GSLetterNeo vol.138

2020 年 1 月

形式手法コトハジメ

–TLA$^+$ Toolbox を使って (5)-

熊澤 努 kumazawa @ sra.co.jp

はじめに

GSLetterNeo Vol.136 では、PlusCal の代表的な制御構造について解説しました。繰り返しや条件分岐を使いこなせれば、かなり複雑な仕様を形式的に書くことができるようになります。

今回も引き続き、仕様の書き方について解説します。形式的な仕様記述法が活躍する分野の一つが、並行システムの仕様の記述です。並行システムは、スレッドやプロセスに代表される計算する実体が複数存在するシステムのことです。このようなシステムの挙動を予測することは難しいため、深刻で解析の難しい不具合をしばしば引き起こすことが知られています。そのため、並行システムの開発の際には、システムの振る舞いに細心の注意を払う必要があります。TLA$^+$ Toolbox には、そのような並行システムのための仕様記述の方法と、不具合解析機能が提供されています。今回は、プロセスを用いた仕様記述について考えていくことにします。これまでの記事より、内容が若干複雑になると思いますので、じっくり読んでみていただきたいと思います。

Vol.136 の最後に、今回はデータ構造を採り上げると予告していましたが、形式手法において重要な概念である並行性について優先して扱ったほうが良いと考え、勝手ながら予定を変更しました。

信号機が 2 基の場合を考える

これまでは、信号機が 1 基の場合の仕様についてのみ考えてきました。ここでは、信号機が 2 基ある場合はどうすればよいか考えてみましょう。以降では、2 基の信号機をそれぞれ信号機 1、信号機 2 と呼ぶこととします。なお、説明を簡単にするため、仕様に関して次のような仮定を置くことにします。

仮定1. 全ての信号機は「赤→青→黄」の順に点灯を繰り返す。故障や電源 OFF については考えない。

仮定2. 各信号機は、他の信号機の灯火や挙動に依存しない固有のタイミングで、灯火を変化させる。

仮定 2 は少々わかりにくいかもしれません。これは、例えば、「信号機 1 の灯火が赤の時には、信号機 2 の灯火が青でなければならない」、のような制約はなく、信号機 2 の灯火の変化は、信号機 1 とは無関係の固有の適当なタイミングで発生する、ということです。信号機 1 についても同様に、信号機 2 とは無関係に灯火を変化させます。

さて、一つの書き方として考えられるのは、各信号機の灯火に対応する変数を用意することです。ここでは、信号機 1 と信号機 2 のそれぞれの灯火色に対応する変数を `light1`、`light2` とします。後はこれまで通り、赤、青、黄の 3 色に相当する文字列 `"red"` 、`"green"` 、 `"yellow"` を適当なタイミングで代入すれば、仕様は完成するでしょう。仕様の例として、 `bad_spec1` を次に示します。

```
---------------------------- MODULE bad_spec1 ----------------------------
(* --algorithm TrafficLight
variables
  light1 = "red",
  light2 = "red";
  begin
    Signal:
      while TRUE do
        Green:
          light1 := "green";
          light2 := "green";
        Yellow:
          light1 := "yellow";
          light2 := "yellow";
        Red:
          light1 := "red";
          light2 := "red";
      end while;
end algorithm;*)
=========================================
```

この仕様は、前回(Vol.136)の「繰り返しを書く」で考えた仕様をもとにして、変数を 2 つにしたものです。仮定 1 と 2 を満たすか検討しましょう。

仮定1. 変数 light1、light2 は、ともに初期値赤から「青→黄→赤」の順に値が変化します。つまり、信号機 1、信号機 2 ともにこの仮定を満たしています。

仮定2. 変数 light1 が赤から青に変化したときには、必ず変数 light1 の値も赤から青に変化します。これは各信号機の灯火色は互いに依存しないという仮定に反しています。この仕様では、例えば、信号機 1 が青で信号機 2 が黄という場合を表現することができません。

仮定2を満たしていないということは、灯火の組合せに漏れがあるということです。漏れをなくすために、light1 と light2 へ値を順に代入するのではなく、それぞれの代入文を「異なる個所に記述する」ように仕様を変更してみましょう。下の図に、この考え方のイメ

```
 (* --algorithm TrafficLight
variables
  light1 = "red",
  light2 = "red";
  begin
    Signal:
      while TRUE do
        \* light1の灯火変化(light2への代入はしない)
        ...
        \* light2の灯火変化(light1への代入はしない)
        ...
      end while;
end algorithm;*)
```

ージを示します。

では、上の図中の ... にあたる個所を埋めた、次の仕様 bad_spec2 はどうでしょうか。この仕様は非決定的な振る舞いを使って、書いています。

```
------------------------------ MODULE bad_spec2 ------------------------------
(* --algorithm TrafficLight
variables
  light1 = "red",
  light2 = "red";
  begin
    Signal:
      while TRUE do
        \* light1
        either
          light1 := "red";
        or
          light1 := "green";
        or
          light1 := "yellow";
        end either;
        \* light2
        either
          light2 := "red";
        or
          light2 := "green";
        or
          light2 := "yellow";
        end either;
      end while;
end algorithm;*)
==========================================
```

この仕様も仮定 1 と 2 を満たすか検討しましょう。

仮定1. 変数 light1、light2 には、ともに初期値赤から繰り返しのたびに赤、青、黄のいずれかの値が代入されます。したがって、赤から黄に変化するという不正な変化を盛り込んだ仕様になっているため、この仮定を満たしません。

仮定2. 変数 light1 は light2 の値にかかわらず、赤、青、黄のどの値も取りえます。同じことは light2 についてもいえますので、この仮定を満たしていることがわかります。

今度は仮定 1 に違反してしまいました。なかなか思うようにいきません。ここではこれ以上追求しませんが、条件や繰り返し構文をうまく使って、仮定を満たす仕様を作ってみてください。その際、信号機の数が 3 個、4 個、…と増えていった場合でも、追加した信号機の挙動を簡単に追加できる書き方を考えると良いと思います。信号機の数が 1 基増えただけでも、結構複雑な仕様になるのではないでしょうか。

幸いにも、 PlusCal に用意されているプロセスという記法を使うと、この場合の仕様を簡単に書くことができます。

プロセスを書く

信号機 1 と信号機 2 の仕様を書く際のポイントは、仮定 2 にうまく対処することです。仮定 2 は、両信号機ともに、互いの挙動とは独立かつ自律的に灯火を変化させる、ということを述べています。このような自律的な振る舞いをする実体は、プロセスと呼ばれます。この記事の最初に述べたように、プロセスは並行システムや並列プログラミングで登場する概念です。信号機をプロセスと考えると、仕様をすっきりと書くことができます。
PlusCal にはプロセスを個々に記述する機構が備わっています。早速、信号機の仕様の例を見てみましょう。

```
------------------------------ MODULE spec ------------------------------
(* --algorithm TrafficLight
  process tl \in {1, 2}
    variables
      light = "red";
    begin
      Signal:
        while TRUE do
          Green:
            light := "green";
          Yellow:
            light := "yellow";
          Red:
            light := "red";
        end while;
  end process;
end algorithm;*)
=============================================
```

プロセスの挙動は `process` と `end process;` の間に記述します。`process` の後の `tl` は、プロセスの名称です。PlusCal のプロセスには名前をつけなくてはいけません。その後の `\in {1, 2}` は少しわかりにくいので、詳しく説明します。順番が前後しますが、`{1, 2}` はプロセス識別子といいます。中括弧 {} は集合を表します。この例の場合は、要素 1 と 2 からなる集合で、数学における外延的記法 {1,2} と同じです。または、プログラミング言語 Python の set の記法と該当すると考えても構いません。要するに、1 と 2 の異なるプロセスの実体があることを意味していて、信号機 1 と 2 に対応しています。次に、プロセ

ス識別子の前 `\in` は、集合の演算記号 ∈ です。∈は、左辺が右辺の集合の要素であることを意味します。Python の in と同じです。`tl \in {1, 2}` を数学的に書き直すと、tl ∈ {1,2} となり、 `tl` が 1 か 2 のいずれかであることを述べています。PlusCal ではこの記法を利用して、プロセス `tl` に 1 と 2 の 2 つの実体があることを表現します。なお、`\in` という書き方は Latex に由来しています。

残りの部分は、前回(Vol. 136)の「繰り返しを書く」に掲載した仕様と全く同じです。ただし、プロセス内で宣言した変数 `light` は局所変数であるという点に注意が必要です。プロセス内で宣言された局所変数のスコープは、プロセス内のみに限定されます。つまり、プロセス外部の処理や、異なるプロセスからアクセスすることはできません。このことは、異なるプロセス 1 と 2 とでは変数の値を共有しないことを意味しています。プロセス 1 と 2 に同一の `light` と名付けられた異なる変数があると考えてください。これらは先の仕様で用意した 2 つの変数 `light1`、`light2` に対応しています。

信号機プロセスの詳しい挙動を見てみるために、先の信号機に関する仮定を検討することにしましょう。

仮定1. 信号機プロセスの変数 `light` の初期値は赤です。その次の繰り返しを実行するたびに、「青→黄→赤」の順に値が変化します。つまり、信号機 1、 2 ともにこの仮定を満たしています。

仮定2. 2 つのプロセスは互いの局所変数 `light` を参照することも、変更することもできません。このことは、信号機 1 と 2 が、互いの灯火の状況に依存せずに、自身の灯火を変更することを意味しています。したがって、この仮定も満たしています。

以上から、仮定を満たす仕様が記述できたことがわかります。今回の信号機のように、独立して動く実体がある場合には、プロセスを使うと非常に簡単に仕様を書くことができます。さらに、各プロセスには、自身の灯火色の変化のみが記述されているため、全体として読みやすい仕様にもなっています。PlusCal の仕様から TLA$^+$の仕様を出力して、この仕様が構文上正しい記述であることも忘れずに確認しましょう。

最後に、プロセスは信号機が増えた場合にも有効な記法です。例えば、信号機が 2 基から 3 基に増えたとしましょう。仕様をどのように変更すればよいでしょうか。答えは、プロセス識別子を `{1, 2}` から `{1, 2, 3}` にするだけです。他の個所を変更する必要はありません。信号機が何基あっても、プロセス識別子を変更するだけなので、仕様を簡単に書き直すことができます。

おわりに

本稿では、プロセスを使った並行システムの仕様の記述の書き方の基本を説明しました。近年は、マルチプロセス・マルチスレッド環境の下で実行するソフトウェアが増えてきています。システムの並行性に起因する不具合は、一般に取り扱いが難しいのですが、TLA$^+$ Toolbox には開発者を支援する各種の機能が提供されています。その一つには、今回は紹介できませんが、書かれた仕様の正しさを自動的に検査するモデル検査という機能があります。仕様記述と検査機能を上手に組み合わせることで、複雑なシステムを正しく記述することができるので、機会があれば挑戦していただきたいと思います。

GSLetterNeo Vol.138
2020 年 1 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