The AltaRica 3.0 Project

Antoine Rauzy
Chair Blériot-Fabre* - Ecole Centrale de Paris
FRANCE
Antoine.Rauzy@ecp.fr
http://www.lgi.ecp.fr/pmwiki.php/PagesPerso/ARauzy

*Sponsored by SAFRAN group
Agenda

• Context
• Event-Based Modeling
• Guarded Transition Systems
• More about Assertions
• Formal Semantics
• Timed and Stochastic Guarded Transition Systems
• Prototypes & Classes
• Relationships with other Formalisms
• Wrap-Up
Agenda

- Context
- Event-Based Modeling
- Guarded Transition Systems
- More about Assertions
- Formal Semantics
- Timed and Stochastic Guarded Transition Systems
- Prototypes & Classes
- Relationships with other Formalisms
- Wrap-Up
Safety Analyses

System Specification

Models

Virtual Experiments

- Failure Scenarii
- Failure Probabilities

Two categories of formalisms:

- Boolean formalisms: Fault Trees, Event Trees, Block Diagrams
- Transition Systems: Markov Chains, Stochastic Petri Nets...

Virtual experiments are extremely resource consuming (#P-complete)

- Approximations
- Tradeoffs accuracy of models/ability to perform computations

Classical formalisms of safety analyses stand at a very low level

- Distance between system specifications and models
- Models are hard to design and even harder to maintain throughout the life cycle of systems
**The AltaRica Language**

**AltaRica promise:** to model systems at higher level so to reduce the distance between systems and models, without increasing the complexity of calculations.

### System Specification

```plaintext
class HydraulicPump
    Boolean working (reset = false);
    event failure (delay = exponential(lambda));
    transition
        failure: working -> working := false;
end
```

### Features of the language
- Formal
- Event-Based
- Textual & graphical
- Multiple assessment tools

### More than 10 years of industrial experience

Cecilia OCAS

Safety Designer
## Lessons Learned

<table>
<thead>
<tr>
<th>Helpful</th>
<th>Harmful</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>internal origin</strong></td>
<td></td>
</tr>
<tr>
<td>• Theoretical framework</td>
<td>• Theoretical framework</td>
</tr>
<tr>
<td>– Guarded Transition Systems [Rauzy08]</td>
<td>– Looped systems</td>
</tr>
<tr>
<td>• High Level Models are much easier to design, to debug, to master, to maintain, to share, to reuse...</td>
<td>• Trend to design too big and unique models</td>
</tr>
<tr>
<td>• Richness of assessment algorithms</td>
<td>– exponential blow-up</td>
</tr>
<tr>
<td><strong>external origin</strong></td>
<td></td>
</tr>
<tr>
<td>• Large audience in France</td>
<td>• Development costs</td>
</tr>
<tr>
<td>• Certification process accepted by FAA and EASA (Dassault F7X)</td>
<td>• Redundant developments</td>
</tr>
<tr>
<td>• Graphical simulation has a value on its own</td>
<td></td>
</tr>
<tr>
<td>• Used beyond safety analyses (performance analysis)</td>
<td></td>
</tr>
</tbody>
</table>
Next Steps

• AltaRica 3.0
  – A major revision of the language (delivery spring 2014)
    • Prototype-orientation (better suited to the engineering process)
    • Improved expressive power (looped systems, mobile components)
  – Development of Open-Source Tools (delivery fall 2014)

• Integration with System Engineering (Design for Reliability)
  – The dream of a unified language (e.g. Modelica) for multi-physics simulation
    and performance analyses is dangerous and counter-productive
  – Transformations from SysML (or the like) won’t suffice
  – The problem stands at the methodological level
  – Engineering of models (of engineering)

Creation of an eco-system for Model-Based Safety Analyses with industrial partners, software editors, service companies and academic labs
The AltaRica 3.0 Project

- Reliability Data
  - MIL-HDBK
  - Libraries patterns
  - Petri Nets
  - Dynamic FaultTrees

- GUI for modeling
  - GUI for simulation

- AltaRica 3.0
  - class Pump
  - ... end

- Version & Configuration Management System
  - AADL
  - SysML
  - FMEA

- compilation to
  - Fault Trees
  - generation of sequences
  - compilation to Markov Chains
  - stochastic simulation
  - model checking
  - reliability allocation

\[ C_i(R_i) = e^{(1-f) \frac{R_i - R_{\text{min},i}}{R_{\text{max},i} - R_{\text{min},i}}} \]
Agenda

• Context
• Event-Based Modeling
  • Guarded Transition Systems
  • More about Assertions
  • Formal Semantics
  • Timed and Stochastic Guarded Transition Systems
• Prototypes & Classes
• Relationships with other Formalisms
• Wrap-Up
Boolean formalisms

Characteristics of Boolean Formalisms:
- Event centered
- Probabilized Boolean formulas
  - Block Diagrams and Event Trees are compiled into Fault Trees
- A strong assumption: statistical independence of events
  ⇒ Events can always occur
  ⇒ The state of the system does not depend on the order of occurrence of events
- Dedicated algorithms:
  - Minimal Cutsets
  - Binary Decision Diagrams
- Two interesting modeling properties:
  - Naturally hierarchical
  - Capture of remote interactions
States/Transitions Formalisms

State/Transitions formalisms are used when:

- The assumption/approximation that events are statistically independent cannot be made, and/or
- More quantitative measures, e.g. the level of a production, should be handled.

Candidate Formalisms:

- Finite State machines / Markov Chains
- (Stochastic ) State charts*
- Stochastic Petri Nets
- Stochastic Process Algebras*

Evaluation criteria to compare these formalisms:

- Expressiveness: both expressive power and “easiness” of modeling (includes graphics)
- Optimality: as expressive as allowed by assessment algorithms

*Never used at industrial scale for Safety Analyses
• Event-Based

In Safety Analyses, one tries to capture (most probable) failure/accident scenarios, i.e. sequences/sets of events leading from a nominal state to a failure/accident state. There are potentially different types of events (stochastic, instantaneous...)

Expected properties (1)
Expected Properties (2)

- Event-Based
- Compositional & Implicit

Models of systems should be obtained by composing models of subsystems. States of the system should be given in a implicit way to avoid the combinatorial explosion of the size of the model and to allow approximation based on most probable scenarios/states.
Expected properties (3)

• Event-Based
• Compositional & Implicit
• Hierarchical

Models of systems should be obtained by composing models of subsystems or different views of the system into hierarchies.
Expected Properties (4)

- Event-Based
- Compositional & Implicit
- Hierarchical
- Remotely acting

Compositions mechanisms should be able to handle remote interactions amongst components (without enumerating them explicitly).
Expected Properties (5)

- Event-Based
- Compositional & Implicit
- Hierarchical
- Remotely acting
- Graphical

One should be able to represent graphically models, at least partly, for communication and animation purposes.
One cannot design a modeling formalism without having in mind assessment algorithms to use. A good formalism is one that is as expressive as allowed by the targeted assessment algorithms.

- Event-Based
- Compositional & Implicit
- Hierarchical
- Remotely acting
- Graphical
- Optimal w.r.t. Algorithms

- Automatic Generation of FMEA
- Compilation to Fault Trees
- Compilation to Markov Chains
- Generation of Critical Sequences
- Stochastic Simulation
- Model Checking
- Reliability Allocation
# Expected Properties (Summary)

<table>
<thead>
<tr>
<th></th>
<th>Markov Chains</th>
<th>Petri Nets</th>
<th>State Charts</th>
<th>Sequence Algebras</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event based</td>
<td><img src="red" alt="not suitable" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="green" alt="good" /></td>
</tr>
<tr>
<td>Compositional &amp; Implicit</td>
<td><img src="red" alt="not suitable" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="green" alt="good" /></td>
</tr>
<tr>
<td>Hierarchical</td>
<td><img src="red" alt="not suitable" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="green" alt="good" /></td>
</tr>
<tr>
<td>Remotely Acting</td>
<td><img src="red" alt="not suitable" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="green" alt="good" /></td>
</tr>
<tr>
<td>Graphical</td>
<td><img src="blue" alt="acceptable" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="blue" alt="acceptable" /></td>
<td><img src="green" alt="good" /></td>
</tr>
<tr>
<td>Optimal w.r.t. algorithms</td>
<td><img src="red" alt="not suitable" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="green" alt="good" /></td>
<td><img src="green" alt="good" /></td>
</tr>
</tbody>
</table>

- **not suitable**
- **acceptable**
- **good**
- **very good**
Agenda

• Context
• Event-Based Modeling
• Guarded Transition Systems
  • More about Assertions
  • Formal Semantics
  • Timed and Stochastic Guarded Transition Systems
• Prototypes & Classes
• Relationships with other Formalisms
• Wrap-Up
Guarded Transition Systems (1)

A Tank

- State variables are used to model the state of the systems. They are modified by actions of transitions.
- Flow variables are used to model flows circulating through the model. They are updated by means of the assertion after each transition firing.
- Variables can take their values into predefined domains (Boolean, Integer, Real, Symbol) or used defined domain (sets of symbolic constants)

```plaintext
class Tank
  Boolean isEmpty (init = false);
  Boolean outFlow (reset = true);
  event getsEmpty;
  transition
    getsEmpty: not isEmpty -> isEmpty := true;
  assertion
    outFlow := not isEmpty;
end
```
Guarded Transition Systems (2)

A Valve

- **domain** `ValveStatus` { WORKING, STUCK }
- **class** `Valve`
  - `ValveStatus status (init = WORKING);`
  - `Boolean open (init = true);`
  - `Boolean rightFlow (reset = false);`
  - `Boolean leftFlow (reset = false);`
  - **event** `open, close, failure;`
  - **transition**
    - `open: status==WORKING and closed -> closed := false;`
    - `close: status==WORKING and not closed -> closed := true;`
    - `failure: status==WORKING -> status := STUCK;`
  - **assertion**
    - `if not closed then leftFlow :== rightFlow;`
    - `end`

- GTS are acausal: the direction of flow propagation is determined at execution time
The composition of two (or more) GTS is a GTS. This latter GTS is obtained by flattening.

class TankWithValve  // Flattened
Boolean T.isEmpty (init = false);
Boolean T.outFlow (reset = true);
ValveStatus V.status (init = WORKING);
Boolean V.leftFlow (reset = false);
Boolean V.rightFlow (reset = false);
Boolean outFlow (reset = false);
event T.getsEmpty, V.close, V.open, V.failure;

transition
T.getsEmpty: not T.isEmpty-> T.isEmpty := true;
V.open: V.status==WORKING and V.closed ->
   V.closed := false;
V.close: V.status==WORKING and not V.closed ->
   V.closed := true;
V.failure: V.status==WORKING -> V.status := STUCK;

assertion
T.outFlow := not T.isEmpty;
if not V.closed then V.leftFlow := V.rightFlow;
V.leftFlow := T.outFlow;
outFlow := T.rightFlow;
end
The graph of reachable states of a GTS (pedantically a Kripke structure) is obtained by step wisely adding states and transitions, starting from the initial state, until a fixpoint is reached.
class TankWithValves
Tank T;
Valve V1, V2;
Boolean outFlow (reset = false);
event openValves, closeValves, CCF;
transition
openValves: !V1.open & !V2.open;
closeValves: !V1.close & !V2.close;
CCF: ?V1.failure & ?V2.failure;
T.getsEmpty: !T.getsEmpty & outFlow -> skip;
hide V1.open, V1.close;
hide V2.open, V2.close;
assertion
T.outFlow :=: V1.leftFlow;
V1.rightFlow :=: V2.leftFlow;
V2.rightFlow :=: outFlow;
end

Synchronizations:
• Valves are open and closed simultaneously
• Valves have a common cause failure
• The tank cannot get empty is a valve is closed
Synchronization rules

- Modality "!": the event is mandatory
- Modality "?": the event is optional
- At least one of the synchronized transitions must be fireable
- Fireable synchronized transitions are fired

```plaintext
class TankWithValves

transition
  openValves: !V1.open & ! V2.open;
  closeValves: !V1.close & !V2.close;
  CCF: ?V1.failure & ?V2.failure;
  T.getsEmpty:
    !T.getsEmpty & outFlow -> skip;
hide V1.open, V1.close;
hide V2.open, V2.close;
assertion
end

Flattening
```

```plaintext
class TankWithValves // Flattened

transition
  openValves:
    V1.status==WORKING and V1.close and V2.status==WORKING and V2.close
    -> { V1.closed := false, V2.closed := false; }
  closeValves:
    V1.status==WORKING and not V1.closed and V2.status==WORKING and not V2.closed
    -> { V1.closed := true, V2.closed := true; }
  CCF:
    V1.status==WORKING or V2.status==WORKING
    -> {
      if V1.status==WORKING then V1.status := STUCK;
      if V2.status==WORKING then V2.status := STUCK;
    }
  T.getsEmpty:
    not T.isEmpty and outFlow -> T.isEmpty := true;
hide V1.open, V1.close;
hide V2.open, V2.close;
assertion
end
```
Agenda

• Context
• Event-Based Modeling
• Guarded Transition Systems
  • More about Assertions
  • Formal Semantics
  • Timed and Stochastic Guarded Transition Systems
• Prototypes & Classes
• Relationships with other Formalisms
• Wrap-Up
### AltaRica variants

<table>
<thead>
<tr>
<th>variant</th>
<th>AltaRica</th>
<th>AltaRica Data-Flow</th>
<th>AltaRica 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Underlying mathematical Model</td>
<td>Constraint Automata</td>
<td>Mode Automata</td>
<td>Guarded Transition Systems</td>
</tr>
<tr>
<td>Mechanism to update flows</td>
<td>Constraint solving</td>
<td>Propagation</td>
<td>Fixpoint</td>
</tr>
<tr>
<td>Complexity</td>
<td>High</td>
<td>Low</td>
<td>Low (one pass only)</td>
</tr>
<tr>
<td>Acausal</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Loop Systems</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Now the field F1 is irrigated through T1, and V1:

- not T1.isEmpty => T1.outFlow
- T1.outFlow => V1.leftFlow
- V1.leftFlow and not V1.closed => V1.rightFlow
- V1.rightFlow => F1.inFlow

T1.outFlow := not T1.isEmpty
V1.leftFlow := T1.outFlow
if not V2.closed then V1.leftFlow := V1.rightFlow
F1.inFlow := V1.rightFlow or V3.rightFlow
Now the field F1 is irrigated through T2, V2 and V3:

- not T2.isEmpty => T2.outFlow
- T2.outFlow => V2.rightFlow
- V2.rightFlow and not V2.closed => V2.leftFlow
- V2.leftFlow => V3.rightFlow
- V3.rightFlow and not V3.closed => V3.leftFlow
- V3.leftFlow => F1.inFlow

T2.outFlow := not T2.isEmpty
V2.outFlow :=: T2.outFlow
if not V2.closed then V2.leftFlow :=: V2.rightFlow
V2.leftFlow :=: V3.rightFlow
if not V3.closed then V3.leftFlow :=: V3.rightFlow
F1.inFlow := V1.rightFlow or V3.rightFlow
Now the field F1 is not irrigated
- not T2.isEmpty => T1.outFlow
- T1.outFlow => V1.leftFlow

T1.outFlow := not T1.isEmpty
V1.leftFlow :=: T1.outFlow

The other flow variables are reset to their default values (false) and instructions are used to check the solution
Agenda

• Context
• Event-Based Modeling
• Guarded Transition Systems
• More about Assertions
• **Formal Semantics**
  • Timed and Stochastic Guarded Transition Systems
  • Prototypes & Classes
  • Relationships with other Formalisms
• Wrap-Up
A Guarded Transition Systems is a quintuple \( \langle V, E, T, A, \iota \rangle \), 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 \( \langle e, G, P \rangle \), where \( e \) is an event of \( E \), \( G \) is a Boolean expression built on variables of \( V \) and \( P \) is an instruction built on variables of \( V \). For the sake of the clarity, we shall write a transition \( \langle e, G, P \rangle \) as \( e : G \to P \).
- \( A \) is an assertion, i.e. an instruction built on variables of \( V \).
- \( \iota \) is an assignment of variables of \( V \), so-called initial or default assignment.

The set of instructions is the smallest set such that.

- “skip” is an instruction.
- If \( v \) is a variable and \( E \) is an expression, then “\( v := E \)” is an instruction.
- If \( C \) is a (Boolean) expression, \( I \) is an instruction, then “if \( C \) then \( I \)” is an instruction.
- If \( I_1 \) and \( I_2 \) are instructions, then so is “\( I_1 ; I_2 \)”. 
Semantics of Actions (SOS style)

An action of a transition extends a given partial variable assignment \( \tau \) of \( S \) by means of a total variable assignment \( \sigma \) of \( V \), according to the following rule.

\[
\begin{align*}
S0: & \quad \langle \text{skip}, \sigma, \tau \rangle \to \tau \\
S1: & \quad \tau(v) = ? \quad \sigma(E) \in \text{dom}(v) \\
& \quad \langle v := E, \sigma, \tau \rangle \to \tau[\sigma(E)/v] \\
S2: & \quad \tau(v) = \sigma(E), \quad \sigma(E) \in \text{dom}(v) \\
& \quad \langle v := E, \sigma, \tau \rangle \to \tau \\
S3: & \quad \sigma(E) = \text{ERROR} \text{ or } \sigma(E) \notin \text{dom}(v) \text{ or } \tau(v) \neq ?, \quad \sigma(E) \neq \tau(v) \\
& \quad \langle v := E, \sigma, \tau \rangle \to \text{ERROR} \\
S4: & \quad \sigma(C) = \text{TRUE} \\
& \quad \langle \text{if } C \text{ then } l, \sigma, \tau \rangle \to \langle l, \sigma, \tau \rangle \\
S5: & \quad \sigma(C) = \text{FALSE} \\
& \quad \langle \text{if } C \text{ then } l, \sigma, \tau \rangle \to \tau \\
S6: & \quad \sigma(C) = \text{ERROR} \\
& \quad \langle \text{if } C \text{ then } l, \sigma, \tau \rangle \to \text{ERROR} \\
S7: & \quad \langle l_1, \sigma, \tau \rangle \to \tau' \\
& \quad \langle l_1; l_2, \sigma, \tau \rangle \to \langle l_2, \sigma, \tau' \rangle \\
S8: & \quad \langle l_2, \sigma, \tau \rangle \to \tau' \\
& \quad \langle l_1; l_2, \sigma \rangle \to \langle l_1, \sigma, \tau' \rangle \\
S9: & \quad \langle l_1, \sigma, \tau \rangle \to \langle l_1', \sigma, \tau' \rangle \\
& \quad \langle l_1; l_2, \sigma, \tau \rangle \to \langle l_1', l_2, \sigma, \tau' \rangle \\
S10: & \quad \langle l_2, \sigma, \tau \rangle \to \langle l_2', \sigma, \tau' \rangle \\
& \quad \langle l_1; l_2, \sigma \rangle \to \langle l_1; l_2', \sigma, \tau' \rangle \\
S11: & \quad \langle l_1, \sigma, \tau \rangle \to \text{ERROR} \\
& \quad \langle l_1; l_2, \sigma, \tau \rangle \to \text{ERROR} \\
S12: & \quad \langle l_2, \sigma, \tau \rangle \to \text{ERROR} \\
& \quad \langle l_1; l_2, \sigma, \tau \rangle \to \text{ERROR}
\end{align*}
\]
An assertion extends a variable assignment \( \tau \) which is total on \( S \) but possibly partial on \( F \), according to the following rules.

**F0:**

\[
\langle \text{skip}, \tau \rangle \rightarrow \tau
\]

**F1:**

\[
\tau(v) = ?, \; \tau(E) \neq ?, \tau(E) \in \text{dom}(v) \quad \Rightarrow \quad \langle v := E, \tau \rangle \rightarrow \tau[v/E, \tau]
\]

**F2:**

\[
\tau(v) = \tau(E), \; \tau(E) \in \text{dom}(v) \quad \Rightarrow \quad \langle v := E, \tau \rangle \rightarrow \tau
\]

**F3:**

\[
\tau(E) = \text{ERROR} \quad \text{or} \quad \tau(E) \notin \text{dom}(v) \quad \text{or} \quad \tau(v) \neq ?, \tau(E) \neq \tau(v) \quad \Rightarrow \quad \langle v := E, \tau \rangle \rightarrow \text{ERROR}
\]

**F4:**

\[
\tau(C) = \text{TRUE} \quad \Rightarrow \quad \langle \text{if } C \text{ then } I, \tau \rangle \rightarrow \langle I, \tau \rangle
\]

**F5:**

\[
\tau(C) = \text{FALSE} \quad \Rightarrow \quad \langle \text{if } C \text{ then } I, \tau \rangle \rightarrow \tau
\]

**F6:**

\[
\tau(C) = \text{ERROR} \quad \Rightarrow \quad \langle \text{if } C \text{ then } I, \tau \rangle \rightarrow \text{ERROR}
\]

**F7:**

\[
\langle I_1, \tau \rangle \rightarrow \tau' \quad \Rightarrow \quad \langle I_1; I_2, \tau \rangle \rightarrow \langle I_2, \tau' \rangle
\]

**F8:**

\[
\langle I_2, \tau \rangle \rightarrow \tau' \quad \Rightarrow \quad \langle I_1; I_2, \tau \rangle \rightarrow \langle I_1, \tau' \rangle
\]

**F9:**

\[
\langle I_1, \tau \rangle \rightarrow \langle I_1', \tau' \rangle \quad \Rightarrow \quad \langle I_1; I_2, \tau \rangle \rightarrow \langle I_1', I_2, \tau' \rangle
\]

**F10:**

\[
\langle I_2, \tau \rangle \rightarrow \langle I_2', \tau' \rangle \quad \Rightarrow \quad \langle I_1; I_2, \tau \rangle \rightarrow \langle I_1, I_2', \tau' \rangle
\]

**F11:**

\[
\langle I_1, \tau \rangle \rightarrow \text{ERROR} \quad \Rightarrow \quad \langle I_1; I_2, \tau \rangle \rightarrow \text{ERROR}
\]

**F12:**

\[
\langle I_2, \tau \rangle \rightarrow \text{ERROR} \quad \Rightarrow \quad \langle I_1; I_2, \tau \rangle \rightarrow \text{ERROR}
\]
The regular semantics of the Guarded Transitions System \( \langle V=S\cup F,E,T,A,\iota \rangle \) is the smallest Kripke structure \( \langle \Sigma, \Theta \rangle \), where \( \Sigma \) is a set of states and \( \Theta \) is a set of transitions, such that:

- \( \sigma_0 = \text{Propagate}(A,1,1) \in \Sigma \). \( \sigma_0 \) is the initial state of the Kripke structure.
- If \( \sigma \in \Sigma \) and \( T \) contains a transition \( e: G \rightarrow P \) which is fireable in \( \sigma \), then \( \tau = \text{Fire}(P,A,1,\sigma) \in \Sigma \) and \( \sigma \xrightarrow{e} \tau \in \Theta \).

Where,

- \( \text{Propagate}(l,I,1,\tau) \) is the propagation mechanism defined for assignments, i.e. the function that:
  1. Extends the partial variable assignment \( \tau \) by the instruction \( I \).
  2. Completes \( \tau \) by setting each unassigned variable \( v \) to its default value \( \iota(v) \).
- \( \text{Fire}(P,A,1,\sigma) = \text{Propagate}(A,1,\text{Update}(P,\sigma)) \)
- \( \text{Update}(P,\sigma) \) is the update mechanism defined for actions of transitions, i.e. the function that:
  1. Extends the partial variable assignment \( \tau \) of \( S \) from the instruction \( I \) and the total variable assignment \( \sigma \) of \( V \) starting with \( \tau = \emptyset \).
  2. Completes \( \tau \) by setting \( \tau(v) = \sigma(v) \) for all variables that are not given a value by the previous step.

The calculation of \( \langle \Sigma, \Theta \rangle \) may raise errors.
Agenda

• Context
• Event-Based Modeling
• Guarded Transition Systems
• More about Assertions
• Formal Semantics
• Timed and Stochastic Guarded Transition Systems
• Prototypes & Classes
• Relationships with other Formalisms
• Wrap-Up
In Timed GTS, a time structure is set over the GTS:

- A function “delay”, that may depend on the current state, calculates a delay for each fireable event.
- This delay is used to calculate a firing date.
- If the transition remains fireable until the firing date, it is fired at this date.

Non deterministic choice. Firing a may prevent to fire b.

class SimpleTimed
  event a (delay = 2);
  event b (delay = 3);
  transition
     a: true -> skip;
     b: true -> skip;
end

A run:
- t=0: schedule a at t=2, schedule b at t=3
- t=2: fire a
- t=4: fire a
- t=6: fire a, then b or fire b then a
Without the MEMORY policy, B would never get the resource

class TwoUsersAndAResource
state Boolean A.busy, B.busy, R.busy (init = false);
event A.getResource (delay = 2, policy = MEMORY);
event B.getResource (delay = 3, policy = MEMORY);
event A.releaseResource (delay =1);
event B.releaseResource (delay =1);
transition
A.getResource : not A.busy and not R.busy -> {
A.busy := true; R.busy := true;
}
B.getResource : not B.busy and not R.busy -> {
B.busy := true; R.busy := true;
}
A.releaseResource : A.busy and R.busy -> {
A.busy := false; R.busy := false;
}
B.releaseResource : B.busy and R.busy -> {
B.busy := false; R.busy := false;
}
end
• The semantics of a Timed GTS is the set of its valid runs

• A valid run is a sequence

\[(\sigma_0, d_0) \xrightarrow{T_0} (\sigma_1, d_1) \xrightarrow{T_1} \cdots \xrightarrow{T_{n-1}} (\sigma_n, d_n) \xrightarrow{T_n} (\sigma_{n+1}, d_{n+1})\]

where:
- the \(\sigma_i\)'s are variable assignments (\(\sigma_0\) is the initial state of the GTS)
- the \(d_i\)'s are the firing dates
- the \(T_i\)'s are transitions such that \(\sigma_i \xrightarrow{T_i} \sigma_{i+1}\)
- at each step, no transition is scheduled before the fired one

**Property:**
Any valid run is a valid path in the reachability graph of the GTS without the time structure (delay + policy)
Stochastic GTS (1)

Stochastic GTS are Timed GTS with three additional ideas/constructs:

• A weight, so-called “expectation” in the AltaRica jargon, is associated with events. When several transitions are fireable at the same date, one of them is picked up at random according its expectation:

\[ p(e: G \rightarrow P) = \frac{\text{expectation}(e)}{\sum_{f: H \rightarrow Q \text{ scheduled at time } t} \text{expectation}(f)} \]

• The delay calculated for an event may be deterministic or stochastic. Stochastic delay are given by means of as inverse of a Cumulative Distribution Function (so they are in \( \mathbb{R}^+ \cup \infty \) :

\[ \text{CDF}(t) = \{ \text{probability that the transition is fired before time } t \} \]

• Runs take an oracle as parameter, i.e. an infinite sequence of real numbers comprised between 0 and 1 (included). The only operation available on an oracle is to consume its first element. This operation returns the first element and the remaining of the sequence (which is itself an oracle).

Oracles concentrate the stochastic aspect of otherwise deterministic models.
domain PumpState { OFF, ON, FAILED}

class SparePump

  state  PumpState state (init = OFF);
  Boolean active (reset = false);
  Boolean inFlow (reset = false);
  Boolean outFlow (reset = false);
  event turnOff (delay = 0);
  event turnOn (delay = 0, expectation = 1-gamma);
  event failureOnDemand (delay = 0, expectation = gamma);
  event failure (delay = exponential(lambda));
  event repair (delay = exponential(mu));
  parameter Real lambda = 1.0e-4;
  parameter Real mu = 0.05;
  parameter Real gamma = 0.02;

transition

  turnOff: state=ON and not active -> state:= OFF;
  turnOn: state=OFF and active -> state:= ON;
  failureOnDemand: state=OFF and active -> state:= FAILED;
  failure: state=ON -> state:= FAILED;
  repair: state=FAILED -> state := OFF;

assertion

  if state=ON then outFlow := intFlow;
end
Agenda

- Context
- Event-Based Modeling
- Guarded Transition Systems
- More about Assertions
- Formal Semantics
- Timed and Stochastic Guarded Transition Systems
- Prototypes & Classes
- Relationships with other Formalisms
- Wrap-Up
CK Theory applied to modeling

Knowledge space (K)

*Stabilized knowledge*

Libraries of «on-the-shelf» components

- **Reuse**
- **Add new components**

Concept space (C)

«Sandbox»

Model creation

- **Create models**
- **Reuse**
- **Add new components**

Final model
All advanced modeling languages provide constructs to structure models, i.e. to group together elements of models and to organize models into hierarchies of components. **AltaRica 3.0** is a **prototype-oriented language**. Prototypes are the basic construct to group model elements. They are called **blocks**. A model is thus a **hierarchy** of nested blocks.
It is often convenient to have a common definition for different elements of a model (e.g. the four wheels of a car). A class is a separately defined, reusable (on the shelf) block. 

- A block can aggregate (contain) other blocks as well as instances, i.e. copies, of classes.
- A class can aggregate blocks and instances of classes.
- Definitions of classes cannot be recursive nor circular.

```
block System
  Pump P
  Valve V
end

class Pump
  // elements of the pump
end

class Valve
  // elements of the valve
end

block System
  Pump P;
  Valve V;
  /* elements to connect the pump P and the valve V*/
end
```
The semantics of AltaRica 3.0 is defined by means of **flattening rules**, i.e. that the hierarchy of blocks (and instances of classes) is collapsed into a single block.

Elements of the nested block (or instance of class) are copied; names are prefixed with the name of the block (or instance of class)
Inheritance

There are two main types of hierarchical links in models: the \textbf{part-of} link (\textit{aggregation}: the left-front wheel is part of the car) and the \textbf{is-a} link (\textit{inheritance}: a car is a vehicle). In AltaRica 3.0, a block or a class can \textit{inherit} from a class. Flattening rules for inheritance are the same as those for aggregation, expected that the inherited component are copied without prefix.

```plaintext
class NonRepairableComponent
  Boolean working (init = true);
  event failure;
  transition
    failure : working -> working := false;
end

class Pump extends NonRepairableComponent
  Boolean inflow (reset = false);
  Boolean outflow (reset = false);
  assertion
    outflow := inflow and working;
end
```

```plaintext
class Pump
  Boolean working (init = true);
  Boolean inflow (reset = false);
  Boolean outflow (reset = false);
  event failure;
  transition
    failure : working -> working := false;
  assertion
    outflow := inflow and working;
end
```
Hierarchical decompositions are often thought only as tree-like structures. But in practice it is often convenient for two branches to share a component (especially when reasoning in functional terms).

*AltaRica 3.0* makes it possible to share blocks through the “embeds” clause.
AltaRica 3.0: Structural Constructions

**Stabilized knowledge**
- Libraries of reusable components

**Sandbox**
- System model
  - Assemble models of subsystems or different views into hierarchies
  - Break down structure

**Create**
- **Classes** (reused by)
  - Extension
  - Instantiation

**Reuse**
- **Blocks**
  - Are unique
  - Structure models into hierarchies
Agenda

• Context
• Event-Based Modeling
• Guarded Transition Systems
• More about Assertions
• Formal Semantics
• Timed and Stochastic Guarded Transition Systems
• Prototypes & Classes
  • Relationships with other Formalisms
• Wrap-Up
GTS generalize Reliability Block Diagrams

class Block
  Boolean OK (init = true);
  Boolean inFlow (reset = false);
  Boolean outFlow (reset = false);
  event failure (delay = exponential(lambda));
  transition
    failure: OK -> OK := false;
assertion
  if OK then outFlow := inFlow;
end

class MyDiagram
  Block A, B, C, D, E, F;
assertion
  A.inFlow := true;
  B.inFlow := A.outFlow;
  C.inFlow := B.outFlow;
  D.inFlow := A.outFlow;
  E.inFlow := D.outFlow;
  F.inFlow := C.outFlow or E.outFlow;
end
GTS generalize Stochastic Petri Nets (and various extensions of)

\[
\begin{align*}
\text{class} & \ MyNet \\
\text{Integer} & \ \text{working} \ (\text{init} = 1); \\
\text{Integer} & \ \text{failed} \ (\text{init} = 0); \\
\text{Integer} & \ \text{in\_repair} \ (\text{init} = 0); \\
\text{Integer} & \ \text{free} \ (\text{init} = 1); \\
\text{Integer} & \ \text{busy} \ (\text{init} = 0); \\
\text{Integer} & \ \text{counter} \ (\text{init} = 0); \\
\text{event} & \ \text{failure} \ (\text{delay} = \text{exponential}(\lambda)); \\
\text{event} & \ \text{startRepair} \ (\text{delay} = 0); \\
\text{event} & \ \text{enRepair} \ (\text{delay} = \text{exponential}(\mu)); \\
\text{parameter} & \ \text{Real} \ \lambda = 1.0e-3; \\
\text{parameter} & \ \text{Real} \ \mu = 1.0e-1; \\
\text{transition} & \\
\text{failure: working} \geq 1 & \rightarrow \{ \\
& \quad \text{working} := \text{working} - 1; \ \text{failed} := \text{failed} + 1; \} \\
\text{startRepair: failed} \geq 1 & \text{ and free} \geq 1 \rightarrow \{ \\
& \quad \text{failed} := \text{failed} - 1; \ \text{free} := \text{free} - 1; \\
& \quad \text{in\_repair} := \text{in\_repair} + 1; \ \text{busy} := \text{busy} + 1; \} \\
\text{endRepair: in\_repair} \geq 1 & \text{ and busy} \geq 1 \rightarrow \{ \\
& \quad \text{in\_repair} := \text{in\_repair} - 1; \ \text{busy} := \text{busy} - 1; \\
& \quad \text{working} := \text{working} + 1; \ \text{free} := \text{free} + 1; \\
& \quad \text{counter} := \text{counter} + 1; \} \\
\end{align*}
\]
GTS generalize Stochastic Petri Nets (and various extensions of)

\[
\begin{align*}
\text{Engine State} & = \{ \text{WORKING, FAILED, IN_REPAIR} \} \\
\text{Repairman State} & = \{ \text{FREE, BUSY} \}
\end{align*}
\]

class MyNet

EngineState engine (\(\text{init} = \text{WORKING}\));
RepairManState repairMan (\(\text{init} = \text{FREE}\));
Integer counter (\(\text{init} = 0\));
event failure (delay = exponential(\(\lambda\)));
event startRepair (delay = 0);
event endRepair (delay = exponential(\(\mu\)));
parameter Real \(\lambda = 1.0e-3\);
parameter Real \(\mu = 1.0e-3\);

transition

failure: engine==\text{WORKING} \rightarrow engine := \text{FAILED};
startRepair: engine==\text{FAILED} and repairMan==\text{FREE} \rightarrow \{ 
  engine := \text{IN_REPAIR}; repairMan := \text{BUSY}; 
\} 
endRepair: engine==\text{IN_REPAIR} and repairMan==\text{BUSY} \rightarrow \{ 
  engine := \text{WORKING}; repairMan := \text{FREE};
  counter := counter+1; 
\} 
end
In an industrial scale model, one would rather use on-the-shelf components...
Dynamic Fault Trees

Idea: Gates and Basic Events
- calculates their status bottom-up
- are activated top-down

```
class Trigger  // à la BDMP
  Boolean failed (reset = false);
  Boolean mainFailed (reset = false);
  Boolean spareFailed (reset = false);
  Boolean active (reset = false);
  Boolean mainActive (reset = false);
  Boolean spareActive (reset = false);

assertion
  failed := mainFailed and spareFailed;
  mainActive := active and not mainFailed;
  spareActive := active and mainFailed and not spareFailed
end
```
Compilation into Fault Trees

A 3 steps algorithm:

1. Flattening
   - Hierarchical Model
   - Flattened Model

2. Partitioning
   - independent automata

3. Calculation of augmented Reachability Graphs

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

Property: if the GTS model was combinatorial, the compilation is efficient and does not lose information
Agenda

• Context
• Event-Based Modeling
• Guarded Transition Systems
• More about Assertions
• Formal Semantics
• Timed and Stochastic Guarded Transition Systems
• Prototypes & Classes
• Relationships with other Formalisms
• Wrap-Up
Guarded Transitions Systems can be described by 3 types of diagrams (Petri-nets like diagrams could be considered as well).

None of them (and even a combination of them) can fully achieved the description (that’s probably the case for any rich enough formalism)

Automaton Diagram

- **state==WORKING**
  - not closed
  - leftFlow==rightFlow

- **state==STUCK**
  - not closed
  - leftFlow==rightFlow

Synchronization Diagram

- **openValves**
  - open

- **closeValves**
  - close

- **CCF**
  - failure
  - failure

- **V1.failure**
  - failure

- **V2.failure**
  - failure

- **T.getsEmpty**
  - getsEmpty

- **outFlow**
  - outFlow -> skip;

Structural Diagram

- **T1**

- **V1**

- **V3**

- **V2**

- **T2**
## Expected Properties (Summary)

<table>
<thead>
<tr>
<th></th>
<th>Markov chains</th>
<th>Petri Nets</th>
<th>State Charts</th>
<th>Sequence Algebras</th>
<th>GTS/AltaRica</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event based</td>
<td>![not suitable]</td>
<td>![acceptable]</td>
<td>![good]</td>
<td>![acceptable]</td>
<td>![good]</td>
</tr>
<tr>
<td>Compositional &amp; Implicit</td>
<td>![not suitable]</td>
<td>![good]</td>
<td>![good]</td>
<td>![good]</td>
<td>![good]</td>
</tr>
<tr>
<td>Hierarchical</td>
<td>![not suitable]</td>
<td>![good]</td>
<td>![good]</td>
<td>![good]</td>
<td>![good]</td>
</tr>
<tr>
<td>Remotely Acting</td>
<td>![not suitable]</td>
<td>![good]</td>
<td>![good]</td>
<td>![good]</td>
<td>![good]</td>
</tr>
<tr>
<td>Graphical</td>
<td>![acceptable]</td>
<td>![good]</td>
<td>![good]</td>
<td>![good]</td>
<td>![good]</td>
</tr>
<tr>
<td>Algorithm Friendly</td>
<td>![not suitable]</td>
<td>![good]</td>
<td>![good]</td>
<td>![good]</td>
<td>![good]</td>
</tr>
</tbody>
</table>
Are Guarded Transition Systems an Every Purpose Formalism?
No, definitely not, of course not!

Some of the limitations of GTS:

• GTS are not designed to capture intrinsically continuous phenomena typically modeled with Ordinary Differential Equations.
  Some formalisms to handle continuous phenomena:
  – Matlab/Simulink, Modelica
  – Hybrid & Timed Automata (Alur & Dill, Sifakis & Al., Hentzinger, Larsen,...)

• GTS are not designed to handle processes or actors which are dynamically created and destroyed during a mission (a simulation run).
  Some formalisms to handle dynamically created processes:
  – Process Algebras, e.g. $\pi$-calculus (Milner)
  – Colored Petri-Nets (to some extent)
  – Actor languages

• GTS are not very well suited to handle mobile components
  Some formalisms to handle mobile components:
  – All the previous ones
  – PEPA-nets
• Guarded Transitions Systems are a state/transition formalism dedicated to Safety Analyses.
• GTS have many interesting modeling features:
  – State/transitions
  – Remote interactions thanks to flow variables and assertions
  – Implicit representation, compositionality, ability to describe hierarchies
  – Versatile synchronization mechanism
  – Acausality, ability to handle looped systems
• The semantics of (Stochastic) GTS is formally defined
• GTS are nearly as expressive as allowed by targeted assessment algorithms.

• GTS are probably the best trade-off between expressivity and ability to take the best of assessment algorithms in this class of formalisms and for this purpose.
• GTS are at the core of AltaRica 3.0