Niezawodne Systemy Informatyczne

Lista 4

PRZEMYSŁAW KOBYLAŃSKI

Rozwiąż samodzielnie jedno z poniższych zadań za maksymalnie 16 punktów i przedstaw prowadzącemu najpóźniej na 15. zajęciach.

Za zadanie otrzymuje się:

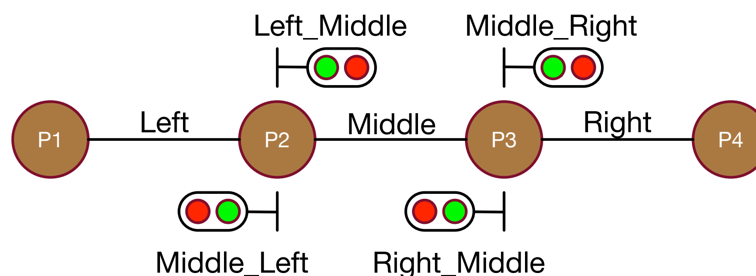
- 100% punktów gdy program **gnatprove** udowodnił wszystkie asercje (*Assert*), kontrakty (*Post*), kontrole indeksów (*index check*) i zakresów (*overflow check*) oraz nie użyto założeń (*Assume*);
- 75% punktów gdy program **gnatprove** udowodnił wszystkie asercje, kontrakty, kontrole indeksów;
- 50% punktów gdy program **gnatprove** udowodnił wszystkie kontrakty i kontrole indeksów (nieudowodniona asercja *Assert* będzie uznawana za użycie założenia *Assume* i wymaga uzasadnienia słownego podczas zaliczania).

Każde użyte założenie *Assume* należy udowodnić prowadzącemu podczas zaliczania listy ale zasadność jego użycia może być zakwestionowana przez prowadzącego a ma on głos rozstrzygający.

Wskazówka: poczytaj o parametrach polecenia **gnatprove** (*level*, *steps*, *timeout* i innych).

Zadanie 1

Na poniższym rysunku przedstawiono fragment sieci kolejowej:



Fragment ten składa się z trzech segmentów:

- **Left**,
- **Middle**,
- **Right**.

Segment sieci, to jej część, w której nie może przebywać jednocześnie więcej niż jeden pociąg (zakładamy, że na segment składa się tylko jeden tor).

Między segmentami znajdują się punkty, w których umieszczono sygnalizatory:

- **Left_Middle** — dla pociągów jadących z segmentu **Left** do segmentu **Middle**,
- **Middle_Right** — dla pociągów jadących z segmentu **Middle** do segmentu **Right**,
- **Right_Middle** — dla pociągów jadących z segmentu **Right** do segmentu **Middle**,
- **Middle_Left** — dla pociągów jadących z segmentu **Middle** do segmentu **Left**.

Każdy sygnalizator może być w jednym z dwóch stanów:
- **Green** — droga wolna,
- **Red** — stop.

Każdy z segmentów może być w jednym ze stanów:
- **Occupied_Standing** — pociąg w segmencie i czeka,
- **Occupied_Moving_Left** — pociąg w segmencie i porusza się w lewo,
- **Occupied_Moving_Right** — pociąg w segmencie i porusza się w prawo,
- **Reserved_Moving_From_Left** — nie ma pociągu ale zaraz nadjedzie z lewej,
- **Reserved_Moving_From_Right** — nie ma pociągu ale zaraz nadjedzie z prawej,
- **Free** — nie ma pociągu i w najbliższym czasie nie pojawi się.

Otwieranie trasy i przemieszczanie pociągu zależy od tego skąd dokąd chce on się przemieścić, przy czym jest osiem następujących możliwości:
- **Route_Left_Middle** — gdy pociąg jedzie z segmentu **Left** do segmentu **Middle**,
- **Route_Middle_Right** — gdy pociąg jedzie z segmentu **Middle** do segmentu **Right**,
- **Route_Right_Middle** — gdy pociąg jedzie z segmentu **Right** do segmentu **Middle**,
- **Route_Middle_Left** — gdy pociąg jedzie z segmentu **Middle** do segmentu **Left**,
- **Route_Enter_Left** — gdy pociąg wjeżdża z lewej strony do segmentu **Left**,
- **Route_Leave_Right** — gdy pociąg wyjeżdża w prawą stronę z segmentu **Right**,
- **Route_Enter_Right** — gdy pociąg wjeżdża z prawej strony do segmentu **Right**,
- **Route_Leave_Left** — gdy pociąg wyjeżdża w lewą stronę z segmentu **Left**.

Poniżej przedstawiono plik **railway.ads** zawierający specyfikację pakietu **Railway**:

```
package Railway with SPARK_Mode is

   type One_Signal_State is (Red, Green);

   type Route_Type is (Route_Left_Middle,
                       Route_Middle_Right,
                       Route_Right_Middle,
                       Route_Middle_Left,
                       Route_Enter_Left,
                       Route_Leave_Right,
                       Route_Enter_Right,
                       Route_Leave_Left);

   type One_Segment_State is (Occupied_Standing,
                              Occupied_Moving_Left,
                              Occupied_Moving_Right,
                              Reserved_Moving_From_Left,
                              Reserved_Moving_From_Right,
                              Free);

   type Segment_State_Type is
      record
         Left,
         Middle,
         Right : One_Segment_State;
      end record;

   type Signal_State_Type is
      record
         Left_Middle,
         Middle_Left,
         Middle_Right,
```

```
      Right_Middle: One_Signal_State;
   end record;

   Segment_State : Segment_State_Type := (others => Free);
   Signal_State  : Signal_State_Type  := (others => Green);

   function Correct_Signals return Boolean
   is
     (
        (if Signal_State.Left_Middle = Green then
              Segment_State.Left = Occupied_Moving_Right and
                Segment_State.Middle = Reserved_Moving_From_Left) and then
        (if Signal_state.Middle_Left = Green then
              Segment_State.Middle = Occupied_Moving_Left and
                Segment_State.Left = Reserved_Moving_From_Right) and then
        (if Signal_state.Middle_Right = Green then
              Segment_State.Middle = Occupied_Moving_Right and
                Segment_State.Right = Reserved_Moving_From_Left) and then
        (if Signal_state.Right_Middle = Green then
              Segment_State.Right = Occupied_Moving_Left and
                Segment_State.Middle = Reserved_Moving_From_Right));

   function Correct_Segments return Boolean
   is
     (
        (if Segment_State.Left /= Reserved_Moving_From_Right then
              Signal_State.Middle_Left = Red) and
        (if Segment_State.Middle /= Reserved_Moving_From_Left then
              Signal_State.Left_Middle = Red) and
        (if Segment_State.Middle /= Reserved_Moving_From_Right then
              Signal_State.Right_Middle = Red) and
        (if Segment_State.Right /= Reserved_Moving_From_Left then
              Signal_State.Middle_Right = Red));

   procedure Open_Route (Route: in Route_Type; Success: out Boolean)
     with
       Global => (In_Out => (Segment_State, Signal_State)),
       Depends => ((Segment_State, Success) => (Route, Segment_State),
                   Signal_State => (Segment_State, Route, Signal_State)),
       Post => Correct_Signals and Correct_Segments;

   procedure Move_Train (Route: in Route_Type; Success: out Boolean)
     with
       Global => (In_Out => (Segment_State, Signal_State)),
       Depends => ((Segment_State, Success) => (Route, Segment_State),
                   Signal_State => (Segment_State, Route, Signal_State)),
       Post => Correct_Signals and Correct_Segments;

end Railway;
```

Zdefiniowane zmienne **Segment_State** i **Signal_State** przechowują, odpowiednio, bieżący stan wszystkich segmentów i bieżący stan wszystkich sygnalizatorów.

W specyfikacji zdefiniowane następujące funkcje pomocnicze o wartościach boolowskich:

- **Correct_Signals** — wyraża warunek spełniony gdy stany świateł w sygnalizatorach nie grożą katastrofie,
- **Correct_Segments** — wyraża warunek spełniony gdy stany segmentów nie grożą katastrofie.

Zadeklarowano również dwie procedury:

- **Open_Route** — która dla pociągu poruszającego się trasą **Route**, gdy jest to możliwe (**Success** = True), otwiera trasę przełączając odpowiednie sygnalizatory (w przeciwnym przypadku **Success** = False),
- **Move_Train** — która przenosi pociąg poruszający się trasą **Route**, gdy jest to możliwe (**Success** = True), do następnego segmentu (w przeciwnym przypadku **Success** = False).

Zgodnie z kontraktem dla procedury **Open_Route**:
- procedura zmienia globalne zmienne **Segment_State** i **Signal_State**,
- nowe wartości **Segment_State** i **Success** zależą od wartości **Route** i **Segment_State**,
- nowa wartość **Signal_State** zależy od wartości **Segment_State**, **Route** i **Signal_State**,
- po wykonaniu procedury **Open_Route** zarówno stan sygnalizatorów jak i segmentów nie zagraża katastrofie.

Dla przykłady, w procedurze **Open_Route,** gdy pociąg chce przejechać z segmentu **Left** do segmentu **Middle** (**Route** jest równe **Route_Left_Middle**) sprawdzane jest czy segment **Left** jest w stanie **Occupied_Standing** a segment **Middle** jest w stanie **Free**. Jeśli tak, to segment **Left** przechodzi w stan **Occupied_Moving_Right**, segment **Middle** przechodzi w stan **Reserved_Moving_From_Left**, sygnalizator **Left_Middle** zapala się na zielono i **Success** przyjmuje wartość True. W przeciwnym przypadku **Success** przyjmuje wartość False.

Zgodnie z kontraktem dla procedury **Move_Train**:
- procedura zmienia globalne zmienne **Segment_State** i **Signal_State**,
- nowe wartości **Segment_State** i **Success** zależą od wartości **Route** i **Segment_State**,
- nowa wartość **Signal_State** zależy od wartości **Segment_State**, **Route** i **Signal_State**,
- po wykonaniu procedury **Move_Train** zarówno stan sygnalizatorów jak i segmentów nie zagraża katastrofie.

Dla przykładu, w procedurze **Move_Train**, gdy pociąg chce przejechać z segmentu **Left** do segmentu **Middle** (**Route** jest równe **Route_Left_Middle**) sprawdzane jest czy segment **Left** jest w stanie **Occupied_Moving_Right** a segment **Middle** w stanie **Reserved_Moving_From_Left**. Jeśli tak, to sygnalizator **Left_Middle** zapala się na czerwono, segment **Left** przechodzi w stan **Free**, segment **Middle** przechodzi w stan **Occupied_Standing** o **Success** przyjmuje wartość True. W przeciwnym przypadku, **Success** przyjmuje wartość False.

Polecenia

1. Napisz treść pakietu **Railway** definiującą odpowiednio procedury **Open_Route** i **Move_Train**.
2. Jeśli natrafisz na problemy ze zweryfikowaniem kontraktów dla tych procedur, to zastanów się czy nie należy dopisać dla nich odpowiedniego warunku wstępnego (Pre).

Zadanie 2

Rozpatrzmy następujący fragment programu w języku maszynowym procesora MOS 6502:

```
INIT    LDA #$00
        STA AB
        STA AB+1
        STA AA+1
        LDA A
        STA AA
        LDA B
        BEQ CONT
LOOP    AND #$01
        BEQ SHIFT
        LDA AA
        CLC
```

```
        ADC AB
        STA AB
        LDA AA+1
        ADC AB+1
        STA AB+1
SHIFT   ASL AA
        ROL AA+1
        LDA B
        LSR
        STA B
        BNE LOOP
CONT
```

Jeśli początkowo pod adresami A i B znajdowały się dwie wartości ośmiobitowe, to po dojściu do etykiety CONT w pamięci pod adresem AB znajdzie się szesnastobitowy iloczyn tych dwóch wartości.

Chcielibyśmy mieć możliwość weryfikacji poprawności takich programów jak powyższy. W tym celu należy przygotować pakiet **MOS_6502**, który definiuje instrukcje języka maszynowego procesora 6502 w postaci procedur. Działanie tych procedur powinno być wyspecyfikowane w postaci odpowiednich kontraktów.

Początkowy fragment specyfikacji pakietu **MOS_6502** mógłby wyglądać następująco:

```
package MOS_6502 with SPARK_Mode is

   type Byte is mod 2 ** 8;
   type Word is mod 2 ** 16;

   ACC      : Byte    := 0;
   CARRY    : Boolean := False;
   ZERO     : Boolean := False;

   RAM : array (Word) of Byte := (others => 0);

   function Val1 (Addr : Word) return Word is
     (Word (RAM (Addr)))
     with
       Post => Val1'Result = Word (RAM (Addr));

   function Val2 (Addr : Word) return Word is
     (Word (RAM (Addr)) + 256 * Word (RAM (Addr + 1)))
     with
       Post => Val2'Result = Word (RAM(Addr))+256*Word (RAM(Addr+1));

   procedure LDA_imm (Arg : Byte)
     with
       Post => ACC = Arg and ZERO = (Arg = 0);

   procedure LDA_abs (Arg : Word)
     with
       Post => ACC = RAM (Arg) and ZERO = (ACC = 0);

...

end MOS_6502;
```

W pakiecie **MOS_6502** zdefiniowane są między innymi typy ośmio- i szesnastobitowe (**Byte** i **Word**), akumulator **ACC**, flagi **CARRY** i **ZERO**, pamięć operacyjna (**RAM**) i pomocnicze funkcje zwracające w postaci szesnastobitowej wartości jednego (**Val1**) lub dwóch (**Val2**) kolejnych słów pamięci.

Najwygodniej przyjąć dla procedur wyrażających działanie instrukcji nazwy postaci **MNEMONIC_trybadresacji**.

Pamiętajmy, że SPARK nie będzie dowodził poprawności programów zawierających instrukcje skoku. Dlatego aby zrobić w nim pętle i rozgałęzienia użyjemy instrukcji strukturalnych **if then—end if** oraz **while loop—end loop**. Jako warunki sterujące tymi instrukcjami użyjemy flagi jakie dostępne są w mikroprocesorze MOS 6502 (np. **ZERO**, **CARRY** itp).

Poniżej zamieszczono procedurę, która oczekuje dostarczenia dwóch danych ośmiobitowych **In1** i **In2**. Wylicza ona ich iloczyn i wynik oddaje parametrem **Out1**.

```
with MOS_6502; use MOS_6502;

procedure Assembler (In1, In2 : Byte; Out1 : out Word)
is
   A    : constant Word := 1000; -- adres pierwszego czynnika
   B    : constant Word := 1001; -- adres drugiego czynnika
   AB   : constant Word := 1002; -- adres iloczynu
   AA   : constant Word := 1004; -- adres wielokrotnosci pierwszego
                                 -- czynnika
begin
   RAM (A) := In1;
   RAM (B) := In2;
   LDA_imm (0);                  -- INIT   LDA #$00
   STA_abs (AB);                 --        STA AB
   STA_abs (AB + 1);             --        STA AB+1
   STA_abs (AA + 1);             --        STA AA+1
   LDA_abs (A);                  --        LDA A
   STA_abs (AA);                 --        STA AA
   LDA_abs (B);                  --        LDA B
   while not ZERO loop           --        BEQ CONT
      AND_imm (1);               -- LOOP   AND #$01
      if not ZERO then           --        BEQ SHIFT
         LDA_abs (AA);           --        LDA AA
         CLC_imp;                --        CLC
         ADC_abs (AB);           --        ADC AB
         STA_abs (AB);           --        STA AB
         LDA_abs (AA + 1);       --        LDA AA+1
         ADC_abs (AB + 1);       --        ADC AB+1
         STA_abs (AB + 1);       --        STA AB+1
      end if;                    --
      ASL_abs (AA);              -- SHIFT  ASL AA
      ROL_abs (AA + 1);          --        ROL AA+1
      LDA_abs (B);               --        LDA B
      LSR_acc;                   --        LSR
      STA_abs (B);               --        STA B
   end loop;                     --        BNE LOOP
   Out1 := Val2 (AB);            -- CONT
end Assembler;
```

Parametry **In1** i **In2** umieszczane są w pamięci **RAM** pod adresami, odpowiednio **A** i **B**. Po wykonaniu kodu wynik odczytywany jest za pomocą funkcji **Val2** i umieszczany w parametrze **Out1**.

Polecenia

1. Dopisz w specyfikacji pakietu **MOS_6502** te procedury, wraz z kontraktami, które potrzebne są do wykonania powyższej procedury **Assembler**.
2. Napisz treść pakietu **MOS_6502** tak aby spełniał kontrakty zamieszczone w specyfikacji.
3. Dodaj w procedurze **Assembler** następujące aspekty: **with SPARK_Mode, Post => Out1 = Word (In1) * Word (In2)**.
4. Dopisz w procedurze **Assembler** odpowiednie niezmienniki pętli i asercje aby program **gnatprove** mógł automatycznie udowodnić poprawność procedury **Assembler**.
5. Napisz procedurę **Main** aby przetestować działanie procedury **Assembler** (powinna czytać dwie wartości typu **Byte**, wywołać procedurę **Assembler** i wydrukować zwróconą wartość typu **Word**).