From UML Activity Diagrams To Stochastic Petri Nets: Application To Software Performance Engineering*

Juan Pablo López-Grao, José Merseguer, Javier Campos
Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza
Zaragoza, Spain
E-mail: {jpablo,jmerse,jcampos}@unizar.es

ABSTRACT

Over the last decade, the relevance of performance evaluation in the early stages of the software development life-cycle has been steadily rising. We honestly believe that the integration of formal models in the software engineering process is a must, in order to enable the application of well-known, powerful analysis techniques to software models. In previous papers the authors have stated a proposal for SPE, dealing with several UML diagram types. The proposal formalizes their semantics, and provides a method to translate them into (analyzable) GSPN models. This paper focuses on activity diagrams, which had not been dealt with so far. They will be incorporated in our SPE method, enhancing its expressivity by refining abstraction levels in the statechart diagrams. Performance requirements will be annotated according to the UML profile for schedulability, performance and time. Last but not least, our CASE tool prototype will be introduced. This tool deals with every model element from activity diagrams and ensures an automatic translation from ADs into GSPNs strictly following the process related in this paper.

Keywords: UML, software performance, Generalized Stochastic Petri nets, compositionality, activity diagrams, CASE tool, UML Profile for schedulability performance and time specification

1. INTRODUCTION

The Unified Modeling Language (UML) [26] is a semi formal language developed by the Object Management Group [28] 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. Being the objective of our works the performance evaluation [24] of software systems at the first stages of the software development process, as proposed by the software performance engineering (SPE) [30], behavioural diagrams play a prominent role, since they are intended to describe system dynamics. These diagrams belong to five kinds: Use Case diagram (UC), Sequence diagram (SD), Activity diagram (AD), Collaboration diagram and Statechart diagram (SC).

In this paper we explore a possible role for the AD in the SPE process: the description, at a lower level, of specific activities from an SC diagram. Employing the SC (which models the life cycle of the objects in the system) together with the AD allows us to model all the paths in the (potential) system dynamics. Actually, this work fits in a more general SPE approach developed by the authors, that deals with other UML behavioural diagrams: The UC diagram was proposed in [22] to model the usage of the system for each actor; in [21], the SC was addressed (by means of the UML state machines package) to obtain a performance model of a system described as a set of SCs; while in [6], the SD was studied together with the SCs to obtain a performance model representing a concrete execution of the system.

In this environment, we base our interpretation of the AD on the fact that they are suitable for internal flow process modeling, as expressed in [26]. Therefore they are relevant to describe activities performed by the system, usually expressed in the SC as *doActivities* in the states. Other interpretations for the AD propose it as a high level modeling tool, that of the workflow systems [11], but at the moment we do not consider this role in our SPE approach.

In the following we give the big picture of our SPE approach before the inclusion of the AD. First the system is modeled by means of the proposed UML diagrams, and performance requirements are gathered according the UML profile for performance [25]. Since UML defines "informally" their semantics, we propose a method to translate each diagram into a Labeled Generalized Stochastic Petri net (LGSPN), an extension of the well-known GSPN formalism [1], then gaining a formal semantics for them. Afterwards, we give a procedure to compose these LGSPNs, therefore gaining an analyzable model (a performance model) for the system or for a particular scenario (depending on which diagrams have been modeled, as expressed above). Obviously, the translation method implies taking decisions on the interpretation of the diagrams. Finally, the performance models obtained can be analyzed or

*This work has been developed within the project P084/2001 of the Gobierno de Aragón, and the projects TIC2002-04334-C03-02, TIC2003-05226, and TIC2001-1819 of the Spanish Ministry of Science and Technology.

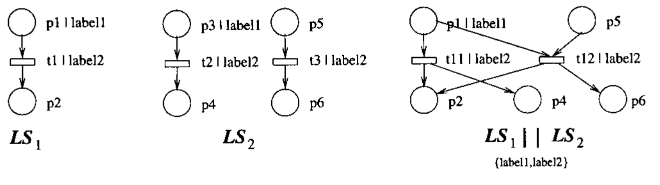

Figure 1: Superposition of places and transitions

simulated using the GSPN tools [13] to obtain performance indices. Note that the LGSPN formalism was chosen for our method due to the number of analysis and simulation tools available for it.

Despite being aware that resource modeling is a fundamental issue in SPE terms, at the moment we take an "infinite resource" assumption. Taking into account that our approach is not meant for real-time domain but to establish primary results in early stages of distributed systems modeling, it could be a reasonable assumption. Since the approach is based on a compositional modelling of components (LGSPNs), the consideration of the use of resources could be included by modelling them (mainly in the deployment diagram) as additional components and by using similar connection rules between these components and the rest of the system.

In this work, and in order to integrate the AD in our approach according to the previously referred interpretation (i.e. activities description), we give a formal semantics to the AD. It is accomplished by translating into an LGSPN each one of the concepts defined in the UML activity graphs package and by composing the resulting LGSPNs, using the operators defined in Appendix A, into a new one that represents the whole AD. Figure 1 depicts an informal representation of the composition (superposition) operator over places and transitions. Finally, it is shown how to compose the LGSPN representing an AD with the LGSPN representing the SCs that use the *doActivity* modeled by the AD. The result of the composition is a new LGSPN that represents a performance model for the system, according to our proposal [20].

Therefore in the final performance model, we obtain a reasonable degree of expressivity to deal with the description and evaluation of the dynamics of large and complex systems, such as distributed systems where we may want to take into account inter-node communication and the occurrence of external events (using SCs, SDs) as well as the internal, concurrent processing within the nodes (using ADs).

Regarding the expressivity enhancement, several reasons back the assertion above. First, it is well-known that ADs are more appropriate for modelling parallelism. Certainly, one can use nested states in SCs, but this is undesirable (even deprecated) and, furthermore, inhibits us from modelling unsafe system behaviour. Additionally, we are unable to model shared methods without ADs: We have not dealt so far with static diagrams (as the class diagram), and thus we cannot define (yet) shared methods by means of class inheritance. However, we can define a method dynamics using an AD, invoking this model from different SCs.

Furthermore, in this paper we briefly overview our prototype tool, which implements our method for ADs: a CASE tool front end is used to design performance annotated models whereas the tool itself constructs their translation into GSPNs. These nets are finally analyzed with a proper performance evaluation tool (namely, GreatSPN [13]).

The rest of the article is organized as follows: Section 2 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. Note that a fairly concise example has been included at the end of the section in order to illustrate this process. Section 5 briefly presents our tool prototype. Section 6 explores the bibliography. Finally, section 7 summarizes the paper and discusses future extensions.

Definitions

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 [9], 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.

2. ACTIVITY DIAGRAMS FOR PERFORMANCE EVALUATION

Activity diagrams represent UML activity graphs and are just a variant of UML state machines (see [26], 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 2). The main goal of ADs is to stress the internal control flow of a process in contrast to SC diagrams, which are often driven by external events.

As our objective is to use ADs to refine *doActivities* in SCs and then to obtain predictive performance measures from the performance model obtained from these diagrams, we need additional modeling information, such as routing rates or the duration of the basic actions. We propose to annotate the AD to gather this information according to the UML profile [25]: 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 ([26], section 3.84), almost every state in an AD should be an action or subactivity state, so almost every

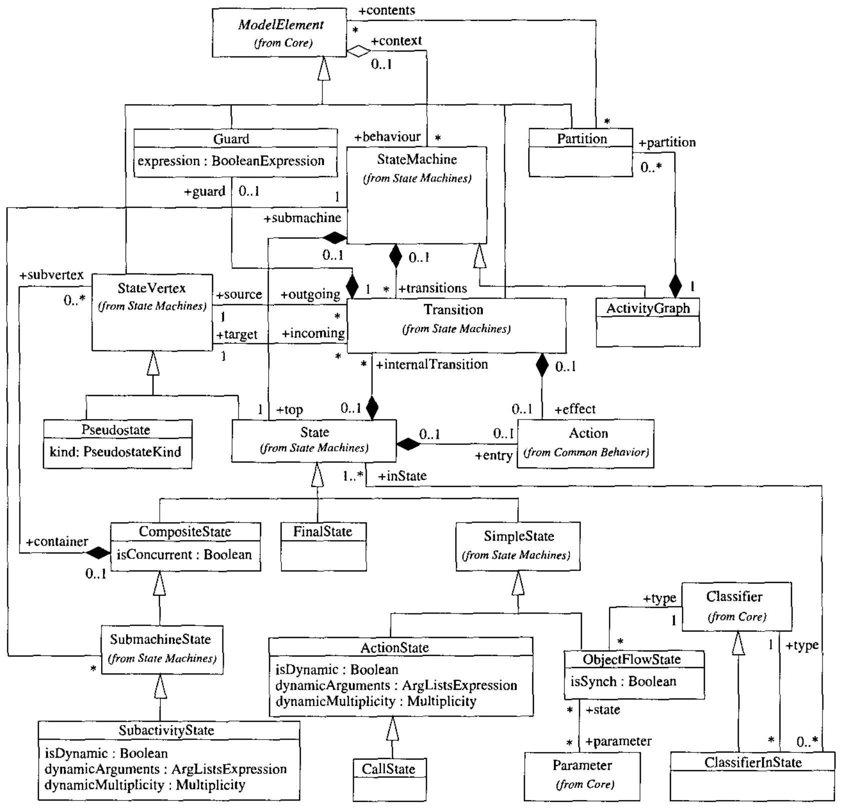

Figure 2: UML Activity Graphs metamodel

transition[1] should be triggered by the ending of the execution of the entry action or activity associated to the state. Since UML is not strict at this point, then the elements from the SMs package could occasionally be used with the interpretation given in [21].

As far as this issue is concerned, our decision is not to allow other states than action, subactivity or call states, and thus to process external events just 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.

2.1 Performance annotations

Concerning performance requirements, as we have expressed above, we consider to gather in the AD routing rates as well as the action durations. Then, from the annotations proposed in the UML profile [25], we consider the stereotype ≪PAstep≫ together with its tag definitions PAprob and PArespTime, that will allow to annotate respectively routing rates and action durations.

Therefore, the format will be: PAprob = P(k) for routing rates (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); and PArespTime = (<source-modifier>,'max',(n,'s.')), or PArespTime = (<source-modifier>,'dist',(n-m,'s.')). Where <source-modifier>::= 'req'|'assm'|'pred'|'msr'; 'dist' is assumed to be an exponential negative distribution and n-m expresses a range of time.

Such annotations will be attached to transitions in order to allow the assignment of different action durations depending on the decision. It implies that the class *Transition* should be included as a base class for the stereotype ≪PAstep≫ in [25].

Time annotations will be allocated wherever an action is executed (outgoing transitions of action states, or outgoing transitions of decision pseudostates with an action state as input) and probability annotations wherever a decision is taken, i.e. in the transition 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 usually has a different semantics (it is never used for modelling the control flow, except in this particular case).

2.2 Proposed translation rules and formal definitions

A brief description of each AD element and their translation to LGSPNs is presented in section 3. Section 4 illustrates the method to compose those LGSPNs to obtain

[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')

the whole model for a specific AD, as well as the method to compose the previous LGSPNs with the LGSPNs of the SCs obtained according to [21], then obtaining a model that comprehends all the possible dynamics for the whole system. The overall method is illustrated in Figure 3

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. Figures 4 and 6 depict the subnets that each kind of transition is translated into.

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. This composition is formally defined in section 3.

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 so-called 'basic' subnets for subactivity states and fork pseudostates, see figures 4 and 6).

Formal definition of the LGSPN subsystems

The formal definition of one of the LGSPN systems shown in Figure 4 is stated below. The rest of the cases in Figures 4 and 6 can be straightforwardly derived from this example, so they will not be explicitly illustrated.

From now onward, we will adopt the Object Constraint Language [26] (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 consider the relation between the classes *Transition* and *Action* and the role *effect* between them, then the image of an instance of class *Action*, through the relation *effect* is denoted as *Action.effect.* Also the attributes of a class A, say at_1, and at_2 are denoted using the dot notation, $A.at_1$, and $A.at_2$.

Also, 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.

A system for an outgoing timed transition *ott* of an action state *AS* (see figure 4, case 1.a) is an LGSPN $\mathcal{LS}_{AS}^{ott} = (S_{AS}^{ott}, \psi_{AS}^{ott}, \lambda_{AS}^{ott})$ characterized by the set of transitions $T_{AS}^{ott} = \{t_1, t_2\}$, and the set of places $P_{AS}^{ott} = \{p_1, p_2, p_3\}$. The input and output functions are respectively equal to:

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

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

$$\Pi_{AS}^{ott}(t) = \begin{cases} 0 & \text{if } t = t_2 \\ 1 & \text{if } t = t_1 \end{cases}$$

$$W_{AS}^{ott}(t) = \begin{cases} r_{ott} & \text{if } \Pi_{AS}^{ott}(t) = 0 \\ p_{cond} & \text{if } \lambda_{AS}^{ott}(t) = cond\_ev \\ 1 & otherwise \end{cases}$$

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} assigns the value of the probability annotation, that is attached to the AD transition *ott* with the format PAprob = p_{cond}. 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 $1/n$, when the time annotation attached to the AD transition is expressed in the format PArespTime = (<source-modifier>,max,(n,'s.')), when it is expressed in the format PArespTime = (<source-modifier>,'dist',(n-m,'s.')), then r_{ott} is equal to $2/(n+m)$.

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

$$\psi_{AS}^{ott}(p) = \begin{cases} ini\_AS & \text{if } p = p_1 \\ execute & \text{if } p = p_2 \\ ini\_nextx & \text{if } p = p_3 \end{cases}$$

$$\lambda_{AS}^{ott}(t) = \begin{cases} cond\_ev & \text{if } t = t_1 \\ out\_lambda & \text{if } t = t_2 \end{cases}$$

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 an 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' ([26], section 3.85). According to this definition and the translation of simple states in SMs [23] we should interpret the action atomic and therefore represent it by an immediate transition within the LGSPN corresponding to the state.

However, for the sake of an easier performance modelling, we will allow here timed actions (i.e., actions with a significant duration). To do so, we will distinguish timed from non-timed outgoing transitions. As explained in section 2, annotations are attached to transitions in order to allow the assignment of different action durations depending on the

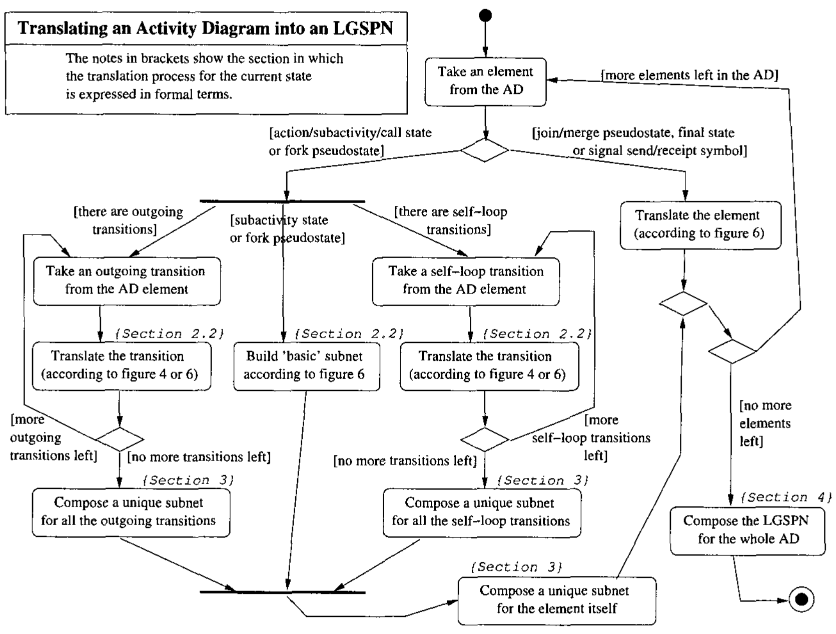
Translating an Activity Diagram into an LGSPN
The notes in brackets show the section in which the translation process for the current state is expressed in formal terms.
Take an element from the AD
[more elements left in the AD]
[action/subactivity/call state or fork pseudostate]
[join/merge pseudostate, final state or signal send/receipt symbol]
[there are outgoing transitions]
[subactivity state or fork pseudostate]
[there are self-loop transitions]
Translate the element (according to figure 6)
Take an outgoing transition from the AD element
Take a self-loop transition from the AD element
{Section 2.2}
{Section 2.2}
{Section 2.2}
Translate the transition (according to figure 4 or 6)
Build 'basic' subnet according to figure 6
Translate the transition (according to figure 4 or 6)
[more outgoing transitions left]
[no more transitions left]
[no more transitions left]
[more self-loop transitions left]
[no more elements left]
{Section 3}
{Section 3}
Compose a unique subnet for all the outgoing transitions
Compose a unique subnet for all the self-loop transitions
{Section 4}
Compose the LGSPN for the whole AD
{Section 3}
Compose a unique subnet for the element itself

Figure 3: The whole translation method for an AD

termination condition (note that, in ADs, outgoing transitions from action states model decision branching). A timed transition in an AD will entail the inclusion of a timed transition (with a rate associated in function of the performance annotation) in the resulting LGSPN. A non-timed transition will result in an immediate transition in the LGSPN model.

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 non-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 non-timed transitions ONJ_n that end in a join pseudostate, s the number of self-loop timed transitions ST_k and s' the number of self-loop non-timed transitions SN_l.

Then for each outgoing or self-loop transition t, we have an LGSPN $\mathcal{LS}^t_{AS} = (S^t_{AS}, \psi^t_{AS}, \lambda^t_{AS})$ as shown in figure 4, 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:

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

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

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

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

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

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 special 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 the LGSPN formalism takes those three steps pointed out 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 4, cases 2.a-2.d, we have a basic LSGPN $\mathcal{LS}^B_{SS} = (S^B_{SS}, \psi^B_{SS}, \lambda^B_{SS})$ and one LGSPN for each outgoing or self-loop transition t, $\mathcal{LS}^t_{SS} = (S^t_{SS}, \psi^t_{SS}, \lambda^t_{SS})$. Therefore, we have $q + r + s + 1$ LGSPN models that need to be combined to get a model of

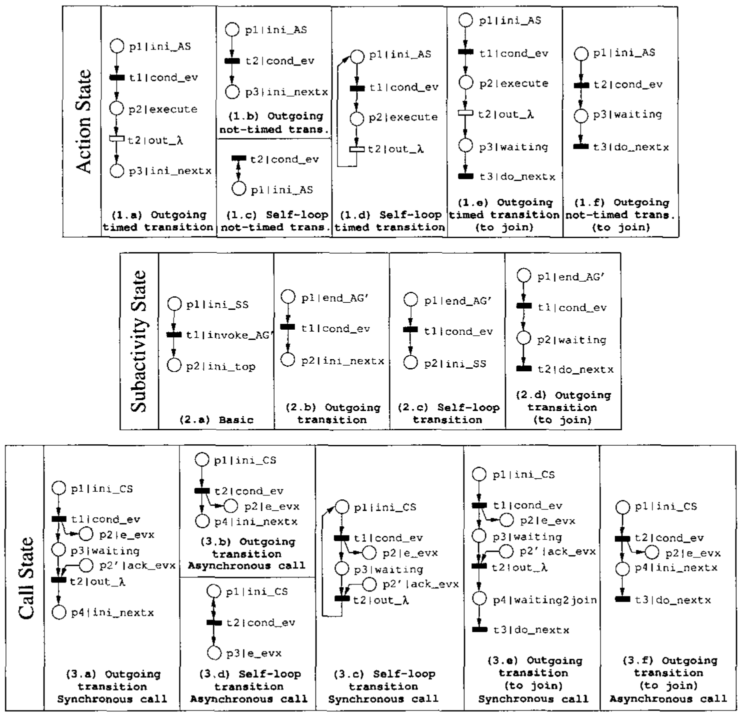
Action State
(1.a) Outgoing timed transition
(1.b) Outgoing not-timed trans.
(1.c) Self-loop not-timed trans.
(1.d) Self-loop timed transition
(1.e) Outgoing timed transition (to join)
(1.f) Outgoing not-timed trans. (to join)
Subactivity State
(2.a) Basic
(2.b) Outgoing transition
(2.c) Self-loop transition
(2.d) Outgoing transition (to join)
Call State
(3.a) Outgoing transition Synchronous call
(3.b) Outgoing transition Asynchronous call
(3.d) Self-loop transition Asynchronous call
(3.c) Self-loop transition Synchronous call
(3.e) Outgoing transition (to join) Synchronous call
(3.f) Outgoing transition (to join) Asynchronous call

Figure 4: Action, Subactivity and Call States to LGSPN

the state SS, $\mathcal{LS}_{SS} = (\mathcal{S}_{SS}, \psi_{SS}, \lambda_{SS})$. The LGSPNs corresponding to each set of kind of transitions are now obtained by superposition:

$$\mathcal{LS}^{O}_{SS} = \underset{Lstvertex^P, end\_AG}{\overset{i=1,\dots,q}{||}} \mathcal{LS}^{O_i}_{SS} \qquad \mathcal{LS}^{OJ}_{SS} = \underset{end\_AG}{\overset{k=1,\dots,r}{||}} \mathcal{LS}^{OJ_k}_{SS}$$

$$\mathcal{LS}^{S}_{SS} = \underset{end\_AG, out\_\lambda, ini\_SS}{\overset{j=1,\dots,s}{||}} \mathcal{LS}^{S_j}_{SS}$$

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 rather 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 consumed by the receiver. In that case, we need a new place and transition in the corresponding LGSPN to model the synchronization (see figure 4, 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, $evx = S.entry.operation \rightarrow occurrence$. Considering this, we have one LGSPN for each outgoing or self-loop transition t, $\mathcal{LS}^{t}_{CS} = (\mathcal{S}^{t}_{CS}, \psi^{t}_{CS}, \lambda^{t}_{CS})$, as shown in figure 4, 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, $\mathcal{LS}_{CS} = (\mathcal{S}_{CS}, \psi_{CS}, \lambda_{CS})$. The LGSPNs corresponding to each set of kind of transitions are now obtained by superposition:

$$\mathcal{LS}^{OS}_{CS} = \underset{Lstvertex^P, Lev^P}{\overset{i=1,\dots,u}{||}} \mathcal{LS}^{OS_i}_{CS} \qquad \mathcal{LS}^{OA}_{CS} = \underset{Lstvertex^P, Lev^P}{\overset{j=1,\dots,u'}{||}} \mathcal{LS}^{OA_j}_{CS}$$

$$\mathcal{LS}^{OJS}_{CS} = \underset{ini\_CS, Lev^P}{\overset{k=1,\dots,v}{||}} \mathcal{LS}^{OJS_k}_{CS} \qquad \mathcal{LS}^{OJA}_{CS} = \underset{ini\_CS, Lev^P}{\overset{l=1,\dots,v'}{||}} \mathcal{LS}^{OJA_l}_{CS}$$

$$[\ \mathcal{LS}_{CS}^{SS} = \underset{ini\_CS,Lev^P}{\overset{m=1,\dots,w}{||}} \mathcal{LS}_{CS}^{SSm} \quad \mathcal{LS}_{CS}^{SA} = \underset{ini\_CS,Lev^P}{\overset{n=1,\dots,w'}{||}} \mathcal{LS}_{CS}^{SAn}$$

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 *(a)* or asynchronous *(b)*:

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

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

3.4 Decisions

Decisions are preprocessed before the AD translation, as it will be mentioned in section 4.1.1. They are substituted by equivalent outgoing transitions on action states (as shown in figure 5), preserving the properties inherent in performance annotations. Therefore, they do not have to be translated. Note that performance annotations in figure 5 do not strictly follow the UML Profile [25], in order to obtain a more compact notation (otherwise, the figure would be rather overloaded).

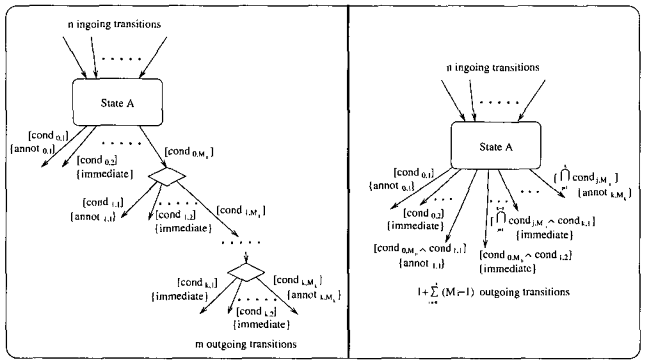

Figure 5: Decision to LGSPN (Pre-transformation)

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 6 (cases 5.a and 5.b) shows the direct translation of the model, $\mathcal{LS}_M$, according to the condition expressed below.

(a) $\mathcal{LS}_M = \mathcal{LS}'_M \iff (PS.outgoing.target \notin Pseudostate \lor PS.outgoing.target.kind \neq join)$ (to join)

(b) $\mathcal{LS}_M = \mathcal{LS}''_M \iff (PS.outgoing.target \in Pseudostate \land PS.outgoing.target.kind = join)$ (not to join)

3.6 Concurrency support items

UML provides two elements to model concurrency in an AD: forks and joins. It is well known that Activity Diagrams were born as a mixture of characteristics from three different sources: the event diagram of Odell, SDL and Petri nets. Forks and joins seem to have been directly inherited from the latter (although there was already some concurrency support in Odell's event diagrams). Translation into LGSPN models is quite simple in both cases.

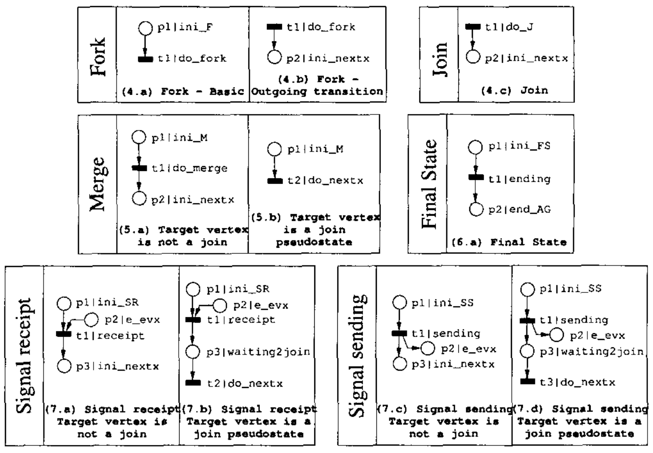

Figure 6: Fork, Join, Merge, Final State, Signal Sending and Signal Receipt to LGSPN

Given a join pseudostate J, it is translated into the labelled system $\mathcal{LS}_J$, shown in figure 6, 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 6, 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:

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

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 an 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 6, 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 fact, they seem to be the clearest evidence of the SDL notation inheritance. 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 the 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 an LGSPN model $\mathcal{LS}_{SIGS}$ is shown in figure 6, cases 7.c-7.d.

- If the symbol is a signal receipt, then let *SIGR* be its pre-assigned identificator. Its translation into an LGSPN model $\mathcal{LS}_{SIGR}$ is shown in figure 6, 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 Constructions not considered yet

Some elements from ADs are not considered as relevant for performance evaluation in the scope of our work; thus they are not translated into LGSPN models. These elements are:

- *Swimlanes*, which are mechanisms to organize visually the states within the diagram, lack a well-defined semantics. In our interpretation, we did not assign them any particular role; and therefore they are not translated. Anyway, we are aware that they could be used to model where the processes are executed, providing then a useful performance information. This possibility should be evaluated as soon as we eliminate our 'infinite resource' assumption.

- *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 LGSPN

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.

4.1.1 Pre-transformations

Before translating the AD into an 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 5 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 (this process is trivial).

- 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).

4.1.2 Translation process

Once pre-transformations are applied we can proceed to translate the diagram into an 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 instances of each AD element type:

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

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

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

$$\mathcal{LS}^{sigse}_{AG} = \underset{Lstvertex^P, Lev^P}{\overset{SIGS\in SignalSendings}{||}} \mathcal{LS}_{SIGS}$$

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

$$\mathcal{LS}_{AG}^{merge} = \mathop{||}\limits_{Lstvertex^P}^{M\in Merges} \mathcal{LS}_M \quad \mathcal{LS}_{AG}^{fork} = \mathop{||}\limits_{Lstvertex^P}^{F\in Forks} \mathcal{LS}_F$$

$$\mathcal{LS}_{AG}^{join} = \mathop{||}\limits_{Lstvertex^P}^{J\in Joins} \mathcal{LS}_J \quad \mathcal{LS}_{AG}^{finst} = \mathop{||}\limits_{end\_AG}^{FS\in FinalStates} \mathcal{LS}_{FS}$$

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

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

It must be noted that the compositional approach does not deal with recursive invocations between activities. 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, $AG_1 = SS'.submachine$. For this kind of situations, we would need coloured Petri nets (with an unbounded number of colours), in order to identify different invocations. Note that the method to combine different activity diagrams (as well as other diagram types) is depicted in section 4.2.

4.1.3 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 compact 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 $\mathcal{LS}_{AS}^{OT}$, 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.

In case an activity (modelled with an AD) is invoked from different states in (one or several) SCs/ADs (by means of *doActivities* or subactivity states, respectively), we must replicate the LGSPN of the corresponding AD (one AD per invocation). Otherwise, undesirable situations could happen which would degrade the performance evaluation results (the resulting Markov chain does not capture properly the system behaviour). A different solution for this issue would be using coloured Petri nets (applying a different colour for each *doActivity*/subactivity invocation). Obviously this implies that the activities invocation graph must be acyclic; hence neither support for Knuth's coroutines nor recursion are offered, as already commented in section 4.1.2.

Let us suppose the replication process has already been executed. 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):

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

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 $\mathcal{LS}_{ad}$:

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

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 7. The reader is encouraged to check out [18] to obtain a wider vision of our proposal under the prism of a more complex case study. Here we will focus on a small portion of the system modelled in that paper. The example is quite representative as it formalizes the POP3 protocol, a well-known instance of the client-server paradigm in which nodes perform time-relevant internal processing while there is some intercommunication between them.

More concretely, we built the model to evaluate the behaviour of a mail client using the referred protocol. Thus, we used three SCs (to respectively model the client, server and user dynamics), one SD (to model the use case we wanted to analyze) and one AD (to model the internal processing in the server for the authentication phase), which is shown in Figure 7. Additionally, we include there the LGSPN obtained by applying the proposal described in this paper to this last diagram.

The referred AD represents how the login (authentication) process is performed at the server side. The server waits for a username from the client, and then for a password; if both match up with those held in the local machine, the maildrop is locked and the server ends up the authentication phase. On the other hand, if anything fails it returns a error status message and returns to the initial state. Note that we estimated some (hypothetical) event probabilities and task durations and annotated them as tagged values. Those annotations will allow us to perform some quantitative (performance) analysis over the model.

It must be remarked that the AD is just a part of the whole system description. That results in the lack of tokens in the initial marking of the net in Figure 7. The reader is refered again to the paper [18] to understand how the Petri net for the whole system is composed.

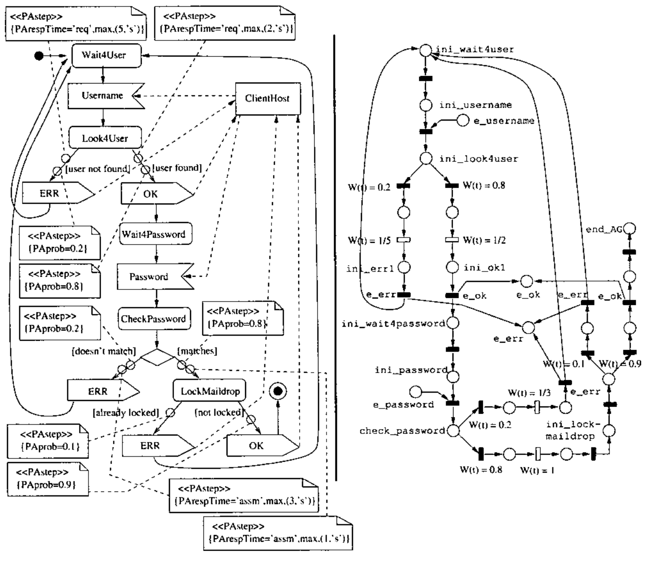

Figure 7: POP3 Protocol (Server side): Authentication

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 module in Java^{r} that translates the SC as proposed in [21] and the AD as proposed in this work. The module has been incorporated to the ArgoUML CASE tool [4]: The input diagrams are provided in the XMI [14] format, while the output produces a LGSPN in the file format [8, 13].

The module also implements the composition of the resulting LGSPN models for the SCs and ADs as explained this work. Then, the resulting LGSPN, that represents a performance model for the modeled system, is directly processable by the GreatSPN tool [13] and it has full capability to make quantitative analysis and obtain performance rates.

Although ArgoUML does not allow a full exploitation of all the expressivity in the SC and AD that we have dealt with (since it does not support every model element), our module provides full capability to translate them. Therefore, those model elements not contemplated by ArgoUML should be written by the modeller directly in the XMI file (or simply using a different front-end CASE tool). Note that this limitation is introduced by the current state of the ArgoUML tool.

Finally, it must be noted that an special effort has been made to obtain highly-legible GreatSPN nets, avoiding the superimposition of places and transitions in the generated GreatSPN files.

6. RELATED WORK

Although there are several works devoted to obtain formal models from the UML SC [16, 15, 29] or the UML SD [31, 7, 3], some of them with performance evaluation purposes, the AD has not been studied yet so intensively. However, we would like to remark two significant works, which will be reviewed below. The first one (Eshuis et al.) is concerned with semantical issues of the AD, while the second one (Petriu et al.) deals with usign ADs for performance evaluation on stochastic models.

One of the main challenges in adapting UML diagrams to performance evaluation purposes is choosing an appropriate formal semantics. I.e., neither too restrictive (allowing the modeler a good degree of expressivity) nor too permissive. After all, UML's *informal* semantics should be respected; among other reasons, because communication fluency between modelers is a very basic, strong SPE principle.

UML 1.5 defines AD semantics in terms of SCs. That is subject to change in UML 2.0, which will define a (entirely new) token-based semantics. An interesting contribution to the semantics discussion can be found in [10]. Eshuis et al. had previously defined a step-based, STATEMATE-like, semantics [11] for ADs. In the former cite, the authors discuss the (un)suitability of Petri nets for workflow modelling, in contrast to their formalized ADs. The reasoning is well justified under the light of the application field, as these may be more appropriate for modelling reactive systems (i.e., dependent on the environment) as common workflow processes.

Here we define AD semantics in terms of (labelled) GSPNs. We do not strive for reactive systems since, for now, we strictly utilize ADs for modelling processes not dependent on external events, as the UML specification [26] suggests. Eshuis semantics are aimed to business modelling. Meanwhile, we apply ADs to describe method invocations internals, especially when complex concurrent behavior must be depicted. Needless to say, this is one of the basic roles defined by OMG for the AD.

Due to the nature of this application, we are (theoretically speaking) closer to OMG's perspective, when defining a token-game semantics in the UML 2.0 final draft, than to Eshuis step-based semantics. That is not very exact either, as there are rather profound revisions in the AD semantics (e.g., it seems that outgoing transitions from action states will have now a fork-like semantics, instead of conditional branching-like). Moreover, our interpretation is stochastic, *not exactly* non-deterministic as in plain Petri nets, so as to allow performance evaluation. But we share a focus on the modelling of active systems, while we allow complex, parallel, and even unsafe, behaviors.

An interesting work has been developed in [27], where activity diagrams are translated into layered queue networks (LQN) using a graph grammar based transformation. A graph grammar is a set of production rules that generates a language of terminal graphs and produces non terminal graphs as intermediate results. A production rule is applied to the abstractions that represent the activity diagram, then the activity diagram graph is parsed to check its correction and to divide it into subgraphs that correspond to the LQN elements. As it can be seen the approach to formalize activity diagrams is absolutely different from ours, which is based in the composition of the submodels obtained for each abstraction.

Concerning our tool, it is difficult to make a reasonable comparison because to our knowledge there exist six tools [5, 2, 19, 27, 12, 17] for performance evaluation based on UML, but only the last one uses stochastic Petri nets as performance model. Besides, their model semantics and supported diagrams strongly differ from our approach. DSPNExpress2000 [17], syntactically speaking, seems to allow only the modelling of simple SCs. In SimML [5], simulation queuing networks models [24] 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. Finally, Gilmore et al. [12] employ class and collaboration diagrams to obtain analyzable stochastic process algebra models (namely PEPA models). It is interesting to note that the supporting software architecture for Gilmore's proposal includes model checking facilities.

7. CONCLUSIONS

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

- We have incorporated the AD into our SPE approach with an specific role: modelling the *doActivity* concept of the SCs. We have found that under this role, the AD is a tool to gather performance requirements: routing rates and actions duration. The annotations are proposed according to the UML profile [25].

- 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 corresponding *doActivity*, thus gaining an analizable model for the system.

- 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. Moreover, we have recalled an example [18] in the client-server paradigm where the presented approach was successfully applied.

- A Java^r module has been incorporated to the ArgoUML CASE tool. It allows to translate all the elements in UML SC and ADs notation as proposed by our approach. Performance annotations can be introduced to produce a LGSPN model, representing the system, that can be analyzed by the GreatSPN tool [13], therefore it is possible to obtain performance measures in the steady or transient state. The processing of XMI files as input by our module ensures compliance with current standards.

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 solve the problem of 'uninterruptable' activities due to the use of action states.

- Extension of the Java^r module to support UCs and SDs in order to increase the expressivity at system description.

APPENDIX

A. FORMAL DEFINITION OF COMPOSITION OF LGSPNS

A.0.0.1 Place and transition superposition of two ordinary LGSPNs..

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 exactly as in our previous works. We encourage the reader to check out any of the following references for further information: [6, 20]. Nonetheless, figure 1 depicts informally the semantics of the superposition operator (that should be sufficient for a basic comprehension).

A.0.0.2 Place and transition superposition and simplification of two ordinary LGSPNs..

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 $\mathcal{LS} = (S, \psi, \lambda)$:

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

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

Then P, T, and the functions $F \in \{I(), O(), H(), \Pi(), M^0(), \psi(), \lambda()\}$ are defined exactly as it was made for the previous operator (| |), whereas function $W()$ is equal to:

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

B. 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 *Proc. of* 2^{nd} *Int. Workshop on Software and Performance (WOSP2000)*, pages 47–57, Ottawa, Canada, September 2000. ACM Press.

[4] ArgoUML project. `http://argouml.tigris.org/`.

[5] L.B. Arief and N.A. Speirs. A UML tool for an automatic generation of simulation programs. In *Proc. of* 2^{nd} *Int. Workshop on Software and Performance (WOSP2000)*, pages 71–76, Ottawa, Canada, September 2000. ACM Press.

[6] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In *Proc. of* 3^{rd} *Int. Workshop on Software and Performance (WOSP2002)*, pages 35–45, Rome, Italy, July 2002. ACM Press.

[7] J. Cardoso and C. Sibertin-Blanc. Ordering actions in sequence diagrams of UML. In *Proc. of* 23^{th} *Int. Conference on Information Technology Interfaces (ITI2001)*, Pula, Croatia, 2001.

[8] G. Chiola. GreatSPN 1.5 software architecture. Technical report, Università di Torino, April 1991.

[9] S. Donatelli and G. Franceschinis. PSR Methodology: integrating hardware and software models. *LNCS 1091, in Proc. Application and Theory of Petri Nets*, pages 133–152, June 1996.

[10] R. Eshuis and R. Wieringa. A comparison of Petri net and activity diagram variants. In Reisig Weber, Ehrig, editor, *Proc. of 2nd Int. Collaboration on Petri Net Technologies for Modelling Communication Based Systems*, pages 93–104. DFG Research Group "Petri Net Technology", September 2001.

[11] R. Eshuis and R. Wieringa. A real-time execution semantics for UML activity diagrams. In Heinrich Hußmann, editor, *LCNS 2029, in Fundamental Approaches to Software Engineering (FASE2001)*, pages 76–90. Springer-Verlag, 2001.

[12] S. Gilmore and L. Kloul. A unified approach to performance modelling and verification. Paper presented at Dagstuhl seminar on "Probabilistic Methods in Verification and Planning".

[13] The GreatSPN tool. `http://www.di.unito.it/~greatspn`.

[14] Object Management Group. XML Metadata Interchange (XMI) specification, January 2002. version 1.2.

[15] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. of* 3^{rd} *Int. Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS99)*, pages 331–347, Florence, Italy, February 1999. Kluwer.

[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 *Proc. of* 2^{nd} *Int. Workshop on Software and Performance (WOSP2000)*, pages 12–17, Ottawa, Canada, September 2000. ACM Press.

[18] J. P. López-Grao, J. Merseguer, and J. Campos. Performance engineering based on UML & SPN's: A software performance tool. In *Proc. of* 7^{th} *Int. Symposium On Computer and Information Sciences (ISCIS2002)*, pages 405–409, Orlando, Florida, October 2002. CRC Press.

[19] 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/`.

[20] J. Merseguer. *Software Performance Engineering based on UML and Petri nets.* PhD thesis, Departamento de Informatica e Ingenieria de Sistemas. Universidad de Zaragoza, Spain, March 2003.

[21] J. Merseguer, S. Bernardi, J. Campos, and S. Donatelli. A compositional semantics for UML state machines aimed at performance evaluation. In M. Silva, A. Giua, and J.M. Colom, editors, *Proc. of the* 6^{th} *Int. Workshop on Discrete Event Systems (WODES2002)*, pages 295–302, Zaragoza, Spain, October 2002. IEEE Computer Society Press.

[22] J. Merseguer and J. Campos. Exploring roles for the UML diagrams in software performance engineering. In *Proc. of* 3^{rd} *Int. Conference on Software Engineering Research and Practice (SERP'03)*, pages 43–47, Las Vegas, USA, June 2003. CSREA Press.

[23] 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*, 9(3), May 2003.

[24] M.K. Molloy. *Fundamentals of Performance Modelling.* Macmillan, 1989.

[25] Object Management Group, http:/www.omg.org. *UML Profile for Schedulabibity, Performance and Time Specification*, March 2002.

[26] Object Management Group, http:/www.omg.org. *OMG Unified Modeling Language Specification*, March 2003. version 1.5.

[27] D. Petriu and H. Shen. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In Tony Field, Peter G. Harrison, Jeremy Bradley, and Uli Harder, editors, *LNCS 2324, in TOOLS 2002*, pages 159–177, London, UK, April 2002. Springer-Verlag.

[28] Object Management Group. http://www.omg.org.

[29] A.J.H. Simons. On the compositional properties of UML statechart diagrams. In *Proc. of Rigorous Object-Oriented Methods (ROOM2000)*, January 2000.

[30] C. U. Smith. *Performance Engineering of Software Systems.* The Sei Series in Software Engineering. Addison–Wesley, 1990.

[31] A. Tsiolakis. Integrating model information in UML sequence diagrams. In *Proc. of* 2^{nd} *Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT2001).* Electronic Notes in Theoretical Computer Science. Springer-Verlag, July 2001.