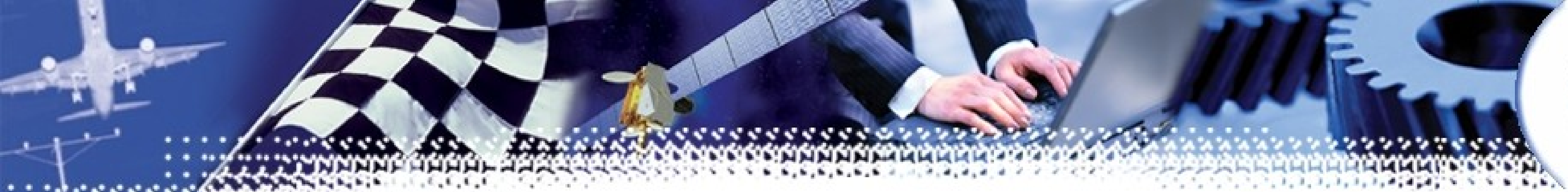
Overview of Formal Methods for Verification

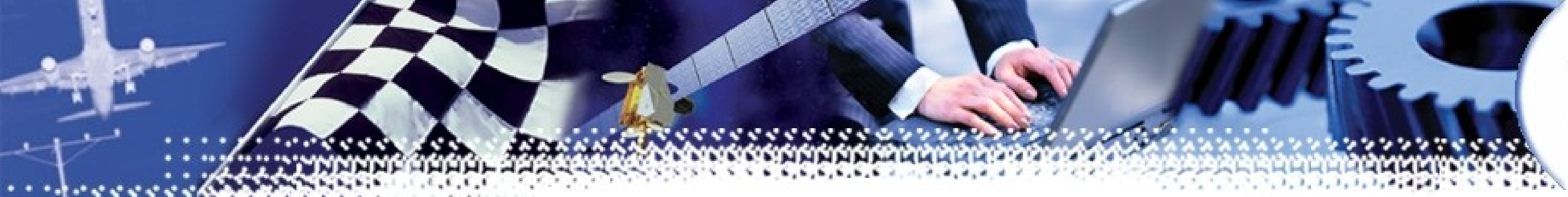
Techniques for Improving Software Quality

Dr. James J. Hunt, aicas GmbH
Systems & Software Technology Conference 2008
1st May 2008, Las Vegas, NM



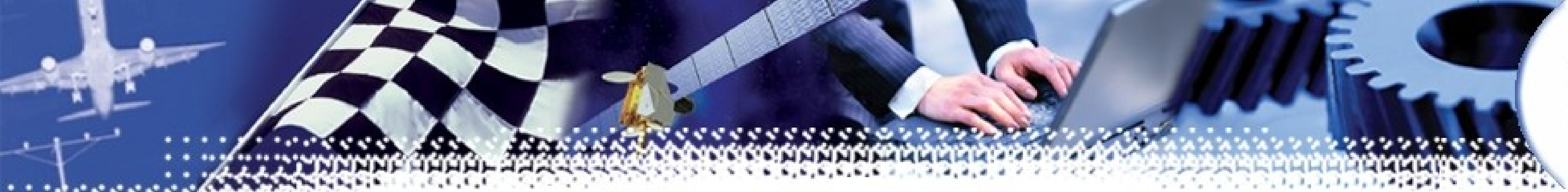
Definition of Formal Methods

analysis of software (and hardware) using rigorous mathematical methods such as calculi, logic, automata, or graph theory



Why Formal Methods?

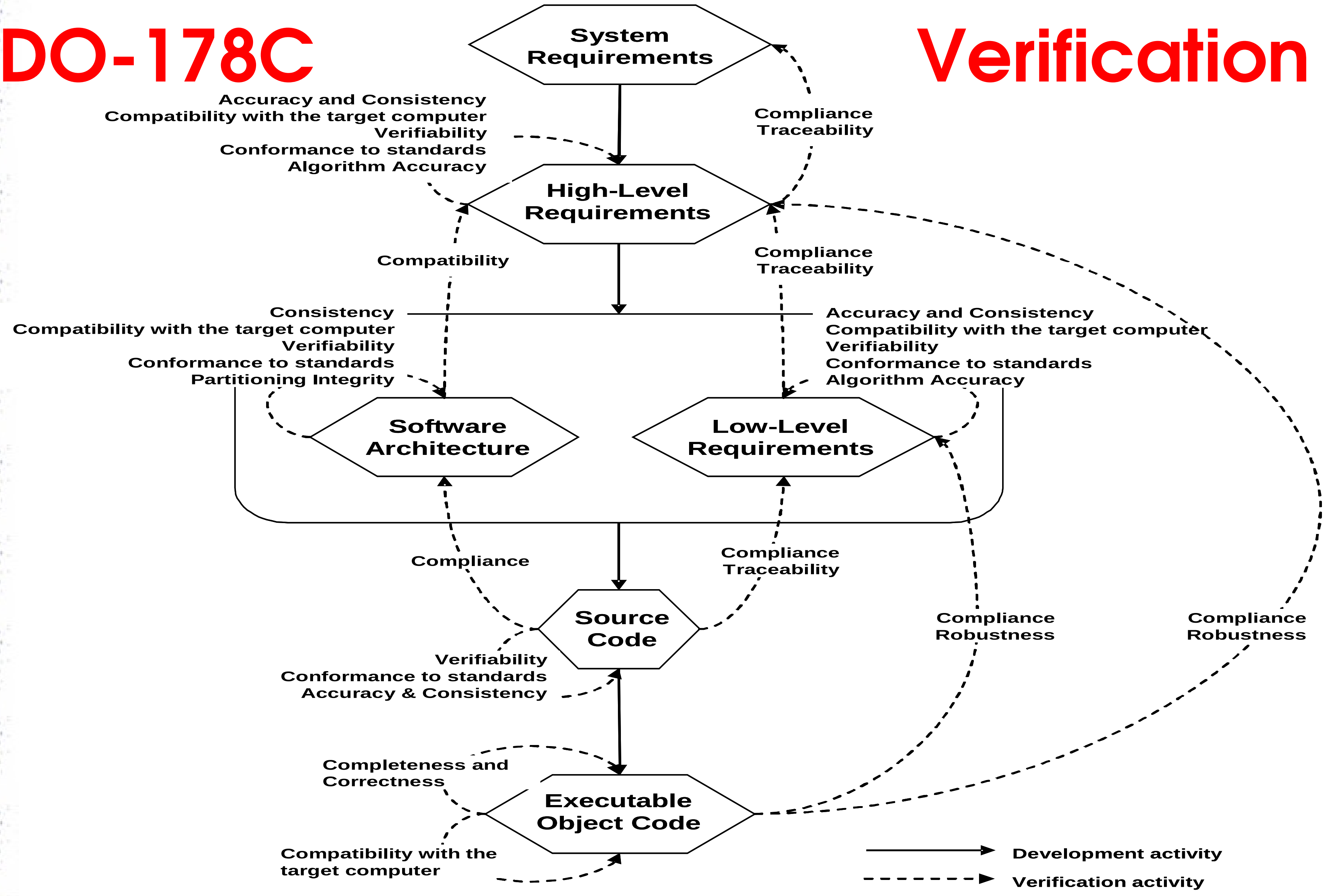
- Errors can not be tolerated in safety critical applications.
- Security is not possible without safety.
- System complexity is increasing dramatically.
- Increasingly critical decisions are being made automatically in software.
- Testing is not good enough.



Strengths and Weaknesses of Testing

- Strengths
  - Well understood
  - Mostly language independent
  - Includes execution environment
- Weaknesses
  - Hard to cover all execution paths
  - Hard to cover all possible parallel paths
  - Internal states are not visible

DO-178C Verification



System Requirements

Accuracy and Consistency
Compatibility with the target computer
Verifiability
Conformance to standards
Algorithm Accuracy

Compliance
Traceability

High-Level Requirements

Compatibility

Compliance
Traceability

Consistency
Compatibility with the target computer
Verifiability
Conformance to standards
Partitioning Integrity

Accuracy and Consistency
Compatibility with the target computer
Verifiability
Conformance to standards
Algorithm Accuracy

Software Architecture

Low-Level Requirements

Compliance

Compliance
Traceability

Source Code

Compliance
Robustness

Compliance
Robustness

Verifiability
Conformance to standards
Accuracy & Consistency

Completeness and
Correctness

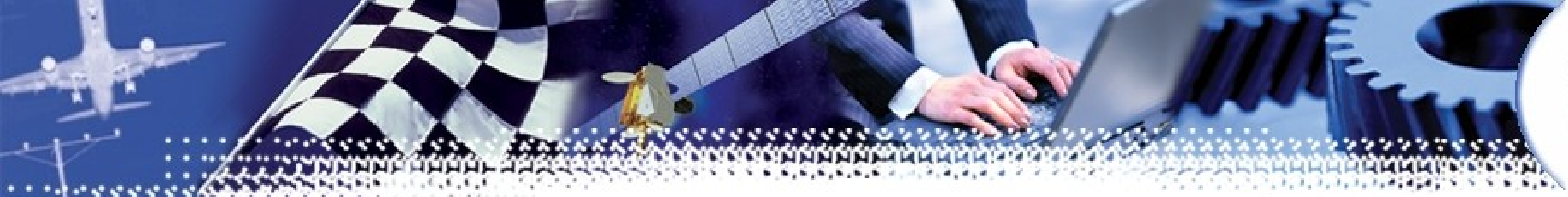
Executable Object Code

Compatibility with the
target computer

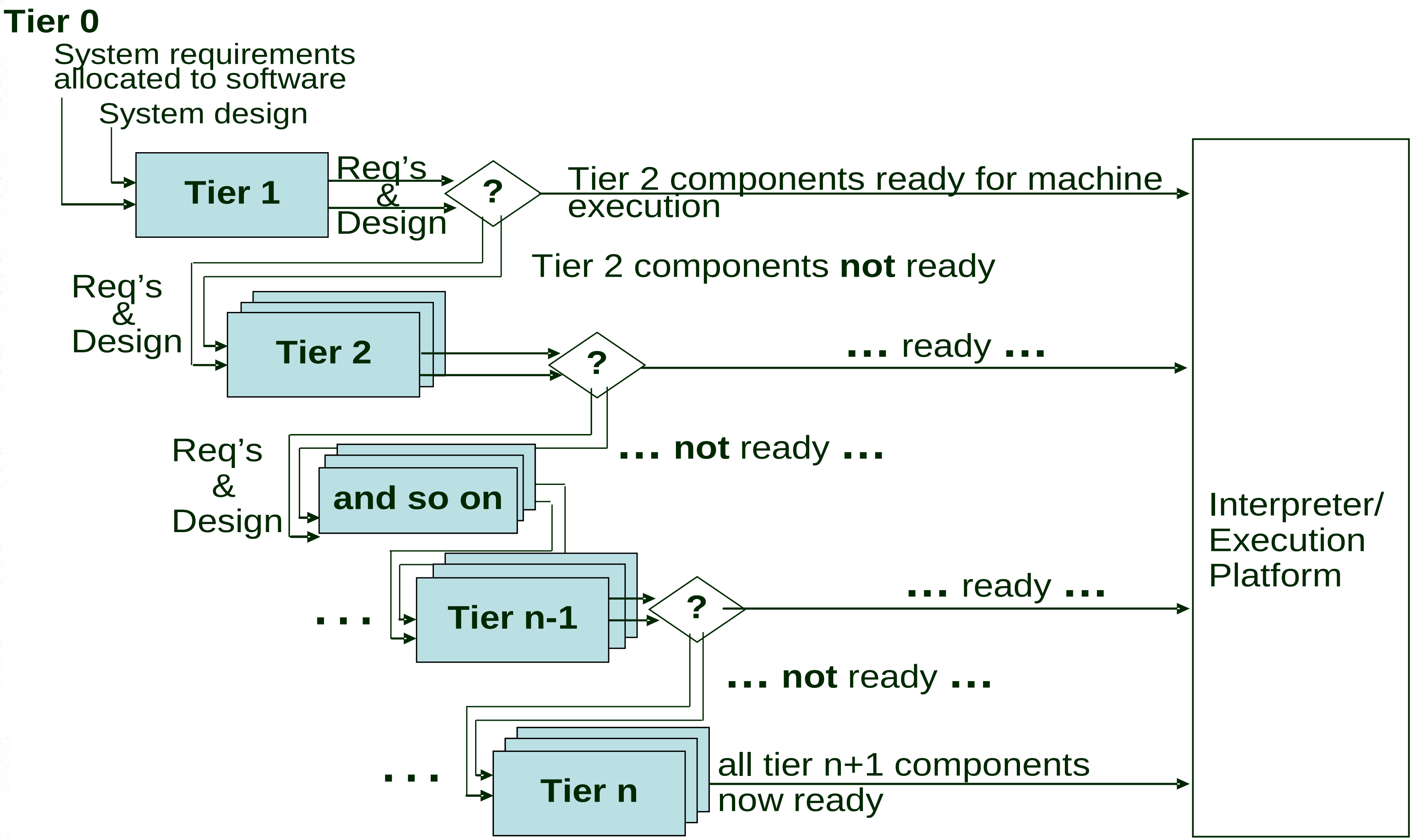
Development activity

Verification activity

Note: Requirements include Derived requirements



DO-178C Multitiered Development



Tier 0

System requirements allocated to software

System design

Tier 1

Req's & Design

?

Tier 2 components ready for machine execution

Tier 2 components **not** ready

Req's & Design

Tier 2

?

… ready **…**

… **not** ready **…**

Req's & Design

and so on

. . .

Tier n-1

?

… ready **…**

… **not** ready **…**

. . .

Tier n

all tier n+1 components now ready

Interpreter/ Execution Platform

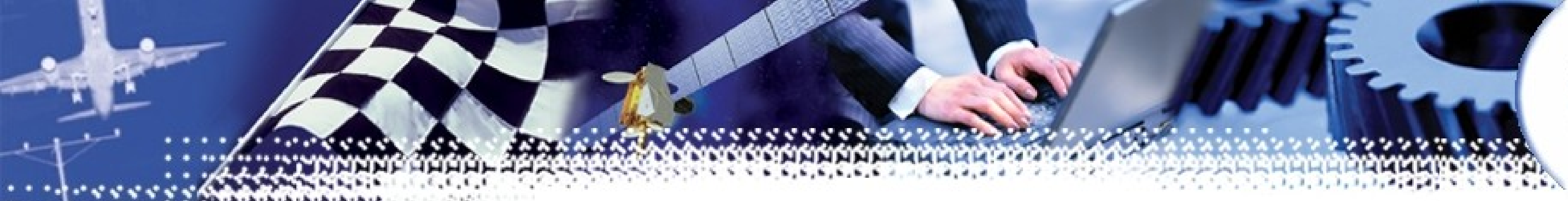
Dynamic Analysis

- Testing
- Profiling & Monitoring
  - Path Tracing
  - Call tracing
  - Time tracing
  - Bounds tracing
- Simulation
- Instrumentation
  - Race detection
  - Assertion checking
  - Aliasing detection
  - Memory analysis
  - Invariant inference

Coverage must be determined!

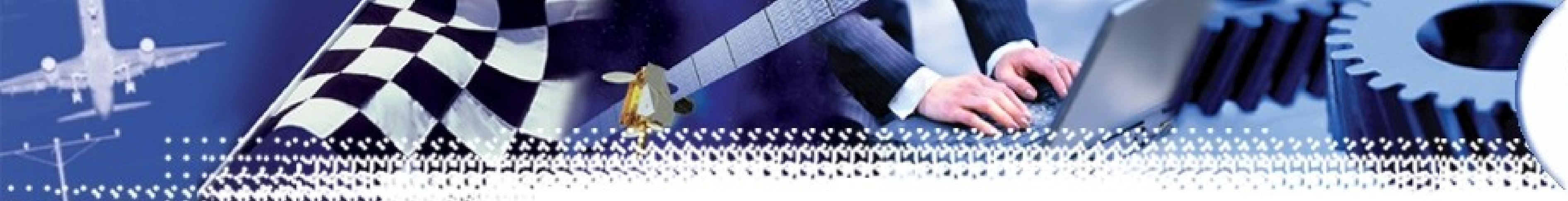
Static Analysis (Formal)

- Type Analysis
- Control Flow Analysis
- Data Flow Analysis
- Abstract Interpretation
- Symbolic Execution
- Model Checking
- Deductive verification



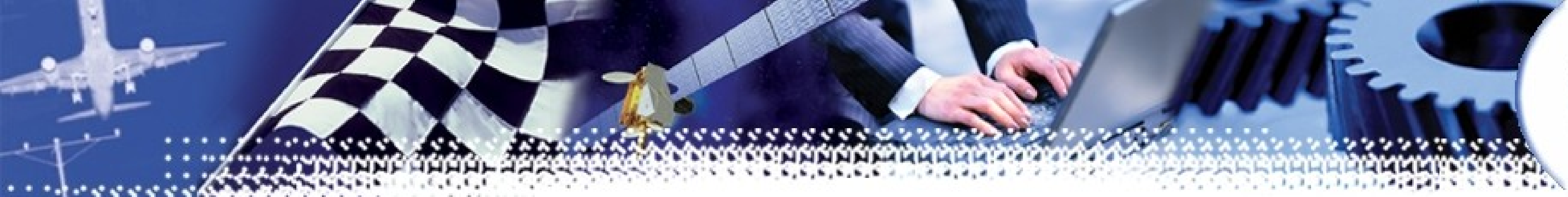
Comparison of Analysis Techniques

Static analysis	Dynamic analysis
• Abstract domain: slow but precise	• Concrete execution: slow but exhaustive
• Conservative: due to abstraction	• Precise: no approximation
• Sound: due to conservatism	• Unsound: does not generalize



Type Checking

- Most common formal method
- Attributes used to ensure consistency
  - Ensure that a given variable or field is always used as intended
  - Limits what can be assigned to a given variable or field
- Base type can be augmented with refinement types
- Checking can be done modularly



Examples of Type Checking

- Unit consistency

```
@unit("meters") int a = 4;
@unit("feet")   int b;
b = a;  /* Assignment Error */
```

- Null pointer detection
- Invariance checking
- Tool examples
  - Most modern compilers
  - JavaCop

Control Flow Analysis

- Exhaustive search of all paths through a graph representing program execution
- Code divided into basic block and links
  - Basic block is a sequence of statements or instructions that do not change control flow
  - Links can be method or function calls, branches, and links to next basic block
- Features of graph can be identified more easily

Uses of Control Flow Analysis

- Worst case execution time analysis
- Stack analysis
- Test coverage analysis
- Reachability analysis
- Numerous tools on the market

Data Flow Analysis

- Extension of control flow analysis
- Data values are propagated as well
- Fixed point algorithm
- Necessary extension for OO Languages
  - Method dispatch is data dependent
  - More precise than considering all possible subclasses at each call point

Uses of Data Flow Analysis

- all uses of control flow analysis with more precision
- Exception checking
- Memory usage
- Shared object detection
- Synchronization (deadlocks)
- Tools emerging

Detecting Runtime Errors

```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;


  int value = s.reading();

  ...
}
...
```

Detecting Runtime Errors

```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;


  int value = s.reading();

  ...
}
...
```

NullPointerException

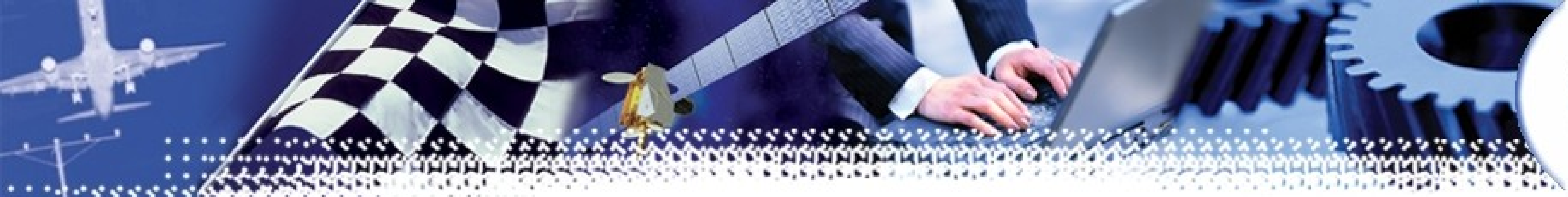
Detecting Runtime Errors

```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

NullPointerException

ClassCastException

HIJA High-Integrity Java Application



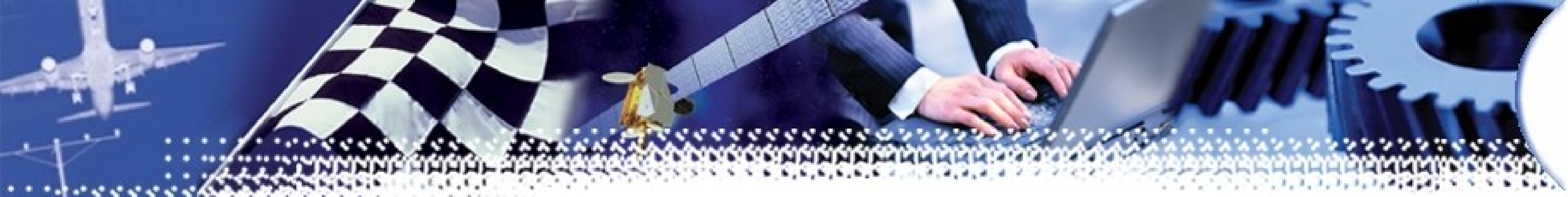
Detecting Runtime Errors

```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

NullPointerException

ClassCastException

NullPointerException



Detecting Runtime Errors

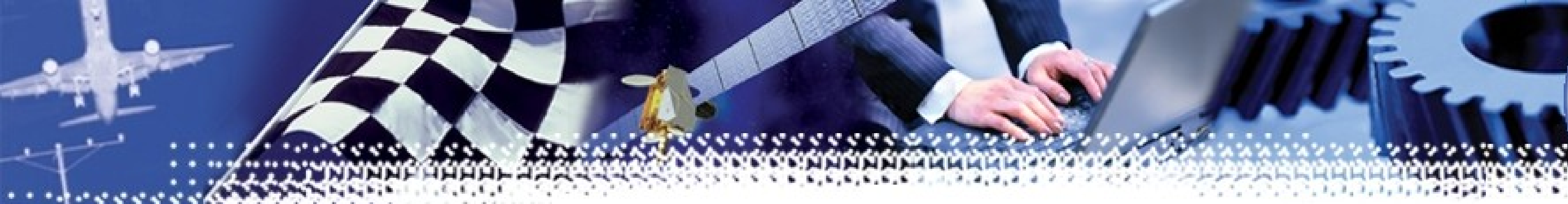
```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

device != null

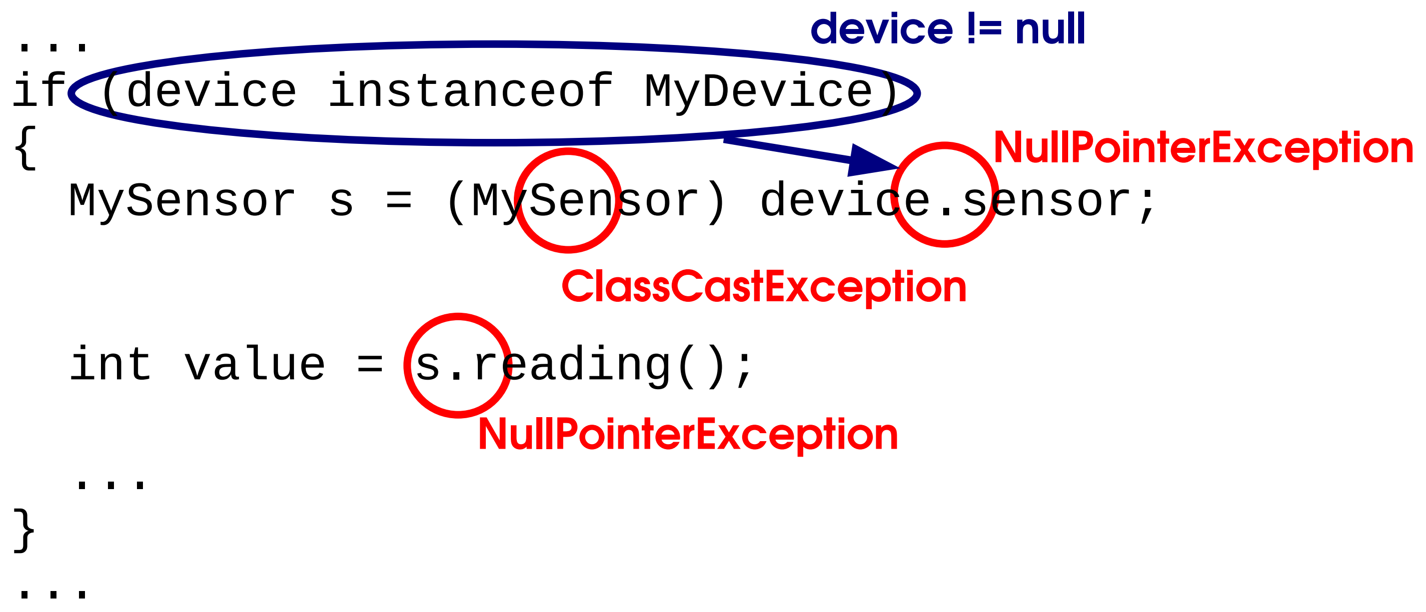
NullPointerException

ClassCastException

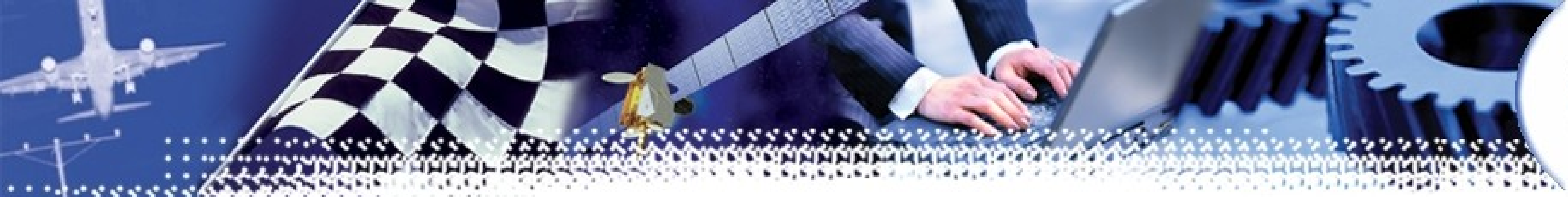
NullPointerException



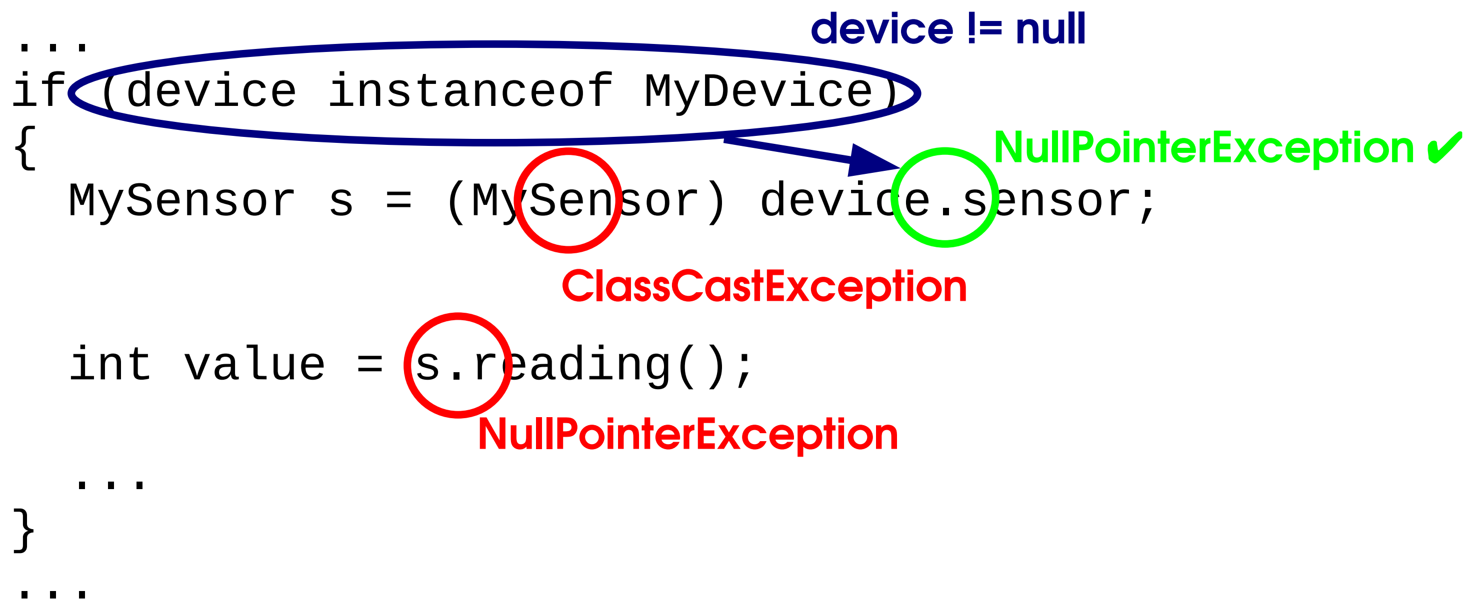
Detecting Runtime Errors



```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```



Detecting Runtime Errors



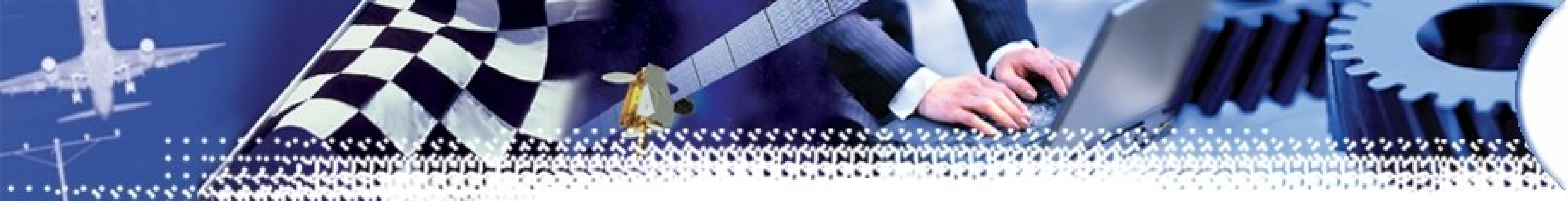
device != null

NullPointerException ✔

ClassCastException

NullPointerException

```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```



Detecting Runtime Errors

```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

NullPointerException ✔

ClassCastException

NullPointerException

Detecting Runtime Errors

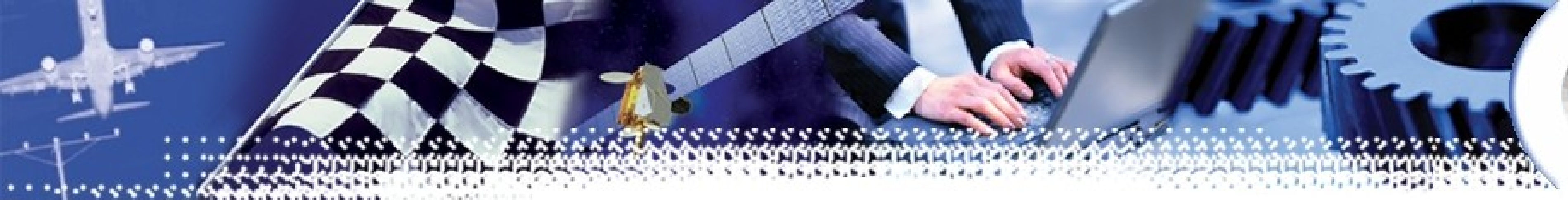
```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

NullPointerException ✔

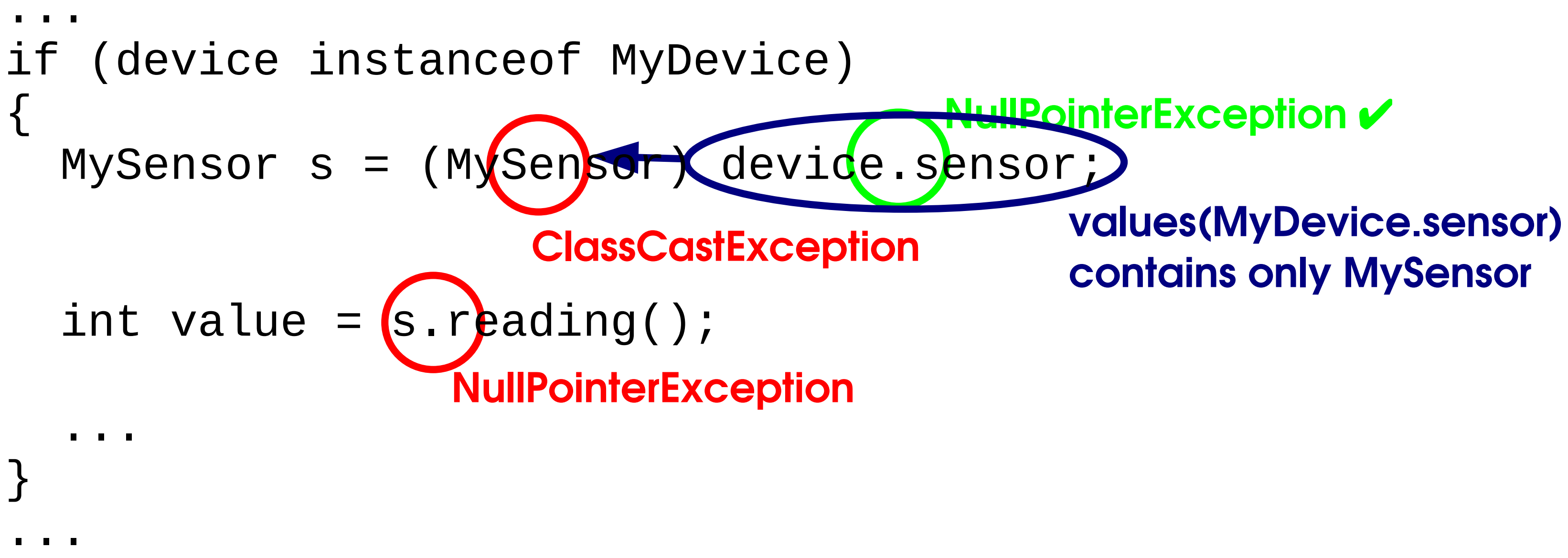
values(MyDevice.sensor)
contains only MySensor

ClassCastException

NullPointerException



Detecting Runtime Errors



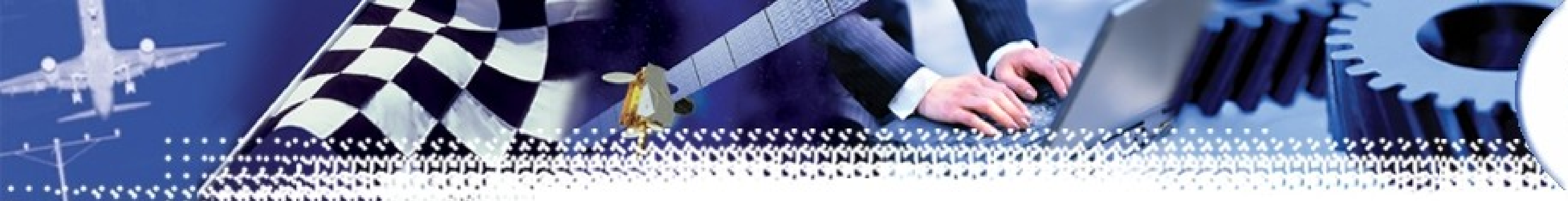
```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

NullPointerException ✔

ClassCastException

values(MyDevice.sensor)
contains only MySensor

NullPointerException



Detecting Runtime Errors

```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

NullPointerException ✔

ClassCastException ✔

values(MyDevice.sensor) contains only MySensor

NullPointerException

Detecting Runtime Errors

```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

NullPointerException ✔

ClassCastException ✔

NullPointerException

Detecting Runtime Errors

```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

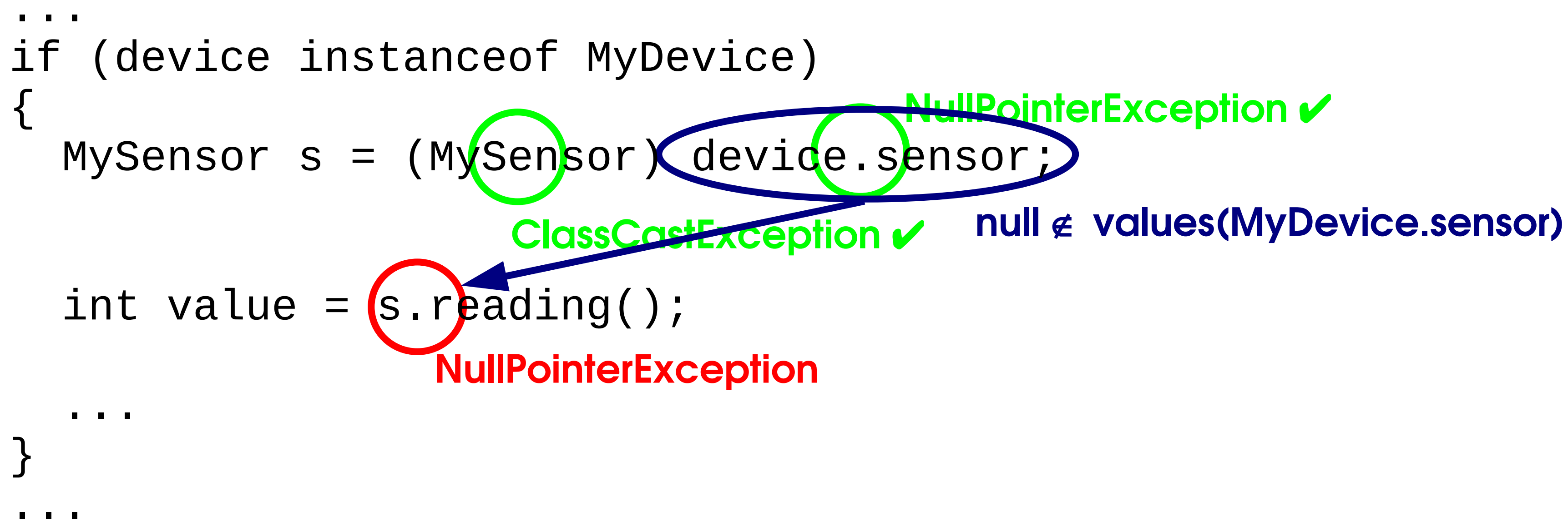
NullPointerException ✔

ClassCastException ✔

null $\notin$ values(MyDevice.sensor)

NullPointerException

Detecting Runtime Errors



```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

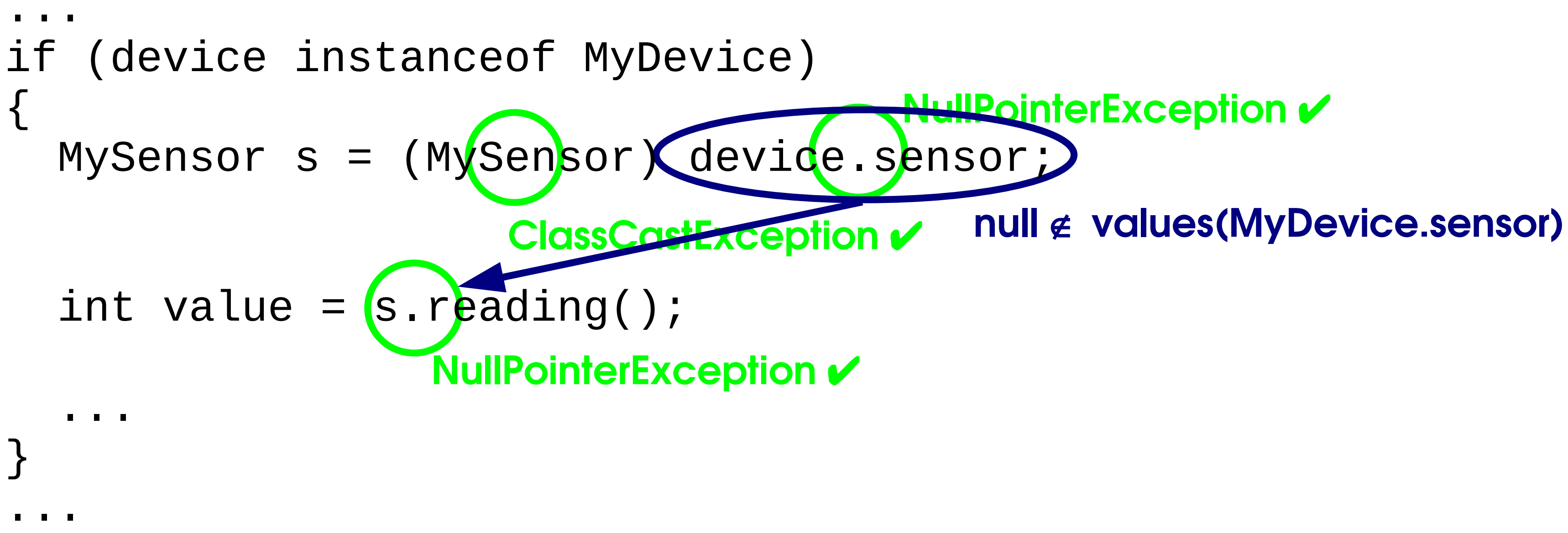
NullPointerException ✔

ClassCastException ✔

null ∉ values(MyDevice.sensor)

NullPointerException

Detecting Runtime Errors



```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

NullPointerException ✔

null $\notin$ values(MyDevice.sensor)

ClassCastException ✔

NullPointerException ✔

HIJA High-Integrity Java Application

Detecting Runtime Errors

```
...
if (device instanceof MyDevice)
{
  MySensor s = (MySensor) device.sensor;

  int value = s.reading();

  ...
}
...
```

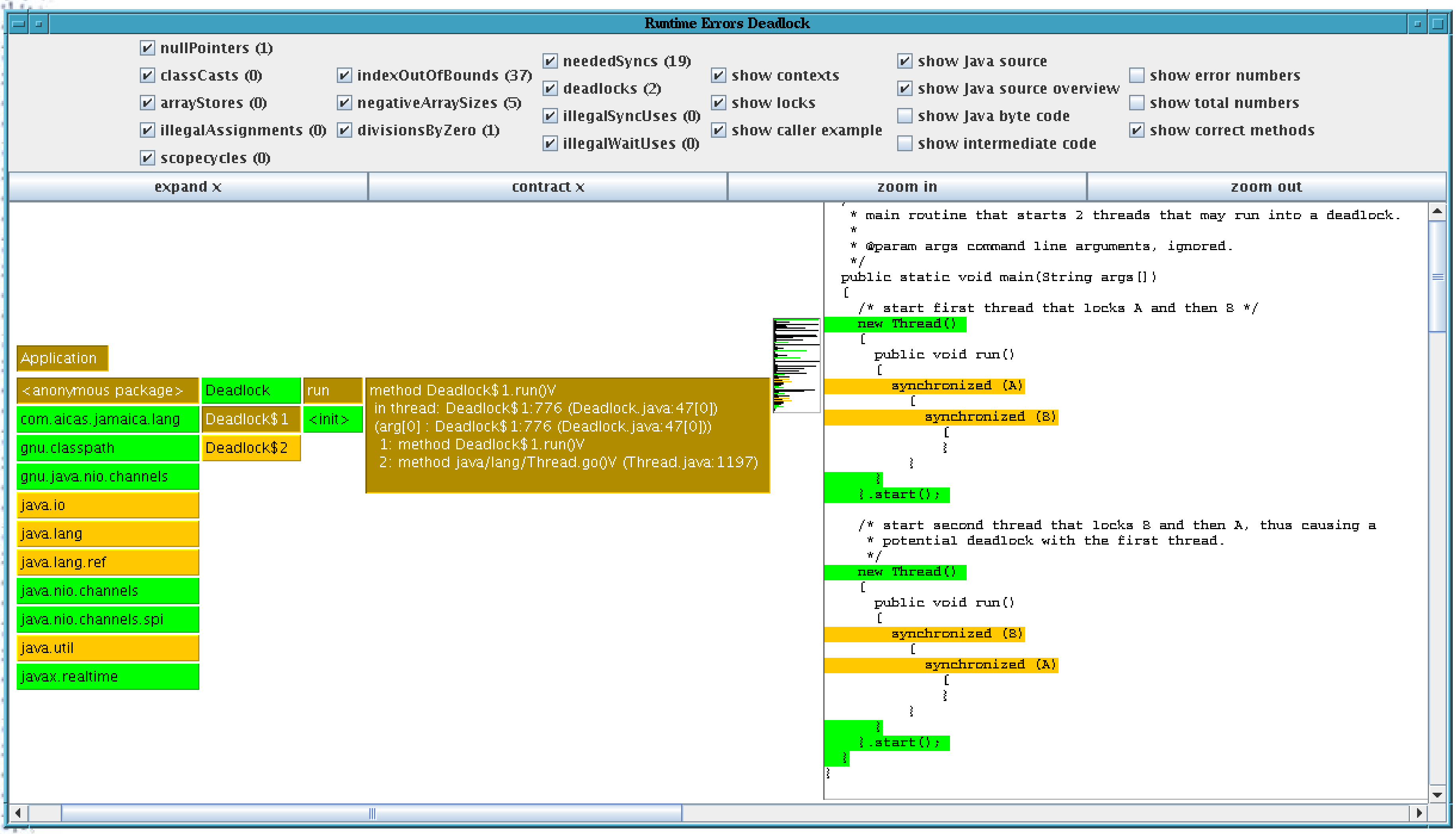
NullPointerException ✔

ClassCastException ✔

NullPointerException ✔

HIJA High-Integrity Java Application

Example DFA Tool from aicas



Runtime Errors Deadlock

☑ nullPointers (1)
☑ classCasts (0)
☑ arrayStores (0)
☑ illegalAssignments (0)
☑ scopecycles (0)
☑ indexOutOfBounds (37)
☑ negativeArraySizes (5)
☑ divisionsByZero (1)
☑ neededSyncs (19)
☑ deadlocks (2)
☑ illegalSyncUses (0)
☑ illegalWaitUses (0)
☑ show contexts
☑ show locks
☑ show caller example
☑ show Java source
☑ show Java source overview
☐ show Java byte code
☐ show intermediate code
☐ show error numbers
☐ show total numbers
☑ show correct methods

```
method Deadlock$1.run()V
 in thread: Deadlock$1:776 (Deadlock.java:47[0])
 (arg[0] : Deadlock$1:776 (Deadlock.java:47[0]))
  1: method Deadlock$1.run()V
  2: method java/lang/Thread.go()V (Thread.java:1197)
```

```
 * main routine that starts 2 threads that may run into a deadlock.
 *
 * @param args command line arguments, ignored.
 */
public static void main(String args[])
{
  /* start first thread that locks A and then B */
  new Thread()
  {
    public void run()
    {
      synchronized (A)
        {
          synchronized (B)
            {
            }
        }
    }
  }.start();

  /* start second thread that locks B and then A, thus causing a
   * potential deadlock with the first thread.
   */
  new Thread()
  {
    public void run()
    {
      synchronized (B)
        {
          synchronized (A)
            {
            }
        }
    }
  }.start();
 }
}
```

Abstract Interpretation

- A theory of sound approximation of the semantics of a program
- Concrete state and operations mapped to abstract state and operations
- Based on monotonic functions over ordered sets, especially lattices
- Can be viewed as a partial execution of a program to gain semantic information without performing all calculations

Uses of Abstract Interpretation

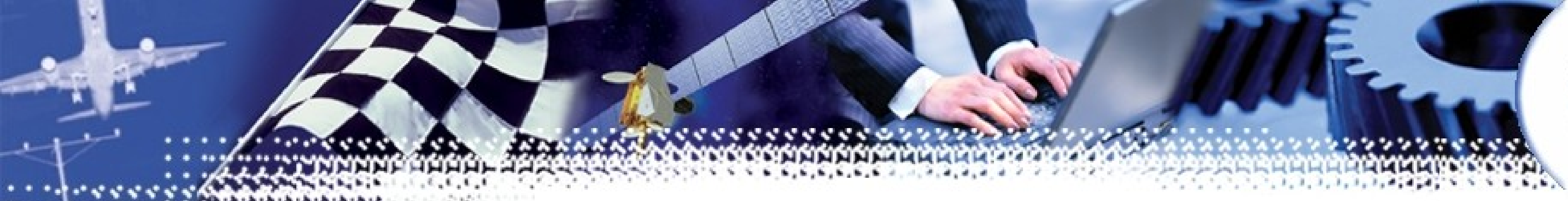
- Liveliness
- Race conditions
- Simultaneous access
- Tools

Academic

- ASTRÉE (CNRS)
- Airac (SNU)

Commercial

- CodeHawk (KT)
- PAG (AbsInt)
- PolySpace



Symbolic Execution

- Also known as Symbolic Simulation
- Considers all possible execution paths
- Many possible executions of a system are considered simultaneously
- Models concrete semantics of all primitive operations (calculus)
- Set of values instead of concrete value
- Base for other techniques

Model Checking

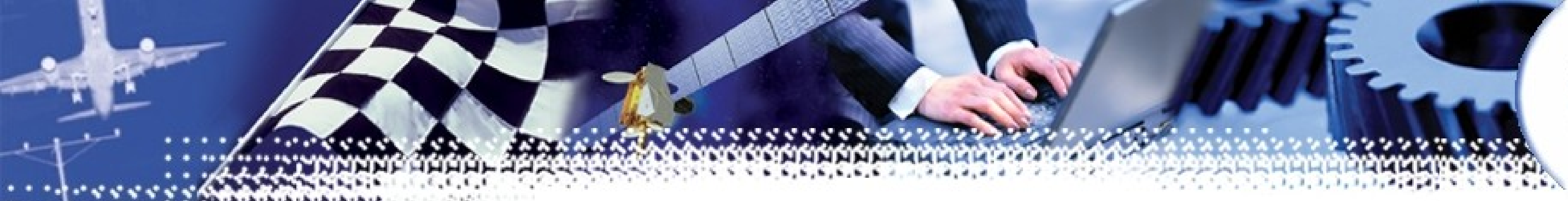
- A variant of abstract interpretation
- Abstraction is a finite state machine
- Some aspect of program is modeled as states and transitions in state machine
- Both simulation and reachability analysis can be performed on state machine
- Error states are used to detect faults

Examples of Model Checking

- Model consistency (e.g., UML models)
- Checking parallel execution
- Some runtime errors (entry into a state)
- Numerous Tools

  Model Level
  - SPIN
  - PROSPER
  - Uppaal

  Java Programs
  - Pathfinder

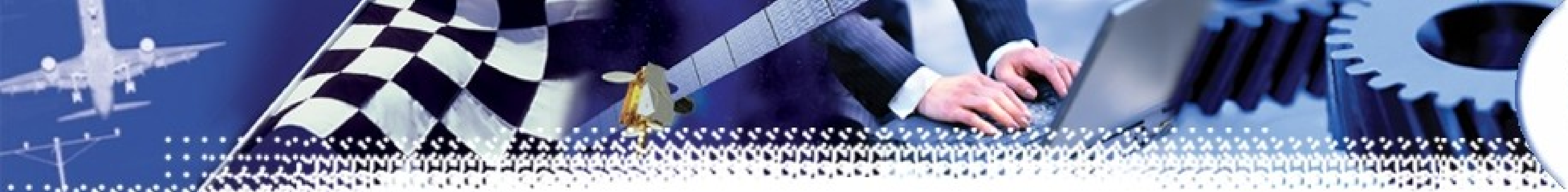


Deductive Verification

- Uses formal specification language
  - Preconditions
  - Postconditions
  - invariants
- Checks program code against specification
- Based on theorem proving, Hoare Logic, and Liskov Substitution Principle

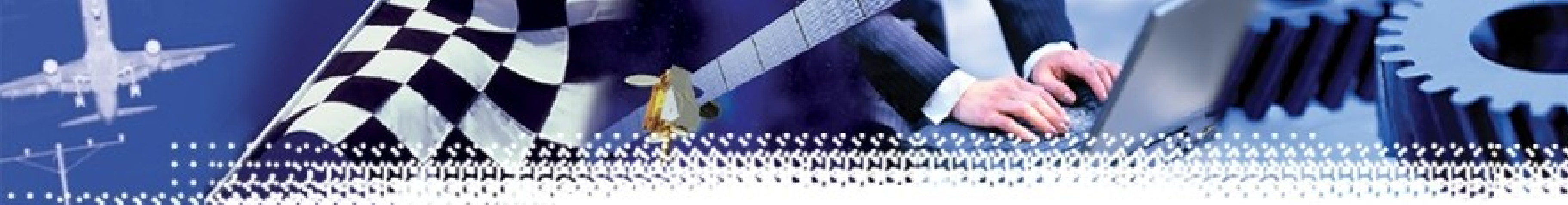
Formal Specification Languages

- Z notation (Specification)
- B method (Refinement)
- Object Constraint Language (OCL)
- Java Modeling Language (JML)



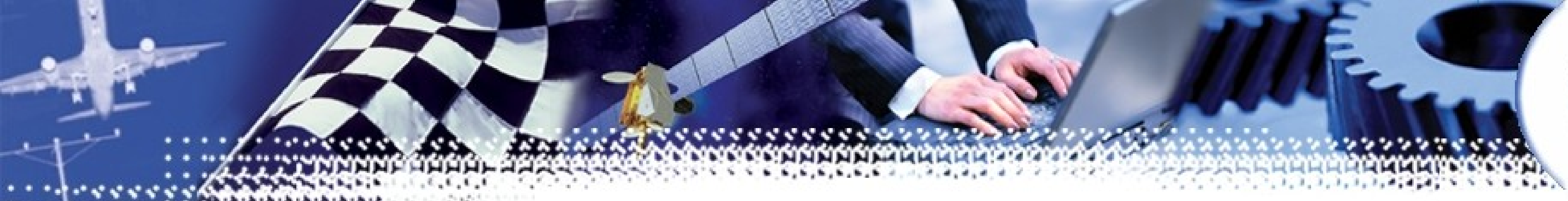
Examples of Deductive Verification

- Proving that a given Java method respects its post conditions given it preconditions
- Showing that invariants are respected
- Numerous Tools for Java (JML)
  - ESC/Java2 (Simplify)
  - JACK (B-Method, Simplify, PVS, Coq)
  - KeY (Dynamic Logic) (OCL & JML)



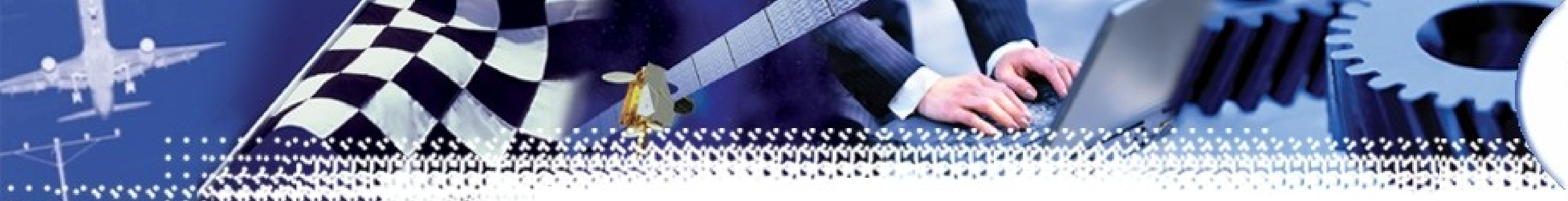
Which Methods to use?

- Depends on what is to be checked and when in the development process
- Each tools has its strength and weaknesses and point of application
- A combination of tools works best
- Choice of programming language has an impact, e.g., Java is easier to analyze than C and C++.



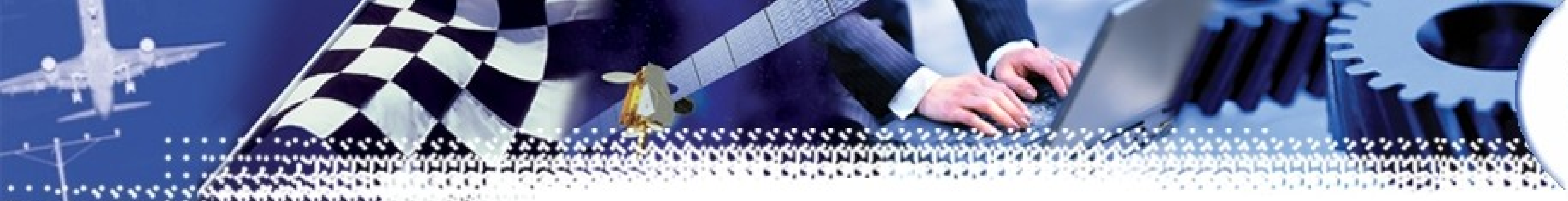
Formal Code Generation

- Based on abstract models and compiler theory
- Specify what instead of how
- Correct by construction
- Methods
  - Rule based selection and composition
  - Graph transformations
  - Algebraic transformations



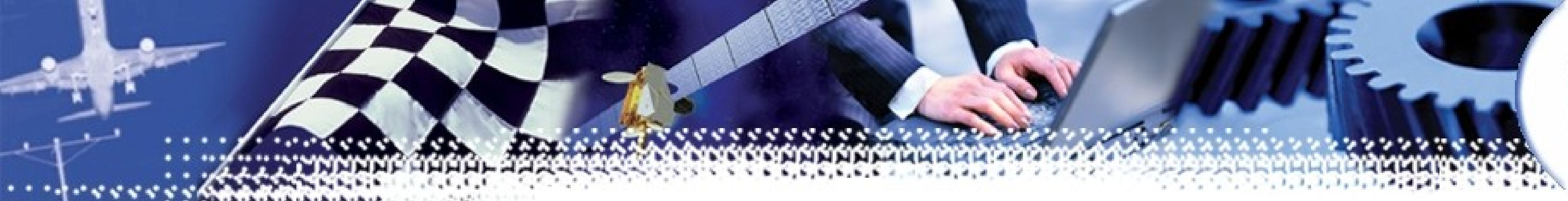
Code from Abstract State Machine

- System is described as a finite state machine
- State machine description is translated into program code
- Generation can be done completely automatically
- Language less powerful than touring machine



Code from Metalanguage

- Program described in terms of transformations (what)
- Translator chooses between implementations (how)
- Transformations are composable
- Correctness must be proven for each implementation of each transformation
- User interaction often required



Formal Code Generation Tools

- State Machine
  - SCADE (Esterel)
- UML & Metalanguage
  - Perfect Developer (Escher Technologies)
- Meta Language
  - Specware (Kestrel Technology)
  - ACL2 (University of Texas)

Caveats

- Analysis is limited to information at hand
  - Check internal consistency
  - Compare alternative descriptions, e.g., code against specification
  - State explosion must be managed
- Code generation
  - may not produce efficient programs
  - Individual transitions must be checked
- Neither can fully replace testing

Conclusion

- Formal methods can drastically improve program quality.
- Can validate code against requirements.
- Different techniques for different aspects of interest.
- Can be combined to be more complete.
- More effective than test alone.
- Some testing will always be necessary.

Contact Information

aicas GmbH
Haid-und-Neu-Straße 18
D-76139 Karlsruhe
+49 721 663 968 22

aicas incorporated
69 West Rock Ave.
New Haven, CT 06515
+1 203 676 9807 jjh@aicas.com