Anton Setzer
Dept. of Computer Science, Swansea University

http://www.cs.swan.ac.uk/~csetzer/lectures/critsys/14/index.html

November 15, 2014
Remark

Sections 2 (a) - (g) will be highly based on John Barnes: High Integrity Software. The SPARK approach. [Ba03].
2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Conditions (Loops, Procedures and Functions)

2 (h) Example: A railway interlocking system
2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Conditions (Loops, Procedures and Functions)

2 (h) Example: A railway interlocking system
Ada Lovelace (1815 – 1852)
Ada Lovelace (1815 – 1852)

- Ada Lovelace wrote programs for Charles’ Babbage analytical machine.
  - The first general purpose computer (mechanical; Turing complete).
- Therefore often regarded as the first computer programmer.
In this subsection we introduce the basic features from Ada used in our introduction into SPARK Ada.

This is not a thorough introduction into Ada.

It is not necessary to have a complete understanding of Ada for this module. You only need to be able to understand the SPARK Ada code used.

There is no need to write full Ada programs.
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.
  - 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.
- First release: Ada 83 (1983 – same year C++ was released).
- Ada 2012: Revision of Ada, including many features of SPARK Ada into the core language.
  - Allows therefore Design by Contract.
Books About Ada 2012

Springer Lecture Notes in Computer Science, 2013.
Available on Campus via Springer Link using University Subscription.

(Contains a short chapter 27.6 on SPARK Ada).
Books About Older Versions of Ada and SPARKAda


- Main documentation of SPARK 2014 is
  - online documentation
  - Book [Ba03] (however SPARK Ada syntax need to be adapted, as made clear in the notes for this module).
Ada can be seen as the culmination of the development of imperative programming languages.

It was developed at a time when object oriented languages started to be used widely.

Syntax of object orientation added to Ada was not as easy to use as in other object-oriented languages.
Reasons for Using Ada in Critical Systems

- 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**.
- 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).
Ada is an imperative language.

- There are global variables.
- Instead of methods we have functions and procedures.
Ada is **not** case sensitive.

- So $x$ and $X$ are the same,
- `HelloWorld`, `Helloworld` and `helloworld` are the same.

SPARKAda will replace in generated code all variables by their lower case versions.
Compilation

One can compile Ada using the gcc-ada compiler (available under Linux). Compilation using

    gnatmake file

This will automatically compile the file and all packages referred to in the file.
Division into Specification and Implementation

All units in Ada (programs, functions, packages) have two parts

▶ The specification.
   ▶ It declares the interface visible to the outside.
   ▶ E.g. for a function the specification declares its arguments and return type:

   ```ada
   function Myfunction (A : Integer) return Integer;
   ```

▶ The implementation.
   ▶ E.g. for a function it would be

   ```ada
   function Myfunction (A : Integer) return Integer is
   begin
     return A ** 3;
   end Myfunction;
   ```

▶ Note that that the implementation repeats the specification.
Division into Specification and Implementation

- Usually specification and implementation are separated into two files:
  - `file.ads` contains the specification.
  - `file.adb` contains the implementation.
Basic Syntax

- **Typing** is written as in Haskell
  - X : A stands for “X has type A”.
    - In Java or C this is written as “A X”.
- `--` introduces a comment.
We have enumeration types. E.g.

```ada
type Genders is (Male, Female);
```

introduces a new type having elements Male and Female.
Ranges in Ada

- Ada allows to define ranges.

```ada
subtype Year is Integer range 0..2100;
```

Myyear : Year := 2010;
results in a compiler error.

- If an out of range error is detected at compile time, the compiler will report an error.
- If it isn’t detected at compile time we get a run time error:

```
Get(Myyear);
If you enter at runtime 2101 we get:
```

```
raised CONSTRAINT_ERROR : example5.adb:15 range check failed
```
Ranges in Ada

- We have as well subranges of enumerations:

  ```ada
  type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

  subtype Weekday is Day range Mon .. Fri;
  ```
Arrays

Arrays are declared and used as follows:

```ada
type myArrayRange is new Integer range 0 .. 100;
type Day is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type myArrayType is array (myArrayRange) of Day;

myArray : myArrayType;
myArray(17) := Sun;
```

One can have as well anonymous arrays (index type is not defined separately):

```ada
type Vector is array (0 .. 100) of Integer;
```
If and When

The syntax for if and nested ifs is as follows:

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

E.g.

```ada
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 and While Loops

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

- A while loop is written as follows:
  - while (X > 0.0) loop
    - Y := Y + 1.0;
    - exit when Y > 5.0;
    - X := X - 1.0;
  end loop;
Nested Loops can be Labelled

- One can label loops. This allows to exit not only to the currently innermost loop.

Example:

Outer\_Loop:

```ada
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
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.
      All parameters in Java are `in` parameters.
  - output parameters; symbol: `out`,
  - input/output parameters; symbol: `in out`.
    - Reference parameters.
      The same as a **var** parameter in Pascal
      In Java non-constant instance variables of object parameters behave essentially as `in out` parameters.

- 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
    ```ada
    procedure Myproc(B: out Integer)
    is begin
      B := 0;
    end Myproc;
    ```
  - It doesn’t make sense to make a call
    ```ada
    Myproc(0)
    (Compiler error)
    ```
    it only makes sense to call
    ```ada
    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:
    ```ada
    procedure Myproc(B: in Integer, C: out Integer)
    is begin
       C := B;
    end Myproc;
    ```
  - It makes sense to call
    ```ada
    Myproc(0,D)
    ```
    where D is a variable that can be changed.
  - It makes as well sense to call
    ```ada
    Myproc(3 + 5,D)
    ```
    or
    ```ada
    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

```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
```
Packages

- Programs are divided into packages.
- A package is a collection of types, global variables (with their types), functions and procedures.
- Packages are specified (in the .ads file) as follows:

```ada
package Mypackage is
  -- Specification of the types, functions and procedures
  -- of this package
end Mypackage;
```
The implementation (in the .adb file) of packages is given as follows:

```ada
package body Mypackage is
  -- Implementation of all types functions and procedures
  -- of this package
begin
  -- Initialisation part of the packages
  -- Initialises global variables
end Mypackage;
```
Initialisation of Variables

Variables in a package can be initialised

- In their declaration e.g.

  Sum: Integer := 0;

  declares a new variable Sum of type Integer which is initialised to 0.
Initialisation of Variables

prosperrm Variables in a package can be initialised

- Or in the initialisation part of the procedure, e.g.

  ```ada
  package body MyPackage is
  X : Integer range 0 .. 1000

  -- Declaration of procedures

  begin
    X := 0;
  end MyPackage
  ```
Using other Packages

If one wants to refer to another package, one has to write before the procedure/package:

\[
\text{with nameOfPackageToBeUsed;}
\]

Then no identifiers are directly visible, they have to be used in the form

\[
\text{nameOfPackageToBeUsed.X}
\]

If one adds in addition

\[
\text{use nameOfPackageToBeUsed;}
\]

then the identifier \(X\) can be used directly.
Record Types

- A record is essentially a class without methods.
- Syntax in Ada:
  ```ada
  type Person is
    record
      Yearofbirth : Integer
      Gender : Genders
    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 : Genders.
- 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 := (Yearofbirth => 1975,Gender => Male)
- There are as well Variant Records, where the type of other fields depends on one field. See additional material.
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

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

function "+" (X, Y : Integer) return Integer  
    -- + can be used infix

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.  
```
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.
2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Conditions (Loops, Procedures and Functions)

2 (h) Example: A railway interlocking system
Change of Slides

- Since distribution of the slides last Friday I have gotten an improved understanding of the revised architecture of SPARK Ada.
- Please use this version of the slides and discard of Sect (b) distributed before. (Sect. (a) remains unchanged).

- SPARK Ada is changing frequently. Especially new restrictions (which almost always make sense) are introduced and other restrictions are removed, presumably in response to requests by their users.
  - Although doing my best to detect those changes, I cannot guarantee that I am able to follow all of them.
Overview and History 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 and recently to Ada 2012 (SPARK 2014).
  - Commercial tools available from Praxis Critical Systems.
Instructions of How to Install and Use SPARK Ada

▶ You will need to install GNAT Ada GPL 2014 and then SPARK GPL 2014.

▶ Instructions on how to install and use SPARK Ada can be found at http://www.cs.swan.ac.uk/~csetzer/lectures/critsys/14/index.html

▶ There you can find as well examples, and how to use IO in 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.
3 Levels of Analysis of SPARK Ada

- **Level 1 Analysis** (mode “check”)
  - Correct Ada syntax used.
  - SPARK-subset of Ada chosen, as described in the next subsection.

- **Level 2 Analysis** (mode “flow”)
  - Data flow analysis.
  - Information flow analysis.

- **Level 3 Analysis** (mode “prove”)
  - Generation of verification conditions.
  - Proof of verification conditions using automated and interactive theorem provers.
Two components of Level 2 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.
Annotations are added to the program expressing the correctness of the program. (Called Pre- and Post-conditions, more about this later.) SPARK Ada extracts verification conditions. They express that the program fulfils these correctness conditions. These verification are then fed into an automated theorem prover. The automated theorem prover is not always able to prove the correctness of the verification conditions, even if they are true. In that case one needs to feed verification conditions into an interactive theorem prover, e.g. the theorem prover Coq.
Three Levels of Analysis

- Idea is that the 3 different levels of analysis are applied depending on the criticality of the program.
- Some non-critical parts of the program might not be checked at all.
Annotations in Pre-2014 SPARK Ada

- In SPARK Ada additional annotations are added to the Ada code.
  - Specific to the 3 levels of analysis.
- In the versions of SPARK Ada up to 2012
  Referred to in the following as **Pre-2014 SPARK Ada**
  these annotations were written as **Ada comments**:
  - Ignored by Ada compilers.
  - Used by the SPARK Ada tools.
  - Syntax: start with **-- #**
  - Example:
    ```ada
    -- # global in out MyGlobalVariable;
    ```
Since Ada 2012 SPARK Ada commands are now part of the core language of Ada.

Therefore **SPARK Ada 2014** uses now these commands as part of its language.

Since the main book on SPARK Ada hasn’t been updated yet, in this module
- we will use **SPARK 2014 syntax**
- but outline as well how to write the same in **SPARK Pre-2014 syntax** using comments.
2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Conditions (Loops, Procedures and Functions)

2 (h) Example: A railway interlocking system
Activation of SPARK Mode

- To turn SPARK mode on:
  - Add in front of a compilation unit:
    ```ada
    pragma SPARK_Mode;
    ```
    or
    ```ada
    pragma SPARK_Mode(On);
    ```
  - or add after a package or procedure:
    ```ada
    with SPARK_Mode;
    ```
    or
    ```ada
    with SPARK_Mode => On;
    ```
- To turn it off:
  - Add in front of a compilation unit:
    ```ada
    pragma SPARK_Mode(Off);
    ```
  - or add after a package or procedure:
    ```ada
    with SPARK_Mode => Off;
    ```
- If spark mode is off it needs to be turned off in all units used by it.
Switching of SPARK Mode (of/off)

- Even with SPARK mode off, the first two levels of analysis are carried out.
- However no verification conditions are generated.
Pragmas

- **Pragma** are annotations to a unit in Ada. They occur at the beginning of a file. They were originally compiler directives, but are used in SPARK Ada for giving correctness conditions.

Syntax:

**Pragma** Mypragma;

or

**Pragma** Mypragma(parameter);
Aspects

Aspects are annotations to a package, procedure or function. They are used in SPARK Ada to give correctness annotations to a unit.

Syntax:

```
procedure A (X : in Integer)
   with keyword;
```

or

```
with keyword => value;
```

or

```
with keyword => (keyword2 => value, keyword3 => value);
```
Factors for Programming Languages Addressed by SPARK Ada

- **Logical Soundness.**
  - Problem: statement like $Y := F(X) + G(X)$:
    Order of evaluation not specified in Ada.
    Problem if $F(X)$ and $G(X)$ have side effects.
  - E.g. $F(X)$ has effect $Z := 0$,
    $G(X)$ has effect $Z := 1$. 
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.
    - They are a form of dependent types, but without compile time checking.
    - See Additional Material for Sect. 2 (a).
  - No threads (concurrency).
  - No generic types.
Expressive power.

- Hiding of variables allowed.
- Allows to specify strong assertions about variables
  - E.g. that a variable is only read but not written – an input variable
  - or only written but not read – an output variable.
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).
Support of Verifiability (Cont.)

- Some restrictions in previous versions of SPARK Ada have been relaxed:
  - Restrictions of procedures having no return statement and functions having exactly one return statement have been removed.
  - Restrictions regarding jumping out of if statements and loops have been removed.
No structures which are too complex to grasp.

- Almost **no subtyping**.
- Details about subtyping can be found in “Additional Material”.

SPARK Ada Concepts
SPARK Ada Concepts

Language should be as explicit as possible.

- Rules regarding **polymorphism** from pre-2014 SPARK Ada now relaxed
  - Overloading of functions and operators is now allowed.
  - **default parameters** are allowed from SPARK 2014 onwards.
  - **default record components** are not allowed.

- However **Array sliding** is not allowed:
  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.
Bounded space and time requirements.

- **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).

However **Recursion** (disallowed in pre-2014 SPARK Ada) is now allowed, so no bounded space guaranteed.
Avoiding Aliasing Problem

- If a procedure has
  - one input or input-output parameter
  - and another output or input-output parameter,
then it is not allowed to instantiate both parameters by the same variable.
- This is disallowed because it could cause the aliasing problem:
  - If it were allowed and the output parameter is changed while the input parameter is still used then the input parameter would be changed in the middle of the code.
  Under these circumstances correctness is no longer guaranteed.
Example: Wrong Exchange program

```ada
procedure Exchange (X : in out Boolean; Y : in out Boolean)

is begin
  X := X xor Y;
  Y := X xor Y;
  X := X xor Y;
end Exchange;

procedure Main (X : in out Boolean)

is begin
  Exchange (X, X);
  -- input/output parameter X and input/output parameter Y
  -- are instantiated by the same variable X
end Main;
```

---

Language Restrictions in SPARK Ada
Error Message by SPARK Ada

SPARK Ada reports at the call Exchange(X, X) for both parameters:
- main.adb:7:16: writable actual for “X” overlaps with actual for “Y” which is exported.
Sophisticated Built-in Checks

- SPARK Ada has lots of very sophisticated checks for programming errors built in.

- Example: code
  
  ```ada
  if A = 5 then
    if A = 6 then ···
  end if;
  ```

  is rejected, because the second statement is unreachable.

  the following code is accepted:

  ```ada
  if A = 5 then
    if A + 0 = 6 then ···
  end if;
  ```

- The number of features for detecting programming errors is growing with each version.
2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Conditions (Loops, Procedures and Functions)

2 (h) Example: A railway interlocking system
Data Flow Analysis – Parameters

- Remember the use of in/out/in-out parameters in Ada.
  - in - parameters 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**.
Initialisation of Variables

Initialisation of variables needs to happen in all branches of a conditional. For instance after the following statement

```plaintext
if Y > Z then
  X := 0.0;
end if;
```

X is considered as not initialised unless it has been initialised before.
Example (Data Flow Analysis)

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

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

- Mistake: ???
```
Example (Data Flow Analysis)

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

- **SPARK Ada reports 4 error messages:**
  - At “Exchange (X, Y : in out Float)”:
    warning: unused initial value of “X”.
  - At “T : Float;”:
    warning: variable “T” is assigned but never read
  - At “T := X”:
    warning: useless assignment to “T”, value never referenced
    warning: unused assignment
Example (Data Flow Analysis)

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

Note that SPARK Ada doesn’t notice that Y doesn’t change (that de-facto Y 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:

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

SPARK Ada reports the following error message:

- At “procedure Exchange (X, Y : in out Float)”
  warning: unused initial value of “X”
Example 3

Assume a procedure with body

\[ \begin{align*}
  Y & := X; \\
  Z & := Y;
\end{align*} \]

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 \textbf{output} variable (keyword “out”).
The following illustrates the dependencies of the variables on the original value \((X\sim, Y\sim, Z\sim)\) values:

\[
Y := X

Z := Y
\]
Global variables must be given status as input or output or input/output variables by the aspect “global”.

Syntax examples:

- **procedure** test (X : in out Integer)
  ```
  with Global => (Input => A, Output => B, In_Out => C);
  ```
- Or (after the same procedure declaration line)
  ```
  with Global => (Output => B);
  ```
- Or (after the same procedure declaration line)
  ```
  with Global => (In_Out => C);
  ```

Data flow analysis carries out the same analysis as for parameters.
Global variables pre-2014 syntax

- **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;
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:

```
with Global => A,B,C;
```

Neither parameters nor global variables can be changed. Therefore functions don’t have side effects.
Global Annotation pre-2014 SPARK Ada

-- # global A;
or
-- # global A,B,C;
State Variables from Other Packages

- If you are using a different package, you can change global variables in that other package.
- However you need to annotate them using the “Global” aspect.
- You need as well to have a `use` clause to access that other variable, even if only using variables in aspects.
- If state variables are not declared in the `.ads` file but the `.adb` file, then the situation is a bit more complicated.
  - You need to
    - have abstract state variables in the specification which are refined in the implementation file
    - deal with those abstract and refined variables in the package making use of it.
Example (Content of .ads Files)

File example.ads:

```solidity
package Example
    with SPARK_MODE
is
    Glob : Integer;

    procedure Init
        with Global => (Output => Glob);
end Example;
```

File main.ads:

```solidity
with Example; use Example;
-- Needed, although used only in annotations
procedure Main
    with SPARK_MODE,
        Global => (Output => Glob);
```
2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Conditions (Loops, Procedures and Functions)

2 (h) Example: A railway interlocking system
Use of Aspects for Annotating Information Flow

- Additional annotations on how variables depend on each other.
  Syntax examples:
  - `with Depends => (Y => X);`
    meaning $Y$ depends on $X$.
  - Several dependencies:
    `with Depends => (Y => X, X => Y);`
  - If $X$ gets a value independent of other variables:
    `with Depends => (Y => null);`
  - or depending on two variables:
    `with Depends => (X => (Y,Z));`
  - Often the new value of a variable depends on its own values, so we may have e.g.
    `with Depends => (X => X);`
    or
    `with Depends => (X => (X,Y));`
Syntax in pre 2014 SPARK Ada

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

```plaintext
procedure Exchange_And_Count(X,Y,Z: in out Integer)
with Depends => (X => Y,
                Y => X,
                Z => Z);

procedure Exchange_And_Count(X,Y,Z: in out Integer) 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 an error, since $Z$ depends on $Z$ and $X$ (via $T$):
  warning: missing dependency “$Z \Rightarrow X$”
- After correcting the specification to

```prolog
procedure Exchange_And_Count(X,Y,Z: in out Integer) with Depends => (X => Y,
Y => X,
Z => (X,Z));
```

information analysis succeeds.
Example 2

► What happens if we vary this program as follows?

```plaintext
procedure Exchange_And_Count(X,Y,Z: in out Integer) with Depends => (X => Y,
Y => X,
Z => Z);

procedure Exchange_And_Count(X,Y,Z: in out Integer) is
  T: Integer;
begin
  T:= X; X:= Y; Y:= T; Z:= T;
end ExchangeAndCount;
```
Example 2

- warning: unused initial value of “Z”
- warning: missing dependency “null => Z”
- warning: missing dependency “Z => X”
- warning: incorrect dependency “Z => Z”
Example 2 Corrected

```haskell
procedure Exchange_And_Count(X,Y: in out Integer; Z : out Integer)
  with Depends => (X => Y, Y => X, Z => X);

procedure Exchange_And_Count(X,Y: in out Integer; Z : out Integer) is
t: Integer;
begin
  t := x; x := y; y := t; z := t;
end ExchangeAndCount;
```
Example 3

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

\[
Y := X \\
Z := Y
\]

- **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.
Example 4

```plaintext
procedure Decrease_Increase( X,Y : in out Integer;
                                Z : in Integer)
    with Depends => (X => (X,Z), Y => (X,Y,Z));

procedure Decrease_Increase( X,Y : in out Integer;
                                Z : in Integer) is
begin
    loop
        exit when Z < 0 or X < 0;
        X := X - 1; Y := Y + 1;
    end loop;
end Decrease_Increase;
```
Example 4

- Y depends on X,Z although the only assignment to Y is
  \[ Y := Y + 1; \]
- This is since the resulting value of Y depends on how often the loop is executed.
- The condition for terminating the loop depends on Z and X, therefore Y depends on X,Z (and Y).
- The program is correct (no information flow errors found).
2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Conditions (Loops, Procedures and Functions)

2 (h) Example: A railway interlocking system
Section (f) divided into Section (f) and (g)

- We have divided Section (f), which was very long, into two Sections
  - Section (f) Verification Conditions (Simple Programs)
  - Section (g) Verification Conditions (Loops, Procedures and Functions)
- The Section on the railway example is now called Section (h).
GUI of SPARK Ada
GUI of SPARK Ada
GUI of SPARK Ada

- Started using command “gps”.
- In Bottom right you can see the command line commands executed, which you can use for executing commands from a command line directly (faster).
- I use gnatprove -P <path>/main.gpr however sometimes use options such as -proof=per_path.
GUI of SPARK Ada

gnatprove -P/home/csetzer/Dropbox/sparkada2014/critsys2014/sect2f/example2f-14/main.gpr --ide-progress-bar -U --proof=per_path
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate language ...
Phase 3 of 3: generation and proof of VCs ...
analyzing Test, 6 checks
test.adb:5:11: warning: overflow check might fail
test.adb:7:10: warning: overflow check might fail
gprbuild: *** compilation phase failed
gnatprove: error during generation and proof of VCs, aborting.

[2014-11-02 23:12:42] process exited with status 1, 100% (8/8), elapsed time: 10.76s
Pre- and Post-Conditions

- Two kinds of annotations are relevant:
  - **Pre-conditions**, e.g.:
    
    \[ \text{with Pre } \implies M \geq 0 \text{ and } N < 0; \]
    
    Expressing: at the beginning variable M is \( \geq 0 \) and variable N is \( < 0 \).
  - **Post-conditions**, e.g.:
    
    \[ \text{with Post } \implies M = M'_{\text{Old}} + 1; \]
    
    Expressing: after having executed the procedure the value of M is the same as the value of M when beginning the procedure (denoted by \( M'_{\text{Old}} \), incremented by 1.

- Then examiner generates formulae 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.
Notation in pre-2014 SPARK Ada

- **#pre** \( M \geq 0 \) and \( N < 0 \);
- **#post** \( M = M' + 1 \);
- Note that the \( M' \) is the pre-2014 SPARK Ada notation for \( X' \text{Old} \).
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-condition**s,
  - X’\textit{Old} stands for the value of X before executing the procedure,
  - X stands for the value after executing it,
  - e.g. X=X’\textit{Old} +1; expresses:
    The value of X after executing the procedure is the value of X before executing it + 1.
- X’\textit{Old} can occur only in **post-condition**s (especially not in assert statements introduced later);
  seems to be due to a restriction of Ada.
Verification Conditions

- When writing mathematical code (non-Ada code) we will write in the following for shortness $X \sim$ instead of $X'\text{Old}$. For instance we write

$$X = X \sim + 1$$

instead of

$$X = X'\text{Old} + 1$$
Connectives in Verification Conditions

- Formulae are built using:
  - Boolean valued expressions
    - E.g. $N = M$, $N < M$ or $F(N)$, if $F(N)$ of type Boolean,
  - Boolean connectives `and`, `or`, `not`, `xor`,
  - "if condition then conclusion" which stands for "condition $\rightarrow$ conclusion"
  - "if condition then ifcondition else elsecondition" which stands for
    $(\text{condition} \rightarrow \text{ifcondition}) \land
     (\neg (\text{condition}) \rightarrow \text{elsecondition})$
  - quantifiers `for all` $\cdots \Rightarrow \cdots$
    `for some` $\cdots \Rightarrow \cdots$
  - **Remark:** If in the pre-condition there are occurrences of quantifiers, the whole pre-condition is at the moment ignored. In the post-condition they are treated correctly.
Example Verification Conditions

Example (not very meaningful):

\[
\text{with} \implies \text{Pre} \implies ((M \geq 0 \lor N < 0) \text{ or } \left( \text{if not } (N > 0) \text{ then } M > 0 \right) \text{ and } \left( \text{if } N > 0 \text{ then } M > 0 \text{ else } M < 0 \right)),
\]

\[
\text{Post} \implies ((\text{for all } X \text{ in } 0..3 \implies g(X, M)) \text{ or } \left( \text{for some } X \text{ in Integer } \implies X = M' \text{Old } + 1 \right));
\]
Remark on $X'\text{Old}$

- $X'\text{Old}$ should only be used if $X$ is an input-output parameter (locally or globally)
  - If it is an input parameter only, then it cannot be changed, so $X$ and $X'\text{Old}$ are always the same.
    In that case SPARK Ada issues a warning.
  - If it is an output parameter only, then it has no initial value, so $X'\text{Old}$ is in fact undefined.
    In that case SPARK Ada doesn't (yet?) issue a warning.

- An output parameter of a procedure cannot occur in the pre-condition, because its initial value is undefined.
Example 1 (Verification Conditions)

- Assume the following wrong program:

```plaintext
procedure Wrong_Increment(X : in out Integer)
with Depends => (X => X),
Pre => X >= 0,
Post => X = X'Old + 1 and X >= 1;

procedure body Wrong_Increment(X : in out Integer) is
begin
X := X + X;
end Wrong_Increment;
```

- The mistake in this program is that the body should read `X := X + 1;` instead of `X := X + X;`.

- The post-condition is not fulfilled in general.
procedure Wrong_Increment(X : in out Integer)
  with Depends => (X => X),
  Pre => X >= 0,
  Post => X = X’Old + 1 and X >= 1;

procedure body Wrong_Increment(X : in out Integer) is
  begin
    X := X + X;
  end Wrong_Increment;

When going through the program we see that at the end
  X = X~ + X~
  (note we write for brevity X~ instead of X’Old).
Therefore, replacing X by X~ + X~ the post-condition reads:
  X~ + X~ = X~ + 1 \land X~ + X~ \geq 1
Example 1 (Verification Conditions)

procedure Wrong_Increment(X : in out Integer)
    with Depends => (X => X),
    Pre => X >= 0,
    Post => X = X'Old + 1 and X >= 1;

procedure body Wrong_Increment(X : in out Integer) is
    begin
        X := X + X;
    end Wrong_Increment;

That the pre-condition implies the post-condition therefore reads

\[ X \geq 0 \rightarrow (X + X = X + 1 \land X + 1 \geq 1) \]

Here \( \land \) binds more than \( \rightarrow \) therefore the formula is the same as

\[ X \geq 0 \rightarrow (X + X = X + 1 \land X + X \geq 1) \]

The above is called the verification condition derived from the pre-and post-conditions.
Example (Verification Conditions)

- Since in mathematics one usually writes lower case characters we replace $X$ by $x$.
- Since there are only occurrences of $x\sim$, we replace them by $x$.
- We obtain
  \[ x \geq 0 \rightarrow x + x = x + 1 \land x + x \geq 1 \]
- This above formulae has to be treated as being implicitly universally quantified, and stands for
  \[ \forall x.(x \geq 0 \rightarrow x + x = x + 1 \land x + x \geq 1) \]
Example (Verification Conditions)

\[ \forall x. (x \geq 0 \rightarrow x + x = x + 1 \land x + x \geq 1) \]

- This corresponds to what the correctness of the function means:
  - **For all inputs** \( x \), we have that, if \( x \) fulfils the **pre-condition**, then after executing the program it fulfils the **post-condition**.
- That the above formula does not hold can be shown by giving a counter example:
  - Take \( x := 0 \). Then \( x \geq 0 \) holds, but not \( x + x = x + 1 \).
- The program is incorrect, since the verification conditions don't hold (universally).
  - In order for the program to be correct it needs to be correct for all arguments.
  - Note that for \( x = 1 \) the formula is true, but the program is still incorrect.
SPARK Ada seems to work implicitly using the Why3 system, which is free software developed by the French research organisation INRIA.

Why3 is a tool which converts imperative code from an intermediate language (mlw and why3) into verification conditions which are then fed into various automated theorem provers Alt-ergo, CVC3, CVC4, and Z3.

- interactive theorem provers Coq and Isabelle.
Architecture of Why3 Platform

(Source: http://why3.lri.fr/queens/queens.pdf)
Why3 Platform

- Normally SPARK Ada seems to feed the verification conditions it has generated through why3 into alt-ergo and displays the verification conditions as proven or not proven.
  - There is an option of as well feeding them into Coq, but I have not succeeded in setting it up.
- In order to display the verification conditions, I found it easier to install why3 separately and run why3 on the generated .mlw files.
Result of Applying Why3 to .mlw Files
Main Verification Conditions in Why3

```why3
constant x : integer
axiom H : to_int x >= 0
constant o : int = to_int x + to_int x

goal WP_parameter_def :
in_range o &&
  (forall o1:integer.
   to_int o1 = o ->
   (forall x1:integer.
    x1 = o1 ->
    in_range (to_int x + 1) \ /
    to_int x1 = (to_int x + 1) \ /
    to_int x1 >= 1))
end
```
Interpreting the output of Why3

- SPARK Ada makes sure that one deals with **machine integers** (which are represented by finitary number of bits and therefore are between a **minimum** and **maximum** value) rather than **mathematical integers** which are **infinite**.

- Therefore there are
  - Two types of integers:
    - **integer** which are **machine integers** and therefore bounded,
    - **int** which are unbounded mathematical integers.
  - Operation **to_int** : integer → int
  - Relation **in_range** on **int** which checks whether int is in the range of integer.

- For a program SPARK Ada generates verification conditions which check that in each step of the program all integers stay in the range of machine integers.
In order to concentrate on the essence of Hoare logic, I will in the following ignore the fact that machine integers are finite.

I will extract from the generated verification conditions verification conditions one would get if we didn’t have this distinction.
In this example we obtain the following

\[ H1 : x \geq 0. \]
\[ H2 : o = x + x . \]
\[ H3 : o1 = o . \]
\[ H4 : x1 = o1 . \]
\[ \rightarrow \]
\[ C1 : x1 = x + 1. \]
\[ C2 : x1 \geq 1. \]

In the above (a format used in pre-2014 SPARK Ada)
H1, H2, ... are hypotheses,
C1, C2, ... are conclusions
and the above formula stands for

\[ x \geq 0 \land o = x + x \land o1 = o \land x1 = o1 \rightarrow x1 = x + 1 \land x1 \geq 1 \]
Simplified Generated Verification Conditions

- We choose the above since usually verification conditions will have the form

\[ H_1 \land H_2 \land \cdots \land H_n \rightarrow C_1 \land C_2 \land \cdots \land C_m \]
Simplified Generated Verification Conditions

H1 : \( x \geq 0 \).
H2 : \( o = x + x \).
H3 : \( o1 = o \).
H4 : \( x1 = o1 \).

→

C1 : \( x1 = x + 1 \).
C2 : \( x1 \geq 1 \).

We simplify this to

H1 : \( x \geq 0 \).

→

C1 : \( x + x = x + 1 \).
C2 : \( x + x \geq 1 \).

which is the formula we obtained before:

\[ x \geq 0 \rightarrow x + x = x + 1 \land x + x \geq 1 \]
Example 2 (Verification Conditions)

- Assume the correct program:

```plaintext
procedure Correct_Increment(X : in out Integer)
  with Depends => (X => X),
  Pre => X >= 0,
  Post => X = X'Old + 1 and X >= 1;

procedure body Correct_Increment(X : in out Integer) is
begin
  X := X + 1;
end Correct_Increment;
```
Example 2 (Cont.)

procedure Correct_Increment(X : in out Integer)
    with Depends => (X => X),
    Pre    => X >= 0,
    Post   => X = X'Old + 1 and X >= 1;
procedure body Correct_Increment(X : in out Integer) is
    begin
        X := X + 1;
    end Correct_Increment;

- When going through the program, we see that at the end
  - $X = X_{\sim} + 1$.
- Therefore the **post-condition** reads
  - $X_{\sim} + 1 = X_{\sim} + 1 \land X_{\sim} + 1 \geq 1$. 
Example 2 (Cont.)

procedure Correct_Increment(X : in out Integer)
    with Depends => (X =>> X),
    Pre => X >= 0,
    Post => X = X'Old + 1 and X >= 1;

procedure body Correct_Increment(X : in out Integer) is
    begin
        X := X + 1;
    end Correct_Increment;

▶ The complete verification condition is therefore

\[ X \geq 0 \rightarrow X + 1 = X + 1 \land X + 1 \geq 1 \]

which is then written as

\[ x \geq 0 \rightarrow x + 1 = x + 1 \land x + 1 \geq 1 \]
Output of Why3 Applied to Output from Spark Ada

```plaintext
constant x : integer

axiom H : to_int x >= 0

constant o : int = to_int x + 1

goal WP_parameter_def:
    in_range o &&
    (forall o1:integer.
     to_int o1 = o ->
     (forall x1:integer.
      x1 = o1 ->
       in_range (to_int x + 1) \/
       to_int x1 = (to_int x + 1) \/ to_int x1 >= 1))
end
```
Example 2 (Cont.)

- If we omit any condition needed because of the distinction between machine and mathematical integers this reads

  \[ H_1 : x \geq 0 \]
  \[ H_2 : o = x + 1 \]
  \[ H_3 : o_1 = o \]
  \[ H_4 : x_1 = o_1 \]
  \[ \rightarrow \]
  \[ C_1 : x_1 = x + 1 \]
  \[ C_2 : x_1 \geq 1 \]

- The last condition can be simplified to

  \[ H_1 : x \geq 0 \]
  \[ \rightarrow \]
  \[ C_1 : x+1 = x + 1 \]
  \[ C_2 : x+1 \geq 1 \]
Example 2 (Cont.)

H1 : \( x \geq 0 \)
\[ \rightarrow \]
C1 : \( x+1 = x + 1 \)
C2 : \( x+1 \geq 1 \)

- This condition is the same as what we obtained by hand:
  \[ x \geq 0 \rightarrow x + 1 = x + 1 \land x + 1 \geq 1 \]

- This condition would be provable, however the conditions on machine integers are not provable, as reported by SPARK Ada.
Assert Condition

- One can insert in between the procedures an **assert condition**.
- Now the formulae express:
  - From the **pre-condition** follows at that position the **assert condition**.
  - From the **pre-condition and the Assert condition** at that position follows the **post-condition**.
  - **Assert** conditions serve therefore as **intermediate proof-goals**.
- If one has several **assert-conditions**, they **accumulate**:
  If one has 2 **assert-conditions** for instance the conditions express
  - From the **pre-condition** follows the **first assert-condition** (at its place).
  - From the **pre-condition and the first assert-condition** (at its place) follows the **second assert-condition** (at its place)
  - From the **pre-condition and the first assert-condition** (at its place) and the **second assert-condition** (at its place) follows the **post-condition**.
The pre-2014 syntax was

```ada
--# check X = 1;
```

The new syntax is

```ada
pragma Assert (X = 1);
```
Example Assert Condition

Consider the program:

```plaintext
procedure One_Assert(X : in out Integer)
    with Depends => (X => X),
    Pre      => X = 0,
    Post     => X = 2;

procedure body One_Assert(X : in out Integer) is
    begin
        X := X + 1;
        pragma Assert (X = 1);
        X := X + 1;
    end One_Assert;
```

That the **pre-condition** implies the assert condition is the following verification condition:

\[ X \sim = 0 \rightarrow X \sim + 1 = 1; \]
Example Assert Condition

Consider the program:

```plaintext
procedure OneAssert(X : in out Integer)
    with Depends => (X => X),
    Pre => X = 0,
    Post => X = 2;

procedure body OneAssert(X : in out Integer) is
    begin
        X := X + 1;
        pragma Assert (X = 1);
        X := X + 1;
    end OneAssert;
```

That the pre- and the assert-condition implies the post-condition is the following verification condition:

\[ X \sim_0 \land X \sim_1 + 1 = 1 \rightarrow (X \sim_1 + 1) + 1 = 2; \]
Output of Why3 Applied to Output from Spark Ada

```plaintext
constant x : integer

axiom H : to_int x = 0

constant o : int = to_int x + 1

goal WP_parameter_def :
  in_range o &&
  (forall o1:integer.
   to_int o1 = o ->
    (forall x1:integer.
     x1 = o1 ->
      to_int x1 = 1 &&
       (let o2 = to_int x1 + 1 in
        in_range o2 &&
        (forall o3:integer.
         to_int o3 = o2 ->
          (forall x2:integer. x2 = o3 -> to_int x2 = 2)))))
```

Generated Verification Conditions

- If we omit any condition needed because of the distinction between machine and mathematical integers the part until the assert conditions reads

\[
\begin{align*}
H1 : & \quad x = 0 \\
H2 : & \quad o = x + 1 \\
H3 : & \quad o1 = o \\
H4 : & \quad x1 = o1 \\
\rightarrow & \\
C1 : & \quad x1 = 1
\end{align*}
\]

which is just

\[x = 0 \rightarrow x + 1 = 1\]
Generated Verification Conditions

- Condition that pre-condition and assert condition imply post-condition reads

<table>
<thead>
<tr>
<th>Condition</th>
<th>Simplified</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1 : x = 0</td>
<td>H1 : x = 0</td>
</tr>
<tr>
<td>H2 : o = x + 1</td>
<td>H2 : x + 1 = 1</td>
</tr>
<tr>
<td>H3 : o1 = o</td>
<td>H2 : x + 1 = 1</td>
</tr>
<tr>
<td>H4 : x1 = o1</td>
<td>C1 : (x + 1) + 1 = 2</td>
</tr>
<tr>
<td>H5 : x1 = 1</td>
<td></td>
</tr>
<tr>
<td>H6 : o2 = x1 + 1</td>
<td></td>
</tr>
<tr>
<td>H7 : o3 = o2</td>
<td></td>
</tr>
<tr>
<td>H8 : x2 = o3</td>
<td></td>
</tr>
<tr>
<td>C1 : x2 = 2</td>
<td></td>
</tr>
</tbody>
</table>

which is just

\[ x = 0 \land x + 1 = 1 \rightarrow (x + 1) + 1 = 2 \]
There is as well an assert-and-cut directive.

The difference is:

- If one has an assert-and-cut directive, then previous assert conditions are not included.
- So if one has two assert-and-cut conditions then the verification condition are
  
  - From the **pre-condition** follows the **first assert condition** (at its place).
  - From the **first assert-condition** (at its place) follows the **second assert condition** (at its place)
  - From the **second assert condition** (at its place) follows the **post-condition**.
SPARK Ada Syntax for Assert

- The pre-2014 syntax was
  
  ```ada
  -- # assert X = 1;
  ```

- The new syntax is

  ```ada
  pragma Assert_And_Cut (X = 1);
  ```
If one has assert conditions instead, then the first condition is as before, but the second and third are

- From the **pre-condition and the first assert condition** (at its place) follows the **second assert condition** (at its place)
- From the **pre-condition and the first assert condition** (at its place) and the **second assert condition** (at its place) follows the **post-condition**.
Consider the program:

```haskell
procedure Ex_ASSERT_AND_CUT(X : in out Integer) with Depends => (X => X),
    Pre  => X = 0,
    Post => X = 3;

procedure Ex_ASSERT_AND_CUT(X : in out Integer) is
    begin
        X := X + 1;
        pragma Assert_and_Cut (X = 1);
        X := X + 1;
        pragma Assert_and_Cut (X = 2);
        X := X + 1;
    end Ex_ASSERT_AND_CUT;
```

Example: Assert and Cut Condition

Consider the program:

```haskell
procedure Ex_ASSERT_AND_CUT(X : in out Integer)
    with Depends => (X => X),
    Pre  => X = 0,
    Post => X = 3;

procedure Ex_ASSERT_AND_CUT(X : in out Integer) is
    begin
        X := X + 1;
        pragma Assert_and_Cut (X = 1);
        X := X + 1;
        pragma Assert_and_Cut (X = 2);
        X := X + 1;
    end Ex_ASSERT_AND_CUT;
```
Example Assert Condition

- The implication from the **pre-condition** to the first assert condition is:
  \[ X \sim = 0 \rightarrow X \sim +1 = 1. \]

- That the first assert condition implies the second assert condition is the following:
  \[ X = 1 \rightarrow X +1 = 2; \]

- That the second assert condition implies the **post-condition** is the following:
  \[ X = 2 \rightarrow X +1 = 3; \]
constant x : integer

axiom H : to_int x = 0

goal WP_parameter_def :
  ((let o = to_int x + 1 in
    in_range o &&
    (forall o1:integer.
      to_int o1 = o -> (forall x1:integer. x1 = o1 -> to_int x1 = 1))) \/
    (forall x1:integer.
      to_int x1 = 1 -> (let o = to_int x1 + 1 in
        in_range o &&
        (forall o1:integer.
          to_int o1 = o -> (forall x2:integer. x2 = o1 -> to_int x2 = 2)))))) \/
    (forall x1:integer.
      to_int x1 = 2 -> (let o = to_int x1 + 1 in
        in_range o &&
        (forall o1:integer.
          to_int o1 = o -> (forall x2:integer. x2 = o1 -> to_int x2 = 3))))

and
Example 2 (Cont.)

- If we omit any condition needed because of the distinction between machine and mathematical integers we obtain 3 conditions. Pre condition implies first assert condition:

H1 : x = 0  
H2 : o = x + 1  
H3 : o1 = o  
H4 : x1 = o1

→

C1 : x1 = 1

- This can be simplified to

H1 : x = 0

→

C1 : x + 1 = 1
Example 2 (Cont.)

- First assert condition implies second assert condition

  H1 : \( x1 = 1 \)
  H2 : \( o = x1 + 1 \)
  H3 : \( o1 = o \)
  H4 : \( x2 = o1 \)
  \(
  \Rightarrow
  
  C1 : \( x2 = 2 \)
  
- This can be simplified to

  H1 : \( x1 = 1 \)
  \(
  \Rightarrow
  
  C1 : \( x1 + 1 = 2 \)
Example 2 (Cont.)

- Second assert condition implies post-condition

H1 : \( x_1 = 2 \)
H2 : \( o = x_1 + 1 \)
H3 : \( o_1 = o \)
H4 : \( x_2 = o_1 \)
\[ \rightarrow \]
C1 : \( x_2 = 3 \)

- This can be simplified to

H1 : \( x_1 = 2 \)
\[ \rightarrow \]
C1 : \( x_1 + 1 = 3 \)
Additional Use of Pre-Condition

- In the .mlw file the pre-condition is formulated as an axiom.
- Therefore the pre-condition can be used for proving all statements including
  - from first Assert_And_Cut statement follows second Assert_And_Cut statement,
  - from last Assert_And_Cut statement follows post-condition.
Many many languages (e.g. C++, Java) have an assert directive.

- The condition is then usually checked at run time, e.g. in C++

```cpp
#include <assert.h>
assert (myInt!=NULL);
```

- Checking of this condition can be disabled using

```cpp
#define NDEBUG
```

- In the Boost library for C++ there is as well a static assert.

```
BOOST_STATIC_ASSERT(std::numeric_limits<int>::digits >= 32);
```

This doesn’t allow to check values of the program which are obtained after the execution of some code.
In Ada 2005 assert was introduced to be used similarly to other languages:

- depending on pragmas (Assert, Assertion_Policy), the assert statement is checked with messages being printed out, or it is ignored.

Therefore they were went to be decidable conditions.

- That explains why X’Old is not allowed in assert statements, since it would require the compiler to keep track of the original value of X.

In Ada 2012 pre- and post-conditions were added, which can be treated similarly.

In Ada 2012 the intention for pre- and post-conditions and assert pragmas was that they can be used as well for verification tools, such as in SPARK Ada.
Work Around for \( X' \text{Old} \) not allowed in Assert statements

- \( X' \text{Old} \) is not allowed in the pragmas Assert and Assert_And_Cut.
- The way around this is to use an auxiliary constant which contains a copy of \( X' \text{Old} \):

Example:

```pascal
procedure Test(X : in out Integer) with Depends => (X => X),
Post => (X = X'Old + 2);

procedure body Test(X : in out Integer) is
    Xold : constant Integer := X;
begin
    X := X + 1;
    pragma Assert_And_Cut (X = Xold + 1);
    X := X + 1;
end Test
```
Work Around for X’\texttt{Old} not allowed in Assert statements

- This is legal in SPARK Ada, although X\texttt{old} is not used in the Ada code, since it occurs in the assert statement.
- It works because one of the axioms added by SPARK Ada is that X\texttt{old} = X’\texttt{Old}, which can be used in proving the two verification conditions.
  - This is formulated as an axiom, which can be used in all verification conditions.
Conditionals

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

```plaintext
procedure Simple (X : in out Integer)
with Depends => (X => X),
    Pre => X < 0,
    Post => X > 0;

procedure body Simple (X : in out Integer) is
begin
    if X > 1 then X:= -1;
    else X:= 2;
end if;
end Simple;
```
Conditionals

procedure Simple (X : in out Integer) 
  with Depends  =>  (X => X),  
                 Pre   =>  X < 0,  
                 Post  =>  X > 0;

procedure body Simple (X : in out Integer) is
begin
  if X > 1 then  X:= -1;
          else  X:= 2;  end if;
end Simple;

The verification conditions are:
  ▶ From the pre-condition follows the post-condition, in case X ∼ > 1,  
     (then X at the end is -1):
    ▶ X ∼ < 0 ∧ X ∼ > 1 → -1 > 0
  ▶ From the pre-condition follows the post-condition, in case ¬(X ∼ ≥ 1), (then X at the end is 2):
    ▶ X ∼ < 0 ∧ ¬(X ∼ > 1) → 2 > 0 .
Main Verification Conditions in Why3

constant x : integer

axiom H : to_int x < 0

goal WP_parameter_def:
  if to_int x > 1 then forall o:integer.
    to_int o = (-1) ->
    (forall x1:integer. x1 = o -> to_int x1 > 0)
  else forall o:integer.
    to_int o = 2 -> (forall x1:integer. x1 = o -> to_int x1 > 0)
end
Here an if then else construct is used which is not a standard logical constructs, but can be translated into two conditionals: The first one corresponding to the if clause:

\[ H1 : x < 0. \]
\[ H2 : x > 1. \]
\[ H3 : o = -1. \]
\[ H4 : x1 = o. \]
\[ \rightarrow \]
\[ C1 : x1 > 0. \]

Simplified:

\[ H1 : x < 0. \]
\[ H2 : x > 1. \]
\[ \rightarrow \]
\[ C1 : -1 > 0. \]

This corresponds to the true formula

\[ x < 0 \land x > 1 \rightarrow -1 > 0 \]
The second one is the else clause:

$$\begin{align*}
H1 & : x < 0. & \quad \text{Simplified:} & \quad H1 & : x < 0. \\
H2 & : \neg (x > 1). & \quad H2 & : \neg (x > 1). \\
H3 & : o = 2. & \quad \rightarrow & \quad C1 & : 2 > 0. \\
H4 & : x1 = o. & \quad \rightarrow & \quad C1 & : x1 > 0.
\end{align*}$$

This corresponds to the **true formula**

$$x < 0 \land \neg (x > 1) \rightarrow 2 > 0$$
One can add as well `Assert` and `Assert_And_Cut` pragmas in branches:

```plaintext
procedure Simple2 (X : in out Integer)
with Depends => (X => X),
    Pre    => X < 0,
    Post   => X > 0;

procedure body Simple2 (X : in out Integer) is
begin
    if X > 1 then X := -1;
        pragma Assert (X = 1);
    else
        X := 2;
        pragma Assert (X = 2);
    end if;
end Simple2;
```
Conditionals

procedure Simple2 (X : in out Integer)
  with Depends ==> (X => X),
    Pre ==> X < 0,  Post=> X > 0;

procedure body Simple2 (X : in out Integer) is
begin if X > 1 then X:= -1; pragma Assert (X = 1);
    else X:= 2;  pragma Assert (X = 2);
end if;
end Simple2;

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

procedure Simple2 (X : in out Integer)
   with Depends  =⇒ (X =⇒ X),
      Pre       =⇒ X < 0,  Post=⇒ X > 0;

procedure body Simple2 (X : in out Integer) is
begin if X > 1 then X:= -1; pragma Assert (X = 1);
   else X:= 2; pragma Assert (X = 2);
end if;
end Simple2;

- So we have the following condition:
  - Pre-condition implies assert-condition in the if case:
    \[ X < 0 \land X > 1 \rightarrow -1 = 1 \]
  - From the **pre-condition** and the assert-condition in the if case follows
    the **post-condition** (where X = -1):
    \[ X < 0 \land X > 1 \land -1 = 1 \rightarrow -1 > 0. \]
Conditionals

procedure Simple2 (X : in out Integer) with Depends => (X => X),
     Pre => X < 0, Post=> X > 0;

procedure body Simple2 (X : in out Integer) is
begin if X > 1 then X:= -1; pragma Assert (X = 1);
   else X:= 2; pragma Assert (X = 2);
end if;
end Simple2;

▶ Precondition implies assert-condition in the else case:
   X~ < 0 ∧¬(X~ > 1) → 2 = 2
▶ From the pre-condition and the assert-condition in else case follows
   the post-condition (where X = 2):
   X~ < 0 ∧¬(X~ > 1) ∧ 2 = 2 → 2 > 0.
Main Verification Conditions in Why3

constant x : integer

axiom H : to_int x < 0

goal WP_parameter_def :
  if to_int x > 1 then forall o:integer.
    to_int o = (- 1) ->
      (forall x1:integer.
        x1 = o -> to_int x1 = 1 && to_int x1 > 0)
  else forall o:integer.
    to_int o = 2 ->
      (forall x1:integer. x1 = o -> to_int x1 = 2 && to_int x1 > 0)
end
If case up to assert statement:

- H1 : \( x < 0 \).
- H2 : \( x > 1 \).
- H3 : \( o = -1 \).
- H4 : \( x1 = o \).

\[ \rightarrow \]

- C1 : \( x1 = 1 \).

This corresponds to the true formula

\( x < 0 \land x > 1 \rightarrow -1 = 1 \)
If case via assert statement implies post-condition:

H1 : x < 0.  Simplified:  H1 : x < 0.
H4 : x1 = o.  →
H5 : x1 = 1.  C1 : -1 > 0.

→
C1 : x1 > 0.

This corresponds to the true formula

\[ x < 0 \land x > 1 \land -1 = 1 \rightarrow -1 > 1 \]
Extracted Verification Conditions from Why3

- Else case up to assert statement:

  \( H_1 : x < 0. \)
  \( H_2 : \neg(x > 1). \)
  \( H_3 : o = 2. \)
  \( H_4 : x_1 = o. \)

  \( \rightarrow \)

  \( C_1 : 2 = 2. \)

  Simplified: \( H_1 : x < 0. \)
  \( H_2 : \neg(x > 1). \)

  \( \rightarrow \)

  \( C_1 : x_1 = 2. \)

This corresponds to the true formula

\[ x < 0 \land \neg(x > 1) \rightarrow 2 = 2 \]
Else case via assert statement implies post-condition:

\[ H1 : x < 0. \]
\[ H2 : \neg(x > 1). \]
\[ H3 : o = 2. \]
\[ H4 : x1 = o. \]
\[ H5 : x1 = 2. \]
\[ \rightarrow \]
\[ C1 : x1 > 0. \]

This corresponds to the true formula:

\[ x < 0 \land \neg(x > 1) \land 2 = 2 \rightarrow 2 > 0 \]
Consider replacing check by assert:

```plaintext
procedure Simple2 (X : in out Integer)
  with Depends  =>  (X => X),
  Pre          =>  X < 0,
  Post         =>  X > 0;

procedure body Simple2 (X : in out Integer)
begin
  if X > 1 then
    X := -1;
    pragma Assert_And_Cut (X = 1);
  else
    X := 2;
    pragma Assert_And_Cut (X = 2);
  end if;
end Simple2;
```
Using Assert_And_Cut Above

- We get similar statements from the **pre-condition** to the conditional as for `assert`.
- However from the conditional to the **post-condition** we get two conditions:
  - From the assert condition in the if-branch follows the **post-condition**:
    \[ X = 1 \rightarrow X > 0 \]
  - From the assert condition in the else-branch follows the **post-condition**:
    \[ X = 2 \rightarrow X > 0 \]
Main Verification Conditions in Why3

```
constant x : integer

axiom H : to_int x < \theta

goal WP_parameter_def :
  if to_int x > 1 then (forall o:integer.
    to_int o = (- 1) ->
    (forall x1:integer. x1 = o -> to_int x1 = 1)) \/
    (forall x1:integer. to_int x1 = 1 -> to_int x1 > \theta)
  else (forall o:integer.
    to_int o = 2 -> (forall x1:integer. x1 = o -> to_int x1 = 2)) \/
    (forall x1:integer. to_int x1 = 2 -> to_int x1 > \theta)
end
```
If case up to assert statement same as before:

H1 : \( x \leq 0 \).
H2 : \( x \geq 1 \).
H3 : \( o = -1 \).
H4 : \( x1 = o \).

\( \rightarrow \)

\( C1 : x1 = 1 \).

Simplified:

H1 : \( x < 0 \).
H2 : \( x > 1 \).

\( \rightarrow \)

C1 : \( -1 = 1 \).

This corresponds to the true formula

\( x < 0 \land x > 1 \rightarrow -1 = 1 \)
Assert case in if-case implies post-condition:

\[ H_1 : x_1 = 1. \]
\[ \rightarrow \]
\[ C_1 : x_1 > 0. \]

This corresponds to the true formula

\[ x_1 = 1 \rightarrow x_1 > 0 \]
Else case up to assert statement same as before:

<table>
<thead>
<tr>
<th>H1</th>
<th>x &lt; 0.</th>
<th>Simplified:</th>
<th>H1</th>
<th>x &lt; 0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2</td>
<td>( \lnot (x &gt; 1) ).</td>
<td>H2</td>
<td>( \lnot (x &gt; 1) ).</td>
<td></td>
</tr>
<tr>
<td>H3</td>
<td>o = 2.</td>
<td>(\to) C1</td>
<td>2 = 2.</td>
<td></td>
</tr>
<tr>
<td>H4</td>
<td>x1 = o.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \to \)

\( C1 \) : x1 = 2.

This corresponds to the true formula

\[ x < 0 \land \lnot (x > 1) \to 2 = 2 \]
 Assert statement in else case implies post-condition:

\[
\begin{align*}
\text{H1} & : \ x1 = 2. \\
\rightarrow \\
\text{C1} & : \ x1 > 0.
\end{align*}
\]

This corresponds to the true formula

\[
x1 = 2 \rightarrow x1 > 0
\]
2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Conditions (Loops, Procedures and Functions)

2 (h) Example: A railway interlocking system
Use of pre- and post-conditions is often referred to by “contract”.

- Goes back to Bertrand Meyer who coined the notion “design by contract (TM)”.
- Unfortunately “design by contract (TM)” is a trade mark of Eiffel.
  Eiffel = programming language designed by Bertrand Meyer.
- Therefore we can only use the word contract freely.

The contract given by pre- and post-condition is:
If the procedure or function is provided with data which fulfil the pre-conditions,
it will, if it terminates, have produced data which fulfil the post-conditions.
Assert between conditionals:

- Using `Assert_And_Cut` directives between conditionals allows to avoid an exponential blow up if one has too many conditionals.
- If we have $n$ “if then else” statements in sequence without any `Assert_And_Cut` conditions
  - then we have $2^n$ conditions,
  - namely that from the pre-condition and one selected branch of each conditional (if-case or else-case) follows the post-condition.
  - Note that there are $2^n$ possibilities of selecting from each conditional either the if or the else case.
- If we have `Assert_And_Cut` statements between each conditional this is
  - broken up into two statements per conditionals,
  - namely that from the previous `Assert_And_Cut` statement (or the pre-condition) and one branch follows the next `Assert_And_Cut` statement (or post-condition) assert condition and one branch (the if-case or the else-case) follows the next assert statement.
  - In total we have $2n$ verification conditions.
Procedures with Loops

- If one has a loop, a so called **loop invariant** is required.
- A loop invariant is essentially an **Assert_And_Cut** statement however we have access to the values of the variables they had before starting the loop.
  - Value before entering the loop is denoted by $X'_{\text{Loop_Entry}}$.
- Syntax example:
  ```
  pragma Loop_Invariant (M + N = M'_{\text{Loop_Entry}} + N'_{\text{Loop_Entry}});
  ```
- If one has one **pre-condition**, one **Loop_Invariant** at the beginning, and one **post**-condition, the examiner generates verification conditions expressing:
  - From the **pre-condition** follows, when first entering the loop, the condition of **Loop_Invariant**.
  - From the **Loop_Invariant** follows, if exit conditions are false, the condition of **Loop_Invariant** after one step.
  - From the **Loop_Invariant** follows, if one exit condition is true, the **post**-condition.
Pre-2014 syntax

- Syntax example:
  ```
  -- # assert M +N= M~+N~;
  ```

- In pre-2014 SPARK Ada one could make use of M~; this is no longer allowed in SPARK Ada 2014; instead M’Loop_Entry is introduced in SPARK Ada 2014. Note that if the procedure starts with the loop then

  \[ M'\text{Loop\_Entry} = M~ \]
Procedures with Loops

▶ The loop-invariant can occur anywhere in the main path of the loop
  ▶ Especially it cannot be inside an if clause.
▶ If the loop-invariant occurs after an exit statement, one needs as well to verify that from the pre-condition follows the post-condition, in case the condition for exiting the loop was true (in this case the Loop_Invariant will not be reached).
  ▶ In the examples below the assert condition will always occur at the beginning of the loop, before any exit statement.
Example

procedure Example(M, N : in out Integer)
with Depends => (M => M,
N => (M,N)),
Pre => (M >= 0),
Post => (M + N = M’Old + N’Old and M < 0);

procedure body Example(M,N : in out Integer) is
begin
loop
pragma Loop_Invariant (M + N
= M’Loop_Entry + N’Loop_Entry);
M := M - 1;
N := N + 1;
exit when M < 0;
end loop;
end Example;
Verif. Cond. 1: **Pre-Condition Implies Loop_Invariant**

procedure Example(M, N : in out Integer)
  with Pre => (M >= 0),
  Post => (M + N = M’Old + N’Old and M < 0);
procedure body Example(M, N : in out Integer) is
  begin loop
    pragma Loop_Invariant (M + N = M’Loop_Entry + N’Loop_Entry);
    M := M - 1; N := N + 1; exit when M < 0;
  end loop; end Example;

- When entering the loop we have
  \[ M = M’\text{Loop\_Entry} = M\sim, \ N = N’\text{Loop\_Entry} = N\sim. \]

- So that from the pre-condition \( M\sim \geq 0 \) follows that at the beginning of the loop we have \( M + N = M’\text{Loop\_Entry} + N’\text{Loop\_Entry} \), reads:
  \[ M\sim \geq 0 \rightarrow M\sim + N\sim = M\sim + N\sim \]
Verification Condition by SPARK Ada

H1 : m \geq 0

\rightarrow

C1 : m + n = m + n
Verif. Cond. 2: **Loop_Invariant** Implies **Loop_Invariant**

procedure Example(M, N : in out Integer)
  with Pre => (M >= 0),
  Post => (M + N = M'Old + N'Old and M < 0);

procedure body Example(M,N : in out Integer) is
  begin loop
    pragma Loop_Invariant (M + N = M'Loop_Entry + N'Loop_Entry);
    M := M - 1; N := N + 1; exit when M < 0;
  end loop; end Example;

▶ Note M'**Loop_Entry** is value before starting the loop, therefore is always M~.

▶ That if loop invariant is fulfilled, and the exit condition is false the loop invariant follows in the next step is the condition:

\[
M + N = M\sim + N\sim \land \neg(M - 1 < 0) \\
\to (M - 1) + (N + 1) = M\sim + N\sim
\]
Verification Condition by SPARK Ada

H1 : \( m_1 + n_1 = m + n \)
H2 : \( o = m_1 - 1 \)
H3 : \( o_1 = o \)
H3 : \( m_2 = o_1 \)
H4 : \( o_2 = n_1 + 1 \)
H5 : \( o_3 = o_2 \)
H6 : \( n_2 = o_3 \)
H7 : \( \neg (m_2 < 0) \)

\[ \rightarrow \]

C1 : \( m_2 + n_2 = m + n \)

This can be simplified to

H1 : \( m_1 + n_1 = m + n \)
H2 : \( \neg (m_1 - 1 < 0) \)

\[ \rightarrow \]

C1 : \( (m_1 - 1) + (n_1 + 1) = m + n \)
Verif. Cond. 3: **Loop_Invariant** Implies **Post**-Condition

procedure Example(M, N : in out Integer) 
  with Pre => (M >= 0), 
  Post => (M + N = M’Old + N’Old and M < 0);

procedure body Example(M,N : in out Integer) is 
  begin loop 
    pragma Loop_Invariant (M + N = M’Loop_Entry + N’Loop_Entry); 
    M := M - 1; N := N + 1; exit when M < 0; 
  end loop; end Example;

 ► That if loop invariant is fulfilled, and the exit condition is true the post-condition follows is the condition:

\[
M + N = M' + N' \land M - 1 < 0 
\rightarrow (M - 1) + (N + 1) = M' + N' \land M - 1 < 0
\]
Verification Condition by SPARK Ada

H1 : \( m_1 + n_1 = m + n \)
H2 : \( o = m_1 - 1 \)
H3 : \( o_1 = o \)
H3 : \( m_2 = o_1 \)
H4 : \( o_2 = n_1 + 1 \)
H5 : \( o_3 = o_2 \)
H6 : \( n_2 = o_3 \)
H7 : \( m_2 < 0 \)

→

C1 : \( m_2 + n_2 = m + n \)
C2 : \( m_2 < 0 \)

Simplified Version:

H1 : \( m_1 + n_1 = m + n \)
H2 : \( m_1 - 1 < 0 \)

→

C1 : \( (m_1 - 1) + (n_1 + 1) = m + n \)
C2 : \( m_1 - 1 < 0 \)
SPARK Ada verifies all verification conditions except for those referring to integers being in the range of machine integers.
If there is no `Loop_invariant`, SPARK Ada doesn’t seem to behave correctly at the moment.
- It only generates a loop condition corresponding to the case where the loop is always exited the first time, which is not sufficient to guarantee correctness.

In pre-2014 SPARK Ada it issued a warning.
- That’s correct behaviour since one cannot generate any verification conditions for a loop without a `Loop_Invariant`.
- This is because we don’t know how many times the loop is executed.
Verification Conditions when Combining Procedures

- 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.
Example (Combining Procedures, test.ads)

```
pragma SPARK_MODE;
package Test is
  procedure Init (A : out Integer)
    with Depends => (A => null),
    Post => (A = 0);
  procedure Inc (A : in out Integer)
    with Depends => (A => A),
    Pre => (A >= 0),
    Post => (A >= 2);
  procedure Main (A : out Integer)
    with Depends => (A => null),
    Post => (A > 1);
end Test;
```
Example (Combining Procedures, test.adb)

```ada
pragma SPARK_MODE;
package body Test is

procedure Init (A : out Integer) is
begin
A := 0;
end Init;

procedure Inc (A : in out Integer) is
begin
A := A + 2;
end Inc;

procedure Main (A : out Integer) is
begin
Init(A);
Inc(A);
end Main;
end Test;
```
Combining ads and adb files into One

- For ease of discussion we will in the following combine for the main function the ads and adb parts into one.
  - This is in this form not correct SPARK Ada code.
Data/Information Flow Analysis for Main

procedure Init (A : out Integer)
  with Depends => (A => null),
  Post => (A = 0);

procedure Inc (A : in out Integer)
  with Depends => (A => A),
  Pre => (A >= 0); Post=> (A >= 2);

procedure Main (A : out Integer)
  with Depends => (A => null),
  Post => (A > 1) is
begin Init(A); Inc(A); end Main;

▶ That in Main variable A is an out variable follows since in Init variable A is output variable, and in Inc it is an input/output variable.
▶ That Main derives variable A from nothing follows, since Init derives A from nothing, and Inc derives A from A.
Verification Condition 1 for Main

```
procedure Init (A : out Integer)
  with Depends => (A => null),
  Post => (A = 0);

procedure Inc (A : in out Integer)
  with Depends => (A = A),
  Pre => (A >= 0); Post=> (A >= 2);

procedure Main (A : out Integer)
  with Depends => (A => null),
  Post => (A > 1) is
begin
  Init(A); Inc(A);  end Main;
```

- The first condition is that the pre-condition of Main(A) implies in its body the pre-condition of Init(A).
- Since there are no such explicit condition, they are set to true and we get (or can ignore it)

\[
\text{True} \rightarrow \text{True}
\]
Verification Condition 2 for Main

**procedure** Init (A : **out** Integer)
  **with** Depends  =>  (A => null),
  Post  =>  (A = 0);

**procedure** Inc (A : **in out** Integer)
  **with** Depends  =>  (A => A),
  Pre  =>  (A >= 0);  **Post** =>  (A >= 2);

**procedure** Main (A : **out** Integer)
  **with** Depends  =>  (A => null),
  Post  =>  (A > 1) **is**
  **begin** Init(A); Inc(A);  **end** Main;

- The second condition, is that in the body of Main(A), the post-condition of the call of Init(A) implies the pre-condition of the call of Inc(A).

\[
A = 0 \rightarrow A \geq 0
\]
Verification Condition 2 for Main

**procedure** Init \((A : \textbf{out} \text{ Integer})\)

with Depends \(\Rightarrow (A \Rightarrow \text{null})\),

Post \(\Rightarrow (A = 0)\);

**procedure** Inc \((A : \textbf{in out} \text{ Integer})\)

with Depends \(\Rightarrow (A \Rightarrow A)\),

Pre \(\Rightarrow (A \geq 0)\); Post\(\Rightarrow (A \geq 2)\);

**procedure** Main \((A : \textbf{out} \text{ Integer})\)

with Depends \(\Rightarrow (A \Rightarrow \text{null})\),

Post \(\Rightarrow (A > 1) \textbf{is}\)

begin Init(A); Inc(A); end Main;

- The third condition, is that in the body of Main(A), the post-condition of the call of Inc(A) implies the post-condition of Main(A):

\[ A \geq 2 \rightarrow A > 1 \]
Verification Conditions for Functions

- We can define pre- and post-conditions for functions as for procedures.
- If a function has name Myfunction, one can in the post-condition refer to the result returned by this function using $\text{Myfunction}'\text{Result}$.

Example:

```plaintext
function Myfunction (A : Integer) return Integer
    Post => (Myfunction'\text{Result} = A + 3);
```

- Since functions have no side effects, except in rare cases the result of the function should depend on all arguments, and on any global variables mentioned in the specification.
  - Therefore usually no \texttt{Depends} clause is needed.
For a function returning argument X + 1:

```ada
-- # return X+1
```

For a function with post-condition that the returned value is > Y:

```ada
-- # return R => R > Y;
```
Example 1 (Verification Conditions for Functions)

\[
\text{function Test}(X : \text{Integer}) \ \text{return} \ \text{Integer} \\
\quad \text{with Pre} \quad = > \quad (X > 0), \\
\quad \text{Post} \quad = > \quad (\text{Test’Result} > 1);
\]

\[
\text{function Test}(X : \text{Integer}) \ \text{return} \ \text{Integer} \ \text{is} \\
\quad \text{begin} \\
\quad \quad \text{return} \ X + 1; \\
\quad \text{end Test;}
\]

- The verification condition is

\[
X > 0 \rightarrow X + 1 > 1
\]

- Does it hold?
Example 2 (Verification Conditions for Functions)

function Test(X : Integer) return Integer
    with Pre  =>  (X > 0),
    Post    =>  (Test’Result = X + 1);

function Test(X : Integer) return Integer is
begin
    return X + 1;
end Test;

▶ The generated verification condition (omitting side conditions) is

\[ X > 0 \rightarrow X + 1 = X + 1 \]

▶ Does it hold?
Functions Which Return Fixed Expressions

- If a function returns always the same expression E, we can omit any post-condition.
- Then no verification conditions are generated (except for expressing range checks for integers, floats, etc.).
- When using this function in other procedures SPARK Ada adds axioms expressing that this function returns this value.
- Example:
  Specification file:
  ```
  function F(X : Integer) return Integer;
  ```
  Implementation file:
  ```
  function F(X : Integer) return Integer is (X + 1);
  ```
- Whenever F(X) occurs we get axiom
  \[ F(X) = X + 1 \]
Example 3 (Verification Conditions for Functions)

function Test(X : Integer) return Integer
with Pre => (X > 0),
Post => (Test’Result = X + 1);

function Test(X : Integer) return Integer is
begin
if X = 0 then return -1;
else return X + 1;
end if;
end Test;

▶ Is this program correct?
Example 3 (Verification Conditions for Functions)

```haskell
function Test(X : Integer) return Integer
  with Pre => (X > 0),
        Post => (Test’Result = X + 1);
function Test(X : Integer) return Integer is
begin
  if X = 0 then return -1;
  else return X + 1;
end if;
end Test;
```

- The verification conditions are
  \[
  X > 0 \land X = 0 \rightarrow -1 = X + 1 \\
  X > 0 \land \neg(X = 0) \rightarrow X + 1 = X + 1
  \]

- Which ones do hold?
2 (a) Introduction into Ada

2 (b) Architecture of SPARK Ada

2 (c) Language Restrictions in SPARK Ada

2 (d) Data Flow Analysis

2 (e) Information Flow Analysis

2 (f) Verification Conditions (Simple Programs)

2 (g) Verification Conditions (Loops, Procedures and Functions)

2 (h) Example: A railway interlocking system
We will look at the following at an example of verify a small railway interlocking system using SPARK Ada.

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 points in between (but a set of points might form one segment)
  - there are no crossings in between,
  - they are not divided by signals into parts.
Example

- In the following example we have track segments s1 - s6.
- The two branches of the set of points p1 form segment s2.
- The two branches of the set of points p2 form segment s4.
Sets of Points

- A set of points has two positions, normal and reverse.
- In the example the normal position of p1 connects it with s3, the reverse position connects it with s6;
- the normal position of p2 connects it with s3, the reverse position connects it with s6.
Signals

- Signals control the access from one train segment to the next one.
- They are drawn in the direction of use, e.g. Signal sig2 is visible from s1 and controls access to s2.
- In the example sig2, sig7, sig9, control access to the set of points p1, and sig3, sig6, sig10 control access to p2.
- sig1, sig5 control access to s1, s5 respectively, and sig8, sig4 control access to the neighbouring rail yards.
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).
  - The beginning of a train route and its end should be delimited by signals.
    - The first one prevents entering the train route, the second one, delimits access from this train route to the following train routes.
In the example, \((s2,s3), (s2,s6), (s4,s3), (s4,s6), (s1)\) moving left, \((s1)\) moving right, \((s5)\) moving left, \((s5)\) moving right form train routes.
One might have a train routes (s1, s2, s3, s4, s5) (s1, s2, s6, s4, s5), (s5, s4, s3, s2, s1), (s5, s4, s6, s2, s1) for fast trains, which pass 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

- Apart from main signals there are as well advance 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.
- The distant signal will signal whether the next main signal will be red or green.
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 none of the segments used by that train route is locked.
- 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 as part of the lecture examples
  http://www.cs.swan.ac.uk/~csetzer/lectures/critsys/14/SPARK_Ada/examplesAdaSparkAdaCriticalSystems.zip
We will investigate a very simple example of a rail yard.

- We have 3 segments:
  - “Left”
  - “Middle”
  - “Right”
Case Study

We have 8 signals:
“Enter_Left”, “Leave_Left”, “Left_Middle”, “Middle_Left”, “Middle_Right”, “Right_Middle”, “Leave_Right”, “Enter_Right”.

- “Right_Middle” guards access from Segment “Right” to segment “Middle”.
- “Leave_Left” guards access from Segment “Left” to the neighbouring rail yard to the left.
- “Enter_Left” guards access to Segment “Left” from the neighbouring rail yard to the left.
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”.

灸2 (h) Example: A railway interlocking system
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”.
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 8 components, which give the signal state for each of signal: “Enter_Left”, “Leave_Left”, “Left_Middle”, “Middle_Left”, “Middle_Right”, “Right_Middle”, “Leave_Right”, “Enter_Right”.

▶
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”.

![Diagram of a railway interlocking system with states]

2 (h) Example: A railway interlocking system
Modes of Operation

- There are two modes of operation:
  - “Mode_Open”.
    - In this mode we open a route for one train, which is currently standing in one segment, to the next segment.
  - “Mode_Move”.
    - 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.
pragma SPARK_Mode;
with SPARK.Text_IO; use SPARK.Text_IO;
package Simple_Railway is

type One_Signal_State is (Red, Green);

type Mode is (Mode_Open, Mode_Move);
--  Mode_Open = Open a route
--  Mode_Move = Move a train
Data Structure in Ada

```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;
Data Structure in Ada

```ada
type Signal_State_Type is record
   Enter_Left, Leave_Left, Left_Middle, Middle_Left, Middle_Right, Right_Middle, Leave_Right, Enter_Right : 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;
Procedure Init

- We have one procedure which initializes the data structures (the post condition will be discussed later):

```plaintext
procedure Init with
  Global  =>  (Output => (Segment_State, Signal_State)),
  Depends =>  (Signal_State  =>  null,
                Segment_State  =>  null),
  Post    =>  ...;
```
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 (pre- and post-conditions will be discussed later):

```
procedure Open(Route : in Route_Type;
    Success : out Boolean ) with

    Global  => (In_Out => (Segment_State, Signal_State)),

    Depends => (Segment_State  => (Segment_State,Route),
                Success     => (Segment_State,Route),
                Signal_State => (Segment_State,Route, Signal_State)),

    Pre     => ...,
    Post    => ...;
```
Procedure Move

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

```plaintext
procedure Move(Route : in Route_Type;
    Success : out Boolean ) with

Global  => (In_Out => (Segment_State, Signal_State)),
Depends => (Signal_State => (Segment_State,Route, Signal_State),
            Segment_State => (Segment_State,Route, Signal_State),
            Success => (Segment_State,Route, Signal_State)),

Pre   => ...,
Post  => ...;
```
IO Using Input- and Output-Streams

- For dealing with Input/Output (IO) we use a package `SPARK.Text_IO`, which deals with file I/O and standard I/O (via console).
- Spark.Text_IO has internal state variables
  - `Standard_Output` representing all outputs made up to now by the program.
  - `Standard_Input` representing all inputs made up to now, which were not yet used by the program,
Input and Output Streams

- When we make some output (e.g., print some characters on the console), we add some elements to the Standard_Output. Therefore the new Standard_Output depends on the previous Standard_Output and any reasons for making that output.

- When we get some input (here from console), we take some first elements off Standard_Input. Therefore the new Standard_Input depends on the previous Standard_Input and any reasons for requesting that input.

Any variables changed because of some input depend as well on Standard_Input

- Apart from this every input and output has pre-condition Status (Standard_Output) = Success
Therefore the procedure, which prints the state to console, has the following specification:

```plaintext
procedure Print_State with
Global => (In_Out => Standard_Output,
           Input  => (Segment_State, Signal_State)),
Depends => (Standard_Output => (Standard_Output,
                                Segment_State,
                                Signal_State)),
Pre     => Status (Standard_Output) = Success,
Post    => Status (Standard_Output) = Success;
```

There are no other verification conditions, since this procedure is not safety-critical.
Procedure Get_Action

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

```plaintext
procedure Get_Action(Route : out Route_Type;
                      The_Mode : out Mode ) with

Global  =>  (In_Out  =>  (Standard_Input,Standard_Output),
Depends =>  (Standard_Output  =>  (Standard_Output,
                 Standard_Input),
            Standard_Input  =>  Standard_Input,
            Route          =>  Standard_Input,
            The_Mode       =>  Standard_Input),
Pre     =>  Status (Standard_Output) = Success,
Post    =>  Status (Standard_Output) = Success;
```
2 (h) Example: A railway interlocking system

Dependencies in Get_Action

- Note that the fact that the procedures Print_State and Get_Action don’t change Segment_State and Signal_State (they are only output variables), means that they don’t perform any safety critical action.

- In Get_Action
  - **Standard_Output** depends on **Standard_Input**, since, depending on the input, different information is output.
  - **Standard_Input** depends on **Standard_Input**, 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 used.
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”.

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

- "Move" checks for instance in case the route is "Route_Left_Middle"
  - whether segment "Left" is in state "Occupied_Moving_Right"
  - and signal "Left_Middle" is green
  - 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".
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 Signal_State.Left_Middle = Green
    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 ···
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:

```plaintext
if Signal_State.Middle_Left = Green
then (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:
  (\textbf{if} (Segment\_State'\textbf{Old}.Left = Occupied\_Moving\_Left
  \textbf{or}
  Segment\_State'\textbf{Old}.Left = Occupied\_Moving\_Right
  \textbf{or}
  Segment\_State'\textbf{Old}.Left = Occupied\_Standing)
  \textbf{then} (Segment\_State.Left = Occupied\_Moving\_Left
  \textbf{or}
  Segment\_State.Left = Occupied\_Moving\_Right
  \textbf{or}
  Segment\_State.Left = Occupied\_Standing))

- We need corresponding formulae for Middle and Left as well.
3. In procedure “Move”, 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:

  (if Segment_State’\textbf{Old}.Left = Occupied_Moving_Right
   then
   (Segment_State.Left = Occupied_Moving_Right
    or
    (Route = Route_\textbf{Left}_Middle
     and Segment_State’\textbf{Old}.Middle =
      Reserved_Moving_From_Left
     and Segment_State.Left = Free
     and Segment_State.Middle = Occupied_Standing)))

  and

  (if Segment_State’\textbf{Old}.Left = Occupied_Standing
   then
   Segment_State.Left = Occupied_Standing)
Correctness Conditions

and
(if Segment_State'\text{Old}.Left = Occupied\_Moving\_Left
then
(Segment\_State.Left = Occupied\_Moving\_Left
or
(Route = Route\_Leave\_Left
and Segment\_State.Left = Free))))

- Corresponding formulae for the other two segments.
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

- In pre-2014 SPARK-Ada, data- and information-flow analysis succeeded, verification conditions were generated, but not proved.
- 2014-SPARK-Ada uses powerful automated theorem provers and was able to in addition to data- and information-flow analysis generate and prove all verification conditions.
  - Some small typos in the unchecked program used with the pre-2014 version were found.
Future Work

- In the approach above each signal, segment, route had it's individual program code and verification conditions.
- This may lead to errors since each entity needs to be checked individually.
  - Note that the verification conditions are not verified, only program code is verified.
- This approach was taken because of limitations of pre-2014 SPARK-Ada for dealing with quantified formulas.
- It seems that SPARK-Ada 2014’s theorem proving capabilities are much more powerful.
- It may be that a more generic approach with generic verification conditions is now possible.
- This needs to be explored.
The restrictions imposed by SPARK-Ada on the Ada language are very sensible, and help avoid errors.

Flow and information analysis finds many programming errors without requiring much work by the programmer.

In SPARK-Ada 2014 a huge progress has been made and lots of verification conditions can now be shown automatically.
Evaluation

- If automated theorem proving fails one can use an interactive theorem prover such as Coq or Isabelle/HOL.
  - One problem is that the machine generated conditions are very long, which makes it infeasible for interactive theorem provers.
  - A second 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.
    - the verification conditions are well connected to the program code.

- In real world applications, I assume proving of the verification conditions interactively is rarely carried out.
Testing of Proof Conditions

- If proving of verification conditions fails, one can check them using 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

- Automated theorem proving has made big progress recently and the verification of programs using formal methods is now possible and feasible.
  - There are still problems to be solved, eg. how to deal with concurrency or the full complexity of object-orientation.
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 product 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.