A Semantic Web Architecture for Model based Safety Engineering

Vangelis Vassiliadis
Gnomon Informatics
www.gnomon.com.gr

AADL Standards Meeting
Toulouse, FR
May 17-18, 2010
Contents

• The project
• The context, safety engineering
• Modeling
• Ontologies
• System Architecture
• Altarica and Safety Analysis
Project organisation

• Gnomon Informatics
  – Project management
  – System Architecture, Design and Implementation.
  – Ontology Engineering
  – Vangelis Vassiliadis

• Arist. Univ. Thessaloniki
  – Dept. of Informatics.
  – Safety Analysis expertise
  – Transformation to Altarica
  – Panagiotis Katsaros

• Ellidiss Technologies
  – LabAssert design tool.
  – Pierre Dissaux

• Thales Alenia Space
  – Project extension
  – Provide industrial models for system validation
  – Xavier Olive

• Funding ESA / ESTEC
  – Technical Officer: Yuri Yushtein
Objectives

Develop a system to support model-based safety engineering

- Distributed, Reusable, Collaborative Repository / Ontology of Components and Error models
- AADL as system architecture and design language. Error and Behaviour Annex.
- Integration with the ASSERT process through LABASSERT design tool.
- Support safety analyses by transformation to Altarica.

Benefits

- Model-based: Integration of design and safety analysis.
  - Safety considerations early in the development phase.
  - Formal methods and traditional techniques (FTA, FMEA – manual techniques).
  - Compositional safety analysis
  - Handle system complexity (esp. Embedded systems complex SW/HW).
- Use of Semantic Web and Ontology standards.
  - Semantics based interoperability at a language and tool level.
  - Web-ready (Collaborative, Distributed)
- Modular and extensible component representation.
Project timeline and status

- First phase study: Dec ‘08
- Second phase kick-off: March ‘09
- Requirements Analysis: Sep ‘09
- Ontology Engineering: Sep ‘09
- System Specification and Design: Feb ‘10
- Qualification Review: Sep ‘10
- Final System (including Validation): Dec ‘10
Model based Safety analysis process

- Design Assessment
- Functional Requirements
- Nominal System Modeling (System Design)
- Model Analysis Tool
- Extended System Model
- Model Extension
- Fault Modeling
- Failure Modes
- Safety Requirements
- Safety Assessment

Safety Analysis Tools
- Simulation
- Fault Tree Analysis
- Model Checking
ASSERT Process (ESA)

- **Modeling phase**
  - Models described using AADL with TASTE / LABASSERT views
    - Data
    - Functional
    - Interface
    - Deployment Views
  - Function interfaces wrappers (AADL **Subprograms** with function parameters as Features of given **Data** component).

- **Model transformation and Verification**
  - Safety analysis with Altarica
  - Transform AADL model to Altarica Model

- **Code generation**
ASSERT process

FUNCTIONAL

DATA

INTERFACE

DEPLOYMENT

Ontology of Error Models

GLOBAL FAILURE

AADL BEHAVIOR ANNEX

AADL

AADL ERROR ANNEX

VERITCAL TRANSF

ALTARICA MODEL

WARNINGS

glue code

LINK

POLYORB configured

functional code

MATLAB

SDL

ASN.1

Cyclic, 20ms
SDL
ASN.1

Hardware

LABASSERT Interface

CHEDDAR MAST+
Error Modeling

Representation

• As collections of error States, and Transitions between states that are triggered by Events. (Labeled transition system)

Model Extension

• Applying an Error Model to a Component (SW or HW) results in State space expansion: partitioning of its state space into normal / nominal and error states. For both SW and HW components:
  • Identify specific features as state variables
  • Applying Error Model to Component is to specify the values of state variables for each state in the model
  • Applicability of Error Models to Components based on Ontology reasoning (Classification Hierarchies). (E.g. Specific Error Models for Bus, Ethernet Bus etc).

Error Models in AADL

• Combined Error and Behaviour Annex.

Ontology

• Provides a higher level – an abstraction – of the specific syntactic representation of the Annexes
Error Model hierarchy: Feature Inheritance

**error model stuck**
features
- NominalState : error state
- StuckState : error state
- StuckEvent : error event
- StuckProp : in out error propagation

end stuck

**error model recover**
features
- RecoverEvent : error event

end recover

Inherits

**error model recoverable_stuck**
features
- NominalState : error state
- StuckState : error state
- StuckEvent : error event
- StuckProp : out error propagation
- RecoverEvent : error event

end recoverable_stuck

The implementation is defined in the ‘child’
Device Error models

Open Circuit (OC)
- Time drifts (Outputs delay or reach faster the desired value)

Short Circuit (SC)
- Stuck to 0/1 (digital outputs) or Stuck to (Blocked at) open/close for valves, thermostats etc.

S/C with structure
- Out of Range (Too low or too high, Erroneous) output.

Current Leakage
- Leakage (for pipes)

Output value drift (e.g. Frequence drift for crystals)

Failure modes / error states

NominalState
- Event
  - InPropagation
- ErrorToNominalTransition

ErrorState
- RecoverEvent
  - OutPropagation

NominalToErrorTransition
error model recoverable_stuck
features
  NominalState : error state
  StuckState : error state
  StuckEvent : error event
  StuckProp : out error propagation
  RecoverEvent : error event
end recoverable_stuck
// -----------------------------------------------
error model implementation recoverable_stuck.impl
Transitions
  NominalState -> [StuckEvent] StuckState
  NominalState -> [StuckProp] StuckState
  StuckState -> [StuckProp] StuckState
  StuckState -> [RecoverEvent] NominalState
end recoverable_stuck.impl

device Button
Features
  user_input : in event data port;
  output : out event data port;
end Button;
// -----------------------------------------------
device implementation Button.impl
annex Error_Model {**
  Model => recoverable_stuck.impl;
**};

annex behavior_specification {**
  states
    NominalState : initial state // copied from EA
    StuckState : return state // copied from EA
  transitions
    NominalState -> [StuckEvent] StuckState
    {output = stuck} // functional, data code
    NominalState -> [StuckProp] StuckState
    {output = stuck}
    StuckState -> [StuckProp] StuckState
    {output = stuck}
    StuckState -> [RecoverEvent] NominalState
    {output = user_input}
**}
end Button.impl;
subprogram ADD
  features
    a: in parameter behavior::integer ;
    b: in parameter behavior::integer ;
    r: out parameter behavior::integer ;
  properties
    Compute_Execution_Time => 1 Ms .. 1 Ms;
end ADD;

subprogram implementation ADD.impl
end ADD.impl;

error model OVERFLOW
  features
    Error_Free: initial error state;
    Over_Flow: error state;
    overflow: error event
      {occurrence => poisson 1E-4};
    overflow_prop: in out error propagation
      {occurrence => poisson 1E-4};
end OVERFLOW;

error model implementation OVERFLOW.Notional
  transitions
    Error_Free -[ overflow ]-> Over_Flow;
    Error_Free -[ in overflow_prop ]-> Over_Flow;
    Over_Flow -[ out overflow_prop ]-> Over_Flow;
end OVERFLOW.Notional;
example Behaviour Annex

```plaintext
subprogram ADD
    features
        a: in parameter behavior::integer ;
        b: in parameter behavior::integer ;
        r: out parameter behavior::integer ;

    properties
        Compute_Execution_Time => 1 Ms .. 1 Ms;
end ADD;
subprogram implementation ADD.impl

annex behavior_specification {**
    states
        Error_Free : initial state;
        Over_Flow : return state;
    transitions
        normal:Error_Free -[ ]-> Error_Free
            { r:=(a+b);};
        overflow:Error_Free -[ ]-> Over_Flow
            {r:=OVERFLOW;};
        out_overflow_prop:Over_Flow -[ ]-> Over_Flow
            {r:=OVERFLOW;};
        in_overflow_prop:Error_Free -[ ]-> Over_Flow
            {r:=OVERFLOW;};
    **};
annex error_model {**
    Model => error_models::OVERFLOW.Notional;
    **};
end ADD.impl;
```

States, Events and Transitions are copied from the Error Annex.
The user can add action code for the transitions.
Taste (Assert) example

```
-- AP-Level Container: Double_Adder::AU

system AU
features
  ADD: in event port {    Compute_Entrypoint => "ADD";
    ASSERT_Properties::RCMoperation =>
      subprogram Double_Add::ADD;
    ...
  };
end AU;

system implementation AU.others
properties
  Source_Language => Ada;
annex error_model {**
    Model => error_models::OVERFLOW.Notional;
    Guard_In => overflow_prop when
      (ADD[overflow]),
    mask when others
    applies to ADD;
    Guard_Event => ADD[overflow,overflow_prop]
    applies to ADD;
    Report => Error_Free; **};
end AU.others;
```
Example Interfaces (Function calls)

```
-- AP-Level Container: Double_Adder::Calculator
---------------------------------------------------

system Calculator
features
    Perform: in event port {
        Compute_Entrypoint => "Perform";
        ASSERT_Properties::RCMoperation => subprogram Double_Add::Perform;
        ...
    }
    ADD1: out event data port add_data {
        ASSERT_Properties::RCMoperation => subprogram Double_Add::ADD;
        ...
    }
    ADD2: out event data port add_data {
        ASSERT_Properties::RCMoperation => subprogram Double_Add::ADD;
        ...
    }

end Calculator;

system implementation Calculator.others
properties
    Source_Language => Ada;
    annex error_model {**
        Model => error_models::OVERFLOW.Notional;
        Guard_Out => overflow_prop when (Perform[overflow_prop]),
                        mask when others
                        applies to ADD1;
        Report => Error_Free;
        **};

end Calculator.others;
end Double_Adder;
```
BA attached ....

..to System:

SYSTEM IMPLEMENTATION Calculator.others ...

annex behavior_specification {**
states
s_OK : initial;            // copied from EA
s_NoData;                  // copied from EA
s_Overflow

transitions
s_OK -> s_OK { ADD1.a = Perform.a;
ADD1.b = Perform.b;
ADD2.a = ADD1.r;
ADD2.b=Perform.b;
r=ADD2.r}

Error_Free -> Over_Flow {
...
}

..to Subprogram:

subprogram Perform
features
a: in parameter behavior::integer {
   ASSERT_Properties::Encoding => native;};
b: in parameter behavior::integer {
   ASSERT_Properties::Encoding => native;};
c: out parameter behavior::integer {
   ASSERT_Properties::Encoding => native;};
ove: out parameter behavior::boolean {
   ASSERT_Properties::Encoding => native;};
...
end Perform;

subprogram implementation Perform.impl
annex behavior_specification {**
state variables
temp1: behavior::integer;
temp2: behavior::boolean;
temp3: behavior::boolean;

states
Error_Free : initial state;
Over_Flow : return state;

transitions
normal:Error_Free -> Error_Free {
   ADD!(a->a,b->b,temp1->c,temp2->ove);
   ADD!(temp1->a,b->b,c->c,temp3->ove);
   ove = (temp2 and temp3);};

overflow:Error_Free -> Over_Flow {
   (...);
out_overflow_prop:Over_Flow -> Over_Flow {
   (...);
   in_overflow_prop:Error_Free -> Over_Flow {
   (...);
   **};
end Perform.impl;
Semantic Web intro

- Web of (HTML) documents → Web of Data

Human interpreted

- Simple Metadata: XML
- Richer Metadata: RDF/S
- Very Rich Metadata: OWL

Computer interpreted

Interpretation Continuum

DATA
- Relatively unstructured
- Random

- Relatively unstructured
- Random

- Info retrieval
- Web search

- Text summarization
- Content extraction
- Topic maps

- Reasoning services
- Ontology induction

KNOWLEDGE
- Very structured
- Logical

- Very structured
- Logical

Display raw documents; All interpretation done by humans

Find and correlate patterns in raw docs; display matches only

Store and connect patterns via conceptual model (i.e., an ontology); link to docs to aid retrieval

Automatically acquire concepts; evolve ontologies into domain theories; link to institution repositories (e.g., MII)

Automatically span domain theories and institution repositories; inter-operate with fully interpreting computer

Moving to the right depends on increasing automated semantic interpretation

Adapted from: Leo Obrst, “Ontologies and the Semantic Web: An Overview” Mitre, June 2005
Protocol Stacks
Model based systems engineering (ex. from NASA)

Adapted from TopQuadrant Inc., "Ontology Engineering", 20
Ontology Vocabulary

• “Class” ≈ “Concept” ≈ “Category” ≈ “Type”
• “Instance” ≈ “Individual”
• “Entity” ≈ “object”, Class or individual
• “Property” ≈ “Slot” ≈ “Relation” ≈ “Relationtype” ≈ “Attribute” ≈ “Semantic link type” ≈ “Role”

• **OWL**
  – A language to define and use Ontologies on the (Semantic) Web.
  – Named Classes, Instances and Properties are represented by IRIS.
    • Both Symbols and Documents are dereferencable.
  – Based on DAML (DARPA) and OIL (OntoKnowledge)
  – Extension of RDFS.
  – OWL DL Based on Description Logics. Well defined semantics for reasoning, Sound and complete reasoning algorithms.
  – W3C recommendation since Feb ’04.
Component and Failure Model Ontology

Diagram:
- Function Hierarchy
- AADL Hierarchy (Structure)
- Error Model Hierarchy

Component:
- Features
- Subcomponents
- Connections

Error Model:
- States
- Events
- Transitions

Error Model Implementation TransitionCode
Software Error Models: Events - Propagations

**Classes**

<table>
<thead>
<tr>
<th>Event</th>
<th>Subclasses</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComputationalEvent</td>
<td>DivideByZero</td>
<td>InsufficientPrecision</td>
</tr>
<tr>
<td>HardwareEvent</td>
<td>CorruptMemory</td>
<td>Crash</td>
</tr>
<tr>
<td>InsufficientDiskSpace</td>
<td>Disconnected</td>
<td>Dismounted</td>
</tr>
<tr>
<td>InsufficientMemory</td>
<td>PowerOutage</td>
<td>SpuriousInterrupts</td>
</tr>
<tr>
<td>Timeout</td>
<td>TransientErrors</td>
<td></td>
</tr>
</tbody>
</table>

**Exceptions as Events, Categories as States**

**Propagation**

<table>
<thead>
<tr>
<th>Propagation</th>
<th>DataInputPropagation</th>
<th>returnValuePropagation</th>
<th>FailureHandleReturnCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmptyDataFile</td>
<td>ExcessiveData</td>
<td>IncorrectDelimiter</td>
<td>MissingData</td>
</tr>
<tr>
<td>InsufficientEOF</td>
<td>NonAscii</td>
<td>NonNumericField</td>
<td>OutOfRange</td>
</tr>
</tbody>
</table>

**UserInputPropagation**

<table>
<thead>
<tr>
<th>Error as Event</th>
<th>CommandLineArguments</th>
<th>PromptResponse</th>
<th>NoPromptResponse</th>
</tr>
</thead>
<tbody>
<tr>
<td>IncorrectPromptResponse</td>
<td>IncorrectCommand</td>
<td>LatePromptResponse</td>
<td>NoPromptResponse</td>
</tr>
</tbody>
</table>
Software Error Models: Guidelines

All Error Models for Computational (Functional) components are defined as subclasses of the ComputationalErrorModel subclass of the ErrorModel class.

By definition all members of the ComputationalErrorModel class have ComputationErrorState and the MemoryErrorState as Error states.

Error models for Computational components that have hardware-related exceptions are defined as subclasses of the Hardware-generatedErrorModel class.

Error models for Computational components that have features in the left column should be defined as subclasses of the class in the right side of the following table:

<table>
<thead>
<tr>
<th>Feature of Computational component</th>
<th>SuperClass of the associated ErrorModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Input - Output</td>
<td>IOFileErrorModel</td>
</tr>
<tr>
<td>Calling external libraries</td>
<td>LibraryFunctionErrorModel</td>
</tr>
<tr>
<td>Gets / reads input from external data</td>
<td>ReturnValueErrorModel</td>
</tr>
<tr>
<td>Waits/Prompts for user input</td>
<td>TransientComputationalErrorModel</td>
</tr>
<tr>
<td>Handles return values from functions</td>
<td>ReturnValueErrorModel</td>
</tr>
<tr>
<td></td>
<td>HardwareGeneratedErrorModel</td>
</tr>
</tbody>
</table>

- error_events has InsufficientPrecision
- error_events has DivideByZero
- error_events some HardwareEvent
- error_events some ComputationalEvent
- error_states has HardwareErrorState
- error_states has MemoryErrorState
- error_states has ComputationalErrorState
- error_transitions some NominalToComputationalTransition
Software Error Models: Hierarchy
Proposed System Architecture

Ontology Engineering Environment

Component and FM Ontology

FM Ontology Front End (UI)

Labassert UI extensions for Error View

AADL/AAXL components/error models [Web/HTTP Service]

Failure Model Ontology Server

Transform to Altarica model

Model Checking Tool
Ontology Server

- Implements the Ontology API
- Use of external OWL reasoner (e.g. Pellet, RacerPro).
- Contains also a parser and a serialiser between OWL axioms and components and error model frames to AADL and AAXL.
LABASSERT Edit/View Error Models

Get Applicable Error Model List
- Use the `getApplicableErrorModels(Comp)`

Select an Error Model
- from the list of applicable Error Models
- Use the `getComponent(Name)`

View the Error Model
- States
- Events
- Transitions

View – Edit Error Model association
- Error Model implementation details (Error + Behaviour Annex related) for this component.
- Optionally post the updates back to the repository. (Iff Component Owner)
LABASSERT Editing Behaviour Annex

Text editing browser for each transition of the error model.

Editing help to provide the list of valid parameter identifiers

Global syntactic check
Altarica language

• Formal language to model critical systems for safety analysis.
• Developed at LaBRI
• Failure propagation model. Constraint automata.
• Components are represented as Nodes.
  – Flow variables: how components connect and pass data to each other.
  – State variables: internal state of component
  – Events: trigger state changes.
  – Transitions: Guard, Event, New state assignments.
  – Assertions: Global relations between state and flow variables (constraints).
• Complex components, sub nodes.
• Event synchronisation mechanisms.
• Model checking tools: OCAS Cecilia, MEC, ARC
Transformation to Altarica

AADL Components to Altarica Nodes
- Components to Nodes
- Data Features (Port and Parameters) to Flows
- Event Features to Event synchronisations
- Features Data types to flows data types
- Boolean and Enumerated types only
- Each interface call generates a separate callee node.

Error Models and BA
- Events, State and Transitions
- BA state variables to AR state variables
- Implicit _state variable enumerates all EA states
- EA transitions to _state variable conditions and assignments
- BA Transition codes to Transitions action code
- Implicit OK $\rightarrow$ OK transition to assert code

Filtering and Masking
- Guard_In, Guard_Out to Altarica synchronizations
- Actual_Bindings generate synchronisation events between HW and SW
Error View Example: Time_Conversion

SYSTEM Calculator
FEATURES
Mul: IN OUT EVENT DATA PORT;
{
    Compute_Entrypoint => "Mul";
    Assert_Properties::RCMoperation => SUBPROGRAM Calc::Mul.impl;
    ...
};

SUBPROGRAM Mul
FEATURES
o1: in PARAMETER behavior::integer
    { Assert_Properties::encoding => NATIVE;};
o2: in PARAMETER behavior::integer
    { Assert_Properties::encoding => NATIVE;};
r: out PARAMETER behavior::integer
    { Assert_Properties::encoding => NATIVE;};
...
SUBPROGRAM Mul
FEATURES
  c1: in PARAMETER behavior::integer
  { Assert_Properties::encoding => NATIVE;};
  c2: in PARAMETER behavior::integer
  { Assert_Properties::encoding => NATIVE;};
  r: out PARAMETER behavior::integer
  { Assert_Properties::encoding => NATIVE;};
PROPERTIES
  Compute_Execution_Time => 1ms..1ms;
END Mul;

SUBPROGRAM implementation Mul.impl
  annex error_model {**
    Model => error_models::subprogram_ASM.impl;
  **};
  annex behavior_specification {**
    states
      NominalState : initial state;
      ComputationalErrorState : return state;
    transitions
      normal:NominalState -[ ]-> NominalState { r:=(c1*c2);};
      InsufficientPrecision:NominalState -[ ]-> ComputationalErrorState { r:=0;};
      Overflow:NominalState -[ ]-> ComputationalErrorState { r:=0;};
      In_OutOfRange:NominalState -[ ]-> ComputationalErrorState { r:=0;};
    ComputationalErrorState {
      r:=0;};
    RecoverEvent:ComputationalErrorState -[ ]-> NominalState { r:=(c1*c2);};
  **};
END Mul.impl;
Example Transformation to Altarica

node Time_Conversion_Calculator_Mul
flow
Mul^o1:int:in;
Mul^o2:int:in;
Mul^r:int:out;
state
_state:{ComputationalErrorState, NominalState};
event
Overflow,
RecoverEvent,
OutOfRange,
InsufficientPrecision;
trans
(_state = NominalState) |- InsufficientPrecision -> _state := ComputationalErrorState;
(_state = NominalState) |- Overflow -> _state := ComputationalErrorState;
(_state = NominalState) |- OutOfRange -> _state := ComputationalErrorState;
(_state = ComputationalErrorState) |- OutOfRange -> _state := ComputationalErrorState;
(_state = ComputationalErrorState) |- RecoverEvent -> _state := NominalState;
assert
Mul^r = (if (_state = NominalState) then (if ((Mul^o1 * Mul^o2) > 42949676) then 0 else (Mul^o1 * Mul^o2)) else 0);
init
_state := NominalState;
edon
Summary – key points

1. Repository of Components → Reusability
2. Hierarchy / Classification not Partonomy (sub-component)
3. Ontology to annotation of components / associate them with Error Models.
5. Transformation to Altarica for Safety Analysis
Next Steps

• Develop the system
• Validate with industrial case studies (Components and Systems).
End – Q&A

Thank you