On the use of formal models in Software Performance Evaluation[*]

Juan Pablo López-Grao, José Merseguer, and Javier Campos

Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain,
{jpablo,jmerse,jcampos}@posta.unizar.es

Abstract. Importance of performance evaluation at first stages of the software development life-cycle has been progressively rising. We believe that the need for integration of formal models in the software engineering process is a must in order to apply well-known analysis techniques to software models. In previous papers it has been stated our proposal of extension of UML semantics for some diagram types and a complete method to translate them into GSPN models. Here we will focus on activity diagrams: a new translation method for them will be presented, while we explain their link with other UML diagrams such as statecharts so as to amplify the expressivity at system description. Last but not least, our CASE tool prototype will be introduced. As it will be seen, every modeling aspect for these diagrams will be covered and, thanks to it, the translation process will be automatically performed.

Keywords: UML, software performance, Generalized Stochastic Petri nets, compositionality, modeling, CASE tool

1 Introduction

Unified Modeling Language (UML) [7] is a semi formal language developed by the Object Management Group [7] to specify, visualize and document models of software systems and non-software systems too. UML defines three categories of diagrams: static diagrams, behavioural diagrams and diagrams to organize and manage application modules. Behavioural diagrams are intended to describe system dynamics, therefore they play a prominent role for us since the objective of our works is the performance evaluation [20] of software systems at the first stages of the software development process [23, 25]. These diagrams belong to five kinds: Use Case diagram, Sequence diagram, Activity diagram, Collaboration diagram and Statechart diagram.

Our proposal consists in introducing new syntactical elements in UML diagrams to model performance concepts. By doing so, the software engineer can model behavioural, functional and performance requirements in a consistent fashion. In this paper the role played by the Activity diagram (AD) for the

[*] This work has been developed within the project P084/2001 of the Gobierno de Aragón, and the project UZ00-TEC-03 of the Universidad de Zaragoza.

performance evaluation of software systems is fully analyzed under the perspective of this proposal [19]. Since UML diagrams are not meant for performance evaluation and moreover its semantics is "informally" defined, we translate them into Generalized Stochastic Petri nets (GSPN) [1], gaining a formal semantics for them and besides an analyzable model. Obviously, the translation implies taking decisions on the interpretation of the diagrams.

So far we have dealt with the Sequence diagram (SD) (by means of the UML collaborations package) and the Statechart diagram (SC) [6] (by means of the UML state machines package). In the case of the AD we base our interpretation on the fact that ADs are suitable for internal flow process modeling, therefore they are relevant to describe activities performed by the system, usually expressed in the SC as doActivities in the states.

In this paper we investigate the key concepts to describe performance issues in the context of the AD and we give a formal semantics for the AD in terms of GSPN, compatible with that proposed for the SD and the SC in [6]. Furthermore, we briefly overview our prototype tool, which implements both aspects: its GUI is designed as a front end to model annotated ADs whereas the tool itself constructs their translation into GSPNs in order to analyze them with the GreatSPN tool [12].

We adopt the notation defined in [1] for GSPNs, but simplified to consider only ordinary systems (Petri nets in which arcs have weight at most one). A GSPN system is a 8-ple $S = (P, T, \Pi, I, O, H, W, M^0)$, where P is the set of places, T is the set of immediate and timed transitions, $P \cap T = \emptyset$; $\Pi : T \longrightarrow \mathbb{N}$ is the priority function that maps transitions onto natural numbers representing their priority level, by default, timed transitions have priority equal to zero; $I, O, H : T \longrightarrow 2^P$ are the input, output, inhibition functions, respectively, that map transitions onto the power set of P; $W : T \longrightarrow \mathbb{R}$ is the weight function that assigns real (positive) numbers to rates of timed transitions and to weights of immediate transitions. Finally, $M^0 : P \longrightarrow \mathbb{N}$ is the initial marking function.

A labeled ordinary GSPN (LGSPN) is then a triplet $\mathcal{LS} = (S, \psi, \lambda)$, where S is a GSPN ordinary system, as defined above, $\lambda : T \longrightarrow L^T \cup \tau$ is the labeling function that assigns to a transition a label belonging to the set $L^T \cup \tau$ and $\psi : P \longrightarrow L^P \cup \tau$ is the labeling function that assigns to a place a label belonging to the set $L^P \cup \tau$. τ-labeled net objects are considered to be internal.

Note that, with respect to the definition of LGSPN system given in [10], here both places and transitions can be labeled, moreover, the same label can be assigned to place(s) and to transition(s) since it is not required that L^T and L^P are disjoint.

The rest of the article is organized as follows: Section 2 describes the proposed annotations for the ADs and enumerates the main rules of the translation method. Section 3 analyzes the translation of each element in the AD into a stochastic Petri net model. Section 4 discusses how the stochastic Petri net model for the whole AD is obtained. Section 5 briefly presents our tool prototype. Finally, section 6 summarizes the paper, explores the bibliography and discusses future extensions.

2 Activity Diagrams for Performance Evaluation

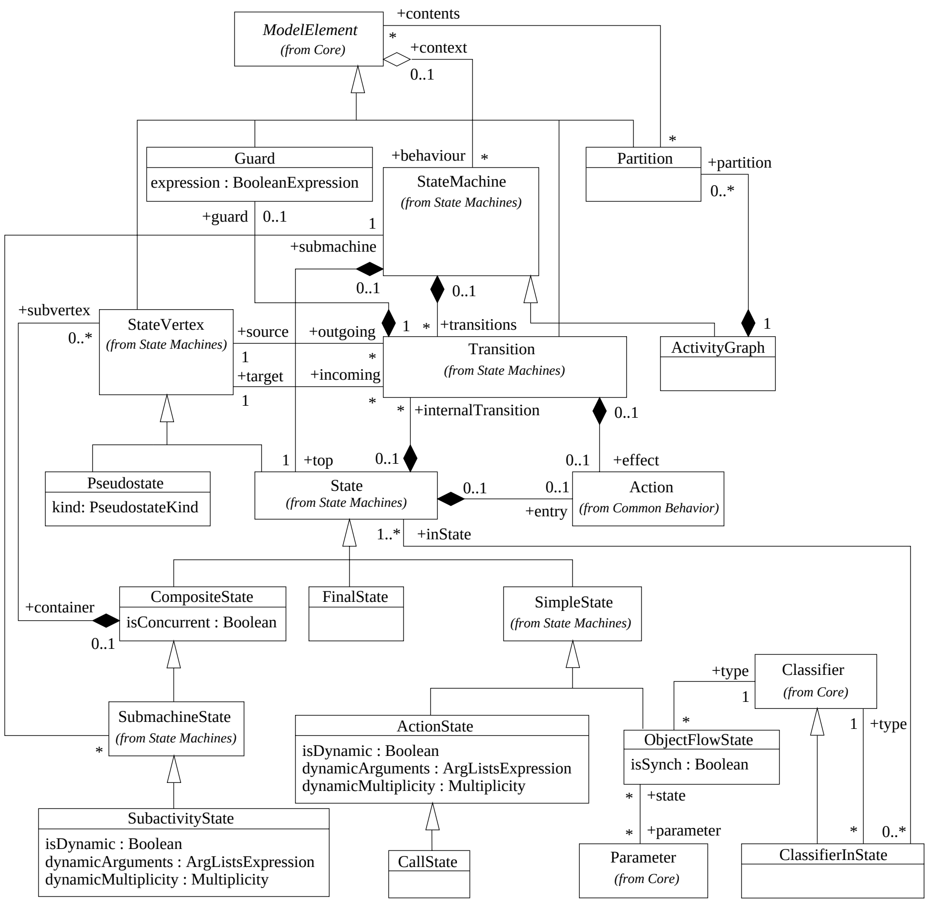

Fig. 1. UML Activity Graphs metamodel (extended)

Activity Diagrams represent UML activity graphs and are just a variant of UML state machines (see [7], section 3.84). In fact, a UML activity graph is a specialization of a UML state machine (SM), as it is expressed in the UML metamodel (see figure 1). The main goal of ADs is to stress the internal control flow of a process in contrast to statechart diagrams, which represent UML SMs and are often driven by external events.

As our objective is to use ADs to obtain performance measures of the model element they describe, we need additional modeling information, such as routing rates or the duration of the basic actions. We propose to annotate the AD to collect this information: subsection 2.1 describes this proposal.

It must be noted that in this paper we only focus in those elements proper of ADs. See that, according to UML specification ([7], section 3.84), almost every state in an AD should be an action or subactivity state, so almost every transition[1] should be triggered by the ending of the execution of the entry action or activity associated to the state. Anyway, UML is not strict at this point, so elements from state machines package could occasionally be used.

As far as it is concerned, our decision is not to allow other states than action, subactivity or call states, and thus to accept only the use of external events by means of call states and control icons involving signals, i.e. signal sendings and signal receipts. As a result of this, events are always deferred (as any event is always deferred in an action state), so an activity will not ever be interrupted when it is described by an AD. Further attempts to include other SM elements are not discarded and could be object of future work, although they introduce some new problems.

Anyway, we suggest the use of SCs to describe the dynamical behaviour of those parts of the system dependable of external events.

2.1 Performance annotations

Our proposal is to include two different aspects in our annotations: time and probability. As it was stated in previous papers [19], we will use tagged values as an extensibility mechanism to integrate them in our UML models. Annotations will be attached to transitions instead of states as in previous works, in order to allow the assignment of different action durations depending on the decision. Anyway, any other syntax is accepted as long as it is consistent with the process described below.

It must be noticed that, in the following, we will use the notion of not-timed transitions in the scope of ADs to specify those arcs which have no time annotation or to which a duration equal to zero is assigned. Doing so, we are trying to avoid confusions with immediate transitions, as long as they are different concepts in the domain of UML SMs.

The format suggested is {n sec.; P(k)} or {n-m sec.; P(k)} for timed transitions and {P(k)} for not-timed transitions. If no probability P(k) is provided we will assume an equiprobable sample space, i.e., identical probability for each 'brother' transition to be triggered. As it is shown, we allow time expressed in terms of an estimated value or a range of them. We have discarded the usage of packet size annotations as proposed for SMs [19] due to the fact that ADs are commonly used to model internal control flow. Figure 5 shows some examples of annotations (in braces).

Time annotations will be allocated wherever an action is executed (outgoing transitions of such states, including outgoing transitions of decision pseudostates with an action state as input) and probability annotations wherever a decision

[1] Notice that the word 'transition' has different meanings in UML and PNs domain. We preserve both meanings in this paper as the context should be enough to discriminate the 'transition' we are referring to (UML or PN 'transition')

is taken, i.e. next to guard conditions. It must be noticed that there is a special case where the performance annotation is attached to the state instead of the outgoing transition: when the control flow is not shown because it is implicit in the action-object flow. We do so because we do not want to have performance annotations applied to it, as it has a different semantics.

2.2 Translation rules and formal definitions

A brief description of each AD element and their translation to LGSPNs is presented in the next section. Section 4 illustrates the method to compose those LGSPNs to obtain the whole model for a concrete AD according to our proposed semantics. We must note that, in the following, we suppose that every object derived from ModelElement metaclass has an unique name within its namespace, although it could be not explicitly shown in the model.

As a rule, the translation of each one of AD elements can be summarized as a three-phased process:

step 1 Translation of each outgoing and self-loop transition. Applicable to action, subactivity and call states, and to fork pseudostates. Depending on the kind of transition, a different rule must be applied (see figures 2 and 4).

step 2 Composition of the LGSPNs corresponding to the whole set of each kind of transitions considered in step 1. Applicable to action, subactivity and call states, and to fork pseudostates.

step 3 Working out the LGSPN for the element by superposition of the LGSPNs obtained in the last step (if any) and, occasionally, an additional LGSPN corresponding to the entry to its associated state.

The formal definition of one of the LGSPN systems shown in Figure 2 is stated below. The rest of the cases in Figures 2 and 4 are straightforward from this example, so they will not be explicitly illustrated.

From now onward, we will adopt the Object Constraint Language [7] (OCL) syntax to indicate the image of an element (or of a set of elements) belonging to the domain of a certain relation. Let us suppose there exists an association between the classes D and C and let *rel* be the name of the role played by the class C in the association, then the image of an instance d of class D, through the derived relation *rel*: $D \rightarrow C$, is denoted as $d.rel$. Also the attributes of a class D, say at_1, and at_2 are denoted using the dot notation, $D.at_1$, and $D.at_2$.

A system for an outgoing timed transition *ott* of an action state AS (see figure 2, case 1.a) is a LGSPN $\mathcal{LS}^{ott}_{AS} = (S^{ott}_{AS}, \psi^{ott}_{AS}, \lambda^{ott}_{AS})$ characterized by the set of transitions $T^{ott}_{AS} = \{t_1, t_2\}$, and the set of places $P^{ott}_{AS} = \{p_1, p_2, p_3\}$. The input and output functions are respectively equal to:

$$I^{ott}_{AS}(t) = \begin{cases} \{p_1\} & \text{if } t = t_1 \\ \{p_2\} & \text{if } t = t_2 \end{cases} \qquad O^{ott}_{AS}(t) = \begin{cases} \{p_2\} & \text{if } t = t_1 \\ \{p_3\} & \text{if } t = t_3 \end{cases}$$

There are no inhibitor arcs, so $H^{ott}_{AS}(t) = \emptyset$. The priority and the weight functions are respectively equal to:

$$\Pi^{ott}_{AS}(t) = \begin{cases} 0 & \text{if } t = t_2 \\ 1 & \text{if } t = t_1 \end{cases} \qquad W^{ott}_{AS}(t) = \begin{cases} r_{ott} & \text{if } \Pi^{ott}_{AS}(t) = 0 \\ p_{cond} & \text{if } \lambda^{ott}_{AS}(t) = cond\_ev \\ 1 & \text{otherwise} \end{cases}$$

where, in this case, r_{ott} is the rate parameter of the timed transition t_2 and p_{cond} is the weight of the immediate transition t_1.

The weight p_{cond} is assigned the value of the probability annotation attached to the AD transition *ott*. If there is not such annotation, p_{cond} is equal to $1/nt$, where nt is the number of elements in the set $AS.outgoing$.

The rate r_{ott} is equal to $1/n$ if the time annotation attached to the AD transition is expressed in the format $\{n\ sec.\}$, or equal to $2/n{+}m$ if it is expressed in the format $\{n-m\ sec.\}$. Furthermore, in our CASE tool (presented in section 5) the last case is considered as a parameter of the system in order to automatize the analysis of the final LGSPN for different values within the range specified.

The initial marking function is defined as $\forall p \in P^{ott}_{AS} : M^{ott0}_{AS}(p) = \emptyset$. Finally, the labeling functions are equal to:

$$\psi^{ott}_{AS}(p) = \begin{cases} ini\_AS & \text{if } p = p_1 \\ execute & \text{if } p = p_2 \\ ini\_nextx & \text{if } p = p_3 \end{cases} \qquad \lambda^{ott}_{AS}(t) = \begin{cases} cond\_ev & \text{if } t = t_1 \\ out\_\lambda & \text{if } t = t_2 \end{cases}$$

where, for abuse of notation, $AS = AS.name$ and $nextx = ott.target.name$.

As they are profusely used in next section, we also define AG as the activity diagram, $Lstvertex^P$ the set of labels of state vertices in it, $Lstvertex^P = \{ini\_target, \forall target \in AG.transitions{\rightarrow}target.name\}$ and Lev^P as the set of events in the system, $Lev^P = \{e\_evx, \forall evx \in Ev\} \cup \{ack\_evx, \forall evx \in Ev\}$.

3 Translating Activity Diagram elements

The following subsections are devoted to translate each diagram element into a LGSPN; the composition of these nets (section 4) results in a stochastic Petri net system that will be used to obtain performance parameters for the modelled element.

3.1 Action states

An action state is 'a shorthand for a state with an entry action and at least one outgoing transition involving the implicit event of completing the entry action' ([7], section 3.85). According to this definition and the translation of simple states in SMs [19] we should interpret the action atomic and therefore represent it by an immediate transition within the LGSPN corresponding to the state. However, if we considered every action immediate (for action states), then most of the activities modelled by ADs would be immediate too, when they are expected to have a concrete duration. So we will distinguish between timed and not-timed transitions (in ADs) to determine the type of transition needed —timed or immediate— and its rate associated in the resulting LGSPN.

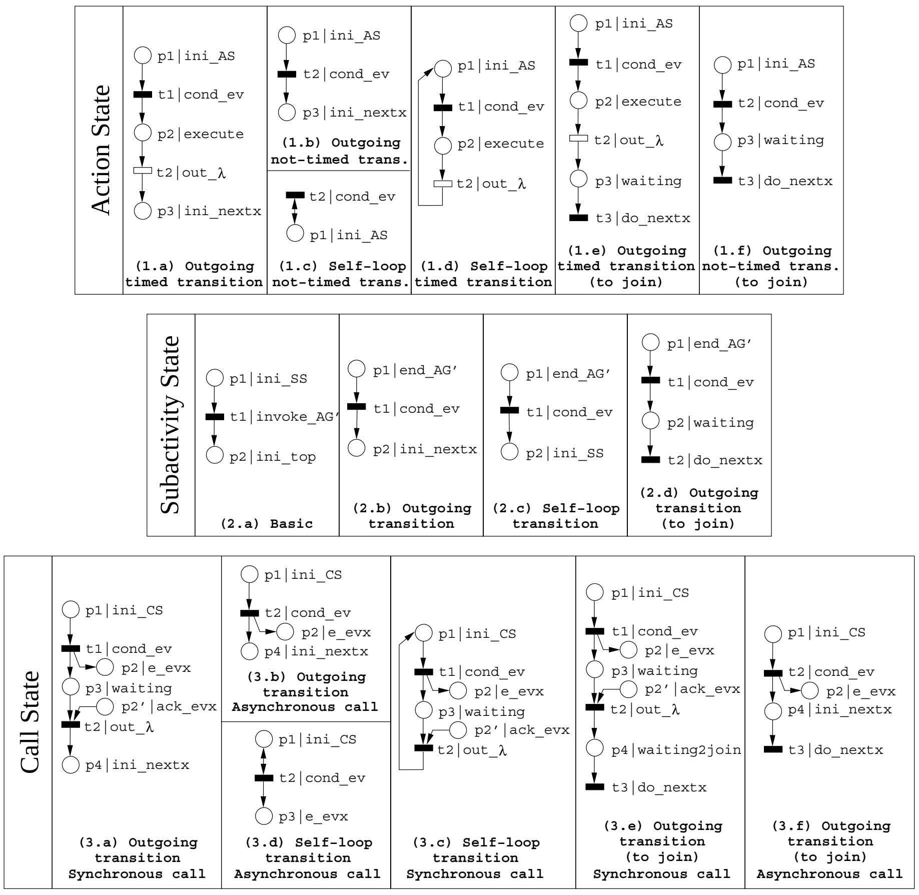

Fig. 2. Action, Subactivity and Call States to LGSPN

Translating an action state into LGSPN formalism takes the three steps expressed in section 2.2. Given an action state AS let q be the number of outgoing timed transitions OT_i of the state (which do not end in a join pseudostate), q' the number of outgoing not-timed transitions ON_j (which do not end in a join pseudostate), r the number of outgoing timed transitions OTJ_m that end in a join pseudostate, r' the number of outgoing not-timed transitions OTN_n that end in a join pseudostate, s the number of self-loop timed transitions ST_k and s' the number of self-loop not-timed transitions SN_l.

Then for each outgoing or self-loop transition t, we have a LGSPN $\mathcal{LS}^t_{AS} = (S^t_{AS}, \psi^t_{AS}, \lambda^t_{AS})$ as shown in figure 2, cases 1.a-1.f. This results in a set of $q + q' + r + r' + s + s'$ LGSPN models that need to be combined to get a model of the state AS, $\mathcal{LS}_{AS} = (S_{AS}, \psi_{AS}, \lambda_{AS})$.

Firstly we must compose the submodels of the transitions of the same type, using the superposition operators defined in Appendix A and the following equations:

$$\mathcal{LS}_{AS}^{OT} = \underset{Lstvertex^P}{\overset{i=1,\ldots,q}{||}} \mathcal{LS}_{AS}^{OT\,i} \qquad \mathcal{LS}_{AS}^{ON} = \underset{Lstvertex^P}{\overset{j=1,\ldots,q'}{||}} \mathcal{LS}_{AS}^{ON\,j}$$

$$\mathcal{LS}_{AS}^{ST} = \underset{ini\_AS}{\overset{k=1,\ldots,s}{||}} \mathcal{LS}_{AS}^{ST\,k} \qquad \mathcal{LS}_{AS}^{SN} = \bigsqcup_{ini\_AS,out\_\lambda}^{l=1,\ldots,s'} \mathcal{LS}_{AS}^{SN\,l}$$

$$\mathcal{LS}_{AS}^{OTJ} = \underset{ini\_AS}{\overset{m=1,\ldots,r}{||}} \mathcal{LS}_{AS}^{OTJ\,m} \qquad \mathcal{LS}_{AS}^{ONJ} = \underset{ini\_AS}{\overset{n=1,\ldots,r'}{||}} \mathcal{LS}_{AS}^{ONJ\,n}$$

Again composing the subsystems just shown, the LGSPN model $\mathcal{LS}_{AS}$ is now defined by:

$$\mathcal{LS}_{AS} = ((((\mathcal{LS}_{AS}^{SN} \underset{ini\_AS}{||} \mathcal{LS}_{AS}^{ST}) \underset{ini\_AS}{||} \mathcal{LS}_{AS}^{ON}) \underset{Lstvertex^P}{||} \mathcal{LS}_{AS}^{OT}) \underset{ini\_AS}{||} \mathcal{LS}_{AS}^{OTJ}) \underset{ini\_AS}{||} \mathcal{LS}_{AS}^{ONJ}$$

Finally we must remember that UML lets any kind of action to be executed inside an action state. That means we might find a CallAction or a SendAction there. However, UML syntax provides two concrete elements for this type of states: call states and signal sending icons. We suggest their use, but if an action state is used instead, then we should apply the translation method described for the equivalent element (call state or signal sending control icon).

3.2 Subactivity states

A subactivity state always invokes a nested AD. Its outgoing transitions do not have time annotations attached, as the duration activity can be determined translating the AD and composing the whole system (that will be seen later in this paper).

Translating a subactivity state into LGSPN formalism takes the three steps expressed in section 2.2. Notice that there is an additional LGSPN that corresponds with the entry to the state, called *basic*.

Then, given a subactivity state SS let q be the number of outgoing transitions O_i of the state (which do not end in a join pseudostate), r the number of outgoing transitions OJ_k that end in a join pseudostate, and s the number of self-loop transitions S_j. Also let AG' be the nested activity diagram and *top* the name of the first element of AG', $top = AG'.top$.

According to the translations shown in figure 2, cases 2.a-2.d, we have a basic LSGPN $\mathcal{LS}_{SS}^{B} = (S_{SS}^{B}, \psi_{SS}^{B}, \lambda_{SS}^{B})$ and one LGSPN for each outgoing or self-loop transition t, $\mathcal{LS}_{SS}^{t} = (S_{SS}^{t}, \psi_{SS}^{t}, \lambda_{SS}^{t})$. Therefore, we have $q + r + s + 1$ LGSPN models that need to be combined to get a model of the state SS, $\mathcal{LS}_{SS} = (S_{SS}, \psi_{SS}, \lambda_{SS})$. The LGSPNs corresponding to each set of kind of transitions are now obtained by superposition:

$$\mathcal{LS}^{O}_{SS} = \underset{Lstvertex^P,end\_AG}{\overset{i=1,\dots,q}{||}} \mathcal{LS}^{O_i}_{SS} \qquad \mathcal{LS}^{S}_{SS} = \underset{end\_AG,out\_\lambda,ini\_SS}{\overset{j=1,\dots,s}{\bigsqcup}} \mathcal{LS}^{S_j}_{SS}$$

$$\mathcal{LS}^{OJ}_{SS} = \underset{end\_AG}{\overset{k=1,\dots,r}{||}} \mathcal{LS}^{OJ_k}_{SS}$$

And the final LGSPN model $\mathcal{LS}_{SS}$ for the subactivity state is now defined by:

$$\mathcal{LS}_{SS} = ((\mathcal{LS}^{OJ}_{SS} \underset{end\_AG}{||} \mathcal{LS}^{S}_{SS}) \underset{end\_AG}{||} \mathcal{LS}^{O}_{SS}) \underset{ini\_SS}{||} \mathcal{LS}^{B}_{SS}$$

3.3 Call states

Call states are a particular case of action states in which its associated entry action is a CallAction, so translation of these elements is quite similar. It must be noted that when a CallAction is executed a set of CallEvents may be generated. For the sake of simplicity, we assume that at most one event is generated, but definition can be extended adding new places in the LGSPN to consider that possibility as well.

Besides, the CallAction may be synchronous or not depending on the value of its attribute *isAsynchronous*, where *synchronous* means that the action will not be completed until the event eventually generated by the action is not consumed by the receiver. In that case, we need a new place and transition in the corresponding LGSPN to model the synchronization (see figure 2, cases 3.a, 3.c and 3.e).

To translate a call state, steps to follow are similar to those described in section 2.2. Given a call state CS,

- If verifies $S.entry.IsAsynchronous = false$ (i.e., its associated CallAction is a synchronous call) we define u as the number of outgoing transitions OS_i of the state (which do not end in a join pseudostate), v the number of outgoing transitions OJS_k that end in a join pseudostate and w the number of self-loop transitions SS_m.
- If verifies $S.entry.IsAsynchronous = true$ -i.e., its associated CallAction is an asynchronous call- we define u' as the number of outgoing transitions OA_j of the state (which do not end in a join pseudostate), v' the number of outgoing transitions OJA_l that end in a join pseudostate, and w' the number of self-loop transitions SA_n.

Also let evx be an event generated by the call action, $evx = S.entry.operation \rightarrow ocurrence$. Considering this, we have one LGSPN for each outgoing or self-loop transition t, $\mathcal{LS}^{t}_{CS} = (S^{t}_{CS}, \psi^{t}_{CS}, \lambda^{t}_{CS})$, as shown in figure 2, cases 3.a-3.f. Therefore, we have either $u + v + w$ or $u' + v' + w'$ LGSPN models that need to be combined to get a model of the state CS, $\mathcal{LS}_{CS} = (S_{CS}, \psi_{CS}, \lambda_{CS})$.

The LGSPNs corresponding to each set of kind of transitions are now obtained by superposition:

$$\mathcal{LS}_{CS}^{OS} = \mathop{||}\limits_{Lstvertex^P,Lev^P}^{i=1,\ldots,u} \mathcal{LS}_{CS}^{OS_i} \quad \mathcal{LS}_{CS}^{OA} = \mathop{||}\limits_{Lstvertex^P,Lev^P}^{j=1,\ldots,u'} \mathcal{LS}_{CS}^{OA_j}$$

$$\mathcal{LS}_{CS}^{OJS} = \mathop{||}\limits_{ini\_CS,Lev^P}^{k=1,\ldots,v} \mathcal{LS}_{CS}^{OJS_k} \quad \mathcal{LS}_{CS}^{OJA} = \mathop{||}\limits_{ini\_CS,Lev^P}^{l=1,\ldots,v'} \mathcal{LS}_{CS}^{OJA_l}$$

$$\mathcal{LS}_{CS}^{SS} = \mathop{||}\limits_{ini\_CS,Lev^P}^{m=1,\ldots,w} \mathcal{LS}_{CS}^{SS_m} \quad \mathcal{LS}_{CS}^{SA} = \mathop{||}\limits_{ini\_CS,Lev^P}^{n=1,\ldots,w'} \mathcal{LS}_{CS}^{SA_n}$$

The final LGSPN for the state $\mathcal{LS}_{CS}$ is defined by one of the two following equations, depending on whether the action was synchronous or not:

$$\mathcal{LS}_{CS} = (\mathcal{LS}_{CS}^{SS} \mathop{||}\limits_{ini\_CS,Lev^P} \mathcal{LS}_{CS}^{OS}) \mathop{||}\limits_{ini\_CS,Lev^P} \mathcal{LS}_{CS}^{OJS} \qquad (synchronous)$$

$$\mathcal{LS}_{CS} = (\mathcal{LS}_{CS}^{SA} \mathop{||}\limits_{ini\_CS,Lev^P} \mathcal{LS}_{CS}^{OA}) \mathop{||}\limits_{ini\_CS,Lev^P} \mathcal{LS}_{CS}^{OJA} \qquad (asynchronous)$$

3.4 Decisions

Decisions are preprocessed before the AD translation, as it will be mentioned in section 4.1. They are substituted by equivalent outgoing transitions on action states (as shown in figure 3), preserving the properties inherent in performance annotations. Therefore, they do not have to be translated.

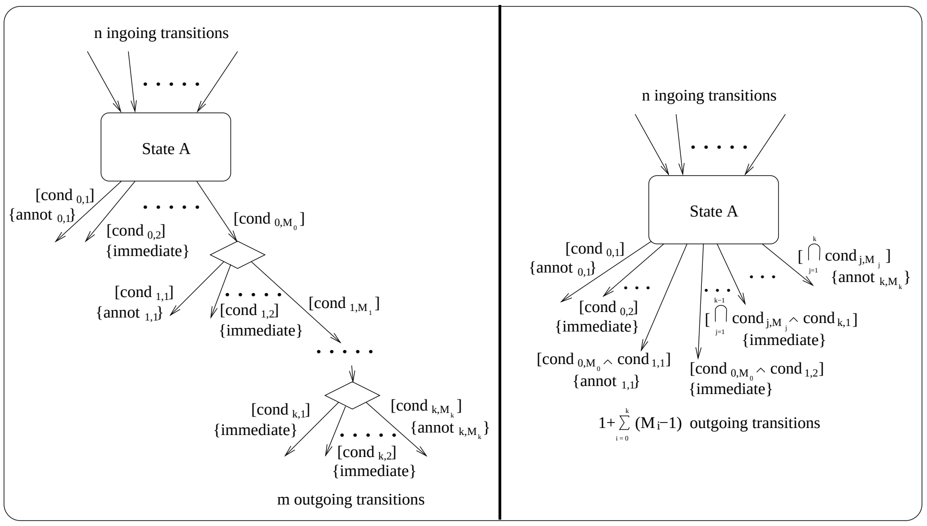

Fig. 3. Decision to LGSPN (Pre-transformation)

3.5 Merges

Merges are used to reunify control flow, separated in divergent branches by decisions (or outgoing transitions of states labelled with guards). Often they are just a notational convention, as reunification may be modelled as ingoing transitions of a state.

Translation of a merge pseudostate M depends on the kind of target element of its outgoing transition. Figure 4 (cases 5.a and 5.b) shows the direct translation of the model, $\mathcal{LS}_M$, according to the condition expressed below.

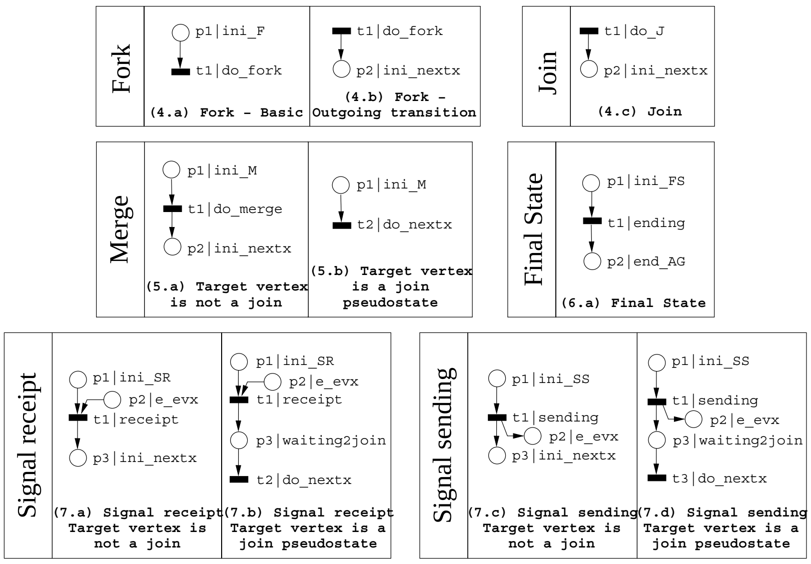

Fig. 4. Fork, Join, Merge, Final State, Signal Sending and Signal Receipt to LGSPN

(a) $\mathcal{LS}_M = \mathcal{LS}'_M \iff (PS.outgoing.target \notin Pseudostate \vee PS.outgoing.target.kind \neq join)$ (to join)
(b) $\mathcal{LS}_M = \mathcal{LS}''_M \iff (PS.outgoing.target \in Pseudostate \wedge PS.outgoing.target.kind = join)$ (not to join)

3.6 Concurrency support items

UML provides two elements to model concurrency in an AD: forks and joins. Their use and meaning do not need further explanation, as they have been commonly explained in classic literature. Translation into LGSPN models is quite simple in both cases.

Given a join pseudostate J, it is translated into the labelled system $\mathcal{LS}_J$, shown in figure 4, case 4.c.

To translate forks, three steps must be followed:

Given a fork pseudostate F let q be the number of its outgoing transitions O_i. Then, according to the translations shown in figure 4, we have a basic LSGPN $\mathcal{LS}_F^B = (S_F^B, \psi_F^B, \lambda_F^B)$ (case 4.a in the figure) and one LGSPN (case 4.b) for each outgoing transition t, $\mathcal{LS}_F^t = (S_F^t, \psi_F^t, \lambda_F^t)$. Therefore, we have $q+1$ LGSPN models that need to be combined to get a model of the pseudostate, $\mathcal{LS}_F = (S_F, \psi_F, \lambda_F)$. The LGSPNs corresponding to each set of kind of transitions are now obtained by superposition:

$$\mathcal{LS}_F^O = \underset{do\_fork, Lstvertex^P}{\overset{i=1,\ldots,q}{||}} \mathcal{LS}_F^{O_i}$$

And the final LGSPN $\mathcal{LS}_F$ is composed following the expression:

$$\mathcal{LS}_F = \mathcal{LS}_F^B \underset{do\_fork}{||} \mathcal{LS}_F^O$$

3.7 Initial and final states

Initial pseudostates and final states are elements inherited from UML state machines semantics. However, unlike it happened on UML SMs [6] the initial pseudostate is not translated into a LGSPN model when translating an AD, as no action can be attached to its outgoing transition. On the other hand, final states are translated, but the resulting LGSPN is different from that shown in [6].

Given a final state FS, the LGSPN model $\mathcal{LS}_{FS} = (S_{FS}, \psi_{FS}, \lambda_{FS})$ equivalent to the state is defined according to the translation shown in figure 4, case 6.a.

3.8 Signal sending and signal receipt

Signal sending and signal receipt symbols are control icons. That means they are not really necessary, but are used as a notational convention to specify common modeling matters. In our specific case, these symbols are the only mechanisms we allow to model the processing of external events, and are equivalent to labelling the outgoing transition of a state with a SendAction corresponding to the signal as an effect or with the name of the SignalEvent expected as the trigger event, respectively.

As these symbols are control icons, there is not a metaclass corresponding to this elements in UML metamodel. So we assume that before translating the diagram a unique identificator is assigned to each one of these elements, so when we say $t.target.name$, where t is a incoming transition of the control icon, we are refering to this identificator (instead of the name of the real target StateVertex according to the metamodel).

Given a signal sending/receipt symbol CS, the translation of the symbol depends on whether this target element is a join pseudostate or not:

– If the symbol is a signal sending, then let $SIGS$ be its pre-assigned identificator. Its translation into a LGSPN model $\mathcal{LS}_{SIGS}$ is shown in figure 4, cases 7.c-7.d.

– If the symbol is a signal receipt, then let *SIGR* be its pre-assigned identificator. Its translation into a LGSPN model $\mathcal{LS}_{SIGR}$ is shown in figure 4, cases 7.a-7.b.

It must be noted that, as far as signal sendings is concerned, we have assumed that at most one event is generated for simplicity, but definition can be extended adding new places in the LGSPN to consider that possibility as well.

3.9 Performance-irrelevant constructs

Some elements from ADs are not relevant for performance evaluation, so they are not translated into LGSPN models. These elements are:

– *Swimlanes*, which have no meaning in the dynamics of the system modelled, as they are mechanisms to organize visually the states within the diagram.
– *Action-Object Flow relationships*, as they do not provide any additional concrete information about the behavior of the system.
– *Deferrable events* as, according to our interpretation (see section 2), any event is deferred in an AD (except, obviously, SignalEvents when a signal receipt symbol is found).

4 The System Translation Process

In the previous section we have presented our method to translate every AD element into LGSPN models. Here, we will focus on the whole system translation process, presenting an overview of the steps to follow and allocating the ideas already presented in their own timing. The process includes the complete translation method for ADs and the way to integrate the resulting LGSPN with the ones obtained from the translation of UML SMs and SDs [6].

4.1 Translating activity diagrams into GSPN

As an initial premise we assume that every AD in the system description has exactly one initial state plus, at least, one final state and another state from one of the accepted types (action, subactivity or call state). The translation of an AD can then be divided in three phases, which are presented in the subsequent paragraphs.

Pre-transformations Before translating the AD into a LGSPN model, we need to apply some simplifications to the diagram in order to properly use the translations given in section 3. These simplifications are merely syntactical so the system behaviour is not altered. Most relevant ones are:

– Suppression of decisions. Figure 3 shows a particular case of this kind of transformation. New decisions could be found in any branch of the chaining tree, but the figure has been simplified for the sake of simplicity.

– Suppression of merges / forks / joins chaining, bringing them together into a unique merge / fork / join pseudostate.
– Deducting and making explicit the implicit control flow in action-object flow relationships, where aplicable.
– Avoidance of bad design cases (e.g., when the target of a fork pseudostate is a join pseudostate).

Translation process Once pre-transformations are applied we can proceed to translate the diagram into a LGSPN model. This is done following three steps:

step 1 Translation of each diagram element, as shown in section 2.
step 2 Superposition of the LGSPNs corresponding to the whole set of each kind of diagram elements:

$$\mathcal{LS}_{AG}^{actst} = \underset{Lstvertex^P}{\overset{AS\in ActionStates}{||}} \mathcal{LS}_{AS} \qquad \mathcal{LS}_{AG}^{subst} = \underset{Lstvertex^P}{\overset{SS\in SubactivityStates}{||}} \mathcal{LS}_{SS}$$

$$\mathcal{LS}_{AG}^{calst} = \underset{Lstvertex^P,Lev^P}{\overset{CS\in CallStates}{||}} \mathcal{LS}_{CS} \qquad \mathcal{LS}_{AG}^{merge} = \underset{Lstvertex^P}{\overset{M\in Merges}{||}} \mathcal{LS}_{M}$$

$$\mathcal{LS}_{AG}^{fork} = \underset{Lstvertex^P}{\overset{F\in Forks}{||}} \mathcal{LS}_{F} \qquad \mathcal{LS}_{AG}^{join} = \underset{Lstvertex^P}{\overset{J\in Joins}{||}} \mathcal{LS}_{J}$$

$$\mathcal{LS}_{AG}^{finst} = \underset{end\_AG}{\overset{FS\in FinalStates}{||}} \mathcal{LS}_{FS} \qquad \mathcal{LS}_{AG}^{sigse} = \underset{Lstvertex^P,Lev^P}{\overset{SIGS\in SignalSendings}{||}} \mathcal{LS}_{SIGS}$$

$$\mathcal{LS}_{AG}^{sigre} = \underset{Lstvertex^P,Lev^P}{\overset{SIGR\in SignalReceipts}{||}} \mathcal{LS}_{SIGR}$$

step 3 Working out the LGSPN for the diagram itself by superposition of the LGSPNs obtained in the last step:

$$\begin{aligned}\mathcal{LS}_{AG} = (((((((\mathcal{LS}_{AG}^{sigre} \underset{Lstvertex^P,Lev^P}{||} \mathcal{LS}_{AG}^{sigse}) \underset{Lstvertex^P}{||} \mathcal{LS}_{AG}^{finst})\\ \underset{Lstvertex^P}{||} \mathcal{LS}_{AG}^{join}) \underset{Lstvertex^P}{||} \mathcal{LS}_{AG}^{fork}) \underset{Lstvertex^P}{||} \mathcal{LS}_{AG}^{merge})\\ \underset{Lstvertex^P,Lev^P}{||} \mathcal{LS}_{AG}^{calst}) \underset{Lstvertex^P,end\_AG}{||} \mathcal{LS}_{AG}^{subst}) \underset{Lstvertex^P}{||} \mathcal{LS}_{AG}^{actst}\end{aligned}$$

Thanks to this compositional approach, all kind of legal dependencies between diagrams (as looping dependencies) can be processed. E.g., let AG_1 be an activity graph where SS is a subactivity state in it, $SS \in AG_1.transitions.source$, and let AG_2 be the activity graph that the state invokes, $AG_2 = SS.submachine$. Also let SS' be a subactivity state in AG_2, $SS' \in AG_2.transitions.source$, which invokes AG_1, $AG_1 = SS'.submachine$. The superposition operators allows the performance engineer to deal with such syntactical issues.

Post-optimizations Contrasting with pre-transformations, which are mandatory, post-optimizations are optional. Their objective is just to eliminate some spare places and transitions in the resulting LGSPN so as to make it more attractive without altering its semantics. One example of these kind of transformations would be, in subnets of the LGSPN corresponding to outgoing timed transitions of action states $\mathcal{LS}^{OT}_{AS}$, the removal of the superfluous immediate transitions (and their output place) in case of no conflict.

4.2 Composing the whole system

As it has been stated before, in terms of performance evaluation we use UML ADs exclusively to describe doActivities in SCs or activities inside subactivity states of others ADs. Hence, the merging of nets corresponding to SCs and ADs will be dealt with first.

Let d be the number of ADs used at system description and $Linterfaces^P = \{Lini\_top^P, Lev^P, Lend\_AG^P\}$, where $Lini\_top^P$ is the set of initial places of the LGSPNs corresponding to the ADs and $Lend\_AG^P$ the set of final places of those nets. Now, we can merge the referred LGSPNs by superposition (of places):

$$\mathcal{LS}_{ad} = \underset{Linterfaces^P}{\overset{AG \in ActivityDiagrams}{||}} \mathcal{LS}_{AG}$$

Now let $\mathcal{LS}''_{sc}$ be the LGSPN corresponding to the translation of the set of SCs in the model. $\mathcal{LS}''_{sc}$ was previously obtained by composition (superposition of places) of the nets obtained for each SC and subsequent removal of sink *acknowledge* places (see [6]).

Then let T_act be the set of transitions in $\mathcal{LS}''_{sc}$ labelled *activity* [6] which represent activities that are described with activity diagrams. $\mathcal{LS}_{sc}$ will be the result of that labelled system with the removal of this set of transitions, $\mathcal{LS}_{sc} = \mathcal{LS}''_{sc} \setminus T\_act$. Ingoing places for these transitions (labelled *end_entry_A* in $\mathcal{LS}''_{sc}$) will be now labelled *ini_top*, where *top* is the name of the first element of the activity diagram AG' that represents the activity, $top = AG'.top.name$. Similarly, outgoing places (labelled *compl_A*) will be now labelled *end_AG'*.

Once done, we can merge the LGSPN systems $\mathcal{LS}_{sc}$ and $\mathcal{LS}_{ad}$:

$$\mathcal{LS}_{sc-ad} = \mathcal{LS}_{ad} \underset{Linterfaces^P}{||} \mathcal{LS}_{sc}$$

The resulting net $\mathcal{LS}_{sc-ad}$ often represents the whole system behavior. However, this behavior can be constrained to obtain performance measures for a particular scenario (pattern of interaction). That is done by merging $\mathcal{LS}_{sc-ad}$ and the LGSPN corresponding to a specific SD into a unique LGSPN $\mathcal{LS}$, mainly by synchronization (i.e., superposition of transitions). Paper [6] describes two approaches for doing an analogous operation, using the referred net $\mathcal{LS}_{sc}$ instead of $\mathcal{LS}_{sc-ad}$. Nevertheless, both procedures are still directly applicable to the resulting LGSPN $\mathcal{LS}_{sc-ad}$.

A sample case of the translation of a very simple system is illustrated in figure 5. Two SC and AD models for the system are presented on the left side of the figure. We have obviated some diagrams of the system description so only part of the resulting LGSPN is included on its right side. That results in the lack of tokens in the initial marking of the net.

The SC represents the life-cycle of an object from the class *car wash machine*, that can be either working or inactive (i.e, waiting for a new car to be washed). The activity performed by the machine when it is working is described by the AD below. As it is shown, the machine works in a different way depending on the amount of money spent by the driver, and can do some tasks simultaneously.

It must be noted that the LGSPN subsystem for the SC has been simplified. In order to proceed to the composition of the LGSPN corresponding to the whole system we should eliminate the transition $t1$ and change the labels of the places $p2$ and $p3$ to *ini_weighcoins* and *end_wash_car*, respectively.

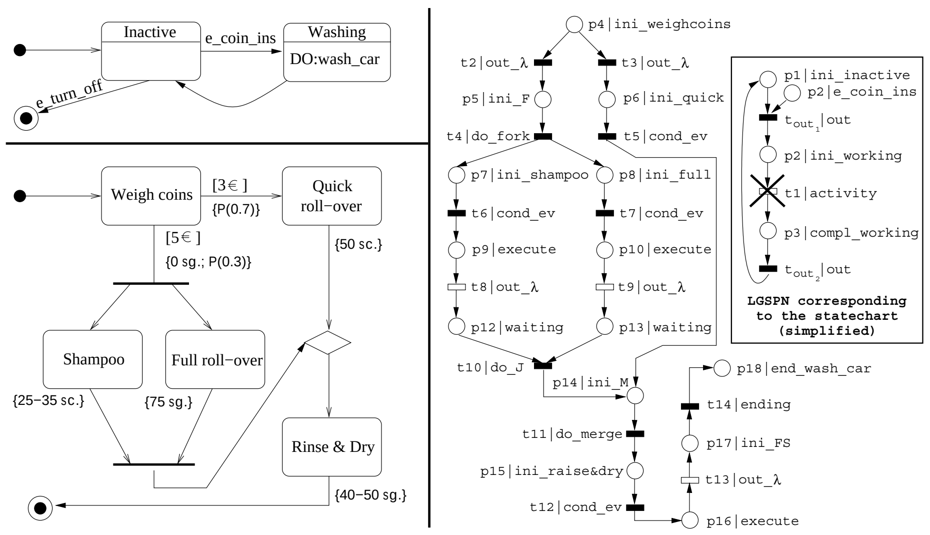

Fig. 5. Car wash machine example

5 Software Performance Tool

To accomplish our objective of successfully integrating techniques of performance evaluation in the software engineering process, an special effort in the automatization of the method is required. To do so, we have developed a CASE tool prototype that generates generalized stochastic Petri nets (GSPN) in a file format [9, 12] directly processable by the GreatSPN tool [12], which is used to make quantitative analysis and obtain performance rates.

The prototype itself provides full capability to model and translate any aspect of ADs as described in this paper, by means of an intuitive, flexible and highly configurable GUI. Furthermore, support for importing and exporting models in XMI [13] format, standard for CASE tools, is currently in development phase.

In order to easily describe complex systems with a number of diagrams, the tool and the internal file formats are fully project-oriented. This means that every UML element and diagram our tool handles always belongs to a project.

Finally it must be noted that an special effort has been made to obtain highly-legible GreatSPN nets, avoiding superposition of places and transitions. Figure 6 shows a snapshot of a diagram in our tool (the classical ‘coffee’ example, shown in UML specification [7]) and its resulting translation in GreatSPN, as it was obtained originally.

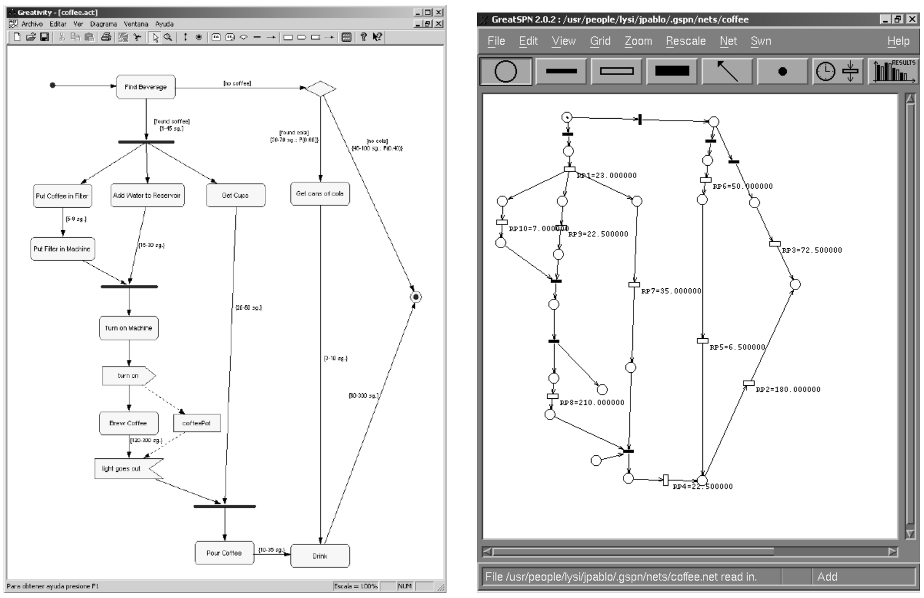

Fig. 6. Tool - Extended coffeepot example and results in GreatSPN

6 Conclusions

The main contributions of this paper can be summarized as follows:

– We have described the kind of annotations suitable to model performance requirements in the context of ADs.
– We have given a translation of the AD (that models a doActivity) into a stochastic Petri net model. In this way, it can be composed with any other stochastic Petri net model that represents a SC that uses the doActivity.
– A formal semantics for the AD is achieved in terms of stochastic Petri nets that allows to check logical properties as well as to compute performance

indices. Obviously, this formal semantics represents an interpretation of the "informally" defined concepts of the UML AD. Our interpretation is focused on the basis that the AD is meant for the description of the doActivities in a SC.
- A prototype tool has been implemented for the Windows® platform. It offers a front end that allows to model all the elements in UML ADs notation in contrast with other tools such as Rose [21] which does not allow to model important features such as signal sending or signal receipt symbols. Performance annotations can be introduced to produce a GSPN system that can be analyzed by the GreatSPN tool [12], therefore it is possible to obtain performance measures in the steady or transient state.

Concerning related work, in [5] can be found a survey of the different approaches for performance evaluation based on UML diagrams. Although there are several works devoted to obtain formal models from the UML SC [16, 15, 22] or the UML SD [24, 8, 3], some of them with performance evaluation purposes, the AD has not been studied yet so intensely. But an interesting work can be found in [11], where a formal semantics for the AD is given based on the STATEMATE semantics [14] of the statecharts.

To date it is not possible to compare our tool because to our knowledge there exist three tools [4, 2, 18] for performance evaluation based on UML but they do not use stochastic Petri nets as performance models. On the other hand, although DSPNExpress2000 [17] claims to be, it seems that only very simple SCs can be modelled with this tool. In SimML [4], simulation queuing networks models [20] for performance evaluation are obtained from UML class diagram and SD, while in the PERMABASE project [2] models for simulation are obtained from UML SD and class and deployment diagrams.

As future work we are working on the following open issues:

- With respect to UML ADs, conditional forks and more complex external event processing support, especially important to resolve the problem of 'uninterruptable' activities due to the use of action states.
- Extension of the prototype tool to support SCs and SDs in order to increase the expressivity at system description.
- Possibility of processing XMI files in our CASE tool prototype, in order to import models from other CASE tools and thus ensure compliance with current standards.

A Formal definition of composition of GSPNs

Place and transition superposition of two ordinary labeled GSPNs. Given two LGSPN ordinary systems $\mathcal{LS}_1 = (S_1, \psi_1, \lambda_1)$ and $\mathcal{LS}_2 = (S_2, \psi_2, \lambda_2)$, the LGSPN ordinary system $\mathcal{LS} = (S, \psi, \lambda)$:

$$\mathcal{LS} = \mathcal{LS}_1 \underset{L_T, L_P}{||} \mathcal{LS}_2$$

resulting from the composition over the sets of (no τ) labels L_T and L_P is defined as follows. Let $E_T = L_T \cap \lambda_1(T_1) \cap \lambda_2(T_2)$ and $E_P = L_P \cap \psi_1(P_1) \cap \psi_2(P_2)$ be the subsets of L_T and of L_P, respectively, comprising place and transition labels that are common to the two LGSPNs, P_1^l (T_1^l) be the set of places (transitions) of $\mathcal{LS}_1$ that are labeled l and $P_1^{E_P}$ ($T_1^{E_T}$) be the set of all places (transitions) in $\mathcal{LS}_1$ that are labeled with a label in E_P (E_T). Same definitions apply to $\mathcal{LS}_2$.

Then: $T = T_1 \backslash T_1^{E_T} \cup T_2 \backslash T_2^{E_T} \cup \bigcup_{l \in E_T} \{T_1^l \times T_2^l\}$, $P = P_1 \backslash P_1^{E_P} \cup P_2 \backslash P_2^{E_P} \cup \bigcup_{l \in E_P} \{P_1^l \times P_2^l\}$, the functions $F \in \{I(), O(), H()\}$ are equal to:

$$F(t) = \begin{cases} F_1(t) & \text{if } t \in T_1 \backslash T_1^{E_T} \\ F_2(t) & \text{if } t \in T_2 \backslash T_2^{E_T} \\ F_1(t_1) \cup F_2(t_2) & \text{if } t \equiv (t_1, t_2) \in T_1^{E_T} \times T_2^{E_T} \wedge \lambda_1(t_1) = \lambda_2(t_2) \end{cases}$$

where $\cup$ on the third line is the union over sets. Functions $F \in \{\Pi(), W()\}$ are equal to:

$$F(t) = \begin{cases} F_1(t) & \text{if } t \in T_1 \backslash T_1^{E_T} \\ F_2(t) & \text{if } t \in T_2 \backslash T_2^{E_T} \\ min(F_1(t_1), F_2(t_2)) & \text{if } t \equiv (t_1, t_2) \in T_1^{E_T} \times T_2^{E_T} \wedge \lambda_1(t_1) = \lambda_2(t_2) \end{cases}$$

The initial marking function is equal to:

$$M^0(p) = \begin{cases} M_1^0(p) & \text{if } \ p \in P_1 \backslash P_1^{E_P} \\ M_2^0(p) & \text{if } \ p \in P_2 \backslash P_2^{E_P} \\ M_1^0(p_1) + M_2^0(p_2) & \text{if } \ p \equiv (p_1, p_2) \in P_1^{E_P} \times P_2^{E_P} \wedge \psi_1(p_1) = \psi_2(p_2) \end{cases}$$

Finally, the labeling functions for places and transitions are respectively equal to:

$$\psi(x) = \begin{cases} \psi_1(x) & \text{if } \ x \in P_1 \backslash P_1^{E_P} \\ \psi_2(x) & \text{if } \ x \in P_2 \backslash P_2^{E_P} \\ \psi_1(p_1) & \text{if } \ x \equiv (p_1, p_2) \in P_1^{E_P} \times P_2^{E_P} \wedge \psi_1(p_1) = \psi_2(p_2) \end{cases}$$

$$\lambda(x) = \begin{cases} \lambda_1(x) & \text{if } \ x \in T_1 \backslash T_1^{E_T} \\ \lambda_2(x) & \text{if } \ x \in T_2 \backslash T_2^{E_T} \\ \lambda_1(t_1) & \text{if } \ x \equiv (t_1, t_2) \in T_1^{E_T} \times T_2^{E_T} \wedge \ \ \lambda_1(t_1) = \lambda_2(t_2). \end{cases}$$

Place and transition superposition and simplification of two ordinary labeled GSPNs. Given two LGSPN ordinary systems $\mathcal{LS}_1 = (S_1, \psi_1, \lambda_1)$ and $\mathcal{LS}_2 = (S_2, \psi_2, \lambda_2)$, the LGSPN ordinary system $\mathcal{LS} = (S, \psi, \lambda)$:

$$\mathcal{LS} = \mathcal{LS}_1 \bigsqcup_{L_T, L_P} \mathcal{LS}_2$$

resulting from the composition over the sets of (no τ) labels L_T and L_P is defined as follows. Let $E_T = L_T \cap \lambda_1(T_1) \cap \lambda_2(T_2)$ and $E_P = L_P \cap \psi_1(P_1) \cap \psi_2(P_2)$ be the subsets of L_T and of L_P, respectively, comprising place and transition labels that are common to the two LGSPNs, P_1^l (T_1^l) be the set of places (transitions) of $\mathcal{LS}_1$ that are labeled l and $P_1^{E_P}$ ($T_1^{E_T}$) be the set of all places (transitions) in $\mathcal{LS}_1$ that are labeled with a label in E_P (E_T). Same definitions apply to $\mathcal{LS}_2$.

Then: $T = T_1 \backslash T_1^{E_T} \cup T_2 \backslash T_2^{E_T} \cup \bigcup_{l \in E_T} \{T_1^l \times T_2^l\}$, $P = P_1 \backslash P_1^{E_P} \cup P_2 \backslash P_2^{E_P} \cup \bigcup_{l \in E_P} \{P_1^l \times P_2^l\}$, the functions $F \in \{I(), O(), H(), \Pi(t), M^0(), \psi(), \lambda()\}$ are defined exactly as it was made for the last operator, whereas function W(t) is equal to:

$$W(t) = \begin{cases} W_1(t) & \text{if } t \in T_1 \backslash T_1^{E_T} \\ W_2(t) & \text{if } t \in T_2 \backslash T_2^{E_T} \\ W_1(t_1) + W_2(t_2) & \text{if } t \equiv (t_1, t_2) \in T_1^{E_T} \times T_2^{E_T} \wedge \lambda_1(t_1) = \lambda_2(t_2) \end{cases}$$

References

1. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley Series in Parallel Computing - Chichester, 1995.
2. D. Akehurst, G. Waters, P. Utton, and G. Martin. Predictive Performance Analysis for Distributed Systems - PERMABASE position. In *One Day Workshop on Software Performance Prediction extracted from Designs*, Heriot-Watt University, Edinburgh, November 1999.
3. F. Andolfi, F. Aquilani, S. Balsamo, and P. Inverardi. Deriving performance models of software architectures from message sequence charts. In *Proceedings of the Second International Workshop on Software and Performance (WOSP2000)*, pages 47–57, Ottawa, Canada, September 2000. ACM.
4. L.B. Arief and N.A. Speirs. A UML tool for an automatic generation of simulation programs. In *Proceedings of the Second International Workshop on Software and Performance (WOSP2000)*, pages 71–76, Ottawa, Canada, September 2000. ACM.
5. S. Balsamo and M. Simeoni. On transforming UML models into performance models. In *Proceedings of the Workshop on Transformations in UML, ETAPS 2001*, April 7th 2001.
6. S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In *Third International Workshop on Software and Performance (WOSP2002)*, Rome, Italy, July 2002. ACM. To appear.
7. G. Booch, I. Jacobson, and J. Rumbaugh. OMG Unified Modeling Language specification, September 2001. version 1.4.
8. J. Cardoso and C. Sibertin-Blanc. Ordering actions in sequence diagrams of UML. In *Proc. of* 23th *Int. Conf. on Information Technology Interfaces - ITI2001*, Pula, Croatia, 2001.
9. G. Chiola. GreatSPN 1.5 software architecture. Technical report, Università di Torino, April 1991.
10. S. Donatelli and G. Franceschinis. PSR Methodology: integrating hardware and software models. *LNCS 1091, in Proceedings Appl. and Theory of PNs, Osaka, Japan*, pages 133–152, June 1996.
11. R. Eshuis and R. Wieringa. A real-time execution semantics for UML activity diagrams. In Heinrich Hußmann, editor, *Fundamental Approaches to Software Engineering (FASE2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2001.
12. The GreatSPN tool. `http://www.di.unito.it/~greatspn`.

13. Object Management Group. XML Metadata Interchange (XMI) specification, January 2002. version 1.2.
14. D. Harel and A. Naamad. The STATEMATE semantics of the statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.
15. D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999*, pages 331–347. Kluwer, 1999.
16. J. Lilius and I.P. Paltor. The semantics of UML state machines. *Technical report no.273 - Turku Centre for Computer Science, Finland*, May 1999.
17. C. Lindemann, A. Thummler, A. Klemm, M. Lohmann, and O.P. Waldhorst. Quantitative system evaluation with DSPNexpress 2000. In *Proceedings of the Second International Workshop on Software and Performance (WOSP2000)*, pages 12–17, Ottawa, Canada, September 2000. ACM.
18. J. Medina, M. González, and J. M. Drake. MAST-UML: Visual modeling and analysis suite for real-time applications with UML. `http://mast.unican.es/umlmast/`.
19. J. Merseguer, J. Campos, and E. Mena. Analysing internet software retrieval systems: Modeling and performance comparison. *Wireless Networks: The Journal of Mobile Communication, Computation and Information*, 2002. To appear.
20. M.K. Molloy. *Fundamentals of Performance Modelling*. Macmillan, 1989.
21. Rational Software Corporation, 2002. http://www.rational.com.
22. A.J.H. Simons. On the compositional properties of UML statechart diagrams. In *Proceedings of the Rigorous Object-Oriented Methods,ROOM 2000*, January 2000.
23. C. U. Smith. *Performance Engineering of Software Systems*. The Sei Series in Software Engineering. Addison–Wesley, 1990.
24. A. Tsiolakis. Integrating model information in UML sequence diagrams. In *Proceedings GT-VMT2001, Electronic Notes in Theoretical Computer Science*, July 2001.
25. M. Woodside, C. Hrischuck, B. Selic, and S. Bayarov. A wide band approach to integrating performance prediction into a software design environment. In *Proceedings of the 1st International Workshop on Software Performance (WOSP'98)*, pages 31–41, 1998.