AltaRica models and tools for system safety assessment of dynamic systems

Tatiana Prosvirnova (Tatiana.Prosvirnova@centralesupelec.fr)
Christel Seguin (Christel.Seguin@onera.fr)
Lecture outline

• Model Based Safety Assessment Rationals

• AltaRica Basics
  • AltaRica DataFlow Language
  • Assessment tools

• Exercises
Lecture outline

• Model Based Safety Assessment Rationals

• AltaRica Basics
  • AltaRica DataFlow Language
  • Assessment tools

• Exercises
Classical failure propagation models and safety assessment techniques (cf ARP 4761)

- **Failure mode and effect analysis (FMEA)**
  - Model: from a local failure to its system effects / natural languages

  Functional FMEA template

- **Fault tree analysis (FTA)**
  - Model: from a system failure to its root causes / boolean formulae
  - Computation: minimal cut sets / probability of occurrence of top event

- And also Markov chain ....
Drawbacks of the classical Safety Assessment Approaches

• Fault Tree, FMEA
  - Give failure propagation paths without referring explicitly to a commonly agreed system architecture / nominal behavior =>
    - Misunderstanding between safety analysts and designers
    - Potential discrepancies between working hypothesis

• Manual exhaustive consideration of all failure propagations become more and more difficult, due to:
  - increased interconnection between systems,
  - integration of multiple functions in a same equipment
  - dynamic system reconfiguration
Model based safety assessment rationales

- **Goals**
  - Propose formal failure propagation models closer to design models
  - Develop tools to
    - Assist model construction
    - Analyze automatically complex models
  - For various purposes
    - FTA, FMEA, Common Cause Analysis, Human Error Analysis, …
    - since the earlier phases of the system development

- **Approaches**

  Extend design models (Simulink, SysML, AADL...) with failure modes
  Build dedicated failure propagation models (Figaro, AltaRica, Slim...)
  Transform into analyzable formalisms (Boolean formulae, automata, ...)
  Develop specialized analysis tools
What are the tools/languages supporting the MBSA approach?

- AltaRica
  - Simfia (EADS Apsys)
  - Safety Designer (Dassault Systemes)
  - Cecilia OCAS (Dassault Aviation)
  - OpenAltaRica tools (IRT SystemX & AltaRica Association)
  - ARC/AltaRica Studio (University of Bordeaux)
- Figaro (EDF)
- SAML (University of Magdeburg)
- AADL EMV2 (Software Engineering Institute (SEI))
- HiP-HOPS (to some extent) (University of Hull)
- SOPHIA (to some extent) (CEA-LIST)
- Petro (specific to Oil & Gas) (SATODEV)
Lecture outline

• Model Based Safety Assessment Rationals

• AltaRica Basics
  • AltaRica DataFlow Language
  • Assessment tools

• Exercises
AltaRica language at a glance

• Language designed in late 90's at University of Bordeaux
  • for modelling both *combinatorial* and *dynamic* aspects of *failure propagation*
  • in a structured (*hierarchical* and *modular*) way
  • formally.

• AltaRica *node*: structural unit with a temporal behaviour

---

**Input flows**

0. Transitions

- Normal state
- Error state

0. Assertion

- Output = f (inputs, states)

**Output flows**

0. Fault occurrence event
Case study: COM/MON Pattern

- Command/monitoring pattern of safety architecture to compute correct orders even if one fault occurs.

**Structure:**
- Two numerical functions $F1$ and $F2$
- A comparator $Cmp$ that checks the equality of two inputs
- A contactor $Ct$ that is closed as long as the equality check is true. When it is closed, it transmits $F1$ output; else, it transmits no output.

**The functions have two failure modes:**
- They may produce an erroneous output;
- They may produce no output at all.

**The safety requirements of interest for this pattern are:**
- FC_B1: an erroneous output is CAT.
- FC_B2: the output loss is minor.
Case study: the source block

- Let be a basic source function $Source$ that
  - produces data represented by
    - An output $O$

- Source may $fail$.
  - In this case, the output $O$ is lost.

- Source may produce $errors$.
  - In this case the output $O$ is erroneous.

- Initially, the source performs the nominal function
AltaRica basic component: a source function

- **State variables** are used to model the state of the systems.
- **Flow variables** are used to model flows circulating through the model.
- Variables can take their values into predefined domains (Boolean, Integer, Real) or user defined domain (sets of symbolic constants).

```plaintext

domain FailType = {OK, LOST, ERR};

node Source
  flow
    O:FailType:out;
  state
    St:FailType;
  event
    fail_loss, fail_err;
  init
    St := OK;
  trans
    (St = OK) |- fail_loss -> St := LOST;
    (St = OK) |- fail_err -> St := ERR;
  assert
    O = St;
  extern
    law <event fail_loss> = exp(1.0E-4);
    law <event fail_err> = exp(1.0E-5);

```

```text

St == OK
O = OK

fail_loss
St == ERR
O = ERR

fail_error
St == ERR
O = LOST

```

```text

St == OK
O = ERR

```
AltaRica basic component: a source function

Variables change their value when and only when an event occurs, i.e. when the transition it labels is fired.

A transition is a triple <e, G, P>, where e is an event, G is a guard (pre-condition) and P is an action (post-condition).

A transition is enabled only when its guard (pre-condition) is satisfied.

State variables are modified only by actions of transitions.

domain FailType = {OK, LOST, ERR};

node Source
  flow
    O:FailType:out;
  state
    St:FailType;
  event
    fail_loss, fail_err;
  init
    St := OK;
  trans
    (St = OK) |- fail_loss -> St := LOST;
    (St = OK) |- fail_err -> St := ERR;
  assert
    O = St;
  extern
    law <event fail_loss> = exp(1.0E-4);
    law <event fail_err> = exp(1.0E-5);
  edon
Flow variables represent flows of information/matter/energy circulating in the system.

Flow variables depend functionally on state variables: their value is entirely determined by the values of state variables.

They are updated by means of the assertion after each transition firing.
Use of AltaRica components

- AltaRica nodes are similar to classes in the object oriented programming languages.
- They represent reusable (« on-the-shelf ») components.
- They can be instantiated inside other nodes.
- Definitions of nodes cannot be recursive nor circular.
- The names of variables and events of instantiated nodes are prefixed by the name of the instance followed by a dot.

```plaintext
node Comparator
   // body of the node Comparator
edon
node Source
   // body of the node Source
edon
node main
   sub
      Cmp:Comparator;
      F1:Source;
      F2:Source;
      assert
         Cmp.In1 = F1.O,
         Cmp.In2 = F2.O;
   edon
```
Connection of AltaRica components

- Connections of instances:
  - **Assertion** linking inputs and outputs of two different instances.

```plaintext
node Comparator
  // body of the node Comparator
  edon
node Source
  // body of the node Source
  edon
node main
  sub
    Cmp:Comparator;
    F1:Source;
    F2:Source;

  assert
    Cmp.In1 = F1.O,
    Cmp.In2 = F2.O;
  edon
```
Guaranteed Transition Systems is a quintuple \( \langle V, E, T, A, i \rangle \), where:

- \( V \) is a set of variables. \( V \) is the disjoint union of the set \( S \) of state variables and the set \( F \) of flow variables: \( V = S \cup F \).
- \( E \) is a set of events.
- \( T \) is a set of transitions, i.e. of triples \( <e, G, P> \), where
  - \( e \) is an event of \( E \),
  - \( G \) is a Boolean expression built on variables of \( V \)
  - \( P \) is an instruction built on variables of \( V \).
- \( A \) is a set of assertions, i.e. data-flow instructions built on variables of \( V \).
- \( i \) is an assignment of variables of \( V \), so-called initial or default assignment.
Formal definition: example

Source function

- The set of **state** variables: \( S = \{ St \} \)
- The set of **flow** variables: \( F = \{ O \} \)
- The set of **events**: \( E = \{ \) fail\_error, fail\_loss \}
- The set of **transitions**: \( T = \{ <\) fail\_error, St==OK, St :=ERR>,
  <fail\_loss, St==OK, St:=LOST} \)
- The **assertion**: \( A = \{ O=St \} \)
- The **initial assignment**: \( i = \{ St=OK \} \)
Formal definitions: expressions

- The set of expressions is the smallest set such that
  - A **constant** \( c \) is an expression (e.g. true, false, 1, 2, 0.5, OK, ERR)
  - A **variable** is an expression (e.g. F1.st, F2.O, Cmp.Out)
  - \( \text{op}(\text{exp}_1, \ldots, \text{exp}_n) \), is an expression, where \( \text{op} \) is an operator of arity \( n \) and \( \text{exp}_1, \ldots, \text{exp}_n \) are expressions.

- Examples of operators:
  - Boolean: and, or, not
  - Arithmetic: +, -, *, /, ==, >, <
  - Conditional:
    - **if** \( \text{exp}_1 \) **then** \( \text{exp}_2 \) **else** \( \text{exp}_3 \)
    - **case** \{ \( \text{exp}_1 \): \( \text{exp}_2 \), \( \text{exp}_3 \): \( \text{exp}_4 \), \ldots, **else** \( \text{exp}_n \) \}
Formal definitions: actions of transitions

• The set of actions is the smallest set such that:
  • If $v$ is a **state variable** and $E$ is an expression, then “$v := E$” is an instruction (**Assignment**).
  • If $C$ is a (Boolean) expression, $I$ is an instruction, then “if $C$ then $I$” is an instruction (**Conditional instruction**).
  • If $I_1$ and $I_2$ are instructions, then so is “$I_1 ; I_2$” (**Composition**).

• Examples
  • $F1.st := ERR$;
  • $F2.st := LOST$;
Formal definitions: Data-Flow instructions

• The set of instructions is the smallest set such that:
  • If $v$ is a flow variable and $E$ is an expression, then “$v = E$” is an instruction (Assignment).
  • If $C$ is a (Boolean) expression, $I$ is an instruction, then “if $C$ then $I$” is an instruction (Conditional instruction).
  • If $I_1$ and $I_2$ are instructions, then so is “$I_1 ; I_2$” (Composition).
  • Each flow variable is assigned only once.
  • There is no circular definitions.

• Examples:
  • $\text{Cmp.In1} = \text{F1.O}; \text{Cmp.In2} = \text{F2.O}; \text{Cmp.Out} = \text{case} \{ (\text{Cmp.In1} = \text{Cmp.In2}) : \text{true}, \text{else} \text{false} \};$
  • $\{\text{if} \ c_1 \ \text{then} \ I_1; \ \text{if not} \ c_1 \ \text{then} \ I_2;\}$ is equivalent to
    • $\text{if} \ c_1 \ \text{then} \ I_1 \ \text{else} \ I_2;$
Formal definition: composition

- A **composition** of two (or more) Guarded Transition Systems is a Guarded Transition System.
- Let \( G_1 = \langle V_1, E_1, T_1, A_1, i_1 \rangle \) and \( G_2 = \langle V_2, E_2, T_2, A_2, i_2 \rangle \) be two Guarded Transition Systems then \( G = G_1 \circ G_2 = \langle V, E, T, A, i \rangle \) is a Guarded Transition System such that
  - \( V = V_1 \cup V_2 \)
  - \( E = E_1 \cup E_2 \)
  - \( T = T_1 \cup T_2 \)
  - \( A = A_1; A_2 \)
  - \( i = i_1 \circ i_2 \)
• The composition of two (or more) GTS is a GTS. This latter GTS is obtained by flattening.
Formal semantics: reachability graph

- **Configuration**
  - Assignment $\sigma$ of a value to all flow and state variables

- **Kripke structure/ Reachability graph**
  - A graph $<\Sigma, \Theta>$, where
    - $\Sigma$ is a set of nodes, labeled by model configurations $\sigma$
    - $\Theta$ is a set of edges $<\sigma_1, e, \sigma_2>$ labeled by the events

- **The initial state** $\sigma_0$ is calculated as follows
  - First, assign state variables to their initial values (**init** clause)
  - Second, compute the value of flow variables according to the **assertion** $A$: $\sigma_0 = A(\iota)$
Formal semantics: reachability graph

- **Enabled** transition =
  - transition whose guard is true in the current model configuration

- Computation of the next model configurations
  - For each enabled transition, build a next configuration

- In each next configuration:
  - Assign state variable values according to the selected transition action
  - Compute the values of flows variables as in the initial configuration according to the laws in the `assert` clause

- If $\sigma_1$ is in $\Sigma$ and there is a transition $t = <e, G, P>$ such that $t$ is enabled in $\sigma_1$ then $\sigma_2 = A(P(\sigma_1))$ is in $\Sigma$ and $<\sigma_1, e, \sigma_2>$ is in $\Theta$.

- Iterate the computation until no new configuration is reached
Reachability graph: example

initial configuration
Synchronization

- Parallel composition with event grouping: synchronized product of mode automata
  - preserves all states, variables, transitions of ungrouped event, assertions
  - Introduces new grouped transitions $E: <e_1, ..., e_n>$
    - Initially $G_1 |- e_1 -> P_1 ,..., G_n |- e_n -> P_n$
    - Replaced by
      - strong synchronisation: $G_1 \text{ and... and } G_n |- E -> P_1 ;...; P_n$
      - broadcast: $G_1 \text{ or... or } G_n |- E -> \text{ if } G_1 \text{ then } P_1 ;...; \text{if } G_n \text{ then } P_n$
  - interleaving parallelism (only one atomic or a grouped transition at a time)

- Ex: modeling of common cause of failures not propagated by interfaces
  - Explosion, fire, loss of power, ... of a zone

- Comment: “common cause failure” grouping
  - Equivalent to “broadcast” + initial events available
• Common cause failure: loss of power.
• Produces the loss of both functions.
• Is represented by a synchronization of type CCF.
The **synchronized composition** of two (or more) GTS is a GTS. This latter GTS is obtained by **flattening**.
Synchronization: example

- **F1.St**
- **F2.St**
- **Cmp.Out**

**Initial configuration**

- OK, OK (true)
- ERR, OK (false)
- LOST, OK (false)
- LOST, ERR (false)
- ERR, ERR (true)

**Transitions:**

- F1.fail_err → OK, OK (true)
- F1.fail_loss → ERR, OK (false)
- F1.fail_loss → LOST, OK (false)
- F2.fail_err → OK, LOST (false)
- F2.fail_loss → ERR, LOST (false)
- F2.fail_loss → LOST, LOST (true)
- F2.fail_err → ERR, ERR (true)
- F2.fail_loss → ERR, ERR (false)
- F1.fail_err → ERR, OK (false)
- F1.fail_err → LOST, OK (false)
- F1.fail_err → ERR, LOST (false)
- F1.fail_loss → ERR, ERR (false)
- F1.fail_loss → ERR, ERR (true)
- F2.fail_err → ERR, ERR (false)
- F2.fail_loss → ERR, ERR (false)
- F2.fail_err → ERR, ERR (true)
Timed/Stochastic models

• Events are associated with “delay” functions.
• The “delay” functions are used to calculate firing dates for each enabled transition.
• If a transition remains enabled until the firing date, it is fired at this date.

• **Deterministic** transitions
  • Delay function: Dirac(d), d≥0
  • If a transition is enabled at time t, it SHALL be triggered at time t+d

• **Stochastic** transitions
  • Probability distributions for delays: exponential, Weibull, etc.
  • If a transition is enabled at time t, its firing date is t + δ, where δ is calculated randomly according to the probability distribution.
Deterministic transitions

Example: a contactor

- Reconfigurations modeling
  - Event \textit{open\_ct} is associated with \texttt{delay} function \texttt{Dirac(0)}.
  - The transition labeled by \textit{open\_ct} shall be fired as soon as its guard becomes true.
Events `fail_loss` and `fail_err` are **stochastic**.

They are associated with **exponential** probability distributions.

Their firing dates are calculated randomly.
Timed/stochastic models

- Run

\[
\langle \sigma_0, d_0, \Gamma_0 \rangle \xrightarrow{t_1} \langle \sigma_1, d_1, \Gamma_1 \rangle \xrightarrow{t_2} \ldots \xrightarrow{t_n} \langle \sigma_n, d_n, \Gamma_n \rangle
\]

where
- \( \sigma_i \) are configurations,
- \( d_i \) are current firing dates,
- \( \Gamma_i \) are schedulers, functions that associate with each transition its firing date.
- \( t_i \) are transitions.

- In the initial state
  - \( \sigma_0 \) is the initial configuration,
  - \( d_0 = 0 \),
  - \( \Gamma_0 \) is the initial scheduler. For each transition \( t \) it is calculated as follows:

\[
\begin{align*}
- \Gamma_0(t) &= delay_e(t) \text{ for some } z \in [0, 1] \text{ if } G(t) = true. \\
- \Gamma_0(t) &= +\infty \text{ if } G(t) = false.
\end{align*}
\]
Timed/stochastic models

- If the execution $\Lambda$ is a valid execution then so is

$$\Lambda \xrightarrow{t_{n+1}} \langle \sigma_{n+1}, d_{n+1}, \Gamma_{n+1} \rangle$$

if the following conditions hold:
- $t_{n+1}$ is enabled in $\sigma_n$ and its firing date is such that $\Gamma_n(t_{n+1}) \leq \Gamma_n(t)$,
- $\sigma_{n+1} = A(P(\sigma_n))$ is the next configuration,
- $d_{n+1} = \Gamma_n(t_{n+1})$,

- $\Gamma_{n+1}$ is obtained from $\Gamma_n$ by applying the following rules to all transitions $t : G \xrightarrow{e} P$ of $T$.
  - If $G(\sigma_{n+1}) = true$, then:
    - If $G(\sigma_n) = true$ and $t \neq t_{n+1}$, i.e. if the transition was already scheduled, then $\Gamma_{n+1}(t) = \Gamma_n(t)$, i.e. the previous firing date is kept.
    - Otherwise, $\Gamma_{n+1}(t) = d_{n+1} + delay(z)$ for some $z \in [0, 1]$, i.e. a new firing date is chosen.
  - If $G(\sigma_{n+1}) = false$, then $\Gamma_{n+1}(t) = +\infty$.

Note that executions are fully determined by the choices of the $z$'s.
 AltaRica model of the case study

```
domain FailType = {OK, LOST, ERR};

node Source
  flow
    0:FailType:out;
  state
    St:FailType;
  event
    fail_loss, fail_err;
  init
    St := OK;
  trans
    (St = OK) ||- fail_loss -> St := LOST;
    (St = OK) ||- fail_err -> St := ERR;
  assert
    0 = St;
  extern
    law <event fail_loss> = exp(1.0E-4);
    law <event fail_err> = exp(1.0E-5);
  edon

node Comparator
  flow
    In1:FailType:in;
    In2:FailType:in;
    Out:bool:out;
  assert
    Out = case {
      (In1 = In2) : true,
      else false
    };
  edon
```
AltaRica model of the case study

\[\text{domain } \text{FailType} = \{\text{OK, LOST, ERR}\};\]

\text{node} \text{Source}
\begin{align*}
\text{flow} \\
O: & \text{FailType:out} \\
\text{state} \\
St: & \text{FailType}; \\
\text{event} \\
& \text{fail\_loss}, \\
& \text{fail\_err}; \\
\text{init} \\
& \text{St} := \text{OK}; \\
\text{trans} \\
& (\text{St} = \text{OK}) \rightarrow \text{fail\_loss} \rightarrow \text{St} := \text{LOST}; \\
& (\text{St} = \text{OK}) \rightarrow \text{fail\_err} \rightarrow \text{St} := \text{ERR}; \\
\text{assert} \\
& O = \text{St}; \\
\text{extern} \\
& \text{law} <\text{event} \text{fail\_loss}> = \exp(1.0\times10^{-4}); \\
& \text{law} <\text{event} \text{fail\_err}> = \exp(1.0\times10^{-5}); \\
\end{align*}
edon

\text{node} \text{Comparator}
\begin{align*}
\text{flow} \\
& \text{In1:FailType:in;} \\
& \text{In2:FailType:in;} \\
& \text{Out:boolean:out}; \\
\text{assert} \\
& \text{Out} = \text{case} \\
& \quad (\text{In1} = \text{In2}) : \text{true}, \\
& \quad \text{else} \text{false} \\
\end{align*}
edon

\text{node} \text{Contactor}
\begin{align*}
\text{flow} \\
& \text{In:FailType:in}; \\
& \text{Check:boolean:in}; \\
& \text{Out:boolean:out}; \\
\text{state} \\
& \text{Open:boolean}; \\
\text{event} \\
& \text{open\_ct}; \\
\text{init} \text{Open} := \text{false}; \\
\text{trans} \\
& (\text{Open=false}) \text{ and } (\text{Check=false}) \rightarrow \text{open\_ct} \rightarrow \text{Open} := \text{true}; \\
\text{assert} \\
& \text{Out} = \text{case} \\
& \quad \text{Open} : \text{lost}, \\
& \quad \text{else In}; \\
\text{extern} \\
& \text{law} <\text{event open\_ct}> = \text{Dirac(0)}; \\
edon
• **Recall:** The safety requirements of interest for this pattern are:
  - FC_B1: an erroneous output is CAT.
  - FC_B2: the output loss is minor.
Case study: reachability graph

Initial configuration

F1, fail_err

F2, fail_loss

Ct, open_ct

F1, fail_loss

F2, fail_err

Ct, Out

ERR, LOST, open

LOST
Case study: execution

Observed system state: Ct.Out

F1.fail_err  4380
F1.fail_loss  6340
F2.fail_err  5150
F2.fail_loss  5300
Ct.open_ct  +∞

Scheduler

OK  ERR  LOST
Case study: execution

<table>
<thead>
<tr>
<th>State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1.fail_err</td>
<td>+∞</td>
</tr>
<tr>
<td>F1.fail_loss</td>
<td>+∞</td>
</tr>
<tr>
<td>F2.fail_err</td>
<td>5150</td>
</tr>
<tr>
<td>F2.fail_loss</td>
<td>5300</td>
</tr>
<tr>
<td>Ct.open_ct</td>
<td>4380+0</td>
</tr>
</tbody>
</table>

Scheduler

Firing date

Observed state: Ct.Out

OK ERR LOST
Case study: execution

<table>
<thead>
<tr>
<th>State</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1.fail_err</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>F1.fail_loss</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>F2.fail_err</td>
<td>5150</td>
</tr>
<tr>
<td>F2.fail_loss</td>
<td>5300</td>
</tr>
<tr>
<td>Ct.open_ct</td>
<td>$+\infty$</td>
</tr>
</tbody>
</table>

Scheduler

Firing date: 4380 hours

Observed state: Ct.Out

OK  ERR  LOST
Case study: execution

State transition diagram:

- Firing date:
  - F1.fail_err: 4380 hours
  - F2.fail_err: 5150 hours

Observed state:
- Ct.Out

Scheduler:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1.fail_err</td>
<td>+∞</td>
</tr>
<tr>
<td>F1.fail_loss</td>
<td>+∞</td>
</tr>
<tr>
<td>F2.fail_err</td>
<td>+∞</td>
</tr>
<tr>
<td>F2.fail_loss</td>
<td>+∞</td>
</tr>
<tr>
<td>Ct.open_ct</td>
<td>+∞</td>
</tr>
</tbody>
</table>
Guarded Transition Systems

• Guarded Transition Systems are a state/transition formalism dedicated to Safety Analyses

• GTS have many interesting modeling features:
  • States/transitions
  • Remote interactions thanks to flow variables and assertions
  • Implicit representation, compositionality, ability to describe hierarchies
  • Versatile synchronization mechanism

• They encompass
  • Boolean formulae thanks to assertion part
  • Labeled transition system (e.g. Petri Nets) thanks to the transition part
Lecture outline

- Model Based Safety Assessment Rationals

- AltaRica Basics
  - AltaRica DataFlow Language
  - Assessment tools

- Exercises
Complexity of Calculations

- **Calculations** of risk and safety related indicators are extremely resource consuming.

  → **Models** result always from a tradeoff between the accuracy of the description and the ability to perform calculations.
Guarded Transition Systems: assessment tools

**AltaRica models:**
- Hierarchical representation

**Compilation to Fault Trees**
- Minimal cut sets
- Probabilities
  - (Not always possible)

**Stepwise simulation**
- Validate the model
- Play scenarios

**Stochastic simulation**
- Simulate histories
- Calculate statistics

**Sequence generation**
- Explore paths in the reachability graph
- Generate failure scenarios

**Implicit representation of the reachability graph**
Tools for analyzing AltaRica Data-Flow models

• Industrial tools
  • Cecilia OCAS from Dassault Aviation
    • Used for the first time for certification of flight control system of Falcon 7X in 2004
    • Tested by contributors of ARP 4761 (cf MBSA appendix)
  • Simfia (EADS Apsys)
  • Safety Designer (Dassault Systèmes)

• Research workbenchs compatible with AltaRica data flow
  • AltaRica free suite from Labri [http://altarica.labri.fr/wp/](http://altarica.labri.fr/wp/)
  • Open AltaRica 3.0 from IRT SystemX [https://www.openaltarica.fr/](https://www.openaltarica.fr/)
Stepwise simulation: principle

- To **validate/debug** the model
- To play scenarios
- **Principle**
  - Starts from the initial state: \( \sigma_0 = A(\nu) \)
  - Calculates the next configuration \( \sigma_{k+1} = A(P(\sigma_k)) \)
- **Commands**
  - Fire transition
  - Get enabled transitions
  - Get state/flow variables values
  - Back/Forward/Restart/History
- Textual or graphical
- Plays the same role as a debugger for programming languages
Compilation to Fault Trees: principle

- Several Fault Trees can be generated from the same AltaRica model
- **Observers** and their values are transformed to **top events** of the Fault Tree
- **Events** of nodes are transformed to **basic events** of the generated Fault Tree
Compilation to Fault Trees: principle

To compute a fault-tree for FC from an AltaRica Model:

1. Generate the model reachability graph
2. Select the states where the FC holds
3. Compute event paths that lead from the initial state to the selected states

\[
\begin{align*}
\text{init} & \rightarrow s2 \\
\text{s1} & \rightarrow s2 \\
\text{s3} & \rightarrow s2 \\
\text{s4} & \rightarrow s2 \\
\text{FC} & = F_{S3} \text{ or } F_{S4} \\
F_{S3} & = (f1 \text{ and } f3) \text{ or } (f2 \text{ and } f5) \\
F_{S4} & = f2 \text{ and } f4
\end{align*}
\]
Limitations of the compilation

1. Order of occurrence

![Diagram](image)

\[ FC = F_{S4} \]
\[ F_{S4} = b \text{ and } a = a \text{ and } b \]

2. Events having the same name

![Diagram](image)

\[ FC = F_{S3} \]
\[ F_{S3} = b \text{ and } a \text{ and } b = a \text{ and } b \]
Compilation to Fault Trees: an optimized algorithm

A 3 steps algorithm:

1. Flattening
   - Hierarchical Model
   - Flattened Model

2. Partitioning
   - Independent automata
   - Independent assertion

3. Calculation of augmented Reachability Graphs

4. Separate Compilation of Assertion and Reachability Graphs into Fault Trees

Property: if the GTS model is combinatorial, the compilation is efficient and does not lose information
Sequence generation: principle

- Generate sequences of events that lead from the initial state to the state where FCs are hold
- Define targets
  - Observers and their values
- Define stopping criteria:
  - Max number of events in the sequence
Sequence generation: principle

To compute sequences of maximal size $S$ for FC from an AltaRica Model:

- Set $N=1$
- While $N$ is smaller than $S$
  
  1. Generate a sequence of $N$ events
  2. Compute the state reached by the sequence
  3. Check whether the reached state satisfies FC
  4. Increase $N$

Search options:

- $a \ b = b \ a$ => Event **combination**: explore $a;b$
- $a \ b \neq b \ a$ => Event **permutation**: explore $a;b \ & \ b;a$
- $a \ a \neq a$ => Event **repetition**: explore $a;a$
The Monte-Carlo simulation consists in drawing at pseudo-random $N$ possible evolutions, called runs, of the AltaRica model and to make statistics on these $N$ runs.

1. Each run starts at time 0 and ends at time $T$. $T$ is called the mission time.
2. Statistics are made not only at date $T$, but also at observation dates $0 \leq d_1 < \ldots < d_k < T$.
3. Making statistics means calculating moments (mean, standard deviation, confidence ranges).
Static and dynamic models

- **Static** model: the order of the events in the sequence has no influence on the current configuration.

- **Dynamic** model: the last property is not verified => use sequence generation rather than fault tree generation.

```
ok, idle  ok, started  lost, started
  go        fail

ok, idle  ok, started
  fail       go
```

- **Stepwise simulation**
- **Compilation to Fault Trees**:
  - Minimal cut sets
  - Probabilities
- **Stochastic simulation**
- **Sequence generation**
Conclusion

• **Models** result always from a **tradeoff** between the accuracy of the description and the ability to perform calculations.

• **Static** models
  - Efficient assessment algorithms
  - Stepwise simulation
  - Compilation to Fault Trees

• **Dynamic** models
  - Sequence generation
  - Stochastic simulation
  - Stepwise simulation
Lecture outline

- Model Based Safety Assessment Rationals

- AltaRica Basics
  - AltaRica DataFlow Language
  - Assessment tools

- Exercises
Starting point: the leading example
Exercise 1

• Add an activation to a source function
  • If the function is not activated its output is lost
  • Modify the following model to take into account the activation

```
domain FailType = {OK, LOST, ERR};

node Source
  flow
    O:FailType:out;
  state
    St:FailType;
  event
    fail_loss,
    fail_err;
  init
    St := OK;
  trans
    (St = OK) | fail_loss -> St := LOST;
    (St = OK) | fail_err -> St := ERR;
  assert
    O = St;
  extern
    law <event fail_loss> = exp(1.0E-4);
    law <event fail_err> = exp(1.0E-5);
edom
```
domain FailType = {OK, LOST, ERR};

node Source
    flow
        O:FailType:out;
        A: bool: in;
    state
        St:FailType;
    event
        fail_loss,
        fail_err;
    init
        St := OK;
    trans
        (St = OK) | fail_loss -> St := LOST;
        (St = OK) | fail_err -> St := ERR;
    assert
        O = (if A then St else LOST);
    extern
        law <event fail_loss> = exp(1.0E-4);
        law <event fail_err> = exp(1.0E-5);
edon
Exercise 2:

- Write the AltaRica code of the functional block which checks the data integrity
  - Input: Data
  - Output: Boolean
    - *true* if the input data is OK, *false* otherwise
  - Failures
    - Stuck
      - Always sends *true*
      - Always sends *false*
Exercise 2: correction

domain FailType = {OK, LOST, ERR};
domain CheckState = {OK, STUCK_TRUE, STUCK_FALSE};

node CheckOKFunction
  flow
    I:FailType:in;
    O: bool: out;
  state
    St:CheckState;
  event
    stuck_on_true, stuck_on_false;
  trans
    St=OK |- stuck_on_true -> St:= STUCK_TRUE;
    St=OK |- stuck_on_false -> St := STUCK_FALSE;
  assert
    O = case {St=OK : (I=OK),
              St=STUCK_TRUE : true,
              else false };

edon
Exercise 3:

• Build the reachability graph of the following model
Exercise 3: correction

- The assertion is not DataFlow.
- The model is not correct.
Exercise 3 correction: flat model

```plaintext
domain FailType = {OK, LOST, ERR};
domain CheckState = {OK, STUCK_TRUE, STUCK_FALSE};

node Main
  flow
    F1.A: bool: in; F1.O:FailType:out;
    Check.I:FailType:in; Check.O: bool: out;
  state
    F1.St:FailType; Check.St:CheckState;
  event
    ...
  trans
    ...
assert
    F1.A=Check.O;
    F1.O = (if F1.A then F1.St else LOST);
    Check.I = F1.O
    Check.O = case {Check.St=OK : (Check.I=OK),
                  Check.St=STUCK_TRUE : true,
                  else false };
    ...
edon
```
Exercise 3 correction: assertion solving

1) \( F1.A = \text{Check.O} \);
2) \( F1.O = (\text{if } F1.A \text{ then } F1.St \text{ else } \text{LOST}); \)
3) \( \text{Check.I} = F1.O \)
4) \( \text{Check.O} = \text{case } \{ \text{Check.St}=\text{OK} : (\text{Check.I}=\text{OK}), \) \\
   \text{Check.St}=\text{STUCK_TRUE} : \text{true}, \) \\
   \text{else false } \} ; \)

=> Circular definition
Exercise 4:

- Write the AltaRica code of the block « Pre » in order to delay the propagation of data.
Exercise 4: correction

domain FailType = {OK, LOST, ERR};

node PRE
  flow
    O:FailType:out;
    I:FailType: in;
  state
    St:FailType;
  event
    update;
  init
    St := OK;
  trans
    (St != I) |- update -> St := I;
  assert
    O = St;
  extern
    law <event update> = Dirac(0);
edon
Exercise 5:

- Build the reachability graph of the following model:
Exercise 5: correction

Initial configuration

OK, OK, OK

OK, STUCK1, OK

LOST, STUCK0, STUCK1

ERR, STUCK0, LOST

ERR, STUCK1, LOST

ERR, OK, OK

OK, STUCK0, OK

ERR, OK, ERR

ERR, STUCK1, ERR

ERR, STUCK0, ERR

ERR, OK, LOST

ERR, STUCK1, LOST

ERR, STUCK0, LOST

LOST, STUCK0, LOST

LOST, STUCK1, LOST

LOST, STUCK0, LOST

LOST

ERR

OK

Check1.stuck_true

Check1.stuck_false

F1.fail_loss

F1.fail_err

Pre.St

F1.St

Check1.St

update
Cecilia OCAS workbench

- Stepwise simulation
- Sequence generation
- Fault Tree generation and assessment
Another version of the AltaRica model of the case study: comparator with loss failure mode

```plaintext
domain FailType = {OK, LOST, ERR};

node Source
  flow
    0:FailType:out;
  state
    St:FailType;
  event
    fail_loss, fail_err;
  init
    St := OK;
  trans
    (St = OK) |-> fail_loss -> St := LOST;
    (St = OK) |-> fail_err -> St := ERR;
  assert
    0 = St;
  extern
    law <event fail_loss> = exp(1.0E-4);
    law <event fail_err> = exp(1.0E-5);
  edon

node Comparator
  flow
    In1:FailType:in;
    In2:FailType:in;
    Out:bool:out;
  state
    Working:bool;
  event
    fail_loss;
  init Working := true;
  trans
    Working |-> fail_loss ->
      Working := false;
  assert
    Working and (In1 = In2): false,
    else true
  edon
```

Another version of the AltaRica model of the case study: contactor without state

**Domain:**

**Node Source**

**Flow**

0:FailType:in

**State**

St:FailType

**Event**

fail_loss

fail_err

**Init**

St := OK

**Trans**

(St = OK) | -fail_loss -> St := LOST

(St = OK) | -fail_err -> St := ERR

**Assert**

O = St

**Extern**

law <event fail_loss> = exp(1.0E-4);

law <event fail_err> = exp(1.0E-5);

**Node Contactor**

**Flow**

In:FailType:in;

Check:bool:in;

Out:FailType:out;

**Assert**

Out = case {Check : In,

else lost };

**Node Comparator**

**Flow**

In1:FailType:in;

In2:FailType:in;

Out:bool:out;

**State**

Working:bool;

**Event**

fail_loss;

**Init**

Working := true;

**Trans**

Working | -fail_loss -> Working := false;

**Assert**

Out = case {Working and (In1 = In2): false,

else true

};

**Edon**
Recall: The safety requirements of interest for this pattern are:
- FC_B1: an erroneous output is CAT.
- FC_B2: the output loss is minor.
Implementation of this model in Cecilia OCAS workbench

- Graphical and textual edition of models
- Creation of libraries of reusable components
- Safety analyses
Graphical stepwise simulation

Interactive simulation = user driven exploration of the Kripke structure

→ play simple combination of failures (in the style of FMEA)
Define graphical simulation

- Two types of graphical animation of models
  - Icons (to represent the state of nodes)
  - Colored connections (to represent the value of flow variables)

- Define icons and how they change during the simulation

1. Define icons
2. Define how the icons change
Define graphical simulation

- Two types of graphical animation of models
  - Icons (to represent the state of nodes)
  - Colored connections (to represent the value of flow variables)

- Define colors for values of flows variables

1. Select a type
2. Select a color for each value
Graphical stepwise simulation

Start the simulation
Graphical stepwise simulation
Graphical stepwise simulation

- To open simulation view
- Execution history
- Enabled transitions
- Flow variables values
- State variables value
Graphical stepwise simulation

- Fired transition
- Next enabled transitions
- Observation
Graphical stepwise simulation

- Fired transition
- Next enabled transitions
- Observation
Graphical stepwise simulation

- Start the simulation
- Restart (back to the initial state)
- Back
- Forward
- Stop the simulation
- Save as initial configuration
- Open simulation view
- Enabled transitions
Menu MBSA > Sequence generation
1. Define targets (Failure Conditions to observe)

1.1 Select the failure condition

1.2 Select the output file path

Several targets can be defined at the same time.
Sequence generation

2. Select the order for search

3. Select the type of exploration

- \( a b = b a \) : combination
- \( a b \neq b a \) : permutation
- \( a a \neq a \) : repetition

4. Launch the simulation
Sequence generation: results
Fault Tree generation and assessment

Menu MBSA > Fault Tree generation
Fault Tree generation and assessment

1. Select the target (top events, failure conditions to observe)
   - Several targets can be defined at the same time

1.1 Select the failure condition

1.2 Select the output file path
2. Select the algorithm

3. Launch the tool
Fault Tree generation: results
Fault Tree assessment

Import the generated Fault Tree back to Cecilia OCAS
Fault Tree assessment

Imported Fault Tree, graphical view
Fault Tree assessment

Perform calculations:
- Minimal cutsets,
- Probabilities

Results: minimal cutsets
Conclusion

• Model based safety assessment
  • Has been widely tested with aeronautic systems: flight control, electrical, hydraulic, bleed, ...
  • Remain extensible for further researches (e.g. easier handling of a-causal systems)