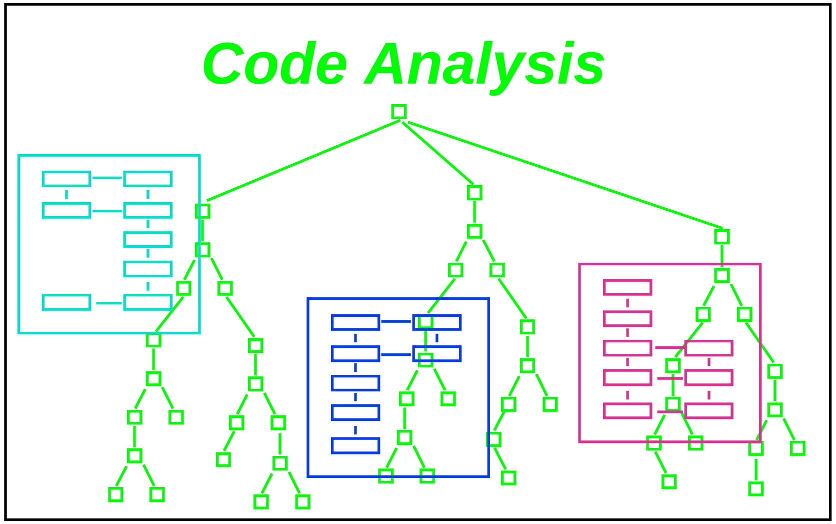
Code Analysis for Quality in High Integrity Systems



Currie Colket
The MITRE Corporation

Phone: (703) 883-7381
Email: ***colket@mitre.org | colket@acm.org***

American Society for Quality Meeting
23 March 2004

Acknowledgement and Thanks to Bill Thomas for his help and ideas.

So Why Code Analysis?

Many lessons learned from hind-sight

We now know what to look at to identify expected potential problems in software

This isn't really rocket science

Overview

- Introduction
- High Integrity
- Code Analysis
- Quality
- Problems with Code Analysis
- Automation of Code Analysis
- Experiences with Code Analysis
- Conclusion

High Integrity Systems - 1

Adherence to Guidelines or Standards has to be demonstrated

- **Safety – IEC 610508 International Generic Standard (Part 3 concerned with software)**
- **Security – ISO 15408 – Multinational Generic Assessment Guide**
- **Sector Specific Guidelines**
  - **Airborne Civil Avionics [DO-178B]**
  - **Nuclear Power Plants [IEC 880]**
  - **Medical Systems [IEC 601-4]**
  - **Pharmaceutical [GAMP]**

Many others

- **National Regional Guidance**
  - **UK Defence [DS 00-55]**
  - **European Rail [EN-50128]**
  - **US Automotive [MISRA]**

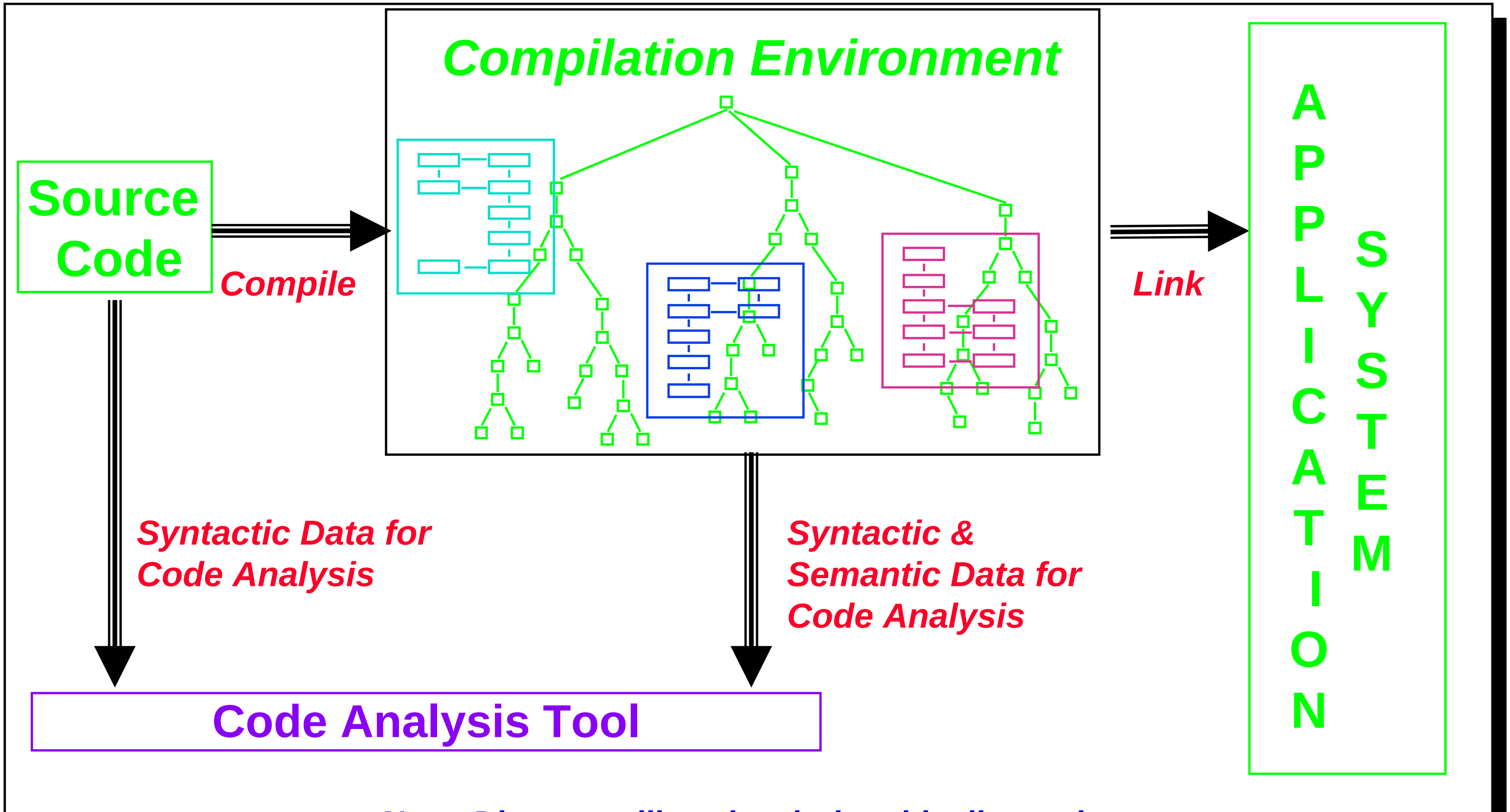
High Integrity Systems - 2

Verification – ***The confirmation by examination and provision of objective evidence that the specified requirements have been fulfilled [ISO 8402:2.18]***

In General, Adherence demonstrated to an Independent Body by:

- **Traceability – required to establish implementation is complete**
- **Reviews – review requirements, design, code, test procedures, analysis reports, etc. – Facilitated by adherence to Coding standards (e.g., Ada Quality and Style Guide)**
- **Analysis – (static analysis) analysis of the design or code**
- **Testing – (dynamic analysis) execution of software on a digital computer, providing tangible, auditable evidence of software execution behavior**
  - **Can be done at module, integration, & system level**
  - **Requirements (black box) or Structure (white box)**

Code Analysis



Note: Diagram will evolve during this discussion

Introduction
Code Analysis

10 Analyses frequently called out by High-Integrity Standards

- **Control Flow Analysis**
- **Data Flow Analysis**
- **Information Flow Analysis**
- **Symbolic Flow Analysis**
- **Formal Code Verification**
- **Range Checking**
- **Stack Usage Analysis**
- **Timing Analysis**
- **Other Memory Usage Analysis**
- **Object Code Analysis**

Many more, but these are the common ones

Code Analysis –
1. Control Flow Analysis

- **Conducted to:**
  - **Ensure code executed is in the right sequence**
  - **Ensure code is well structured**
  - **Locate any semantically unreachable code**
  - **Highlight parts of Code where termination needs to be considered (i.e., loops and recursion)**
- **Analyses:**
  - **Sequencing Analysis (verify to design)**
  - **Call Tree Analysis**
    - **Detects Dead Code, Direct Recursion (Bad), Indirect Recursion (Really Bad)**
  - **Partitioning Analysis (critical & non-critical)**
  - **Structure Analysis (GOTOs, Use of Loop Control Variables, placement of Exit & Return Statement, etc.)**

Lends itself nicely to automatic code analysis

Code Analysis – 2. Data Flow Analysis

- Conducted to:
  - Ensure no execution path in software that would
    - Access a variable that has not been set
    - Insure all input only parameters are not set
    - Ensure all output parameters are set

      (both for procedures and for functions)
  - Ensure global data is shared properly
- Analyses:
  - Uses results of Control Flow Analysis to ensure data is set before used
  - Evaluates read or write access to variables
  - Identifies data that is globally shared without protection

Lends itself nicely to automatic code analysis

Code Analysis –
3. Information Flow Analysis

- **Conducted to:**
  - **Ensure dependencies between inputs and outputs are verified to the specification**
- **Analyses:**
  - **Internal to a module (e.g., procedure or function)**
  - **Across modules**
  - **Entire CSCI**
  - **Entire System**

Particularly valuable for critical output that can be traced to inputs of software/hardware interface

Example:

```
X := A + B;
Y := D – C;
if X>0 then
    Z := Y + 1;
end if;
```

Here:

X depends on A & B
Y depends on C & D
Z depends on A, B, C, & D
and implicitly on Z's initial value

Partially supported by automatic code analysis

Code Analysis – 4. Symbolic Execution

- **Conducted to:**
  - **Verify properties of a program by algebraic manipulation of the source text without requiring a formal specification**
- **Analyses:**
  - **Typically performed where the program is "executed" statically by performing back-substitution**
  - **Converts sequential logic into a set of parallel assignments in which output values are expressed in terms of input values**

Previous Example:

```
X := A + B;
Y := D – C;
if X>0 then
    Z := Y + 1;
end if;
```

```
A + B <= 0:
    X = A + B
    Y = D – C
    Z = not defined
A + B > 0:
    X = A + B
    Y = D – C
    Z = D – C + 1
```

Partially supported by automatic code analysis

MITRE

Code Analysis –
5. Formal Code Verification

- **Conducted to:**
  - **Prove the code of a program is correct with respect to the formal specification of its requirements**
  - **Explore all possible program executions, which is infeasible by dynamic testing alone**

- **Analyses:**
  - **Pre-condition/Post-condition analysis**
  - **Demonstrate a particular safety/security property**
  - **Termination of all loops**
  - **Termination of any recursion (not normally permitted)**
  - **Proof of absence of run time errors**

Partially supported by automatic code analysis

Code Analysis –
6. Range Checking

- **Conducted to:**
  - **Ensure data values lie within the specified ranges**
  - **Ensure data maintains specified accuracy**

- **Analyses:**
  - **Overflow and Underflow Analysis**
  - **Range Checking Analysis**
  - **Array Bounds Checking**
  - **Rounding Errors Analysis**

Special Problems
when mixing
different languages

Discrete static bounds can often be checked automatically

Checking is straight forward for Enumeration Types

Absence of overflow for Real Types can be demanding

Lends itself nicely to automatic code analysis

Code Analysis –
7. Stack Usage Analysis

- **Conducted to:**
  - **Ensure sufficient physical memory to support the maximum stack size (for each stack)**
  - **Ensure no possible stack/heap collision at run-time**
- **Analyses:**
  - **Verify stack memory requirements each subprogram, block, task, or other construct implemented**
  - **Identify maximum possible size of the stack required by the system.**
  - **Verify that dynamic heap allocation is prohibited**

Note: Analysis is made easier with static types;
dynamic types (e.g., OO) complicate analysis.

Lends itself nicely to automatic code analysis

Code Analysis – 8. Timing Analysis

- **Conducted to:**
  - **Ensure temporal properties of the input/output dependencies**

- **Analyses:**
  - **Worse Case Timing Analysis**
  - **Identification of infinite loops (frequently desired)**

Certain constructs make this analysis impossible

(e.g., infinite loops, manipulation of dynamic data structures)

Lends itself nicely to automatic code analysis

Code Analysis –
9. Other Memory Usage Analysis

- **Conducted to:**
  - **Ensure Memory usage does not exceed capacity**

- **Analyses:**
  - **Analysis of Heap Memory**
  - **Analysis of I/O Ports**
  - **Analysis of special purpose hardware**

Lends itself nicely to automatic code analysis

Code Analysis –
10. Object Code Analysis

- **Conducted to:**
  - **Ensure object code is a direct translation of source code (errors have not been introduced as a direct result of a compiler bug)**
- **Analyses:**
  - **Manual inspection of critical areas of object code**

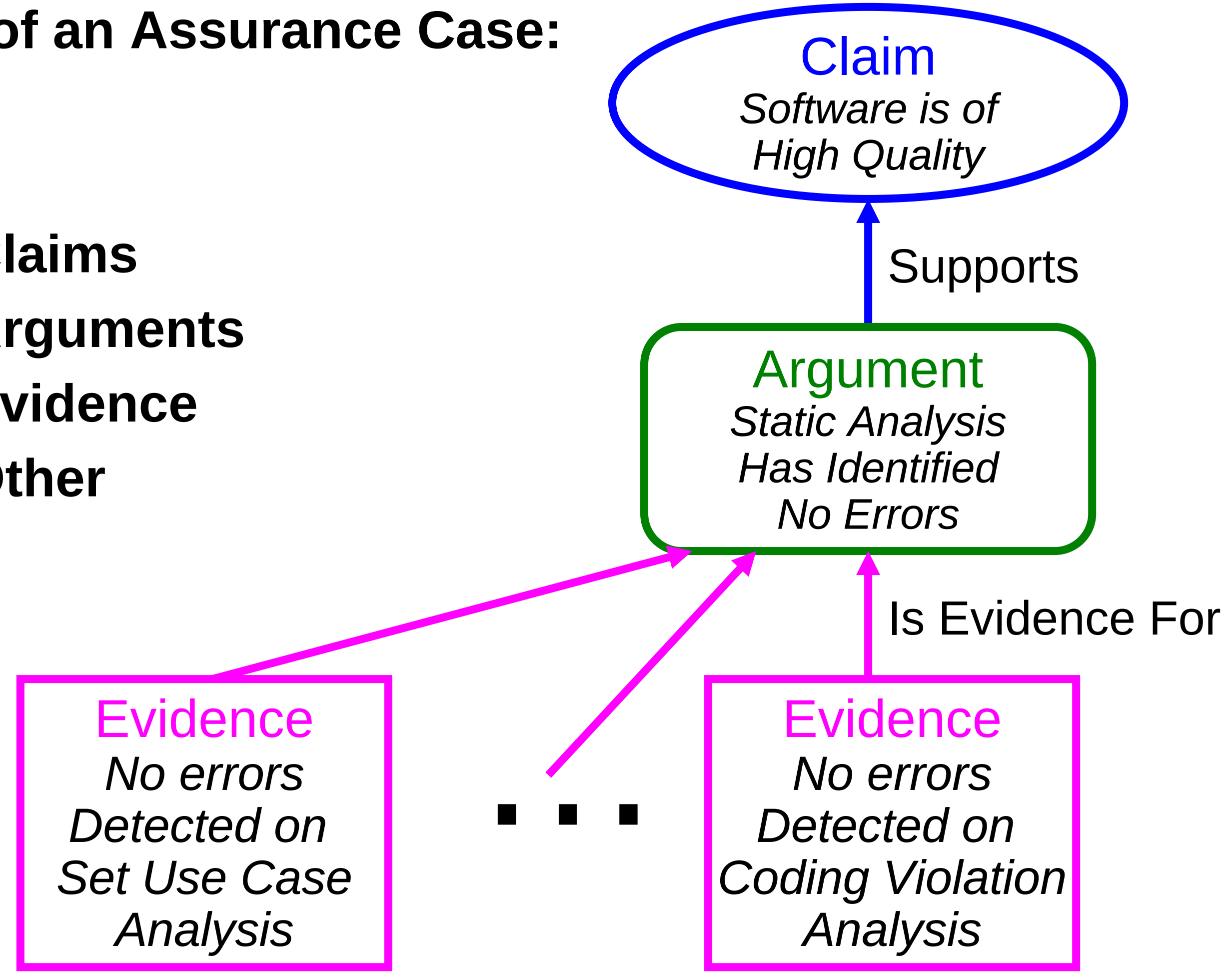
Supported in Ada by Pragma Inspection_Point to determine the exact status of variable at specific points

Not really a candidate for automatic code analysis

Information Assurance & Code Analysis

Elements of an Assurance Case:

- **Claims**
- **Arguments**
- **Evidence**
- **Other**



MITRE

Overview

- Introduction
- High Integrity
- Code Analysis
- Quality
- Problems with Code Analysis
- Automation of Code Analysis
- Experiences with Code Analysis
- Conclusion

How Not To Do Systems Engineering For Quality

And The Sinking Of The Largest Offshore Oil Platform

March 2001

Disclaimer: Slides Received From Unknown Author


For those of you who may
be involved in the
engineering of systems

Please read this quote from
a Petrobras executive,


extolling the benefits of
cutting quality assurance
and inspection costs,


on the project that
sunk into the Atlantic
Ocean off the coast of
Brazil in March 2001.


"Petrobras has established new global benchmarks
for the generation of exceptional shareholder wealth


through an aggressive and innovative programme
of cost cutting on its P36 production facility.


Conventional constraints have been successfully challenged

and replaced with new paradigms appropriate to the globalised corporate market place.


Through an integrated network
of facilitated workshops,

the project successfully rejected the established constricting and negative influences of prescriptive engineering,

onerous quality requirements, and
outdated concepts of inspection
and client control.

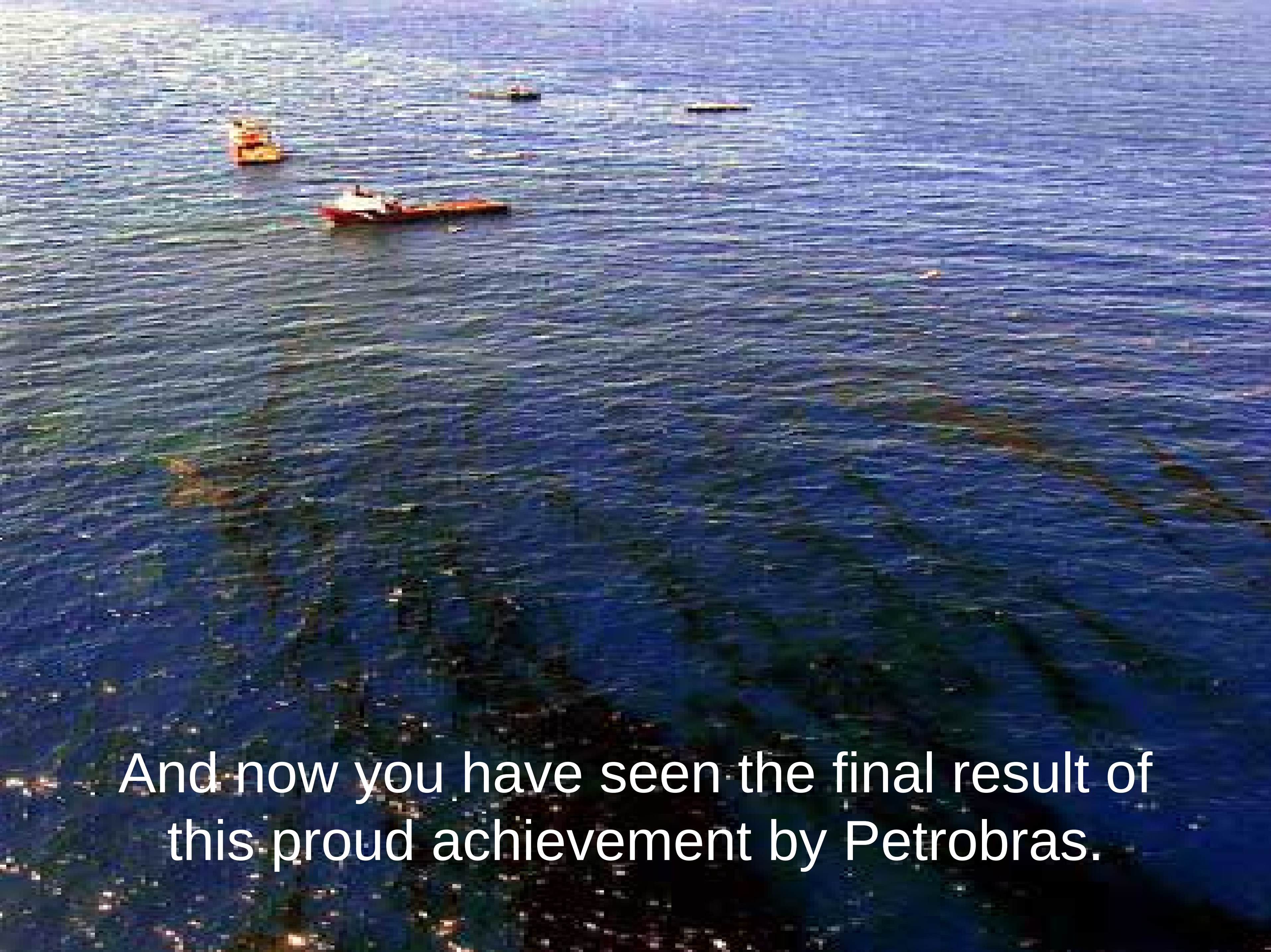

Elimination of these unnecessary straitjackets has empowered the project's suppliers and contractors to propose highly economical solutions,


with the win-win bonus of enhanced
profitability margins for themselves.

The P36 platform shows the shape of things to come

http://www.sustainability.com/developing-value/details.asp?bcid=128&sfid=3&bsid=3

in unregulated global market economy of the 21st Century.”


And now you have seen the final result of
this proud achievement by Petrobras.

Ada Engineered Product
Boeing 777 Commercial Aviation

Boeing 737
Boeing 747
Boeing 757
Boeing 767
Boeing 777

Boeing was leader in ASIS development

Today Boeing uses automatic code analysis on every piece of software that controls commercial and military aircraft

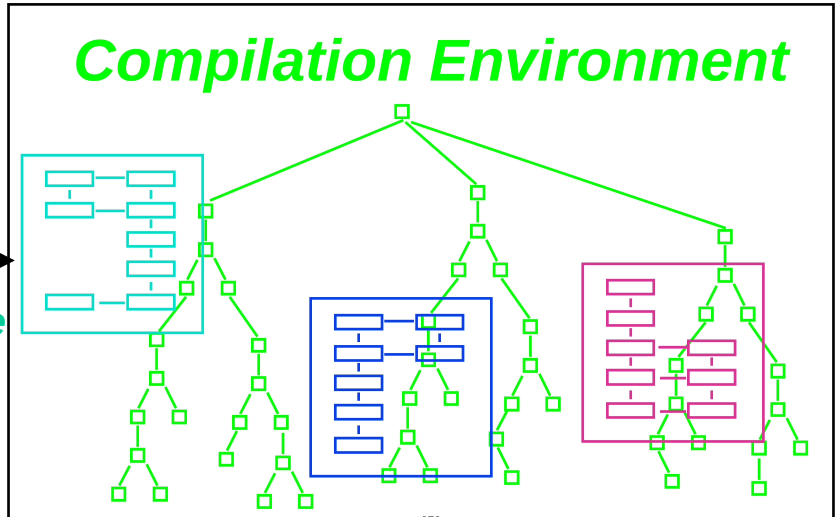
Overview

- Introduction
- High Integrity
- Code Analysis
- Quality
- Problems with Code Analysis
- Automation of Code Analysis
- Experiences with Code Analysis
- Conclusion

Problems With Code Analysis

Source Code

Compile



Link

APPLICATION SYSTEM

Syntactic Data for Code Analysis

1. Semantic Data not true to Compiler
2. Many Semantic analyses too complex
3. Tools are costly

Syntactic & Semantic Data for Code Analysis

1. Costly to implement
2. Change in compiler breaks tool
3. No consistent set of queries

Code Analysis Tool

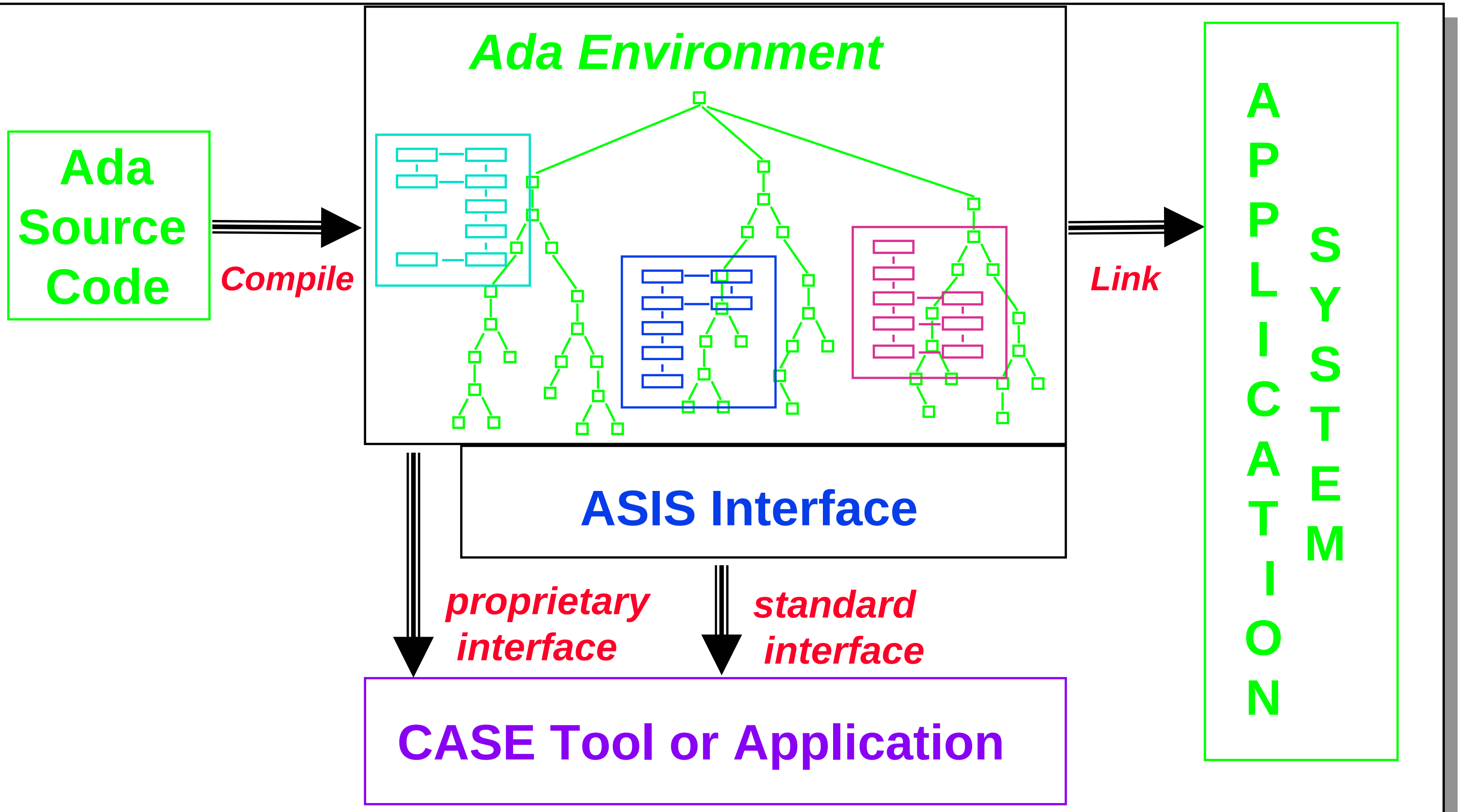
ASIS – An Interface to Support Code Analysis

- **The Ada Semantic Interface Specification (ASIS) exception handling mechanism is a good example demonstrating the types of things useful to do in a design.**

  The Ada Semantic Interface Specification (ASIS) is an interface between an Ada environment (as defined by ISO/IEC 8652:1995) and any tool or application requiring information from it. An Ada environment includes valuable semantic and syntactic information. ASIS is an open and published callable interface which gives CASE tool and application developers access to this information. ASIS has been designed to be independent of underlying Ada environment implementations, thus supporting portability of software engineering tools while relieving tool developers from needing to understand the complexities of an Ada environment's proprietary internal representation. In short, ASIS can provide the foundation for your code analysis activities.

ASIS became an ISO Standard in 1999
See *http://www.acm.org/sigada/WG/asiswg*

What is ASIS?



Provides Syntactic and Semantic Information from Ada Environment using a standard interface

MITRE

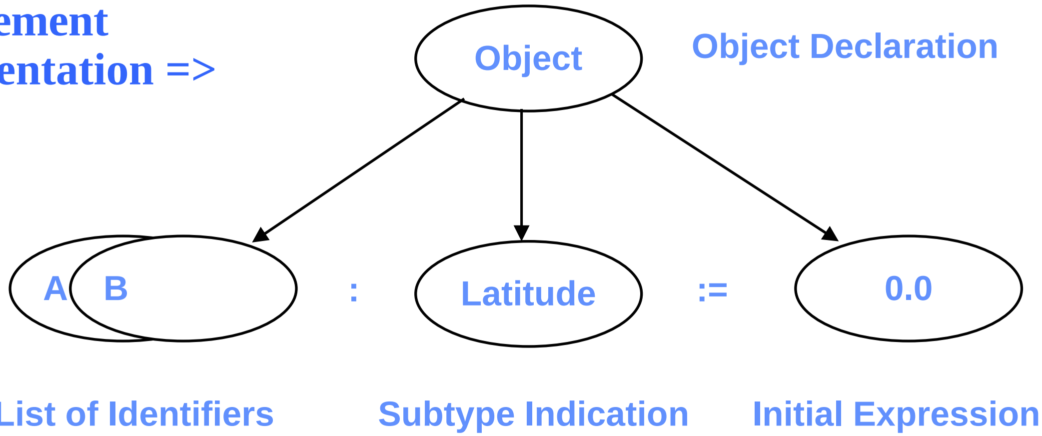
Syntactic Information

Ada syntax is summarized in Ada 95 RM, Annex P as variant of Backus-Naur Form

For example:
object_declaration ::==
defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression]; | ...

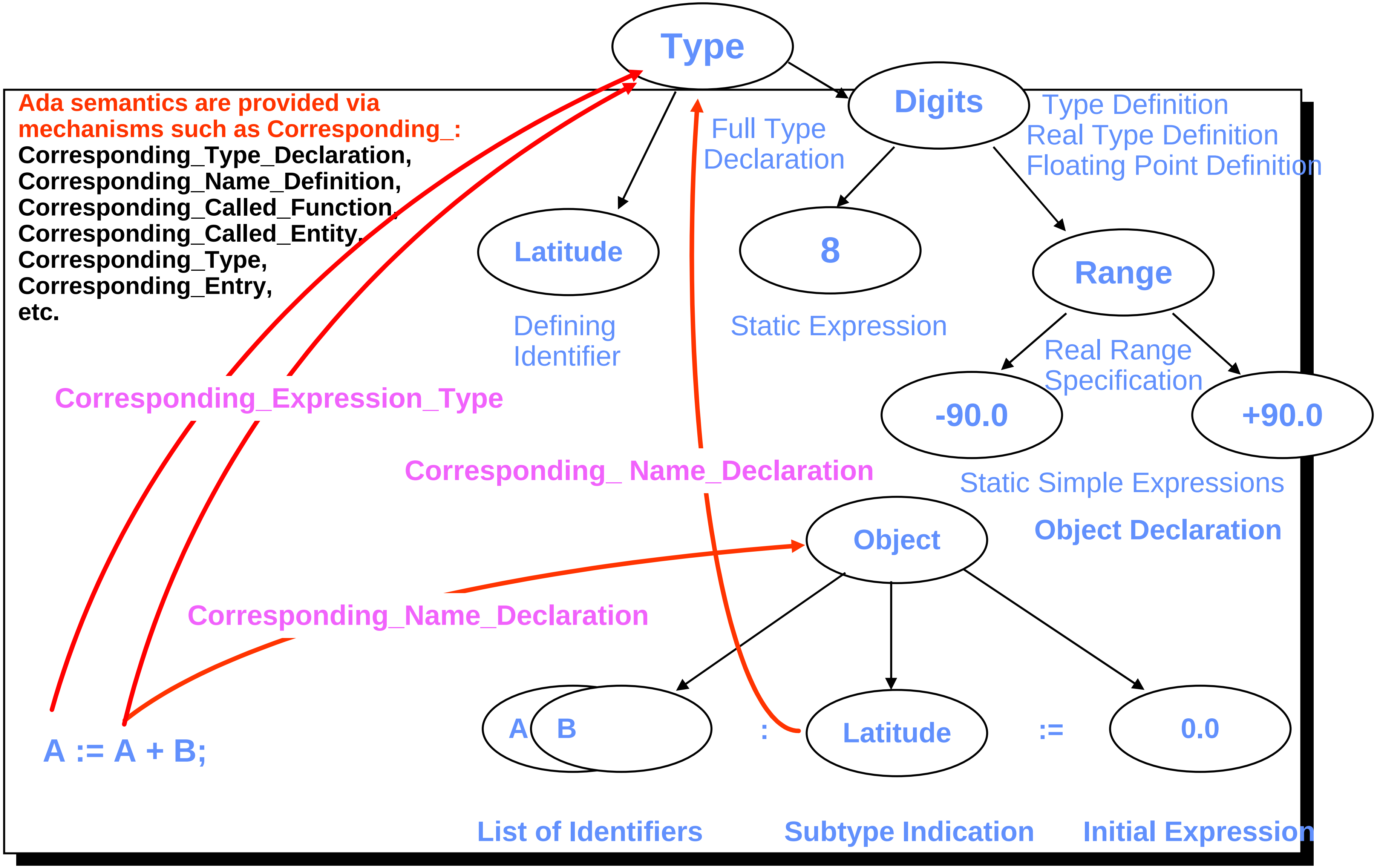
For the Ada object declaration => A,B: Latitude := 0.0;

Syntactic Element
Tree Representation =>



ASIS can extract desired syntactic information for every syntactic category
Of the 367 ASIS Queries, most support syntactic tree analysis

Semantic Information

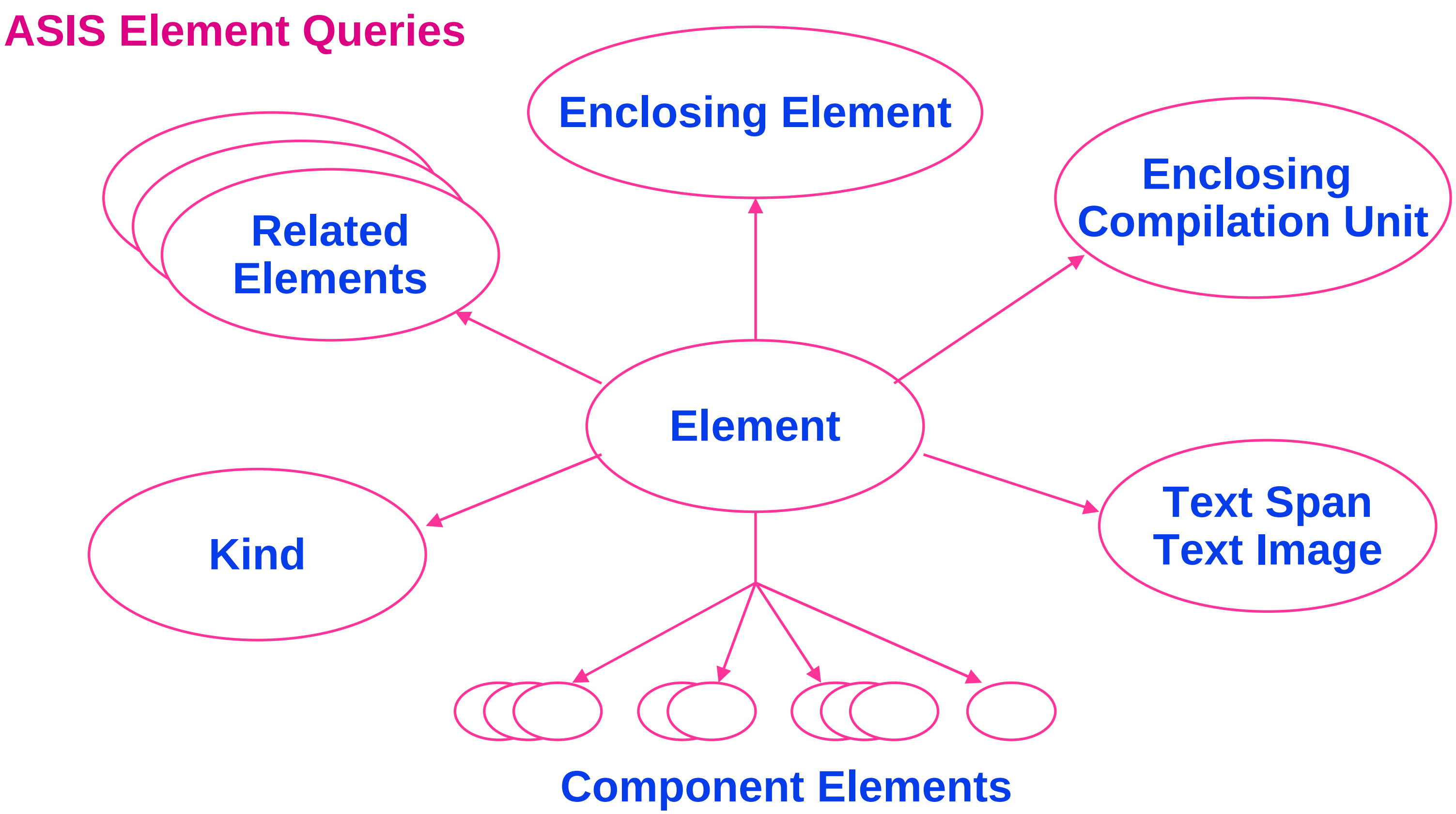


These mechanisms allow ASIS to traverse the syntactic tree like Hypertext allows one to traverse a document

MITRE

Operations on Elements

Element. A common abstraction used by ASIS to denote the syntax components (both explicit and implicit) of ASIS compilation units.



Overview

- Introduction
- High Integrity
- Code Analysis
- Quality
- Problems with Code Analysis
- Automation of Code Analysis
- Experiences with Code Analysis
- Conclusion

Experience - Objectives of Code Analysis

- Determine existing software suitability as a foundation for subsequent development
- Identify strengths and weaknesses in the design and implementation, and identify areas for improvement

General approach for the Code Analysis:

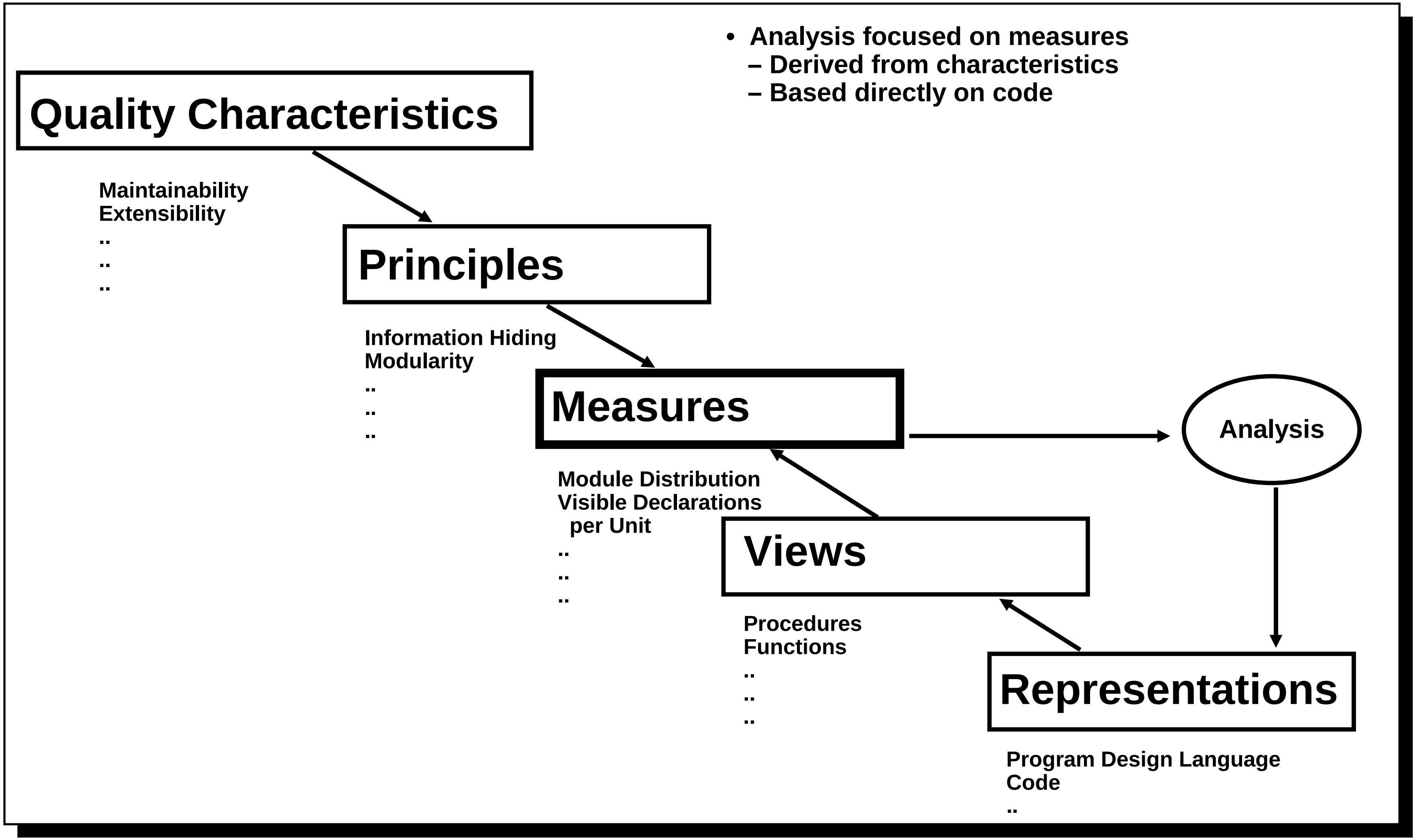
- Examine code for evidence of sound workmanship.
- Isolate areas of the product that, while not necessarily incorrect, may lead to difficulty in test and/or maintenance.
- Examine code for selected specific anomalies that may impact the correctness or reliability of the product.
- Examine code for selected specific anomalies that, while not directly impacting correctness, may impact portability or maintainability

Analysis Tools

- **Design Assessment Workbench (ASIS-based)**
  - **Metrics for Design Assessment**
  - **Tailorable for specific needs**
- **Ada System Dependency Analyzer (SDA) (ASIS-based)**
  - **Provides reports on various types of dependencies**
- **Ada Analyzer (ASIS-based)**
  - **Useful for identification of detailed anomalies**
  - **Violation of coding standards**
- **Battlemap**
  - **Provides a variety of metrics that focus on module cyclomatic complexity**
- **Exception Propagation Analysis Tool (EPAT) (ASIS-based)**
  - **Determines exception propagation paths**

Other ASIS-based tools e.g., AdaSTAT

Design Assessment Framework



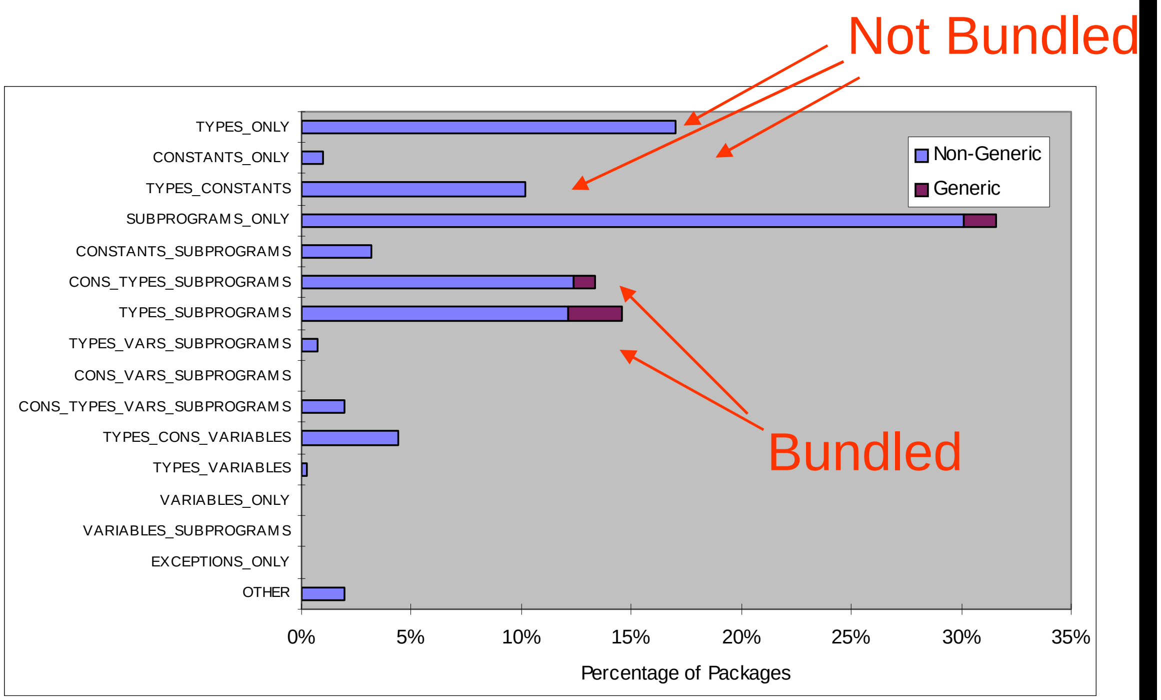
Views and Measures for Data Extraction

VIEW	MEASURE	AFFECTED PRINCIPLE
Program Units	Unit Distribution	Abstraction, Information Hiding
	Package Distribution	Abstraction, Explicit Interfaces, Information Hiding, Weak Coupling
Name Space	Visible Declarations in a Library Package	Abstraction, Information Hiding, Modularity, Weak Coupling
Subprograms	Subprogram Parameter Count	Explicit Interfaces, Modularity, Weak Coupling
	Cyclomatic Complexity	Modularity
Compilation Unit Dependency	Recompilation Effect	Minimal Interfaces, Modularity
Structure Chart	Fan Out	Modularity

Package Distribution

Observation:

- **Significant percentage of packages that do not bundle types and subprograms**

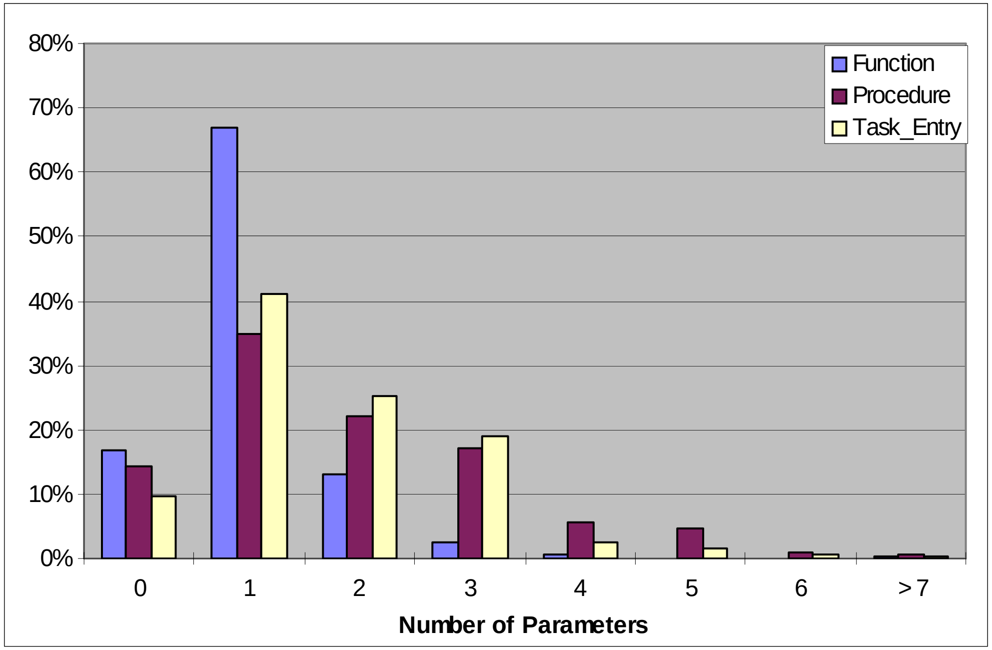


Subprogram Parameter Distribution

Observations:

- **Nine percent of the packages export global data**
- **Significant percentage of procedures / functions having no parameters**

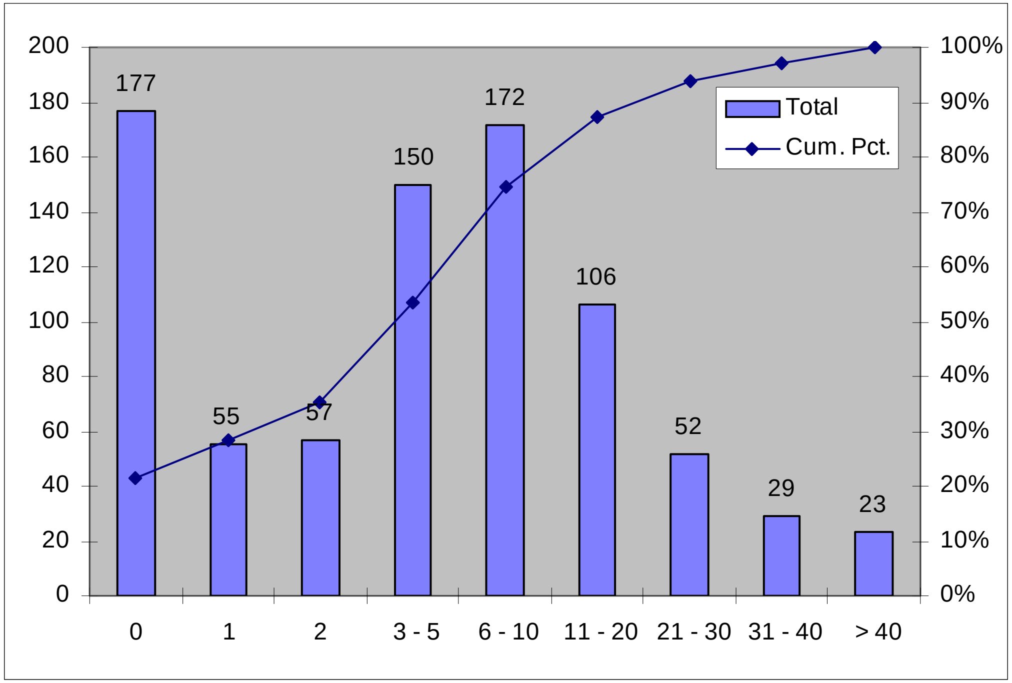
  Are these abstract state machines, or is this excessive use of global data?



Compilation Unit Context Coupling Distribution

Observation:

- **Compilation units that have a large number of imports may be indicative of a lack of cohesion, and as such could be candidates for further decomposition**

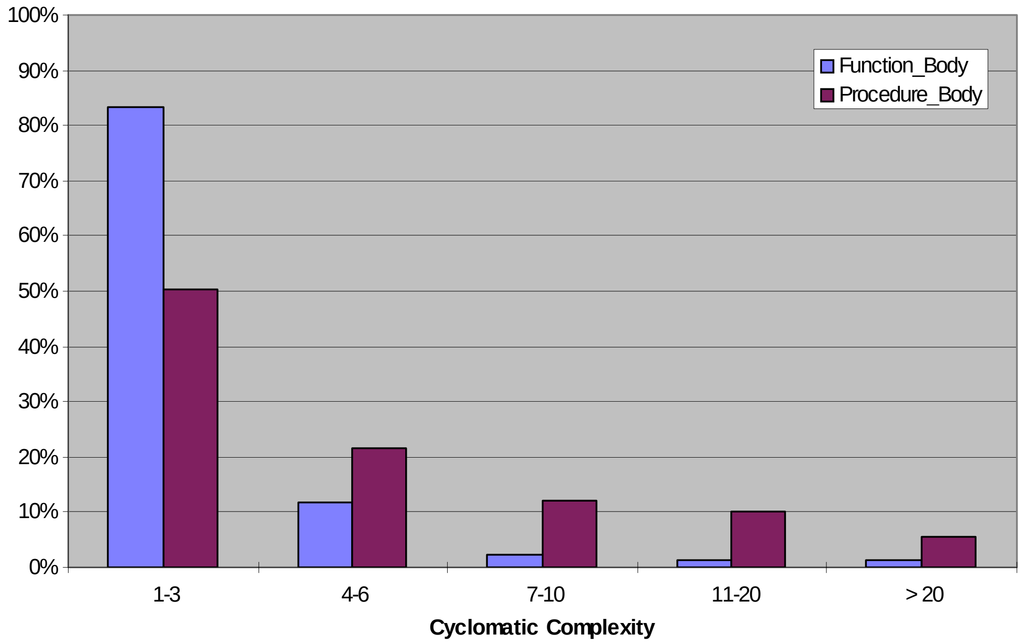


MITRE

Cyclomatic Complexity Distribution

Observations:

- **Most subprograms have cyclomatic complexity less than 10**
- **Those with higher values in some cases may benefit from further decomposition**



MITRE

Ada System Dependency Analyzer Metrics 3

Subunits	150
Parent Program Units	10
Subunits Specified but not Found	0
Subunits Found but not Specified	0
Missing Parent Units	0
Exceptions	677
Exceptions Declared More Than Once	67
Exceptions Declared but not Raised	145
Missing Exception Declarations	0
Raise Statements	2,944
Raises Not Identified At All	0
Raises Identified More Than Once	0
Machine Code Statements	0

MITRE

Ada Analyzer – Classification of Anomalies

Category	Description	Recommendation
Serious	This represents a **possible** error, which could cause the program to generate erroneous results or crash.	These should be evaluated by the development contractor and fixed, if valid
???	This represents a **possible** error, which could cause the program to generate erroneous results or crash. However, this is most likely caused by differences in the development environment and the analysis environment.	These should be evaluated by the development contractor and fixed, if valid
Nit	This represents a **possible** error, which should not impact the mission (e.g., logging operation).	These should be evaluated by the development contractor and fixed, if valid
Performance	This represents a situation where throughput **might** be improved if violation addressed.	These should be evaluated and fixed, if a significant performance improvement is possible and desirable.
Maintenance 1	Changes in these areas **might** provide significant short-term benefits to the software	Changes should be implemented in the short-term.
Maintenance 2	Changes in these areas **might** provide long-term benefits to the software	Changes should be implemented as units are modified
Maintenance 3	Changes in these areas **might** provide significant benefits to the system for eventual porting to new hardware or compilation environment	Changes should be implemented as units are modified

Ada Analyzer Analyses

Analysis	Reports
Annotations Analysis	0
Coding Violations Analysis	**55**
Collected Metrics	6
Compatibility Analysis	7
Exception Analysis	1
Miscellaneous Programming Error Analysis	**11**
Misspelling Analysis	1
Name Anomalies Analysis	0
Representation Specification Consistency Analysis	**11**
Set Use Problems Analysis	**3**
Static Constraint Violation Analysis	**7**
Subprogram Execution Analysis	**2**
Tasking Analysis	1
Unused With Clause Analysis	7

Ada Analyzer – Coding Violations Analysis

Set Use Analysis Summary	# Violations	# Serious	# ???	# Nit	# Perf	# Main1	# Main2	# Main3
Address Of Dynamic Objects	149	0	149	0	0	0	0	0
Static Constraint Violations	51	37	14	0	0	0	0	0
Missing Storage Error Handler For Allocators	143	0	143	0	0	0	0	0
Blocking Operations in Protected Types	7	7	0	0	0	0	0	0
Non Locals in Guards	3	3	0	0	0	0	0	0
UnInitialized Variable Declarations	592	0	0	0	0	0	592	0
Recursive Subprograms	1	1	0	0	0	0	0	0
Functions Returning Large Types	8	0	0	0	8	0	0	0
Functions Returning Unconstrained Types	160	0	0	0	160	0	0	0
Redeclaration Of Standard Names	1	0	0	0	0	1	0	0
Task Stack Size Not Specified	8	0	0	0	0	0	0	8
Anonymous Array Type Usage	69	0	0	0	0	69	0	0
Raised Exceptions Non Propagating	7	0	7	0	0	0	0	0
Raise Predefined Exceptions	8	0	0	0	0	8	0	0
Functions With Side Effects	56	0	0	0	0	56	0	0
Exit in While and For Loops	84	0	0	0	0	84	0	0
Total	8,288	58	1,465	0	1,879	426	1,958	376

Sample of 55 different Coding Violations Analysis Reports

Set Use Analysis

Set Use Analysis Summary	# Violations	# Serious	# ???	# Nit	# Perf	# Main1	# Main2	# Main3
Out Parameter Not Set	16	6	0	0	0	10	0	0
Use Before Set	36	4	0	0	0	24	0	0
In Out Parameter Not Set	70	?	0	0	0	70	0	0
Total	159	10	0	0	0	104	0	0

These are violations of a parameter with a mode of "in out" which either is not used as an "in" parameter in all paths, or is not set as an out parameter in all paths.
Most of these violations are OK. Some of these violations could be quite serious.

Constraint Analysis

Static Constraint Analysis Summary	# Violations	# Serious	# ???	# Nit	# Perf	# Main 1	# Main 2	# Main 3
Constraint Violations in String Objects	0	0	0	0	0	0	0	0
Constraint Violations in Integer Objects	11	11	0	0	0	0	0	0
Constraint Violations in Real Objects	0	0	0	0	0	0	0	0
Constraint Violations in Array Objects	1	1	0	0	0	0	0	0
Constraint Violations in Record Objects	63	63	0	0	0	0	0	0

Static Constraint Violation Analysis identifies those objects whose range constraints or index constraints will be violated at execution. These violations when caught during run-time will result in the Constraint_Error exception. This analysis only identifies violations that can be determined to occur statically. It will not catch violations that occur dynamically.

Would result in Run-Time Error if path tested

MITRE

More on Errors Detected

Attempt to Define Equality with renames (1 instance)

```
function "=" (Left : Sr_Da_Setup; Right : Sr_Da_Setup)
  return Boolean renames "="; -- renames self; not allowed by RM
```

Ambiguous Operation Should Be Qualified (3 instances)

“*” Defined for System_Types.Natural_Type

“*” Also Inherited

which one is it???

Pragma Inline may be ignored (4 instances)

- **Subprogram Not declared in same Declarative Part [RM_83 6.3.2(3)]**
- **May not provide intended performance characteristics**

Potentially Serious Violations - Infinite Recursion

Infinite Recursion results a task continuously expanding its stack until it runs out of resources and raises Storage_Error and possibly Tasking_Error.

- **4 Incidences flagged, such as:**

```
function "-" (Left : Transaction_Index; Right : Transaction_Count)
                              return Transaction_Index is
    begin
      return Transaction_Index (Left - Transaction_Count (Right));
    end "-";
```

Perhaps should be Transaction_Index???
Infinite Recursion was not planned here

MITRE

Potentially Serious Violations
- Functions Missing Return or Raise

- **Function paths with no return of raise statement (9 instances)**

Functions must either return a value or propagate an exception. Otherwise the exception Program_Error is raised.

- **Example:**

```
if Fd = Error then
    Error_Detected := True;
    Convert_Exception;
else
    return Fd;
end if;
```

Needs return Fd;

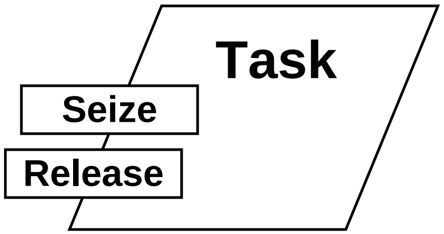
Evaluates global & raises exception for all but 1 case; Also global "Fd" subject to race condition

```
*** if exception not raised
    Then => Program_Error
```


MITRE

Potentially Serious Violations - Non-Locals in Guards

References to non-locals in guards should not be made since references will not be synchronized.




```
task Critical Resource is
     entry Seize;
     entry Release;
end Critical_Resource;
```

If Claimed Non-local (e.g., declared in specification or another package, another task could change Claimed to seize resource; also subject to race conditions

```
Task body Critical_Resource is
   Claimed: Boolean := False; -- local
begin                          -- declaration
   loop -- Simple Example with Local Guard
      select
         when not Claimed =>
            accept Seize do
               Claimed := True;
            end;
      or
         when Claimed =>
            accept Release do
               Claimed := False;
         end;
      ...
      end select;
   end loop;
end Critical_Resource;
```

3 Instances Flagged

Unintended Violations - Operator Renames

- **Operator Renames: (4 instances)**

```
function ">=" (L, R : Discm_Types.Bit_Type) return Boolean
      renames Discm_Types."<=";
```

Maintenance Violations - Anonymous Types

Anonymous Arrays have limited usefulness and complicate program modifications. For example, they cannot be formal parameters in subprograms as anonymous types have no type.

- **39 instances, such as:**

```
type Physical_Channels_Type is range 0 .. 2;

Device : array (Adapt.Physical_Channels_Type) of String (1 .. 7)

    := ("/tyCo/0", "/tyCo/1", others => "       ");
```

- **Recommend changing all:**

```
type Physical_Channels_Type is range 0 .. 2;

type Device_Type is array (Adapt.Physical_Channels_Type) of String (1 ..
7);

Device : Device_Type := ("/tyCo/0", "/tyCo/1", others => "       ");
```

Exception Gotchas
Missing Storage Error Handler For Allocators

- Any allocator can exhaust the available space for the collection, the use of allocators should be limited and the "out of memory" case handled locally. An exception handler for Storage Error should be provided in the local scope for each allocator.

143 Violations in recent analysis

Exception Gotchas

Exception handler with "*when others => null* "

- Use of a "when others" whose statement body is "null" may be inappropriate in that they catch all exceptions but provide no further processing of conditions that led to the exception.
- Exceptions should be used to trap expected problems and revert to some known safe state. These are normally classified as serious errors since should an exception be raised, processing continues which is likely degraded.

```
when others =>
   null;
```

13 Violations in recent analysis

Catches the exception, but erroneous problem is not resolved, resulting in erroneous execution, which some folks call “Graceful Degradation”

Exception Gotchas

Use of When Others in an Exception Handler

- Such handlers are a catchall and may be inappropriate in some cases. It prevents the opportunity to return the system to a known safe state based on the exception named. Typical action is to log the exception and to allow the system to perform in a degraded state. It is better practice to handle all potential exceptions explicitly. It should be noted that it is quite valuable to have such an exception handler within a looped block for tasks and main programs, as the absence there can result in a system crash.

659 Violations in recent analysis

```
when Exception_Id: others =>
    Log_Error (Package_Id & ".Next_Packet: Unhandled
           exception: " &
   Ada.Exceptions.Exception_Information (Exception_Id));
```

Logs the exception, but erroneous problem is not resolved allowing for "Graceful Degradation"

Useful to Have "when others" at Subsystem/System Level with operator Decision to Reinitialize Subsystem/System

Exception Gotchas
Raised Exceptions Non Propagating

- Explicit raising of exceptions that are caught in the local scope is similar in nature to a Goto statement. Use of exceptions in this manner represents another form of an unstructured program jump. It makes programs harder to understand, test, and modify.
- If the problem can be resolved at the local level, perhaps the use of exceptions is the wrong abstraction.

7 Violations in recent analysis

Exception Gotchas
Raise Predefined Exceptions

- Raising predefined exceptions adds confusion as to the source of the exception. The declaration of application exceptions keeps system run-time errors and application errors separate. This is considered to be a poor programming practice.

```
Raise Constraint_Error;
```

8 Violations in recent analysis all for Constraint_Error

MITRE

Exception Gotchas
Non-Visible Exception Declarations

- This is the declaration of an exception in a non-visible part of the program. Non-visible declarations can be very dangerous as they can be only handled within the scope of the declaration (except with a *when others*). Unintended propagation outside this scope may impact remote sections of the code and be a difficult error to find.

11 Violations in recent analysis

```
package QUEUE is
    procedure DEQUEUE (Object : out Object_type);
    procedure ENQUEUE (Object : in   Object_type);
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
end QUEUE;
```

and in body:

```
 OVERFLOW, UNDERFLOW : exception;
```

Declarations of exceptions in body is not useful for desired abstractions

Exception Gotchas

Exceptions that Propagate Out of Visible Scope of Subprogram

- Subprograms should not raise exceptions that are outside the visible scope of calling programs. This creates a serious problem where the exception cannot be handled by name to take the appropriate action for the raised exception. The raised exception can only be handled by a "***when others =>***" option which cannot distinguish which exception has been raised.

```
procedure Erroneous_Propagation_Demo is
   My_Exception: exception;
begin
   ...
   raise My_Exception;
exception
   when My_Exception => raise;
end Erroneous_Propagation_Demo;
```

4 Violations in recent analysis

Propagates Outside of Scope

Conclusion

- Code Analysis is an important part of providing evidence that software satisfies requirements for high-integrity systems
- Can provide Evidence to support Arguments to support Claims for High Integrity Assurance
- Code Analysis supports:
  - Identification of Coding Anomalies
  - Places where performance can be improved
  - Places where maintenance and portability can be improved
  - Identification of the strengths and weaknesses of a design
  - Identification of conformance to Coding Standards
- It is important to have an automated tool based on an interface to the compilation environment