Safety Assessment of an Electrical System with AltaRica 3.0

Hala Mortada[1], Tatiana Prosvirnova[1], and Antoine Rauzy[2]

[1] Computer Science Lab, Ecole Polytechnique, Route de Saclay, Palaiseau, France
{Hala.Mortada,Prosvirnova}@lix.polytechnique.fr
[2] Chaire Blériot Fabre, LGI Ecole Centrale de Paris Grande voie des vignes, 92295 Châtenay-Malabry, France
Antoine.Rauzy@ecp.fr

Abstract. This article presents the high level, modeling language AltaRica 3.0 through the safety assessment of an electrical system. It shows how, starting from a purely structural model, several variants can be derived.Two of them target a compilation into Fault Trees and two others target a compilation into Markov chains. Experimental results are reported to show that each of these variants has its own interest. It also advocates that this approach made of successive derivation of variants is a solid ground to build a modeling methodology onto.

Keywords: AltaRica3.0, Complex systems, Reliability, Modeling, Safety.

1 Introduction

The increasing complexity of industrial systems calls for the development of sophisticated engineering tools. This is indeed true for all engineering disciplines, but especially for safety and reliability engineering. Experience shows that traditional modeling formalisms such as Fault Trees, Petri nets or Markov processes do not allow a smooth integration of risk analysis within the overall development process. These analysis require both considerable time and expertise. The specialization and the lack of model's structures make it difficult to share models amongst stakeholders, to maintain them throughout the life-cycle of the systems, and to reuse them from one project to another.

The AltaRica modeling language ([1],[2]) has been created at the end of the nineties to tackle these problems. AltaRica makes it possible to design high-level models with a structure that is very close to the functional or the physical architecture of the system under study. Its constructions allow models to be structured into a hierarchy of reusable components. It is also possible to associate graphical representations to these components in order to make models visually close to Process and Instrumentation Diagrams. The formal semantics of AltaRica allowed the development of a versatile set of processing tools such as compilers into Fault Trees ([2]), model-checkers ([3]) or stochastic simulators ([4]). A large number of successful industrial experiments with the language have been reported (see e.g. [5], [6], [7], [8] and [9]). Despite its quality, AltaRica faced

F. Ortmeier and A. Rauzy (Eds.): IMBSA 2014, LNCS 8822, pp. 181–194, 2014.

two issues of very different natures. First, systems with instant feedback's loops turned out to be hard to handle. Second, constructs of model's structuring were not fully satisfying.

AltaRica 3.0 [10] is a new version of the language that has been designed to tackle these two issues. Regarding model structuring, AltaRica 3.0 implements the prototype-oriented paradigm [11]. This paradigm fits well with the level of abstraction reliability and safety analysis stand at. Regarding mathematical foundations, AltaRica 3.0 is based on Guarded Transition Systems (GTS) [12]. GTS combine the advantages of state/event formalisms such as Petri nets and combinatorial formalisms such as block diagrams. This combination is necessary to model system patterns namely cold redundancies, cascading failures or remote interactions.

AltaRica 3.0 comes with a variety of assessment tools. In this article, we show how, starting from the same root model, different variants can be obtained by successive refinements: a first series targeting a compilation into Fault Trees and a second one targeting a compilation into Markov chains. Each of these variants capture a particular aspect of the system under study. We advocate that this approach made of successive derivation of variants is a solid ground to build a modeling methodology onto.

The remainder of this article is organized as follows. Section 2 presents the electrical system that is used as a red-wire example throughout the paper. Section 3 discusses how to describe the architecture of the system with a purely structural model. Section 4 proposes a first variant of this structural model which targets a compilation into Fault Trees. Section 5 presents a second variant which targets a compilation into Markov Chains. Finally, section 6 concludes this article.

2 Red Wire Example

Figure 1 shows a simple electrical system with cascade redundancies borrowed from [13] (we present it here with some additional complexity).

In a normal operating mode, the busbar BB is powered by the grid GR either through line 1 or through line 2. Each line is made of an upper circuit breaker CBUi, a transformer TRi and a lower circuit breaker CBDi. The two lines are in cold redundancy: Let's assume for instance that line 1 was working and that it failed either because one of the circuit breakers CBU1 or CBD1 failed, or because the transformer failed. In this case, the line 2 is attempted to start. This requires opening the circuit breaker CBD1 (if possible/necessary) and closing the circuit breakers of line 2. Since line 2 was out of service, the circuit breaker CBD2 was necessarily open.

If both lines fail, the diesel generator DG is expected to function, which requires closing the circuit breaker CB3. Circuit breakers may fail to open and to close on demand. The diesel generator may fail on demand as well. The grid GR may be lost either because of an internal failure or because of a short circuit in the transformer TRi followed by a failure to open the corresponding circuit breaker CBUi.

The two transformers are subject to a common cause failure.

There is a limited repair crew that can work on only two components at a time. After maintenance, the components are as good as new, but may be badly reconfigured.

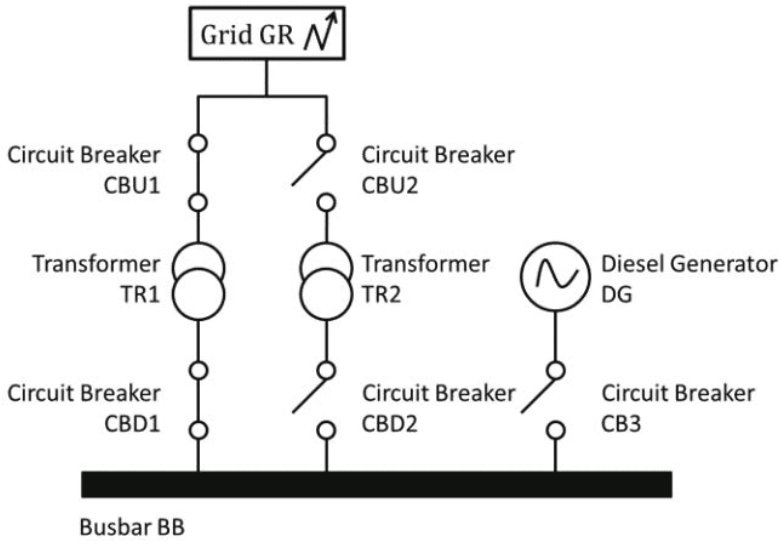

Fig. 1. A small electrical system

The problem is to estimate the reliability and the availability of this system. This example is small but concentrates on a number of modeling difficulties (warm redundancies, on demand failures, short-circuit propagation, common cause failures, limited resources), due to its multi-domains aspects.

3 Describing the Architecture of the System

The first step in analyzing a system consists of describing its functional and physical architecture. Figure 1 describes a possible decomposition of our electrical system. This decomposition deserves three important remarks.

First, it mixes functional and physical aspects. In fact, due to the small size of the example, only basic blocks (leaves of the decomposition) represent physical components. The others represent functions. We could consider functional and physical architectures separately. However, considering both in the same diagram simplifies things here. Moreover, it matches better with the usual way of designing models for safety and dependability analysis. Note also that at this step, we do not consider interactions between components.

Second, the underlying structure of this decomposition is not a tree, but a directed acyclic graph for the external power supply is shared between Line 1 and Line 2. As we shall see, this has very important consequences in terms of structuring constructs.

Third, the system embeds five circuit breakers and two transformers. We can assume that the circuit breakers on the one hand, the transformers on the other hand are all the same. From a modeling point of view, it means that we need to be able to define generic components and to instantiate them in different places in our model. On the contrary, components like "Primary Power Supply", "Backup Power Supply" and "Busbar Power Supply" are specific to that particular system.

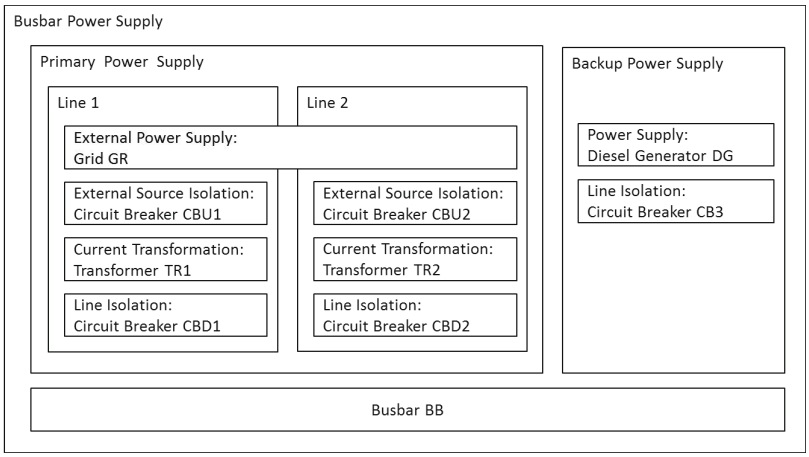

Fig. 2. Architecture of the Busbar Power Supply System

The structure of the AltaRica 3.0 model that reflects this architecture is sketched in Figure 2. In AltaRica 3.0, components are represented by means of blocks. Blocks contain variables, events, transitions, and everything necessary to describe their behavior. At this step, the behavioral part is still empty. Blocks can also contain other blocks and form hierarchies. The block "BusbarPowerSupply" contains two blocks: "PrimaryPowerSupply" and "BackupPowerSupply". "BusbarPowerSupply" is the parent block of "PrimaryPowerSupply" and an ancestor of "CBU1". Objects defined in a block are visible in all its ancestors. For instance, if the class "CircuitBreaker" defines an event "failToOpen", the instantiation of this event in "CBU1" is visible in the block "BusbarPowerSupply" through the dot notation, i.e. "PrimaryPowerSupply.Line1.CBU1.failToOpen".

An instance "GR" of the class "Grid" is declared in the block "PrimaryPowerSupply". It is convenient to be able to refer to it as "GR" as if it was declared in "Line1". This is the purpose of the "embeds" clause. This clause makes it clear that "GR" is part of "Line1", even if it is probably shared with some sibling blocks.

Classes in AltaRica 3.0 are similar to classes in object-oriented programming languages (see e.g. [14], [15] for conceptual presentations of the object-oriented paradigm). A class is a block that can be instantiated, i.e. copied, elsewhere in the model. There are several differences however between blocks and classes. AltaRica 3.0 makes a clear distinction between "on-the-shelf", stabilized knowledge, for which classes are used, from the model itself, i.e. the implicit main block

and all its descendants. Such a distinction has been conceptualized in C-K-theory ([16]). The implicit main block can be seen as the sandbox in which the analyst is designing his model. Declaring a class is in some sense creating another sandbox. Amongst other consequences, this means that it is neither possible to refer in a class to an object which is declared outside of the class, nor to declare a class inside another one or in a block. A class may of course contain blocks and instances of other classes up to the condition that this introduces no circular definition (recursive data types are not allowed in AltaRica 3.0). To summarize, AltaRica 3.0 borrows concepts to both object-oriented programming and prototype-oriented programming [11] - blocks can be seen as prototypes - so to provide the analyst with powerful structuring constructs that are well suited for the level of abstraction of safety analysis.

```
block BusbarPowerSupply
  block PrimaryPowerSupply
    Grid GR;
    block Line1
      embeds GR;
      CircuitBreaker CBU1, CBD1;
      Transformer TR1;
    end
    block Line2
      embeds GR;
      CircuitBreaker CBU2, CBD2;
      Transformer TR2;
    end
  end
  block BackupPowerSupply
    DieselGenerator DG;
    CircuitBreaker CB3;
  end
end
class Grid
end
...
```

Fig. 3. Structure of the AltaRica 3.0 Model for the Electrical System (partial view)

4 Targeting Compilation into Fault Trees

4.1 A Simple Block-Diagram like Model

We shall consider first a very simple model, close to a block diagram, in which basic blocks have a (Boolean) input, a (Boolean) output and an internal state (WORKING or FAILED). This basic block changes its state, from WORKING

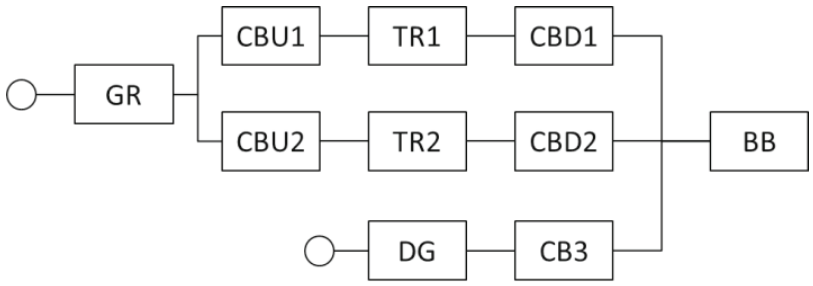

Fig. 4. A Block Diagram for the electric supply system

to FAILED, when the event "failure" occurs. The diagram for the whole system is pictured in Figure 4.

The AltaRica code for the diagram (with the architecture defined in the previous section) is sketched in Figure 5. The class "NonRepairableComponent" declares a state variable "s" and two Boolean flow variables: "inFlow" and "outFlow". "s" takes its value in the domain "ComponentState" and is initially set to WORKING. "inFlow" and "outFlow" are (at least conceptually) reset to false after each transition firing. Their default value is false. The class also declares the event "failure" which is associated with an exponential probability distribution of parameter "lambda". This parameter has the value "1.0e-4" unless stated otherwise.

After the declaration part, which consists in declaring flow and state variables, events and parameters, comes the behavioral part. This behavioral part itself includes transitions and assertions. In our example, there is only one transition and one assertion. The transition is labeled with the event "failure" and can be read as follows. The event "failure" can occur when the condition "s == WORKING" is satisfied. The firing of the transition gives the value FAILED to the variable "s".

The assertion describes the action to be performed to stabilize the system after each transition firing (and in the initial state). In our example, the variable "outFlow" takes the value true if "s" is equal to WORKING and "inFlow" is true, and false otherwise. In the initial state, all components are working, so the value true propagates from the input flow of the grid "GR" to the output flow of the system. If the circuit breaker "CBU2" fails, then the value false propagates from the output flow of "CBU2" to the output flow of the Line 2.

It would be possible to copy-paste the declaration of "NonRepairableComponent" in the declaration of the basic components of our model ("Grid", "CircuitBreaker", etc.). However, AltaRica 3.0 is an object-oriented language and thus provides a much more elegant way to obtain the same result: inheritance. It suffices to declare that the class "Grid" inherits from class "NonRepairableComponent". This is done in the code of Figure 5. In the class "Grid" the default value of the input flow is set to "true". This change makes the grid a source block. The remainder of the model consists in plugging inputs and outputs of the components in order to build the system. Note that the resulting model is

```
domain ComponentState { WORKING, FAILED }
class NonRepairableComponent
  Boolean s (init = WORKING);
  Boolean inFlow, outFlow (reset = false);
  event failure (delay = exponential(lambda));
  parameter Real lambda = 0.0001;
  transition
    failure: s == WORKING -> s := FAILED;
  assertion
    outFlow := s == WORKING and inFlow;
end
class Grid extends NonRepairableComponent(inFlow.reset = true);
end
...
block BusbarPowerSupply
  Boolean outFlow(reset = false);
  Grid GR;
  block PrimaryPowerSupply
    Boolean outFlow (reset = false);
    block Line1
      Boolean outFlow (reset = false);
      embeds GR;
      CircuitBreaker CBU1, CBD1;
      Transformer TR1;
      assertion
        CBU1.inFlow := GR.outFlow;
        ...
    end
    block Line2
      ... // similar to Line1
    end
    assertion
      outFlow := Line1.outFlow or Line2.outFlow;
  end
  ...
  assertion
    outFlow := PrimaryPowerSupply.outFlow or BackupPowerSupply.outFlow;
end
```

Fig. 5. A simple model targeting a compilation into Fault Trees (partial view)

not just a flat block diagram, but a hierarchical one. The compilation of this model into Fault Trees is performed according to the principle defined in [2]. The idea is to build a Fault Tree such that:

- The basic events of this Fault Tree are the events of the AltaRica model.
- There is (at least) an intermediate event for each pair (variable, value) of the AltaRica model.

– For each minimal cutset of the Fault Tree rooted by an intermediate event (variable, value), there exists at least one sequence of transitions in the AltaRica model labeled with events of the cutset that ends up in a state where this variable takes this value. Moreover, this sequence is minimal in the sense that no strict subset of the minimal cutsets can label a sequence of transitions ending up in a state where this variable takes this value.

For technical reasons, the Fault Trees generated by the AltaRica compiler are quite different from those an analyst would write by hand. The minimal cutsets are however the expected ones. For instance, the minimal cutsets for the target "(BusbarPowerSupply.outFlow, false)", i.e. the busbar is not powered, with our first model are as follows.

GR.failure DG.failure	GR.failure CB3.failure
CBU1.failure CBU2.failure DG.failure	CBU1.failure TR2.failure DG.failure
CBU1.failure CBU2.failure CB3.failure	TR1.failure CBD2.failure DG.failure
CBU1.failure CBD2.failure CB3.failure	TR1.failure CBU2.failure DG.failure
CBU1.failure CBD2.failure DG.failure	TR1.failure CBU2.failure CB3.failure
CBD1.failure CBU2.failure DG.failure	TR1.failure TR2.failure CB3.failure
CBU1.failure TR2.failure CB3.failure	TR1.failure CBD2.failure CB3.failure
CBD1.failure CBU2.failure CB3.failure	TR1.failure TR2.failure DG.failure
CBD1.failure CBD2.failure DG.failure	CBD1.failure TR2.failure CB3.failure
CBD1.failure CBD2.failure CB3.failure	CBD1.failure TR2.failure DG.failure

4.2 Taking into Account Common Cause Failures

We shall now design a second model in order to take into account the common cause failure of the two transformers (due for instance to fire propagation). To do so, we have to model that transformers fail simultaneously when the common cause failure occurs. AltaRica provides powerful synchronization mechanisms to make transitions simultaneous. The idea is to create an event "CCF" and a transition at the first common ancestor of the two transformers, i.e. "PrimaryPowerSupply". The new code for the "PrimaryPowerSupply" is sketched in Figure 6. The operator & synchronizes the transitions "failure" defined for each transformer. The operator & is associative and commutative. Any number of transitions can be thus synchronized. To fire the synchronizing transition, at least one of the synchronized transitions must be fireable. If the synchronizing transition is fired, then all the possible synchronized transitions are fired simultaneously. The modality ? indicates that the corresponding synchronized transition is not mandatory to fire the synchronizing transition. The modality ! indicates that the corresponding transition is mandatory.

Note that the synchronized transitions continue to exist independently of the synchronizing transition. It is possible to hide transitions by means of a special clause "hide". Our second model has the following two additional minimal cutsets.

PrimaryPowerSupply.CCF, BackupPowerSupply.DG.failure
PrimaryPowerSupply.CCF, BackupPowerSupply.CB3.failure

```
block PrimaryPowerSupply
  ...
  event CCF (delay = exponential(lambdaCCF));
  parameter Real lambdaCCF = 1.0e-5;
  ...
  transition
    CCF: ?Line1.TR1.failure & ?Line2.TR2.failure;
  assertion
  ...
end
```

Fig. 6. Synchronization mechanism to model the Common Cause Failures

5 Targeting Compilation into Markov Chains

In this section we consider repairs of components, reconfigurations and limited resources. First, we assume that there is an unlimited number of repairers. Then, we refine our model to take into account a limited number of repairers. Both models are compiled into Markov chains.

5.1 Unlimited Number of Repairers

All components are repairable. The AltaRica code in this case is similar to the one of the "NonRepairableComponent" (see Figure 5), except that a new event "repair", the corresponding parameter mu and the corresponding transition are added to the previous model. Instead of the "NonRepairableComponent", the classes "Transformer" and "Grid" of this model, will inherit from a "RepairableComponent".

The on demand failures of the circuit breakers and the diesel generator are also considered. The automata describing the behavior of the diesel generator, the transformer, the grid and the circuit breakers are figured in 7 and 8. The solid lines correspond to the stochastic transitions, whereas the dashes correspond to the immediate ones.

Figure 9 represents the AltaRica 3.0 model of the spare component corresponding to the left automaton depicted in Figure 7. Transitions "stop", "start" and "failureOnDemand" are immediate (their delays are equal to 0). When the state variable "s" is equal to STANDBY and the flow variable "demanded" is true, the event "start" may occur with the probability "1-gamma" and the event "failureOnDemand" may occur with the probability "gamma". The values of the probabilities are given through the attribute "expectation".

In this example, we also take into consideration the short circuit case (see the automaton in the right hand side of the Figure 8). For the transformer, the event failure is considered as a short circuit, that will propagate into the whole line and make it instantly fail. If the short circuit is in the "Grid", the whole "Primary Power Supply" system will eventually fail, inducing the spare block (the "Backup Power Supply" system) to take over. The structure of the whole

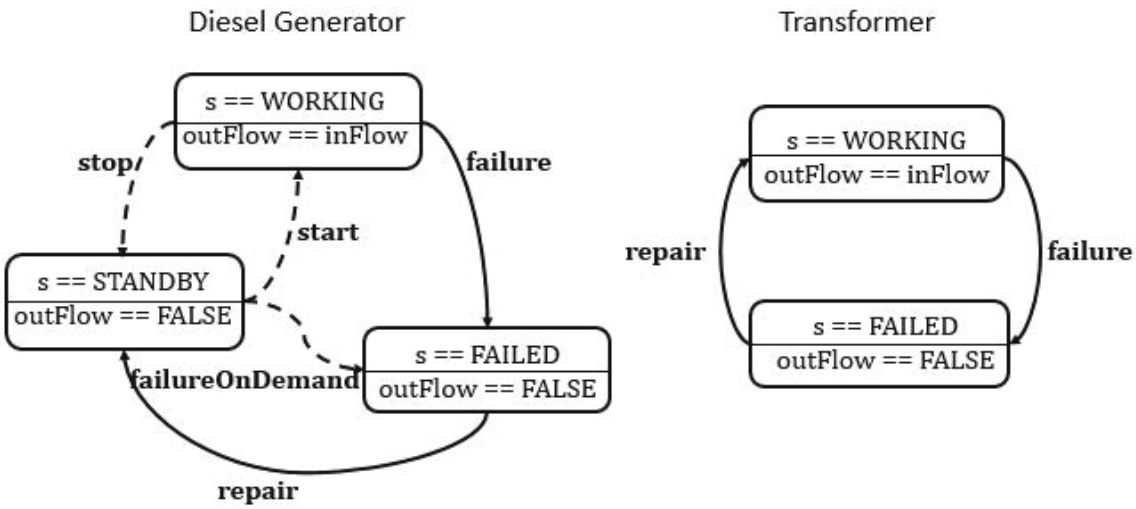

Fig. 7. Two automata describing the behavior of the Diesel Generator and the Transformer

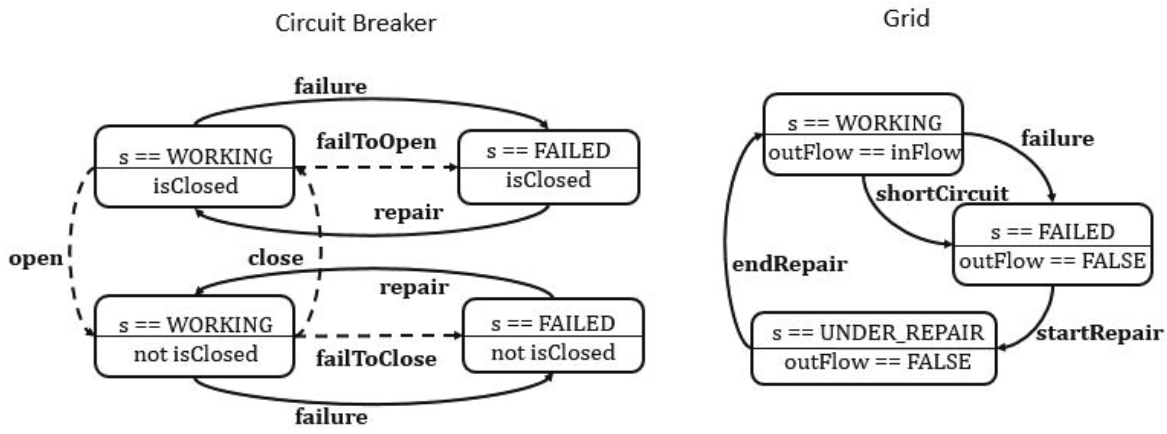

Fig. 8. Two automata describing the behavior of the Circuit Breaker and the Grid

model remains the same as in Figure 3. Some additional assertions are added in order to represent the propagation of the short circuit from the transformers to the grid and the reconfigurations (orders to open/close circuit breakers, to start/stop the diesel generator).

The semantics of AltaRica 3.0 are a Kripke structure (a reachability graph) with nodes defined by variable assignments (i.e. variables and their values) and edges defined by transitions and labeled by events. If the delays associated to the events are exponentially distributed, then the reachability graph can be interpreted as a continuous time Markov chain. In the case when the graph contains immediate transitions, they are just collapsed using the fact that an exponential delay with rate λ followed by an immediate transition of probability p is equivalent to a transition with an exponential delay of rate $p\lambda$.

```
domain SpareComponentState { STANDBY, WORKING, FAILED }
class SpareComponent
  Boolean s (init = WORKING);
  Boolean demanded, inFlow, outFlow (reset = false);
  event failure (delay = exponential(lambda));
  event repair (delay = exponential(mu));
  event start (delay = 0, expectation = 1 - gamma);
  event failureOnDemand (delay = 0, expectation = gamma);
  event stop(delay = 0);
  parameter Real lambda = 0.0001;
  parameter Real mu = 0.1;
  parameter Real gamma = 0.001;
  transition
    failure: s == WORKING -> s := FAILED;
    repair: s == FAILED -> s := STANDBY;
    start: s == STANDBY and demanded -> s := WORKING;
    failureOnDemand: s == STANDBY and demanded -> s := FAILED;
    stop: s == WORKING and not demanded -> s := STANDBY;
  assertion
    outFlow := s == WORKING and inFlow;
end
```

Fig. 9. AltaRica 3.0 model of a spare component (Diesel generator)

The generated Markov Chain contains 7270 states and 24679 transitions. The tool XMRK calculates the unavailability for different mission times. For $\lambda = 10^{-4}$, $\gamma = 10^{-3}$ and $\mu = 10^{-1}$, the probabilities are represented in Figure 12.

5.2 Limited Number of Repairers

In this part, we consider the case of a limited number of repairers, namely lower than the number of failures. Counter to the previous model, in order for a repair to take place, the repairer should be available and not used by another component. In this case, some changes in the behavior of the system take place. We will not only be interested in the "repair" transition, but also in the time it starts and ends at. Therefore, the "repair" transition is replaced by a whole set of transitions: startRepair and endRepair (see for example the automaton in the right hand side of the Figure 8). Besides, a new class called "RepairCrew" that defines when a job can start is added to the previous model (see Figure 10).

The transitions "startRepair" and "startJob", as well as "endRepair" and "endJob" are synchronized using the operator & as shown in Figure 11.

Compared to the definition of the common cause failure (see Figure 6), here the modality ! is used in the synchronization, which means that both synchronized events should be fireable to be able to fire the synchronizing transitions. In this example, the synchronized events are hidden explicitly using the clause "hide".

```
class RepairCrew
  Integer numberOfBusyRep (init = 0);
  parameter Integer totalNumberOfRepairers = 1;
  event startJob, endJob;
transition
  startJob: numberOfBusyRep < totalNumberOfRep ->
       numberOfBusyRep := numberOfBusyRep + 1;
  endJob: numberOfBusyRep > 0 ->
       numberOfBusyRep := numberOfBusyRep - 1;
end
```

Fig. 10. AltaRica model of the Repair Crew

```
block BusbarPowerSupply
  RepairCrew R;
  block PrimaryPowerSupply
    ...
  end
  block BackupPowerSupplySystem
    ...
  end
  event PPS_GR_startRepair, PPS_GR_endRepair;
  ...
transition
  PPS_GR_startRepair: !R.startJob & !PrimaryPowerSupply.GR.startRepair;
  PPS_GR_endRepair: !R.endJob & !PrimaryPowerSupply.GR.endRepair;
  hide R.startJob, PrimaryPowerSupply.GR.startRepair;
  hide R.endJob, PrimaryPowerSupply.GR.endRepair;
  ...
end
```

Fig. 11. A model targeting a compilation into Markov Chains (partial view)

In order to make the results more interesting, two numbers of repairers $n = 1$ and $n = 3$ are considered. This will allow us to compare the two graphs of unavailability. The same parameters mentioned in the first subsection are used here as well. The Markov Chain consists of 29332 states and 98010 transitions. The graph in Figure 12 shows indeed that the unavailability is lower when the number of repairers is bigger, and even lower when it is unlimited.

6 Conclusion

In this paper we showed, using an electrical system as a red-wire example, how AltaRica 3.0 can be used to model complex phenomena. A purely structural model was designed. Then, we derived four variants from it: two of them targeting a compilation into Fault Trees and two others targeting a compilation into

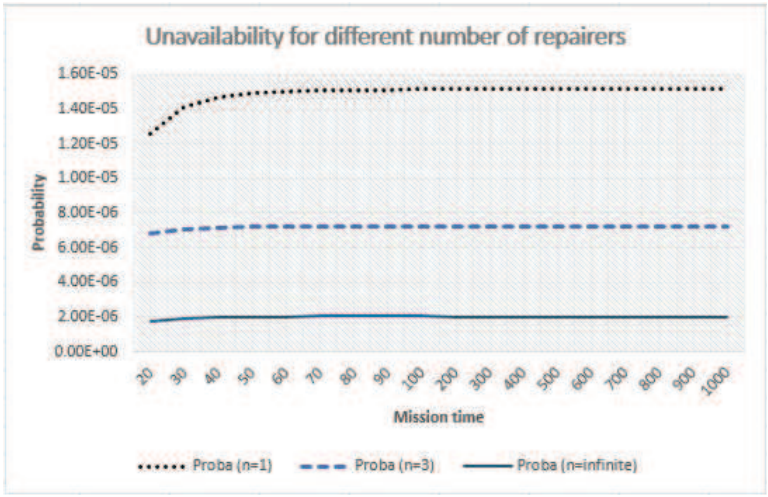

Fig. 12. Unavailability for a different number of repairers

Markov chains. Each variant, or subset of variants, was tailored for a particular assessment tool, i.e. to capture a particular aspect of the system under study. Based on this experience (and several others we have performed), we are convinced that this approach, consisting of deriving models by means of successive refinements, is a solid ground to build a modeling methodology. The calculations to be performed are actually very resource consuming. Therefore, a model is always a trade-off between the accuracy of the description and the ability to perform calculations. Refining a model in successive variants is a good way to seek a good trade-off. Moreover, the trade-off depends on the characteristics of the system to be observed. Therefore, different tools must be applied. As a consequence, the refinement process should not be linear, but rather have a tree-like structure.

References

1. Arnold, A., Griffault, A., Point, G., Rauzy, A.: The altarica formalism for describing concurrent systems. Fundamenta Informaticae 34, 109–124 (2000)
2. Rauzy, A.: Modes automata and their compilation into fault trees. Reliability Engneering and System Safety (2002)
3. Griffault, A., Vincent, A.: The mec 5 model-checker. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 488–491. Springer, Heidelberg (2004)
4. Khuu, M.: Contribution à l'accélération de la simulation stochastique sur des modèles AltaRica Data Flow. PhD thesis, Université de la Mèditerranée, Aix-Marseille II (2008)
5. Humbert, S., Seguin, C., Castel, C., Bosc, J.-M.: Deriving safety software requirements from an altarica system model. In: Harrison, M.D., Sujan, M.-A. (eds.) SAFECOMP 2008. LNCS, vol. 5219, pp. 320–331. Springer, Heidelberg (2008)
6. Quayzin, X., Arbaretier, E.: Performance modeling of a surveillance mission. In: Proceedings of the Annual Reliability and Maintainability Symposium, RAMS 2009, Fort Worth, Texas USA, pp. 206–211 (2009) ISBN 978-1-4244-2508-2

7. Sghairi, M., De-Bonneval, A., Crouzet, Y., Aubert, J.J., Brot, P., Laarouchi, Y.: Distributed and reconfigurable architecture for flight control system. In: Proceedings of 28th Digital Avionics Systems Conference (DASC 2009), Orlando, USA (2009)
8. Chaudemar, J.C., Bensana, E., Castel, C., Seguin, C.: Altarica and event-b models for operational safety analysis: Unmanned aerial vehicle case study. In: Proceedings Formal Methods and Tools, FMT 2009, London, England (2009)
9. Adeline, R., Cardoso, J., Darfeuil, P., Humbert, S., Seguin, C.: Toward a methodology for the altarica modelling of multi-physical systems. In: Proceedings of European Safety and Reliability Conference, ESREL 2010, Rhodes, Greece (2010)
10. Prosvirnova, T., Batteux, M., Brameret, P.A., Cherfi, A., Friedlhuber, T., Roussel, J.M., Rauzy, A.: The altarica 3.0 project for model-based safety assessment. In: Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS 2013, pp. 127–132. International Federation of Automatic Control, York (2013) ISBN: 978-3-902823-49-6, ISSN: 1474-6670
11. Noble, J., Taivalsaari, A., Moore, I.: Prototype-Based Programming: Concepts, Languages and Applications. Springer, Heidelberg (1999) ISBN-10: 9814021253. ISBN-13: 978-9814021258
12. Rauzy, A.: Guarded transition systems: A new states/events formalism for reliability studies. Journal of Risk and Reliability 222, 495–505 (2008)
13. Bouissou, M., Bon, J.L.: A new formalism that combines advantages of fault-trees and markov models: Boolean logic-driven markov processes. Reliability Engineering and System Safety 82, 149–163 (2003)
14. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988) ISBN-10: 0136290493. ISBN-13: 978-0136290490
15. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer-Verlag. New York Inc. (1998) ISBN-10: 0387947752. ISBN-13: 978-0387947754
16. Hatchuel, A., Weil, B.: C-k design theory: An advanced formulation. research in engineering design. Research in Engineering Design 19, 181–192 (2009)