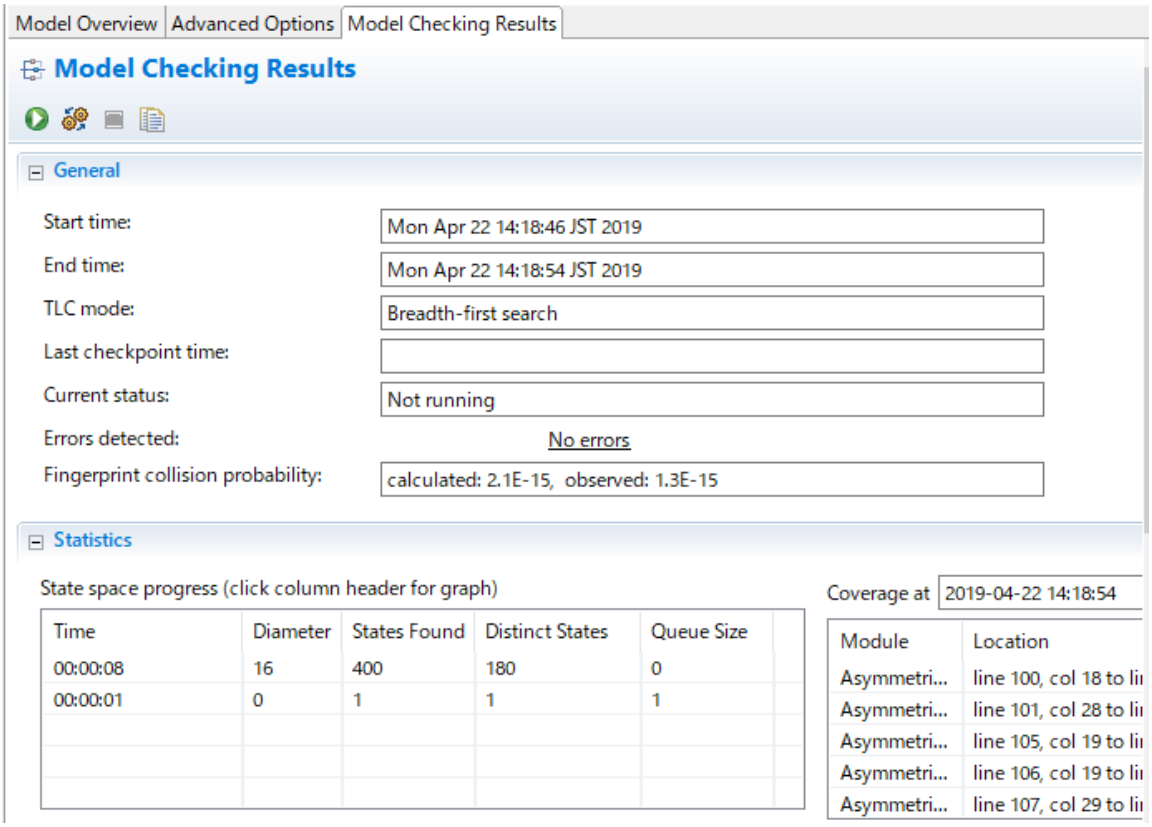
検証の際にはモデルを作成します。モデルには、システムの振る舞い、検証する項目、定数の値など、検証に必要な設定事項を記述します。下記のモデルの設定には、3 つの哲学者プロセスに対して、デッドロックが発生しないことの検証を行う旨を記載しています。

設定をもう少し詳しく見てみましょう。システムの振る舞いは Temporal formula という設定に書かれた Spec が該当します。Spec は PlusCal で記述した振る舞いを表しており、PlusCal から TLA$^+$への変換で自動的に作られます。検証項目は、仕様の正しさの基準となるようなシステムが満たすべき望ましい性質で、What to check? 以下で設定します。ここでは、Deadlock にチェックが入っていて、デッドロックの有無の検証をする設定となっています。定数の値は What is the model? という項目で設定しています。

最後に、モデルの左上にある実行ボタンを押すと、自動的に検証が実行されます。



検証結果は Model Checking Results という画面に表示されます。Errors detected に No errors と表記されていることからわかるように、エラーは検出されませんでした。したがって、この仕様は、デッドロックを引き起こすことはありません。重要なことは、検証とは、デッドロックが発生しないことを仕様に対して形式的・数学的に保証するものであるということです。この点が形式的な検証がテスティングとは大きくテストとは異なります。また、ユーザがボタンを押すだけで、自動的に検証が実行される点も注目に値します。このことは、TLA$^+$ Toolbox が検証技術に採用しているモデル検査に依るところが非常に大きいですが、検証のために複雑な手順を踏まなくてよいことは、ツールの使いやすさの点からも大事なことであると思います。



おわりに

本稿では形式的仕様記述・検証ツール TLA$^+$ Toolbox を簡単に紹介しました。TLA$^+$ Toolbox で仕様をもっと書いてみようと考えている読者は、まず公式サイトやツールのヘルプから辿れる各種チュートリアルを読んでみてください。その後、Lamport の著書[3]を読むのがよいでしょう。決してやさしい本ではなく、また、本稿の執筆時点では日本語訳もありませんが、挑戦していただきたいと思います。

形式手法は、開発工程の様々な局面で活躍する可能性を秘めています。例えば、設計時に、自然言語の仕様書を形式的に書き直してみることで、仕様書の不備や矛盾点を発見することができるでしょう。また、作成したプログラムを記述言語に書き直せば、プログラムの不具合の発見に繋げることもできます。現場に合ったやり方を模索しながら、TLA$^+$ Toolbox というツールを気軽に使ってみてください。

[3] L. Lamport. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc.

GSLetterNeo Vol.130
2019 年 5 月 20 日発行
発行者　株式会社 SRA 先端技術研究所

編集者　土屋正人
バックナンバー http://www.sra.co.jp/gsletter
お問い合わせ　gsneo@sra.co.jp

株式会社SRA
〒171-8513 東京都豊島区南池袋 2-32-8

夢を。Yawaraka Innovation
やわらかいのべーしょん