Ada Code Analysis:
Technology, Experience, and Issues

C. Daniel Cooper
cdaniel.cooper@boeing.com
The Boeing Company
Ada Code Analysis

1: Technology
- capability, applicability, limitations

2: Experience
- examples: problems found, useful reports
- challenges: underleveraged technology

3: Issues
- vision of how things could be
- overcoming barriers to adoption

4: Summary
1: Code Analysis Technology

- Capability
  - current realities
  - technology enablers
- Applicability
  - usage categories
  - beneficiaries
- Limitations
Capability: Current Realities

- Traditionally minimal expectations
  - only call trees, cross-reference tables, metrics
  - home-grown, ad hoc tools: full language not supported

- Analysis feedback is typically missing

- But for some problem classes, analysis is:
  - the earliest, cheapest means of detection
  - possibly the *only* means of detection

- Analysis technology is now mature
  - available: industrial strength tools
  - scalable: thousands of files, MSLOCs
Capability: Technology Enablers

- Ada Semantic Interface Specification (ASIS)
  - binding for ARM-based query language
  - ISO Standard 15291
- Compiler provides the analysis database
  - analysis tools only query the compilation data
  - certified full language support
- Other analysis technologies also available
- Results displayed textually or graphically
Applicability: Usage Categories

- **Descriptive Use:** as-built information
  - code comprehension and review
  - architectural dependencies (inter-unit, inter-layer)
  - document generation

- **Prescriptive Use:** assessment
  - inefficiency and error discovery
  - architectural enforcement, complexity reduction
  - style and standards compliance
  - portability and reuse, dead code elimination
  - maintainability and quality improvement

- **Corrective Use:** repair and refactoring
Applicability: Beneficiaries

- Back end of life-cycle (as soon as code exists)
- Individual developers: daily “hygiene”
- Code reviews
  - unbiased, automated
  - issue-based versus reviewer-based
- Formal releases, quality assurance
- Range of domains
  - high-integrity, safety-critical applications
  - less emphasis where time-to-market dominates
Technology Limitations

- **Code analysis is static, not dynamic**
  - capitalizes on structural/semantic information
  - path-based but not dynamic run-time behavior
- **Analyzed code must be compilable**
- **Computationally intensive**
  - high-end hardware needed
- **Under/over-reporting of analysis results**
  - may miss some cases of interest
  - may report “potential” cases (false positives)
2: Code Analysis Experience

- **Examples**
  - problems found
  - useful reports
- **Challenges**
  - underleveraged technology
Experience: Problems Found

Typographical errors:

```ada
function Convert is new Unchecked_Conversion
  (Source  =>  System.Address,
   Target  =>  State_Table);
```

Type-cast of 32 bits into 2048 bits! Should be:

```ada
function Convert is new Unchecked_Conversion
  (Source  =>  System.Address,
   Target  =>  State_Table_Access);
```
Experience: Problems Found

Broken, but seems okay (not WYSIWYG):

```ada
Fahrenheit : Temperature.Value;
Centigrade  : Temperature.Value;
begin
  Fahrenheit := ...something...
  Centigrade := (Fahrenheit - 32.0) * 5.0 / 9.0;
```

Grossly miscalculates conversion… can stare at this for hours… maybe a compiler bug?
Experience: Problems Found

function "+" (Left, Right : Temperature.Value) return Temperature.Value renames Temperature."+";
function "-" (Left, Right : Temperature.Value) return Temperature.Value renames Temperature."-";
function "*" (Left, Right : Temperature.Value) return Temperature.Value renames Temperature."*";
function "/" (Left, Right : Temperature.Value) return Temperature.Value renames Temperature."/";

Fahrenheit : Temperature.Value;
Centigrade : Temperature.Value;
begin
  Fahrenheit := ...something...
  Centigrade := (Fahrenheit - 32.0) * 5.0 / 9.0;
Experience: Problems Found

Okay, but seems broken:

```ada
type Fail_Type is (Overrun, RAM_Error, HW_Error,
                  Hard_Failure, RTC_Failure, Watchdog_Timer);

subtype Hard_Fail_Type is Fail_Type range
                        Hard_Failure .. Watchdog_Timer;

Error: Hard_Fail_Type;
begin
    Error := Overrun;

Should raise Constraint_Error, but…
```
Experience: Problems Found

Missing code:

```ada
function Convert (Color : Colors) return Integer is
begin
  case Color is
    when Blue   => return 6;
    when Green  => return 17;
    when Red    => return 23;
    when others => null;
  end case;
end Convert;
```

Should raise Program_Error, but…
Experience: Problems Found

Extraneous dead code: four granularities

- unWITHed units
- uncalled subprograms
- infeasible execution paths (unreachable statements)
- unneeded declarations or clauses

Dead code includes “weak dependencies” e.g.:

- a representation clause for an unused type
- a pragma for an uncalled subprogram
- a WITH clause supporting only an unneeded USE
- the chain of weak dependencies can be long
Experience: Problems Found

Questionable oddities:

-- single-literal enum versus a constant

```ada
type Foo is (Bar);
for Foo use (Bar => 23);
```

-- unnecessary redundant rep spec (Ada95)

```ada
type Foo is (A, B, C, D);
for Foo use (A => 0,
              B => 1,
              C => 2,
              D => 3);
```
Experience: Problems Found

Local cohesion, global coupling:

unintended indirect recursion
Experience: Problems Found

(More) local cohesion, global coupling:

protected action (eventually) calls blocking op
Experience: Useful Reports

Architectural Issues

- unneeded WITH clauses
- over-scoped WITH clauses
- excessive WITHing (or trend)
- variables declared in package spec
- one-way IN OUT parameters
- exception propagation
- potentially redundant record types
- candidate subunits
Experience: Useful Reports

Performance Issues

- potential race conditions
- run-time expensive types
- dynamic instantiations
- elaboration impacts
- variables set twice before use
- candidate InLine subprograms
- candidate short-circuit operators
- unneeded IF to assign Booleans
Experience: Useful Reports

**Programming Issues**
- variables used before set
- potential division by zero
- equality operators for reals
- inevitable constraint violations
- inconsistent representation clauses
- side-effects in functions
- unneeded type conversions
- redeclared predefined names
Experience: Challenges

- Appallingly underleveraged technology!
  - non-technical reasons
  - unrecognized benefits, values not shared
  - schedule and time-to-market pressures

- Disruption of the status quo
  - developers: analysis results are job interruption
  - testers: easier workload, but it’s not their job

- Tyranny of the functional
  - build the house, but leave the trash
  - “It ain’t broke, so don’t fix it!”
  - “Better is the enemy of Good Enough.”
3: Code Analysis Issues

- Vision of how things could be
  - improved code quality
  - improved tool usage
- Overcoming barriers to adoption
  - management barriers
  - environment barriers
  - cultural barriers
Vision of Improved Code Quality

• Enhanced coding standards
  • enabled by automation
  • living, evolving: not frozen at project start

• Deeper code assessment
  • expose hard-to-find problems
  • for some problems, may be the *only* recourse
  • easy for users to perform ad hoc analyses
  • greater code consistency

• Continuous, real quality improvement
  • refactoring: code evolution, anti-entropy
Vision of Improved Tool Usage

• Traditional approach:
  ```
  while Code_Not_Working loop
    Edit; Compile; Link; Test;
  end loop;
  ```

• Visionary approach:
  ```
  while Code_Not_Working loop
    while Code_Not_Clean loop
      Edit; Compile; Analyze;
    end loop;
    Link; Test;
  end loop;
  ```
Overcoming Management Barriers

• Support
  • quality improvement must be a shared value

• Benefit versus Cost
  • reduce cost of testing, downstream maintenance
  • bigger cost is engineering time, not tool licenses
  • history confirms cost of low quality

• Planning
  • must be part of the infrastructure, institutionalized

• Time-to-Market Pressures
  • different needs: embedded versus office products
Overcoming Environment Barriers

- Recognized Process
  - Analysis results need somewhere to go
  - Must be overt part of the development process

- Integrated Environment
  - Assure easy access, non-obtrusive
  - Include industrial-strength tools
  - Support alternate approaches, tech-transfer

- Evangelism
  - Capability/resources must be public knowledge
  - Software architect should be strongest proponent
Overcoming Cultural Barriers

- Analysis results can be overwhelming
  - need strategy to prioritize, filter

- User-friendliness
  - analysis must be usable as well as useful
  - users may need special training

- “It ain’t broke, so don’t fix it.”
  - more kinds of “broke” than just functional
  - “so what” attitude: must understand the issues
  - pride in high-quality code

- Quality is everyone’s job
4: Code Analysis Summary

- Addresses significant problems
- On real platforms
  - mature, industrial strength, ASIS-based
- Reducing the cost
  - earlier detection = cheaper to fix
  - improved quality = cheaper to maintain
- Non-obtrusively
  - incorporated into development process
  - integrated with development environment

Go forth and do great things!