Reliability Engineering and System Safety 167 (2017) 561–571

Contents lists available at ScienceDirect

Reliability Engineering and System Safety

journal homepage: www.elsevier.com/locate/ress

Production trees: A new modeling methodology for production availability analyses

Leïla Kloul [a], Antoine Rauzy [b,*]

[a] DAVID, Université de Versailles St-Quentin-en-Yvelines, Versailles, France
[b] Department of Production and Quality Engineering, Norwegian University of Science and Technology, Trondheim, Norway

ARTICLE INFO

Keywords:
Production availability
AltaRica
Modeling patterns

ABSTRACT

In this article, we propose a new modeling methodology for production availability analyses. These analyses are typically carried out by means of flow network models and Monte-Carlo simulations. The design of flow network models is often delicate because the production of a unit may depend on the states of other units located downstream and upstream in the production line.

We show here how to handle this problem by means of operators working on three flows: a capacity flow moving forward from source to target units, a demand flow moving backward from target units to source units, and finally a production flow moving forward from source to target units. The production depends on the demand which itself depends on the capacity. Models designed according to this scheme caneventually be seen either as flow networks or as an extension of (Dynamic) Fault Trees to production availability analyses. We present the AltaRica 3.0 library of modeling patterns we designed to represent the different operators. We report results of experiments we performed on models designed using this library.

© 2017 Published by Elsevier Ltd.

1. Introduction

The production availability of a system is defined, according to ISO 20,815 standard [1], as the ratio of the actual production of this system to its planned production, or any other reference level, over a specified period of time. Production assurance or production regularity has similar meaning [2–5]. Production availability analyses are typically carried out in practice by means of flow network models and Monte-Carlo simulations (see e.g. [6–8]). One of the main difficulties of production availability modeling is that the actual production level of a unit may depend not only on its internal state, but also on the production levels (and internal states) of units located downstream and upstream in the production line. Such a situation induces circularities because the production depends on the demand and vice-versa. In practice, this problem is solved either by calculating production levels outside the model, which is obviously not a very sustainable solution, or by using ad-hoc modeling gadgets, making models difficult to design, to understand and to maintain.

In this article, we propose a new modeling methodology to address this issue. Namely, we introduce operators working on three flows: a capacity flow moving forward from source to target units, a demand flow moving backward from target units to source units, and finally a production flow moving forward from source to target units. The production depends on the demand which itself depends on the capacity. Models designed according to this scheme can eventually be seen either as flow networks or as an extension of (Dynamic) Fault Trees to production availability analyses. For this reason, we call them *Production Trees*. They are made of two types of components: first, basic components, i.e. leaves of the hierarchical model, which represent production units; second, gates that compose behaviors and are used to represent behaviors of (sub-)systems. The numerical capacity/demand/production flow scheme generalizes the Boolean availability/demand scheme introduced in reference [9] to provide a sound semantics to both Dynamic Fault Trees [10] and Boolean Driven Markov Processes [11].

To support and test the above idea, we designed a library of AltaRica 3.0 modeling patterns to represent basic components and gates. The high level modeling language AltaRica 3.0 (see e.g. [12–14]) provides the two basic features required to implement production trees: first, constructs to represent states of components and changes of states under the occurrence of stochastic events such as failures and repairs as well as deterministic events such as reconfigurations; Second, constructs to represent flow propagation. The AltaRica classes of the *Production Trees* library can be instantiated directly and adjusted for specific needs. This is the reason why we prefer to call them modeling patterns (in the sense of design patterns [15]) rather than modeling components (see e.g. [14,16] for discussions about modeling patterns in AltaRica).

* Corresponding author.
E-mail addresses: Leila.Kloul@uvsq.fr (L. Kloul), Antoine.Rauzy@ntnu.no, antoine.rauzy@ntnu.no (A. Rauzy).

http://dx.doi.org/10.1016/j.ress.2017.06.017
Received 29 March 2016; Received in revised form 24 April 2017; Accepted 12 June 2017
Available online 21 June 2017
0951-8320/© 2017 Published by Elsevier Ltd.

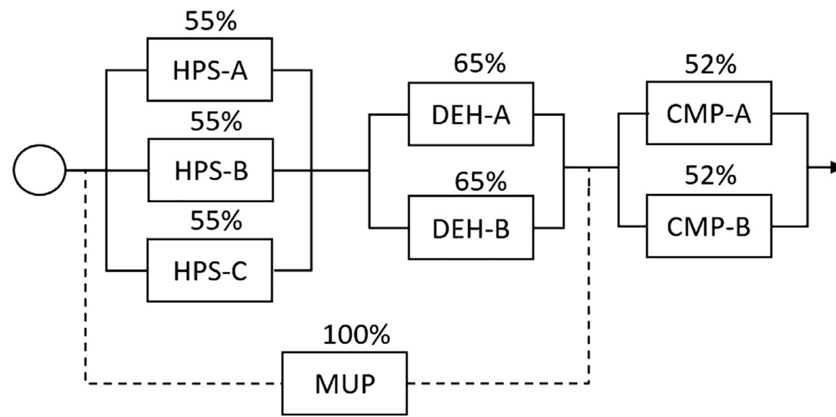

Fig. 1. Simplified production plant.

Table 1
Reliability data for the case study.

Units	Capacity (%)	Failure rate (per hour)	Repair rate (per hour)
HPS	55	8.91e-5	2.54e-3
DEH	65	3.11e-5	3.95e-3
CMP	52	3.50e-5	5.14e-3
MPU	100	1.00e-6	1.00e-3

The contribution of this paper is therefore twofold: first, it provides a new methodological framework for production availability analyses. Second, it presents an AltaRica library of modeling patterns dedicated to production availability analyses and discusses its use on a case study.

The remainder of this article is organized as follows. Section 2 presents the case study we shall use throughout the article to illustrate the design of production trees. Section 3 recalls basics about guarded transition systems and AltaRica. Section 4 is the core of the article: it introduces base modeling patterns of production trees and gives their implementation in AltaRica. Section 5 proposes advanced modeling patterns. Section 6 presents experimental results. Finally, Section 7 concludes the article.

2. Case study

As an illustrative example for our presentation of production trees, we shall use a simple case study taken from Kawauchi and Rausand's article [4]. This test case is much too simple to be realistic, but its purpose is primarily to support the discussion.

The system is a production facility consisting of eight units (see Fig. 1). Gas separated from the well fluid at upstream side is fed to the facility, treated through separators ($HPS - A, B, C$) and dehydrators ($DEH - A, B$), and led to compressors ($CMP - A, B$). The Make-Up Compressor (MUP) is a spare unit installed to enable both compressors CMP-A and CMP-B to discharge gas with full flow rate even if some of gas treatment units (HPS or DEH) are failed. The MUP is assumed to be equipped with its own gas treatment units. The maximum production capacity of units is provided in Fig. 1. As only two out of the three separators are necessary to reach a full production The separator HPS-C is in cold redundancy.

Separators, dehydrators and compressors can fail and be repaired independently with known failure and repair rates. The make-up compressor has a probability of failure (for the period of time it used) obtained from operation log. Reliability data for the different units are given in Table 1.

Fig. 2 shows the functional breakdown of the simplified plant. We shall use this decomposition to design our production availability model. Each leaf of the tree is represented by a basic component and each internal node by a gate. With that respect, the methodology to design production trees is similar as the one for fault trees.

Although this case study is easy to modularize, we shall consider it as if it was not. We could have make the problem more complex by introducing dependencies between treatment units (e.g. common cause failures, limited capacity of the repair crew, periodic inspections...). However, we preferred to keep the use case in its original form for the sake of the clarity of the presentation. The forthcoming developments do not make any assumption about the independence of components and subsystems.

3. AltaRica

In this section, we give an overview of Guarded Transition Systems and the AltaRica modeling language upon which we built the modeling methodology we propose. For more details, we refer the reader to [12,14,17].

3.1. Guarded transition systems

Formally, a Guarded Transition System (GTS) is a quintuple $\langle V, E, T, A, \iota\rangle$ where V is a finite set of variables, E is a finite set of events, T is a finite set of transitions, A is an assertion and ι is the initial assignment of variables.

- Each variable $v \in V$ takes its value into a domain denoted by $domain(v)$. Variables can be Boolean, integers, floating point numbers, members of finite sets of symbolic constants or anything convenient for the modeling purpose.
- A variable assignment is a function from V to $\prod_{v\in V} domain(v)$. A variable update is a function from $\prod_{v\in V} domain(v)$ into itself, that is a function that transforms a variable assignment into another one.
- Each event $e \in E$ is associated with a deterministic or stochastic delay, as well as with a weight.
- Each transition $t \in T$ is a triple $\langle e, g, a\rangle$ where e is an event in E, g is a Boolean condition over the variables in V and a is an instruction over the variables in V, that is a variable update. a is called the action of the transition. We shall denote the transition $\langle e, g, a\rangle$ by $g \xrightarrow{e} a$.
- The assertion A is an instruction over the variables in V.

Variables in V are separated into two groups: *states variables* whose values are modified only in the actions of transitions and *flow variables* whose values are modified only by the assertion. The values of flow variables are fully determined by the values of states variables. The calculation of values of flow variables is achieved by means of a fixpoint mechanism.

Let σ be a variable assignment and let $t : g \xrightarrow{e} a$ be a transition which is fireable in σ, i.e. such that $g(\sigma) = 1$. Firing t updates σ into the assign-

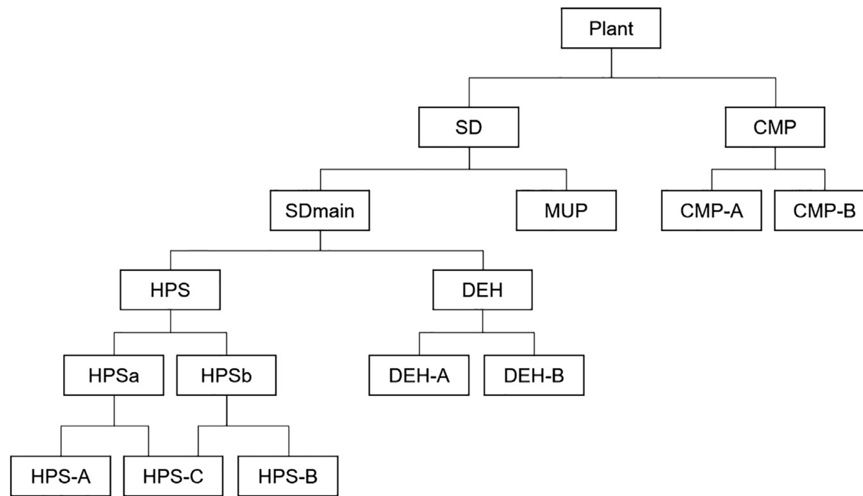

Fig. 2. A functional breakdown of the simplified production plant.

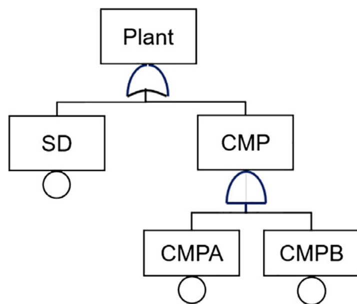

Fig. 3. A partial view of the production facility fault tree.

ment $\rho = A(a(\sigma))$, that is applying to σ, first the update a (the action of the transition) and then the update A (the global assertion).

The semantics of a GTS is formally defined as the set of its possible runs or executions. Each execution is a finite or infinite sequence of pairs (d_0, σ_0), ...,(d_i, σ_i), ...where the d_i's are dates such that $0 = d_0 \leq d_1 \leq d_2 \ldots$, and the σ_i's are states such that $\sigma_0 = \iota$ and $\sigma_{i+1} = \rho = A(a(\sigma_i))$. Each pair is obtained from the previous one by firing a (fireable) transition. Transitions are scheduled according to the delays associated with their event. Additional rules are applied, using weights, to select the transition to be fired when two or more transitions are scheduled at the same date.

The semantics of GTS is actually similar to the one of Generalized Stochastic Petri Nets [18]. GTS is however much more powerful: transitions do not need to be Markovian, states, pre- and post-conditions of transitions are also more general than those of Petri nets (and even Petri nets with assertions and predicates [19]). The fixpoint semantics of GTS makes also possible to handle looped systems while keeping an excellent algorithmic efficiency [20].

As an illustration, consider the small fault tree pictured in Fig. 3. This is a partial view of the fault tree modeling the production facility in Fig. 1.

A GTS that encodes this fault tree, written in AltaRica 3.0, is given Fig. 4. It involves: three Boolean state variables `SD`, `CMPA` and `CMPB` (introduced by the keyword `init`) to represent the states of the basic components; two Boolean flow variables `CMP` and `Plant` (introduced by the keyword `reset`) to represent flows going out of the corresponding gates; three events `failureSD`, `failureCMPA` and `failureCMPB` to represent the failures of the basic components. The delays of these events are exponentially distributed. The corresponding transitions turn state variables from **false** to **true**. Finally, the assertion is made of two equations defining each of the flow variables.

All the executions of this GTS are indeed finite because the components are not repairable.

3.2. Structural constructs

Guarded Transitions Systems provide a very powerful mathematical framework but can hardly be used directly to describe complex systems. Their flat structure makes models very hard to design and even harder to maintain. AltaRica provides thus a whole set of constructs to structure models. Models are built typically by instantiating and adjusting components described in libraries of reusable components and patterns.

AltaRica 3.0 implements the so-called prototype-oriented paradigm to describe structures [21]. It is actually the combination of GTS with S2ML, a domain specific language dedicated to the description of structures [14,22,23].

For the purpose of the present study, the idea is to design a library of modeling patterns to represent basic events and gates. Fig. 5 provides an example of a possible AltaRica code for basic non repairable components. Such a component has a given capacity set by default to 100, no matter the unit. It produces at its capacity if it is working (not failed) and 0 otherwise. Reusable modeling components are described as classes using keyword `class`.

We can similarly define classes for series (and) gates and parallel (or) gates (see Section 4). Then, basic components and gates are assembled (and connected) as illustrated in Fig. 6. The parameters of components can be changed while instantiating them. The hierarchical model of Fig. 6 is automatically flattened into a GTS by the AltaRica compiler prior applying assessment tools (such as the stochastic simulator). This operation is nearly linear in the size of the model.

3.3. Graphical representations

Both guarded transition systems and hierarchical models can be graphically represented. For instance, Fig. 7 and Fig. 2 are graphical representations of the guarded transition system in Fig. 5 and the hierarchical model of the whole plant, respectively. These graphical representations call for several remarks of general scope and interest.

First, graphical representations are of a great help as a communication means between stakeholders. We use them as much as possible to discuss about AltaRica models.

Second, it is worth standardizing graphical representations in order to avoid as much as possible ambiguities. Graphical notations such as SysML [24] are nothing but a standardization of a number of graphics. The graphical representations presented in Fig. 7 and Fig. 2 are in this sense standard for AltaRica.

```
block FaultTree
  Boolean SD, CMPA, CMPB (init = false);
  Boolean CMP, Plant (reset = false);
  event failureSD (delay = exponential(1.0e-9));
  event failureCMPA, failureCMPB (delay = exponential(3.5e-5));
  transition
    failureSD: not SD -> SD := true;
    failureCMPA not CMPA -> CMPA := true;
    failureCMPB: not CMPB -> CMPB := true;
  assertion
    CMP := CMPA and CMPB;
    Plant := SD or CMP;
end
```

Fig. 4. A GTS for the fault tree in Fig. 3.

```
class NonRepairableComponent
  Boolean failed (init = false);
  Integer outProduction (reset = 0);
  event failure (delay = exponential(lambda));
  parameter Real lambda = 1.0e-3;
  parameter Integer capacity = 100;
  transition
    failure: not failed -> failed := true;
  assertion
    outProduction := if failed then 0 else capacity;
end
```

Fig. 5. A class for basic non-repairable components.

Third, graphics are a communication means, and as a such, they should be kept simple. As soon as models get complex enough, and they quickly get, no graphics can fully represent them. Models of complex systems are necessarily complex. Therefore, graphics cannot contain all the information. Moreover, they are useful because they do not contain all the information. In a word, it is of primary methodological importance not to confuse the model and its graphical representation(s).

Fourth, as sizes and positions of graphical elements are not relevant, but from an aesthetic point of view, it is better, when possible, to generate graphical representation automatically from the model. Unfortunately, no universal algorithm exists to size and position graphical elements aesthetically even for very simple graphical notations such as block diagrams. Graphical representations have thus to be persistent which is a source of incoherence between the model and its representation(s).

Fifth, hierarchy of components can be represented in (at least) two ways: by means of nested boxes, like in (hierarchical) flow diagrams, or by means of tree like representations as illustrated Fig. 2. Eventually, these two types of representations are equivalent (like Reliability Block Diagrams and Fault Trees). The choice of one type rather than the other depends on which aspect one wants to emphasize. It is to a large extent a matter of taste. When graphical representations can be automatically generated from the model, it is possible to switch at will from one to the other. This another reason not to confuse the model and its graphical representation(s).

4. Production trees

4.1. Flows in the hierarchical decomposition

The production tree methodology assumes a hierarchical breakdown of the system such as the one pictured Fig. 2. Three types of flows are circulating between the components of this hierarchical breakdown: a capacity flow moving forward from source to target units, a demand flow moving backward from target units to source units, and finally a production flow moving forward from source to target units. The production depends on the demand which itself depends on the capacity. Fig. 8 shows a hypothetical component with m parents and n children.

```
block System
  NonRepairableComponent SD(lambda=1.0e-9);
  NonRepairableComponent CMPA(lambda=3.5e-5, capacity=52);
  NonRepairableComponent CMPB(lambda=3.5e-5, capacity=52);
  SeriesGate Plant;
  ParallelGate CMP;
  assertion
    Plant.inProduction1 := SD.outProduction;
    Plant.inProduction2 := CMP.outProduction;
    CMP.inProduction1 := CMPA.outProduction;
    CMP.inProduction2 := CMPB.outProduction;
end
```

Fig. 6. A hierarchical model for a fault tree.

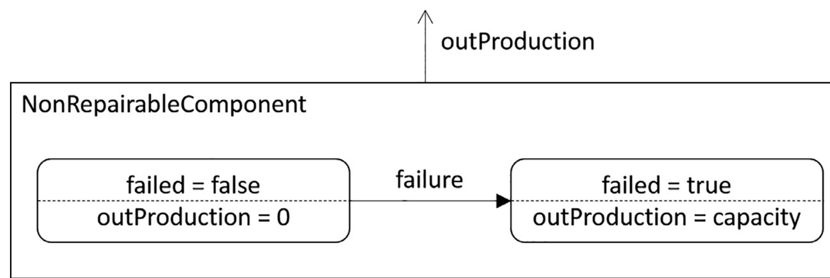

Fig. 7. Graphical representation of the GTS in Fig. 5.

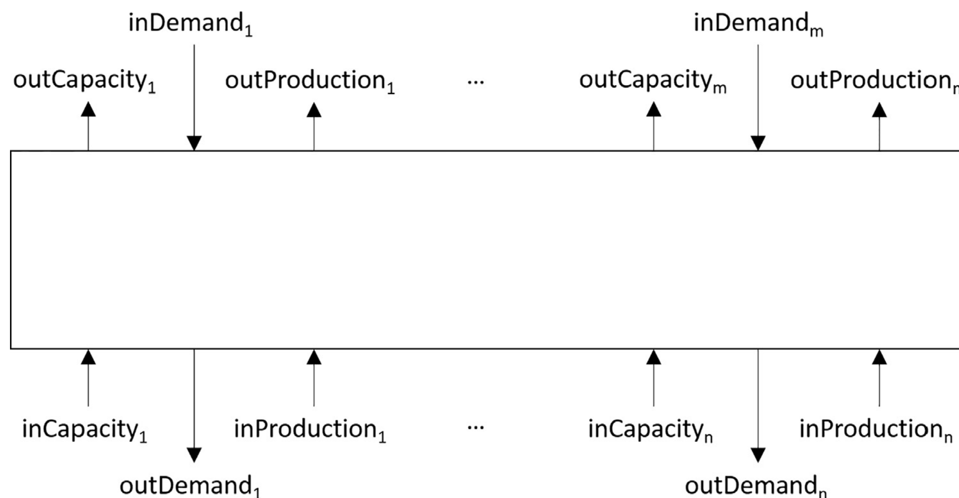

Fig. 8. Flows circulating in and out a component.

Models designed according to this scheme are made of two types of components: basic components and gates. The formers, which represent production units, are the leaves of the hierarchical model. The latters' compose behaviors and are used to represent behaviors of (sub-)systems.

4.2. Modeling patterns

In the following, we present the modeling patterns of the *Production Trees* library. We prefer the term pattern, in the sense of design patterns [15], to the term reusable components because modeling elements presented here have sometime to be adjusted to the specifical needs of the system under study.

4.2.1. Basic components

Basic components represent production/treatment units. They play a similar role as basic event in fault trees. There are several typical patterns for basic components. We shall see in this section the base one. More advanced patterns will be presented in the next section.

The base pattern represents a repairable component that can be either active or in standby. Using the hierarchical decomposition shown in Fig. 8, this pattern involves a Boolean state variable `failed` and three flow variables: `outCapacity`, `inDemand` and `outProduction`. First, the component exports its actual capacity, which is null if it is failed and its potential capacity otherwise. Second, the component receives a demand which, in stabilized situations, should not exceed the component capacity (there are intermediate situations however where this can happen). Third, the component exports a production, which is clearly the minimum of its actual capacity and the demand. The component is considered in standby if the demand is null. The base pattern presented here assumes that the component cannot fail when the component is in standby. It is however easy to extend the pattern so to represent hot and warm redundancies.

This pattern can be used directly for all production units (HPS-x, DEH-y, CMP-z). It has to be adjusted for the MUP because it cannot be repaired and has a failure probability rather than a failure rate.

The AltaRica code of this pattern is given in Fig. 9.

4.2.2. Combinatorial gates

The second series of patterns involves two combinatorial gates (*min* and *plus*) and a *splitter* gate. The latter is a modeling element to split the production. All these gates involve no event, and therefore no transitions, only flow variables and assertions.

The *min*-gate, denoted as $\odot$, extends the *and*-gate of fault trees. It has one parent and any number of children. Its output capacity is minimum of the output capacities of the children and of its intrinsic capacity. The input demand of the gate (coming from its parent) is propagated unchanged to its children. Finally, the output production of the gate is the minimum of the output production of its children. In stabilized situations, the demand is always smaller than the capacity. Therefore the production equals the demand. The *min*-gate pattern can be used for the nodes Plant and SDmain of the functional breakdown presented Fig. 2.

The *plus*-gate, denoted as $\oplus$, extends the *or*-gate. This extension is slightly more difficult than the previous one. The output capacity of the gate is the minimum of its intrinsic capacity and the sum of the output capacities of its children. Similarly to the *min*-gate, the input demand of the gate (coming from its parent) is propagated unchanged to its children. Finally, the output production of the gate is the sum of the output productions of its children. Except in the case where the output capacity of the gate matches the sum of the output capacities of its children, an allocation strategy is required to allocate the input demand to the children. For instance, the demand can be allocated according to a pro-rata of their capacities. Another choice is to allocate the maximum production to the first child, then the maximum of the remainder to the second child and so on (priority policy). Other allocations policies are

```
class RepairableComponent
  Boolean failed (init = false);
  Integer outCapacity (reset = 0);
  Integer inDemand (reset = 0);
  Integer outProduction (reset = 0);
  event failure (delay = exponential(lambda));
  event repair (delay = exponential(mu));
  parameter Real lambda = 1.0e-3;
  parameter Real mu = 1.0e-1;
  parameter Integer capacity = 100;
  transition
    failure: not failed and (inDemand>=0) -> failed := true;
    repair: failed -> failed := false;
  assertion
    outCapacity := if failed then 0 else capacity;
    outProduction := min(outCapacity, inDemand);
end
```

Fig. 9. A base pattern for repairable components.

```
class ProRataPlusGate
  Integer outCapacity (reset = 0);
  Integer inDemand (reset = 0);
  Integer outProduction (reset = 0);
  Integer inCapacity1, inCapacity2 (reset = 0);
  Integer outDemand1, outDemand2 (reset = 0);
  Integer inProduction1, inProduction2 (reset = 0);
  parameter Integer capacity = 100;
  assertion
    outCapacity := min(capacity,inCapacity1+inCapacity2);
    outDemand1 := if (inCapacity1+inCapacity2==0)
                  then 0
                  else inDemand * inCapacity1/(inCapacity1+inCapacity2);
    outDemand2 := min(capacityIn2, inDemand - outDemand1);
    outProduction := inProduction1+inProduction2;
end
```

Fig. 10. The pattern for pro-rata *plus*-gate.

of course possible. As previously, in stabilized situations, the demand is always smaller than the capacity. Therefore the production equals the demand. Fig. 10 shows the code for a binary *plus*-gate implementing the pro-rata allocation policy. Nodes HPS, DEH and CMP of the functional breakdown in Fig. 2 can be implemented according to this pattern.

The composition of a main component (or subsystem) and a spare component (or subsystem) can be represented to some extent with priority *plus*-gates. In this scheme, the main component produces at the maximum of its actual capacity. The remainder of the demand, if any, is allocated to the spare component. This pattern generalizes triggers of BDMP [11] or more exactly the scheme proposed in [9] to give them a sound semantics. It can be used to model node SD of the functional breakdown of Fig. 2 as we can assume that the MUP is used to achieve a 100% production in case of failure of the HPS-x and the DEH-y units.

This approach may however introduce a bias because it considers that the capacity of the whole system (main component plus spare component) is the sum of their individual capacities. To solve this problem the solution consists in defining the intrinsic capacity of the gate as the capacity of the main component when it is fully operational. In that way, the input demand for the system never exceeds (in stabilized states) the capacity of the system.

Note that *min*- and *plus*-gates strongly recall Simon's tropical algebra (also called min-plus algebra) [25]. Notations $\odot$ and $\oplus$ are inspired from this work. There is an important difference however, because in production trees a basic unit or a gate can have more than one parent.

Consider nodes HPSa and HPSb of the functional breakdown in Fig. 2. The above pattern cannot be used, at least alone, to implement these nodes, because the HPS-C unit is a cold spare for both HPS-A and HPS-B. I.e. it can be substituted for either of HPS-A and HPS-B, but not for both at the same time. The same production cannot be counted twice. This problem does not arise in fault trees because of the idempotence of Boolean connectives ($f \vee f = f \wedge f = f$ for all f). For production trees, we have to introduce modeling elements to split the production of a child between its parents. For this reason we call them splitters.

The situation is in some sense dual to the *plus*-gate. A *splitter*-gate, denoted as $\oslash$, has one child and several parents. The input capacity of the splitter is transmitted unchanged to its parents. The output demand of the splitter is the sum of its input demands. The input production of the splitter is split among its parents according to some policy that can be a pro-rata policy, a priority policy or any other suitable policy. We can use the priority splitter for the case study: node HPS-C is substituted in priority to HPS-A.

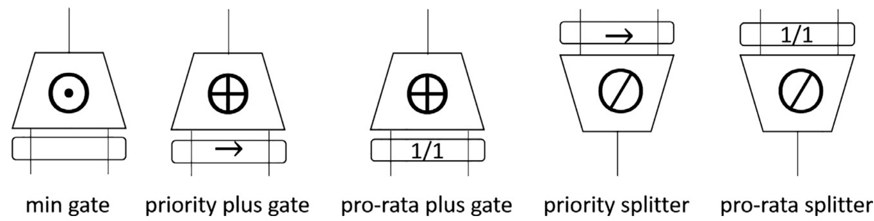

Fig. 11. Graphical representation of *min*-, *plus*-, and *splitter*-gates.

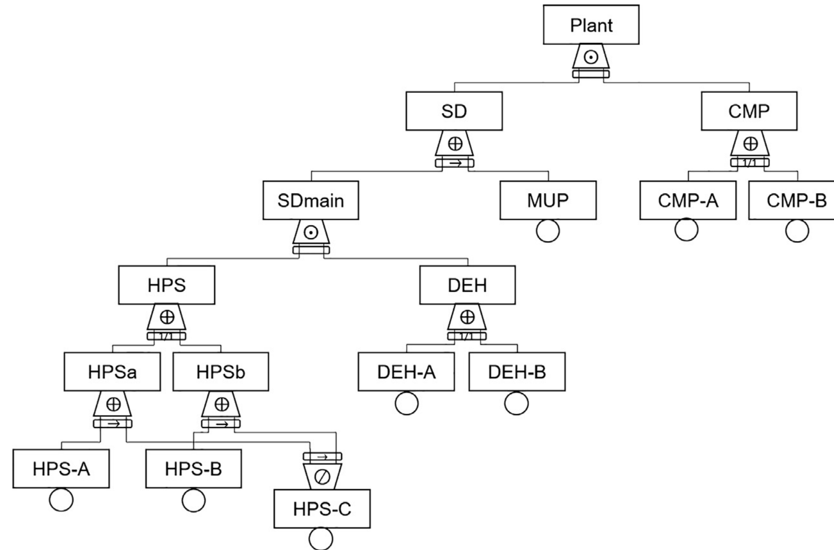

Fig. 12. Tree-like representation of the model of the plant.

Table 2
Gates of the model.

Element	Type	Policy	Capacity	Fanins
Plant	*min*-gate		100	SD, CMP
SD	*plus*-gate	priority	100	SDMain, MUP
SDMain	*min*-gate		100	HPS, DMP
HPS	*plus*-gate	pro-rata	100	HPAam, HPAbm
HPSa	*plus*-gate	priority	55	HPS-A, SPLT(1st fanout)
HPSb	*plus*-gate	priority	55	HPS-B, SPLT(2nd fanout)
SPLT	*splitter*-gate	priority		HPS-C
DEH	*plus*-gate	pro-rata	100	DEH-A, DEH-B
CMP	*plus*-gate	pro-rata	100	CMP-A, CMP-B

The necessary introduction of the *splitter*-gate denoted as explains in turn why three flows (capacity, demand, production) are mandatory.

Table 2 presents gates of the model in a tabular form.

4.3. Graphical representation

Fig. 11 gives a possible graphical representation for *min*-, *plus*-, and *splitter*-gates.

The graphical representation for the whole model is obtained from the functional decomposition of Fig. 2 by adding representations of gates. The *splitter*-gates SPLT is not represented in the functional breakdown and must thus be added. The tree-like representation for the model of the plant is given Fig. 12.

4.4. Block diagrams

Some analysts may prefer to represent the system by means a flow network (block diagrams). In such diagram, the functional breakdown is represented by means of hierarchy of nested boxes. In our use case, the outer most box would represent the Plant and would have two nested boxes in series SD and CMP. These two boxes would in turn contain nested boxes (in series or in parallel) and so on. Intuitively, *min*-gates and *plus*-gates in a production trees correspond respectively to series and parallel assemblies in block diagrams (and of course, basic components in production trees correspond to basic blocks in block diagrams).

The modeling patterns presented above require only minor adjustments to describe block diagram representations.

In block diagrams, basic blocks have an input. Therefore, we need to add three flow variables to our pattern for basic components, namely an `inCapacity`, an `outDemand` and an `inProduction`. The assertion is modified accordingly:

```
assertion
outCapacity:=iffailedthen0elsemin(inCapacity,
capacity);
outDemand:=inDemand;
outProduction:=min(outCapacity,
inProduction);
```

To represent basic blocks with no input, it suffices to set `inCapacity` to the reference value (100 in our case), and `inProduction` to `outDemand`.

Now, let consider the series assembly of two (non necessarily basic) blocks *A* and *B*. The `outCapacity` of *A* is thus plugged on the `inCapacity` of *B*. Therefore, the `outCapacity` of the assembly, which is the `outCapacity` of *B*, is the minimum of the `outCapacity` of A and the intrisic capacity of *B*, just as what is obtained by means of a *min*-gate. A similar reasoning can be applied to demand and production flows.

For parallel assemblies, the situation is even simpler as *plus*-gates can be used directly.

Block diagram and tree-like representations are thus equivalent, up to minor adjustments. The choice between the two is to a large extent a matter of taste.

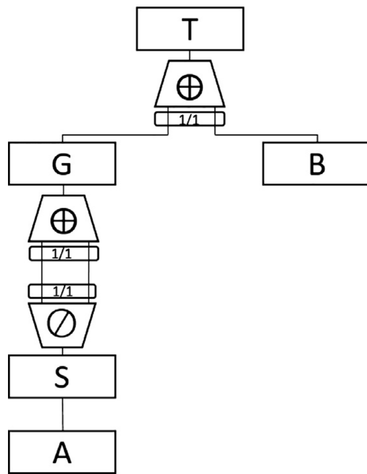

Fig. 13. Splitter deceiver.

4.5. Discussion

The patterns for basic components, combinatorial gates and splitters are thus sufficient to address our case study, which is relatively simple. To complete the model, it suffices to plug the output capacity of the top node, in our case node Plant, to its input demand. The overall production of the system will be then the output production of the top node.

We shall see in the next section more advanced patterns to handle more complex systems. But even these advanced patterns are not sufficient to warranty an optimal production in all case. In fact, no model in which decisions about demand splitting are made locally can.

To illustrate this point, consider the model pictured Fig. 13. We could call this arrangement of basic components and gates a *"splitter deceiver"*.

Assume that both basic components *A* and *B* can produce 50, so that the expected production is 100. Moreover, assume that gate *S* is a pro-rata splitter and that both gates *G* and *T* are pro-rata *plus*-gates. In this case, the two output capacities of *S* (50 each) are added up in the gate *G* whose output capacity is thus 100. Added up with the output capacity of the basic component *B*, the total output capacity of *T* is thus wrongly evaluated at 150. Considering this over-capacity, the top gate *T* then allocates a demand 100/150, that is, about 66 to gate *G* and the remainder (34) to the basic component *B*. But the basic component cannot produce more than 50. So, eventually, the maximum production will be evaluated to 84 while it is actually 100. Thus the gate *G* deceives the splitter *S*.

This is a quite artificial construct, but the reader can convince herself or himself that whatever local policy is chosen, there is always an arrangement of components and gates that deceives it. This problem has however limited practical consequences if the analyst takes care not to design models with this kind of splitter deceiving arrangement of components and gates.

To conclude, note that there exist algorithms to calculate maximum flows in a network like the well known Ford-Fulkerson method [26]. However, the application of this type of algorithms at each step of a Monte-Carlo simulation would be prohibitive in terms of calculation resources.

5. Advanced patterns

In this section, we shall review some advanced patterns for both basic components and gates.

5.1. Basic components

The pattern for basic components proposed in Section 4.2.1 is rather simple. It can be extended in several ways. For instance, it is possible to introduce degraded states in which the component is still producing, but not at its full capacity.

In the pattern, it is assumed that the component starts producing when it receives an input demand. However, some components, like diesel generators, have a small but non null probability to fail on demand. Thus states and transitions can be introduced to capture failures on demand.

Moreover, the repair of the component may not start immediately after its failure, which is the case when the repair crew is busy with another component. Here again states and transitions can be introduced to capture non immediate repairs.

Fig. 14 shows the graphical representation of an advanced pattern for basic components.

The component in Fig. 14 may be in one of the four states represented by symbolic constants `STANBY`, `WORKING`, `FAILED` and `REPAIR`.

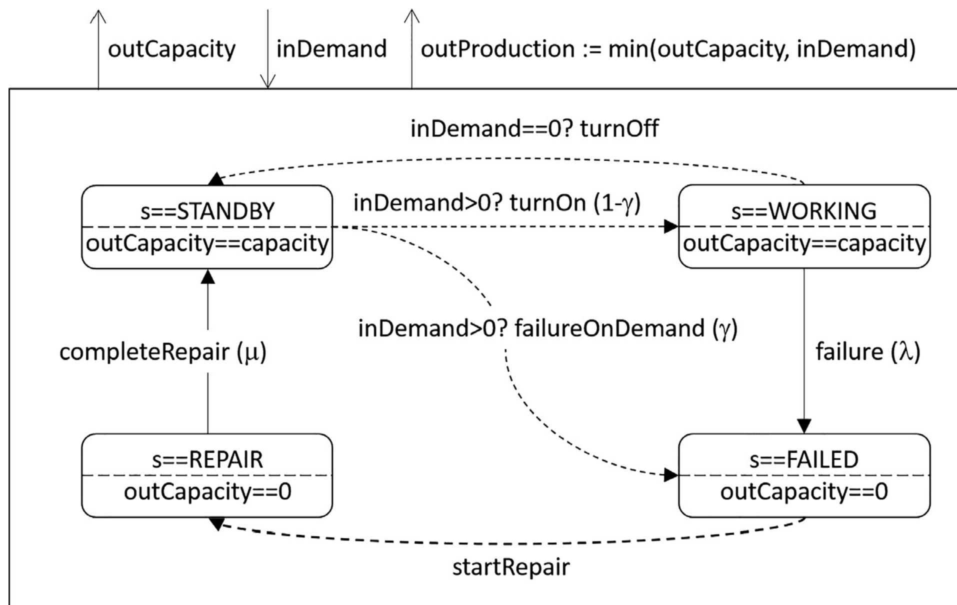

Fig. 14. Advanced pattern for basic components.

```
domain Mode {AB, AC, BC}
class RoundRobinGate
  Mode mode (init = AB);
  Integer outCapacity (reset = 0);
  Integer inDemand (reset = 0);
  Integer outProduction (reset = 0);
  Integer inCapacity1, inCapacity2, inCapacity3 (reset = 0);
  Integer outDemand1, outDemand2, outDemand3 (reset = 0);
  Integer inProduction1, inProduction2, inProduction3 (reset = 0);
  parameter Integer capacity = 100;
  parameter Real tau = 100;
  event turnToNextMode (delay = tau);
  transition
    turnToNextMode: mode==AB -> mode := AC;
    turnToNextMode: mode==AC -> mode := BC;
    turnToNextMode: mode==BC -> mode := AB;
  assertion
    outCapacity := min(capacity,inCapacity1+inCapacity2+inCapacity3);
    switch {
      case mode==AB: {
        outDemand1 := inDemand * inCapacity1/(inCapacity1+inCapacity2);
        outDemand2 := inDemand * inCapacity2/(inCapacity1+inCapacity2);
        outDemand3 := 0;
        outProduction := inProduction1+inProduction2;
        }
      case mode==AC: {
        outDemand1 := inDemand * inCapacity1/(inCapacity1+inCapacity3);
        outDemand2 := 0;
        outDemand3 := inDemand * inCapacity3/(inCapacity1+inCapacity3);
        outProduction := inProduction1+inProduction3;
        }
      case mode==BC: {
        outDemand1 := 0;
        outDemand2 := inDemand * inCapacity2/(inCapacity2+inCapacity3);
        outDemand3 := inDemand * inCapacity3/(inCapacity2+inCapacity3);
        outProduction := inProduction2+inProduction3;
        }
      }
end
```

Fig. 15. A gate implementing a round-robin controller.

Stochastic transitions are pictured with plain arrows. Immediate transition, that is transitions with a null delay, is represented with dashed arrows.

Transitions may be conditioned not only by the value of the state variable `s`, but also by the value of the flow variable `inDemand`. These conditions are represented by prefixing the event with a transition with the condition followed by a question mark.

Transitions `failureOnDemand` and `turnOn` have the same guard. The weight of the former is γ (the probability of failure on demand) while the weight of the later is $1 - \gamma$.

The transition that starts the repair is an immediate transition. It may be however synchronized with a transition of the GTS representing the repair crew so that the repair starts only when there is a repair crew available. Synchronizations are a very powerful mechanism (in addition to flow variables) to represent interactions between components. They can be used also to represent common cause failures (see e.g. [17]).

5.2. Switches

The advanced pattern proposed in the previous section introduces a form of control in the model. Changes of states, typically from `STANDBY` to `WORKING` and vice-versa, and from `FAILED` to `REPAIR` are control actions, that is the decisions are not made by the component itself, but

Table 3
Values of observers for a mission time of 10 years (87600 hours).

Observers	Mean	Standard deviation
Plant.outProduction	99.3034	5.38
HPS.outProduction	99.3026	5.10
DEH.outProduction	99.3516	4.89
CMP.outProduction	99.3523	5.56
HPSC.outProduction	3.1915	12.22
MUP.outProduction	0.6477	4.89

by an automated controller or a human operator, even though this controller may not be explicitly modeled.

Introducing control aspects into safety (and production availability) models gets mandatory because software plays an increasing role in all types of industrial systems, as advocated for instance in [27].

Control actions may take place not only at component level, but also at system and subsystems levels. For instance, in the case study, the operation policy may consist in switching alternately between the high pressure separators to treat the incoming gas (rather than to have HPS-C as a cold spare). The idea is thus to implement a round-robin $AB \rightarrow AC \rightarrow BC \rightarrow AB \ldots$ with a change every time period τ.

The corresponding controller can be represented in the *production trees* framework by introducing a dedicated gate. This gate will embed some behavior, conversely to purely combinatorial gates we have seen so far. A possible implementation of this gate is given Fig. 15. Note that, for the sake of the brevity, changes of modes due to failures of active separators have not been included.

5.3. Multi-flow

In oil and gas production systems, the flow is a combination of oil, gas and water. Units like separators of our illustrative example are precisely in charge of separating a mix of oil, gas and water. It is quite easy to accommodate our modeling patterns to handle flows of different physical nature. It suffices basically to create as many capacity/demand/production flow variables as there are different physical flows. Constraints can be added in assertions to ensure that the proportion of each physical flow corresponds to the reality.

6. Experimental results

We assessed the production tree model of the use case with the AltaRica 3.0 stochastic simulator (see [28] for a description of this tool). This simulator works by defining observers, that is quantities on which statistics are made. For the test case, we defined the following observers:

- The total production of the system.
- The productions of subsystems HPS, DEH and CMP.
- The production of units HPSC and MUP.

Defining these observers is very easy in our framework, since it suffices to look at outProduction of the corresponding gates (or basic units).

Numerical results are not very important here. Nevertheless, Table 3 provides values of observers obtained using a Monte-Carlo simulation of 10^6 histories (which is completed in about 1 minute on a desktop computer) for a mission time of 10 years (87600 hours). These results are in line with those obtained analytically by Kawauchi and Rausand [4].

7. Conclusion

In this article, we proposed a new modeling methodology for production availability analyses. This methodology aims at combining the simplicity of fault trees with the expressive power of guarded transition systems and AltaRica 3.0.

The approach relies on the library *Production Trees* of AltaRica modeling patterns we developed. These patterns can be used as on-the-shelf, reusable modeling components. They can also be easily adjusted to particular needs. The library can be extended with new patterns, typically to describe advanced control policies for plant reconfiguration strategies or maintenance strategies. Incorporating descriptions of controllers into safety (or production availability) models becomes mandatory because industrial systems embed more and more softwares.

We showed by means of experiments on a small test case that very interesting insights about the system can be obtained from production trees, beyond the expected production of the system as a whole throughout a given period of time.

Beyond this study, we strongly believe that guarded transition systems (and AltaRica 3.0) provide a very powerful mathematical and algorithmic framework to assess performance of systems subject to uncertainties (such as hazards, failures...). Moreover, the modeling patterns approach makes it possible to design, on top of this framework, efficient engineering methodologies.

Acknowledgments

This work has been partially supported by the Chaire Blériot-Fabre (CentraleSupélec/SAFRAN). The authors would like also to thank Benjamin Aupetit, from IRTSystemX, for his help for the experiments.

References

[1] International Standardization Organization, ISO 20815 Petroleum, petrochemical and natural gas industries: Production assurance and reliability management. 2008.
[2] Aven T. Availability evaluation of oil/gas production and transportation systems. Reliab Eng 1987;18(1):35–44.
[3] Hokstad P. Assessment of production regularity for subsea oil/gas production systems. Reliab Eng Syst Saf 1988;20:127–46.
[4] Kawauchi Y, Rausand M. A new approach to production regularity assessment in the oil and chemical industries. Reliab Eng Syst Saf 2002;75:379–88.
[5] Signoret J-P. Production availability. In: Soares CG, editor. Proceedings of Probabilistic Safety Assessment and Management, ESREL'2010. CRC Press; 2010. p. 331–45. ISBN 9780415663922.
[6] Boiteau M, Dutuit Y, Rauzy A, Signoret J-P. The AltaRica data-flow language in use: assessment of production availability of a multistates system. Reliab Eng Syst Saf 2006;91(7):747–55.
[7] Zio E, Baraldi P, Patelli E. Assessment of the availability of an offshore installation by Monte-Carlo simulation. Int J Press Vessels Pip 2006;83(4):312–20.
[8] Aven T, Pedersen LM. On how to understand and present the uncertainties in production assurance analyses, with a case study related to a subsea production system. Reliab Eng Syst Saf 2014;124:165–70.
[9] Rauzy A. Towards a sound semantics for dynamic fault trees. Reliab Eng Syst Saf 2015;142:184–91.
[10] Dugan JB, Bavuso SJ, Boyd MA. Dynamic fault-tree models for fault-tolerant computer systems. IEEE Trans Reliab 1992;41(3):363–77.
[11] Bouissou M, Bon J-L. A new formalism that combines advantages of fault-trees and Markov models: Boolean logic-driven Markov processes. Reliab Eng Syst Saf 2003;82(2):149–63.
[12] Rauzy A. Guarded transition systems: a new states/events formalism for reliability studies. J Risk Reliab 2008;222(4):495–505.
[13] Prosvirnova T, Batteux M, Brameret P-A, Cherfi A, Friedlhuber T, Roussel J-M, et al. The AltaRica 3.0 project for model-based safety assessment. In: Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS'2013. York, Great Britain: International Federation of Automatic Control; 2013. p. 127–32. ISBN 978-3-902823-49-6.
[14] Prosvirnova T. AltaRica 3.0: a model-based approach for safety analyses. Palaiseau, France: Ecole Polytechnique; 2014.
[15] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns – Elements of Reusable Object-Oriented Software. Boston, MA 02116, USA: Addison-Wesley; 1994. ISBN 978-0201633610.
[16] Kehren C. Motifs formels d'architectures de systèmes pour la sûreté de fonctionnement, Toulouse, France: Ecole Nationale Supérieure de l'Aéronautique et de l'Espace (SUPAERO); 2005. Thèse de doctorat.
[17] Mortada H, Prosvirnova T, Rauzy A. Safety Assessment of an Electrical System. In: Proceedings of the 4th International Symposium on Model-Based Safety Assessment, IMBSA 2014. In: LNCS, 8822. Munich, Germany: Springer Verlag; 2014. p. 181–94.
[18] Ajmone-Marsan M, Balbo G, Conte G, Donatelli S, Franceschinis G. Modelling with Generalized Stochastic Petri Nets, Wiley Series in Parallel Computing. New York, NY, USA: John Wiley and Sons; 1994. ISBN 978-0471930594.
[19] Dutuit Y, Châtelet E, Signoret J-P, Thomas P. Dependability modeling and evaluation by using stochastic Petri nets: application to two test cases. Reliab Eng Syst Saf 1996;55:117–24.
[20] Batteux M., Prosvirnova T., Rauzy A.. Altarica 3.0 assertions: the why and the wherefore. 2016a. Article submitted to Journal of Risk and Reliability.

[21] Noble J, Taivalsaari A, Moore I. Prototype-Based Programming: Concepts, Languages and Applications. Berlin and Heidelberg, Germany: Springer-Verlag; 1999. ISBN 978-9814021258.

[22] Batteux M, Prosvirnova T, Rauzy A. System Structure Modeling Language (S2ML). AltaRica Association; 2015. Archive hal-01234903, version 1.

[23] Batteux M, Prosvirnova T, Rauzy A. Models of structures, structures of models. 2016b. Article submitted to Systems Engineering.

[24] Friedenthal S, Moore A, Steiner R. A Practical Guide to SysML: The Systems Modeling Language. San Francisco, CA 94104, USA: Morgan Kaufmann. The MK/OMG Press; 2011. ISBN 978-0123852069.

[25] Simon I. Limited subsets of a free monoid. In: Proceedings of the 19th Annual Symposium on Foundations of Computer Science. Piscataway, NJ, USA: IEEE; 1978. p. 143–50. doi:10.1109/SFCS.1978.21.

[26] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms (Second ed.). Cambridge, MA, USA: The MIT Press; 2001.

[27] Piriou P-Y, Faure J-M, Lesage J-J. Modeling standby redundancies in repairable systems as guarded preemption mechanisms. In: Mellado EL, Lefebvre D, Ortmeier F, editors. Proceedings of 5th IFAC Conference on Dependable Control of Discrete Systems (DCDS'2015), 48. Cancun, Mexico: IFAC-PapersOnLine; 2015. p. 147–53.

[28] Aupetit B, Batteux M, Rauzy A, Roussel J-M. Improving performance of the AltaRica 30 stochastic simulator. In: Podofillini L, Sudret B, Stojadinovic B, Zio E, Kröger W, editors. Proceedings of Safety and Reliability of Complex Engineered Systems: ESREL 2015. CRC Press; 2015. p. 1815–24. ISBN 9781138028791.