CodePeer

About This Course

Styles

- *This* is a definition
- `this/is/a.path`
- `code is highlighted`
- `commands are emphasised --like-this`

```
console outputs
are shown like that
```

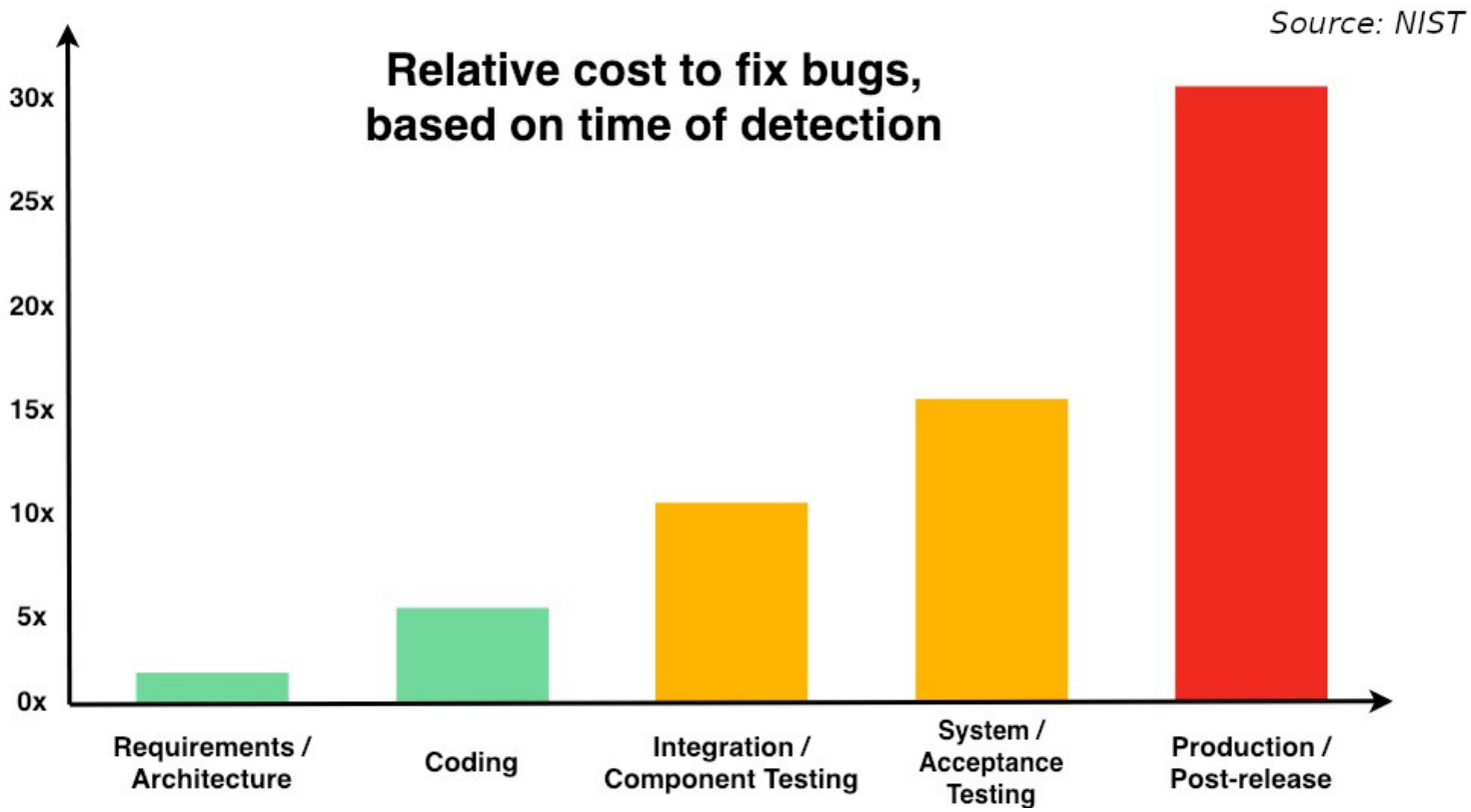
Advanced Static Analysis

What is Static Analysis?

- **Symbolic** interpretation of **source code**
  - Find what could go wrong
  - No execution
- **Formally** verifying **high level** or **abstract** properties
  - Strong guarantees
- *May* be exhaustive
  - All possible errors are reported
  - No false negatives; there may be false positives
  - If the analyzer does not report a problem, there is no problem

Why Static Analysis Saves Money

- Costs shift
- From later, **expensive** phases
- To earlier, **cheaper** phases



Why Use CodePeer?

- Efficient, potentially exhaustive code reviewer
  - Identifies run-time errors with a **level of certainty**
    - E.g. buffer overflows, division by zero
  - Flags legal but **suspect** code
    - Typically logic errors
- Detailed subprograms analysis
- Can analyze existing code bases
  - Detect and remove **latent bugs**
  - Legacy code
  - Code from external sources

Detailed Subprogram Analysis

- **Explicit** specification
  - Written in the code
  - Types
  - Contracts
  - Assertions
  - etc...
- **Implicit** specification
  - Assumptions by CodePeer
  - *Deduced preconditions*

CodePeer Overview

CodePeer In A Nutshell (1/2)

- CodePeer is a static analysis tool
  - Provides feedback **before** execution and test
  - Provides *as-built documentation* for code reviews
- Helps identify and eliminate **vulnerabilities and bugs** early
- Modular
  - Analyze entire project or a single file
  - Configure strictiness level
- Review features
  - Filtering messages by category, severity, package...
  - Comparative analysis between runs
  - Shareable reviews database

CodePeer In A Nutshell (2/2)

- Large Ada support
  - Usable with Ada 83, 95, 2005, 2012
  - No vendor lock-in, supports GNAT, Apex, GHS, ObjectAda, VADS
- Bundled with a Coding Standards Checker and a Metrics Tool
  - GNATcheck and GNATmetric
- Detects runtime and logic errors exhaustively
  - Initialization errors, run-time errors and assertion failures (16 rules)
  - Race condition errors: unprotected access to globals (3 rules)
- Warns on dead or suspicious code (21 rules)

CodePeer Integration

- Output: textual, XML, CSV, HTML
- Command-line tool (uses GNAT project files)
- Interactive use in GNAT Studio and GNATbench IDEs
- Integration with Jenkins (continuous builder)
- Integration with SonarQube (continuous inspection of code quality)

INFER Integration

- INFER for Ada on top of main analysis
- Based on Facebook's INFER engine
- Adds **lightweight** checks
- Disable with `--no-infer` switch

Typical Users And Use Cases

- Developers, during code-writing
  - **Fix** (local) problems before integration
- Reviewers
  - **Annotate** code with analysis of potential problems
  - **Analyse** specific CWE issues
- Project managers and quality engineers
  - **Track** reported vulnerabilities regularly
  - **Identify** new issues quickly
- Software auditors
  - **Identify** overall vulnerabilities or hot spots
  - **Verify** compliance to quality standards

Getting Started

Command Line Interface (1/2)

```
codepeer -P <project> [-level <level>] ...
```

`-P <gpr project-file>` Note: All files from the project (including subprojects) will be analyzed.

Tip: if missing a project file, use the `--simple-project` switch

`-level 0|1|2|3|4|min|max` Specify the level of analysis performed:

- 0/min (default): fast and light checkers
- 1: fast and per subprogram analysis
- 2: more accurate/slower, automatic partitioning per set of units
- 3: more accurate and much slower
- 4/max: global (exhaustive) analysis, no partitioning

Warning: Level 4 may exceed memory capacity or take a very long time

Command Line Interface (2/2)

`codepeer` ... `[-output-msg[-only]] [-html[-only]]`

`-output-msg[-only] [-output-msg switches]` If specified, CodePeer will output its results, in various formats.

If `-output-msg` is given, CodePeer will perform a new analysis, and output its results.

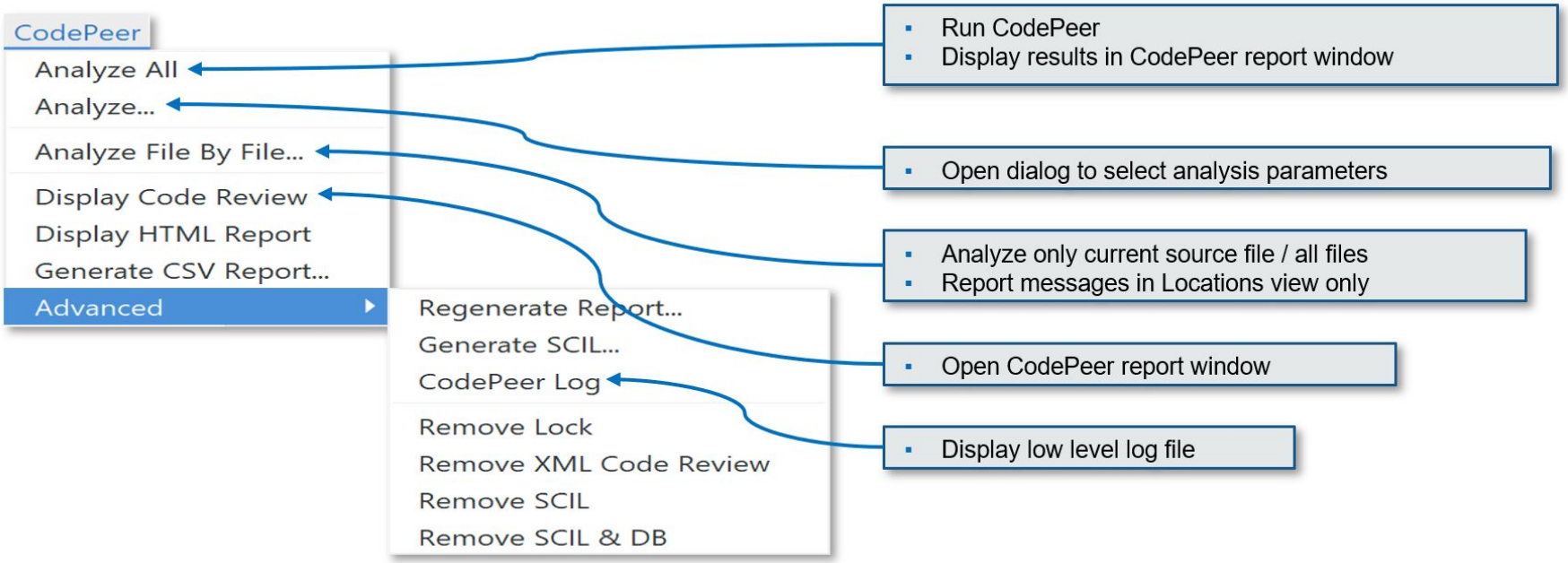
If `-output-msg-only` is specified, no new analysis is performed, and the results from the previous run (of the same level) will be emitted.

You can control this output by adding switches.

e.g. `-output-msg -csv -out report.csv` to generate a CSV file

`-html, -html-only` Generate HTML output. If `-html-only`, do not run any analysis but use the previous run.

Running CodePeer in GNAT Studio



Project File Set Up

Let's explore sections 1.4, 1.5 and 1.6 of the User's Guide

- Link: Basic Project File Setup
- Link: Project File Setup
- Link: Advanced Project File Setup

CodePeer Levels Depth and Constraints

- The **higher** the level the **deeper** and **costlier** the analysis

Level	*Description*	*Code size*	*False positives*
0	Infer only (default)	No limits	Lowest
1	Subprograms	No limits	Few
2	Groups of units	No limits	Some
3	Semi-global	$<$ 1 million SLOC	High
	Automatic partitioning	CC $<$ 40	
4	Global and **exhaustive**	$<$ 200 KSLOC	Highest
	Flag all issues	CC $<$ 20	

- *SLOC* : Source lines of code
- *CC* : Cyclomatic Complexity

CodePeer Levels Use Case

- The levels adapt to various **workflows** and **users**
- The **lower** the level the **more frequently** it should be run

Level	*Condition*	*Workflow Step*	*Goal*
0	None	Initial static analysis	Quick feedback
1	Project set-up	After each commit	Sanity check
2	Level 1 results clean	Integration, CI	Regular check
3	Medium code base Server run	Integration, Nightly	Manual review Baseline
4	Small code base Server run	Before production	Exhaustive review

"No False Positive" Mode

- `-level 0` or `-messages min`
- Suppresses messages **most likely** to be false positives
- Allows programmers to **focus** initial work on likely problems
- Can be combined with **any level** of analysis
- `-messages min` is default for levels 0, 1, and 2

Running CodePeer regularly

- Historical database (SQLite) stores all results **per level**
  - Can be stored in Configuration Management
- *Baseline* run
  - **Previous** run each new run is compared to
  - Differences of **messages** in CodePeer report
  - Default: first run
  - `-baseline` to change it
- Typical use
  - **Nightly** `-baseline` run on servers
  - **Daily** development compares to baseline
- `-cutoff` overrides it for a **single** run
- Compare between two arbitrary runs with `-cutoff` and `-current`

CodePeer Tutorial

Instructions

- Walk through the steps of the CodePeer tutorial

CodePeer Checks

Messages Categories

- **Run-Time Checks**
  - Errors that will raise built-in exceptions at runtime
  - Or fail silently with `-gnatp`
- **User Checks**
  - Errors that will raise user exceptions at runtime
  - Or fail silently with `-gnatp`
- **Validity Checks**
  - Mishandled object scope and value
- **Warnings**
  - Questionable code that seems to have logic flaws
  - Hints at logical errors
- **Race Conditions**
  - Code unsafe due to multi-tasking

Run-Time Checks

Run-Time Check Messages

Message	*Definition*
`divide by zero`	The second operand could be zero On a division, `mod` or `rem` operation
`range check`	A discrete could reach a value out of its `range`
`overflow check`	An operation could overflow its numeric type Note: Depends on the **'Base** representation
`array index check`	Array index could be out of bounds
`access check`	A `null` access could be dereferenced
`aliasing check`	A subprogram call could cause an aliasing error eg. passing a single reference as two parameters
`tag check`	A dynamic `'Class` or `'Tag` check could fail
`validity`	An uninitialized or invalid object could be read
`discriminant check`	The wrong variant could be used eg. copy with the wrong discriminant
`precondition`	A subprogram call could violate its deduced precondition

Divide By Zero

- The second operand of a divide, **mod** or **rem** operation could be zero
- Runtime Constraint_Error

```
1  procedure Div is
2     type Int is range 0 .. 2**32 - 1;
3     A : Int := Int'Last;
4     X : Integer;
5  begin
6     for I in Int range 0 .. 2 loop
7        X := Integer (A / I); -- division by zero when I=0
8     end loop;
9  end Div;
```

```
high: divide by zero fails here: requires I /= 0
```

Range Check

- Calculation may generate a value outside the **range** of an Ada type or subtype
- Will generate a Constraint_Error

```
1  subtype Constrained_Integer is Integer range 1 .. 2;
2  A : Integer;
3
4  procedure Proc_1 (I : in Constrained_Integer) is
5  begin
6     A := I + 1;
7  end Proc_1;
8  ...
9  A := 0;
10 Proc_1 (I => A);   --  A is out-of-range of parameter I
```

```
high: range check fails here: requires A in 1..2
```

Overflow Check

- Calculation may overflow the bounds of a numeric type.
- Depends on the size of the underlying (base) type
- Will generate a `Constraint_Error`

```
1  is
2     Attempt_Count : Integer := Integer'Last;
3  begin
4     -- Forgot to reset Attempt_Count to 0
5     loop
6        Put ("Enter password to delete system disk");
7        if Get_Correct_Pw then
8           Allow_Access;
9        else
10          Attempt_Count := Attempt_Count + 1;
```

```
high: overflow check fails here: requires Attempt_Count
/= Integer_32'Last
high: overflow check fails here: requires Attempt_Count
in Integer_32'First-1..Integer_32'Last-1
```

Array Index Check

- Index value could be outside the array bounds
- Also known as **buffer overflow**.
- Will generate a `Constraint_Error`

```
1  procedure Buffer_Overflow is
2     type Int_Array is array (0 .. 2) of Integer;
3     X, Y : Int_Array;
4  begin
5     for I in X'Range loop
6        X (I) := I + 1;
7     end loop;
8
9     for I in X'Range loop
10       Y (X (I)) := I;  -- Bad when I = 2, since X (I) = 3
11    end loop;
12 end Buffer_Overflow;
```

```
high: array index check fails here: requires (X (I)) in
0..2
```

Access Check

- Attempting to dereference a reference that could be `null`
- Will generate an `Access_Error`

```
1  procedure Null_Deref is
2     type Int_Access is access Integer;
3     X : Int_Access;
4  begin
5     if X = null then
6        X.all := 1;  -- null dereference
7     end if;
8  end Null_Deref;
```

```
high: access check fails here
```

Aliasing Check

- Some parameters could be passed as **reference**
- Deduced preconditions:
  - Do not **reference** another parameter
  - Do not **match** the address of a global object

```
1  procedure In_Out (A : Int_Array; B : out Int_Array) is
2  begin
3     B (1) := A (1) + 1;
4     ...
5     B (1) := A (1) + 2;
6  end In_Out;
7  ...
8  In_Out (A, A); -- Aliasing!
```

```
high: precondition (aliasing check) failure on call to
alias.in_out: requires B /= A
```

Tag Check

A tag check operation on a **tagged** object might raise a Constraint_Error

```
1  is
2     type T1 is tagged null record;
3     type T2 is new T1 with null record;
4
5     procedure Call (X1 : T1'Class) is
6     begin
7        An_Operation (T2'Class (X1));
8     end Call;
9
10    X1 : T1;
11    X2 : T2;
12 begin
13    Call (X1); -- not OK, Call requires T2'Class
```

```
high: precondition (tag check) failure on call to
tag.call: requires X1'Tag in {tag.pkg.t2}
```

Validity

```
procedure Uninit is
   A : Integer;
   B : Integer;
begin
   A := B;  --  we are reading B which is uninitialized!
end Uninit;
```

```
high: validity check: B is uninitialized here
```

Discriminant Check

A field for the wrong variant/discriminant is accessed

```
1  type T (B : Boolean := True) is record
2     case B is
3        when True =>
4           J : Integer;
5        when False =>
6           F : Float;
7     end case;
8  end record;
9
10 X : T (B => True);
11
12 function Create (F : Float) return T is
13   (False, F);
14 ...
15 X := Create (6.0);  -- discriminant check failure
```

```
high: discriminant check fails here: requires (Create
(6.0).b = True)
```

Precondition

- Subprogram call could violate preconditions, either
  - Where the error may occur
  - Where a caller passes in a value causing the error
- Need to check generated preconditions
- GNAT Studio or `-show-backtraces` to analyze checks

```
1  function Call (X : Integer) return Integer is
2  begin
3     if X < 0 then
4        return -1;
5     end if;
6  end Call;
7  ...
8  for I in -5 .. 5 loop
9     X := X + Call (I);
10 end loop;
```

```
high: precondition (conditional check) failure on call
to precondition.call: requires X < 0
```

Quiz

- Which check will be raised with the following?

```
function Before_First return Integer is
begin
   return Integer'First - 1;
end Dec;
```

A. Precondition check
B. Range check
C. Overflow check
D. Underflow check

Quiz

- Which check will be raised with the following?

```
function Before_First return Integer is
begin
   return Integer'First - 1;
end Dec;
```

A. Precondition check
B. Range check
C. ***Overflow check***
D. Underflow check

The value is out of representation range so the operation will fail, that is an overflow, not a range check.

Difference between the two: Overflow is checked for intermediate operations, range is then checked at affectation (parameter passing, conversion...).

Quiz

- Which check will be raised with the following?

```
type Ptr_T is access Natural;
type Idx_T is range 0 .. 10;
type Arr_T is array (Idx_T) of Ptr_T;

procedure Update
  (A : in out Arr_T) is
begin
   for J in Idx_T loop
      declare
         K : constant Idx_T := J - 1;
      begin
         A (K).all := (if A (K) /= null then A (K).all - 1 else 0);
      end;
   end loop;
end Update;
```

A. Array index check
B. Range check
C. Overflow check
D. Access check

Quiz

- Which check will be raised with the following?

```
type Ptr_T is access Natural;
type Idx_T is range 0 .. 10;
type Arr_T is array (Idx_T) of Ptr_T;

procedure Update
  (A : in out Arr_T) is
begin
   for J in Idx_T loop
      declare
         K : constant Idx_T := J - 1;
      begin
         A (K).all := (if A (K) /= null then A (K).all - 1 else 0);
      end;
   end loop;
end Update;
```

A. Array index check
B. ***Range check***
C. Overflow check
D. Access check

When `J = 0`, the declaration of `K` will raise a `Constraint_Error`

If any `A (K).all = 0`, a second range check is raised.

User Checks

User Check Messages

Message	*Description*
`assertion`	A user assertion could fail eg. `pragma Assert`
`conditional check`	An `exception` could be raised conditionally
`raise exception`	An `exception` is raised on a reachable path Same as *conditional check*, but unconditionally
`user precondition`	Potential violation of a specified precondition As a `Pre` aspect or as a `pragma Precondition`
`postcondition`	Potential violation of a specified postcondition As a `Post` aspect or as a `pragma Postcondition`

Assertion

A user assertion (using e.g. `pragma Assert`) could fail

```
1  procedure Assert is
2
3     function And_Or (A, B : Boolean) return Boolean is
4     begin
5        return False;
6     end And_Or;
7
8  begin
9     pragma Assert (And_Or (True, True));
10 end Assert;
```

```
high: assertion fails here: requires (and_or'Result) /=
false
```

Conditional Check

An exception could be raised **conditionally** in user code

```
1  if Wrong_Password then
2     Attempt_Count := Attempt_Count + 1;
3
4     if Attempt_Count > 3 then
5        Put_Line ("max password count reached");
6        raise Program_Error;
7     end if;
8  end if;
```

high: conditional check raises exception here: requires Attempt_Count <= 3

Raise Exception

An exception is raised **unconditionally** on a **reachable** path.

```
1  procedure Raise_Exc is
2     X : Integer := raise Program_Error;
3  begin
4     null;
5  end Raise_Exc;
```

```
low: raise exception unconditional raise
```

User Precondition

A call might violate a subprogram's specified precondition.

```
1  procedure Pre is
2     function "**" (Left, Right : Float) return Float with
3        Import,
4        Pre => Left /= 0.0;
5
6     A : Float := 1.0;
7  begin
8     A := (A - 1.0)**2.0;
9  end Pre;
```

```
high: precondition (user precondition) failure on call
to pre."**": requires Left /= 0.0
```

Postcondition

The subprogram's body may violate its specified postcondition.

```
1  type Stress_Level is (None, Under_Stress, Destructive);
2
3  function Reduce (Stress : Stress_Level)
4    return Stress_Level with
5     Pre  => (Stress /= None),
6     Post => (Stress /= Destructive)
7     is (Stress_Level'Val (Stress_Level'Pos (Stress) + 1));
8     --                                                ^
9     --                                                Typo!
10 ...
11 Reduce (My_Component_Stress);
```

```
high: postcondition failure on call to post.reduce:
requires Stress /= Destructive
```

Quiz

- Which user check will be raised with the following?

```
procedure Raise_Exc (X : Integer) is
begin
   if X > 0 or X < 0 then
      raise Program_Error;
   else
      pragma Assert (X >= 0);
   end if;
end Raise_Exc;
```

A. Conditional check
B. Assertion
C. Raise Exception
D. User precondition

Quiz

- Which user check will be raised with the following?

```
procedure Raise_Exc (X : Integer) is
begin
   if X > 0 or X < 0 then
      raise Program_Error;
   else
      pragma Assert (X >= 0);
   end if;
end Raise_Exc;
```

A. ***Conditional check***
B. Assertion
C. Raise Exception
D. User precondition

The exception is raised on `X /= 0`, it is **conditionally** reachable.

In other cases, `X = 0` so the assertion always holds.

Uninitialized and Invalid Variables

Uninitialized and Invalid Variables Messages

Message	*Description*
`validity check`	An uninitialized or invalid value could be read

Validity Check

The code may be reading an uninitialized or invalid value

```
1  procedure Uninit is
2     A : Integer;
3     B : Integer;
4  begin
5     A := B;  --  we are reading B which is uninitialized!
6  end Uninit;
```

```
high: validity check: B is uninitialized here
```

Warnings

Warning Messages (1/3)

Message	*Description*
`dead code`	Also called *unreachable code*. Assumed all code should be reachable
`test always false`	Code always evaluating to `False`
`test always true`	Code always evaluating to `True`
`test predetermined`	Choice evaluating to a constant value For eg. `case` statements
`condition predetermined`	Constant RHS or LHS in a conditional
`loop does not complete normally`	Loop `exit` condition is always `False`
`unused assignment`	Redundant assignment
`unused assignment to global`	Redundant global object assignment
`unused out parameter`	Actual parameter of a call is ignored Either never used or overwritten

- **RHS** : Right-Hand-Side of a binary operation
- **LHS** : Left-Hand-Side of a binary operation

Warning Messages (2/3)

Message	*Description*
`useless reassignment`	Assignment does not modify the object
`suspicious precondition`	Precondition seems to have a logic flaw eg. possible set of values is not contiguous
`suspicious input`	`out` parameter read before assignment should be `in out`
`unread parameter`	`in out` parameter is never read should be `out`
`unassigned parameter`	`in out` parameter is never assigned should be `in`
`suspicious constant operation`	Constant result from variable operands May hint at a typo, or missing operation
`subp never returns`	Subprogram will never terminate
`subp always fails`	Subprogram will always terminate in error

Warning Messages - INFER (3/3)

Message	*Description*
`same operands`	Binary operator has the same argument twice
`same logic`	Same argument appears twice in a boolean expres
`duplicate branches`	Duplicate code in 'if' or 'case' branches
`test duplication`	An expression is tested multiple times in an `if ... elsif ... else`

Dead Code

- Also called **unreachable code**.
- All code is expected to be reachable

```
1  procedure Dead_Code (X : out Integer) is
2     I : Integer := 10;
3  begin
4     if I < 4 then
5        X := 0;
6     elsif I >= 8 then
7        X := 0;
8     end if;
9  end Dead_Code;
```

```
medium warning: dead code because I = 10
```

Test Always False

Redundant conditionals, always `False`

```
1  procedure Dead_Code (X : out Integer) is
2     I : Integer := 10;
3  begin
4     if I < 4 then
5        X := 0;
6     end if;
7  end Dead_Code;
```

```
low warning: test always false because I = 10
```

Test Always True

Redundant conditionals, always True

```
1  procedure Dead_Code (X : out Integer) is
2     I : Integer := 10;
3  begin
4     if I >= 8 then
5        X := 0;
6     end if;
7  end Dead_Code;
```

```
medium warning: test always true because I = 10
```

Test Predetermined

- Similar to `test always true` and `test always false`
  - When choice is not binary
  - eg. `case` statement

```
1  procedure Predetermined is
2     I : Integer := 0;
3  begin
4     case I is
5        when 0 =>
6           null;
7        when 1 =>
8           null;
9        when others =>
10          null;
11    end case;
12 end Predetermined;
```

```
low warning: test predetermined because I = 0
```

Condition Predetermined

- **Redundant** condition in a boolean operation
- RHS operand is **constant** in this context

```
1  if V /= A or else V /= B then
2     --       ^^^^^^^
3     --       V = A, so V /= B
4     raise Program_Error;
5  end if;
```

```
medium warning: condition predetermined because (V /= B)
is always true
```

Loop Does Not Complete Normally

- The loop will never complete its **exit condition**
- Causes can be
  - Exit condition is always `False`
  - An exception is raised
  - The exit condition code is dead

```
1  procedure Loops is
2     Buf : String := "The" & ASCII.NUL;
3     Bp  : Natural;
4  begin
5     Buf (4) := 'a';    -- Eliminates null terminator
6     Bp      := Buf'First;
7
8     loop
9        Bp := Bp + 1;
10       exit when Buf (Bp - 1) = ASCII.NUL; -- Condition never reached
11    end loop;
12 end Loops;
```

```
medium warning: loop does not complete normally
```

Unused Assignment

- Object is assigned a value that is never read
- Unintentional loss of result or unexpected control flow
- Object with the following names won't be checked:
  - `ignore`, `unused`, `discard`, `dummy`, `tmp`, `temp`
  - Tuned via the `MessagePatterns.xml` file if needed.
- `pragma Unreferenced` also ignored

```
1  I := Integer'Value (Get_Line);
2  I := Integer'Value (Get_Line);
```

```
medium warning: unused assignment into I
```

Unused Assignment To Global

- Global variable assigned more than once between reads
- Note: the redundant assignment may occur deep in the **call tree**

```
1  procedure Proc1 is
2  begin
3     G := 123;
4  end Proc1;
5
6  procedure Proc is
7  begin
8     Proc1;
9     G := 456;  -- override effect of calling Proc1
10 end Proc;
```

```
low warning: unused assignment to global G in
unused_global.p.proc1
```

Unused Out Parameter

- Actual `out` parameter of a call is ignored
  - either never used
  - or overwritten

```
1  procedure Search (Success : out Boolean);
2  ...
3  procedure Search is
4     Ret_Val : Boolean;
5  begin
6     Search (Ret_Val);
7  end Search;
```

```
medium warning: unused out parameter Ret_Val
```

Useless Reassignment

- Assignments do not modify the value stored in the assigned object

```
1  procedure Self_Assign (A : in out Integer) is
2     B : Integer;
3  begin
4     B := A;
5     A := B;
6  end Self_Assign;
```

```
medium warning: useless reassignment of A
```

Suspicious Precondition

- Set of allowed inputs is **not contiguous**
  - some values **in-between** allowed inputs can cause **runtime errors**
- Certain cases may be missing from the user's precondition
- May be a **false-positive** depending on the algorithm

```
1  if S.Last = S.Arr'Last then
2     raise Overflow;
3  end if;
4  --  Typo: Should be S.Last + 1
5  S.Last           := S.Last - 1;
6  --  Error when S.Last = S.Arr'First - 1
7  S.Arr (S.Last) := V;
```

```
medium warning: suspicious precondition for S.Last: not
a contiguous range of values
```

Suspicious Input

- `out` parameter read before assignment
- Should have been an `in out`
- Ada standard allows it
  - but it is a bug most of the time

```
1 procedure Take_In_Out (R : in out T);
2 ...
3 procedure Take_Out (R : out T; B : Boolean) is
4 begin
5    Take_In_Out (R);  -- R is 'out' but used as 'in out'
6 end Take_Out;
```

```
medium warning: suspicious input R.I: depends on input
value of out-parameter
```

Unread Parameter

- `in out` parameter is not read
  - but is assigned on **all** paths
  - Could be declared `out`

```
1  procedure Unread (X : in out Integer) is
2  begin
3     X := 0;  -- X is assigned but never read
4  end Unread;
```

```
medium warning: unread parameter X: could have mode out
```

Unassigned Parameter

- `in out` parameter is never assigned
  - Could be declared `in`

```
1  procedure Unassigned
2    (X : in out Integer; Y : out Integer) is
3  begin
4     Y := X;  -- X is read but never assigned
5  end Unassigned;
```

```
medium warning: unassigned parameter X: could have mode
in
```

Suspicious Constant Operation

- Constant value calculated from **non-constant operands**
- Hint that there is a **coding mistake**
  - either a **typo**, using the **wrong variable**
  - or an operation that is **missing**
    - eg `Float` conversion before division

```
1  type T is new Natural range 0 .. 14;
2
3  function Incorrect (X : T) return T is
4  begin
5     return X / (T'Last + 1);
6  end Incorrect;
```

```
medium warning: suspicious constant operation X/15
always evaluates to 0
```

Subp Never Returns

- Subprogram will **never** return
  - presumably **infinite loop**
- Typically, **another message** in the body can explain why
  - eg. `test always false`

```
1  procedure Infinite_Loop is
2     X : Integer := 33;
3  begin
4     loop
5        X := X + 1;
6     end loop;
7  end Infinite_Loop;
```

```
medium warning: subp never returns: infinite_loop
```

Subp Always Fails

- A run-time problem could occur on **every** execution
- Typically, **another message** in the body can explain why

```
1  procedure P is
2     X : Integer := raise Program_Error;
3  begin
4     null;
5  end P;
```

```
high warning: subp always fails: p fails for all
possible inputs
```

Same Operands

- The two operands of a binary operation are syntactically equivalent
- The resulting expression will always yield the same value

```
1  function Same_Op (X : Natural) return Integer is
2  begin
3     --  Copy/paste error? Always return 1
4     return (X + 1) / (X + 1);
5  end Same_Op;
```

```
medium warning: same operands (Infer): operands of '/'
are identical
```

Same Logic

- The same sub-expression occurs twice in a boolean expression
- The entire expression can be simplified, or always return the same value

```
1 function Same_Logic (A, B : Boolean) return Boolean is
2 begin
3    return A or else B or else A;
4 end Same_Logic;
```

```
medium warning: same operands (Infer): 'A' duplicated at
line 3
```

Test duplication

- The same expression is tested twice in successive `if ... elsif ... elsif ...`
- Usually indicates a copy-paste error

```
1  procedure Same_Test (Str : String) is
2     A : constant String := "toto";
3     B : constant String := "titi";
4  begin
5     if Str = A then
6        Ada.Text_IO.Put_Line("Hello, tata!");
7     elsif Str = B then
8        Ada.Text_IO.Put_Line("Hello, titi!");
9     elsif Str = A then
10       Ada.Text_IO.Put_Line("Hello, toto!");
11    else
12       Ada.Text_IO.Put_Line("Hello, world!");
13    end if;
14 end Same_Test;
```

```
medium warning: same test (Infer): test 'Str = A'
duplicated at line 9
```

Duplicate branches

- Branches are duplicated in `if` or `case`
- Should be refactored, or results from incorrect copy-paste

```
1  function Dup (X : Integer) return Integer is
2  begin
3     if X > 0 then
4        declare
5           A : Integer := X;
6           B : Integer := A + 1;
7        begin
8           return B;
9        end;
10    else
11       declare
12          A : Integer := X;
13          B : Integer := A + 1;
14       begin
15          return B;
16       end;
17    end if;
18 end Dup;
```

```
infer.adb:4:10: medium warning: duplicate branches
(Infer): code duplicated at line 11
```

Quiz

- Which warnings will be raised with the following?

```
function F (A : Integer; B : Integer) return Integer is
begin
    if A > B then
       return 0;
    elif A < B + 1 then
       return 1;
    elif A /= B then
       return 2;
    end if;
end F;
```

A. Dead Code
B. Condition Predetermined
C. Test Always False
D. Test Always True

Quiz

- Which warnings will be raised with the following?

```
function F (A : Integer; B : Integer) return Integer is
begin
    if A > B then
       return 0;
    elif A < B + 1 then
       return 1;
    elif A /= B then
       return 2;
    end if;
end F;
```

A. ***Dead Code***
B. Condition Predetermined
C. Test Always False
D. Test Always True

The last elsif can never be reached.

Race Conditions

Race Condition Messages

Message	*Description*
`unprotected access`	Shared object access without lock
`unprotected shared access`	Object is referenced is multiple tasks And accessed without a lock
`mismatch protected access`	Mismatch in locks used Checked for all shared objects access eg. task1 uses lock1, task2 uses lock2

Race Condition Examples

```
1  procedure Increment is
2  begin
3     Mutex_Acquire;
4     if Counter = Natural'Last then
5        Counter := Natural'First;
6     else
7        Counter := Counter + 1;
8     end if;
9     Mutex_Release;
10 end Increment;
11
12 procedure Reset is
13 begin
14    Counter := 0; -- lock missing
15 end Reset;
```

```
medium warning: mismatched protected access of shared
object Counter via race.increment
medium warning: unprotected access of Counter via
race.reset
```

Automatically Generated Annotations

Generated Annotations

- CodePeer generates **annotations** on the code
- Not errors
- Express **properties** and **assumptions** on the code
- Can be reviewed
  - But not necessarily
  - Can help spot **inconsistencies**
- Can help understand and **debug** messages

Annotations Categories

Annotation	*Description*
`precondition`	Requirements imposed on the subprogram's inputs
`postcondition`	Presumption on the outputs of a subprogram
`presumption`	Presumption on the result of an **external** subprogram
`unanalyzed call`	External calls to unanalyzed subprograms
`global inputs`	Global variables **referenced** by each subprogram
`global outputs`	Global variables **modified** by each subprogram
`new objects`	Unreclaimed heap-allocated object

Precondition

- Requirements imposed on the subprogram inputs
  - eg. a certain parameter to be non-null
- Checked at every call site
- A message is given for any precondition that a caller **might** violate.
  - Includes the **checks involved** in the requirements

```
procedure Assign (X : out Integer; Y : in Integer) is
begin
   X := Y + 1;
end Assign;
-- assign.adb:1: (pre)- assign:(overflow check [CWE 190])
-- Y /= 2_147_483_647
```

Postcondition

- Inferences about the outputs of a subprogram

```
2  -- assign.adb:1: (post)- assign:X /= -2_147_483_648
3  -- assign.adb:1: (post)- assign:X = Y + 1
```

Presumption

- Presumption about the results of an **external** subprogram
  - Code is unavailable
  - Code is in a separate partition
- Separate presumptions for each call site

```
<subprogram-name>@<line-number-of-the-call>
```

- Generally not used to determine preconditions of the calling routine
  - but they might influence postconditions of the calling routine.

```
procedure Above_Call_Unknown (X : out Integer) is
begin
  Call_Unknown (X);
  pragma Assert (X /= 10);
end Above_Call_Unknown;
-- (presumption)- above_call_unknown:unknown.X@4 /= 10
```

Unanalyzed Call

- External calls to unanalyzed subprograms
  - Participate in the determination of presumptions
- These annotations include **all** unanalyzed calls
  - **Direct** calls
  - Calls in the **call graph** subtree
    - **If** they have an influence on the current subprograms

```
-- above_call_unknown.adb:2: (unanalyzed)-
--     above_call_unknown:call on unknown
```

Global Inputs/Outputs

- Global variables referenced by each subprogram
- Only includes **enclosing** objects
  - Not e.g. specific components
- For accesses, only the **access object** is listed
  - Dereference to accesses **may** be implied by the access object listed

```
procedure Double_Pointer_Assign (X, Y : in Ptr) is
begin
   X.all := 1;
   Y.all := 2;
end Double_Pointer_Assign;
-- call_double_pointer_assign.adb:4: (global outputs)-
--     call_double_pointer_assign.call:X, Y
```

New Objects

- Unreclaimed heap-allocated objects
  - **Created** by a subprogram
  - **Not reclaimed** during the execution of the subprogram itself
- New objects that are accessible **after** return from the subprogram

```
procedure Create (X : out Ptr) is
begin
   X := new Integer;
end;
-- alloc.adb:2: (post)- alloc.create:X =
--     new integer(in alloc.create)#1'Address
-- alloc.adb:2: (post)- alloc.create:
--     new integer(in alloc.create)#1.<num objects> = 1
```

External Tools Integration

GNAT Warnings

- GNAT warnings can be generated by CodePeer

  `--gnat-warnings=xxx` *(uses -gnatwxxx)*

- Messages are stored in the database
  - Displayed and filtered as any other message
- Manual justification
  - Can be stored in the database
  - Done via `pragma Warnings` instead of `pragma Annotate`

GNATcheck messages

- GNATcheck messages can be generated by CodePeer

  `--gnatcheck`
- Uses the GNATcheck rules file
  - defined in your project file in `package Check`
- Messages are stored in the database
  - Displayed and filtered as any other message
- Manual justification
  - Can be stored in the database
  - Done via `pragma Annotate (GNATcheck, ...)`

Finding the Right Settings

System Requirements

- Fast 64bits machine with multiple cores and memory
- **Server** → 24 to 48 cores with at least 2GB per core (48 to 96GB)
- **Local desktop** → 4 to 8 cores, with at least 8 to 16GB
- **Avoid slow filesystems** → networks drives (NFS, SMB), configuration management filesystems (e.g. ClearCase dynamic views).
  - If not possible, at least generate output file in a local disk via the *Output_Directory* and *Database_Directory* project attributes.
- **Global analysis (-level max)** → At least 12GB + 1GB per 10K SLOC, e.g. At least 32GB for 200K SLOC.

Analyze Messages (1/4)

- Start with default (level 0)
- Check number of **false positives**
- Check number of **interesting** message
- Check **duration** of analysis
- If these conditions are OK
  - Increase level (eg. level 1) and iterate

```
project My_Project is
   ...
   package CodePeer is
      for Switches use ("-level", "1");
   end CodePeer;
end My_Project;
```

```
codepeer -Pmy_project -level 1 ...
```

Analyze Messages (2/4)

- Runs contain many messages
- **Sample** them
- **Identify** groups of **false positives**
- **Exclude** them by categories
  - Using `--infer-messages` for INFER (level 0)
  - Using `--be-messages` for CodePeer (level 1+)
- For example, to disable messages related to access check:

```
--be-messages=-access_check
```

Analyze Messages (3/4)

- Filtering of messages
  - `-output-msg` `-hide-low` on the command line
  - Check boxes to filter on message category / rank in GNAT Studio and HTML
  - `--infer-messages --be-messages --gnat-warnings` switches
  - `-messages min/normal/max`
  - Pattern-based automatic filtering (`MessagePatterns.xml`)
- You can exclude a `package` or a subprogram from analysis
  - `pragma Annotate (CodePeer, Skip_Analysis)`

Analyze Messages (4/4)

- Choose relevant messages based on ranking
  - Rank = severity × certainty
  - **High** → certain problem
  - **Medium** → possible problem, or certain with low severity
  - **Low** → less likely problem (yet useful for exhaustivity)
- When analysing messages
  - Start with **High** rank
  - Then **Medium** rank
  - Finally **Low** rank if needed
- Considering only High and Medium is recommended
  - Default in GNAT Studio and HTML interfaces

Run CodePeer faster

- Hardware
  - 64-bit machine
  - Large amounts of memory
  - Large number of cores
- Command-line switches
  - Lower analysis level `-level <num>`
  - Paralellize `-j0` (default)
- Identify files taking too long to analyze
  - Disable analysis of their packages, subprograms or files

```
analyzed main.scil in 0.05 seconds
analyzed main__body.scil in 620.31 seconds
analyzed pack1__body.scil in 20.02 seconds
analyzed pack2__body.scil in 5.13 seconds
```

Code-Based Partial Analysis

- Excluding subprograms or packages from analysis
- `pragma Annotate (CodePeer, Skip_Analysis)`

```
procedure Complex_Subprogram (...) is
   pragma Annotate (CodePeer, Skip_Analysis);
begin
   ...
end Complex_Subprogram;

package Complex_Package is
   pragma Annotate (CodePeer, Skip_Analysis);
   ...
end Complex_Package;
```

Project-Based Partial Analysis

- Excluding Files From Analysis

```
package CodePeer is
   for Excluded_Source_Files use ( "xxx.adb" );
   -- Analysis generates lots of timeouts, skip for now
end CodePeer;
```

- Excluding Directories From Analysis

```
package CodePeer is
   for Excluded_Source_Dirs use ("directory1",
                                 "directory2");
end CodePeer;
```

- Excluding Projects From Analysis

```
for Externally_Built use "True";
```

Justifying CodePeer Messages

Database Justification

- Add review status in database
  - GNAT Studio: select review icon on message(s)
  - HTML web server: click on Add Review button above messages
  - Displayed with `-output-msg-only -show-reviews (-only)`
- Can run CodePeer as a server
  - Share the database on network
  - `codepeer --ide-server --port=8080`
- Access the IDE server from GNAT Studio
  - Set the project file to the following

```
package CodePeer is
   for Server_URL use "http://server:8080";
end CodePeer;
```

In-Code Justification

- Add message review pragma in code
- `pragma Annotate` added next to code with message
  - `False_Positive`: Condition in question cannot occur
  - `Intentional`: Condition is justified by a design choice
  - Also added in the database

```
...
return (X + Y) / (X - Y);
pragma Annotate (CodePeer,
                 False_Positive,
                 "Divide By Zero",
                 "reviewed by John Smith");
```

Outside Tooling Justification

- Use spreadsheet tool
  - Export messages in CSV format

    ```
    codepeer -Pprj -output-msg-only -csv
    ```

  - Review them via the spreadsheet tool (e.g. Excel)
    - Beware: Fill **all** the columns
  - Import back CSV reviews into the CodePeer database

    ```
    codepeer_bridge --import-reviews
    ```

- Use external justification connected to output
  - Textual output: compiler-like messages or CSV format

CodePeer Review Lab

Instructions

- Follow the `radar/` lab instructions.

CodePeer Workflows

CodePeer Use Cases

- Analyzing code locally prior to **commit** (desktop)
- **Nightly** runs on a server
- Continuous runs on a server after each **push**
- Any **combination** desktop/continuous/nightly run
- **Per-project** software customization
- **Compare** local changes with master
- Multiple teams **reviewing** multiple subsystems
- Use CodePeer to generate a **security report**

Analyzing Code Locally Prior To Commit (1/2)

- Each **developer** as a single user, on a **desktop** machine
- After compilation, before testing.
- Solution #1: File by File analysis
  - Use GNAT Studio menu
  - CodePeer → Analyze File
  - On the files that were **modified**
  - Fastest, incremental
- Solution #2
  - Run `codepeer -level 1/2 -baseline`
  - Local **baseline** database used for comparison
  - Look at **added** messages only
  - More exhaustive
  - Uses past reviews (less false positives)

Analyzing Code Locally Prior To Commit (2/2)

- If duration or number of messages is not good → refine the settings
- For each new message:
  - If a real issue is found → Fix the code
  - If it is a false positive → Justify it with `pragma Annotate`

Nightly Runs

- CodePeer run daily on a dedicated server
  - With large resources
  - Exhaustive level ($2 \rightarrow 4$)
- Typically run nightly
  - Takes into account commits of the day
  - Provides results to users the next morning
- Allows users to analyze and justify messages **manually**
  - Via the **web** interface
  - From GNAT Studio by accessing the **database** remotely
- At release, results can be committed under CM for **traceability** purposes

Continuous Runs

- CodePeer is run on a dedicated server
  - With large resources
  - Fast level (0 or 1)
- No need to be exhaustive
  - Focus on **differences** from previous run
- Continuous runs triggerred on repository events
- Summary is sent to developers
  - Email
  - Web interface

```
codepeer -Pprj -output-msg -only -show-added | grep "[added]"
```

- Developers then *fix the code*, or *justify the relevant messages*
  - via `pragma Annotate` in source code or via web interface.
  - or wait for the next nightly run to post a manual analysis via the HTML Output.

Combined Desktop/Nightly Run

- **Fast** analysis of code changes done at each **developer's desk**
- A longer and **more exhaustive** analysis is performed nightly
- The developer can re-use the **nightly** database as a baseline for analysis
- Database reviews **should** be stored in this database
  - No conflict with nightly runs
  - Updated every morning in the users' databases

Combined Continuous/Nightly Run

- **Fast** analysis of code changes done at each **developer's desk**
- A longer and **more exhaustive** analysis is performed nightly
- Alternatively: a baseline run is performed nightly
  - Same level as continuous runs and `-baseline`
- Database reviews **should** be stored in this database
  - No conflict with nightly runs
  - Updated every morning in the continuous database

Combined Desktop/Continuous/Nightly Run

- **Fast** analysis of code changes done at each **developer's desk**
- A **more exhaustive** analysis of code changes done continuously **on a server**
- A longer and **even more exhaustive** analysis is performed nightly
- Database reviews **should** be stored in this database
  - No conflict with nightly runs
  - Updated every morning in the users' and continuous databases

Software Customization Per Project/Mission

- A *core* version of the software gets branched out or instantiated
  - Modified on a **per-project/mission** basis
- Objectives
  - Separate CodePeer runs on **all** active branches
  - Database is used to **compare** runs on a **single** given branch
- **Continuous solution**
  - Justify message via `pragma Annotate` **only**
  - Merge of justifications handled via **standard CM**
  - Advantage: Code is self-justified
- **One shot solution**
  - **Version** the database alongside the code
  - At branch point database is **forked**
  - Database is maintained separately from there
  - Advantage: Can use database reviews

Multiple Teams Analyzing Multiple Subsystems

- Large software system with **multiple** subsystems
  - Maintained by **different** teams
- Perform a **separate** analysis for each subsystem
  - Using a separate workspace and database
- Create one project file (.gpr) per subsystem
- To resolve dependencies between subsystems, use `limited with`

```
limited with "subsystem1";
limited with "subsystem2";
project Subsystem3 is
   ...
end Subsystem3;
```

- Run CodePeer with:

```
codepeer -Psubsystem1 --no-subprojects
```

Comparing to Baseline

Baseline Runs

- Analysis running with latest source version
  - On a server
- Baseline run
  - **Reference** database
    - Is a *gold* reference
  - **All changes** are compared to it
  - **All reviews** should be pushed to it
- Create a baseline run
  - `codepeer -baseline`

Baseline With Continuous Integration

- Developers pre-validate changes **locally** prior to commit
  - Then create a **separate** branch and commits to it
- The continuous builder is **triggered**
  - Database is copied from the **Baseline** run
  - Setting are copied from the **Reference** run settings
- Results are reviewed via a spreadsheet tool (e.g. Excel)
- Reviews are imported into the CodePeer database
  - Can use `-show-added` to show only the **new** messages

```
codepeer -Pprj -output-msg -show-added | grep "[added]"
```

CodePeer Customization

CodePeer Specific Project Attributes

```
project Prj1 is
   ...

   package CodePeer is
      for Excluded_Source_Files use ("file1.ads", "file2.adb");
      --  similar to project-level attribute for compilation

      for Output_Directory use "project1.output";

      for Database_Directory use "/work/project1.db";
      --  can be local or on shared drive

      for Switches use ("-level", "1");
      --  typically -level -jobs

      for Additional_Patterns use "ExtraMessagePatterns.xml";
      --  also Message_Patterns to replace default one

      for CWE use "true";
   end CodePeer;
end Prj1;
```

Project Specialization For CodePeer

```
type Build_Type is ("Debug", "Production", "CodePeer");
Build : Build_Type := External ("Build", "Debug");

package Builder is
   case Build is
      when "CodePeer" =>
         for Global_Compilation_Switches ("Ada") use
         ("-gnatI",
          -- ignore representation clauses confusing analysis
          "-gnateT=" & My_Project'Project_Dir & "/target.atp",
          -- specify target platform for integer sizes, alignment, ...
          "--RTS=kernel");
          -- specify runtime library

      when others =>
         for Global_Compilation_Switches ("Ada") use ("-O", "-g");
         -- switches only relevant when building
   end case;
end Builder;
```

- Compile with
  `gprbuild -P my_project.gpr -XBuild=Production`
- Analyze with
  `codepeer -P my_project.gpr -XBuild=CodePeer`

Custom API For Race Conditions

- `pragma Annotate` can identify entry points and locks other than Ada tasks and protected objects

```
package Pkg is
   procedure Single;
   pragma Annotate (CodePeer,
                    Single_Thread_Entry_Point,
                    "Pkg.Single");
   procedure Multiple;
   pragma Annotate (CodePeer,
                    Multiple_Thread_Entry_Point,
                    "Pkg.Multiple");
end Pkg;
```

```
package Locking is
   procedure Lock;
   procedure Unlock;
   pragma Annotate (CodePeer, Mutex,
                    "Locking.Lock",
                    "Locking.Unlock");
end Locking;
```

Report File

- You can combine some or all of the following switches to generate a report file
- Mandatory switches:
  - `-output-msg`
  - `-out <report file>`
- Optional switches
  - `-show-header`
  - `-show-info`
  - `-show-removed`
  - `-show-reviews`
  - `-show-added`

```
package CodePeer is
   for Switches use ("-level", "max", "-output-msg",
                     "-out", "report_file.out",
                     "-show-header", "-show-info");
end CodePeer;
```

```
date : YYYY-MM-DD HH:MM:SS
codepeer version : 18.2 (yyyymmdd)
host : Windows 64 bits
command line : codepeer -P my_project.gpr
codepeer switches : -level max -output-msg -out
report_file.out -show-header -show-info
current run number: 4
base run number : 1
excluded file : /path/to/unit3.adb
unit1.ads:1:1: info: module analyzed: unit1
unit1.adb:3:1: info: module analyzed:
unit1__body
unit2.adb:12:25: medium: divide by zero might
fail: requires X /= 0
[...]
```

CodePeer Advanced Customization Lab

Instructions

- Follow the `cruise/` lab instructions.

CodePeer for Certification

CodePeer and CWE

- MITRE's Common Weakness Enumeration (CWE)
  - **Common** vulnerabilities in **software** applications
  - Referenced in many government contracts and cyber-security **requirements**
- CodePeer is officially **CWE-compatible**

  https://cwe.mitre.org/compatible/questionnaires/43.html
- CodePeer findings are **mapped** to CWE identifiers

```
project Prj1 is
   ...
   package CodePeer is
      for CWE use "true";
   end CodePeer;
end Prj1;
```

```
-- assign.adb:1: (pre)- assign:(overflow check [CWE 190])
-- Y /= 2_147_483_647
```

CodePeer and DO178B/C

- CodePeer **supports** DO-178B/C Avionics Standard
- DO-178C Objective A-5.6 (activity 6.3.4.f):

  Code Accuracy and Consistency (emphasis added)

  The objective is to determine the correctness and consistency of the Source Code, including stack usage, memory usage, **fixed point arithmetic overflow and resolution**, **floating-point arithmetic**, resource contention and limitations, worst-case execution timing, exception handling, **use of uninitialized variables**, cache management, **unused variables**, and **data corruption due to task or interrupt conflicts**.

  The compiler (including its options), the linker (including its options), and some hardware features may have an impact on the worst-case execution timing and this impact should be assessed.
- CodePeer **reduces** the scope of manual review
- See Booklet: Link: AdaCore Technologies for DO-178C/ED-12C
  - Authored by Frederic Pothon & Quentin Ochem

CodePeer and CENELEC - EN50128

- CodePeer **qualified** as a T2 tool for this CENELEC Rail Standard
- CodePeer supports:
  - D.4 Boundary Value Analysis
  - D.8 Control Flow Analysis
  - D.10 Data Flow Analysis
  - D.14 Defensive Programming
  - D.18 Equivalence Classes and Input Partition Testing
  - D.24 Failure Assertion Programming
  - D.32 Impact Analysis
- CodePeer is uniquely supportive of Walkthroughs and Design Reviews via its as-built documentation
- See Booklet: Link: AdaCore Technologies for CENELEC EN 50128:2011
  - Authored by Jean-Louis Boulanger & Quentin Ochem

How Does CodePeer Work?

How Does CodePeer Work?

- CodePeer computes the **possible** value
  - Of every **variable**
  - and every **expression**
  - at each **program point**
- Starting with a **leaf** subprograms
- Information is propagated up in the call-graph
  - Iterations to handle **recursion**
- For each subprogram `Sub`
  - It generates a **precondition** guarding against `Sub` check failures
  - It issues **check/warning** messages for `Sub`
  - It generates a **postcondition** ensured by `Sub`
  - It uses the **generated contracts** to analyze calls to `Sub`

How Does CodePeer Work?

See *CodePeer By Example* for more details

From GNAT Studio

Help → Codepeer → Examples → Codepeer By Example

CodePeer Limitations and Heuristics

- Let's explore section 7.13 of the User's Guide
- http://docs.adacore.com/codepeer-docs/users_guide/_build/html/appendix.html#codepeer-limitations-and-heuristics

CodePeer References

- CodePeer User's Guide and Tutorial
  - Online: https://www.adacore.com/documentation#codepeer
  - In local install at share/doc/codepeer/users_guide (or tutorial)
  - From GNAT Studio go to Help → Codepeer → Codepeer User's Guide (or Codepeer Tutorial)
- CodePeer website
  - http://www.adacore.com/codepeer
  - Videos, product pages, articles, challenges
- Book chapter on CodePeer
  - In Static Analysis of Software: The Abstract Interpretation, published by Wiley (2012)