Formal Specification and Verification of Avionics Software

Claus Wonnemann

June 7th, 2006
Outline

1. **Introduction**
   - Software in the avionics domain
   - Certification requirements
   - Object-oriented technologies

2. **Specification of the Java Flight Manager**
   - Flight Management
   - The Java Flight Manager
   - Specification

3. **Runtime Assertion Checking and Verification**
   - Runtime Assertion Checking
   - Verification

4. **Conclusions**
   - Wrap-up
Commercial airplanes feature a high degree of computerization.

- Many onboard computer systems are safety-critical.
  - Navigation and Pilotage Assistance.
  - Engine Control and Breaking Systems.
  - *Fly-by-Wire.*

- Airborne software products must be officially certified.
Software in the avionics domain

- Commercial airplanes feature a high degree of computerization.
- Many onboard computer systems are safety-critical.
  - Navigation and Pilotage Assistance.
  - Engine Control and Breaking Systems.
  - Fly-by-Wire.
- Airborne software products must be officially certified.
Commercial airplanes feature a high degree of computerization.

Many onboard computer systems are safety-critical.

- Navigation and Pilotage Assistance.
- Engine Control and Breaking Systems.
- *Fly-by-Wire.*

Airborne software products must be officially certified.
Software in the avionics domain

- Commercial airplanes feature a high degree of computerization.
- Many onboard computer systems are safety-critical.
  - Navigation and Pilotage Assistance.
  - Engine Control and Breaking Systems.
  - *Fly-by-Wire.*
- Airborne software products must be officially certified.
Software in the avionics domain

Commercial airplanes feature a high degree of computerization.

Many onboard computer systems are safety-critical.

- Navigation and Pilotage Assistance.
- Engine Control and Breaking Systems.
- *Fly-by-Wire.*

Airborne software products must be officially certified.
Software in the avionics domain

- Commercial airplanes feature a high degree of computerization.
- Many onboard computer systems are safety-critical.
  - Navigation and Pilotage Assistance.
  - Engine Control and Breaking Systems.
  - Fly-by-Wire.
- Airborne software products must be officially certified.
RTCA/DO-178B

- RTCA/DO-178B is the major requirements specification.
  - “Software Considerations in Airborne Systems and Equipment Certification.”
- Adopted as an official guideline by the FAA in 1993.
- Airborne software products must comply with stated objectives.
- Considers *Structured Programming*, not *Object-Orientation*.

Objected-oriented technologies (OOT) are much less common in the avionics domain.
RTCA/DO-178B

- RTCA/DO-178B is the major requirements specification.
  - “Software Considerations in Airborne Systems and Equipment Certification.”
- Adopted as an official guideline by the FAA in 1993.
- Airborne software products must comply with stated objectives.
- Considers *Structured Programming*, not *Object-Orientation*.

Object-oriented technologies (OOT) are much less common in the avionics domain.
Object-Oriented Technology in Aviation

- OOT is in many respects different from other approaches.

For instance...

- RTCA/DO-178B requires the elimination of unused code.
  - Polymorphism and dynamic dispatch obviously complicate this task.

- Object-Oriented Technology in Aviation-Program (OOTiA) addresses related issues and concerns.
Object-Oriented Technology in Aviation

- OOT is in many respects different from other approaches.

For instance...

- RTCA/DO-178B requires the elimination of unused code.
  - Polymorphism and dynamic dispatch obviously complicate this task.

- Object-Oriented Technology in Aviation-Program (OOTiA) addresses related issues and concerns.
Object-Oriented Technology in Aviation

- OOT is in many respects different from other approaches.

For instance...

- RTCA/DO-178B requires the elimination of unused code.
  - Polymorphism and dynamic dispatch obviously complicate this task.

- Object-Oriented Technology in Aviation-Program (OOTiA) addresses related issues and concerns.
Considerations and recommended techniques were compiled in a preliminary handbook.

The major issues include:

- Subtypes and Subclasses
- Memory Management
- Dead and Deactivated Code
Elements of OOTiA

- Considerations and recommended techniques were compiled in a preliminary handbook.
- The major issues include:
  - Subtypes and Subclasses
  - Memory Management
  - Dead and Deactivated Code
Considerations and recommended techniques were compiled in a preliminary handbook.

The major issues include:
- Subtypes and Subclasses
- Memory Management
- Dead and Deactivated Code

The OOTiA-Handbook repeatedly mentions Design by Contract and Formal Methods as suggested methodologies for software development in the avionics domain.
Flight Management

- A Flight Manager is part of the onboard navigational equipment.

- A major task is the computation of trajectories.
  - Must comply with air traffic rules.
  - Has to consider the aircraft’s agility.
  - Should be efficient and economic.
  - Further constraints.

- A reliable operation is critical for a safe flight.
  - A failure is considered hazardous by RTCA/DO-178B (2nd highest category).
Flight Management

A Flight Manager is part of the onboard navigational equipment.

A major task is the computation of trajectories.
- Must comply with air traffic rules.
- Has to consider the aircraft’s agility.
- Should be efficient and economic.
- Further constraints.

A reliable operation is critical for a safe flight.
- A failure is considered hazardous by RTCA/DO-178B (2nd highest category).
Flight Management

A Flight Manager is part of the onboard navigational equipment.

A major task is the computation of trajectories.
- Must comply with air traffic rules.
- Has to consider the aircraft’s agility.
- Should be efficient and economic.
- Further constraints.

A reliable operation is critical for a safe flight.
- A failure is considered *hazardous* by RTCA/DO-178B (2nd highest category).
The Java Flight Manager

- Developed by Thales Avionics in Toulouse.
  - For research purposes:
    - Rapid prototyping of new features.
    - Investigation of OOT-related risks and benefits.
    - Java in the avionics domain.

- The lateral module is subject to the formal specification.
  - About 70 classes.
  - Major phenomena have been specified.
The Java Flight Manager

- Developed by Thales Avionics in Toulouse.
- For research purposes:
  - Rapid prototyping of new features.
  - Investigation of OOT-related risks and benefits.
  - Java in the avionics domain.

- The lateral module is subject to the formal specification.
  - About 70 classes.
  - Major phenomena have been specified.

Developed by Thales Avionics in Toulouse.

For research purposes:
- Rapid prototyping of new features.
- Investigation of OOT-related risks and benefits.
- Java in the avionics domain.

The lateral module is subject to the formal specification.
- About 70 classes.
- Major phenomena have been specified.
Developed by Thales Avionics in Toulouse.

For research purposes:
- Rapid prototyping of new features.
- Investigation of OOT-related risks and benefits.
- Java in the avionics domain.

The lateral module is subject to the formal specification.
- About 70 classes.
- Major phenomena have been specified.
The lateral module

- Computes the lateral part of a trajectory.
- The trajectory construction is done in three subsequent steps:
  - Stage 1: A loose set of legs.
  - Stage 2: Fixed positions connected by straight lines.
  - Stage 3: The final trajectory.
The lateral module

- Computes the lateral part of a trajectory.
- The trajectory construction is done in three subsequent steps:
  - Stage 1: A loose set of legs.
  - Stage 2: Fixed positions connected by straight lines.
  - Stage 3: The final trajectory.
The lateral module

- Computes the lateral part of a trajectory.
- The trajectory construction is done in three subsequent steps:
  - Stage 1: A loose set of legs.
  - Stage 2: Fixed positions connected by straight lines.
  - Stage 3: The final trajectory.
The lateral module

- Computes the lateral part of a trajectory.
- The trajectory construction is done in three subsequent steps:
  - Stage 1: A loose set of legs.
  - Stage 2: Fixed positions connected by straight lines.
  - Stage 3: The final trajectory.
The lateral module

- Computes the lateral part of a trajectory.
- The trajectory construction is done in three subsequent steps:
  - Stage 1: A loose set of legs.
  - Stage 2: Fixed positions connected by straight lines.
  - Stage 3: The final trajectory.
The Specification

- The specification is usually model-based.
  - Contracts are based on abstract model.
  - Well supported by model fields in JML.
- It refrains from using JML’s specification library.
  - Java types are usually sufficient.
  - Heavy burden for verification.
- Emphasis on invariants instead of contracts.
  - Reflect characteristics of entity.

Some benefits

- Formal specs convey an unambiguous description.
- Enforce reflections on a system’s characteristics.
- Provide access for CASE tools.
The Specification

- The specification is usually model-based.
  - Contracts are based on abstract model.
  - Well supported by model fields in JML.
- It refrains from using JML’s specification library.
  - Java types are usually sufficient.
  - Heavy burden for verification.
- Emphasis on invariants instead of contracts.
  - Reflect characteristics of entity.

Some benefits

- Formal specs convey an unambiguous description.
- Enforce reflections on a system’s characteristics.
- Provide access for CASE tools.
The Specification

- The specification is usually model-based.
  - Contracts are based on abstract model.
  - Well supported by model fields in JML.
- It refrain from using JML’s specification library.
  - Java types are usually sufficient.
  - Heavy burden for verification.
- Emphasis on invariants instead of contracts.
  - Reflect characteristics of entity.

Some benefits

- Formal specs convey an unambiguous description.
- Enforce reflections on a system’s characteristics.
- Provide access for CASE tools.
The Specification

- The specification is usually model-based.
  - Contracts are based on abstract model.
  - Well supported by model fields in JML.
- It refrains from using JML’s specification library.
  - Java types are usually sufficient.
  - Heavy burden for verification.
- Emphasis on invariants instead of contracts.
  - Reflect characteristics of entity.

Some benefits

- Formal specs convey an unambiguous description.
- Enforce reflections on a system’s characteristics.
- Provide access for CASE tools.
Example I: A Leg Transition

**Model fields**

Fix specPivotFix;
Fix specTurnStart;
Fix specTurnEnd;
Fix specCircleCenter;
double specAngularDistance;
...

**Invariants**

\[\begin{align*}
\text{specDirection} &= \text{specLogicalDirection} \implies \\
\text{specTurnStart} &\leq \text{specPivotFix} \approx_{fp} \text{specBearingToFix} \\
\text{specDirection} &\neq \text{specLogicalDirection} \implies \\
\text{specTurnStart} &\leq \text{specPivotFix} \approx_{fp} (\text{specBearingToFix} + 180)_{360} \\
\text{specTurnStart} &\rightarrow \text{specPivotFix} \approx_{fp} \text{specTAD}
\end{align*}\]

...
Example I: A Leg Transition

Model fields

Fix specPivotFix;
Fix specTurnStart;
Fix specTurnEnd;
Fix specCircleCenter;
double specAngularDistance;
...

Invariants

specDirection = specLogicalDirection \implies
              specTurnStart \triangleleft specPivotFix \approx_{ fp } specBearingToFix
specDirection \neq specLogicalDirection \implies
              specTurnStart \triangleleft specPivotFix \approx_{ fp } (specBearingToFix + 180)_{360}
specTurnStart \rightarrow specPivotFix \approx_{ fp } specTAD
...

Example I: A Leg Transition

Model fields

Fix specPivotFix;
Fix specTurnStart;
Fix specTurnEnd;
Fix specCircleCenter;
double specAngularDistance;
...

Invariant

specDirection = specLogicalDirection \implies
    specTurnStart \triangleleft specPivotFix \approx_{fp} specBearingToFix
specDirection \neq specLogicalDirection \implies
    specTurnStart \triangleleft specPivotFix \approx_{fp} (specBearingToFix + 180)_{360}
specTurnStart \rightarrow specPivotFix \approx_{fp} specTAD
...

Claus Wonnemann
Formal Specification and Verification of Avionics Software
The first invariant in JML

```java
public instance invariant
    (specDirection == specLogicalDirection) ==> 
    Cmp.apprEq( BasicGeo.computeBD(
        specTurnStart.specLatitude, 
        specTurnStart.specLongitude, 
        specPivotFix.specLatitude, 
        specPivotFix.specLatitude)[0], 
        specBearingToFix);
```

- JML-Specs are often lengthy and verbose.
  - Difficult to comprehend and maintain.
  - Facilitates introduction of errors.
- Leads to programming-style specifications.
  - Many typecasts.
  - Numerous method method calls.
The first invariant in JML

public instance invariant
  (specDirection == specLogicalDirection) ==> 
  Cmp.apprEq( BasicGeo.computeBD(
    specTurnStart.specLatitude, 
    specTurnStart.specLongitude, 
    specPivotFix.specLatitude, 
    specPivotFix.specLongitude)[0], 
    specBearingToFix );

- JML-Specs are often lengthy and verbose.
  - Difficult to comprehend and maintain.
  - Facilitates introduction of errors.
- Leads to programming-style specifications.
  - Many typecasts.
  - Numerous method calls.
Example II: A double-linked tree

- Used as a hierarchic route representation.
- Properties of the tree should be expressed by invariants.

For each node must hold:
- The parent reference of all children must point to this.
- If there is a parent node, it must have a child reference to this.
Example II: A double-linked tree

- Used as a hierarchic route representation.
- Properties of the tree should be expressed by invariants.

For each node must hold:
- The parent reference of all children must point to this.
- If there is a parent node, it must have a child reference to this.
Problems with these invariants

- The atomicity of “structural” operations cannot be ensured.
- The use of `Object.clone()` to clone a node violates the invariants.

```java
public boolean add (Object o) {
    ((TreePart)o).setParent(this);
    return super.add(o);
}
```

- `setParent()` and `super.add()` have both public visibility.
- Invariants get violated according to JML’s *Visible State* semantics.
  - Invariants hold with KeY’s *Observable State* semantics.
- Can be fixed through refactoring.
  - Inheritance relation to superclass has to be broken.
Problems with these invariants

- The atomicity of "structural" operations cannot be ensured.
- The use of `Object.clone()` to clone a node violates the invariants.

- The first attempt to adjust the cloned object’s references breaks the invariant.
- A clone method can be implemented without `Object.clone()`.
- Nevertheless: Visible State semantics too restrictive?
Runtime Assertion Checking (RAC) allows to test constraints at runtime.

The JML-Distribution includes a RAC-Compiler (jmlc).

**Benefits**

- An easy means to test both the code and the specification.
  - Specification errors are usually quickly detected through tests.
- Allows a clear separation of code and tests.
  - Tests can be switched off at will to improve performance.
  - No further defensive checks within the code necessary.
- An additional benefit at no extra cost.
  - If a specification exists, no further effort is necessary.
  - jmlc accepts all legal JML and Java code.
Runtime Assertion Checking

- Runtime Assertion Checking (RAC) allows to test constraints at runtime.
- The JML-Distribution includes a RAC-Compiler (jmlc).

Benefits

- An easy means to test both the code and the specification.
  - Specification errors are usually quickly detected through tests.
- Allows a clear separation of code and tests.
  - Tests can be switched off at will to improve performance.
  - No further defensive checks within the code necessary.
- An additional benefit at no extra cost.
  - If a specification exists, no further effort is necessary.
  - jmlc accepts all legal JML and Java code.
Runtime Assertion Checking (RAC) allows to test constraints at runtime.

The JML-Distribution includes a RAC-Compiler (jmlc).

Benefits

- An easy means to test both the code *and the specification*.
  - Specification errors are usually quickly detected through tests.
- Allows a clear separation of code and tests.
  - Tests can be switched off at will to improve performance.
  - No further defensive checks within the code necessary.
- An additional benefit at no extra cost.
  - If a specification exists, no further effort is necessary.
  - jmlc accepts all legal JML and Java code.
Runtime Assertion Checking (2)

To consider

- Poor runtime performance.
- Not every assertion is executable.
- RAC tools depart from JML’s logic.

<table>
<thead>
<tr>
<th>Runtimes for Route</th>
<th>#Legs</th>
<th>jmlc (ms)</th>
<th>javac (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belfast/Liverpool</td>
<td>5</td>
<td>4493</td>
<td>227</td>
</tr>
<tr>
<td>Liverpool/Luton</td>
<td>7</td>
<td>6751</td>
<td>265</td>
</tr>
<tr>
<td>Geneva/Nice</td>
<td>18</td>
<td>46412</td>
<td>285</td>
</tr>
<tr>
<td>Luton/Paris de Gaulle</td>
<td>24</td>
<td>122126</td>
<td>316</td>
</tr>
<tr>
<td>Palma de Mallorce/London</td>
<td>32</td>
<td>216660</td>
<td>397</td>
</tr>
<tr>
<td>Luton/Nice</td>
<td>33</td>
<td>309465</td>
<td>370</td>
</tr>
<tr>
<td>Liverpool/Nice</td>
<td>34</td>
<td>362443</td>
<td>402</td>
</tr>
<tr>
<td>Palma de Mallorca/London</td>
<td>37</td>
<td>391441</td>
<td>413</td>
</tr>
</tbody>
</table>
To consider

- Poor runtime performance.
- Not every assertion is executable.
- RAC tools depart from JML’s logic.

- Limited executability of quantified expressions.
  - It must be possible to restrict the range to a finite set.
- Frame conditions are not regarded.
- Invariant enforcement is limited.
  - Invariants are only checked in the course of a method call.
To consider

- Poor runtime performance.
- Not every assertion is executable.
- RAC tools depart from JML’s logic.

The expression $a[x] == a[x]$ for a null field $a$ or an out-of-range value $x$ is:

- true in JML.
- causes an assertion error in jmlc.
Verification with KeY

```java
public boolean add (Object o) {
    ((TreePart)o).setParent(this);
    return super.add(o);
}
```

- Trajectory construction has been partially verified, e.g.:
  - Maintenance of invariants for the tree’s `add` method.
  - The specified behavior of `LegFactory.mergeProcedures`.

- Invariants of a double-linked tree.
  - Hold with KeY’s Observable State semantics.

- Two method calls.
  - Method contracts are used.

- 200 Nodes, 70 Branches.
Verification with KeY

Trajectory construction has been partially verified, e.g.:
- Maintenance of invariants for the tree’s add method.
- The specified behavior of LegFactory.mergeProcedures.

- Concatenates two leg sequences.
  - Truncated at the front and at the back, respectively.
- Two while-loops.
- 8950 Nodes, 204 Branches.
Scepticism towards OOT in the avionics domain.

- Incertitudes how certification requirements can be met.
- Formal methods and DBC address OOT-related safety issues.
  - Explicitely suggested by the OOTiA-Handbook.
- Specification of the JFM indicates benefits and drawbacks.
  - Unambiguous, enforces better design, accessibility to tools, ...
  - Verbosity, limited readability, difficult semantics, LSP, ...
- RAC can be used with no extra effort.
  - Although: limitations of current RAC compiler.
- Formal verification as the ultimate step towards integrity.
  - Elaborate, but well justified for critical parts.
Scepticism towards OOT in the avionics domain.
  ▶ Incertitudes how certification requirements can be met.

Formal methods and DBC address OOT-related safety issues.
  ▶ Explicitly suggested by the OOTiA-Handbook.

Specification of the JFM indicates benefits and drawbacks.
  ▶ Unambiguous, enforces better design, accessibility to tools, ...
  ▶ Verbosity, limited readability, difficult semantics, LSP, ...

RAC can be used with no extra effort.
  ▶ Although: limitations of current RAC compiler.

Formal verification as the ultimate step towards integrity.
  ▶ Elaborate, but well justified for critical parts.
Scepticism towards OOT in the avionics domain.
  ▶ Incertitudes how certification requirements can be met.

Formal methods and DBC address OOT-related safety issues.
  ▶ Explicitly suggested by the OOTiA-Handbook.

Specification of the JFM indicates benefits and drawbacks.
  ▶ Unambiguous, enforces better design, accessibility to tools, ...
  ▶ Verbosity, limited readability, difficult semantics, LSP, ...

RAC can be used with no extra effort.
  ▶ Although: limitations of current RAC compiler.

Formal verification as the ultimate step towards integrity.
  ▶ Elaborate, but well justified for critical parts.
Scepticism towards OOT in the avionics domain.
- Incertitudes how certification requirements can be met.

Formal methods and DBC address OOT-related safety issues.
- Explicitly suggested by the OOTiA-Handbook.

Specification of the JFM indicates benefits and drawbacks.
- Unambiguous, enforces better design, accessibility to tools, ...
- Verbosity, limited readability, difficult semantics, LSP, ...

RAC can be used with no extra effort.
- Although: limitations of current RAC compiler.

Formal verification as the ultimate step towards integrity.
- Elaborate, but well justified for critical parts.
Scepticism towards OOT in the avionics domain.
  ▶ Incertitudes how certification requirements can be met.

Formal methods and DBC address OOT-related safety issues.
  ▶ Explicitly suggested by the OOTiA-Handbook.

Specification of the JFM indicates benefits and drawbacks.
  ▶ Unambiguous, enforces better design, accessibility to tools, ...
  ▶ Verbosity, limited readability, difficult semantics, LSP, ...

RAC can be used with no extra effort.
  ▶ Although: limitations of current RAC compiler.

Formal verification as the ultimate step towards integrity.
  ▶ Elaborate, but well justified for critical parts.
Thank you!

Any questions?