The AltaRica 3.0 Project for Model-Based Safety Assessment

Tatiana Prosvirnova [*] **Michel Batteux** [*]
Pierre-Antoine Brameret [**] **Abraham Cherfi** [*]
Thomas Friedlhuber [*] **Jean-Marc Roussel** [**] **Antoine Rauzy** [*]

[*] *LIX - Ecole Polytechnique, route de Saclay, 91128 Palaiseau cedex, FRANCE (e-mail: name@lix.polytechnique.fr).*
[**] *LURPA - ENS Cachan, 61 avenue du Président Wilson, 94235 Cachan cedex, FRANCE (e-mail: firstname.name@lurpa.ens-cachan.fr).*

Abstract: The aim of this article is to present the AltaRica 3.0 project. "Traditional" risk modeling formalisms (e.g. Fault Trees, Markov Processes, etc.) are well mastered by safety analysts. Efficient assessment algorithms and tools are available. However, models designed with these formalisms are far from the specifications of the systems under study. They are consequently hard to design and to maintain throughout the life cycle of systems. The high-level modeling language AltaRica has been created to tackle this problem.
The objective of the AltaRica 3.0 project is to design a new version of AltaRica and to develop a complete set of authoring and assessment tools for this new version of the language. AltaRica 3.0 improves significantly the expressive power of AltaRica Data-Flow without decreasing the efficiency of assessment algorithms. Prototypes of a compiler to Fault Trees, a compiler to Markov chains, stochastic and stepwise simulators have been already developed. Other tools are under specification or implementation.

Keywords: Safety and Reliability analysis, Model-Based design.

1. INTRODUCTION

The Model-Based approach for safety and reliability analysis is gradually wining the trust of engineers but is still an active domain of research. Safety engineers master "traditional" risk modeling formalisms, such as "Failure Mode, Effects and Criticality Analysis" (FMECA), Fault Trees (FT), Event Trees (ET), Markov Processes. Efficient algorithms and tools are available. However, despite of their qualities, these formalisms share a major drawback: models are far from the specifications of the systems under study. As a consequence, models are hard to design and to maintain throughout the life cycle of systems. A small change in the specifications may require revisiting completely safety models, which is both resource consuming and error prone.

The high-level modeling language AltaRica Data-Flow (Rauzy, 2002; Boiteau et al., 2006) has been created to tackle this problem. AltaRica Data-Flow models are made of hierarchies of reusable components. Graphical representations are associated with components, making models visually very close to Process and Instrumentation Diagrams. AltaRica Data-Flow is at the core of several Integrated Modeling and Simulation Environments: Cecilia OCAS (Dassault Aviation), Simfia (EADS Apsys), and Safety Designer (Dassault Systèmes). Successful industrial experiments were held using AltaRica Data-Flow (e.g. certification of the flight control system of the aircraft Falcon 7X) (Bernard et al., 2007; Bieber et al., 2008). In a word, AltaRica Data-Flow has reached an industrial maturity.

However, more than ten years of experience showed that both the language and the assessment tools can be improved. AltaRica 3.0 is an entirely new version of the language. Its underlying mathematical model – Guarded Transition Systems (Rauzy, 2008; Prosvirnova and Rauzy, 2012) – makes it possible to design acausal components and to handle looped systems. The development of a complete set of freeware authoring and assessment tools is planned, so to make them available to a wide audience.

The aim of this article is to present the AltaRica 3.0 project. It is organized as follows. Section 2 gives an overview of the project. Section 3 introduces the new version of the language. Section 4 presents Guarded Transition Systems. Section 5 presents the current state of the development of authoring and assessment tools. Section 6 presents related works. Finally Section 7 concludes the article and outlines directions for future works.

2. OVERVIEW OF THE PROJECT

The objective of the AltaRica 3.0 project is to propose a set of modeling and assessment tools to perform preliminary safety analyses. Figure 1 presents the overview of the project.

AltaRica 3.0 is in the heart of the project. It significantly increases the expressive power of AltaRica Data-Flow without decreasing the efficiency of assessment algorithms. Models are compiled into a low level formalism: Guarded Transition Systems (GTS). GTS is a states/transitions

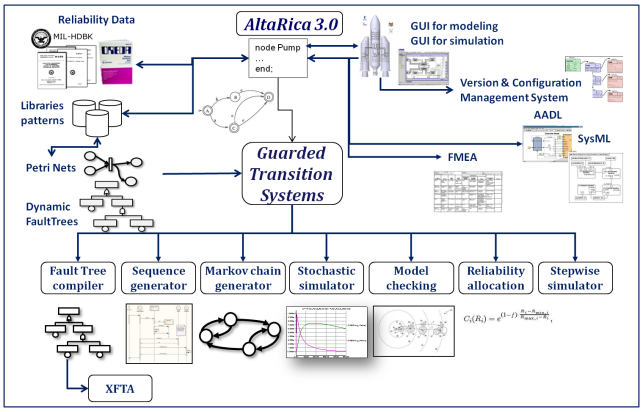

Fig. 1. Overview of the AltaRica 3.0 project

formalism generalizing classical safety formalisms, such as Reliability Block Diagrams and Markov Chains. It is a pivot formalism for Safety Analyses: other safety models can be compiled into GTS to take benefits from assessment tools. The assessment tools for GTS already include a Fault Tree compiler to perform Fault Tree Analysis (FTA), a Markov Chain generator, a stochastic and a stepwise simulators. Other tools are under specification or implementation: a model-checker and a reliability allocation module. These tools will be distributed under a free license in order to make them available to a wide audience, especially in the academic community. They enable users to perform virtual experiments on systems, to compute reliability indicators and, also, to perform cross check calculations.

3. ALTARICA 3.0 MODELING LANGUAGE

AltaRica is an event based modeling language. The state of the system is described by means of variables. The system changes its state when, and only when, an event occurs. The occurrence of an event updates the value of variables. Events can be associated with deterministic or stochastic delays so to obtain (stochastic) timed models. Models of components can be assembled into hierarchies, their inputs and outputs can be connected and their transitions can be synchronized.

Two main variants of AltaRica have been designed so far. These two variants differ essentially with respect to the way variables are updated after each transition firing. In the first version, variables are updated by solving constraints (Point and Rauzy, 1999; Arnold et al., 2000). This mechanism, although very powerful, is too resource consuming for industrial scale applications. To be able to assess industrial scale models, in the second version, AltaRica Data-Flow (Rauzy, 2002; Boiteau et al., 2006), variables are updated by propagating values in a fixed order. This order is determined at compile time, which imposes strong constraints on the way assertions are written. However, some problems stay: located synchronizations cannot be captured, bidirectional flows, circulating through a network, cannot be modeled in a natural way, and looped systems remains difficult to model.

The new version is currently under specification. It improves AltaRica Data-Flow into two directions:

(1) Its semantic is based on the new underlying mathematical model: Guarded Transition Systems (GTS).

(2) It provides new constructs to structure models.

The new underlying formalism, i.e. GTS, makes it possible to handle systems with instant loops and to define acausal components, i.e. components for which the input and output flows are decided at run time. It is much easier to model systems with bidirectional flows, such as electrical systems.

AltaRica 3.0 is a prototype oriented modeling language, see e.g. Noble et al. (1999) for a discussion on objects versus prototypes. Prototype orientation makes it possible to separate the knowledge into two distinct spaces: the stabilized knowledge, incorporated into libraries of on-the-shelf modeling components; the sandbox in which the system under study is modeled. In the sandbox, many components are unique and some others are instances of reusable components. With prototype-orientation, models can be reused in two ways: at component level by instantiating off-the-shelf components; at system level by cloning and modifying a model designed for a previous project.

Example

Consider a part of a system, named *TwoValves* (see Figure 2), composed of two valves connected in series. *rightFlow* and *leftFlow* are bidirectional flows circulating through this sub-system.

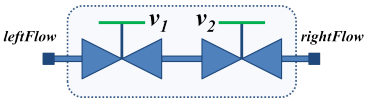

Fig. 2. The component *TwoValves*.

A valve is represented by the class *Valve*. This valve has two states: closed or not. When the valve is opened, the right flow and the left flow are the same. In functional mode, the states change according to the command events open and close. For this class of valves, we consider only a failure behavior: the valve is blocked in the current state. The AltaRica 3.0 model of a valve is the following:

```
class Valve
  Boolean closed (init = true), blocked (init = false);
  Real rightFlow (reset = 0), leftFlow (reset = 0);
  event open, close, failure(delay = exponential(0.0005));
  transition
    open: closed and not blocked -> closed := false;
    close: not closed and not blocked -> closed := true;
    failure: not blocked -> blocked := true;
  assertion
    if not closed then leftFlow :=: rightFlow;
end
```

The class *Valve* contains two state variables *closed* and *blocked*; and two flow variables *rightFlow* and *leftFlow*. Events *open*, *close* and *failure* are declared and then used to define transitions. The behavior of a valve is thus defined by transitions (to update its state) and the assertion (to propagate flows). The operator ":=:" expresses a bidirectional flow circulating through the valve.

The sub-system *TwoValves* is represented by the class *TwoValves*, containing two instances of the class *Valve*. *rightFlow* and *leftFlow* are bidirectional flows circulating through the sub-system when both valves are open. Each valve can fail individually. Both valves can fail simultaneously due to a common cause failure. The corresponding AltaRica 3.0 model is given below:

```
class TwoValves
  Valve v1, v2;
  Real rightFlow (reset = 0), leftFlow (reset = 0);
  event ccf;
  transition
    ccf: ?v1.failure & ?v2.failure;
    v1.failure: !v1.failure;
    v2.failure: !v2.failure;
  assertion
    leftFlow :=: v1.leftFlow;
    rightFlow :=: v2.rightFlow;
    v1.rightFlow :=: v2.leftFlow;
end
```

The event *ccf* represents the common cause failure of the two valves. It is expressed by means of the synchronization of the events *v1.failure* and *v2.failure*. The event *ccf* may occur when at least one of two valves is working. Operator "!"/"?" means that the event is mandatory/optional for the synchronization.

To be able to describe the sub-system *TwoValves* with the previous version of the language, the class *Valve* should include four flow variables: two inputs and two outputs for the left and right hand sides (i.e. *inputRightFlow*, *inputLeftFlow* for inputs and *outputRightFlow*, *outputLeftFlow* for outputs). Thus, the instruction of the assertion should be replaced by a block connecting inputs and outputs:

```
if not closed then
   {outputRightFlow := inputLeftFlow;
   outputLeftFlow := inputRightFlow;}
```

Finally, the connection between the two valves should be done by connecting *inputRightFlow* of *v1* to *outputLeftFlow* of *v2* and *inputLeftFlow* of *v2* to *outputRightFlow* of *v1*:

```
v1.outputRightFlow := v2.inputLeftFlow;
v2.outputLeftFlow := v1.inputRightFlow;
```

When considering a complex system with a significant number of similar components, connecting them together may be a tedious and error prone task.

4. GUARDED TRANSITION SYSTEMS

Before any assessment, AltaRica models are "flattened" (i.e. reduced to a single class without sub-classes). This "flattened" class produces a GTS model. The new semantics of instructions (Prosvirnova and Rauzy, 2012) makes it possible to represent components with bidirectional flows.

4.1 Definition

A Guarded Transition Systems is formally a quintuple $\langle V, E, T, A, \iota \rangle$, where:

- $V = S \uplus F$ is a set of variables, divided into disjoint sets S of state variables and F of flow variables.
- E is a set of symbols, called events.
- T is a set of transitions.
- A is an assertion (i.e. an instruction built over V).
- ι is the initial (or default) assignment of variables of V.

GTS is thus a states/transitions formalism where states are implicit, i.e. given by variables assignments σ. A transition is a triple $\langle e, G, P \rangle$, also denoted $e : G \to P$, where $e \in E$ is an event, G is a guard, i.e. a Boolean formula built over V, and P is an instruction built over V, also called an action or a post-condition. A transition $e : G \to P$ is said *fireable* in a given state σ if its guard G is satisfied in this state.

4.2 Instructions

Both assertions and actions of transitions are described by means of instructions. There are basically four types of instructions:

- The *empty instruction* noted *skip*.
- The *assignment* $v := E$, where v is a variable and E is an expression built over variables from V.
- The *conditional assignment* $if\ C\ then\ I$, where C is a Boolean expression and I is an instruction.
- The *block* $\{I_1, \ldots, I_n\}$, where $I_1, \ldots, I_n$ are instructions.

State variables can be presented as the left member of an assignment only in the action of a transition. Flow variables can be presented as the left member of an assignment only in the assertion. Instructions are interpreted in a slightly different way depending they are used in the actions or in the assertion. Let σ be the variable assignment before the firing of the transition $e : G \to P$. Applying the instruction P to the variable assignment σ consists in calculating a new variable assignment τ. The right hand side of assignments and conditional expressions are evaluated in the context of σ. Thus, the result does not depend on the order in which instructions of a block are applied. In other words, instructions of a block are applied in parallel. Let denote by $Update(P, \sigma)$ the variable assignment τ resulting from the application of the instruction P to σ. In case when the same state variable is affected several times and receives different values, an error is raised.

Let A be the assertion and τ the variable assignment obtained after the application of the action of a transition. Applying A consists in calculating a new variable assignment (of flow variables) π as follows. We start by setting all state variables in π to their values in τ: $\forall v \in S \pi(v) = \tau(v)$. Let D be a set of unevaluated flow variables, we start with $D = F$. Then,

- If A is an empty instruction, then π is left unchanged.
- If A is an assignment $v := E$, then if $\pi(E)$ can be evaluated in π, i.e. all variables of E have a value in π, then $\pi(v)$ is set to $\pi(E)$ and v is removed from D. An error is raised if the value of v has been already modified and is different from the calculated one.
- If A is a conditional assignment $if\ C\ then\ I$ and $\pi(C)$ can be evaluated in π and is true, then the instruction I is applied to π. Otherwise, π is left unchanged.
- If A is a block of instructions $\{I_1, \ldots, I_n\}$ then instructions $I_1, \ldots, I_n$ are repeatedly applied to π until there is no more possibility to assign a flow variable.

If after applying A to π there are unevaluated variables in D, then all these variables are set to their default values $\forall v \in D \pi(v) = reset(v)$ and A is applied to π in order to verify that all assignments are satisfied. If that is not true an error is raised. Let denote by $Propagate(A, \sigma)$ the

variable assignment resulting from the application of the instruction A to σ.

4.3 Reachability graph

Guarded Transition Systems are implicit representations of labeled Kripke structures, i.e. of graphs whose nodes are labeled by variable assignments and whose edges are labeled by events. This graph is constructed in the following way.

Assume that σ is the variable assignment just before the firing of a transition. Then, the firing of the transition transforms σ into the assignment $Fire(e : G \rightarrow P, A, \sigma)$ defined as follows:

$$Fire(e : G \rightarrow P, A, \sigma) = Propagate(A, Update(P, \sigma))$$

The so-called reachability graph $\Gamma = (\Sigma, \Theta)$ is the smallest Kripke structure verifying:

(1) $\sigma_0 = Propagate(A, \iota, \iota) \in \Sigma$. σ_0 is the initial state of the Kripke structure.
(2) If $\sigma \in \Sigma$ and $\exists t = \langle e, G, P \rangle \in T$, such that the guard G is verified in σ then the state $\tau = Fire(P, A, \iota, \sigma) \in \Sigma$ and the transition $(\sigma, e, \tau) \in \Theta$,

In special cases, the calculation of $\Gamma = (\Sigma, \Theta)$ may raise errors, when GTS are not well designed. Currently assessment tools detect modeling errors at the execution of the model.

4.4 Timed/Stochastic Guarded Transition Systems

A probabilistic time structure can be put on top of a Guarded Transition System so to get timed/stochastic models. The idea is to associate to each event:

- A delay which can be deterministic or stochastic and may depend on the state. When a transition labeled with the event becomes fireable at time t, a delay d is calculated and the transition is actually fired at time $t + d$ if it stays fireable from t to $t + d$.
- a weight, called expectation, used to determine the probability that the transition is fired in case of several transitions are fireable at the same date.

In the example given in Section 3, the transition $failure$ is a timed stochastic one, obeying to the exponential distribution with a failure rate 0.0005.

4.5 Example

The GTS model of the sub-system *TwoValves*, introduced in Section 3 and pictured Figure 2, is as follows:

```
class TwoValves
  Boolean v1.closed (init = true);
  Boolean v1.blocked (init = false);
  Real v1.rightFlow (reset = 0);
  Real v1.leftFlow (reset = 0);
  ...
  event v1.failure (delay = exponential(0.0005));
  event v2.failure (delay = exponential(0.0005));
  event ccf;
  transition
    ...
    ccf: not v1.blocked or not v2.blocked ->
        {
          if not v1.blocked then v1.blocked := true;
          if not v2.blocked then v2.blocked := true;
        }
    v1.failure: not v1.blocked -> v1.blocked := true;
    v2.failure: not v2.blocked -> v2.blocked := true;
  assertion
    v1.rightFlow := v2.leftFlow;
    v2.leftFlow := v1.rightFlow;
    ...
end
```

This GTS is obtained by flattening the corresponding AltaRica 3.0 model. All variables, parameters and events are prefixed by the name of the instantiated class. The synchronized transition *ccf* is flattened into a simple transition. Each bidirectional assignment $x :=: y$ is transformed into two simple assignments $x := y$ and $y := x$. A compiler automatically transforms AltaRica 3.0 models to GTS.

5. ASSESSMENT TOOLS

As shown Figure 1, the AltaRica 3.0 project provides a set of assessment tools. Some of them are already available and others are under specification or implementation. In the sequel, we will present first versions of four of them. All assessment tools take a GTS model as input.

5.1 Compiler to Fault Trees

Fault trees are widely used to perform Safety Analyses and some regulation authorities require to use them to support the certification process. From a GTS model, it is possible to generate corresponding Fault Trees (FT), i.e. to transform a states/transitions model into a set of Boolean formulae. It may seem inefficient at a first glance to use a states/transitions formalism to end up with a Fault Tree. However in practice, it is of great interest. It is easier and less time consuming to automatically generate FT from high-level models rather than create them from scratch. High-level models improves greatly the design, the sharing and the maintenance of models. The algorithm of compilation to Fault Trees for AltaRica Data-Flow, described in Rauzy (2002), can be extended to a general case of GTS. As illustrates Figure 3, this algorithm includes 3 steps:

(1) The GTS model is partitioned into independent GTS and an independent assertion.
(2) Reachability graphs of each independent GTS are calculated.
(3) Reachability graphs and the assertion are separately compiled into Boolean equations.

Partitioning is a key point of the algorithm that ensures its efficiency. In practice, components of a system fail in general in a relatively independent way. In that case a partitioning is possible. If the GTS is combinatorial, its compilation to Fault Trees is efficient and does not lose information.

The generated Fault Tree could be assessed with any Fault Tree calculation engine supporting Open-PSA format (Hibti et al., 2012). For example, XFTA (Rauzy, 2012) can be used to calculate minimal cutsets, events probabilities, importance factors, etc.

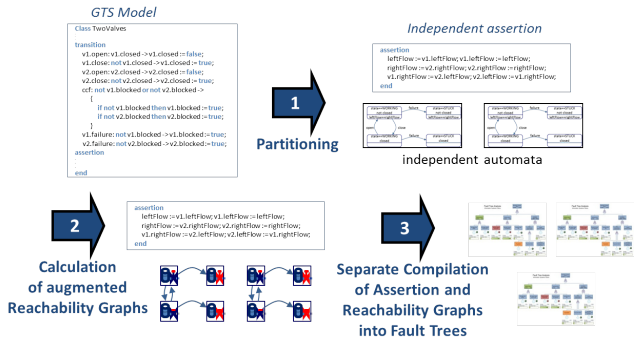

Fig. 3. Algorithm of compilation of GTS to FT

5.2 Stepwise simulator

The stepwise simulator enables to perform an interactive step by step simulation of a GTS model. This interactive tool can be very useful to debug models, to play different failure scenarios, etc. The stepwise simulator can be coupled with a graphical simulator as illustrated in Perrot et al. (2010). Graphical simulation of models can be used to perform virtual experiments on systems, via models, helping to better understand the system behavior.

The first version of the interactive stepwise simulator supports commands to display information about the simulated model (all variables and observers and their values, all transitions fireable in the current state, execution history, etc.) and to perform "actions" (to fire a transition, to cancel the last action, to restart the simulation, etc.).

5.3 Stochastic simulator

Stochastic simulation is a basic tool for safety analyses of systems. It provides fine results to calculate reliability indicators, even with complex systems. The principle is to run many pseudo-random histories of the behavior of the system and to make statistics on them. The stochastic simulator of the AltaRica 3.0 project has been designed by taking into account original features.

A generic mechanism is implemented to define specific delays. In general, simulation tools offer a lot of stochastic delays. Users have to choose the best matching one and fill out parameters. A few ones are widely used for safety/reliability analyses (e.g. exponential, Weibull, etc.), but others are generally difficult to use: difficult to understand or to fill out parameters. Due to this fact, the stochastic simulator implements the widely used delay functions, previously mentioned. For other specific ones, a generic mechanism allows to describe them by means of a set of points interpolated in a triangular way.

Compilation techniques are used to reduce computation time of simulations. In fact, the only limit of stochastic simulation is the number of histories, and their length, necessary to stabilize the measures. But with current computing technology, it is relatively easy to perform up to several million histories (of reasonable length). Beyond, the computation time gets an issue and with that respect, compilation technique may be of a great help. Thus the considered GTS model is translated into C++ classes, representing a set of instructions for the simulator engine. Then, they are compiled, with this simulator engine, to constitute the stochastic simulator of the system to study.

5.4 Markov chains Generator

The different reachable states of a system can be built from its GTS model. The state space of the system can be transformed into a Markov chain to compute the sojourn times / steady state probabilities of the different states of the system. The Markov chain formalism can be efficiently assessed by numerical methods such as developed in Rauzy (2004). It is a very straightforward method to compute mean values of the observers defined in the AltaRica 3.0 model. For instance, the availability of the system can be build from an observer which takes value 1 when system is working and 0 otherwise. Figure 4 illustrates the process.

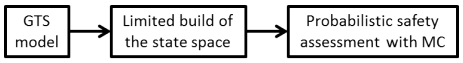

Fig. 4. Illustration of the Markov chain generation process

This computation method has two limits:

(1) the Markov hypothesis must hold for the system, ie. the transition rates between states must be constant,
(2) the size of the Markov model is subject to explode exponentially.

The hypothesis (2) is difficult to overcome. A method has been developed to limit the construction of the state space. It consists in selecting the most influential states toward the given reward to assess, thus giving accurate results while drastically limiting the size of the state space. The influence of the hypothesis (1) really depends on the system modeled. It is usually valid for safety assessment, and it gives good results to quickly assess an AltaRica model with the Markov chain generator. It is particularly valid while designing the system architecture, when engineers need to assess and compare several models.

6. RELATED WORKS

Two approaches for (high-level) Model-Based Safety Assessment can be found in literature. The first one consists in creating extensions of high-level modeling languages used in other domains. The second approach consists in defining domain specific languages, dedicated to Safety Analyses.

In the first category, we can find Feiler and Rugina (2007) who added an Error Model annex to the modeling formalism for embedded real-time systems AADL. In the same way the HiP-HOPS workbench (Pasquini et al., 1999) enables to add reliability data to models imported from different modeling tools: Matlab/SIMULINK, Eclipse-based UML tools, etc., and then to automatically generate Fault Trees and FMEA tables.

Similarly, translations have been defined from specialized UML/SysML models to Fault Trees or Petri nets (Bernardi et al., 2002). In David et al. (2010), the functional design phase, using SysML, is combined with commonly used reliability techniques (i.e. FMEA and construction of AltaRica Data-Flow models).

In the second category, we can find Figaro (Bouissou et al., 1991), developed by EDF R&D. It is a textual modeling language dedicated to dependability assessment of complex systems. It combines object-orientated languages features, such as inheritance, and first order production rules (interaction and occurrence rules). It is used as a description language to create knowledge bases for the workbench KB3 (Bouissou, 2005), to automatically perform systems dependability assessment: Monte-Carlo simulation, Markov Chain generation, quantification and generation of critical sequences, etc.

In between the two categories, we can find SAML (Safety Analysis Modeling Language) (Güdemann and Ortmeier, 2010), which is a synchronous language. It expresses a model in terms of finite stochastic state automata and its semantics is defined as Markov decision process. S3E is a design and verification environment focused on SAML models. It provides a stochastic simulator and translators to the input languages of the model-checkers PRISM and NuSMV.

7. CONCLUSION

In this article, we presented the AltaRica 3.0 Project, which aims to develop a complete set of tools, under a free license, to create, edit, check, simulate, debug and assess models. The new version of AltaRica modeling language increases the expressive power of the previous one without decreasing the efficiency of assessment algorithms. It makes it possible to model looped systems and components with bidirectional flows. First versions of assessment tools include: a compiler to Fault Trees, a compiler to Markov chains, a stepwise and a stochastic simulators.

Forthcoming works will focus, amongst others, on the implementation of other assessment tools (i.e. a model-checker and a reliability allocation module) and the development of modeling methodology and pedagogical materials, including benchmark tests.

REFERENCES

Arnold, A., Griffault, A., Point, G., and Rauzy, A. (2000). The altarica language and its semantics. *Fundamenta Informaticae*, 34, 109–124.

Bernard, R., Aubert, J.J., Bieber, P., Merlini, C., and Metge, S. (2007). Experiments in model-based safety analysis: flight controls. In *Proceedings of IFAC workshop on Dependable Control of Discrete Systems, Cachan*.

Bernardi, S., Donatelli, S., and Merseguer, J. (2002). From uml sequence diagrams and statecharts to analyzable petri net models. In *In Proceedings of the Third International Workshop on Software on Performance*.

Bieber, P., Blanquart, J.P., Durrieu, G., Lesens, D., Lucotte, J., Tardy, F., Turin, M., Seguin, C., and Conquet, E. (2008). Integration of formal fault analysis in assert: Case studies and lessons learnt. In *Proceedings of 4th European Congress Embedded Real Time Software, ERTS 2008*. Toulouse (France).

Boiteau, M., Dutuit, Y., Rauzy, A., and Signoret, J.P. (2006). The altarica data-flow language in use: Assessment of production availability of a multistates system. *Reliability Engineering and System Safety*, 91, 747–755.

Bouissou, M. (2005). Automated dependability analysis of complex systems with the kb3 workbench: the experience of edf r&d. In *Proceedings of the International Conference on Energy and Environment*.

Bouissou, M., Bouhadana, H., Bannelier, M., and Villatte, N. (1991). Knowledge modelling and reliability processing: presentation of the figaro modelling language and associated tools. In *Proceedings of Safecomp'91*.

David, P., Idasiak, V., and Kratz, F. (2010). Reliability study of complex physical systems using sysml. *Reliability Engineering and System Safety*, 431–450.

Feiler, P. and Rugina, A. (2007). Dependability modeling with the architecture analysis & design language (aadl). Technical report, Carnegie Mellon University.

Güdemann, M. and Ortmeier, F. (2010). A framework for qualitative and quantitative model-based safety analysis. In *Proceedings of 12th High Assurance System Engineering Symposium*, 132141.

Hibti, M., Friedlhuber, T., and Rauzy, A. (2012). Overview of the open psa platform. In R. Virolainen (ed.), *Proceedings of International Joint Conference PSAM'11/ESREL'12*.

Noble, J., Taivalsaari, A., and Moore, I. (1999). *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag.

Pasquini, A., Papadopoulos, Y., and McDermid, J. (1999). Hierarchically performed hazard origin and propagation studies. *Computer Safety, Reliability and Security*, 1698 of LNCS, 688–688.

Perrot, B., Prosvirnova, T., Rauzy, A., d'Izarn, J.P.S., and Schoening, R. (2010). Expériences de couplages de modèles AltaRica avec des interfaces métiers. In E. Fadier (ed.), *Actes du congrès LambdaMu'17 (actes électroniques)*. IMdR.

Point, G. and Rauzy, A. (1999). AltaRica: Constraint automata as a description language. *Journal Européen des Systèmes Automatisés*, 33(8–9), 1033–1052.

Prosvirnova, T. and Rauzy, A. (2012). Guarded transition systems: Pivot modelling formalism for safety analysis. In J. Barbet (ed.), *Actes du Congrès Lambda-Mu 18*.

Rauzy, A. (2002). Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78, 1–12.

Rauzy, A. (2004). An experimental study on iterative methods to compute transient solutions of large markov models. *Reliability Engineering & System Safety*, 86(1), 105–115.

Rauzy, A. (2008). Guarded transition systems: a new states/events formalism for reliability studies. *Journal of Risk and Reliability*, 222(4), 495–505.

Rauzy, A. (2012). Anatomy of an efficient fault tree assessment engine. In R. Virolainen (ed.), *Proceedings of International Joint Conference PSAM'11/ESREL'12*.