2. SPARK Ada

(a) Introduction into Ada.

(b) Architecture of SPARK Ada.

(c) Language Restrictions in SPARK Ada.

(d) Data Flow Analysis.

(e) Information Flow Analysis.

(f) Verification Conditions.

(g) Example: A railway interlocking system.

Remark: Sections (a) - (f) will be heavily based on John Barnes: High Integrity Software. The SPARK approach. [Ba03].

(a) Introduction into Ada

Motivation for Developing Ada:

- Original problem of Department of Defence in USA (DOD):
  - Too many languages used and created for military applications ($>$450).
    - Languages largely incompatible and not portable.
    - Often minimal software available.
    - Competition restricted, since often only one vendor of compilers and other tools for one language.

Ada

- Existing languages too primitive.
  - No modularity.
  - Hard to reuse.
- Problems particularly severe in the area of embedded systems.
  - 56% of the software cost of DOD in 1973 for embedded systems.
- Most money spent on maintaining software, not developing it.

Ada

- Decision by DOD: Development of new standard programming language for military applications.
  - Name Ada = name of Ada Lovelace (1815-1852).
    - Wrote programs for Babbage's computer.
    - Therefore called "the first computer programmer".
- First release: Ada 83
  (1983 – same year C++ was released).
- Ada 95: Revision of Ada, integration of object-orientation.

Reasons for Using Ada in Crit. Syster

- Ada is the often **required standard for military applications in the US**, and has therefore adopted by the Western military industry.
- Software for **military applications forms a large portion of critical systems**.
- Therefore lots of **tools for critical systems support Ada**.

Reasons for Using Ada

- Ada was developed taking into account its use in **real-time critical systems:**
  - It is possible to write efficient code and code close to assembler code, while still using high level abstractions.
  - Features were built in which prevent errors (e.g. array bound checks, no writing to arbitrary memory locations).

Basic Syntax

- **Typing** is written as in Haskell
  - We write x : A for “x has type A” .
    - In Java or C this is written as “A x”.

Enumeration Types

- We have enumeration types. E.g.
  - **type** Gender **is** (male,female);

  introduces a new type having elements male and female.

Ranges in Ada

- Ada allows to define ranges.
  - **type** year **is** range 0..2100;

  defines the type of years as integers between 0 and 2100.
- We have as well subranges of enumerations:
  - If Day is of type

    (Mon, Tue,Wed,Thu,Fri,Sat,Sun) ,

    then we can form the subrange

    Mon . . . Wed

Loops

- The basic loop statement is an infinite loop with an exit statement. E.g.
  - ```
    Answer : Character;
    loop
        Put("Answer yes (y) or no (no)");
        Get(Answer);
        if Answer = 'y' then
          Put_Line ("You said yes");
          exit;
        elsif Answer = 'n' then
          Put_Line ("You said no");
          exit;
        else
          Put_Line ("You must type y or n");
        end if;
    end loop;
    ```
    ```

For Loops

- “For loops” are written as follows:
  - ```
    for N in 1..20 loop
       Put ("-");
    end loop;
    ```
    ```

In - out - parameters

- In Ada all parameters have to be labelled as
  - input parameters; symbol: **in**,
    - Value parameters.
      The same as a **non-var** parameter in Pascal.
  - output parameters; symbol: **out**,
  - input/output parameters; symbol: **in out**.
    - Reference parameters.
      The same as a **var** parameter in Pascal
- Example:

```
procedure ABC(A: in Float;
              B: out Integer;
              C: in out Colour)
```

In – Out – In Out Parameters

- **out** and **in out** parameters can only by instantiated by variables.
  - Assume for instance a procedure

    ```
    procedure Myproc(B: out Integer)
      is begin
      B:= 0;
      end Myproc;
    ```

  - It doesn't make sense to make a call

    ```
    Myproc(0)
    ```

    it only makes sense to call

    ```
    Myproc(C)
    ```

    where C is a variable that can be changed.
- We see as well that the variable we instantiate it with cannot be an **in** parameter itself, because then it cannot be changed.

In – Out – In Out Parameters

- **in** parameters can be instantiated by arbitrary terms.
  - Consider:

```
procedure Myproc(B: in Integer,C: out Integer)
  is begin
  C:= B;
  end Myproc;
```

  - It makes sense to call
    Myproc(0,D)
    where D is a variable that can be changed.
  - It makes as well sense to call
    Myproc(3 + 5,D)
    or
    Myproc(f(E,F),D)
    where f is a function with result type Integer.

In – Out – In Out Parameters

- In **Java**
  - Parameters are always passed by value, and behave like **in** parameters.
  - However the value of an object which is passed as a parameter is the pointer to that object. The original value remains, but the object itself can be changed.
  - This is different from Ada where if a record is passed on as an **in** parameter, no fields can be changed.

Example Parameters in Java

```
public static void exchange(int a, int b){
   int tmp = a;
   a = b;
   b = tmp;}
-- The above doesn't exchange a and b

-- Define a wrapper class for integers
class myint{
   public int theint ;
   public myint(int theint){
      this.theint = theint;}
}
```

Example Parameters in Java

```
public static void exchange(myint a, myint b){
   myint tmp = a;
   a = b;
   b = tmp;}
-- The above doesn't exchange a and b

public static void exchange1(myint a, myint b){
   myint tmp = new myint(a.theint);
   a.theint = b.theint;
   b.theint = tmp.theint;}
-- The above does exchange a and b
```

Record Types

- A record is essentially a class without methods.
- Syntax in Ada:

```
type Person is
   record
      yearOfBirth : Integer
      gender : Gender
   end record
```

- This example declares a type Person which has two fields yearOfBirth and gender.
- If a: Person, then we have that a.yearOfBirth : Integer and a.gender : Gender.
- One can make assignments like a.gender = male.

Record Types

- If a : Person one can give a value to its fields by saying
  - a = (1975,male)
  - or
    a = (year => 1975,gender => male)

Jump over Variant Records

Variant Records

- Variant record means that we have a record, s.t. the type of one field depends on the value of some other type.
- Example:

```
type Gender is (Male, Female);

type Person(Sex: Gender:= Female) is
   record
      Birth: Date;
      case Sex is
         when Male =>
            Bearded: Boolean;
         when Female =>
            Children: Integer;
      end case;
   end record;
```

Variant Records

- In the above example the type Gender is defined as a type having two elements, namely Male and Female.
- Person is a type, which has a field Sex, Birth, and depending on the field Sex either a field Bearded or a field Children.
- By default, Person.Gender = Female.
- We can have elements of type Person and of type Person(Male).
  - If John: Person(Male), then John.Sex=Male.

Variant Records

- Whether the field of a variant record accessed is in the variant used **cannot** always **be checked at compile time**.
  - For instance, if we have
    a: Person ,
    a code which accesses
    a.Bearded
    compiles, even if it is clear that
    a.Sex=Female .
  - But this will cause a run time error.
  - In case of
    a: Person(Female) ,
    a warning is issued at compile time if
    a.Bearded
    is accessed.

Example

(For simplicity Date = Integer)

```
John: Person(Male);
Tom : Person;

begin
   John:= (Male,1963,False);
   -- John.Gender:= Female; --would cause compile error
   Tom:= (Male, 1965,False);
   Tom.Children := 5; -- Compiles okay but runtime error.
   -- Tom.Sex := Female; --would cause compile error
```

Variant Records

- Variant records are a restricted form of **dependent types** (see module on interactive theorem proving).
  - In **dependent type theory,** as introduced there, such kind of constructs can be used in a **type safe way.**

Polymorphism

- Ada allows to introduce a new name for a type.
  - Then functions for the new type inherit the functions from the old type.
  - However functions can be **overridden**.

Example for Use of Polymorphism

```
type Integer is ...; -- Some definition.

function "+" (x, y : Integer) return Integer

type Length is new Integer;
-- We can use a + b : Length for a,b : Length

type Angle is new Integer;
function "+" (x, y : Angle) return Angle
-- Now we have overridden + for Angle by a new function.
```

(Example taken from S. Barbey, M. Kempe, and A. Strohmeier: Object-Oriented Programming with Ada 9X. http://www.adahome.com/9X/OOP-Ada9X.html)

Deciding Which Function to Use

- Note in case of polymorphism, which function to be chosen, can be decided at compile time, since it only depends on the (fixed) type of the argument.

Jump over Object Orientation in Ada

Tagged Types

- Record types can be extended.
  - But only if they had been declared to be **tagged**.
  - Tagged means that each variable is associated with a tag which identifies which type it belongs to.
  - This is necessary in case we have a class-wide type (see below) to decide which instance of a function is used.
    - We might define a function which takes an element of one type and a function which takes as argument an element of an extended type.

Example

```
type Student is tagged
   record
      StudentNumber   :   Integer;
      Age             :   Integer;
   end record;

type Swansea_Student is new Student with null record;
-- We extend Student but without adding a new component -- We
```

Example

- Swansea_Student is a subtype of Student.
- Any function having as argument Student can be applied to a Swansea_Student as well.
- We can override a function for Student by a function with argument Swansea_Student.
- Note that which function to be chosen can be decided at compile time, since it only depends on the (fixed) type of the argument.

Class-Wide Types

- Associated with a tagged type such as Student above is as well a Class-wide type.
  - Denoted by Student'Class.
- An element of Swansea_Student is not an element of Student, but can be converted into an element of Student as follows
  A : Swansea_Student := ...
  B : Student = Student(A);
- However an element of Swansea_Student is an element of Student'Class:
  C : Student'Class = A;

Dynamic Dispatch

- Assume an element A : Student'Class
- Assume a function
  function f (X : Student) return ..
- Assume this function is overridden for Swansea_Student:
  function f (X : Swansea_Student) return ..
  - Without this function the function
    function f (X : Student) return ..
    would be applicable to X : Swansea_Student as well.
    Since it is overriden, the new function is the one to be applied.

Dynamic Dispatch

- We can apply f to A : Student'Class.
  - If A was originally an element of Student, the first version of the function is applied.
  - If A was originally an element of Swansea_Student, the second version of the function is applied.
  - At compile time it is usually not known, which of the two cases applies, therefore the decision which function to choose depends on the **tag** of A.
    - The tag tells which type it originally belongs to.
  - This is called **dynamic dispatch** or **late binding**.

Class-Wide Types and Java/C++

- In Java we could say we have only class-wide types.
- In C++ we have as well only class-wide types, but one can control subtyping by using the keyword **virtual**:
  - Only **virtual** methods have late binding.
  - Only virtual methods can be overridden.

Class-Wide Types

- Problem of inheritance: properties are inherited remotely, which makes it difficult to verify programs.
  - If one has a class-wide type A with subtype B, and two different functions f(x:A) and f(x:B), then one
    - might expect that a call of f(a) for a:A refers to the first definition,
    - but in fact, if a:B it will refer to the second definition.
    - That redefinition could have been done by a different programmer in a different area of the code.
- However elements of a subtype in the sense of the restriction of the range of a type (e.g. Integer restricted to 0 . . . 20) can be assigned to elements of the full type.

Object-Orientation in Ada

- Ada's concept of object-orientation is restricted.
  - Ada allows only to form record types, and class-wide types.
  - So instead of
    - having a method f of a class C with parameters x1:A1 ,. . ., xn:An, and then writing O.f(x1 ,. . ., xn) for a method call for object O: C,
    - one has to introduce a polymorphic function f with arguments X: C'Class,x1:A1 ,. . ., xn:An, and then to write f(O,x1 ,. . ., xn) for the call of this function.

Object-Orientation in Ada

- **Disadvantage:** The definition of the functions can be defined completely separted from the definition of the class.
- **Advantage:** More flexibility since one doesn't have to decide for a function, to which object it belongs to.

(b) Architecture of SPARK Ada

- SPARK Ada is a Subset of Ada.
  - Original definition by B. Carré and T. Jennings, Univ. of Southampton, 1988.
  - Several revisions carried out by Praxis Critical Systems Ltd.
  - Adapted to Ada95
  - Commercial tools available from Praxis Critical Systems.

SPARK Ada

- Annotations to Ada.
  - Some required for data and information flow analysis.
  - Others allow to generate and prove verification conditions.
  - It is as well possible to integrate unchecked or even unimplemented Ada code.
- SPARK Ada code compiles with standard Ada compilers.

Architecture of the SPARK Ada

- **SPARK-examiner.**
  - Verifies the following:
    - Correct Ada syntax.
    - SPARK-subset of Ada chosen, as described above.
  - Carries out three levels of analysis:
    - **Data flow analysis**.
    - **Information flow analysis.**
    - **Generation of verification conditions.**

- **SPADE-simplifier.**
  - Simplifies verification conditions of the examiner. Trivial ones are already proved.

Architecture of the SPARK Ada

- **SPADE-proof-checker.**
  - Proof tool for interactively proving verification conditions.

Three Levels of Analysis

- **Data flow analysis.**
  - Checks input/output behaviour of parameters and variables.
  - Checks initialisation of variables.
  - Checks that changed and imported variables are used later (possibly as output variables).
- **Information flow analysis.**
  - Verifies interdependencies between variables.
- **Verification conditions.**
  - Generation of verification conditions, which allow to prove correctness of programs.

Three Levels of Analysis

Idea is that the 3 different levels of analysis are applied depending on the criticality of the program.

(Some parts of the program might not be checked at all).

Annotations

- Certain annotations are added to the programs.
  - Specific to the 3 levels of analysis.
- Written as **Ada comments**:
  - Ignored by Ada compilers.
  - Used by the SPARK Ada tools.
  - Syntax: start with --#, e.g.
    - --# **global in out** MyGlobalVariable;

(c) Lang. Restrict. in SPARK Ada

Factors for Programming Languages Addressed by SPARK Ada

- **Logical Soundness.**
  - Problem: statement like Y:=F(X) + G(X):
    Order of evaluation not specified.
    Problem if F(X) and G(X) have side effects.
    - E.g. F(X) has effect Z:=0,
      G(X) has effect Z:= 1.

SPARK Ada Concepts

- Solution in many languages: define order of evaluation.
  Not possible, if SPARK Ada should compile on standard compilers.
- Solution in SPARK Ada:
  Functions are not allowed to have side-effects.

SPARK Ada Concepts

- **Simplicity of language definition.**
  - Omission of too complex principles.
    - No variant records.
      - (Dependent types, but no complete compile time checking).
    - No threads (concurrency).
    - No generic types.

SPARK Ada Concepts

- **Expressive power.**
  - Hiding of variables allowed.
  - Allows to specify strong assertions about variables (e.g. that a variable is only read but not written, or only written but not read).

SPARK Ada Concepts

- **Security.**
  - Features already supported by Ada:
    - Array bound are checked.
    - Programs does not stray outside the computational model (one cannot jump or write to an arbitrary memory location).
  - Feature enforced by SPARK Ada:
    - In order to be verifiable at compile-time:
      - Constraints (array bounds, ranges) have to be static (determined at compile time).
      - (This makes it as well easier to create verification conditions, since one doesn't have to switch between integers, anonymous subtypes of the integers and declared subtypes of the integers).

SPARK Ada Concepts

- **Verifiability**
  - **Support of automated verifiability**:
    Extra annotations
    - control of data flow,
    - control of information flow,
    - proof annotations (more below).

SPARK Ada Concepts

- Support of Verifiability (Cont.)
  - **Support of verification by hand:**
  - Every fragment of code has a single entry point and limited exit points.
    - Procedures have no **return** statements (= exit from the procedure).
    - Functions have exactly one **return** statement, which must be the last statement.
    - One can break out of a **loop** by the so called **exit** statement, but this exit cannot be
      - inside another loop,
      - inside nested if-then-elses,
      - inside another for-loop.

SPARK Ada Concepts

- **Example:**
  - The following is legal
    (**loop** is an infinite loop
    what is in other languages written as **while (true)**):

```
loop
   . . .
   if some_condition then
         . . .
         exit;
   end if;
   . . .
end loop;
```

SPARK Ada Concepts

- The following is illegal:

```
loop
   . . .
   if some_condition then
         if another_condition then
               . . .
               exit;
         end if;
            . . .
   end if;
end loop;
```

SPARK Ada Concepts

- Exit from loops only possible to innermost loop:
  The following Ada code is illegal in SPARK Ada:

```
Outer_Loop:
for J in 1 . . . M loop
    . . .
    for K in 1 . . . M loop
        . . .
        exit Outer_loop when A = 0;
        . . .
        end loop;
      . . .
end loop Outer_Loop;
-- when exit statement is executed,
-- we continue here
```

SPARK Ada Concepts

- However, the following is legal (if one makes the range more explicit, see later):

```
for J in 1 ... M loop
      ...
      exit when A = 0;
      ...
end loop;
-- when exit statement is executed,
-- we continue here
```

Problem with the Exit Statement

- Consider the following Ada code:

```
while (X > 0.0) loop
      Y := Y - 1.0;
      exit when Y > 5.0;
      X := X - 1.0;
end loop;
```

- If the exit statement is hidden in the code the reader might assume that once the loop is finished we have $\neg(X > 0.0)$, which is not the case if the loop is ended because of the exit statement.

SPARK Ada Concepts

- **No structures which are too complex to grasp.**
  - Almost **no subtyping.**

Omit Details about Subtyping

SPARK Ada Concepts

- **Details about restrictions on subtyping in SPARK Ada.**
  - No derived types (essentially a new name for an existing type or a subrange for an existing type).
  - No type extension (extension of a record by adding further components).
  - No class-wide types (see slides on object-orientation in Subsection a).
    Therefore no **late binding** (dynamic dispatch, called dispatching in Ada).

SPARK Ada Concepts

- **Language should be as explicit as possible.**
  - **No polymorphism** (ie. that an operation is defined for different types):
    - **No overloading** of functions.
    - **No array sliding**:
      Assignment, comparison, operations on arrays only allowed on arrays with same array index sets.
      - Ada behaves in an unexpected way in this case.
      - No concatenation of arrays allowed.
      - However, for strings such kind of operations are allowed.
    - No **default parameters, default record components.**
    - However standard $+$, $*$ are overloaded.

SPARK Ada Concepts

- (Language should be as explicit as possible, cont.)
  - No **anonymous subtypes.**
    - Instead of:
      type Vector **is array** (0. . .100) **of** Integer;
      one has to write
      type Vector_index **is range** 0. . .100;
      type Vector **is array** (Vector_index) **of** Integer;
    - **Exception:** In **for loops**, one can use anonymous subtypes, but one has to explicitly introduce the type, one is forming a subtype of:
      - Instead of
        for I **in** 0. . .100 **loop**
        one has to write
        for I **in** Integer **range** 0. . .100 **loop**

SPARK Ada Concepts

- (Language should be as explicit as possible, cont.)
  - **Unique names of entities** at a given place:
    - Package variables have to be used explicitly:
      A variable X of a package Mypackage has to be referenced as Mypackage.X

SPARK Ada Concepts

- **Bounded space and time requirements.**
  - **Recursion** disallowed.
  - No **arrays** without bounds or with dynamically allocated bounds.
    - Can be declared, but only subtypes of it can be used.
  - No **pointers** (called access types in Ada).
  - The above guarantees bounded space.
    - The space used is exactly that used by all the variables declared in all packages involved.
    - No additional space is allocated by pointers, allocation of space for arrays.
    - No additional space is used by recursion.

SPARK Ada Concepts

- Bounded time difficult to guarantee.
  - One can theoretically determine whether a program using bounded space terminates:
    - Such a program has **finitely many states**.
    - (Only finite amount of information can be stored in bounded space).
    - If a program doesn’t terminate, it needs to enter the **same state twice**.
    - That can be checked.
  - But this is only a **theoretical result**.
  - The bounds for termination easily exceed the life time of the universe.

SPARK Ada Concepts

- One could guarantee bounded time by allowing only for-loops.
  - With some further restrictions one could even obtain polynomial time bounds.
  - But in general disallowing unrestricted loops would be too restricting.
  - Interactive programs usually require at least one unrestricted while loop.
- SPARK Ada allows dynamic for loops, which makes it difficult to guarantee any reasonable time bounds.

(d) Data Flow Analysis

Data Flow Analysis – Parameters

- Remember the use of in/out/in-out parameters in Ada.
  - in - paramters are input parameters.
    - Can be instantiated by terms.
    - Parameter can be used but not be modified.
  - out- parameters.
    - Can only be instantiated by variables.
    - Input cannot be used, parameter can be modified.
  - in-out- parameters.
    - Can only be instantiated by variables. Input can be used, parameter can be modified.

Data Flow Analysis – Parameters

- Examiner verifies that
  - **Input parameters** are
    - **not modified**,
    - but **used at least once**,
      - not necessarily in every branch of the program;
  - **output parameters** are
    - **not read before being initialised**,
    - **initialised**.
      - Has to take place in all branches of the program.
  - **input/output parameters** are
    - **read,**
    - and **changed.**

Data Flow Analysis – Global Variables

- Global variables must be given status as input or output or input/output variables by **annotations**.
  - Syntax examples:
    - **procedure** test (X : in out Integer)
      --# **global in** A; **out** B; **in out** C;
    - Or (after the same procedure declaration line)
      --# **global out** B;
    - Or (after the same procedure declaration line)
      --# **global in out** C;
- Data flow analysis carries out the same analysis as for parameters.

Data Flow Analysis – Functions

- Functions can have only input parameters (no keyword required).
- Functions have only read access to global parameters. Therefore the syntax for global parameters is simply
  --# **global** A;
  or
  --# **global** A,B,C;
- Neither parameters nor global variables can be changed.
  Therefore functions don't have side effects.

Data Flow Analysis – Packages

- Package variables = variables global to a package.
- Package variables must be declared by annotations.
  Syntax example:
  package test
  --# **own** X,Y;
- If a variable is initialised it has to be declared; whether it is initialised will be verified.
  Syntax example:
  package test
  --# **own** X,Y;
  --#**initializes** X;
- If a variable is not initialized in a package SPARK Ada issues a warning.

Data Flow Analysis – Packages

- If a package is used it has to be declared:
  Syntax example:
  −−#**inherits** Mypackage;
  - In Ada a package inherits anything from
    - a package surrounding it,
    - and packages inherited by **with** · · · .
      - **with** is the import statement in Ada.
  - In SPARK-Ada we have to specify, which ones of these are actually used in that package.

Example (Data Flow Analysis)

- Consider the following wrong program, which should exchange X and Y:

```
procedure Exchange(X,Y: in out Float)
is
  T:Float;
begin
  T:= X; X:= Y; Y:= X;
end Exchange;
```

- Mistake: body should be:
  T:= X; X:= Y; Y:= T;

Example (Data Flow Analysis)

```
procedure Exchange(X,Y: in out Float)
is
  T:Float;
begin
  T:= X; X:= Y; Y:= X;
end Exchange;
```

- Data flow analysis results in 3 error messages:
  - T:= X is an ineffective statement.
  - Import of initial value of X is ineffective.
  - T is neither referenced nor used.
- Note that SPARK Ada doesn't notice that Y doesn't change, so Y is really an **in** parameter.
  - It would be complicated, to deal with such kind of phenomena in general.

Example 2 (Data Flow Analysis)

- Here is another wrong program, which should exchange X and Y:

```
procedure Exchange(X,Y: in out Float)
is
begin
 X:= Y; Y:= X;
end Exchange;
```

- Data flow analysis results in error message:
  - Importation of initial value of X is ineffective.

Example 3

- Assume a procedure with body

```
Y := X;
Z := Y;
```

- Then
  - The value of X is used, but never changed.
    - So **X** is an **input variable** (keyword “**in**”).
  - The value of Y is changed, but the original value is never used
    - It seems to be used in the 2nd line, but what is used is actually the value of X.
    - So **Y** is an **output** variable, but not an input variable (keyword “**out**”).

Example 3

```
Y  :=  X;
Z  :=  Y;
```

- The value of Z is changed, but not used.
  - So **Z** is an **output** variable (keyword “**out**”).

(e) Information Flow Analysis

- Additional annotations on how variables depend on each other.
  Syntax examples:
  --#**derives** X **from** Y;
  or, if we have several dependencies

  --#**derives** X **from** Y &
  --#   Y **from** X;

  or, if X gets a value independent of other variables,
  --#**derives** X **from** ;
- Often the new value of a variable depends on its own values, so we have
  --#**derives** X **from** X;
  or
  --#**derives** X **from** X, Y;

Information Flow Analysis

- Information flow verifies that these dependencies are fulfilled

Example

- Consider the following wrong program, which should exchange X and Y and count the number of exchanges in Z:

```
procedure ExchangeAndCount(X,Y,Z: in out Integer)
--#derives X from Y &
--#        Y from X &
--#        Z from Z;
is
T: Integer;
begin
        T:= X; X:= Y; Y:= T; Z:= Z + T;
end ExchangeAndCount;
```

- The error is the line Z:= Z + T;
  should be replaced by Z:= Z + 1;

Example

- Data flow analysis succeeds without problems.
- Information flow analysis gives warning, since Z depends on Z and X (via T).

SW

Example 2

- Assume as in data flow analysis example 3 a procedure with body

$$\begin{array}{lll} Y & := & X \\ Z & := & Y \end{array}$$

  - **X does not derive from anywhere**, since it is not changed (no output variable).
  - **Y derives from X**.
  - **Z derives from X**, (**not** from Y!).
  - Note that
    - only output and input/output variables can derive from somewhere,
    - and they can only derive from input and input/output variables.

(f) Verification Conditions.

Verification Conditions for Procedures without Loops

- Two kinds of annotations are relevant:
  - Pre-conditions, e.g.:
    --#**pre** M >= 0 **and** N < 0;
  - Post-conditions, e.g.:
    --#**post** M = M~ +1;
- Then examiner generates formulas which express:
  - If the pre-conditions hold, and the procedure is executed, afterwards the post-condition holds.
- If there are no pre-conditions, then the formula expresses that the post-condition holds always.

Hoare Calculus

- The basic formal system which takes pre- and post-conditions in an imperative program and generates verification conditions, is called the **Hoare Calculus**
- Named after Sir Tony Hoare (Professor emeritus from Oxford and Microsoft Research, Cambridge), Computer Scientist.
- Winner of the Turing Award 1980.
- Knighted by the Queen for his achievements in Computer Science.

Tony Hoare with his Wife Jill

(After having been knighted by the Queen.)

Verification Conditions

- In post-conditions,
  - X∼ stands for the value of X before executing the procedure,
  - X stands for the value after executing it,
  - e.g. X=X∼+1; expresses:
    The value of X after executing the procedure is the value of X before executing it + 1.
- Formulas are built using:
  - Boolean connectives **and**, **or**, **not**, **xor**, ->, <->
  - quantifiers **for all**, **for some**.

Remark on X∼

- X∼, Y∼ can only be used if X, Y are input and output parameters or global in out variables.
  - If X is only an input variable, it won't change, so X is always equal to X∼.
  - If X is only an output variable, then the initial value of X is undefined.

Example (Verification Conditions)

- Assume the following wrong program:

```
procedure Wrongincrement(X: in out Float);
--# derives X from X;
--# pre      X >=0.0;
--# post     X = X~ + 1.0;
is begin
  X := X + X;
end Wrongincrement;
```

- The mistake in this program is that the body should read $X := X + 1.0;$ instead of $X := X + X;$.
- The post-condition is not fulfilled in general. (The pre-condition has no significance in this example and is only used in order to introduce the syntax).

Example (Cont.)

```
procedure Wrongincrement(X: in out Float);
--# derives X from X;
--# pre      X >=0.0;
--# post     X = X~ + 1.0;
is begin
  X := X + X;
end Wrongincrement;
```

- When going through the program we see that at the end
  - $X = X\sim + X\sim$.
- Therefore, replacing X by $X\sim + X\sim$ the post condition reads:
  - $X\sim + X\sim = X\sim + 1.0$

Example (Verification Conditions)

- Since in the resulting formula, each variable has attached the symbol $\sim$, the examiner uniformely omits the $\sim$ and generates the formula

  H1: x$>=$0.

  H2: true.

  $->$

  C1: x+x =x + 1.

  which is not provable.

Example (Verification Conditions)

- The statements of the form "true" generated by the examiner above correspond probably to conditions of the variables.
  - If one has a variable X in a range $0 \ldots 10$, then for this variable one would have a pre-condition $0 \leq \mathrm{x\ and\ x} \leq 10$.
    - (SPARK-Ada puts variables into lowercase.)
  - Integer variables lie as well in a range, namely between the minimum integer and the maximum integer.
    - These values are called integer__first and integer__last.
    - So there we get additionally the conditions $x \geq \mathrm{integer__first}$ and $x \leq \mathrm{integer__last}$.

Example (Verification Conditions)

- For floating point numbers, there are no conditions on the ranges, so the examiner generates “true”.
- In Ada, variables are not case-sensitive. Therefore the examiner replaces variables by lower case versions, in order to get a unique form.

Example (Verification Conditions)

- The above formulae have to be treated as being implicitly universally quantified,
  - so the formula above stands for

$$\forall \mathrm{x}.((\mathrm{x} \geq 0.0 \wedge \mathrm{true}) \rightarrow (\mathrm{x} + \mathrm{x} = \mathrm{x} + 1.0))$$

- This corresponds to what the correctness of the function means:
  - **For all inputs** x, we have that, if x fulfils the precondition, then after executing the program it fulfils the post condition.
- That the above formula is unprovable can be shown by giving a counter example:
  - Take $\mathrm{x} := 0.0$. Then $(\mathrm{x} \geq 0.0 \wedge \mathrm{true})$ holds, but not $(\mathrm{x} + \mathrm{x} = \mathrm{x} + 1.0)$.

Example 2 (Verification Conditions)

- Assume the correct program:

```
procedure CorrectIncrement(X: in out Float);
--# derives X from X;
--# pre       X >=0.0;
--# post      X = X~ + 1.0;
is
begin
  X := X + 1.0;
end CorrectIncrement;
```

Example 2 (Cont.)

```
procedure Correctincrement(X: in out Float);
--# derives X from X;
--# pre      X >=0.0;
--# post     X = X~ + 1.0;
is
begin
  X := X + 1.0;
end Correctincrement;
```

- When going through the program, we see that at the end
  - $X = X\sim + 1.0$.
- Therefore the post condition reads
  - $X\sim + 1.0 = X\sim + 1.0$.

Example 2 (Verification Conditions)

- The examiner replaces again X$\sim$, Y$\sim$ by X, Y respectively and generates the formula:

  H1: x $>=$ 0 .
  H2: true.
  $->$
  C1: x + 1 = x + 1.

- "true" stands again probably for trivial conditions on the ranges of the variable (see above).
- The simplifier shows that this is provable

Check Conditions

- One can insert in between the procedures a check condition.
- Now the formulas express:
  - From the pre-condition follows at that position the check-condition.
  - From the pre-condition and the check-condition at that position follows the post-condition.
  - Check conditions serve therefore as intermediate proof-goals.

Example Check Condition

Consider the program:

```
procedure Onecheckfun(X: in out Integer)
  --# derives X from X;
  --# pre X = 0;
  --# post X = 2;
is
  begin
    X:= X+1;
    --# check X = 1;
    X:= X +1;
  end Onecheckfun;
```

- That the pre condition implies the check condition is the following verification condition:
  $X\sim = 0 \rightarrow X\sim+1 = 1;$

Example Check Condition

Consider the program:

```
procedure Onecheckfun(X: in out Integer)
  --# derives X from X;
  --# pre X = 0;
  --# post X = 2;
is
  begin
    X:= X+1;
    --# check X = 1;
    X:= X +1;
  end Onecheckfun;
```

- That the pre and the check condition implies the post condition is the following verification condition:
  $(X\sim = 0 \wedge X\sim+1 = 1) \rightarrow (X\sim+1) + 1 = 2;$

Generated Verification Conditions

- The Examiners generates:

```
H1: x = 0 .
H2: x >= integer__first .
H3: x <= integer__last .
      ->
C1: x + 1 = 1 .
```

and

```
H1:x = 0 .
H2:x >= integer__first .
H3:x <= integer__last .
H4:x + 1 = 1 .
      ->
C1:x + 1 + 1 = 2 .
```

Check and Assert

- There is as well an assert directive with similar syntax as the check directive.
- Syntax --# **assert** T $>$ 0.0;
- The difference is:
  - If one has an assert directive, then the verification condition is that
    - From the pre condition follows the assert condition (taking into account changes to variables in between).
    - From the **assert condition** follows the post condition.

Check and Assert

- If one has a check directive, then the first verification condition is as before but the second verification condition states instead
  - From the **pre condition and the check condition** follows the post condition.
- So the pre-condition is not lost if one has a check condition, which makes it easier to prove this result.

- Assert condition should mainly be used in loops (where reuse of the pre-condition is not possible; see below).

Example Assert Condition

Consider the program:

```
procedure Onecheckfun(X: in out Integer)
  --# derives X from X;
  --# pre X = 0;
  --# post X = 2;
is
  begin
    X:= X+1;
    --# assert X = 1;
    X:= X +1;
  end Onecheckfun;
```

- The implication from the pre condition to the check condition is as before.
- That the assert condition implies the check condition is the following: $X\sim = 1 \rightarrow X\sim +1 = 2;$

Generated Verification Conditions

- The Examiners generates:

```
H1: x = 0 .
H2: x >= integer__first .
H3: x <= integer__last .
    ->
C1: x + 1 >= integer__first .
C2: x + 1 <= integer__last .
```

```
H1: x = 0 .
H2: x >= integer__first .
H3: x <= integer__last .
H4: x + 1 >= integer__first .
H5: x + 1 <= integer__last .
    ->
C1: x + 1 = 1 .
C2: x + 1 >= integer__first .
C3: x + 1 <= integer__last .
C4: x = 0 .
```

Generated Verification Conditions

H1: x = 1 .
H2: x >= integer__first .
H3: x <= integer__last .
H4: x~ = 0 .
    ->
C1: x + 1 >= integer__first .
C2: x + 1 <= integer__last .

H1: x = 1 .
H2: x >= integer__first .
H3: x <= integer__last .
H4: x~ = 0 .
H5: x +1 >= integer__first .
H6: x +1 <= integer__last .
    ->
C1: x + 1 = 2 .

Return Conditions

- If one has functions, one can either state the result of the function:
  E.g.
  --# **return** X+1
  expresses: the result is X+1.
  or one can associate with the result a variable and a condition. E.g. one can write:
  --# **return** X => X > Y;
  if Y is a parameter or a global parameter.
  The example expresses:
  the returned value is > Y.

Example

```
function Tmpfun (X: Integer) return Integer
 –# pre X > 0;
 –# return R => R > 1;
 is
 begin
    return X + 1;
 end Tmpfun;
```

- The generated verification condition is
  $X > 0 \to X{+}1 > 1$

Example 2

```
function Tmpfun2 (X: Integer) return Integer
  –# pre X > 0;
  –# return X + 1;
  is
  begin
     return X + 1;
  end Tmpfun;
```

- The generated verification condition is
  $X > 0 \rightarrow X+1 = X + 1$

Conditionals

- If we have a conditional (i.e. an “if” clause), one has several paths through the program.
  - E.g. if we have the following program:

```
procedure Simple(A: in out Float)
  --# derives A from A;
  --# pre A <= 0.0;
  --# post A > 0.0;
  is begin
  if A >= 1.0
     then A := -1.0;
     else A := 2.0;
     end if;
  end Simple;
```

Conditionals

```
procedure Simple(A: in out Float)
  --# derives A from A;
  --# pre A <= 0.0; post A > 0.0;
  is begin
  if A >= 1.0 then A := -1.0;
     else A := 2.0; end if;
  end Simple;
```

Then the verification conditions are:

- From the precondition follows the postcondition, in case $A\sim \geq 1.0$, (then A at the end is -1.0):
  - $(A\sim \leq 0.0 \wedge A\sim \geq 1.0) \rightarrow -1.0 > 0.0$
- From the precondition follows the postcondition, in case $\neg(A\sim \geq 1.0)$, (then A at the end is 2.0):
  - $(A\sim \leq 0.0 \wedge \neg(A\sim \geq 1.0)) \rightarrow 2.0 > 0.0$.

Generated Verif. Cond.

H1: a $<=$ 0 .
H2: true .
H3: a $>=$ 1 .
    $->$
C1: -1 $>$ 0 .

H1: a $<=$ 0 .
H2: true .
H3: not (a $>=$ 1) .
    $->$
C1: 2 > 0 .

Both formulas (including the first one!) are provable and are proved by the simplifier.

Conditionals

- One can as well have check and assert directives in branches, e.g.:

```
procedure Simple2(A: in out Float)
  --# derives A from A;
  --# pre A <= 0.0; post A > 0.0;
  is begin
    if A >= 1.0
    then A := -1.0;
            --# check A = 1.0;
    else
            A := 2.0;
            --# check A = 2.0;
    end if;
  end Simple2;
```

Conditionals

```
procedure Simple2(A: in out Float)
  --# derives A from A;
  --# pre A <= 0.0; post A > 0.0;
  is begin
     if A >= 1.0 then A := -1.0;--# check A = 1.0;
                  else  A := 2;    --# check A = 2.0;
  end if; end Simple2;
```

- In this example the verification conditions are that
  - from the pre-condition follows the check condition, provided the if-condition is fulfilled/is not fulfilled,
  - and that from the pre condition, the check condition, and the fact that the if-condition is fulfilled/is not fulfilled, follows the post condition.

Conditionals

```
procedure Simple2(A: in out Float)
  --# derives A from A;
  --# pre A <= 0.0; post A > 0.0;
  is begin
     if A >= 1.0 then A := -1.0;--# check A = 1.0;
                  else  A := 2;    --# check A = 2.0;
  end if; end Simple2;
```

- So we have the following condition:
  - Precondition implies check condition in branch 1: $(A\sim \leq 0.0 \wedge A\sim \geq 1.0\ ) \rightarrow -1.0 = 1.0$
  - Precondition implies check condition in branch 2: $(A\sim \leq 0.0 \wedge \neg(A\sim \geq 1.0)) \rightarrow 2.0 = 2.0$

Conditionals

```
procedure Simple2(A: in out Float)
  --# derives A from A;
  --# pre A <= 0.0; post A > 0.0;
  is begin
    if A >= 1.0 then A := -1.0;--# check A = 1.0;
                 else  A := 2;     --# check A = 2.0;
  end if; end Simple2;
```

- From the precondition and the checkcondition in branch 1 follows the post condition (where A = 1): $(A\sim \leq 0.0 \wedge A\sim \geq 1.0 \wedge -1.0 = 1.0) \rightarrow -1.0 > 0.0.$
- From the precondition and the checkcondition in branch 2 follows the post condition (where A = 1): $(A\sim \leq 0.0 \wedge \neg(A\sim \geq 1.0) \wedge 2.0 = 2.0) \rightarrow 2.0 > 0.0.$

Generated Ver. Cond.

H1: a <= 0 .
H2: true .
H3: a >= 1 .
    ->
C1: -1 = 1 .

H1: a <= 0 .
H2: true .
H3: not (a >= 1) .
    ->
C1: 2 = 2 .

Generated Ver. Cond.

H1: a $<=$ 0 .
H2: true .
H3: a $>=$ 1 .
H4: -1 = 1 .
$->$
C1: -1 > 0 .

H1: a $<=$ 0 .
H2: true .
H3: not (a $>=$ 1) .
H4: 2 = 2 .
$->$
C1: 2 > 0 .

All 4 verification conditions are provable and are in fact proved by the simplifier.

Using Assert Above

- If we had used assert above, then we get similar statements from the pre condition to the conditional.
- However from the conditional to the post condition we get essentially get two conditions:
  - From the assert condition in the if-branch follows the post condition:
    $a = 1 \rightarrow a > 0$.
  - From the assert condition in the else-branch follows the post condition:
    $a = 2 \rightarrow a > 0$.

Generated Ver. Cond.

From pre condition to if-case, condition 1 :

H1: a $<=$ 0 .
H2: a $>=$ integer__first .
H3: a $<=$ integer__last .
H4: a $>=$ 1 .
$->$
C1: -1 $>=$ integer__first .
C2: -1 $<=$ integer__last .

Generated Ver. Cond.

From pre condition to if-case, condition 2 :

H1: a $<=$ 0 .
H2: a $>=$ integer__first .
H3: a $<=$ integer__last .
H4: a $>=$ 1 .
H4: -1 $>=$ integer__first .
H4: -1 $<=$ integer__last .
$->$
C1: -1 = 1 .
C2: -1 $>=$ integer__first .
C3: -1 $<=$ integer__last .
C4: a $<=$ 0 .

Generated Ver. Cond.

From pre condition to else-case, condition 1 :

H1: a $<=$ 0 .
H2: a $>=$ integer__first .
H3: a $<=$ integer__last .
H4: not(a $>=$ 1) .
$\quad -\,>$
C1: 2 $>=$ integer__first .
C2: 2 $<=$ integer__last .

Generated Ver. Cond.

From pre condition to else-case, condition 2 :

H1: a <= 0 .
H2: a >= integer__first .
H3: a <= integer__last .
H4: not(a >= 1) .
H4: 2 >= integer__first .
H4: 2 <= integer__last .
    - >
C1: 2 = 2 .
C2: 2 >= integer__first .
C3: 2 <= integer__last .
C4: a <= 0 .

Generated Ver. Cond.

From conditional to post condition:

H1: a = 1 .
H2: a $>=$ integer__first .
H3: a $<=$ integer__last .
H4: a $\sim <=$ 0 .
    $->$
C1: a > 0 .

H1: a = 2 .
H2: a $>=$ integer__first .
H3: a $<=$ integer__last .
H4: a $\sim <=$ 0 .
    $->$
C1: a > 0 .

Assert between Conditionals

- Using assert directives between conditionals allows to avoid an expontential blow up if one has too many conditionals.
  - If we have n if then else statements in sequence without any assert conditions, then we have 2^n conditions, namely that from the pre condition and one selected branch of each conditional follows the post condition.
  - If we have assert statements between each conditional this is broken up, into two statements per conditionals, namely that from the previous assert condition and one branch follows the next assert statement.

Procedures with Loops

Verification Conditions for Procedures with Loops

- If one has a loop, a loop invariant is required.
  The syntax is for instance:
  --# **assert** X +Y= X~+Y~;
- If one has one pre-condition, one loop and one post-condition, the examiner generates verification conditions expressing:
  - From the pre-condition follows, when first entering the loop, the condition of **assert**.
  - From **assert** follows, if exit conditions are false, the condition of **assert** after one step.
  - From **assert** follows, if one exit condition is true, the **post-condition**.

Procedures with Loops

- The assert statement can occur anywhere in the loop (provided there is an assert for each possible path through the loop).

Example

```
procedure test(X,Y: in out Float)
--# derives X from X &
--#              Y from X,Y;
--# pre X>0.0;
--# post X+Y=X~+Y~ and X<0.0;
is
begin
  loop
    --# assert X +Y= X~+Y~;
    X := X - 1.0;
    Y:= Y + 1.0;
    exit when X < 0.0 ;
  end loop;
end test;
```

Generated Verification Conditions.

```
procedure test(X,Y: in out Float)
--# pre X>0.0;
--# post X+Y=X~+Y~ and X<0.0;
is begin loop
   --# assert X +Y= X~+Y~;
   X := X - 1.0; Y:= Y + 1.0;
   exit when X < 0.0 ;
end loop; end test;
```

- When entering the loop we have X= X~, Y = Y~.
- So that from the pre condition X~ >0.0 follows that at the beginning of the loop we have X+Y = X~ + Y~, reads:
- $X\sim\ > 0.0 \rightarrow X\sim+Y\sim = X\sim+Y\sim$.

Generated Verification Conditions

The examiner generates the following verification condition for this:

- H1: x> 0.
  H2: true.
  H3: true.
  ->
  C1: x + y = x + y.

Generated Verification Conditions.

```
procedure test(X,Y: in out Float)
--# pre X>0.0;
--# post X+Y=X~+Y~ and X<0.0;
is begin loop
   --# assert X +Y= X~+Y~;
   X := X - 1.0; Y:= Y + 1.0;
   exit when X < 0.0 ;
end loop; end test;
```

- After one iteration through the loop, X is decreased by 1.0 and Y increased by 1.0.
- The loop is left if X afterwards, (= X original -1.0), is $<$ 0.0.

Generated Verification Conditions.

```
procedure test(X,Y: in out Float)
--# pre X>0.0;
--# post X+Y=X~+Y~ and X<0.0;
is begin loop
   --# assert X +Y= X~+Y~;
   X := X - 1.0; Y:= Y + 1.0;
   exit when X < 0.0 ;
end loop; end test;
```

- So that the loop invariant is preserved means that
  - from the loop-invariant, before the iteration took place, i.e. from X+ Y = X~ + Y~
  - and the fact that the exit condition was false, i.e. not(X - 1.0 < 0.0)
  - it follows that the loop invariant in the next iteration is true, i.e. (X-1.0)+ (Y+1.0) = X~ + Y~.

Generated Verification Conditions.

```
procedure test(X,Y: in out Float)
--# pre X>0.0;
--# post X+Y=X~+Y~ and X<0.0;
is begin loop
   --# assert X +Y= X~+Y~;
   X := X - 1.0; Y:= Y + 1.0;
   exit when X < 0.0 ;
end loop; end test;
```

- So the condition is:
  - $X + Y = X\sim + Y\sim \wedge \neg(X - 1.0 < 0.0)$ implies $(X-1.0) + (Y+1.0) = X\sim + Y\sim$.

Generated Verification Conditions.

- The examiner generates (note that here $\sim$ are used, since it is unavoidable):
  - H1: x+y = x$\sim$ + y$\sim$
    H2: not (x-1 $<$ 0)
    $->$
    C1: x-1+(y+1) = x$\sim$ + y$\sim$.

Generated Verification Conditions.

```
procedure test(X,Y: in out Float)
--# pre X>0.0;
--# post X+Y=X~+Y~ and X<0.0;
is begin loop
   --# assert X +Y= X~+Y~;
   X := X - 1.0; Y:= Y + 1.0;
   exit when X < 0.0 ;
end loop; end test;
```

- When getting from the last iteration through the assert formula to the end we have that again X afterwards is X before -1.0 and Y afterwards is Y before + 1.0.
- So that the post condition is implied by the last assert formula means that $X + Y = X\sim + Y\sim$ and the condition for exiting $(X - 1.0 < 0.0)$ both imply $(X - 1.0) + (Y+1.0) = X\sim + Y\sim$ and $X -1.0 < 0.0$.

Generated Verification Conditions.

- The examiner generates:
  - H1: $x+y = x\sim + y\sim$
    H2: $x-1 < 0$
    $->$
    C1: $x-1+(y+1) = x\sim + y\sim$.
    C2: $x-1 < 0$

- In this example all three verification conditions are automatically shown by the simplifier.

Loop Without Assert

- If there is no assert statement, the examiner issues a warning and generates a default assert statement (in the above example the pre condition).

Verification Conditions and Procedure

- In a program with several procedures, SPARK-Ada uses
  - The input output behaviour of the variables of procedures called when determining the input output behaviour of procedures using it.
  - The dependencies of variables in the variables in procedures called when determining the dependencies of variables in procedures using them.
  - The pre- and post conditions in procedures called in order to generate the verification conditions for the procedures using them.
- In the following example, we don't separate the package into a specification and body (as it stands, SPARK Ada won't accept it).

Example

```
package body Test_Package
  --# own A : Integer;
is A : Integer;
  procedure Init
  --# global out A;
  --# derives A from ; post A = 0;
  is begin A:= 0;
  end Init;

procedure Procedure1;
  --# global in out A;
  --# derives A from A;
--# pre A = 0; post A=2;
  is begin A:= A+2;
  end Procedure1;
```

Example

```
  procedure Main;
  --# global out A;
  --# derives A from ;
  --# post A = 2;
  is begin
    Init;
    Procedure1;
  end Main;
end Test_Package;
```

- That in Main A is an out variable follows since in Init A is output variable, and in Procedure1 it is an input/output variable.
- That Main derives A from nothing follows, since Init derives A from nothing, and Procedure1 derives A from A.

Verification Conditions for Main

- Since Init has no pre-condition, we don't have to show any pre condition for Init in Main.
- We have to show however that from the pre condition of Main (true), and the post condition of Init (A=0) follows the pre-condition for Procedure1 (A=0):
  - $(\text{true} \wedge A = 0) \rightarrow A = 0$;

Verification Conditions for Main

- We have to show that from the pre condition of Main (true), the pre condition (A=0) and Post condition (A = 2) of Procedure1 follows the post condition of Main (A=2).
  - Since there are two versions of A involved, one which is as it is when Init is entered, and one, which is as it is when Procedure1 is entered, we have to work with these two versions of A called A_1 and A_2.
  - The verification condition is:
    $(\text{true} \wedge A_1 = 0 \wedge A_1 = 0 \wedge A_2 = 2) \rightarrow A_2 = 2$.

Generated Verification Conditions

H1: true .
H2: a__1 = 0 .
->
C1: a__1 = 0 .

H1: true .
H2: a__1 = 0 .
H3: a__1 = 0 .
H4: a__2 = 2 .
->
C1: a__2 = 2 .

Using Post Conditions of Functions

- Slides for this part have to be rewritten (functions didn't work well in the old edition but seem to work well in the 2003 edition of the book).

Other Annotations

- The main program, which is not part of a package has to be declared by:
  --#**main_program;**
- One can hide the body of a procedure.
  - Allows especially direct interaction with non-critical and therefore non-verified Ada programs.
  - Allows as well to integrate not yet implemented code.

Hide Directive

- Example Syntax:

```
procedure MyGUI
--# global out A; in B;
--# derives A from B;
is
   --# hide MyGUI
   -- Details of MyGUI will not be checked.
end MyGUI;
```

- Although the details will not be checked, the “global” and “derives” statements (and possibly pre and post conditions for the hidden procedure) will be used to verify other procedures using this procedure.
- In addition SPARK Ada will issue a warning to remind the user that this procedure is not checked.

Arrays

- Many practical examples involve arrays.
  - E.g. a lift controller might have as one data structure an array

    Door_Status: **array** (Floor_Index) **of** (Open,Closed);

    where
    - Floor_Index is an index set of floor numbers (e.g. 1 .. 10)
    - and Door_Status(I) determines whether the door in the building on floor I is open or closed,
      - In Ada one writes A(I) for the ith element of array A.

Arrays

Door_Status: array (Floor_Index) of Door_Status_Type;
type Door_Status_Type is (Open,Closed);

- Problem: correctness conditions involve quantification:
  - E.g. the condition that only the door at the current position (say Lift_Position) of the lift is open, is expressed by the formula:

$$\forall I : \text{Floor_Index}.(I \neq \text{Lift_Position} \\ \rightarrow \text{Door_Status}(I) = \text{Closed}) \enspace .$$

- This makes it almost impossible to reason automatically about such conditions.

Arrays

- In simple cases, we can solve such problems by using array formulae.
- Notation: If A is an array, then
  - C:= A[I $=>$ B] stands for the array, in which the value I is updated to B. Therefore
    - C(I) = B,
    - and for J $\neq$ I, C(J) = A(J).
  - Similarly D:= A[I $=>$ B, J $=>$ C] is the array, in which I is updated to B, J is updated to C:
    - D(I)= B,
    - D(J) = C,
    - D(K) = A(K) otherwise.

Example

- The correctness for a procedure which swaps elements A(I) and A(J) is expressed as follows:

```
type Index is range 1 .. 10;
type Atype is array(Index) of Integer;

procedure Swap_Elements(I,J: in Index;
                          A: in out Atype)
--# derives A from A,I,J;
--# post A = A~[I => A~(J); J => A~(I)];
is
    Temp: Integer;
begin
    Temp: = A(I); A(I) := A(J); A(J) := Temp;
end Swap_Elements;
```

Example

- A = A~[I => A~(J); J => A~(I)];
  Expresses that
  - A(I) is the previous value of A(J),
  - A(J) is the previous value of A(I),
  - A(K) is unchanged otherwise.
- In the above example, as for many simple examples, the correctness can be shown automatically by the simplifier.

More Complicated Example

- In the lift controller example, one can express the fact that, w.r.t. to array Floor, exactly the door at Lift_Position is open, as follows:

```
--# post Closed_Lift = Door_Type'(Index=> Closed)
--#      and
--#      Floor = Closed_Lift[Lift_Position => Open];
```

- Here
  Floor_Type'(Index=> Closed)
  is the Ada notation for the array A of type Floor_Type, in which for all I:Index we have A(I) = Closed.

More Complicated Example

- Unfortunately, with this version, the current version of SPARK Ada doesn’t succeed in automatically proving the correctness even of a simple function which moves the lift from one floor to the other (and opens and closes the doors appropriately).

Quantification

- It is usually too complicated to express properties by using array formulae, and one has to use quantifiers instead.
- As soon as one uses quantifiers, the simplifier usually fails, and one has to prove the statements by hand or using the interactive theorem prover.

Reduction of Quantification

- If the quantifiers range over finite sets (as they do when we quantify over index sets of arrays), one could avoid quantifiers and replace formulae by propositional (i.e. quantifier free) formulae. Examples:
  - One can replace the formula
    $\forall$ I: Floor_Index.φ(I)
    (where φ(I) is some property), assuming that Floor_Index is equal to $0 \cdots 10$, by the following one:
    $\varphi(0) \wedge \varphi(1) \wedge \cdots \wedge \varphi(10)$.
  - One can replace the formula
    $\exists$ I: Floor_Index.φ(I)
    (where φ(I) is some property), assuming that Floor_Index is equal to $0 \cdots 10$, by the following one:
    $\varphi(0) \vee \varphi(1) \vee \cdots \vee \varphi(10)$.

Explosion of Formula Size

- This way one can get rid of all quantifiers.
- If the resulting propositional formulae are true and not too big, one can search automatically for a solution.
  - General automatic theorem proving techniques (often using Prolog) can be applied here.

Explosion of Formula Size

- In principal it is undecidable, whether such a formula holds, since the formulae usually involve reference to arbitrary big integers
  - The undecidability holds only if one assumes that integers can be arbitrarily big.
  - If one takes into account that there are only finitely many integers, one obtains that the the validity of propositional formulae referring only to integers is decidable.
    But this is only a theoretical result, since it is infeasible to try out all possible integer values.
- But still in many cases it is feasible to search for solutions, if the formulae involved are not too big.

- The problem is that the size of the formulae explodes.

Explosion of Formula Size

- Assume My_Index = 1 .. N for some fixed number N.
  - Then $\exists$ I: My_Index.φ(I) is the disjunction of N formulae.
  - $\forall$ I: My_Index.$\exists$ J: My_Index. φ(I,J)
    is the conjunction of N formulae, each of which is the disjunction of M formulae, so we get a length of N $*$ N.
- This approach brings easily the simplifier to its limits.
  - Probably with some more sophisticated tools one could solve much more problems automatically.

Explosion of Formula Size

- The size of the formulae explodes much more if one transforms the formulae into some normal form.
  - One can transform propositional formulae into conjunctive normal form.
    - But then for instance $(A \wedge B \wedge C) \vee (D \wedge E \wedge F)$ is translated into

$$
\begin{array}{l}
(A \vee D) \wedge (A \vee E) \wedge (A \vee E) \wedge (A \vee F) \\
\wedge (B \vee D) \wedge (B \vee E) \wedge (B \vee E) \wedge (B \vee F) \\
\wedge (C \vee D) \wedge (C \vee E) \wedge (C \vee E) \wedge (C \vee F)
\end{array}
$$

    - In general one gets an exponential blow up.

Explosion of Formula Size

- Similarly one can transform formulae into a conjunction of Horn formulae, i.e. formulae of the form $A_1 \to \cdots A_n \to B$, where A_i, B are possibly negated atomic formulae.
  - These are the formulae, the simplifier can deal with.
  - One obtains however the same exponential blow up.
- Inherent reason: the number of possible states explodes exponentially.
  - For instance, if one has N state variables with 2 states, the overall system has 2^N possible states.
- A correctness proof needs to show that the formulae hold for each of these 2^N states.

Proof Checker

- Therefore in many cases one needs to carry out some proofs using **interactive theorem proving.**
  - In SPARK Ada done using the "proof checker" (not part of the CD distributed with the book by Barnes [Ba03]).
- **Problem:** Code often changes, but since the theorems to be proved are automatically generated, proofs cannot directly be transported from one version to another version of the program.
  - Need for theories, in which proving and programming are more closely connected.

(g) Example: Railway Interlocking Sys

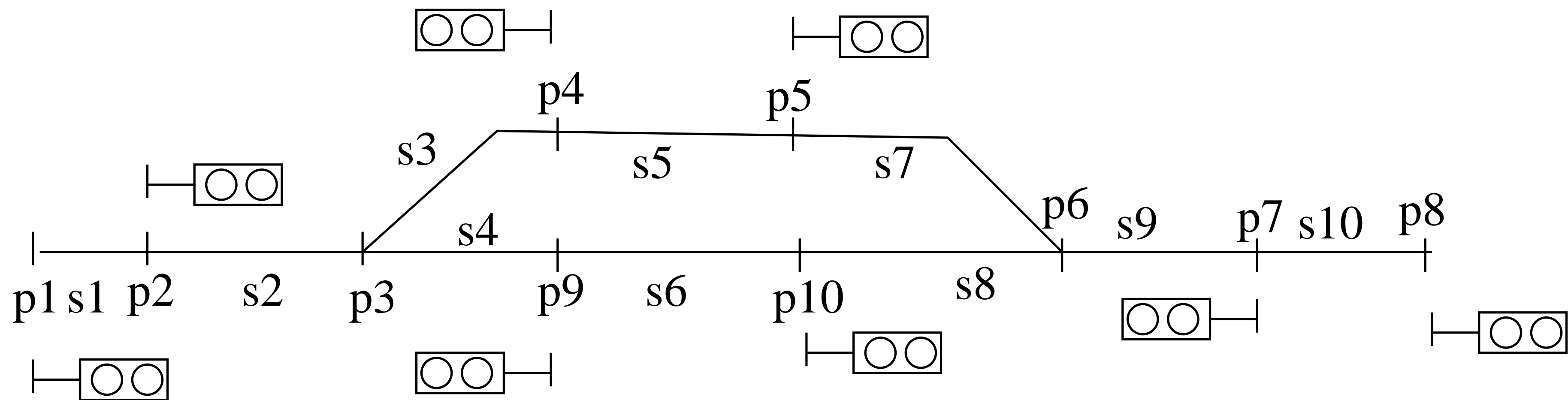
- We will look at the following at an example of verify a small **railway interlocking system** using Ada.
  - We won't achieve full verification.
  - The verification of formulae, which cannot be derived by the simplifier, will be done by hand.
- We will look first at a relative general description of railway systems, and then look more concretely at a simplified example.

Description of Rail Yards

- The basic unit into which one divides a rail yard is that of a **track segment**.
- A track segment is stretch of a track without any further smaller parts, which are significant for an analysis of a interlocking system.
  - there are no branches in between,
  - there are no crossings in between,
  - they are not divided by signals into parts.

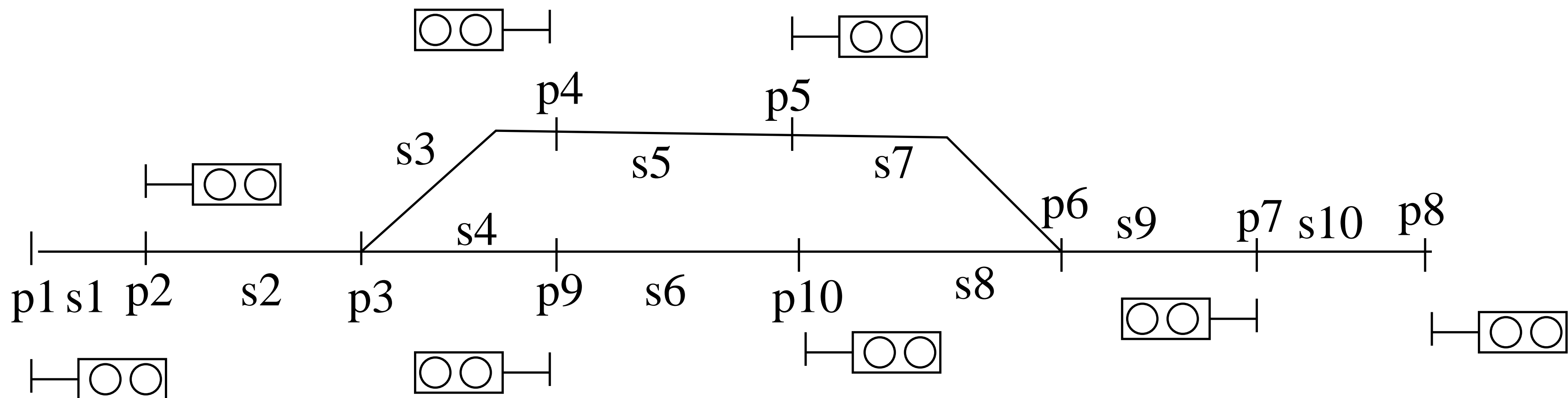
Example

- In the following example we have track segments s1 - s10.



Points

- In the example we see that each track segment is delimited by two sets of points, one on each side.
- We have here sets of points p1 - p10, and s3 is delimited by p3 and p4.

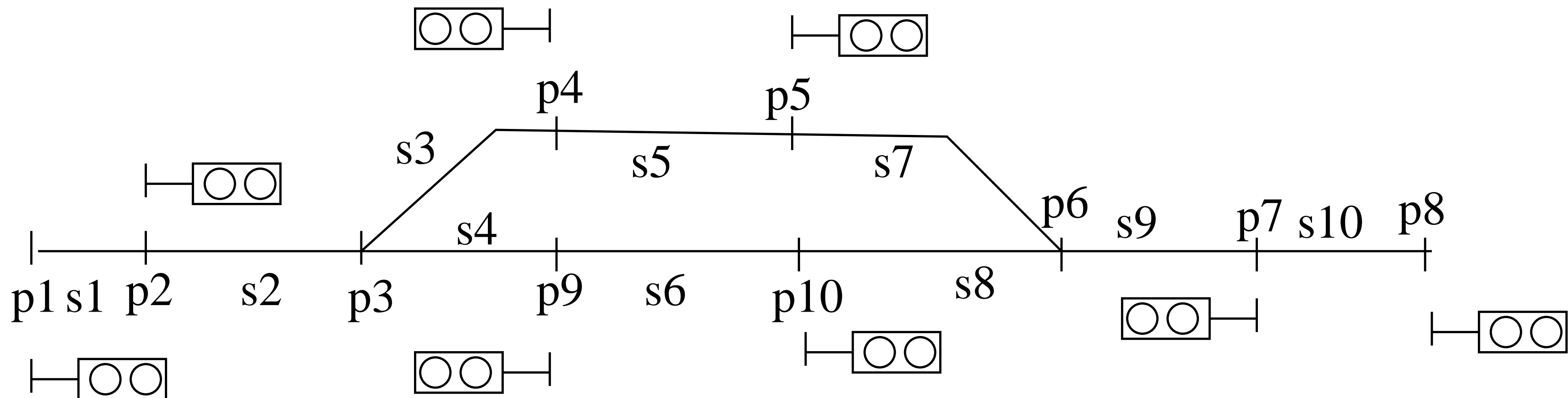


Connections

- At each position, 1, 2 or 3 track segments are connected.
  - Connection of 1 track segment means that we have the end of a track or the end of what is guarded by the interlocking system in question.
    - In the example, these are p1 and p8.
  - Connection of 2 track segments means that we have a line, split into two usually by a signal.
    - In the example, these are p2, p4, p5, p7, p9, p10.
  - Connection of 3 track segments means that we have points in between.
    - In the example, these are p3 and p6.

Connections

- At a connection with 3 track segments connecting at one position, it is only for some of these segments possible to move between those.
  - In the example one can move at p3 from s2 to s3, from s2 to s4, but not from s3 to s4.

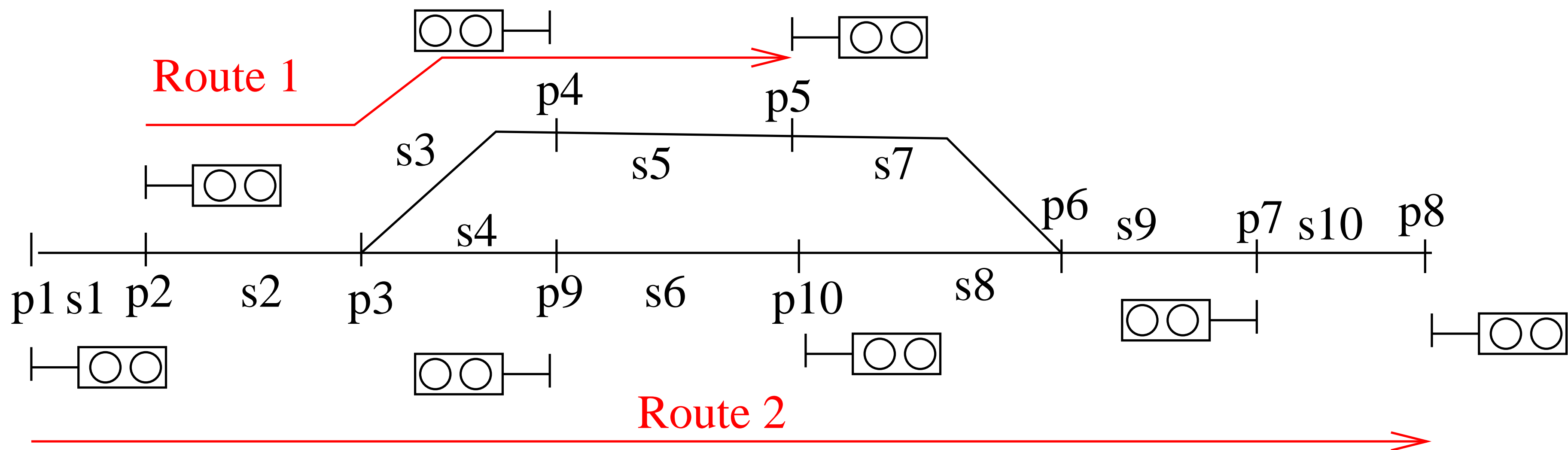


Train Routes

- The control system for such a rail yard has several **train routes**.
  - A **train route** is a sequence of track segments, the train can follow without ever having to stop in between (except in emergency cases).

Train Routes

- In the example, (s2, s3,s5) and (s2,s4) form train routes, but one might have a train route (s1,s2,s4,s6,s8,s9,s10) for a fast train, which passes through this station without stopping.
- Entry to a route is protected by a signal, and the end of a route is protected by a signal.



Signals

- At some positions there are signals, which give access at one position from one segment into another.
- One usually has advance and main signals.
  - Distant signals are needed, since the braking distance of trains is too long for trains to be able to stop when they see the main signal.
- We will look in the following only at main signals, and ignore the speed of the trains.
- Then we have that at the end of each route there must be a main signal.

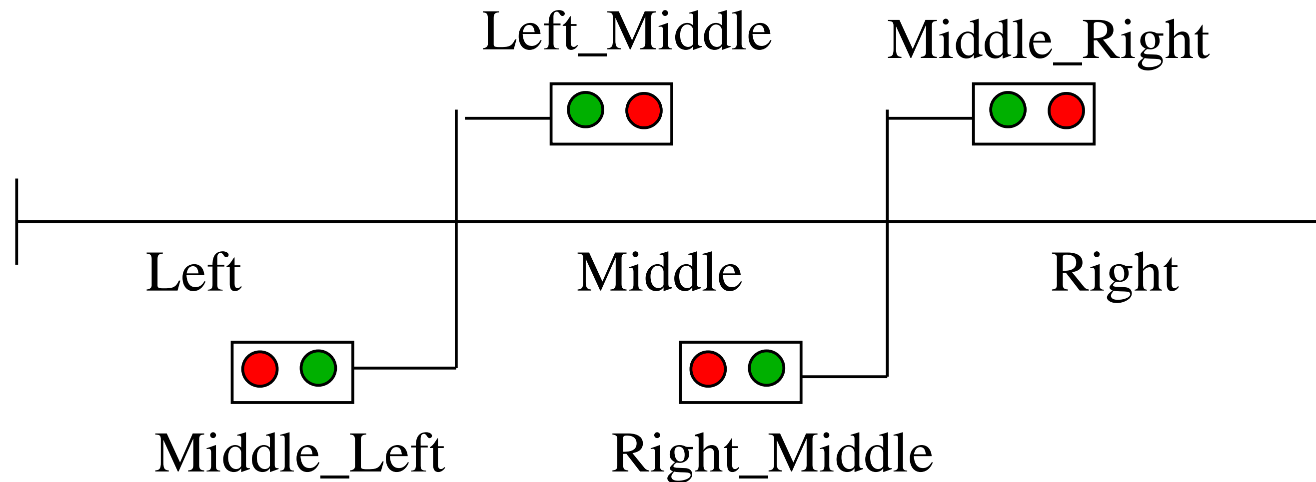
Locking of Segments

- If a train route is chosen for a train, then all segments, the train route consists of, are locked.
- For another train, a train route can only be chosen if it doesn’t overlap with the train route of the first train.
- So a train route can be only chosen, if all segments of the train route are currently unlocked.
- A signal can only be green, if it is leading into or is part of a train route, which has been selected.

Case Study

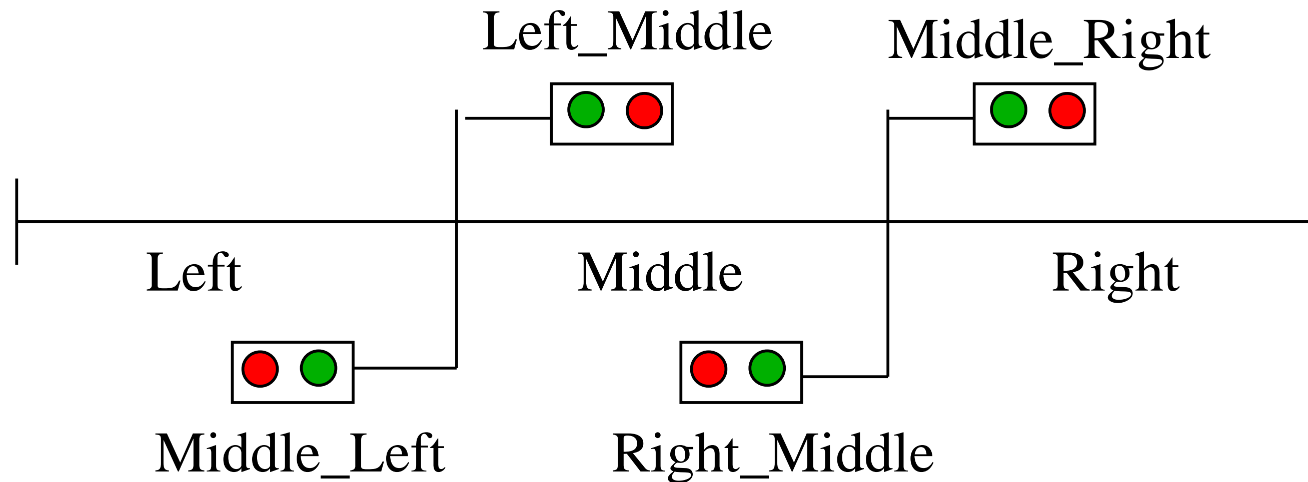
- We will now carry out a very simple case study.
- Available from http://www.cs.swan.ac.uk/∼csetzer/lectures/critsys/03/SPARK_Ada/railwayExample/

Case Study



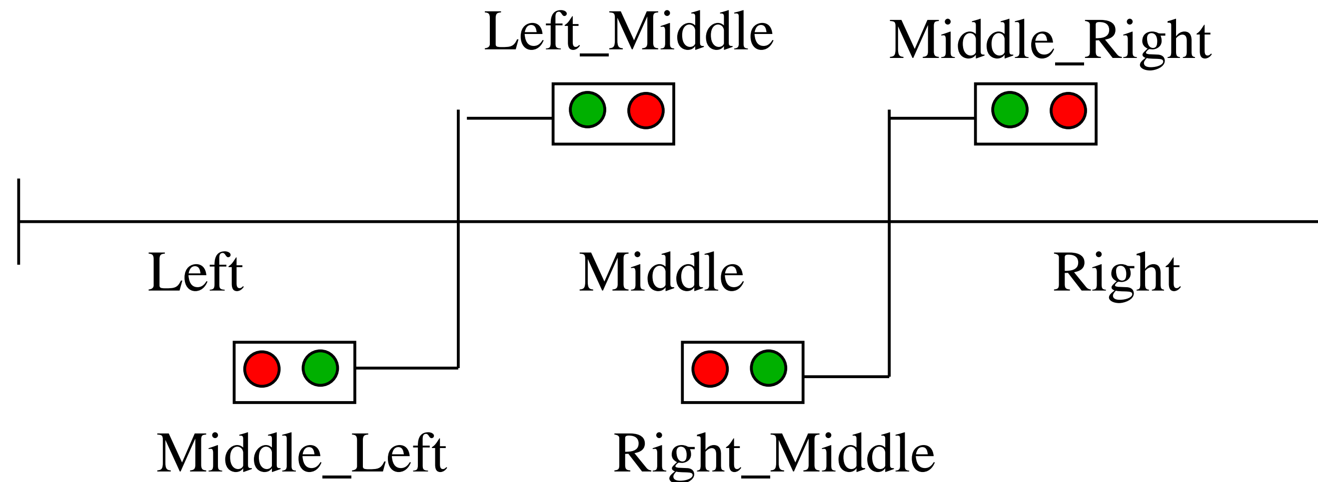
- We will investigate a very simple example of a rail yard.
  - We have 3 segments, “Left”, “Middle”, “Right”.
  - We have 4 signals, “Left_Middle”, “Middle_Left”, “Middle_Right”,“Right_Middle”.
    - E.g. “Right_Middle” guards access from Segment “Right” to segment “Middle”.

Case Study



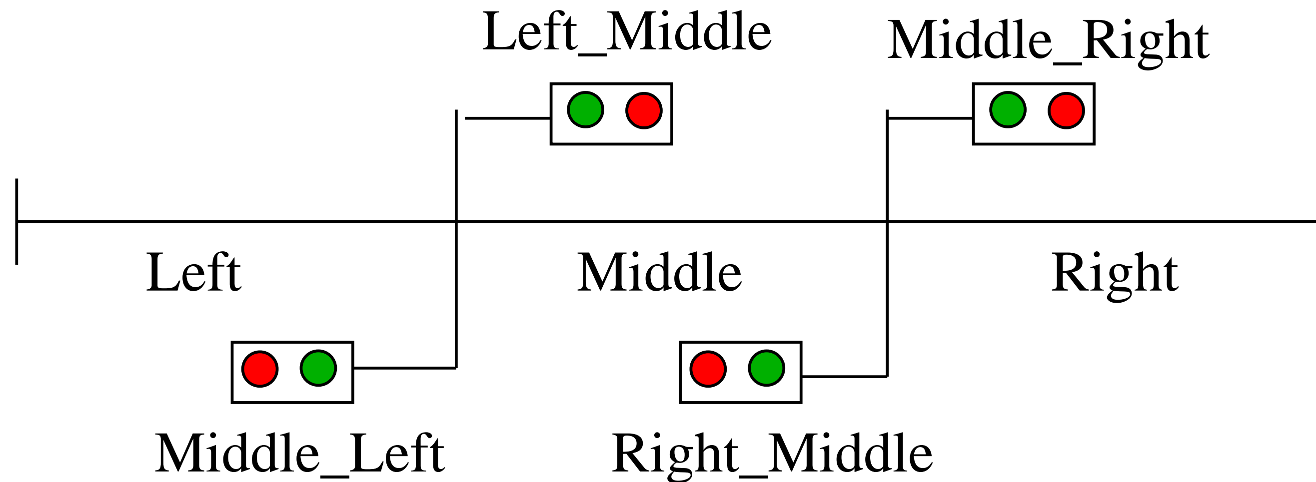
- There are 8 routes:
  - 4 routes leading from one segment to an adjoining one:
    "Route_Middle_Left", "Route_Middle_Right", "Route_Right_Middle", "Route_Left_Middle".
    - E.g. "Route_Middle_Left" is the route for a train moving from segment "Middle" to segment "Left".

Case Study



- 2 routes which allow to give access from the neighbouring rail yards to segments “Left” and “Right”:
  “Route_Enter_Left”, “Route_Enter_Right”.

Case Study

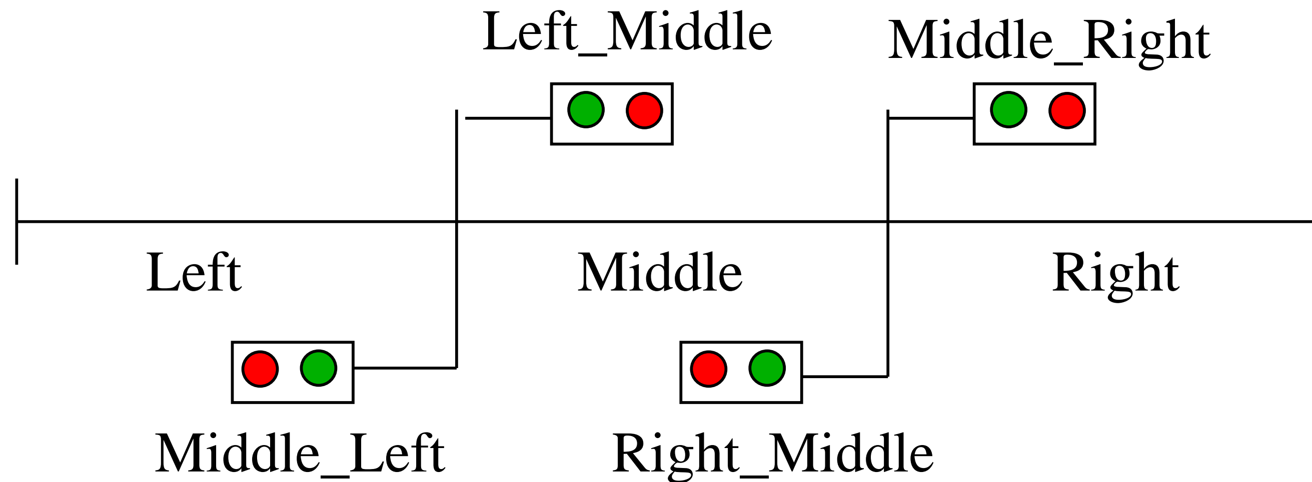


- 2 routes, which allow a train to leave from segments Left and Right to the neighbouring rail yards.
  - "Route_Leave_Left", "Route_Leave_Right".
- Note that in this situation, leaving and entering the yard from the outside is not guarded by signals – it would be easy to take this into account as well.

Case Study

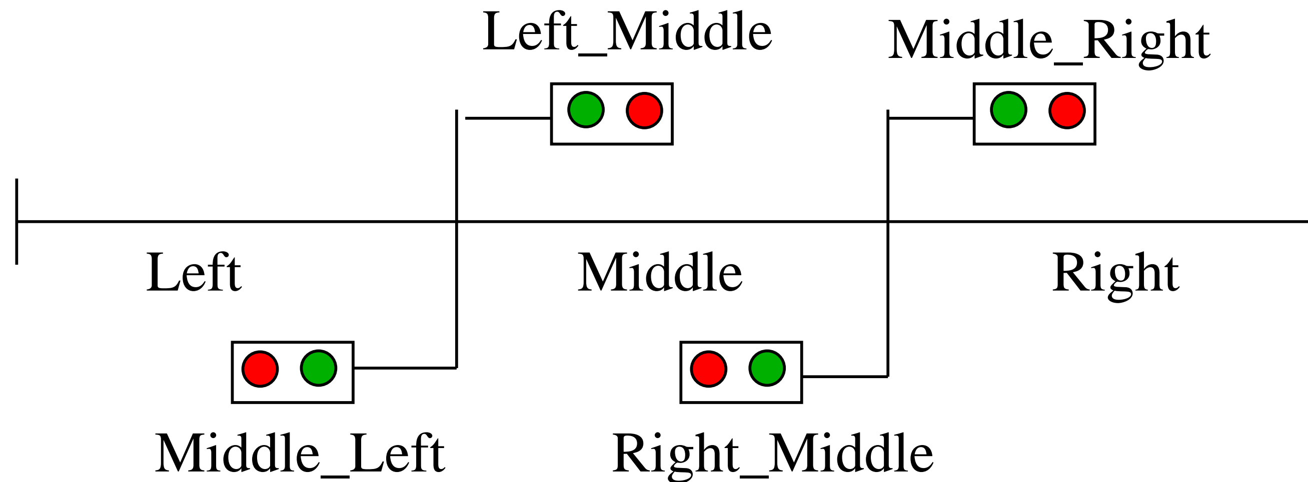
- Each segment will have 6 possible states:
  - “Occupied_Standing”, for a train is in this segment, but not about to leave it.
  - “Occupied_Moving_Left”, “Occupied_Moving_Right” for a train is in this segment and moving to the next segment left or right.
  - “Reserved_Moving_From_Left”, “Reserved_Moving_From_Right” for the segment is reserved for a train coming from the next segment to the left or to the right.
  - “Free”, for there is no train currently in this segment or moving into this segment.

State of Signals



- Each Signal has two states: “Red” and “Green”.
- The state of all signals together is given by a tuple with 4 components, which give the signal state for each of signal:
  “Left_Middle”, “Middle_Left”, “Middle_Right”, “Right_Middle”.

State of the Segments



- The states of all segments together is given by a tuple with 3 components, which give the state of each segment:
  "Left", "Middle", "Right".

Modes of Operation

- There are two modes of operation:
  - “Mode_Open_Route”.
    - In this mode we open a route for one train, which is currently standing in one segment, to the next segment.
  - “Mode_Move_Train”.
    - In this mode we assume that one train has moved from one segment to the next one, and adapt the states of the segments correspondingly.

Data Structure in Ada

```
with Spark_IO;
--# inherit Spark_IO;
package Simple_Railway
--# own Segment_State    : Segment_State_Type;
--#     Signal_State     : Signal_State_Type;
is

   type One_Signal_State is (Red,Green);

   type Mode is (Mode_Open_Route, Mode_Move_Train);
```

Data Structure in Ada

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

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

Data Structure in Ada

```
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;
```

Data Structure in Ada

We have the following two variables, describing the overall state of the system:

```
Segment_State : Segment_State_Type;
Signal_State  : Signal_State_Type;
```

Procedures

- We have one procedure„ which opens a route.
  It has one additional output parameter “Success”, which returns true, if the route could be opened, and false, otherwise:

```
procedure Open_Route(Route   : in Route_Type;
                     Success: out Boolean);
--# global in out Segment_State, Signal_State;
--# derives Segment_State,Success
--#            from Segment_State, Route &
--#         Signal_State
--#            from Segment_State, Route, Signal_State;
```

Procedures

- We have one procedure which moves a train along one route.
  Again we have an output parameter "Success":

```
procedure Move_Train(Route     : in Route_Type;
                    Success : out Boolean);
--# global in out Segment_State, Signal_State;
--# derives Segment_State,Success
--#              from Segment_State, Route &
--#          Signal_State
--#              from Segment_State, Route, Signal_State;
```

Procedures

- Further we have a procedure which prints the state

```
procedure Print_State;
--# global in out Spark_IO.Outputs;
--#            in Segment_State, Signal_State;
--# derives Spark_IO.Outputs
--#   from Spark_IO.Outputs, Segment_State, Signal_State;
```

- We are using here a package Spark_IO, which deals with file I/O and standard I/O (via console).
- Spark_IO has internal state variables Spark_IO.Outputs and Spark_IO.Inputs, which are essentially streams consisting of all outputs and inputs, respectively, which have not been processed yet.

Procedures

```
procedure Print_State;
--# global in out Spark_IO.Outputs;
--#            in Segment_State, Signal_State;
--# derives Spark_IO.Outputs
--#   from Spark_IO.Outputs, Segment_State, Signal_State;
```

- If we output something, we take the stream of outputs Spark_IO.Outputs, and append to it what we output.
- Above, the output depends on Segment_State and Signal_State, and therefore Spark_IO.Outputs depends on itself (we are appending something to it), and Segment_State, Signal_State.

Procedures

- The procedure Get_Action will determine the mode of the next action (from console input) and the route to which it is applied:

```
procedure Get_Action(Route      : out Route_Type;
                     The_Mode   : out Mode);
--# global in out Spark_IO.Inputs, Spark_IO.Outputs;
--# derives Spark_IO.Outputs
--#              from Spark_IO.Outputs, Spark_IO.Inputs &
--#           Spark_IO.Inputs, Route, The_Mode
--#              from Spark_IO.Inputs ;
```

- Note that the fact that the last two functions don't change Segment_State and Signal_State (they are only output variables), means that they don't perform any safety critical action.

Dependencies in Get_Action

```
--# derives Spark_IO.Outputs
--#             from Spark_IO.Outputs, Spark_IO.Inputs &
--#         Spark_IO.Inputs, Route, The_Mode
--#             from Spark_IO.Inputs ;
```

- Spark_IO.Outputs depends on Spark_IO.inputs, since, depending on the input, different informations are output.
- Spark_IO.Inputs depends on Spark_IO.Inputs, since, if we ask for input, the next data from the input stream is cut off and taken as input. Therefore the input stream is changed.

Implementation of Open_Route

- “Open_Route” checks for instance in case the route is “Route_Left_Middle”
  - whether segment “Left” is in state “Occupied_Standing”
  - and segment “Middle” is in state “Free”.
- If yes it will
  - set the state of segment “Left” to “Occupied_Moving_Right”,
  - set the state of segment “Middle” to “Reserved_Moving_From_Left”,
  - set signal “Left_Middle” to “Green”.

Implementation of Open_Route

```
procedure Open(Route   : in Route_Type;
                 Success out Boolean) is
begin
   Success := False;
   if Route = Route_Left_Middle then
      if (Segment_State.Left = Occupied_Standing
         and Segment_State.Middle = Free)
        then
           Segment_State.Left:= Occupied_Moving_Right;
           Segment_State.Middle:= Reserved_Moving_From_Left;
           Signal_State.Left_Middle:= Green;
           Success := True;
      else
           Success := False;
      end if;
   elsif . . .
```

Implementation of Move_Train

- “Move_Train” checks for instance in case the route is “Route_Left_Middle”
  - whether segment “Left” is in state “Occupied_Moving_Right”
  - and segment “Middle” is in state “Reserved_Moving_From_Left”.
- If yes it will
  - set state of segment “Left” to “Free”,
  - set state of segment “Middle” to “Occupied_Standing”,
  - set signal “Left_Middle” to “Red”.

Implementation of Move

```
procedure Move(Route    : in Route_Type;
               Success: out Boolean) is
  begin
  Success := False;
  if Route = Route_Left_Middle then
     if (Segment_State.Left = Occupied_Moving_Right
        and
        Segment_State.Middle = Reserved_Moving_From_Left) then
         Signal_State.Left_Middle:= Red;
         Segment_State.Left:= Free;
         Segment_State.Middle:= Occupied_Standing;
         Success := True;
     else
         Success := False;
     end if;
  elsif . . .
```

Limitations of SPARK Ada

When introducing correctness conditions the following problems were encountered:

- Although an industrial product used in the verification of safety critical systems, in many cases SPARK-Ada generated, when using arrays, formulae which were unprovable, although they were correct.
  - The logic behind array formulae isn’t as well thought through as it could be (at least in the version 1997, we are using).

Limitations of SPARK Ada

- Instead we decided to use, as in the above data structure, single variables for each state.
  - Therefore the resulting formulae become relatively long, since they are conjunctions over all segments or over all signals.
    - In the current toy example not a problem, since it is very small, but this would soon become infeasible for bigger examples.
    - One problem is that this solution doesn't scale (for instance our example doesn't extend easily from 3 segments to 6 segments).
- Because we are using a demo version (a license for the full version would probably cost several thousand pounds), SPARK-Ada is not even able to parse longer formulae.

Limitations of SPARK Ada

- This is a problem which most tools involving machine-assisted theorem show:
  - Even small problems use a lot of memory, and verification can take very long (even months).

Correctness Conditions

We have the following correctness conditions for the train controller:

1. If a signal is green, then
   - the segment it is leaving should be in state Occupied_Moving_{Left,Right}.
   - the segment it is leading to should be in state Reserved_Moving_From_{Right,Left}.

   Expressed for Signal Middle_Left as follows:

```
--#  (Signal_State.Middle_Left = Green
--#    -> (Segment_State.Left =
--#                 Reserved_Moving_From_Right
--#          and Segment_State.Middle =
--#                 Occupied_Moving_Left))
```

   Corresponding formulae are needed for each signal.

Correctness Conditions

- The previous formulae form **pre- and post-conditions** for both procedures Move and Open.

Correctness Conditions

2. In procedure “Open_Route”, which opens a route, we need that no train gets lost:
   - If a train is in one segment
     - i.e. the segment has state Occupied_Moving_Left, Occupied_Moving_Right, Occupied_Standing,

   it should have such a state afterwards as well.

Correctness Conditions

- In case of the left segment, this is expressed as follows:

```
--#     ((Segment_State~.Left = Occupied_Moving_Left
--#       or
--#       Segment_State~.Left = Occupied_Moving_Right
--#       or
--#       Segment_State~.Left = Occupied_Standing)
--#       ->
--#      (Segment_State.Left = Occupied_Moving_Left
--#       or
--#       Segment_State.Left = Occupied_Moving_Right
--#       or
          Segment_State.Left = Occupied_Standing))
```

- We need corresponding formulae for Middle and Left as well.

Correctness Conditions

3. In procedure “Move_Train”, which moves a train from one segment to another, if a segment was occupied, we have
   - either afterwards the segment is again occupied,
   - or we have chosen a corresponding route, the segment is now free, and the new segment contains the train.

Correctness Conditions

- Expressed in case of Segment Left as follows:

```
--#  (Segment_State~.Left = Occupied_Moving_Right
--#   ->
--#   (Segment_State.Left = Occupied_Moving_Right
--#    or
--#    (Route = Route_Left_Middle
--#     and Segment_State~.Middle =
--#                  Reserved_Moving_From_Right
--#     and Segment_State.Left = Free
--#     and Segment_State.Middle = Occupied_Standing)))
--#  and
--#   (Segment_State~.Left = Occupied_Standing
--#    ->
--#    Segment_State.Left = Occupied_Standing)
```

Correctness Conditions

```
--#and
--#(Segment_State~.Left = Occupied_Moving_Left
--# ->
--# (Segment_State.Left = Occupied_Moving_Left
--#  or
--#  (Route = Route_Leave_Left
--#   and Segment_State.Left = Free)))
```

- Corresponding formulae for the other two segments.

Alternative Approach

- Alternatively, one could have introduced a data structure determining the current position of trains. Then one could instead of the last two conditions formulate the condition:
  - the segment at the position of each train must have an “occupied”-state
- Problem, since the number of trains varies over time. But that problem seems solvable.

Result of Verification

- Because of the limitations of the demo-version, only the program with condition 1., and of it only of the following cut down version could even by parsed by the system:

```
--#  pre (Signal_State.Middle_Left = Green
--#->
--#    Segment_State.Middle =
--#           Occupied_Moving_Left;
```

- The simplifier can only prove half of the verification conditions, although there are only propositional formulae (no quantifiers) involved.
- This seems not to be due to the fact that a demo version is used, but due to the general limitations of the simplifier.

Result of Verification

- In case of the first verification condition for Open_Route, the formula to be proved is of the form $(H1 \wedge \cdots \wedge H29) \rightarrow C1$,
  where

  C1 = not (**fld_right_middle(signal_status) = green**) .
  H4 = **fld_right_middle(signal_status) = green**
  -> *fld_middle(segment_status)*
  = reserved_moving_from_right .
  H29= *fld_middle(segment_status) = free* .
- Assuming free $\neq$ reserved_moving_from_right we can easily conclude C1 from H4 and H29.
- But the simplifier doesn't seem to be able to find such inferences.

Result of Verification

- It seems that the simplifier
  - Can use term rewriting in order to
    - make standard transformation of functions (e.g. $a * 2 = a + a$)
    - some operations on array formulae,
    - transformation of functions defined by the user.
  - The simplifier can delete unnecessary hypotheses, and determine, whether the conclusion is contained in one of the hypotheses.
- But the simplifier cannot carry out more complex logical reasoning, even if affects only propositional formulae.

Evaluation

- The restrictions imposed by SPARK-Ada on the Ada language are very sensible, and help avoid errors.
- The flow control seems to help find many errors.
- The possibility of stating pre- and post-conditions is important for the process of verifying the code later (even if this is, as usual, done by hand).

Evaluation

- However, it seems that in most cases, the simplifier doesn't automatically prove the verification conditions, and therefore one has to use the interactive theorem prover provided.
  - Problem is that if one changes the code later, most of the work done using the interactive theorem prover is lost.
  - Need to integrate programming and proving in a better way, so that while changing the program the proofs are adapted.
- In real world applications, proving of the verification conditions is rarely carried out, or, if it is done, it is usually done by hand.

Testing of Proof Conditions

- Because of the problems of proving the correctness of verification conditions, often verification conditions are checked by automatic testing.
  - One chooses large amounts of test cases automatically, and checks whether the verification conditions always hold.
  - In our example one could test the verification conditions by choosing arbitrary values for the signals and the segment states, and checking, whether the generated formulae are always true.
  - Testing can be very successful – often errors have been found this way.

Role of Formal Methods

- The use of formal methods with full verification in real world applications is yet limited since it is too difficult to verify even simple examples using the tools available.
- This is
  - due to unsolved theoretical problems,
  - and the fact that the market for such tools is very limited, and therefore the systems available are not yet very far developed.

Validation vs. Verification

- We saw as well that it is easy to make mistakes in defining the correctness formulae.
  - In a first implementation, we forgot to check that trains don't get lost, and they got actually lost, so the program was not correct, despite being verified.
- **Verification** is the process of verifying that a software project meets its specification.
- **Validation** is the process of confirming that the specification is appropriate and consistent with the customer requirements.

Validation vs. Verification

- Whereas verification can be guaranteed by using formal derivation, validation is more complex and outside the control of formal methods.
  - However, by formalising specifications using formal specification languages, one can make the specifications so precise that hopefully errors are found.