Modeling Systems with Mobile Components: A comparison between AltaRica and PEPA nets

Leïla Kloul*[1,2], Tatiana Prosvirnova†[2] and Antoine Rauzy‡[2]

[1]PRiSM, Université de Versailles, 45 Avenue des États-Unis, 78000 Versailles, France
[2]LIX, Ecole Polytechnique, route de Saclay, 91128 Palaiseau, France

Abstract

Assessing the reliability of systems with mobile components, that is components whose locations and interactions change during the mission of the system, raises a number of specific modeling issues. In this article, we compare two candidate modeling formalisms to do so: AltaRica and PEPA nets. We study their respective advantages and drawbacks and we show benefits of a cross fertilization.

Keywords
Model-based safety analysis, modeling formalisms, mobility modeling, PEPA Nets, AltaRica.

1 Introduction

Many industrial systems embed mobile components, that is components whose locations and interactions change during the mission of the system. Systems of systems, like battlefields or mobile phone networks, enter obviously into this category. But mobile components can also be found in simpler systems, such as production chains. Assessing the reliability of systems with mobile components raises a number of specific modeling issues.

As an illustration, we consider in this article a simple plant that produces different types of goods within the same production chain. To calculate the reliability and other performance indicators on this system, one must be able to follow products individually, i.e. to capture dynamic behaviors such as dynamic change of locations of products and dynamic change of interactions between products and processing units.

We compare two candidate modeling formalisms to do so: PEPA nets [12] and AltaRica [7, 20]. These two formalisms have been developed by two different communities. Their "look-and-feel" are thus quite different. Yet, their underlying mathematical foundations are very similar: both rely on state automata and can be used to generate continuous-time Markov chains. It is therefore of interest to study their ability to assess the reliability of systems with mobile components, their respective advantages and drawbacks and to seek for opportunities of a cross fertilization.

PEPA nets combine the stochastic process algebra PEPA (Performance Evaluation Process Algebra [15]) with (stochastic) colored Petri nets [16]. Components are described as processes just as in PEPA, but they can additionally migrate from one place of the net to another, just as tokens in a colored Petri net. Components can interact (through synchronization of their actions) only if they are located in the same place. Operators behind PEPA nets remain simple to implement for dynamic creation/deletion of processes is not allowed. Any PEPA net model can be eventually compiled into a continuous-time Markov chain. The elegance of PEPA nets comes from its mathematical purity: only a very limited number of operators is sufficient to describe complex behaviors.

AltaRica has been designed with an engineering perspective. In AltaRica, the behavior of components is described by means of Guarded Transition Systems [18, 20]. Guarded Transition Systems (GTS)

*Leila.Kloul@prism.uvsq.fr
†Tatiana.Prosvirnova@polytechnique.edu
‡Antoine.Rauzy@lix.polytechnique.fr

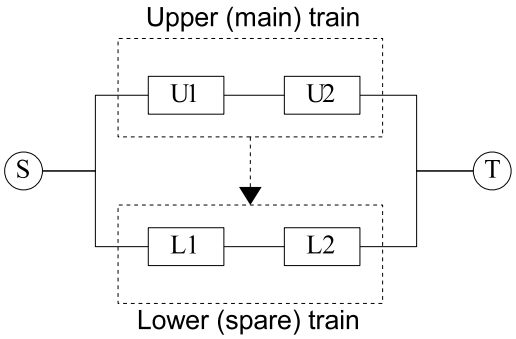

Figure 1: Production system.

generalize widely used formalisms such as Reliability Block Diagrams (see e.g. [4]) and Stochastic Petri nets [3]. Components can be assembled into hierarchies, their inputs and outputs can be connected and their transitions can be synchronized. Any hierarchical description can be "flattened" into a unique GTS. The semantics of a GTS is a Kripke structure (a reachability graph) that can be interpreted as a continuous-time Markov chain, under the condition that delays associated with transitions are exponentially distributed.

The richness of AltaRica makes it possible to design and to maintain industrial scale models [2, 6]. However, the previous versions of AltaRica embed no construct to model mobility. Since PEPA nets and AltaRica rely on similar mathematical foundations, it was worth to establish their respective strengths and weaknesses. This study resulted in the incorporation in the new version of AltaRica (AltaRica 3.0, still under specification) of the concept of guarded synchronization. This new concept unifies and simplifies previous AltaRica description of transitions and synchronizations and thus it eases modeling mobile components.

The contribution of this article is multiple. First, we examine, based on a simple example, the issues raised by the modeling of systems with mobile components. Second, we compare PEPA nets and AltaRica. We discuss their respective advantages and drawbacks. Third, we present the extension of AltaRica with the concept of guarded synchronization.

The remainder of this article is organized as follows. Section 2 presents the production system we shall use as a redwire throughout the article. Section 3 is dedicated to the related works. Section 4 and Section 5 present respectively PEPA nets and AltaRica and illustrate their philosophies by modeling (parts of) the production system. Section 6 compares the two approaches. Section 7 presents some experiences, performed to calculate reliability and performance indicators. Finally, Section 8 concludes the article.

2 Motivating Example

The example described in this section will be used as an illustration in the following sections.

2.1 Production System

We consider a production system made of two chains, as illustrated in Figure 1. The system is supplied by a source unit S. The upper chain, consisting of processing units U1 and U2 in series, is the main chain. The lower chain, consisting of processing units L1 and L2 in series, is a spare chain. The spare chain is normally used for other purposes but products can be rerouted to that chain in case the main chain is not available. The whole system supplies itself a target unit T. Units U1 and L1 on the one hand; U2 and L2 on the other hand play symmetrical roles. When either U1 or U2 fails, the other unit in the same chain is stopped and the lower chain is attempted to start. When both U1 and U2 are restarted (after their repair), the lower chain is stopped (or at least goes back to its primary purpose).

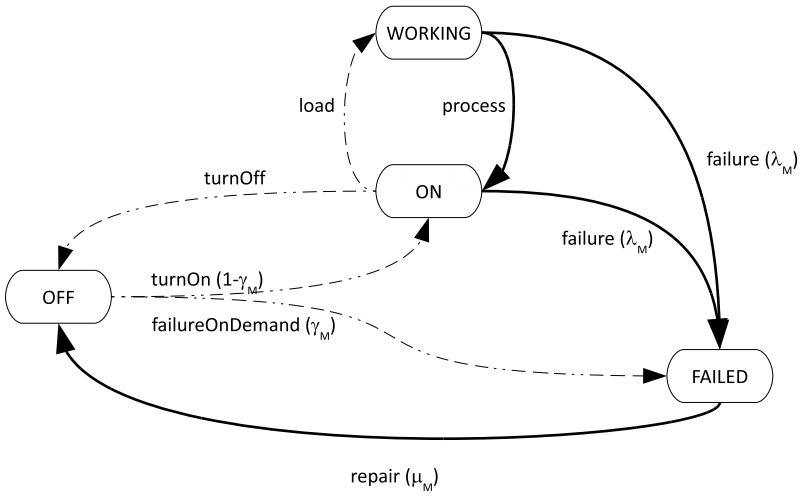

Figure 2: The Finite State Automaton modeling a Machine.

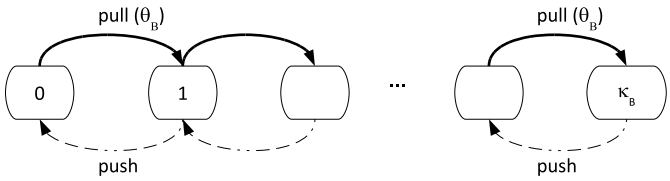

Figure 3: The Finite State Automaton modeling the board (of slots).

2.2 Production Units

For the sake of simplicity, source and target units are assumed to be perfect (and are never stopped). Each processing unit (U1, U2, L1 and L2) is composed of a machine M and a board B with a number κ_B of slots in which products are inserted. Source and target units can be seen simply as boards with slots.

All machines work (and fail) the same way. They can process only one product at a time. A machine M has a per hour failure rate λ_M (e.g. $1.0\ 10^{-3}$) when it is working. It is assumed not to fail when it is in a standby mode. When it is attempted to be turned on, it has a probability to fail on demand γ_M (e.g. 0.02), and thefore a probability $1 - \gamma_M$ to start correctly. It has a per hour repair rate μ_M (e.g. $2.0\ 10^{-1}$). It is assumed to be as good as new after a repair. When a machine fails the product it was processing (if any) needs to be reprocessed from scratch. Machines are not turned off when they are processing a product. The finite state automata modeling a machine is pictured in Figure 2. The machine starts to process a product, i.e. loads it, as soon as there is a non already processed product in a slot. The time taken by the processing of a product depends on the (type of the) product. On this figure, timed transitions are pictured with thick plain lines while instantaneous transitions are pictured with thin dashed lines. We shall keep this convention for subsequent figures.

The finite state automata modeling the board is pictured in Figure 3. For the sake of the simplicity, slots are not distinguished. Two actions can be performed on slots: pulling a product (from the previous unit) and pushing a product (to the next unit). Pulling and pushing a product are actually the two faces of the same action: transferring a product from one production unit to the next one. This action takes some time. We can assume without a loss of generality that the average transfer time t_B depends on the board B that "pulls" the product. A board B has therefore a per hour pulling rate $\theta_B = 1/t_B$ (e.g. $\theta_B = 100$).

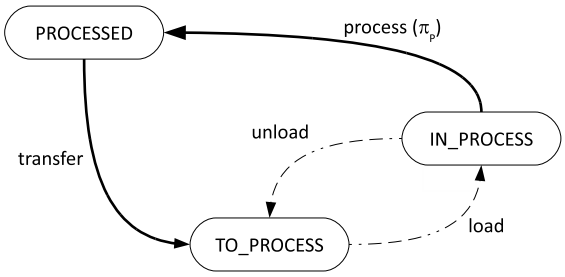

Figure 4: The Finite State Automaton modeling a product.

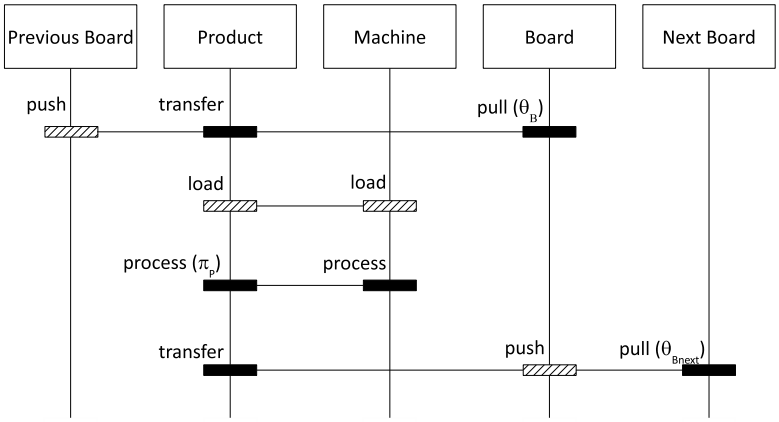

Figure 5: The Sequence Diagram for a successful processing of a product.

2.3 Products

Products are transferred from one unit to the next one. Once in the unit, a product can be either waiting to be processed, in process, or waiting to be transferred to the next unit. The average processing time p_P depends on the product so we have a processing rate $\pi_P = 1/p_P$ (e.g. $\pi_P = 10$). Figure 4 pictures the finite state automaton modeling a product. This automaton does not show the location of the product (which changes with the transition `transfer`).

2.4 Synchronizations/Simultaneity

We described so far individual behaviors of each component of the system. To complete the description, we need to describe which transitions are synchronized.

The sequence diagram pictured in Figure 5 shows synchronizations (horizontal lines) occurring in a successful processing sequence of a product. On this diagram, timed transitions are represented with plain rectangles, instantaneous transitions are represented with hatched rectangles.

The sequence diagram pictured in Figure 6 shows synchronizations occurring in a sequence in which a failure occurs during the processing of a product. Note that timed transitions may be synchronized with instantaneous transitions, e.g. the transition `failure` of the machine and the transition `unload` of the product. In this case, the resulting transition is indeed timed. The instantaneous transition takes place at the end of the timed action.

2.5 Wrap-Up

We want eventually to study the expected production of the system, throughout a given period of time and possibly additional performance indicators such as the mean down time of the main chain.

All the above hypotheses may be not very realistic. We just tried to concentrate into a small example a number of modeling issues:

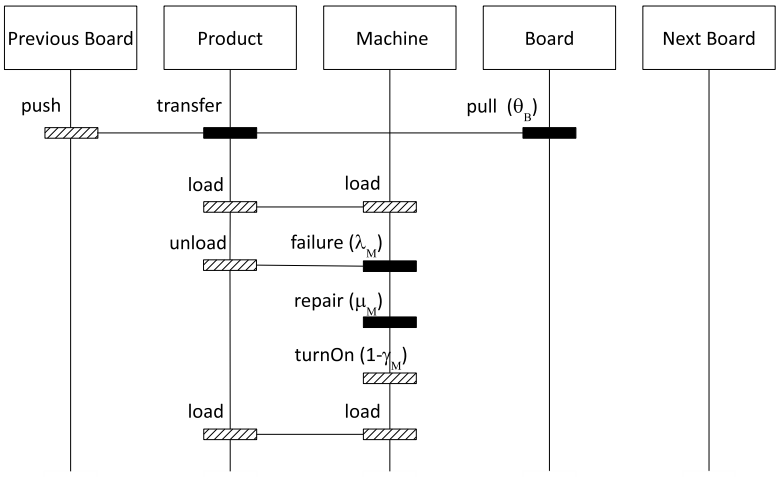

Figure 6: The Sequence Diagram for a sequence with a failure during the processing of a product.

- Products are mobile components. Some parameters, such as the processing rate, depend on products. Moreover, it may be worth to observe the individual trajectories of products.
- Products have to interact with processing units. These interactions can take place only in some locations and circumstances.
- The state of a processing unit depends on the states of other processing units, due to the command strategy of the system.
- Some transitions are instantaneous, some other take time. Timed transitions have rates that differ by orders of magnitude (the model is stiff).

The main modeling issue is to synchronize correctly actions of machines, boards and products and to do that for sufficiently many products while keeping the model tractable.

3 Related Works

Classical safety formalisms, such as Reliability Block Diagrams (RBD), Markov chains and Generalized Stochastic Petri Nets (GSPN) [3] are the most well known and widely used formalisms to assess reliability indicators of systems.

Boolean formalisms, such as RBD, are event based, naturally hierarchical and make it possible to describe remote interactions between components, i.e. flows of matter or information circulating through the system. They can be easily transformed into Fault Trees and assessed with very efficient algorithms (see e.g [19, 21]). However, Boolean formalisms put very strong constraints on events to be considered: they are assumed to be statistically independent, thus, it is not possible to take into account the order in which events occur any time.

States/transitions formalisms, such as Markov chains and Petri Nets, make it possible to capture dependencies amongst components, such as cold redundancies, resource sharing and sequences of actions. But it is quite difficult to use them, on the one hand, to represent remote interactions between components and on the other to compose seamlessly components into hierarchies.

Currently, rare are the modeling techniques that provide modeling mechanisms for systems with dynamic behaviors. Milner's π-calculus [17] is a paradigmatic formalism designed to capture dynamic behaviors, and probably one of the most widely studied. It has been developed to model communicating and mobile agents. It is very simple yet very powerful. However, its ability to create and to delete objects comes with a significant price in terms of assessment algorithms. This cost is so high that it is reasonably arguable whether such powerful formalism can be used for performance analyses. Like the π-calculus, PEPA-nets [8, 11–13] are a simple modeling technique that makes possible the description of migrations of components and of changes in their interactions. However, unlike π-calculus, the operators

Table 1: Comparison of Safety formalisms.

	RBD	Markov chains	GSPN	π-calculus	PEPA Nets	AltaRica DataFlow
Event based	▲	◯	◯	◯	◯	◯
Composition	⊙	■	⊙	⊙	⊙	◯
Hierarchical	◯	■	■	■	■	◯
Remote interactions	◯	■	■	▲	▲	◯
Mobility modeling	■	■	⊙	◯	◯	▲
Algorithm efficiency	◯	⊙	⊙	■	⊙	⊙

■ not suitable ▲ acceptable ⊙ good ◯ very good

and the paradigms behind PEPA nets remain simple to implement for dynamic creation/deletion of objects is not allowed. As all state based modeling techniques, PEPA nets formalism remains prone to the problem of state space explosion. Moreover it does not provide a modeling mechanism to capture remote interactions amongst components.

Petri nets based techniques may be considered candidates to dynamic systems modeling. However, SPN models are constructed without explicit compositional structure regardless of the structure of the system being modeled. Consequently, subsequent techniques, such as Donatelli's Superposed GSPN [10] and Sanders' Stochastic Activity Networks [22] have aimed to provide mechanisms to represent the increasing complexity of the synchronization constraints of modern systems whilst retaining the compositional structure explicitly within the model. However, Petri nets based techniques do not provide an appropriate mechanism to capture dynamic change of interactions between objects as they do not allow the distinction between different contexts.

An extension of (non-stochastic) Petri nets which provides modeling concepts similar to PEPA nets is Valk's Elementary Object Systems (EOS) [23]. The tokens in an elementary object system are themselves Petri nets having individual dynamic behavior. However, like all Petri nets based formalism, EOS formalism suffers from a lack of an explicit compositional structure.

AltaRica DataFlow modeling language generalizes classical safety formalisms: it is a states/transitions formalism that allows hierarchical structuring of models and representing remote interactions. Nevertheless it would be quite difficult, from the practical modeling perspective, to use states/transitions formalisms to describe systems with mobile components, as presented in the example Section 2.

Table 1 summarizes this section.

4 Overview of PEPA nets

PEPA nets [12] combine the stochastic process algebra PEPA (Performance Evaluation Process Algebra) with stochastic colored Petri nets. This hybrid formalism is motivated by the observation that, in many systems, two distinct types of change of state can be identified: the *global* and *local* changes of states. The resulting formalism can be used to model applications such as mobile code systems where the PEPA terms are used to model the program code moving between the network hosts (the places in the net).

In the following, we first give an overview of the modeling language PEPA, then present the hybrid formalism PEPA nets.

4.1 PEPA

In PEPA [15], a system is described as an interaction of *components* which engage, either singly or multiply, in *activities*. These activities represent changes of state within a system. Each activity has an *action type* and a *duration* which is represented by the parameter of the associated exponential distribution: the *activity rate*. This parameter may be any positive real number, or the distinguished symbol $\top$ (read as *unspecified*). Thus each activity is a pair (α, r) where α is the action type and r is the activity rate. We assume a countable set of components, denoted $\mathcal{C}$, and a countable set, $\mathcal{Y}$, of all

possible action types. We denote by $\mathcal{A}ct \subseteq \mathcal{Y} \times \mathbb{R}^+$, the set of activities, where $\mathbb{R}^+$ is the set of positive real numbers together with the symbol $\top$.

PEPA provides a small but expressive set of combinators which allow expressions to be constructed defining the behavior of components, via the activities they undertake and the interactions between them.

Prefix $(\alpha, r).P$: this is the basic mechanism for constructing component behaviors. The component carries out activity (α, r) and subsequently behaves as component P.

Choice $P + Q$: the component may behave either as P or as Q: all the current activities of both components are enabled. The first activity to complete, determined by the race condition, distinguishes one component, the other is discarded.

Cooperation $P \underset{L}{\bowtie} Q$: the components proceed independently with any activities whose types do not occur in the *cooperation set* L (*individual activities*). However, activities with action types in the set L require the simultaneous involvement of both components (*shared activities*). When the set L is empty, we use the more concise notation $P \parallel Q$ to represent $P \underset{\emptyset}{\bowtie} Q$.

The published stochastic process algebras differ on how the rate of shared activities are defined [14]. In PEPA the shared activity occurs at the rate of the slowest participant. If an activity has an unspecified rate, denoted $\top$, the component is *passive* with respect to that action type. This means that the component does not influence the rate at which any shared activity occurs.

Hiding P/L: the component behaves as P except that any activities of types within the set L are *hidden*, i.e. they exhibit the unknown type τ and can be regarded as an internal delay by the component. These activities cannot be carried out in cooperation with another component.

Constant $A \stackrel{def}{=} P$: Constants are components whose meaning is given by a defining equation. $A \stackrel{def}{=} P$ gives the constant A the behavior of component P. This is how we assign names to components (behaviors).

The evolution of a PEPA model is governed by the Structured Operational Semantics (SOS) rules of the language. These rules define the admissible transitions or state changes associated with each combinator. They give rise to a multi-labeled transition system or derivation graph from which a continuous-time Markov chain can be derived.

Example: Consider the upper train of the production system described in Section 2. If we want to model the first processing unit, U_1, one can use two PEPA components, namely $Machine_1$ and $Product$. The first component models the behavior of the machine in the processing unit, whereas the second one models the items provided by the source, in order to be processed by U_1.

- Component $Machine_1$: when the processing unit is working properly, it first loads a new item if this one is available. This is modeled using action type $load_1$ which rate l_1 is supposed to be high as the action of loading is assumed to be instantaneous. Once the item loaded, three different events may occur: the processing of the item, a failure of the machine, or the arrival of an order for the machine to be switched off because U_2, the other processing unit in the main train, is in the failure state. The three events are modeled using action types $process_1$, $failure_1$ and $turnOff_1$, respectively. In the failure state $Machine_{1\_FAILED}$, the machine can be either repaired (action type $repair_1$), or receive a $turn\ off$ order that will not make the component change state. When repaired, the machine is stopped (state $Machine_{1\_OFF}$) until a switch on order is received. This is modeled using activity $(turnOn_1, 1 - \gamma_{u_1})$, whereas the occurrence of a failure on demand is modeled using activity $(failureOnDemand_1, \gamma_{u_1})$. The initial state of the component is $Machine_{1\_ON}$. Thus,

the complete component is defined as follows:

$$
\begin{aligned}
Machine_{1\_ON} &\stackrel{def}{=} (load, l_1).Machine_{1\_WORKING} \\
&+ (failure_1, \lambda_{u_1}).Machine_{1\_FAILED} \\
&+ (turnOff_1, s_1).Machine_{1\_OFF} \\
Machine_{1\_WORKING} &\stackrel{def}{=} (process_1, \pi_p).Machine_{1\_ON} \\
&+ (failure_1, \lambda_{u_1}).Machine_{1\_FAILED} \\
Machine_{1\_FAILED} &\stackrel{def}{=} (repair_1, \mu_{u_1}).Machine_{1\_OFF} \\
&+ (turnOff_1, s_2).Machine_{1\_FAILED} \\
Machine_{1\_OFF} &\stackrel{def}{=} (turnOn_1, s_2 \times (1 - \gamma_{u_1})).Machine_{1\_ON} \\
&+ (failureOnDemand_1, s_0 \times \gamma_{u_1}).Machine_{1\_FAILED}
\end{aligned}
$$

As loading a product to be processed is assumed to be an instantaneous action and does not really make the machine change state, we do not consider it in the PEPA model. However, we have to take into account the instantaneous events *failure on demand*, *turn off* and *turn on*, because they make the machine change states. For that we suppose that these actions occur at the rates s_0, s_1 and s_2, respectively.

- Component *Product*: it can be defined by the loading and processing undertaken in unit U_1 as follows:

$$
\begin{aligned}
Product &\stackrel{def}{=} (load_1, \top).Product' \\
Product' &\stackrel{def}{=} (process_1, \top).Product
\end{aligned}
$$

  In this component the rates associated with action types $load_1$ and $process_1$ are unspecified; component $Machine_1$ specifies the rate at which the loading and the processing occur.

- The behavior of the complete processing unit U_1 is modeled using the PEPA equation which specifies that components $Machine_1$ and *Product* must cooperate (synchronize) on action types $load_1$ and $process_1$.

$$U_1 \stackrel{def}{=} Machine_{1\_ON} \underset{\{load_1, process_1\}}{\bowtie} Product$$

4.2 PEPA nets

As PEPA nets combine PEPA with colored stochastic Petri nets, two types of change of state are possible: the *transitions* of PEPA components and the *firings* of the net. Transitions of PEPA components will typically be used to model small-scale (local) changes of state as components undertake activities. Firings of the net will be used to model macro-step (global) changes of state such as context switches or mobile software agents moving from one network host to another.

A PEPA net is made up of PEPA *contexts*, one at each place in the net. A context consists of a number of *static* components (possibly zero) and a number of *cells* (at least one). A cell is a storage area dedicated to storing a PEPA component of the specified type. The components which fill cells are the *mobile* components and can circulate as the *tokens* of the net. In contrast, the static components cannot move.

The mobile components or tokens of a PEPA net are terms of the PEPA stochastic process algebra which define the behavior of components via the activities they undertake and the interactions between them. Thus each token has a type given by its definition. This type determines the transitions and firings which a token can engage in. It also restricts the places in which it may be, since it may only enter a cell of the corresponding type.

As the firings, on the one hand, and the transitions, on the other hand, are special cases of PEPA activities, we differentiate the action types associated with each of these. We denote by $\mathcal{Y}_f$ the set of action types at the net level and by $\mathcal{Y}_t$ the set of action types inside the places such that $\mathcal{Y} = \mathcal{Y}_f \cup \mathcal{Y}_t$. Similarly, we denote by $\mathcal{A}ct_t \subseteq \mathcal{Y}_t \times \mathbb{R}^+$ the set of activities undertaken by the components inside the places and by $\mathcal{A}ct_f \subseteq \mathcal{Y}_f \times \mathbb{R}^+$ the set of activities at the net level such that $\mathcal{A}ct = \mathcal{A}ct_f \cup \mathcal{A}ct_t$.

Definition 4.1 *A PEPA net $\mathcal{V}$ is a tuple $\mathcal{V} = (\mathcal{P}, \mathcal{T}, I, O, \ell, \pi, \mathcal{F}_P, K, M_0)$ such that*

- *$\mathcal{P}$ is a finite set of places;*
- *$\mathcal{T}$ is a finite set of net transitions;*
- *$I : \mathcal{T} \to \mathcal{P}$ is the input function;*
- *$O : \mathcal{T} \to \mathcal{P}$ is the output function;*
- *$\ell : \mathcal{T} \to (\mathcal{Y}_f, \mathbb{R}^+ \cup \{\top\})$ is the labeling function, which assigns a PEPA activity ((type, rate) pair) to each transition. The rate determines the negative exponential distribution governing the delay associated with the transition;*
- *$\pi : \mathcal{Y}_f \to \mathbb{N}$ is the priority function which assigns priorities (represented by natural numbers) to firing action types;*
- *$\mathcal{F}_P : \mathcal{P} \to P$ is the place definition function which assigns a PEPA context, containing at least one cell, to each place;*
- *K is the set of token component definitions;*
- *M_0 is the initial marking of the net.*

PEPA net behavior is governed by structured operational semantic rules. These consist of the original rules for PEPA and some additional rules capturing the meaning of a cell, as well as the enabling and firing rules of the net level structure [12]. The states of the model are the marking vectors, which have one entry for each place of the PEPA net. The semantic rules govern the possible evolution of a state, giving rise to a labeled transition system or derivation graph. The nodes of the graph are the marking vectors and the activities (individual, shared or firing activities) give the arcs of the graph. This graph gives rise to a CTMC which can be solved to obtain a steady-state probability distribution from which performance measures can be derived.

The syntax of PEPA nets is given in Figure 7.

$$N \quad ::= \quad D^+ M \qquad \text{(net)}$$

(definitions and marking)

$$M \quad ::= \quad (M_{\mathbf{P}}, \ldots) \qquad \text{(marking)}$$

$$M_{\mathbf{P}} \quad ::= \quad \mathbf{P}[C, \ldots] \qquad \text{(place marking)}$$

(marking vectors)

$$D \quad ::= \quad I \stackrel{def}{=} S \qquad \text{(component defn)}$$

$$\mid \quad \mathbf{P}[C] \stackrel{def}{=} P[C] \qquad \text{(place defn)}$$

$$\mid \quad \mathbf{P}[C, \ldots] \stackrel{def}{=} P[C] \underset{L}{\bowtie} P \qquad \text{(place defn)}$$

(identifier declarations)

$$S \quad ::= \quad (\alpha, r).S \qquad \text{(prefix)}$$

$$\mid \quad S + S \qquad \text{(choice)}$$

$$\mid \quad I \qquad \text{(identifier)}$$

(sequential components)

$$P \quad ::= \quad P \underset{L}{\bowtie} P \qquad \text{(cooperation)}$$

$$\mid \quad P/L \qquad \text{(hiding)}$$

$$\mid \quad P[C] \qquad \text{(cell)}$$

$$\mid \quad I \qquad \text{(identifier)}$$

(concurrent components)

$$C \quad ::= \quad \text{`\_'} \qquad \text{(empty)}$$

$$\mid \quad S \qquad \text{(full)}$$

(cell term expressions)

Figure 7: The syntax of PEPA nets.

In the grammar S denotes a *sequential component* and P denotes a *concurrent component* which executes in parallel. I stands for a constant which denotes either a sequential or a concurrent component, as bound by definition.

Example: Consider the sub-system composed of source S, processing units U_1 and U_2, and the target unit T of the production system described in Section 2. The PEPA net model of this sub-system consists of four places: $SOURCE$, $MAIN\_UNIT_1$, $MAIN\_UNIT_2$ and $TARGET$. The net structure of the model is depicted in Figure 8.

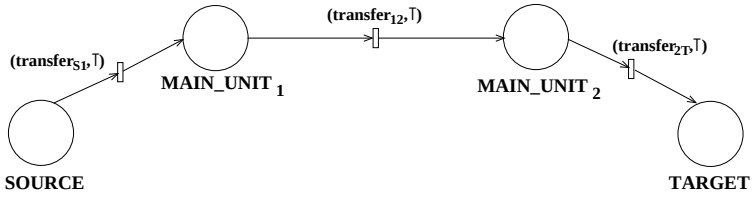

Figure 8: The PEPA net model.

- Place $SOURCE$: it models the source which supplies unit U_1 with the items to process. Thus we model each of these items using component $Product$ (see Section 4.1) which now is a mobile component since each item has to go through all the units of the system to be completed. Thus the definition of component $Product$ is enriched with activities that model the movements of the component between the places. These activities label the firing transitions on the net structure (Figure 8).

$$\begin{aligned}
Product &\stackrel{def}{=} (\mathbf{transfer_{S1}}, \top).Product_1 \\
Product_1 &\stackrel{def}{=} (load_1, \top).Product'_1 \\
Product'_1 &\stackrel{def}{=} (process_1, \top).Product_2 \\
Product_2 &\stackrel{def}{=} (\mathbf{transfer_{12}}, \top).Product_3 \\
Product_3 &\stackrel{def}{=} (load_2, \top).Product'_3 \\
Product'_3 &\stackrel{def}{=} (process_2, \top).Product_4 \\
Product_4 &\stackrel{def}{=} (\mathbf{transfer_{2T}}, \top).Product
\end{aligned}$$

  Place $SOURCE$ consists solely of the items to provide to unit U_1. Thus initially all the items are in this place.

$$SOURCE[\_, \ldots, \_] \stackrel{def}{=} Product[Product] || \ldots || Product[Product]$$

- Place $MAIN\_UNIT_1$: it models the processing unit U_1. The behavior of the corresponding processing machine is modeled using static component $Machine_1$ (see Section 4.1).

  The whole place is then modeled as the interaction of $Machine_1$ and mobile component $Product$ on action type $process_1$. The maximum number of mobile components in the place corresponds to the storage capacity of unit U_1, that is κ_{U_1}.

$$MAIN\_UNIT_1[\_, \ldots, \_] \stackrel{def}{=} Machine_{1\_ON} \underset{\{process_1\}}{\bowtie} (Product[\_] || \ldots || Product[\_])$$

- Place $MAIN\_UNIT_2$: it models the behavior of processing unit U_2 which has the same behavior as U_1. Thus we use a similar component, namely $Machine_2$, to model the processing machine in

U_2.

$$
\begin{array}{lcl}
Machine_{2\_ON} & \stackrel{def}{=} & (load_2, l_2).Machine_{2\_WORKING} \\
& + & (failure_2, \lambda_{u_2}).Machine_{2\_FAILED} \\
& + & (turnOff_2, s'_1).Machine_{2\_OFF} \\
Machine_{2\_WORKING} & \stackrel{def}{=} & (process_2, \pi_p).Machine_{2\_ON} \\
& + & (failure_2, \lambda_{u_2}).Machine_{2\_FAILED} \\
Machine_{2\_FAILED} & \stackrel{def}{=} & (repair_2, \mu_{u_2}).Machine_{2\_OFF} \\
& + & (turnOff_2, s'_2).Machine_{2\_FAILED} \\
Machine_{2\_OFF} & \stackrel{def}{=} & (turnOn_2, s'_2 \times (1 - \gamma_{u_2})).Machine_{2\_ON} \\
& + & (failureOnDemand_2, s'_0 \times \gamma_{u_2}).Machine_{2\_FAILED}
\end{array}
$$

The complete place is then modeled as the cooperation of $Machine_{2\_ON}$ and $Product$ on action types $load_2$ and $process_2$.

$$MAIN\_UNIT_2[\_, \ldots, \_] \stackrel{def}{=} Machine_{2\_ON} \underset{\{load_2, process_2\}}{\bowtie} (Product[\_] || \ldots || Product[\_])$$

Similarly to $MAIN\_UNIT_1$, the maximum number of components $Product$ in $MAIN\_UNIT_2$ is κ_{U_2}, the storage capacity of unit U_2.

- Place $TARGET$: it models the target unit and consists solely of the finished items arriving from unit U_2. Initially, it is empty.

$$TARGET[\_, \ldots, \_] \stackrel{def}{=} Product[\_] || \ldots || Product[\_]$$

Note that the maximum number of components $Product$ in places $SOURCE$ and $TARGET$ is defined by storage capacity κ_S and κ_T, respectively.

5 AltaRica Overview

AltaRica is a high level modeling language initially dedicated to safety analysis. The first version of AltaRica modeling language has been developed in LaBRI in ninetieth [5]. A few years later, a second (data-flow) version has been developed to handle industrial scale models that the first version, too expressive, was inefficient to tackle. A number of processing tools have been developed for AltaRica such as compilers to Fault Trees, compilers to Markov chains, generators of critical sequences, model-checkers and stochastic simulators. Several Integrated Modeling Environments use AltaRica as their internal representation language.

The third version (AltaRica 3.0) is under specification at the time we write these lines. AltaRica 3.0 will be a major evolution of the language (and the processing tools). This new version integrates notions of object-oriented programming languages such as inheritance and prototypes. It improves the reusability of components and knowledge capitalization. It adds also the ability to handle looped systems. The models presented below are written in AltaRica 3.0. The formal semantics of AltaRica 3.0 is based on the notion of Guarded Transition Systems - a states/events formalism defined in Reference [20].

5.1 Guarded Transition Systems

Guarded Transition Systems, GTS for short, are input/output automata. The state space is described implicitly as, for instance, in Petri nets. We shall introduce here GTS by means of an example. Consider first the automaton for the machine pictured Figure 2. The AltaRica code for this automaton is given Figure 9.

```
domain MachineState { OFF, ON, WORKING, FAILED }

class Machine
  MachineState state (init = OFF);
  Boolean demanded (reset = false);
  event turnOn (delay = 0, expectation = 1 - gamma);
  event failureOnDemand (delay = 0, expectation = gamma);
  event failure (delay = exponential(lambda));
  event repair (delay = exponential(mu));
  event turnOff (delay = 0);
  event load (delay = 0);
  event process;
  parameter Real gamma = 0.02;
  parameter Real lambda = 0.001;
  parameter Real mu = 0.1;
transition
  turnOn:  state==OFF and demanded -> state := ON;
  failureOnDemand:  state==OFF and demanded -> state := FAILED;
  turnOff:  state==ON and not demanded -> state := OFF;
  failure:  state==ON or state==WORKING -> state := FAILED;
  repair:  state==FAILED -> state := OFF;
  load:  state==ON -> state := WORKING;
  process:  state==WORKING -> state := ON;
end
```

Figure 9: The AltaRica code for the Finite State Automaton modeling a machine.

Variables: The internal state of the machine is represented by means of the state variable `state`. `state` takes its value in the domain `MachineState` declared upfront. The initial values of state variables (there may be several) are specified by means of the attribute `init`.

Another variable is declared: `demanded`. This variable is a Boolean flow variable. It is used to implement the command, i.e. to tell when to turn on and off the machine. Values of flow variables are reset after each transition firing, then updated by means of an assertion. This mechanism will be described latter. From a syntaxic viewpoint, flow variables are introduced (and distinguished from state variables) by means of the attribute `reset`.

Events: The state of the machine changes under the occurrence of an event. Events are introduced with the keyword `event`. A delay is associated with each event by means of the attribute `delay`. In our example, delays of events `failure` and `repair` are random variables exponentially distributed with respective rates `lambda` and `mu`. In other words, they obey a Markovian hypothesis. Events `turnOn` and `failureOnDemand` are instantaneous (their delay is 0). Both are fireable when the machine is `OFF`. `turnOn` has the probability `1 - gamma` to be fired while `failureOnDemand` has a probability `gamma` to be fired in this state. This probability is given through the attribute `expectation`. Delays of events `load` and `process` are left unspecified.

Transitions: A transition is a triple $\langle e, G, P\rangle$, also denoted $e : G \to P$, where e is an event, G is a Boolean expression, so-called the guard (or the pre-condition) of the transition, P is an instruction, so-called the action (or the post-condition) of the transition. Transitions are described in the clause `transitions`. In the example above if the state of the processing machine is `WORKING`, then two transitions are fireable: the transition labeled with the event `failure` and the transition labeled with the event `process`. If the delay drawn for the transition `failure` is the shortest, then this transition is fired and its action is executed: `state` is switched to `FAILED`.

Parameters: Parameters are constant values that come with the definition of the GTS. When the GTS is instantiated, their values may be changed. In the above example, there are three parameters `gamma`, `lambda` and `mu` that define respectively the probability of failure on demand and the failure and repair rates.

5.2 Composition and Synchronization

The AltaRica code for the automaton describing the board (as pictured Figure 3) is given Figure 10. This code deserves no additional explanation.

```
class Board
  Integer count (init = 0);
  event pull (delay = exponential(theta));
  event push (delay = 0);
  parameter Integer capacity = 3;
  parameter Real theta = 60;
transition
  pull:  count<capacity -> count := count+1;
  push:  count>0 -> count := count-1;
end
```

Figure 10: The AltaRica code for the Finite State Automaton modeling the board.

Now we can consider the model for a processing unit. AltaRica 3.0 is an object oriented language. Therefore, the AltaRica class that describes a processing unit embeds an instance of the class describing the machine and an instance of the class describing the board, as illustrated Figure 11.

```
class ProcessingUnit
  Machine M;
  Board B;
transition
  M.load:  !M.load & B.count>0 -> skip;
end
```

Figure 11: The AltaRica code for a Processing Unit.

This combination is however not a mere product: the machine cannot load a product if the board is empty. This additional constraint is described by means of a synchronization. The transition `load` of the machine `M` (`M.load`) is synchronized with an anonymous transition that just checks that `B.count` is positive and does nothing (its action is the empty instruction `skip`). This synchronization creates a new transition. This transition is obtained by and-ing the guards of synchronized transitions and composing their actions. In our case, the synchronization creates the following transition.

```
M.state==ON and B.count>0 -> M.state := WORKING;
```

A synchronization can involve any number of transitions. Transitions involved in a synchronization cease to exist individually. It is the case here for the transition `M.load`. Since we don't want to create a fresh event for the created transition, we use the event `M.load`. Note finally that `M.load` is prefixed with an exclamation mark (`!`). This modality indicates that the individual transition `M.load` is mandatory for the synchronized transition to be fired. Anonymous transitions are always mandatory. The modality `?` makes it possible to synchronize transitions only when they are fireable. We won't describe it here fully for we shall not use it in our model.

5.3 Flow Variables and Assertions

In the AltaRica code for the machine, given Figure 9, the flow variable `demanded` is used to guard the instantaneous transitions `turnOn` and `turnOff`. Conversely to state variables, that are initialized at the beginning of a run and then modified through actions of transitions, the value of flow variables are recalculated after each transition firing. This recalculation is performed by means of assertions. Assertions are instructions just as actions of transitions. The difference stands in that actions of transitions assign state variables only while assertions assign flow variables only. Moreover, each component has a unique assertion that is applied after each transition firing.

Most of AltaRica models make a great use of flow variables and assertions. They are used to model information flows circulating through a system. They may represent physical connections between components, control commands, fluid circulation, electric power, etc. They offer an easy and elegant way to express dependencies on external factors.

In our example, we shall use them to a limited extent, in order to implement the command strategy. The AltaRica code for the plant is pictured Figure 12. This code composes four processing units. When one of the two main units fails, the other one must be stopped (possibly after finishing to process a product) and the production must be switched to the spare line. Conversely, both units of the main line are attempted to start as soon as they are `OFF`, i.e. after a repair.

5.4 Mobile Components

It remains now to model products and to synchronize them with the plant. The AltaRica code for products is given Figure 13. It implements the automaton pictured Figure 4.

Now, we have to synchronize the plant with a number of products. Figure 14 shows a part of the code to do so. This code uses the same mechanism of synchronization as previously, except that more transitions are involved in synchronizations.

```
class Plant
  Board S, T;
  ProcessingUnit U1, U2, L1, L2;
  Boolean mainLineFailed, spareLineFailed (reset = false);
assertion
  mainLineFailed := U1.M.state==FAILED or U2.M.state==FAILED;
  spareLineFailed := L1.M.state==FAILED or L2.M.state==FAILED;
  U1.M.demanded := not mainLineFailed;
  U2.M.demanded := U1.M.demanded;
  L1.M.demanded := mainLineFailed and not spareLineFailed;
  L2.M.demanded := L1.M.demanded;
end
```

Figure 12: The AltaRica code for the Plant.

```
domain ProductState { PROCESSED, TO_PROCESS, IN_PROCESS }
domain Location { SOURCE, UNIT_U1, UNIT_U2, UNIT_L1, UNIT_L2, TARGET }

class Product
  ProductState state (init = PROCESSED);
  Location location (init = SOURCE);
  event transfer;
  event load (delay = 0);
  event unload (delay = 0);
  event process (delay = exponential(pi));
  parameter Real pi = 10;
transition
  transfer:  state==PROCESSED -> state := TO_PROCESS;
  load:  state==TO_PROCESS -> state := IN_PROCESS;
  unload:  state==IN_PROCESS -> state := TO_PROCESS;
  process:  state==IN_PROCESS -> state := PROCESSED;
end
```

Figure 13: The AltaRica code for the Products.

```
class System
  Plant plant;
  Product p1; Product p2; Product p3; ...
transition
  ⋮
  plant.U1.B.pull:
    !plant.U1.B.pull & !p3.transfer & p3.location==SOURCE -> p3.location := UNIT_U1;
  plant.U1.M.load:
    !plant.U1.M.load & !p3.load & p3.location==UNIT_U1 -> skip;
  plant.U1.M.process:
    !plant.U1.M.process & !p3.process & p3.location==UNIT_U1 -> skip;
  plant.U1.M.failure:
    !plant.U1.M.failure & !p3.unload & p3.location==UNIT_U1 -> skip;
  ⋮
end
```

Figure 14: The AltaRica code for the System.

6 Comparison of two approaches

Both PEPA nets and AltaRica rely heavily on state automata, but are quite different in the way they represent them. To some extent, whether to use a Process Algebra style or a Guarded Transition Systems style is a matter of taste. Guarded Transition Systems are probably more powerful and more compact, thanks to the use of state variables. Aside the way automata are encoded, there are two main differences between the two formalisms:

- AltaRica embeds the concept of flow variables. Flow variables (and assertion) make it possible to describe remote interactions between components. Modeling such interactions with PEPA nets is more complex, although possible.
- PEPA nets provides a mechanism to describe component locations, while such a mechanism has to be modeled in AltaRica.

In the remainder of this section, we shall examine both issues.

6.1 Modeling Remote Interactions between Components in PEPA nets

In PEPA nets, there is no direct means to express behavioral dependencies between components located in different places for components can cooperate or synchronize only if they are in the same place. Thus the notion of flows does not exist as a such. However, it is always possible to model systems with flows. To do so we have to introduce in the model mobile components which are, a priori, unnecessary, but which allow us to express remote interactions between components. The following example is a good illustration of that.

Example: Consider the system used in the example of Section 4.2. In order to complete modeling the upper train of the example, actions $turnOff$ and $turnOn$ have to be synchronizing actions. Indeed both machine units $U1$ and $U2$ must be stopped if one of them fails and both must be restarted if the failure is repaired. Thus, we assume that the production system has a control center in charge of generating the stoppage/restart orders. When a failure occurs at the upper train, the control center has to send a stoppage signal to the working unit in the main train, and a start signal to the units in the spare train. Once the failed unit is repaired, the control center has to stop the spare train while starting the upper train again.

The net structure of the corresponding PEPA net model is depicted in Figure 15. This structure consists of the net structure in Figure 8 to which a new place, namely $CENTER$, has been added. This place models the control center of the production system.

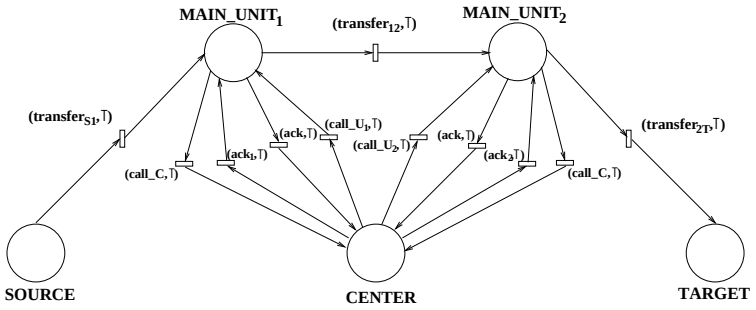

Figure 15: The net structure of the PEPA net model.

In the new PEPA net model, the definitions of places $SOURCE$ and $TARGET$ remain unchanged. However, the definitions of places $MAIN\_UNIT_1$ and $MAIN\_UNIT_2$ have to be changed in order to take into account their interactions with the control center. In the case of the interaction between $MAIN\_UNIT_1$ and $CENTER$, we use two mobile components, $Signal\_1$ and $Signal\_C$. The role of the former is to inform the control center about the state of the machine in $U1$ (failed, repaired). The reception of this information is then acknowledged to $U1$, using the same mobile component ($Signal\_1$). The latter is sent by the control center to $U1$ in order to stop/start its machine in the case of the failure/repair of the machine in $U2$. Thus $MAIN\_UNIT_1$ is defined as follows.

$$\begin{array}{rcl} MAIN\_UNIT_1[\_,\ldots,\_] & \stackrel{def}{=} & ((Signal\_C[\_]||Signal\_1[Signal\_1]) \underset{L}{\bowtie} Machine_{1\_ON}) \\ & \underset{\{process_1\}}{\bowtie} & (Product[\_]||\ldots||Product[\_]) \end{array}$$

where $L = \{turnOn_1, turnOff_1, failureOnDemand_1\}$ and mobile component $Signal\_1$ has the following behavior.

$$\begin{array}{rcl} Signal\_1 & \stackrel{def}{=} & (failure_1, \top).Signal\_1_1 \\ Signal\_1_1 & \stackrel{def}{=} & (\mathbf{call\_C}, e_1).Signal\_1_2 \\ Signal\_1_2 & \stackrel{def}{=} & (failed_1, e_2).Signal\_1_3 \\ Signal\_1_3 & \stackrel{def}{=} & (\mathbf{ack_1}, e_4).Signal\_1_4 \\ Signal\_1_4 & \stackrel{def}{=} & (repaire_1, \top).Signal\_1_5 \\ Signal\_1_5 & \stackrel{def}{=} & (\mathbf{call\_C}, e_1).Signal\_1_6 \\ Signal\_1_6 & \stackrel{def}{=} & (repaired_1, e_2).Signal\_1_7 \\ Signal\_1_7 & \stackrel{def}{=} & (\mathbf{ack_1}, e_4).Signal\_1 \end{array}$$

As specified in the equation of $MAIN\_UNIT_1$, component $Signal\_1$ is initially located in place $MAIN\_UNIT_1$. The behavior of component $Signal\_C$, which is initially located in place $CENTER$, is defined as follows:

$$\begin{array}{rcl} Signal\_C & \stackrel{def}{=} & (failed_2, \top).Signal\_C_1 + (repaired_2, \top).Signal\_C_1 \\ Signal\_C_1 & \stackrel{def}{=} & (\mathbf{call\_U_1}, e_1).Signal\_C_2 \\ Signal\_C_2 & \stackrel{def}{=} & (turnOff, e_2).Signal\_C_3 + (turnOn, e_3).Signal\_C_3 \\ Signal\_C_3 & \stackrel{def}{=} & (\mathbf{ack}, e_4).Signal\_C \end{array}$$

Similarly place $MAIN\_UNIT_2$ is changed in order to include two mobile components, namely $Signal\_2$ and $Signal'\_C$. These components are similar to $Signal\_1$ and $Signal\_C$, respectively. Thus, place $CENTER$ can be modeled as the interaction between the four mobiles components as follows:

$$CENTER[\_,\ldots,\_] \stackrel{def}{=} (Signal\_C[\ Signal\_C\ ] \ ||\ Signal'\_C[\ Signal'\_C\ ]) \underset{M}{\bowtie} (Signal\_1[\ \_\ ] \ ||\ Signal\_2[\ \_\ ])$$

Table 2: Comparison of Safety formalisms: AltaRica 3.0.

	PEPA Nets	AltaRica DataFlow	AltaRica 3.0
Event based	◯	◯	◯
Composition	⊙	◯	◯
Hierarchical	■	◯	◯
Remote interactions	▲	◯	◯
Mobility modeling	◯	▲	⊙
Algorithm efficiency	⊙	⊙	⊙

■ not suitable ▲ acceptable ⊙ good ◯ very good

where cooperation set $M = \{failed_1, repaired_1, failed_2, repaired_2\}$.

This example shows that it is always possible de model remote interactions between components located in different places. However, it comes at a certain price as it requires the use of additional components, which leads to the increase of the model size.

6.2 Modeling Mobility with AltaRica

AltaRica provides no specific construct to model mobility. The location of a component can be modeled as symbolic state variable. In the code presented Figure 13, the type of this variable is an user declared domain. It would be also possible to declare it just as `Symbol`, the set of all symbolic constants. In this way, the topography of the underlying network could be changed without changing the code for products.

PEPA nets synchronize events on their names, so that many components can be synchronized by means of a single rule. AltaRica requires to write down each synchronization explicitly, as sketched Figure 14. Writing all synchronizations for all products by hand would be both tedious and error prone, even if the concept of guarded synchronization, introduced in AltaRica thanks to the present study, simplifies greatly the task. We are presently using scripts (typically written in Python or Pearl) to generate automatically synchronizations. In the future, some specific constructs or some meta-modeling facilities should be added to the language in order to avoid to use external tools.

Table 2 summarizes this section.

7 Experiments

We have performed some experiments with AltaRica and PEPA Nets models of the motivating example.

A continuous time Markov chain (CTMC) has been generated from the AltaRica model. This generation is done in the following way. First, the AltaRica model is flattened into a unique Guarded Transition System [18, 20]. Second, the corresponding reachability graph is generated. Indeed, the semantics of a Guarded Transition System is 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 with the events are exponentially distributed, than the reachability graph can be interpreted as a continuous time Markov chain. In case when the graph contains immediate transitions (delays associated with labeling events are equal to 0), 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 $p\lambda$.

Similarly, we have generated a continuous time Markov chain from the PEPA nets model, using the PEPA Workbench for PEPA nets models [1]. The semantic rules governing the possible evolution of a state, give rise to a multi-labelled transition system or derivation graph. The nodes of the graph are the marking vectors and the activities (individual, shared or firing activities) give the arcs of the graph. This graph gives rise to a CTMC.

Mobile products	AltaRica		PEPA Nets	
	MC states number	Running time	MC states number	Running time
1	155	0.01 sec.	982936	1159 sec.
2	2473	0.62 sec.	-	-
3	37379	257 sec.	-	-
4	-	-	-	-

Table 3: Experiments.

Mobile products	Time	Reliability	Availability
1	24h	0.996656	0.999616
2	24h	0.992918	0.999156
3	24h	0.988702	0.998616

Table 4: Availability and Reliability.

As it is shown in Table 3, the size of the generated Markov chains grows exponentially with the number of mobile products, and this in both cases. However, the problem of exponential growth is more striking in the case of PEPA nets models. Indeed for a model containing only one mobile product, the generated Markov chain has almost one million states. For more than one product, the PEPA Workbench just could not generate the associated Markov chain. This is due to the fact that flows cannot be represented implicitly with PEPA nets; they have to be explicitly modeled using additional components. In our model, four components, with eight, five, eight and nine states respectively, have been added into the model, in order to take into account the flows in the system. Moreover, unlike in AltaRica model, we had to model explicitly the control center by adding a place in the model, and consequently increasing the model size.

The generation of Markov chains seems to be hardly usable in case of such a complex model. It might be promising to generate an approximated Markov chain as proposed in [9].

The generated Markov chains can be assessed with specific tools in order to calculate performance indicators. Some results of system availability and reliability, calculated for the AltaRica model are given in Table 4.

It would be more interesting and efficient in our case to perform stochastic simulations of models. Thus, the model captures not only failures and repairs of components, but also their functional behavior (e.g. pulling, processing and loading of products), we need to focus on short periods of time (e.g. 24h against 10000h in traditional reliability studies) to calculate performance indicators.

8 Conclusion

In this article, we showed that assessing the reliability of systems with mobile components raises a number of specific modeling issues. Most of these issues stand in the modeling of interactions between components: these interactions can take place only under certain conditions, but many different components can exhibit the same behavior.

We investigated the relationship between PEPA nets, a performance modeling process algebra, and AltaRica, an engineering oriented modeling language for safety analysis. These formalisms rely on very similar mathematical foundations: they are based on finite state automata and they can be compiled into Markov chains. Thus, we have sought to compare their expressiveness at the modeling, rather than at the Markovian, level. Our comparison revealed that AltaRica provides no direct mechanisms for mobility modeling, in particular it does not allow modeling location dependent synchronizations. Thus we have enhanced AltaRica by incorporating this modeling construct and showed that it offers increased flexibility to the modeler. On the other way round, while the flow in a system can be naturally modeled

with AltaRica, PEPA nets provide no direct modeling mechanisms for it. The net structure prevents a direct modeling of remote interactions between components located in different places.

References

[1] http://www.dcs.ed.ac.uk/pepa/tools/.

[2] R. Adeline, J. Cardoso, P. Darfeuil, S. Humbert, and C. Seguin. Toward a methodology for the altarica modelling of multi-physical systems. In *Proceedings of European Safety and Reliability Conference, ESREL 2010*, Rhodes (Greece), September 2010.

[3] 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.

[4] J. Andrews and T. Moss. *Reliability and Risk Assessment*. John Wiley & Sons, 1993. ISBN 0-582-09615-4.

[5] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The altarica language and its semantics. *Fundamenta Informaticae*, 34:109–124, 2000.

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

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

[8] J. Bowles and L. Kloul. Synthesising pepa nets from iods for performance analysis. In *Proceedings of the 1st ACM SIGMETRICS/SIGSOFT Joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California*, 2010.

[9] P. Brameret, A. Rauzy, and J. Roussel. Assessing the dependability of systems with repairable and spare components. In J. Barbet, editor, *Actes du Congrès Lambda-Mu 18*, Octobre 2012.

[10] S. Donatelli. Superposed generalised stochastic petri nets: Definition and efficient solution. In M. Silva, editor, *Proceedings of 15th International Conference on Application and Theory of Petri Nets*, 1994.

[11] S. Gilmore, J. Hillston, and L. Kloul. Pepa nets. In M. Calzarossa and E. Gelenbe, editors, *Performance Tools and Applications to Networked Systems*, volume 2965, pages 311–335. LNCS, Springer-Verlag, 2004.

[12] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. Pepa nets: a structured performance modelling formalism. *Performance Evaluation*, 54(2):79–104, 2003.

[13] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. Software performance modelling using pepa nets. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Software and Performance (WOSP'04), Redwood City, California*, 2004.

[14] J. Hillston. The nature of synchronisation. In U. Herzog and M. Rettelbach, editors, *Proceedings of 2nd Process Algebra and Performance Modelling Workshop*, November 1994.

[15] J. Hillston. Tuning systems: From composition to performance. *The Computer Journal*, 48(4):385–400, 2005.

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

[17] R. Milner. Communicating and mobile systems: The pi-calculus. *Cambridge University Press*, 1999.

[18] T. Prosvirnova and A. Rauzy. Système de transitions gardées : formalisme pivot de modélisation pour la sûreté de fonctionnement. In J. Barbet, editor, *Actes du Congrès Lambda-Mu 18*, Octobre 2012.

[19] A. Rauzy. BDD for Reliability Studies. In K. Misra, editor, *Handbook of Performability Engineering*, pages 381–396. Elsevier, 2008. ISBN 978-1-84800-130-5.

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

[21] A. Rauzy. Anatomy of an efficient fault tree assessment engine. In R. Virolainen, editor, *Proceedings of International Joint Conference PSAM'11/ESREL'12*, June 2012.

[22] W. Sanders and J. Meyer. Reduced base model construction methods for stochastic activity networks. *IEEE Journal on Selected Areas in Communications*, 9(1):25?36, January 1991.

[23] R. Valk. Petri nets as token objects-an introduction to elementary object nets. In *Proc. of the 19th International Conference on Application and Theory of Petri Nets, volume 1420 of LNCS, pages 1?25, Lisbon*. Springer Verlag, 1998.