Building Safe and Secure Systems with AADL

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Julien Delange
AADL Meeting February 15
# Building Safe and Secure Systems with AADL

**Delange / Julien**

**Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213**

**Approved for public release, distribution unlimited.**

**The original document contains color images.**
Agenda

Introduction to AADL

AADL modeling patterns for safety and security

AADL validation tools dedicated to security and safety

Demonstration
Agenda

Introduction to AADL

AADL modeling patterns for safety and security

AADL validation tools dedicated to security and safety

Demonstration
Introduction

Systems are becoming extremely software-reliant

Need to verify and validate requirements
• Requirements errors propagate through design
• Need to verify/validate requirements

Major integration and coding issues
• Incur massive re-engineering rework
• Could be removed by early analysis
Architecture Analysis and Design Language

Model-Based Engineering with AADL

- Architecture Language Description standardized by SAE
- Description of Systems and Software Concerns
- Precise & unambiguous semantics
- Textual and Graphical Representation

Support for Model Analysis

- Verify system requirements (i.e. latency, safety)
- Check model integration before producing the implementation
AADL Model-Based Technology Overview

Single Source Annotated Architecture Model with Well-defined Semantics

Safety & Reliability
- MTBF
- FMEA
- Hazard analysis

Security
- Intrusion
- Integrity
- Confidentiality

Data Quality
- Data precision/accuracy
- Temporal correctness
- Confidence

Resource Consumption
- Bandwidth
- CPU time
- Power consumption

Real-time Performance
- Execution time/Deadline
- Deadlock/starvation
- Latency

Auto-generated analytical models

Architecture Model
Understanding Actual Software Issues

High Fault Leakage Drives Major Increase in Rework Cost

Aircraft industry has reached limits of affordability due to exponential growth in SW size and complexity.

70% Requirements & system interaction errors

80% late error discovery at high rework cost

70%, 3.5% 1x

10%, 50.5% 20x

Major cost savings through rework avoidance by early discovery and correction

A $10k architecture phase correction saves $3M

Rework and certification is 70% of SW cost, and SW is 70% of system cost.

Costly certification process leads to high percentage of operational work around.

Sources:
Use of AADL in Development Process

Software and Component Design

Define components requirements & interfaces
Early verification validation of components integration

Code Development

Auto-Generate Code (AADL, Simulink, SCADE)
Avoid traditional coding errors
Ensure correct translation of requirements

Unit & Integration Test

Automatic generation of tests from models
Reduce tests as system was validated earlier
Agenda

Introduction to AADL

AADL modeling patterns for safety and security

AADL validation tools dedicated to security and safety

Demonstration
Security Specifications

Leverage AADL properties for security level specification
 Define security-specific values
 Associate them with components and interfaces

Direct mapping to MILS Security Level concepts
 MILS subjects to AADL runtime components
 MILS objects to AADL interfaces
Partitioning Policy (as in ARINC653 or MILS)

Partitions content and attributes

Use the regular process component
Include partition resources (tasks, data, etc.)

Time and Space Isolation

Time: Partition execution slots
Space: Association of partitions to memory segments
Modeling a MILS architecture - example

Partition Content

Tasks

Software interfaces and data flows

Software (process) & Hardware (cpu/memory) associations/bindings

Separation kernel

Physical Memory

Partition runtime

Logical Memory Segments
Safety Policy with the Error-Model Annex V2

Standardized AADL annex dedicated for safety specification
- Integrated with AADL-core
- Extend/refine existing models

Support of Error Types Ontology
- Characterize the error (i.e. divide by zero, late value)
- Types hierarchy (i.e. late value is an extension of a timing error)

Error Propagations and Behavior Specification
- Errors being propagated by AADL components
- Behavior based on external interfaces or sub-components
Error Propagation

Error Source of Value Error

Sensor ➔ Value Error ➔ Processing ➔ No Data ➔ Actuator

Error Sink for No Data

Sink for Value Error & source for No Data

Error Type Transformation

Errors Propagations through Interfaces
Error Propagation Example

thread producer
features
  dataout : out dataport Character;
annex EMV2 {**
  use types errorlibrary;
  use behavior errorlibrary::FailAndRecover;
error propagations
  dataout : out propagation {ValueError};
flows
  f0 : error source dataout {ValueError};
end propagations;
component error behavior
  events
    ComputationError : error event;
  transitions
    t0 : Operational -[ComputationError] -> Failed;
  propagations
    p0 : Failed -[] -> dataout{ValueError};
end component;
**}

thread consumer
features
  datain : in dataport Character;
annex EMV2 {**
  use types errorlibrary;
  use behavior errorlibrary::FailAndRecover;
error propagations
  datain : in propagation {ValueError};
flows
  f0 : error sink datain {ValueError};
end propagations;
component error behavior
  transitions
    t0 : Operational -[datain{ValueError}] -> Failed;
end component;
**}
properties
  EMV2::severity => ARP4761::Hazardous applies to dataout.ValueError;
  EMV2::OccurrenceDistribution => [ ProbabilityValue => 1.42e-5 ; Distribution => Poisson;]
    applies to dataout.ValueError;
  EMV2::likelihood => ARP4761::Probable applies to dataout.ValueError;
  EMV2::hazards =>
    ([ crossreference => "TBD";
      failure => "";
      phases => "all";
      description => "Bad Value from the thread producer";
      comment => "Must check the software that the value is not faulty";
    ]
     applies to dataout.ValueError;
  **}
Error behavior

States machines
Error-related transitions
Propagation rules
Use of error types

Composite behavior
Define system states according to its parts
ex: “I am failing if one of my component is failing”
Error behavior example

Software Engineering Institute | Carnegie Mellon

 Error behavior example

```
$\text{errorbehavior example}$
```

```
errorlibrary;  
usebehavior errorlibrary::FailAndRecover;  
compositeerrorbehavior  
states  
\[
\begin{align*}
\text{[part1.Failed]} & \rightarrow \text{Failed}; \\
\text{[part2.Failed]} & \rightarrow \text{Failed}; \\
\text{[cpu.Failed]} & \rightarrow \text{Failed}; \\
\end{align*}
\]
endcomposites  
endnode.impl;  
```

```
pu.part1)) applies to part1;  
pu.part2)) applies to part2;  
t1;  
t2;  
```

```
--prove (check_deos_compliance(this))  
prove (check_mils_compliance(this))  
**;  
```

```
useerrorlibrary: 
usebehavior errorlibrary::FailAndRecover;  
compositeerrorbehavior  
states  
\[
\begin{align*}
\text{[part1.Failed]} & \rightarrow \text{Failed}; \\
\text{[part2.Failed]} & \rightarrow \text{Failed}; \\
\text{[cpu.Failed]} & \rightarrow \text{Failed}; \\
\end{align*}
\]
endcomposites  
endnode.impl;  
```

```
pu.part1)) applies to part1;  
pu.part2)) applies to part2;  
t1;  
t2;  
```

```
--prove (check_deos_compliance(this))  
prove (check_mils_compliance(this))  
**;  
```

```
useerrorlibrary: 
usebehavior errorlibrary::FailAndRecover;  
compositeerrorbehavior  
states  
\[
\begin{align*}
\text{[part1.Failed]} & \rightarrow \text{Failed}; \\
\text{[part2.Failed]} & \rightarrow \text{Failed}; \\
\text{[cpu.Failed]} & \rightarrow \text{Failed}; \\
\end{align*}
\]
endcomposites  
endnode.impl;  
```

```
pu.part1)) applies to part1;  
pu.part2)) applies to part2;  
t1;  
t2;  
```

```
--prove (check_deos_compliance(this))  
prove (check_mils_compliance(this))  
**;  
```

```
useerrorlibrary: 
usebehavior errorlibrary::FailAndRecover;  
compositeerrorbehavior  
states  
\[
\begin{align*}
\text{[part1.Failed]} & \rightarrow \text{Failed}; \\
\text{[part2.Failed]} & \rightarrow \text{Failed}; \\
\text{[cpu.Failed]} & \rightarrow \text{Failed}; \\
\end{align*}
\]
endcomposites  
endnode.impl;  
```
Agenda

Introduction to AADL

AADL modeling patterns for safety and security

AADL validation tools dedicated to security and safety

Demonstration
Security Policy Verification

Component integration and composition
- Partitions share the same level with their tasks
- Partitions contain objects at the same level

Runtime issues
- Each process is isolated in a partition
- Partitions has at least one execution slot
- Memory segments contain partitions at the same security level

Communication Policies
- Communication share the same level
- A shared device manages objects at the same level
Specifying Validation Rules with RESOLUTE

Specify constraints on the AADL model
- Check model consistency and properties
- Validation at model level, avoid propagation of errors

List of rules and functions to check the model
- Select elements to be verified
- Filter them according to your constraints
- Check components characteristics

Filter connected partitions with their associated runtime

Check the runtime security level is equal
Generating Assurance Cases

Generate assurance-cases using RESOLUTE and AADL

Show constraints dependencies

Export to Certware

- check_mils_compliance(node_impl_instance: twoparts_mils::node.impl)
  - Check compliance of the model with MILS guidelines
    - Check that component 'node_impl_instance: twoparts_mils::node.impl' and its subcomponents define their security levels
    - Check that connected partitions in 'node_impl_instance: twoparts_mils::node.impl' share the same security level
    - Check that memory segments in system s 'node_impl_instance: twoparts_mils::node.impl' are bound to partitions with the same security level
  - Check that component 'node_impl_instance: twoparts_mils::node.impl' has subcomponents at the same level
    - Check that component 'part1: twoparts_mils::pr_sender.impl' has subcomponents at the same level
    - Check that component 'prod: twoparts_mils::producer.impl' has subcomponents at the same level
  - Check that component 'part2: twoparts_mils::pr_receiver.impl' has subcomponents at the same level
    - Check that component 'recv: twoparts_mils::consumer.impl' has subcomponents at the same level
Safety documentation Generation - FHA

Functional Hazard Assessment

List of all error sources of the system

<table>
<thead>
<tr>
<th>Component</th>
<th>Error Conditions</th>
<th>Hazard Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>part1/prod</td>
<td>ValueError on dataout</td>
<td>Bad Value from the thread producer</td>
</tr>
<tr>
<td>cpu</td>
<td>HardwareError on HardwareError</td>
<td>Hardware Failure of the CPU</td>
</tr>
<tr>
<td>cpu/part1</td>
<td>SoftwareFailure on SoftwareFailure</td>
<td>Software failure from the platform (OS exception; etc.)</td>
</tr>
<tr>
<td>cpu/part2</td>
<td>SoftwareFailure on SoftwareFailure</td>
<td>Software failure from the platform (OS exception; etc.)</td>
</tr>
</tbody>
</table>
Safety documentation Generation - FTA

Fault-Tree Analysis

Bottom-up Approach
Show all contributor of a fault
## Safety documentation Generation – Fault Impact

### Failure Mode and Effect Analysis

- Propagation paths of failures
- Highlight failure containment

### Table: Fault Impact Analysis

<table>
<thead>
<tr>
<th>Component</th>
<th>Initial Failure Mode</th>
<th>1st Level Effect</th>
<th>Failure Mode</th>
<th>second Level Effect</th>
<th>Failure Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>part1.proc</td>
<td>(ValueError)</td>
<td>(ValueError) dataout -&gt; part1.proc:datain</td>
<td>part2.proc (ValueError) [Masked]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu.part1</td>
<td>(ServiceError)</td>
<td>(ServiceError) bindings -&gt; part1:processor</td>
<td>part1 (ServiceError)</td>
<td>(ValueError) dataout -&gt; part2.proc (ValueError) [Masked]</td>
<td></td>
</tr>
<tr>
<td>cpu.part2</td>
<td>internal event Fail</td>
<td>(ItemOmission) bindings -&gt; part2:processor</td>
<td>part2 (ItemOmission) [Failure Effect]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu.part2</td>
<td>internal event Sott</td>
<td>(LateServiceTermination) bindings -&gt; part2:processor</td>
<td>part2 (LateServiceTermination) [Failure Effect]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cpu.part2</td>
<td>(ServiceError)</td>
<td>(ServiceError) bindings -&gt; part2:processor</td>
<td>part2 (ServiceError) [Failure Effect]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Automatic Code Generation

Automatically produce system implementation
   Ensure implementation of system requirements
   Avoid traditional mistakes of manual code generation

Low overhead (memory footprint and additional CPU time)
   Less than 10% in memory and computation increase
   Benefits outweigh the potential

Support for different runtime
   ARINC653/MILS – focus on safety/security (DeOS, POK)
   POSIX (RTEMS, Linux)
Agenda

Introduction to AADL

AADL modeling patterns for safety and security

AADL validation tools dedicated to security and safety

Demonstration
Conclusion

**AADL flexible language to define safety and security concerns**
- Early verification, reducing tests and integration costs
- Automatic code production, avoiding code and integration mistakes

**Integration with existing development methods**
- Safety documentation (i.e. ARP4761)
- Coding standards (i.e. ARINC653)

**Bridge with Validation and Assurance Case tools**
- Check model consistency and composition
- Auto-Generate assurance cases from models
Links & Useful Information

AADL website – http://www.aadl.info

AADL wiki – http://www.aadl.info/wiki

ARINC653 AADL annex standard - http://standards.sae.org/as5506/2/
## Contact

**Dr. Julien Delange**  
RTSS AP Initiative  
Telephone: +1 412-268-9652  
Email: jdelange@sei.cmu.edu

**U.S. Mail**  
Software Engineering Institute  
Customer Relations  
4500 Fifth Avenue  
Pittsburgh, PA 15213-2612  
USA

**Web**  
www.aadl.info  
www.sei.cmu.edu  
www.sei.cmu.edu/contact.cfm

**Customer Relations**  
Email: info@sei.cmu.edu  
Telephone: +1 412-268-5800  
SEI Phone: +1 412-268-5800  
SEI Fax: +1 412-268-6257