# The AltaRica Data-Flow Language in Use: Modeling of Production Availability of a MultiStates System

M. Boiteau(1), Y. Dutuit(2), A. Rauzy(3) and J.-P. Signoret(4)

(1) 8 rue samonzet, 64000 Pau, FRANCE marie.boiteau@fractal-systeme.com

(2) LAP, Université Bordeaux I, 33405 Talence Cedex, FRANCE, Yves.Dutuit@iut.u-bordeaux1.fr

(3) IML/CNRS, 169 avenue de Luminy, 13288 Marseille Cedex 09 FRANCE, arauzy@iml.univ-mrs.fr

(4) Total, CSTJF, Avenue Larribau, 64018 Pau Cedex, Jean-Pierre.Signoret@total.com

**Abstract:** This article presents an application of the AltaRica Data-Flow language to the assessment of the average production of a production line throughout a period of time. Its contribution is twofold. The problem is described. Modelling difficulties are pointed out. Solutions are proposed via the use of the AltaRica Data-Flow language. The ability of this language to represent complex systems is demonstrated. It is shown, by means of an experimental study, that AltaRica Data-Flow models can be assessed efficiently.

## 1. Introduction

This article presents an application of the AltaRica Data-Flow language to the assessment of the average production of a production line throughout a period of time. In many industries, e.g. oil extraction platforms and chemical plants, this problem has a strong economic impact. We consider here the production line as a network (a directed acyclic graph) with production units at the one end (root nodes) and storage units at the other end (sink nodes). In between, a number of treatment units transform the production. Units may be subject to different failure modes. There may be spare units (cold redundancies). Each unit has a limited production (or treatment) capacity that depends on its state. Units may be repaired. There may be a limited number of repair crews. The repairing policy may depend on various parameters. The problem is to assess the average production for a given period of time, i.e. the mathematical expectation of the production, of the whole production line and of each unit independently. This problem raises a number of modeling difficulties, some of which have been already studied in the literature, see e.g. reference [1].

To handle this problem at an industrial scale, a high level description formalism is required. Without such formalism, models are hard to design and to maintain. Low level descriptions are by far too error prone. Moreover, even minor changes in the system specification may affect strongly the model. AltaRica Data-Flow is a high level description language devoted to risk assessment studies. It can be seen as a generalization of both Petri nets and Block Diagrams [2]. To Petri nets, it borrows the notion of states, events and guarded transitions. To Block Diagrams, it borrows the notion of hierarchical descriptions and flows circulating through a network. All of these features make AltaRica Data-Flow especially suitable to model production availability problems.

The design (or the choice) of a description formalism results of a trade-off. On the one hand, its modelling power must be sufficient enough to express the problem to be solved. On the other hand, calculations to be performed must remain tractable. From an algorithmic view point, the assessment of production availability is strongly related to the resolution of Markov Reward Model [3] and Stochastic Petri Nets Reward Models [4, 5]. We report here results of experiments performed with two tools designed to assess AltaRica Data-Flow descriptions, namely a Markov analyser and a stochastic simulator. We show that both are of interest, each with its advantages and its drawbacks. These experiments show that one of the major advantages of high level description formalisms, such as AltaRica Data-Flow, is that they make it possible to apply various algorithmic tools to the same model.

The remainder of this article is organized as follows. Section 2 presents a generic test case that illustrates the problem and its difficulties. Section 3 introduces basic concepts of the AltaRica Data-Flow language. Hierarchical descriptions are introduced in section 4. Synchronization of events, which is one of the most interesting features of the language, is introduced in section 5. Section 6 shows how transitions are interpreted, i.e. how probability distributions for their durations are introduced. Section 7 discusses the remote interaction of treatment units and production units. Section 8 presents experimental results. Finally, sections 9 and 10 conclude the article.

## 2. Working Example

As an illustrative example, we shall consider the system pictured on Fig. 1. It represents a part of an oil extraction installation. Although close to a real-life system, this test case has been designed to concentrate most of the modelling difficulties of the assessment of production availability.

- *W1* and *W2* are wells. Their production capacities are respectively *160* and *70* (for the sake of the simplicity, all capacities are normalized; they actually represent a given amount of barrels/day). When they are failed, the production is stopped.
- *T1* and *T2* are tanks. They are assumed to be perfectly reliable. Their storage capacities are respectively *110* and *100*.

- *A*, *B* are two treatment units. They have two failure modes: a failure that decreases their capacities from *170* to *100* (resp. *120* to *70*), and a severe failure that stops the treatment. The latter may occur either when the unit is working correctly or when it is in a degraded mode.
- Components *Ci*'s, *Di*'s and *Ei*'s are treatment units. Their individual capacities are respectively *120*, *80* and *50*. For all of these units, a failure stops the treatment. Components of bloc *E* are in hot redundancy.
- The line *D* is in cold redundancy with the line *C*. As soon as the line *C* is repaired, the line *D* is stopped. If *C1* fails, then *C2* is stopped and vice-versa. Same thing for *D1* and *D2*.
- *R* is a pool of two repair men (i.e. at most two components can be repaired simultaneously). A component is not able to treat the production during a repair. No particular repair policy is carried out when more than three units are down.
- The production entering in component *A* must be split into two fractions: a fraction goes to the tank *T1* through the top line, the remainder goes to the tank *T2* through the bottom line. This requires defining a splitting policy. We assume the following one. The production of *W1* goes preferably to the top line. The production of *W2* goes preferably to the bottom line. If needed, what remains available from *W1* goes to the bottom line.

Table 2 gives failure rates $\lambda$ ($\lambda_s$ for severe failures of components *A* and *B*), repair rates $\mu$ ($\mu_s$ for repairs of severe failures), and probability to fail on demand $\gamma$ for each component *D1* and *D2*.
The problem is to assess the average production of wells W1 and W2 and storage of tanks T1 and T2 for a given period of time.

It is clear that the supply of oil for a given unit depends on the states of all upper units. The reciprocal is also true: the demand of oil of a given unit depends of the states of all lower units. The model has to take into account these remote (and in some sense circular) interactions. Moreover, from an engineering viewpoint, it is desirable to design models by assembling reusable pieces (like a Lego construction). Therefore, remote interactions should be described by means of local connections only.

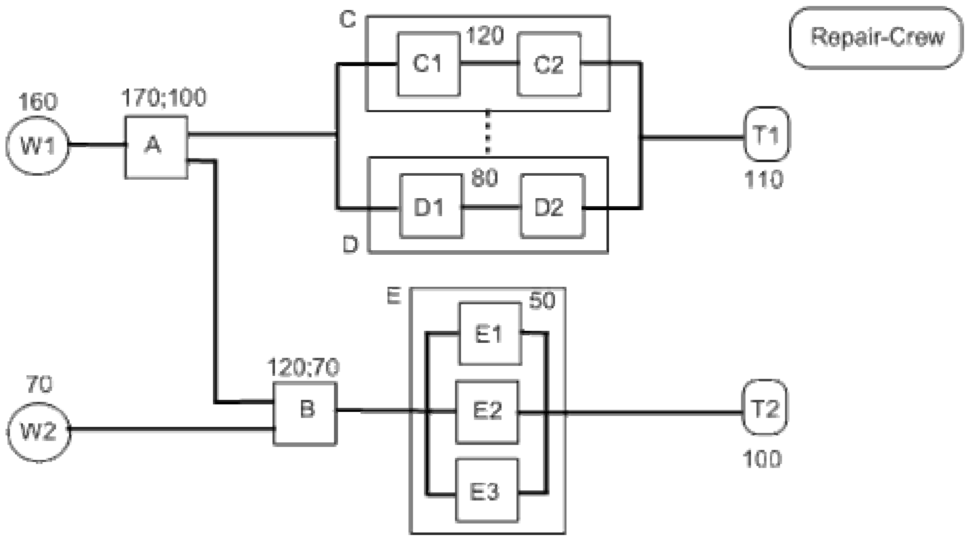

**Figure 1.** A Part of a Production System.

| | $\lambda$ | $\mu$ | $\lambda_s$ | $\mu_s$ | $\gamma$ |
|---|---|---|---|---|---|
| *W1, W2* | $2\ 10^{-5}$ | $5\ 10^{-3}$ | | | |
| *A, B* | $9\ 10^{-3}$ | $2\ 10^{-2}$ | $3\ 10^{-5}$ | $4\ 10^{-3}$ | |
| *C1, C2* | $3\ 10^{-3}$ | $4\ 10^{-2}$ | | | |
| *D1, D2* | $8\ 10^{-3}$ | $4\ 10^{-2}$ | | | $2\ 10^{-2}$ |
| *E1, E2, E3* | $4\ 10^{-3}$ | $5\ 10^{-2}$ | | | |

**Table 2.** Transition rates for the system depicted on Fig. 1.

## 3. Mode Automata

Consider the component `E1` of Fig. 1. It may be in three states: working or failed or in repair. It goes from state “working” to state “failed” when a failure occurs. It goes from state “failed” to state “in repair” when the repair starts. Finally, it comes back from state “in repair” to state “working” when the repair ends. States “failed” and “in repair” need to be distinguished because of the limited number of repair men: the repair of the component does not necessarily start as soon as the failure occurs because it depends on the availability of a repair man. The amount of oil treated by `E1` depends on its treatment capacity, on its internal state and finally on the quantity it receives in input. Fig. 3 shows a state graph to model `E1`.

The AltaRica Data-Flow language relies on the notion of mode automaton [2]. A mode automaton is a states/transitions system with input and output flows. In each state, so-called a mode, the automaton realizes a transfer function, i.e. it computes the values of output flows from the values of input flows. The automaton may change of mode when an event occurs. Boolean conditions, so-called guards, indicate whether an event may occur or not, i.e. whether the system may perform the corresponding transition. A formal definition of mode automata can be found in reference [2]. Fig. 3 shows actually a mode automaton. The AltaRica Data-Flow code for this automaton is given by Fig. 4.

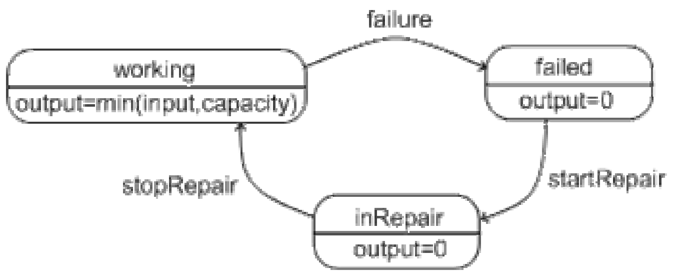

**Figure 3.** A Mode Automaton for the component `E1`.

```
domain unitEState = { working, failed, inRepair };

node unitE
  state s:unitEState;
  flow  oilIn:float:in; oilOut:float:out; capacity:float:in;
  event failure, startRepair, endRepair;
  trans
    (s=working)  |- failure     -> s := failed;
    (s=failed)   |- startRepair -> s := inRepair;
    (s=inRepair) |- endRepair   -> s := working;
  assert
    oilOut = if (s=working) then min(oilIn,capacity) else 0;
  init
    s := working;
edon
```

**Figure 4.** The AltaRica Data-Flow code for the components `Ei`'s.

Modes of basic components of AltaRica Data Flow descriptions are described by means of state variables. In our example there is only one state variable: `s`. Input and output flows are described by means of flow variables. In our example there are two input flows (`oilIn` and `capacity`) and one output flow (`oilOut`). The variable `capacity` does not represent a physical flow. It is just a convenient mean to reuse the same code for components that differ only by their treatment capacity. Variables (states or flows) are typed. A variable may be either a Boolean, or an integer, or a float or an enumerated set (as for the variable `s`). Events and transitions are used to describe changes of modes. Assertions describe transfer functions. Init clauses describe the initial state of the component.

It is worth noticing that, as for Petri nets [6], the values of variables (states and flows) can evolve only when a transition is triggered. When a transition is triggered, first state variables are updated, and then values of output flows are recomputed according to the assertion. The whole operation is assumed to be infinitely fast. Therefore, in AltaRica Data-Flow, the time is just seen as a sequence of events.

## 4. Hierarchical Descriptions

Consider now the subsystem `E` made of components `E1`, `E2` and `E3` (in parallel). These three components are independent. The oil entering the subsystem `E` is distributed over the `Ei`'s. If one of the `Ei`'s is failed, then the oil must be distributed over the two others. The distribution policy therefore depends at least on the states of the `Ei`'s. Other factors could be taken into account, such as preferable loads for units. If one is interested in studying the production of components individually, then this policy has to be represented explicitly. The subsystem `E` may be also considered as a whole, which makes it possible to simplify its description, as follows.

```
node subsystemE
  flow oilIn:float:in; oilOut:float:out;
  sub  E1:unitE; E2:unitE; E3:unitE;
  assert
    E1.oilIn = oilIn,
    E2.oilIn = oilIn,
    E3.oilIn = oilIn,
    oilOut   = min(oilIn,E1.oilOut+E2.oilOut+E3.oilOut);
edon
```

**Figure 5.** The AltaRica Data-Flow code for the Subsystem `E`.

```
node subsystemE
  state E1.s:unitEDomain; E2.s:unitEDomain; E3.s:unitEDomain;
  flow  oilIn:float:in; oilOut:float:out;
        E1.oilIn:float:in; E1.oilOut:float:out; E1.capacity:float:in;
        E2.oilIn:float:in; E2.oilOut:float:out; E2.capacity:float:in;
        E3.oilIn:float:in; E3.oilOut:float:out; E3.capacity:float:in;
  event E1.failure, E1.startRepair, E1.endRepair,
        E2.failure, E2.startRepair, E2.endRepair;
        E3.failure, E3.startRepair, E3.endRepair;
  trans
    (E1.s=working)  |- E1.failure     -> E1.s := failed;
    (E1.s=failed)   |- E1.startRepair -> E1.s := inRepair;
    (E1.s=inRepair) |- E1.endRepair   -> E1.s := working;
    (E2.s=working)  |- E2.failure     -> E2.s := failed;
    (E2.s=failed)   |- E2.startRepair -> E2.s := inRepair;
    (E2.s=inRepair) |- E2.endRepair   -> E2.s := working;
  assert
    E1.oilOut = if (E1.s = working) then min(E1.oilIn,E1.capacity) else 0,
    E2.oilOut = if (E2.s = working) then min(E2.oilIn,E2.capacity) else 0,
    E3.oilOut = if (E3.s = working) then min(E3.oilIn,E3.capacity) else 0,
    E1.oilIn  = oilIn,
    E2.oilIn  = oilIn,
    E3.oilIn  = oilIn,
    oilOut    = min(oilIn,E1.oilOut+E2.oilOut+E3.oilOut);
  init
    E1.s := working, E2.s := working, E3.s := working;
edon
```

**Figure 6.** A basic component equivalent to the subsystem of Fig. 5.

The idea is to consider the maximum amount of oil each `Ei` is able to treat. The amount of oil treated by the subsystem is then the minimum of the amount of oil entering it on the one hand, and the sum of the maximum amounts of oil treated by its components on the other hand. This idea works for any parallel system. It is implemented by the AltaRica Data-Flow code given by Fig. 5. This code is rather straightforward and does not deserve long explanations.

`E` is in some sense the "product" of the `Ei`'s. Reference [2] defines formally the notion of product of two or more mode automata. This operation is internal: its result is a mode automaton. The construction of the mode automaton for the subsystem `E` is performed in two steps:

- First, three fresh copies of component `componentE` are created together with the additional flow variables `oilIn`, `oilOut` and `capacity`. The modes of the product are triples of modes of its components. Its events and transitions are the events and transitions of its components. Its assertion is the conjunction of the assertions of its components.
- Second, flows are connected. This second step restrains the behaviour of the free product obtained at the previous step. In our case, this restriction implements the oil splitting policy.

The resulting automaton could be described as an AltaRica Data-Flow node, as illustrated by Fig. 6. In practice, this operation is performed by assessment tools from the codes of Fig. 4 and Fig. 5.

One of the major advantages of a hierarchical formalism, such as AltaRica Data-Flow, is that our description of the subsystem `E` could be unplugged of the description of the global system and replaced by another one (that implements, for instance, a more detailed oil distribution policy). In this way, a particular component or subsystem can be studied *mutatis mutandis* at various levels of details and may be modified easily as the system specifications evolve.

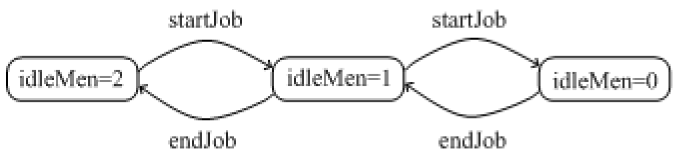

**Figure 7.** Mode automaton for a pool of two repair men.

```
node repairPool
  state idleMen:int;
  event startJob, endJob;
  trans
    (idleMen>0) |- startJob -> idleMen := idleMen-1;
    true        |- endJob   -> idleMen := idleMen+1;
  init
    idleMen := 2;
edon
```

**Figure 8.** AltaRica code for the repair pool.

```
node main
  sub   …; E1:unitE; E2:unitE; R:repairPool; …;
  event …, startRepairE1, startRepairE2, …;
  …
  sync
    …
    <startRepairE1: (E1.startRepair and R.startJob)>;
    <startRepairE2: (E2.startRepair and R.startJob)>;
    …
edon
```

**Figure 9.** A fragment of AltaRica code to illustrate the synchronization of events.

```
    (E1.s=failed) and (R.idleMen>0) |- startRepairE1 ->
        E1.s      := if (E1.s=failed) then inRepair else E1.s,
        R.idleMen := if (R.idleMen>0) then R.idleMen-1 else R.idleMen;

    (E2.s=failed) and (R.idleMen>0) |- startRepairE2 ->
        E2.s      := if (E2.s=failed) then inRepair else E2.s,
        R.idleMen := if (R.idleMen>0) then R.idleMen-1 else R.idleMen;
```

**Figure 10.** Global transitions created by the synchronization of Fig. 9.

## 5. Synchronizations

Consider now the pool of repair men. A repair man may be idle or busy. When a repair man is called on a unit, she/he goes from state idle to state busy. When the repair ends, he makes the reverse transition. If we assume the repair men indistinguishable, the pool can be described by the mode automaton pictured on Fig. 7. The AltaRica Data flow code for this automaton is given by Fig. 8.

At a global level, the events `startRepair` of the units and `startJob` of the repair pool must be simultaneous. They can be actually considered as two faces of the same event (same thing for the events `endRepair` and `endJob)`. In a Petri nets description for instance, to make two events simultaneous, one must merge the corresponding transitions or create a cascade of events with no delay. AltaRica Data-Flow provides a simple construct to perform automatically this operation: the synchronization of events (which is borrowed from the model-checker MEC [7]). This construct is illustrated on Fig. 9. Global events (e.g. `startRepairE1`) are declared together with monotone Boolean functions over local events involved in the synchronization (e.g. `E1.startRepair and R.startJob`). The principle of synchronization is as follows.

- Local events involved in the synchronization are removed from the model together with the transitions they label.
- For each global event `e`, a new transition is created. The guard of this transition is the Boolean function associated with `e` in which the guards of local transitions are substituted for the local events. For each assignment "`v:=E`" of a local transition whose guard is `G`, an assignment "`v:= if G then E else v`" is added to the global transition.

This operation is illustrated on Fig. 10. The global transition labelled with the event `startRepairE1` is fireable if the local transition labelled with the event `E1.startRepair` and the local transition labelled with the event `R.startJob` are fireable. To fire this transition therefore consists in firing simultaneously all the local transitions that can be fired.

This mechanism can be used to implement various types of synchronizations, including sender/receiver communications, broadcasting … It applies also in the case where several transitions are labelled with the same event. Moreover, in AltaRica Data-Flow, synchronizations can be introduced at any level of the hierarchy.

## 6. Interpretation of Transitions

Consider again the automaton pictured on Fig. 3. Events `failure` and `endRepair` are stochastic, i.e. they model ends of actions that have a (possibly unpredictable) duration. In other words, they are fired after a non null randomly distributed delay (if nothing prevents them to be fired in between). This delay may be, for instance, exponentially distributed with a given transition rate. On the converse, the event `startRepair` is immediate, i.e. it is fired as soon as possible. With that respect, AltaRica Data-Flow descriptions can be interpreted in the same way as classical Petri Nets are interpreted into Generalized Stochastic Petri Nets [8]: there are stochastic transitions with given delay distributions and immediate transitions. Priorities may be defined over immediate transitions. Distributions of event durations as well as other tool specific data are given in the clause extern of nodes, as illustrated by Fig. 12. Delays of event `failure` and `endRepair` are assumed to have an exponentially distributed with transition rates respectively `lambda` and `mu`.

Consider now the spare line `D` of Fig. 1. Components `Di`'s are very similar to components `Ej`'s except that they may be in a fourth mode: idle, i.e. waiting to replace components `Ck`'s. When they are requested, i.e. when the line `C` fails, they move from the mode `idle` to the mode `working` (under the occurrence of the event `start`), or to the mode `failed` (under the occurrence of the event `failureOnDemand`). Both transitions are immediate. The behaviour of components `Di`'s is described by the mode automaton pictured on Fig. 11. The AltaRica code given by Fig. 12 implements this mode automaton. The clause extern is used to set the probabilities of events `start` and `failureOnDemand`. The directive bucket indicates that they have to be considered together: when they are both fireable, one (and only one) is chosen and fired. The choice is made at random according to their respective probabilities (here 0.02 and 0.98).

AltaRica Data-Flow makes it possible to give priority to immediate transitions (via the directive `priority`, as illustrated by Fig. 12). These priorities are integers (the higher the number, the higher the probability). Default priority is the lowest, i.e. 0. If two immediate events are fireable simultaneously, the one of higher priority is fired. In our case, priorities are introduced to make sure of the order or events.

On this automaton, `isRequested` is a Boolean flow that goes out of the line `C` and enters into the line `D`. It ensures that the line `D` is started (or at least is attempted to start) as soon as the line `C` fails. The same result could have been achieved by means of synchronizations. However, it is somehow counterintuitive to synchronize a failure of component `C1` with the failure on demand of component D2, although the latter follows the former by an irrelevant time period (at least at the study level).

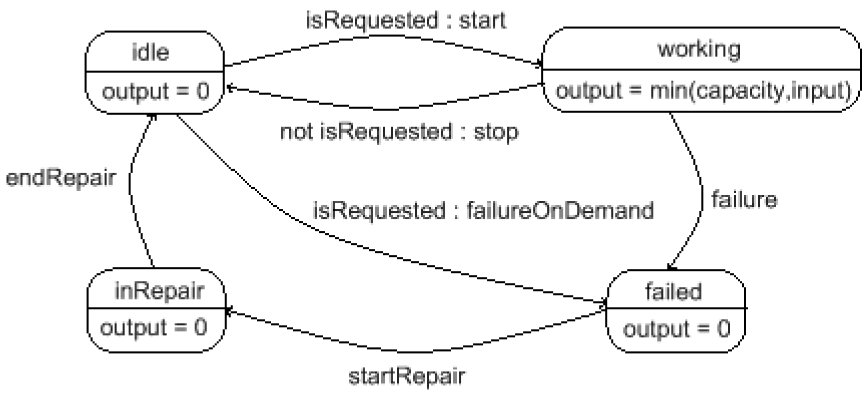

**Figure 11.** A mode automaton to represent spare units.

```
node ComponentD
  state s:{idle, working, failed, inRepair};
  flow  isRequested:bool:in;
        capacity:float:in; input:float:in; output:float:in;
  event failureOnDemand, start, stop, failure, startRepair, endRepair;
  trans
           (s=idle) and isRequested |- failureOnDemand -> s := failed;
           (s=idle) and isRequested |- start           -> s := working;
                        (s=working) |- failure         -> s := failed;
    (s=working) and not isRequested |- stop            -> s := idle;
                         (s=failed) |- startRepair     -> s := inRepair;
                       (s=inRepair) |- endRepair       -> s := idle;
  assert
    output = if (s=working) then min(input,capacity) else 0;
  init
    s := idle;
  extern
    law <event failure>         = exponential(0.008);
    law <event endRepair>       = exponential(0.04);
    law <event failureOnDemand> = constant(0.02);
    law <event start>           = constant(0.98);
    bucket {<event failureOnDemand>, <event start>} = call;
    priority {<event stop>, <event startRepair>} = 1;
edon
```

**Figure 12.** An AltaRica Data-Flow description for a spare unit.

# 7. Production: Supply and Demand

Consider a treatment unit such as `C1`. `C1` receives a given amount of oil from upper units. It delivers it (after treatment) to lower units. This implies that the received amount of oil must not exceed not only the capacity of `C1`, but also the capacity of the whole network below `C1`. In other words, the supply must be regulated by the demand. Both depend on the states of the various units that compose the network.

Flows can be used to model the supply and demand of each unit. Namely, four flow variables are necessary for each unit `U`: `oilInSupply`, `oilOutSupply`, `oilInDemand` and `oilOutDemand` that represent respectively the maximum amount of oil that of upper units can supply to `U`, the maximum amount of oil that `U` can supply to lower units, the actual demand of lower units to `U` and the actual demand of `U` to upper units. The equations that rule these variables are as follows.

```
oilOutSupply = if (s=working) then min(capacity,oilInSupply) else 0;
oilOutDemand = oilInDemand;
```

An implicit additional constraint is that `oilInDemand` must not exceed `oilOutSupply`. In this way, the demand corresponds exactly to what is actually produced. Fig. 13 shows how the consequences of a change of state of one of the unit are propagated through the network: `W.outSupply` depends on `W.state`. `U.inSupply` equals `W.outSupply`. `U.outSupply` depends on `U.state` and `U.inSupply` and so on. Finally, `W.inDemand`, that equals `U.outDemand`, depends on all the variables encountered along the path.

This principle works fine for series systems. However, problems arise when dealing with parallel systems, as illustrated on Fig. 14. This system is made of two parallel lines. The supply that enters in the system (`S.inSupply`) must be split in two parts. A part must go to the upper line and the remainder must go to the bottom line. Similarly, demand `T.inDemand` must be splitted in two parts. The problem is to define the splitting strategy. It would be quite natural to split the in-supply in proportion of out-demands. E.g.

```
U1.inSupply = S.inSupply × U1.outDemand/(U1.outDemand + U2.outDemand)
U2.inSupply = S.inSupply × U2.outDemand/(U1.outDemand + U2.outDemand)
```

Similarly, the global in-demand could be split in proportion of the out-supplies. E.g.

```
U1.inDemand = T.inDemand × U1.outSupply/(U1.outSupply + U2.outSupply)
U2.inDemand = T.inDemand × U2.outSupply/(U1.outSupply + U2.outSupply)
```

This model does not work because of circular dependencies induced by the four above equations. It can be shown that no local description can solve this problem. Therefore, a splitting policy must be defined *a priori*, as we did for the component *A* of our test case, or at a global level.
It is worth noticing that the notion of flows is anyway very useful to model oil demand and supply. Even a powerful formalism such as Petri nets is unable to model this kind of propagation, at least without considering the entire network and performing a case study.

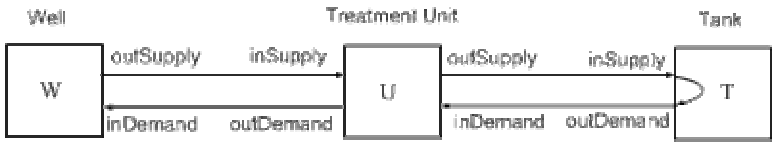

**Figure 13.** Production of a series system.

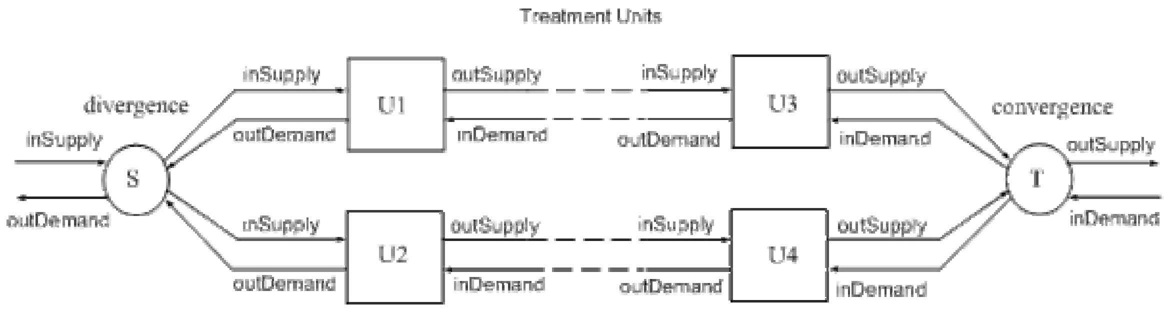

**Figure 14.** Production of two parallel lines.

## 8. Experiments

A number of tools have been designed to assess AltaRica Data-Flow descriptions, including an interactive simulator, a compiler to fault trees, a generator of sequences, a model-checker, a Markov analyzer and a stochastic simulator. We report here results obtained with the two latter tools. Both can be used to assess production availability.

**Markov Analysis:** The Markov graph for the test case presented section 2 has been obtained by compiling the AltaRica description. It is made of *110,768* states and *861,232* transitions. The compilation takes *67* seconds on a modern laptop computer.

To assess this Markov graph, we used the so-called Matrix Exponentiation Method (see [9] for a detailed description of this method). This method is an explicit method to compute transient solutions of Markov graphs. Any such a method can be extended to compute mean sojourn times in each state and from there many quantities of interest, as pointed out by Trivedi & al. in their articles on the notions of Markov Reward Model [3] and Stochastic Petri Nets Reward Models [4,5]. We applied the method for a mission time of 8760 hours (1 year) and with two time steps (the smaller the time step the more accurate the results but the more expensive the algorithm). Results are presented in Tab. 11.

**Stochastic Simulation:** We performed a stochastic simulation over $10^5$ histories. Results are presented in Tab. 12. The running time for a stochastic simulation is roughly proportional to the number of histories. A rather large number of histories is mandatory to obtain precise results.

AltaRica Data-Flow provides special constructs (in the clause `extern`) to describe which quantities are to compute. We introduced various mechanisms to observe these quantities along the time of the mission (mean values, percentiles, chronograms, ...).

|  |  |  |
|---|---|---|
| Time step | 7.08165 | 0.708681 |
| Running time | 176s | 1184s |
| Production of well W1 | 83.6247 | 83.5633 |
| Production of well W2 | 47.6409 | 47.606 |
| Storage in tank T1 | 70.0677 | 70.0163 |
| Storage in tank T2 | 61.1979 | 61.153 |

**Table 11.** Results of the Markov Analysis

|  |  |  |  |  |
|---|---|---|---|---|
| Number of histories | $10^5$ |  |  |  |
| Running time | 5405s |  |  |  |
| Quantity | mean value | standard-deviation | 95% confidence range |  |
| Production of W1 | 83.5477 | 5.65194 | 83.5183 | 83.5771 |
| Production of W2 | 47.6074 | 3.00197 | 47.5918 | 47.623 |
| Storage in tank T1 | 70.0007 | 4.68182 | 69.9763 | 70.025 |
| Storage in tank T2 | 61.1544 | 3.97378 | 61.1337 | 61.175 |

**Table 12.** Results of the Stochastic Simulation

**Comparison:** Markov analysis and stochastic simulation give almost the same results. Markov analysis is more precise and faster. Its drawback stands indeed in the exponential blow up of the size of the underlying graph. Stochastic simulation does not suffer from this problem. It is however very time consuming and unable to deal with rare events. Several authors suggested techniques to decrease the size of the Markov graph, see e.g. reference [10]. Such techniques could be applied here as well. It is certainly even easier to detect symmetries, to group similar states at the description level or while generating the graph.

## 9. Discussion

We have seen so far that AltaRica Data-Flow is a textual language. However, we added here and there drawings to illustrate textual constructs. Graphics are of a definitive interest to design models to understand and to animate them. We believe strongly that no formal language can be used for real if it doesn't provide graphical counterparts to textual constructs. However, it is almost impossible to capture the whole complexity of a model such as those we designed in this article within a single figure. Rather, graphics should be used to show different views of the system under study. Here how flows are circulating, there how spare units are allocated, and so on. The UML description language illustrates this idea in the software design framework [11]. That's the way we use AltaRica Data-Flow in practice.

The modeling of production availability requires real valued variables. Classical Petri nets do not provide that kind of variables. It may be argued that colored Petri nets could be used for that purpose [12, 13]. This is only partly true, for two reasons. First, the notion of flows makes it possible to represent remote interactions in a simple way. This can be done only via cascade of no delay events in Petri nets (colored or not), which by far more error prone. Second, colored Petri nets introduce a degree of complexity in assessment. In colored Petri nets, tokens are dynamically created while in AltaRica Data-Flow, as in classical Petri nets, all the variables are present from the start. In practice, this makes a significant difference in the complexity of algorithms to be used to assess models (and, at a less extent, for stochastic simulation as well). Colored Petri nets have indeed their own merits and make it possible to model systems that neither classical Petri nets nor AltaRica Data-Flow are able to handle.

As pointed out by one of the referees, we did not take into consideration here modeling issues such as:

- The dependence of the failure rate of a unit on the state of neighbouring units;
- The dependencies of failure rates on the flows;
- Transient, continuously evolving production modes.

These problems enter into the so-called dynamic reliability framework. They are indeed of a great importance (and interest). The AltaRica Data-Flow theoretical model can be partly adapted to handle them. For instance, it would be possible to compute new values for transition rates each time an event is fired. On the other hand, continuously evolving processes, that are already hard to describe individually, are even harder to combine one another. Whether a hierarchical formalism such as AltaRica Data-Flow can embed such concepts is an open question.

## 10. Conclusion

In this article, we studied in details the assessment of production availability. We pointed out modeling difficulties. We proposed solutions via the use of the AltaRica Data-Flow language. We showed ability of this language to represent complex systems. We report results of an experimental study that shows that AltaRica Data-Flow model can be assessed efficiently by means of Markov analysis and stochastic simulation.

We advocate the use of such a high level language to model complex problems. Several authors, e.g. Trivedi & al. [4], already advocate the use of Petri nets to generate Markov Rewards Models. AltaRica Data-Flow goes at step farther by embedding important concepts such as the notions of flow, hierarchy, and synchronization of events. Moreover, because of its structure AltaRica Data-Flow makes it possible to detect properties, such as symmetries, that can be used to reduce the size of reachability graphs or to accelerate the Monte-Carlo simulation.

## 11. References

[1] Y. Kawauchi and M. Rausand. A new approach to production regularity assessment in the oil and chemical industries. *Reliability Engineering and System Safety*, 75:379-388, 2002.

[2] A. Rauzy. Modes automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78:1-12, 2002.

[3] A. Reibman, R. Smith and K. S. Trivedi. Markov and Markov Reward model transient analysis: An overview of numerical approaches. *European Journal Of Operational Research*, Vol. 40, pp 257-267. North Holland. 1989.

[4] Stochastic Reward Nets for Reliability Prediction, Jogesh Muppala, Gianfranco Ciardo, and K. S. Trivedi, *Communications in Reliability, Maintainability and Serviceability: An International Journal* published by SAE International, Vol. 1, No. 2, pp. 9-20, July 1994.

[5] G. Ciardo, J. Muppala and K. S. Trivedi. On the Solution of GSPN Reward Models. *Performance Evaluation*, Vol. 12, No. 4, pp. 237-254, July 1991.

[6] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541-580, April 1989.

[7] A. Arnold. MEC: a System for Constructing and Analysing Transition Systems. In J.Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer Verlag, June 1989.

[8] M. AjmoneMarsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. John Wiley and Sons, 1994.

[9] An Experimental Study On Six Algorithms to Compute the Transient Probabilities of Large Markov Models. To Appear In *Reliability Engineering and System Safety*.

[10] M. Lanus, L. Yin and K. S. Trivedi. Hierarchical Composition and Aggregation of State-Based Availability and Performability Models. *IEEE Transitions on Reliability*, vol 52, num 1, pp 44-52, march 2003.

[11] J. Rumbaugh and I. Jacobson and G. Booch, The Unified Modeling Language. Reference Manual, Addison Wesley, ISBN 0-201-30998-X, 1999

[12] K. Jensen. *Coloured Petri Nets, Volume 1: Basic Concepts*. Springer-Verlag, 1992.

[13] K. Jensen. *Coloured Petri Nets, Volume 2: Analysis Methods*. Springer-Verlag. 1994.