ALTARICA WIZARD: AN INTEGRATED MODELING AND SIMULATION ENVIRONMENT FOR ALTARICA 3.0

Michel Batteux, Tatiana Prosvirnova, Antoine Rauzy

ALTARICA WIZARD : UN ENVIRONEMENT INTEGRE DE MODELISATION ET SIMULATION POUR ALTARICA 3.0

ALTARICA WIZARD: AN INTEGRATED MODELING AND SIMULATION ENVIRONMENT FOR ALTARICA 3.0

Michel Batteux
IRT SystemX
Saclay, France

Tatiana Prosvirnova
CentraleSupélec
Gif-sur-Yvette, France

Antoine Rauzy
NTNU
Trondheim, Norway

Résumé

Cette communication propose au lecteur une visite guidée de AltaRica Wizard, un nouvel environnement intégré de modélisation et simulation pour AltaRica 3.0.
AltaRica 3.0 est un langage de modélisation orienté objet dédié aux analyses probabilistes du risque de systèmes complexes. AltaRica Wizard fourni l'ensemble des fonctionnalités attendues d'un environnement de conception pour un tel langage. Il permet de gérer les modèles AltaRica 3.0 en mode projets, qui rassemblent les différents fichiers source d'un modèle, ainsi que les fichiers de configuration et de résultats issus des outils d'évaluation. Il facilite l'écriture, la modification, la correction et l'évaluation des modèles en fournissant une interface graphique unifiée.
AltaRica Wizard est actuellement relié à trois chaînes outillées pour évaluer les modèles AltaRica 3.0 : un simulateur pas-à-pas, un générateur d'arbres de défaillances ainsi qu'un simulateur stochastique.

Summary

This article invites the reader to a guided tour of AltaRica Wizard, a new integrated modeling and simulation environment for AltaRica 3.0.
AltaRica 3.0 is an object-oriented modeling language dedicated to reliability engineering of complex technical systems. AltaRica Wizard provides the expected functionalities of a development environment for such a language. It manages AltaRica 3.0 models into projects that gather the different source files of a model, as well as the configuration and result files of assessment tools. It eases authoring, modifying, debugging and assessing of models by providing a unique, unified graphical interface.
At the time we write these lines AltaRica Wizard embeds three tool chains to assess AltaRica 3.0 models: a stepwise simulator, a compiler to fault trees and a stochastic simulator.

Introduction

This article invites the reader to a guided tour of AltaRica Wizard, a new integrated modeling and simulation environment for AltaRica 3.0. AltaRica 3.0 is an object-oriented modeling language dedicated to reliability engineering of complex technical systems, see e.g. (Batteux & al. 2018, c) for an introduction. AltaRica 3.0 is a very powerful language, more expressive for example than Figaro (Bouissou & al. 1991), extensions of stochastic Petri nets such as the one implemented in the GRIF tool (Signoret & al. 2013) or AltaRica Data-Flow (Boiteau & al. 2006), the previous version of the language. It comes with a versatile set of assessment tools. As of today, AltaRica 3.0 is probably the most advance available technology to perform probabilistic risk and safety analyses.

AltaRica 3.0 models are stochastic discrete event systems. Their design and assessment go through several steps, requiring typically to create auxiliary files describing where the source files are located, what is the chosen mission profile, which performance indicators to calculate, what kind of statistics to get on these indicators, where the result files should be located and so on. AltaRica Wizard aims at simplifying these tasks by providing a unique environment from which everything can be done in a user-friendly way.

The design of AltaRica Wizard is strongly inspired from integrated development environments for programming languages such as Python, C++ or Java. Just as these tools, AltaRica Wizard is designed to maximize analyst productivity by providing tight-knit components with similar user interfaces. It presents a single program in which all modeling and simulation is done. It provides features for authoring, modifying, debugging and assessing AltaRica 3.0 models by means of different technologies.

AltaRica Wizard provides thus the expected functionalities of a code editor: syntax highlighting, management of copy and paste, line numbering and so on. It manages AltaRica 3.0 models into projects that gather the different source files of a model, as well as the files generated by assessment tools. At the time we write these lines, AltaRica Wizard embeds three tool chains to assess AltaRica 3.0 models:

- A graphical stepwise simulator, also called stepper. This tool is used to animate models. It can be seen also as the equivalent of debuggers of programming languages and plays a central role in the validation of models (Batteux & al. 2018, a).
- A compiler to fault trees at Open-AltaRica format (Epstein & al 2009). This compiler is chained with the powerful fault tree engine XFTA (Rauzy 2012).
- A stochastic simulator. Stochastic simulation is a versatile tool for performance engineering, see e.g. (Zio 2013)

A compiler to Markov chains and a generator of critical sequences will be released in the coming months. Both tools are currently under test

AltaRica Wizard is designed by the nonprofit AltaRica Association which has the full copyright of the tool. It is developed in C++ using the Qt5 framework by the Qt Company. This makes the development very efficient and the program portable across different platforms (Windows, Linux, MacOS). This choice has some legal consequences the user should know and accept: AltaRica Wizard can be used, free of charge, for any academic or commercial purposes. It is however distributed as is in the hope that it will be useful, but without any warranty of any kind. Assessment tools embedded in AltaRica Wizard are the copyrights of their respective owners. Most of them are either developed at IRT System X, in the framework of the Open-AltaRica project (which is supported by EADS Apsys, Safran and Thales) or by the AltaRica Association. The user is thus invited to check that he or she has all rights to use these tools for the purpose he or she uses them. As of today, and as permitted by the LGLPv3 license of Qt5, AltaRica Wizard is not an open source tool. This situation will last until the AltaRica Association has established with its partners the economic model of AltaRica tools. The mid-term objective is to put in place a dual licensing model, just as the Qt framework.

The contribution of this article is to provide the reader with a snapshot of AltaRica Wizard features. Doing so, it highlights also AltaRica 3.0 expressive power, elegance and simplicity of use.
The remainder of this article is organized as follows. The next section gives an overview of the AltaRica 3.0 technology. The following sections present successively each of the assessment tools. Finally, the last section concludes the article and gives some perspectives on the forthcoming developments.

The AltaRica 3.0 Technology at a glance

Figure ***1*** gives a global picture of the AltaRica 3.0 technology, as of today.

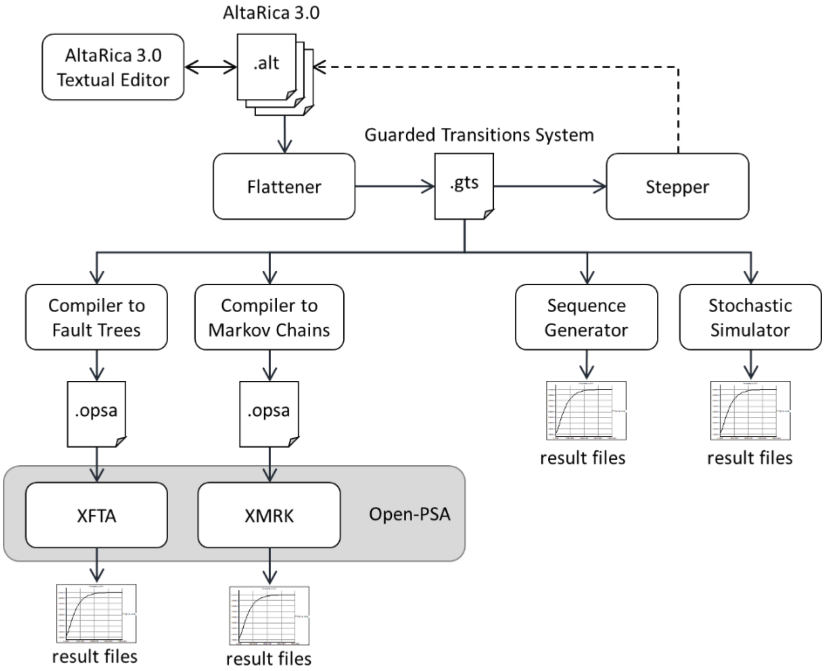


Figure 1. The AltaRica 3.0 Technology at a glance

Projects

AltaRica Wizard manages models through projects. A project contains one or more files organized into folders. Files of a project can be not only AltaRica source files (having usually the extension ".alt"), but also files used to configure assessment tools and result files generated by these tools.

Projects can be created, populated with folders and files, saved and (re)loaded from AltaRica Wizard. Files and folders of a project are those of the underlying operating system. In particular, the project itself is located into a folder of the underlying operating system. All paths to folders and files are considered relatively to the project folder and the file describing the project is saved into this folder. In this way, projects can be moved from one place to another one just by moving their folder. The same principle applies indeed to project copy: to duplicate a project it suffices to copy the folder that contains this project.

Folders are automatically added to and removed from projects when adding and removing files. This means that a project contains a folder if and only if this folder contains a file of the project (possibly in located in a sub-folder, a sub-sub-folder … of that folder). *A contrario*, sub-folders of the project folder do not belong automatically to the project. Only those containing a file of the project do.

Projects make it possible to design libraries of reusable modeling components and to assemble them to create models, or simply to split a model into several files so to make it easier to design and to maintain. Projects make also it possible to add to the model files recording reliability data. These contains typically cumulative distribution functions associated with failures of basic components. They may be automatically extracted from reliability data bases.

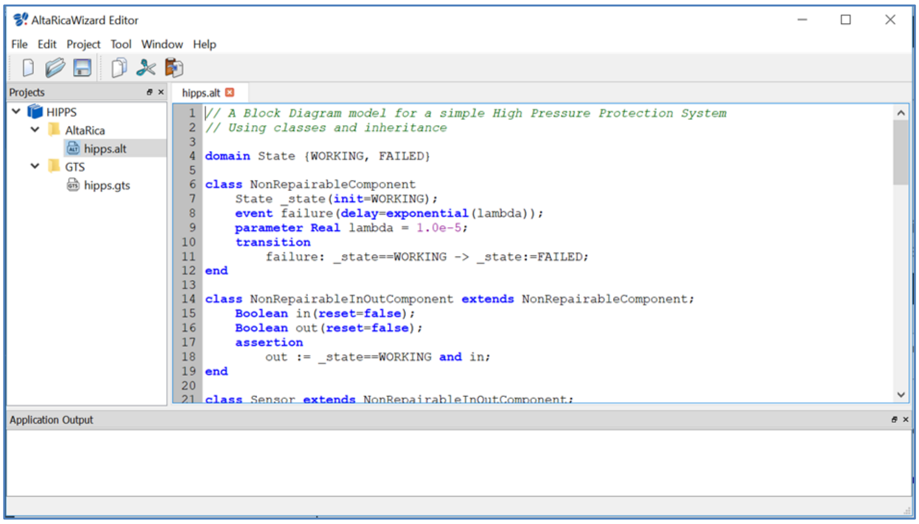
Authoring and Flattening Process

AltaRica models are authored in the AltaRica Wizard code editor. Figure 2 shows a screenshot of the code editor. This editor is very classical. Its commands (or opening and saving files, for cut and paste...) are the usual one. Its look and feel of the code editor (font type, size and color, syntax highlighting …) can be modifier at will.

The first step of all AltaRica 3.0 assessment tools consists in compiling the AltaRica model into a guarded transitions system. The AltaRica model is made of all declarations appearing in all AltaRica files of the current project. A guarded transitions system (GTS) can be seen a “flat” AltaRica model, i.e. a model made of a single block. This is the reason why this compilation process is also called flattening and the compiler a flattener.
For efficiency reasons however, guarded transitions systems, as implemented by the AltaRica 3.0 tools, are

slightly different from flat AltaRica models. Nevertheless, the flattening process preserves the semantics of the original model. For this reason, the analyst does not need to look at the generated guarded transitions system. We could even have hide it completely. The general philosophy of AltaRica is however to give the analyst access to all intermediate files. This approach makes it possible to introduce intermediate steps performed outside of the AltaRica Wizard environment.

The flattening process serves actually two purposes: first, as the first step of any assessment, it is in charge of checking the model against syntax errors, missing declarations, typing problems and the like. Second, it simplifies greatly the task of assessment tools by instantiating classes, resolving references and so on.



```
// A Block Diagram model for a simple High Pressure Protection System
// Using classes and inheritance

domain State {WORKING, FAILED}

class NonRepairableComponent
    State _state(init=WORKING);
    event failure(delay=exponential(lambda));
    parameter Real lambda = 1.0e-5;
    transition
        failure: _state==WORKING -> _state:=FAILED;
end

class NonRepairableInOutComponent extends NonRepairableComponent;
    Boolean in(reset=false);
    Boolean out(reset=false);
    assertion
        out := _state==WORKING and in;
end

class Sensor extends NonRepairableInOutComponent;
```

Figure 2. Screenshot of the AltaRica Wizard code editor

Assessment Tools

As already said in the introduction, at the time we write these lines, AltaRica Wizard embeds three tool chains to assess AltaRica 3.0 models:

- A graphical stepwise simulator, also called stepper.
- A compiler into fault trees at Open-PSA format.
- A stochastic simulator.

We shall describe by means of examples the use of each of these tools in the next sections.

Two additional tool chains will be released in the forthcoming months:

- A compiler to Markov chains. This compiler produces exact or approximated Markov chains according to the principles defined in reference (Brameret & al. 2015). This makes it possible to assess very large model without suffering, at least to some extent, of the combinatorial explosion of the numbers of states and transitions. The Markov chains generated by the compiler are then assessed by means of the XMRK calculation engine. This program is derived from previous work of one of the author (Rauzy 2004).
- A generator of critical sequences relying of efficient algorithmic techniques, not published yet.

This versatile set of tool chains, all implementing state-of-the-art algorithms, makes it possible:

- To use the best suited tool for each analysis purpose. This is of paramount importance as the assessment of probabilistic performance indicators is provably computationally hard, see (Rauzy 2018) for a detailed discussion on this topic
- To validate models by means of multiple experiments. For instance, models can be checked by means of stepwise simulation and critical sequence generation before assessing them by means of Monte-Carlo simulation.
- To cross-check results, again by means of multiple experiments. For instance, results obtained by means of a compilation into fault trees can be cross-checked by means of a stochastic simulation or a compilation into Markov chains.

We shall now present tool chains already available in the current distribution.

Stepwise Simulation

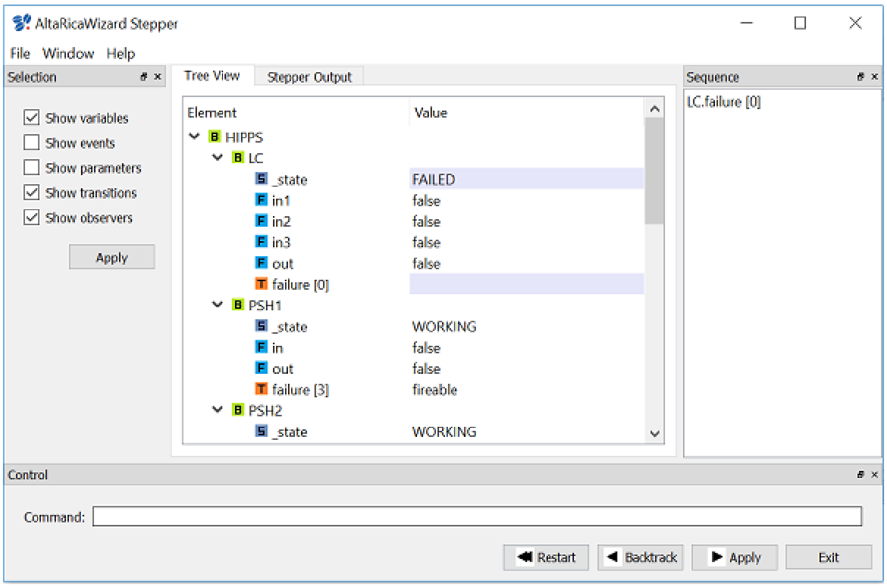
The stepper, or stepwise simulator, makes it possible to animate AltaRica 3.0 models by firing and backtracking transitions, and looking at the value of variables and observers. For this reason, it is very useful to validate models. It plays actually a very similar role as debuggers such as gbd (Matloff & Salzman 2008) of programming languages. The stepper is a command interpreter. There are commands to print out the value of variables, observers and enabled transitions, to fire a given transition, to undo last the firing, to restart a simulation from scratch, to save and reload as sequence of transition firings and so on.

AltaRica Wizard provides a graphical interface that encapsulates the stepper. This interface makes it easier to browse models and to enter commands (basically, the analyst does not need to remember the syntax of commands). Figure 3 shows a screenshot of this graphical interface.

The central panel of this interface shows a tree-like view of the model. Elements (variables, transitions, observers...) are organized by blocks, sub-blocks, sub-sub-blocks and so on. Values of variables and observers as well as status of transitions are displayed in a clear way. Block items are foldable and expandable so to ease the visualization of large models. Enabled transitions can be fired just by double clicking on the corresponding line.

The currently distributed version of the stepper implements the regular semantics of AltaRica 3.0, except on one point: the current version ignores delays and expectations associated with events and transitions. Consequently, some of the sequences of events that are enabled with the stepper are impossible to fire with other simulation-based tools such as the stochastic simulator. Assume for instance that, at a given step of the simulation, two transitions have their guards satisfied, and are thus enabled. Assume moreover that the events of the first and second transitions are associated respectively with an exponential distribution and a Dirac(0) distribution. Then, according to the temporized semantics of AltaRica 3.0, only the second transition is actually enabled. The stepper allows however to fire both. It is thus up to the analyst to ensure that the sequences of events he or she fires are accepted by the timed semantics of AltaRica 3.0.

This inconvenient has been recently solved by introducing an abstract semantics for AltaRica 3.0 (Batteux & al. 2018, a). This abstract semantics verifies that each concrete execution can be simulated by a unique abstract execution and each abstract execution corresponds to at least one concrete execution. The next release of AltaRica Wizard will implement this abstract semantics, making the stepper even more interesting to validate AltaRica 3.0 models.



☑ Show variables
☐ Show events
☐ Show parameters
☑ Show transitions
☑ Show observers

Element	Value
HIPPS	
LC	
_state	FAILED
in1	false
in2	false
in3	false
out	false
failure [0]	
PSH1	
_state	WORKING
in	false
out	false
failure [3]	fireable
PSH2	
_state	WORKING

LC.failure [0]

Command:

Figure 3. Screenshot of the AltaRica Wizard Stepper

Compilation into Fault Trees

The compilation into fault trees made the initial success of AltaRica. AltaRica makes it possible reduce the gap between system specifications and safety model. This gap makes safety models hard to share with stakeholders and, even more importantly, to maintain through the life-cycle of systems. Another advantage of AltaRica is that the same model can be used to assess several safety goals. The first efficient algorithm to compile AltaRica Data-Flow models into fault trees has been proposed in reference (Rauzy 2002). This algorithm has been lifted-up to AltaRica 3.0 models (Prosvirnova & Rauzy 2015) and improved recently (Batteux & al. 2018, b).

The idea of the compilation algorithm is to map each sequence of transition firings that goes from the initial state to a failure state onto the conjunction of the events labeling the fired transition. The set of failure sequences is mapped onto the disjunction of these conjunction.
Applied directly, this idea would suffer from the combinatorial explosion of the number of a failure sequences. To avoid this problem, the model is split into independent parts, thanks to static analysis techniques. Disjunctions of conjunctions are then calculated locally on each independent part and then assembled into a fault tree.

In general, the resulting fault tree does not look like what a human would have designed, even though minimal cutsets are the same. This is the reason why, visualization of fault tree does not play a fundamental role in the assessment process. On the contrary, browsing minimal cutsets may be of great interest. As of today, we rely usually on the tool "Arbre Analyste" developed by E. Clément (Clément & al. 2014) to visualize fault trees when needed.

The compiler to fault trees needs only a target file to be ran. It interface is thus minimal. The generated fault tree is generated at the Open-PSA format (Epstein & Rauzy 2008). This format is read by fault tree assessment tools, including Arbre Analyste, SCRAM, GRIF and XFTA.

AltaRica Wizard proposes a (limited) graphical interface to work with XFTA. A specific dialog makes it possible to create a XFTA command file and then to launch XFTA with this command file. A screenshot of this interface is given Figure 4.

This dialog dialog is restricted to relatively simple, although powerful, scenarios:

- The top event for the analysis is specified and minimal cutsets are calculated. It is possible to define cutoffs (maximum order and/or minimum probability) for cutsets.
- Minimal cutsets can be then printed-out into a file (together with their order, probability and contribution to the top-event probability).

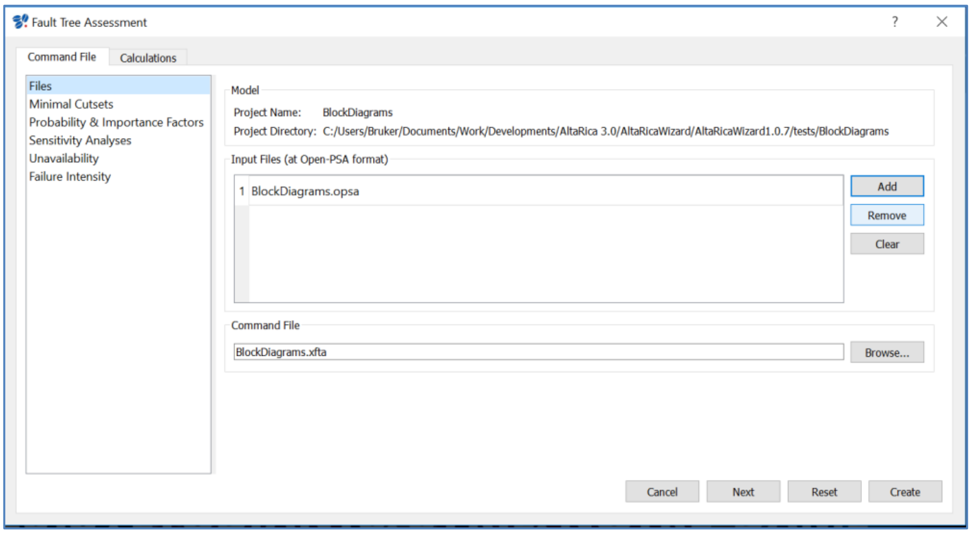
Various calculations of indicators can then be performed from the extract minimal cutsets:

- Top event probability and importance factors of basic events at a given mission time.
- Sensitivity analysis on the top event probability at a given mission time.
- Evolution of the system unavailability (probability of failure on demand), average unavailability and safety integrity levels (for low demand safety systems) throughout a period of time.

- Evolution of the conditional failure intensity, average conditional failure intensity, probability of failure per hour, approximate system reliability and safety integrity levels (for high demand safety systems) throughout a period of time.

The results of these calculations are saved into text files that can be exploited outside of AltaRicaWizard (typically in spreadsheet tools).

Mathematical definitions and calculation algorithms can be found in reference (Rauzy, 2014).



Project Name: BlockDiagrams

Project Directory: C:/Users/Bruker/Documents/Work/Developments/AltaRica 3.0/AltaRicaWizard/AltaRicaWizard1.0.7/tests/BlockDiagrams

1 BlockDiagrams.opsa

BlockDiagrams.xfta

Figure 4. Screenshot of the XFTA interface

Stochastic Simulation

Principle

Stochastic simulation is a versatile tool to assess performance indicators discrete event systems (Zio 2013). The principle of stochastic, or Monte-Carlo, simulation is quite simple. It consists in drawing at random a sample of executions of the model, to observe a number of quantities during these executions, and to make statistics on these observations. To apply this principle, one has to answer the following questions:

- What are the quantities to observe and how to define them?
- When to observe these quantities?
- What kind of statistics should be made on these quantities?
- What is good size of the sample so to get sufficiently robust results?

Quantities to observe depend indeed on the performance indicators one wants to obtain. In AltaRica 3.0, they are defined in two steps: first, one declares observers in the model. These observers can be either symbolic (Boolean or symbolic constants) or numerical (integers or real numbers). In both cases, their values evolve through an execution. Second, one declares indicators in the “indicator declaration file” of the stochastic simulator. This file is automatically generated by AltaRica Wizard.
An indicator defines what to measure about the evolution of an observer, i.e. on which quantity to make statistics.
Each execution runs from time 0 to a given mission time T. Statistics are indeed made on the values of indicators at time T. It may be interesting to make statistics on their values, at least for some of them, at intermediate times 0 ≤ t1 < t2 <... < tk ≤ T. The mission time as well as possible intermediate times are described into the mission description file. This file is automatically generated by AltaRica Wizard. It contains also the number of executions the analyst wants to perform.

Process

Figure 5 shows the AltaRica 3.0 stochastic simulation process.

The first step consists in transforming this model into an equivalent guarded transitions system. This step ***is*** achieved by the AltaRica 3.0 flattener. Guarded transitions systems are usually stored in files with extension ``.gts".

The second step consists in generating a C++ program from the guarded transitions system and the indicator description file. This C++ program is a stochastic simulator specialized for the model and the indicators. This step is achieved by the AltaRica 3.0 stochastic simulator generator. Technically, the indicator description file is a XML file, which usually has the extension ``.idf".

The third step consists in compiling the C++ file into an executable file. This step is achieved by a public domain portable C++ compiler (in the current version, mingw).

The fourth and last step consists in running the stochastic simulation, i.e. in executing the stochastic simulator. The executable stochastic simulator takes the mission description file as input. Consequently, it is possible to use the same simulator for different missions, i.e. for different number of executions, different mission time and intermediate observation dates. Technically, the mission description file is a XML file, which usually has the extension ``.mdf".

The AltaRica Wizard integrated modeling environment makes it possible to perform these four steps in a (relatively) transparent way. In particular, the analyst does not have to learn the syntax of indicator and mission description files, nor how to call the different tools involved in the process.

The results of the stochastic simulation can be written out in different formats: a textual format, the ``.csv" format that eases the loading into a spreadsheet tool and a XML format that eases the loading into a post-treatment tool.

Indicators

They are two types of observers with respect to the definition of indicators:

- symbolic observers, which include both Boolean observers and observers taking their value into a finite set of symbolic constants,
- and numerical observers, which include both integer-valued and real valued observers.

We shall examine them in a row.

Indicators for symbolic observers are actually defined for the observer and a particular value of the observer.
Let O be a symbolic observer and v be a constant of the domain of O. Finally, let t be the time at which the indicator is assessed. The following indicators can be defined for O, v and t.

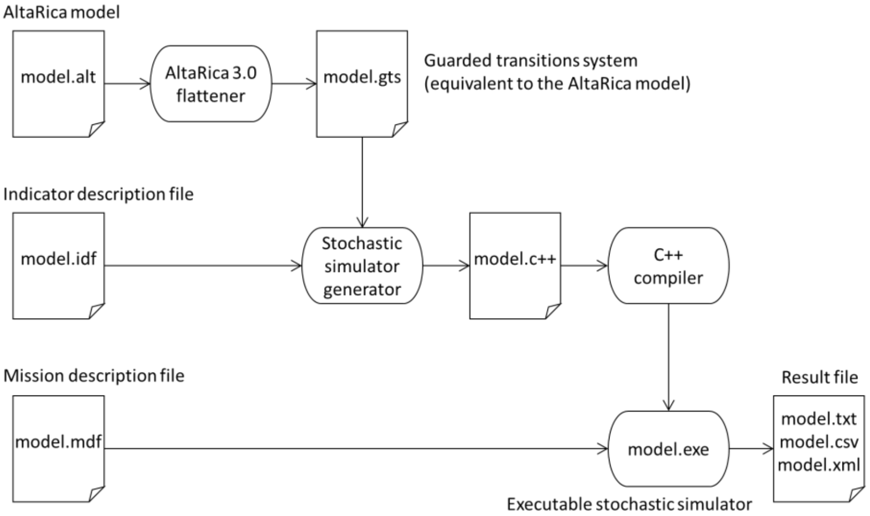


Figure 5. The AltaRica 3.0 stochastic simulation process

- has-value(O, v, t): This Boolean indicator measures the proportion of executions in which the observer O takes the value v at time t.
- had-value(O, v, t): This Boolean indicator measures the proportion of executions in which the observer O takes at least once the value v from time 0 to time t.
- sojourn-time(O, v, t): This numerical indicator measures the time spent by the observer O at the value v from time 0 to time t. This time is averaged over the executions.
- first-occurrence-date(O, v, t): This numerical indicator measures the first date at which the observer O takes the value v from time 0 to time t. This time is averaged over the executions in which the observer O takes the value v. The executions in which the observer O never takes the value v are not considered.
- number-of-occurrences(O, v, t): This numerical indicator measures the number of times the observer O takes the value v from time 0 to time t. This time is averaged over the executions.
- mean-time-between-occurrences(O, v, t): This numerical indicator measures the mean time between two successive dates at which the observer O takes the value v from time 0 to time t. This mean time is averaged over the executions in which the observer O takes the value v at least twice. The executions in which the observer O takes 0 or 1 time the value v are not considered.

The situation is much simpler for indicators built over numerical observers as the values of these indicators cannot be considered individually. Let O be a numerical observer and t be the time at which the indicator is assessed. The following indicators can be defined for O and t.

- value(O, t): This Boolean indicator measures the value the observer O at time t. This value is averaged over the executions.
- mean-value(O, t): This Boolean indicator measures the mean value the observer O from time 0 to time t. This value is averaged over the executions.

Note that if one is interested in the average time the system spends in a state where the value of a given numerical variable is over a given threshold, then it is possible to define a Boolean observer that describes exactly this situation.

Statistics

The following statistical measures can be obtained on an indicator I after n realizations I1, I2, ... In obtained through n random executions of the model (which statistics should be made on which indicators is described in the indicator description file).

- The empirical mean value of the indicator.
- The empirical standard-deviation of the indicator.
- The empirical 95% confidence range of the indicator.

In addition to the above statistical measures, it is possible to calculate the empirical distribution of the indicator. The idea is as follows.

- Sort the values I1, I2, ... In in increasing order.
- Divide the n values into k buckets of nearly equal size. Each bucket contains thus about n/k values (10 or 20 are typical values for k).
- Calculate for each of the buckets the minimum, the mean and the maximum values in the bucket.

Note that maximum values of each bucket are the k quantiles of the indicator.

As it is in general not possible to store all of the values, minimum, the mean and the maximum values of buckets are calculated on the fly.
This technique gives in general good estimations.

Discussion

The stochastic simulation process described in this section may seem a bit complex at a first glance. However, it is both versatile and efficient:

- Available indicators provide a very rich palette of measures that make it possible to characterize a wide set of properties of the system.

- These indicators are defined outside the model which avoid polluting the model with constructs that are not related to the system behavior.
- Nevertheless, indicators are stored into description files so to keep track of experimental protocols and to be able to replay them.
- Mission profiles are also stored into description files, for the very same reasons.
- The stochastic simulation is performed by compiling the AltaRica model into an executable program (for a given set of indicators and via the compilation into GTS, then C++). This makes it extremely efficient. The compilation process itself is efficient as both the AltaRica compiler and the C++ compiler are.

Thanks to AltaRica Wizard this process is greatly facilitated and made transparent for the analyst: compilers are installed with the AltaRica Wizard distribution and description files are automatically generated by filling forms through the graphical interface.

Conclusion

The AltaRica 3.0 technology is now mature and ready for industrial deployment. A number of experiments have been performed, showing its efficiency (several publications are on the way). A lot has been achieved the last four years, with eventually relatively limited means: the AltaRica development team never counted more than three members at a time, and none of them was working full time on the project. The sponsoring by Safran of the chair Blériot-Fabre at CentraleSupélec and the support of Apsys, Safran and Thalès to the OpenAltaRica project have been decisive: they simply made the story possible.

Of course, the AltaRica 3.0 language specification will probably evolve, the AltaRica Wizard environment will be improved, the assessment tools will be given more possibilities and will be made more efficient in the future. But the core of the technology is here that outperforms already all other implementations of the so-called "model-based" approach in reliability engineering and system safety.

To conclude this article, we would like to say few words about mention graphical representations of models. There is a debate about this question each time the model-based approach in reliability engineering is discussed. Our experience goes against the tide: we believe that, although graphical representations are very useful as communication mean, pure graphical modeling is counterproductive in most of the cases. Analysts spend eventually more time in making their drawing "elegant" than in thinking about the system they are studying and the mathematical object they are designing, namely the model. If the system under study is complex, models of this system must be complex as well, or at least simplex in the sense given to this term by Berthoz (Berthoz 2012). In a word, complexity cannot vanish. Modeling is not a magic wand. But as soon a model gets complex, it is not possible to represent it fully graphically. Models are mathematical artifacts. Complex models are complex mathematical artifacts which are better represented by texts. With that respect, we must draw lessons from decades of software engineering: all attempts to design graphical programming languages failed, but for pedagogical purposes (e.g. the Scratch language from the MIT). Drawings are definitely useful to understand the global architecture of a program, but the code remains the ultimate reference. Moreover, drawings (such as UML class diagrams) are never more useful than when they are automatically generated from the code (and not the reverse). We strongly believe that models engineering will follow the same path. At the end of the day, mankind invented writing because drawing was not sufficient to transmit complex thoughts.

References

Michel Batteux, Tatiana Prosvirnova and Antoine Rauzy. Enhancement of the AltaRica 3.0 stepwise simulator by introducing an abstract notion of time. Safe Societies in a Changing World, Proceedings of European Safety and Reliability Conference (ESREL 2018). Trondheim, Norway. June, 2018.

Michel Batteux, Tatiana Prosvirnova and Antoine Rauzy. Advances in the simplification of Fault Trees automatically generated from AltaRica 3.0 models. Safe Societies in a Changing World, proceedings of European Safety and Reliability Conference (ESREL 2018). Trondheim, Norway. June, 2018.

Michel Batteux, Tatiana Prosvirnova and Antoine Rauzy. AltaRica 3.0 in 10 Patterns: Submitted to International Journal of Critical Computer-Based Systems. Inderscience Publishers. 2018.

Alain Berthoz. Simplexity: Simplifying Principles for a Complex World. Yale University Press. New Haven, CT, USA. ISBN 978-0300169348. 2012.

Marie Boiteau, Yves Dutuit, Antoine Rauzy and Jean-Pierre Signoret. The AltaRica Data-Flow Language in Use: Assessment of Production Availability of a MultiStates System. Reliability Engineering and System Safety. Elsevier. 91:7. pp. 747–755. July, 2006. doi:10.1016/j.ress.2004.12.004.

Marc Bouissou, Henri Bouhadana, Marc Bannelier and Nathalie Villatte. Knowledge modelling and reliability processing: presentation of the FIGARO language and of associated tools. Proceedings of SAFECOMP'91, IFAC International Conference on Safety of Computer Control Systems. Johan F. Lindeberg Ed. Pergamon Press. ISBN 0-08-041697-7. pp. 69–75. Trondheim, Norway. 1991.

Pierre-Antoine Brameret, Antoine Rauzy and Jean-Marc Roussel. Automated generation of partial Markov chain from high level descriptions. Reliability Engineering and System Safety. Elsevier. 139. pp. 179–187. July, 2015. doi:10.1016/j.ress.2015.02.009.

Emmanuel Clément, Thierry Thomas and Antoine Rauzy. Arbre Analyste : un outil d'arbres de défaillances respectant le standard Open-PSA et utilisant le moteur XFTA. Actes du congrès Lambda-Mu 19 (actes électroniques). Institut pour la Maîtrise des Risques. ISBN 978-2-35147-037-4. Dijon, France. October, 2014.

Steven Epstein and Antoine Rauzy. Open-PSA Model Exchange format, version 2.0d. 2008. http://www.open-psa.org.

Norman Matloff and Peter Jay Salzman. The Art of Debugging with GDB, DDD, and Eclipse. No Starch Press. San Fransisco, CA, USA. ISBN 978-1593271749. 2008.

Tatiana Prosvirnova and Antoine Rauzy. Automated generation of Minimal Cutsets from AltaRica 3.0 models. International Journal of Critical Computer-Based Systems. Inderscience Publishers. 6:1. pp. 50–79. 2015. doi:10.1504/IJCCBS.2015.068852

Antoine Rauzy. Modes Automata and their Compilation into Fault Trees. Reliability Engineering and System Safety. Elsevier. 78:1. pp. 1–12. October, 2002. doi:10.1016/S0951-8320(02)00042-X.

Antoine Rauzy. An Experimental Study on Six Algorithms to Compute Transient Solutions of Large Markov Systems. Reliability Engineering and System Safety. Elsevier. 86:1. pp. 105–115. October, 2004. doi:10.1016/j.ress.2004.01.007.

Antoine Rauzy. Anatomy of an Efficient Fault Tree Assessment Engine. Proceedings of International Joint Conference PSAM'11/ESREL'12. R. Virolainen Ed. ISBN 978-162276436-5. Helsinki, Finland. June, 2012.

Antoine Rauzy. XFTA: an Open-PSA fault-tree engine. AltaRica Association. 2014.

Antoine Rauzy. Notes on Computational Uncertainties in Probabilistic Risk/Safety Assessment. Entropy. MDPI. 20:3. 2018. doi:10.3390/e20030162.

Jean-Pierre Signoret, Yves Dutuit, Jean-Pierre Cacheux, Cyrille Folleau, Stéphane Collas and Philippe Thomas. Make your Petri nets understandable: Reliability block diagrams driven Petri nets. Reliability Engineering and System Safety. Elsevier. 113. pp. 61–75. 2013. doi:10.1016/j.ress.2012.12.008.

Enrico Zio. The Monte Carlo Simulation Method for System Reliability and Risk Analysis. Springer London. London,