From UML Sequence Diagrams and Statecharts to analysable Petri Net models

Simona Bernardi
Dipartimento di Informatica
Università di Torino, Italy
bernardi@di.unito.it

Susanna Donatelli
Dipartimento di Informatica
Università di Torino, Italy
susi@di.unito.it

José Merseguer*
Dpto. de Informática e
Ingeniería de Sistemas
University of Zaragoza, Spain
jmerse@posta.unizar.es

ABSTRACT

In this paper we study the use of Sequence Diagrams and Statecharts of the Unified Modeling Language (UML) for the validation and the performance evaluation of systems. We assume that the system is specified as a set of Statecharts and that Sequence Diagrams are used to represent "executions of interest". UML lacks a formal semantics and hence it is not possible to apply, directly, mathematical techniques on UML models for system validation. To reach this goal we propose an automatic translation of Statecharts and Sequence Diagrams into Generalized Stochastic Petri Nets, and a composition of the resulting net models suitable for reaching a given analysis goal. The translation into Petri nets is based on the abstract syntax of the UML Collaborations and of the State Machines packages (from which Sequence Diagrams and Statecharts are derived) and the construction of the complete model heavily uses compositionality.

Keywords

UML, GSPN, compositionality, modeling.

1. INTRODUCTION

The Unified Modeling Language [19] (UML) is a semi formal language widely used in the software development process for the specification of systems based on the object-oriented paradigm. UML provides several types of diagrams which allow to capture different aspects and views of the system. The abstract syntax of each modeling notation is described by means of *metamodels*, which are graphically represented with Class Diagrams, together with textual comments.

*This work has been supported by the Gobierno de Aragón within the project P084/2001.

We assume that a system is described by a set of Statecharts (that describe the behaviour of classes that is common to all instances of that class) and that Sequence Diagrams are used to emphasize specific patterns of interaction among Statecharts. Two levels of modeling can be adopted: class level and instance level. At class level the participants of the interaction are generic instances of classes, i.e., generic objects, that communicate by exchanging a set of messages. At instance level, participants are instead specific objects that communicate by dispatching stimuli which conform to messages belonging to the interaction. It is important to clarify from the very beginning that we place this work in the context of a system being described at class level, that can be considered preliminary to the description of objects and interactions among objects that is also allowed by the UML diagrams and that provides a better description of the system.

Our goal being validation and performance evaluation, a number of other diagrams could be taken into consideration: in particular deployment diagrams that can be used to describe the physical allocation of objects onto machines, but we do not consider them for the moment since this makes sense only at the object level.

The proposed analysis is based on the combined use of Sequence Diagrams and Statecharts so as to be able to assess what is the consistency among the two descriptions, and to stochastically evaluate those behaviours of the system that are consistent with the patterns of interaction described by a Sequence Diagram. But what is consistency between Sequence Diagrams and Statecharts? A very nice analysis of the role of Sequence Diagrams with respect to system behaviour can be found in [13] where a number of questions are enounced as a basis for a classification actually in the context of Message Sequence Charts but the adaptation to our context is straightforward and leads to the following questions:

Q1 What does a Sequence Diagram mean with respect to the set of system components described by the Statecharts? Possible answers in our context are: components of the Sequence Diagram are all and only those of the Statecharts, or are a proper subset.

Q2 How complete is the information of a Sequence Diagram with respect to the effective exchange of message described by the Statechart?

Different answers to the above questions lead to different analysis goals, and to different composition of Statecharts with Sequence Diagrams as we shall see in Section 3.

In this paper we reduce Sequence Diagrams and Statecharts into a common framework, that of Petri Nets. The class of Petri Nets we deal with is the class of labeled ordinary Generalized Stochastic Petri Nets (GSPNs), and the approach adopted for the translation of both UML Statechart and Sequence Diagram into GSPNs is compositional.

This paper builds on the formal translation of Statecharts into GSPNs that is given in [18] (that is recalled in this paper only in an informal manner) by proposing a formal translation of Sequence Diagrams into GSPNs. The standard UML still does not have a formal semantics for Statecharts and for Sequence Diagrams (although there are a number of attempts along this line, coordinated by the *precise UML* working group [21]) and there are some ambiguities for which we have made a specific choice, and also some characteristics that we have decided not to translate, so, in a way, the translation proposed here provides a formal semantics for a subset of the Sequence Diagrams (and the same is true for Statecharts in [18]). But a central contribution of the paper is to establish relationships between Sequence Diagrams and Statecharts according to the UML metamodel, that allow to define a rule for Statecharts and Sequence Diagrams composition that has its root in the UML metamodel and that produces a single executable model.

The idea of producing, more or less automatically and systematically, validation and evaluation models from UML-based system descriptions it has been around for a while, so there is a rather large body of literature on the topic: since the main characteristics of our approach are that the translation is systematic and therefore easily automatizable, that consistency between Statecharts and Sequence Diagrams is taken into account and that the translation is meant also for performance evaluation purposes, we review only that part of the literature that is more related to the above three topics.

Statecharts and Sequence Diagrams are used in a combined manner in [15] where the goal of the analysis is to check temporal logic formulas over a Statechart-based description of the system, and Sequence Diagrams are used "only" to show the counterexample produced by the model checker. The paper uses the translation from Statecharts to a term rewriting system that is given in [16].

In [6] a formal translation of Sequence Diagrams into Petri Nets is proposed, based on the UML *Collaborations* package metamodel, using the associations defined on the *Message* metaclass. The resulting Petri Net allows to check if the Sequence Diagram has been well specified.

Consistency between Sequence Diagrams and Statecharts is instead extensively treated in [24], that proposes a method for the verification of consistency and completeness of UML Class, Sequence Diagrams and Statecharts. The work in [14] proposes an analysis of consistency of the timing constraints that can be associated with Sequence Diagrams. The approach followed is deterministic, event occurrences are simple real variables (not random), and no relationships with other UML diagrams is considered.

A good survey of the different approaches for performance evaluation based on UML diagrams can be found in [3], although in the context of software architectures. None of the papers discussed in [3] mixes the use of Sequence Diagrams and Statecharts as in our proposal, but some of them share many similarities with our approach: in [2] an algorithm is given to obtain queuing networks models for performance evaluation purposes from Message Sequence Charts (MSCs) and the main difference with our work is that we embed the Sequence Diagram (or MSC) into a complete system description given by the Statecharts. In [7] instead the Sequence Diagram is used in conjunction with Use Cases and the Deployment Diagram inside a methodological framework to obtain queuing network models for performance evaluation. An algorithm is given to obtain an execution graph from the Sequence Diagram. Again, the role played by the Statecharts is not defined, but the use of Deployment Diagrams allows to represent hardware restrictions and it overcomes the infinite resources assumption that we implicitly do in our approach. In both [2] and [7] the analysable model is a queueing network, which is a positive point for stochastic analysis, but it is not a suitable choice when also the check of qualitative properties is part of the analysis.

A work that is strictly related to ours is [12] in which Petri net models are produced starting from UML diagrams, but the construction in [12] is described only at an intuitive level, through an example and no translation procedure is described.

In [5] UML stereotypes are used to enrich UML diagrams with dependability aspects that are then exploited to build generally distributed stochastic Petri net models. The focus is not on an automatic translation, but on the capturing of the dependability aspects from the UML diagrams.

Paper [17] is instead a case study of the combined use of Sequence Diagrams and Statecharts to produce a GSPN model for performance evaluation purposes, and in this paper we build on the experience gained on that case study.

As we said before, the proposed translation heavily uses compositionality. Compositionality is a widely accepted and may be less widely used practice in modeling with Petri nets. Main reference on the topic is the Box-calculus approach [4], while there are a number of works also for GSPN composition (in the context of tools like UltraSAN [25], HiQPN [11], and GreatSPN [10]). The composition used in this paper is superposition over transitions of equal label, as defined in [9], and over places of equal label. Both operators are available in the software program algebra [22], that is part of the current GreatSPN distribution.

The proposed translation does not take into consideration the full richness of Sequence Diagrams and Statecharts. The Sequence Diagrams feature not considered are iterations (that could be easily included), and timing constraints (since a stochastic approach to evaluation is not meant to model and/or prove timing constraints). The *State Machine* metamodel elements not considered are: composite states, submachine states, stub states and pseudo-states (except initial pseudo-states), and guards; the choice of not translating guards is based on our choice of performing a static analysis [23], that, as such, does not consider variables, and therefore guards that are resolved probabilistically.

The paper is organized as follows: Section 2 recalls the translation of Statecharts into labeled GSPN models given in [18] and formally defines the translation of Sequence Diagrams into labeled GSPN models. Section 3 discusses what are the goals of the joint analysis of Statecharts and Sequence Diagrams, and defines two composition rules to be used to produce the global model, depending on the analysis goals. Section 4 applies the concept of the previous sections to the *watchdog* example. Section 5 concludes the paper by discussing the limits of the current approach and the future extensions.

1.1 Definition and notation

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

A labeled GSPN (LGSPN) is then a triplet $\mathcal{LS} = (\mathcal{S}, \psi, \lambda)$, where $\mathcal{S}$ is a GSPN system, as defined above, $\lambda : T \to L^T \cup \tau$ is the labeling function that assigns to a transition a label belonging to the set $L^T \cup \tau$ and $\psi : P \to L^P \cup \tau$ 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," not visible from the other components. Note that, with respect to the definition in [9], 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. TRANSLATIONS

UML behavioral diagrams are used to model the dynamic behavior of a system; in particular, Statecharts (SCs) allow to specify the behavior of individual entities of the system, such as the class objects, and the Sequence Diagrams (SDs) represent patterns of interaction among the objects. The UML definition document [19] describes the abstract syntax of each modeling notation by means of Class Diagrams, i.e., metamodels, together with textual comments. The static semantics is given as a set of Well Formedness Rules expressed in OCL [19]; the dynamic semantics is given, instead, as English text. The approach adopted for the formalization of SCs and SDs consists in translating the two UML notations into LGSPNs separately, starting from the metamodels of the UML *State Machines* and *Collaborations* packages which describe informally the semantics underlying the SCs and the SDs, respectively. It is important to observe that: *even though each UML notation is translated in a separate way, the reference to the UML metamodels accounts for the relations that exist among the SCs and the SDs, since the two metamodels contain common metaclasses.*

Figure 1 shows a partial view of the *State Machines* and of the *Collaborations* metaclasses, together with several metaclasses belonging to other UML packages (i.e., *Core* and *Common Behavior* packages) which are related to the formers, and it has been extracted from [19]. Each metaclass is depicted as a box divided into two parts: in the upper part the name of the metaclass is shown, while in the lower part, between round brackets, it is specified the name of the package the metaclass comes from. The metaclasses that are directly related to each other are connected either with a line or with an arrow (characterized by a particular arrowhead) depending on the type of association is defined between them.

We adopt 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 to have 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 $rel : D \to C$, is denoted as $d.rel$.

For simplicity we assume that the scope of names of actions, events, messages, etc. are global, but this is not a restriction as there are no real ambiguities (e.g., if two actions have the same name in two different SCs it is enough to prefix it with the name of the state machine to which the action belongs).

Assuming that the behavior of each participant to an interaction modeled by a SD is represented by a SC, the following consistency checks on the underlying semantics have to be satisfied.

For each couple of sender and receiver in the SD there exists a unique couple of SCs that model their corresponding behavior:

Assumption 1. Let $m \in i.message$ be a message exchanged by two participants $r_l = m.sender$ and $r_j = m.receiver$ in the interaction i modeled by a SD sd_i. Then $\exists!\ sm_l, sm_j \in StateMachine$ such that:

1. $sm_l = r_l.available\ Contents.behavior$, and
2. $sm_j = r_j.availableContents.behavior$.

In the state machine of the sender there exists, at least, an action that causes the sending of the message and, in the state machine of the receiver, there exists at least a transition that models the response to the event representing the reception of the message.

Assumption 2. Let $m \in i.message$ be a message exchanged between the two previous state machines sm_l, sm_j and $actx = m.action$ the action that causes the message to be sent. Then:

1. $actx \in (sm_l.transition.effect \cup sm_l.states.entry \cup sm_l.states.exit)$, where

   $sm_l.states = components(sm_l.top)$[1]; and
2. $\exists tr \in sm_j.transition$ such that $(tr.trigger).signal = actx.signal$ or $(tr.trigger).operation = actx.operation$

2.1 From UML Statechart to LGSPN model

A state machine $sm \in StateMachine$ is basically characterized by states and transitions. In particular, a transition $tr \in sm.transition$ represents a complete response of sm to a particular event instance $tr.trigger \in Event$. Firing of a transition provokes an action $tr.effect \in Action$ to be performed. An actions $actx$ may or may not provoke one or more event (for example according to the chain $actx.signal.occurence$), but for simplicity of the example we assume that an action can cause at most an event.

Metaclasses which are related to both *Collaborations* and *State Machines* packages, are *Model Element* and *Action*

[1]Relation *components* applied on the top-level state $sm_l.top$ of the state machine sm_l, gives the set of all the simple and the composite states of sm_l:
$components(s) = \{s' \in (SimpleState\ or\ CompositeState) \mid s' \in s.subvertex\ or\ (s' \in components(s'')\ and\ s'' \in components(s))\}$.

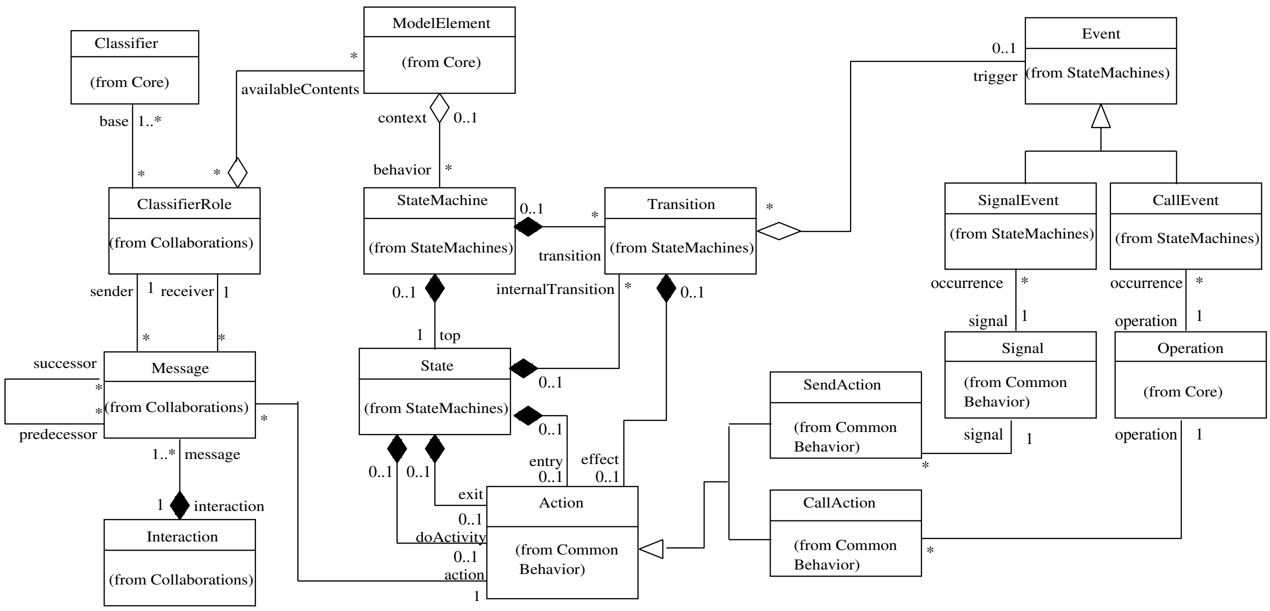

Figure 1: Partial view of *Collaborations* and *State Machines* metamodels

classes (see Figure 1). The action class is particularly interesting for the translation. It belongs to the *Common Behavior* package and its instances represent executable statements that form abstractions of computational procedures. Execution of an action results in a change of state in the model and can be realized by sending a message to an object or by modifying a link or a value of an attribute.

The class Action is what relates states and transitions of SC to messages in the SD (and in general in interaction diagrams). Relevant associations involving the *Action* class are: *action*, that given an instance $msg \in Message$, associates the action $msg.action \in Action$ which causes either a signal or a operation to be invoked; *effect*, that specifies for a given transition $tr \in Transition$ of a state machine, the action to be performed when tr fires; *signal*, defined on the subclass *SendAction* that associates to a send action $act \in SendAction$ the signal which will be invoked when act is executed; *operation*, defined on the subclass *CallAction* that associates to a call action $act \in CallAction$ the operation which will be invoked when act is executed.

Actions are characterized by the attribute $isAsynchronous$ that allows to specify if the dispatched stimulus is asynchronous or not, synchronous meaning that the action will not be completed until the event eventually generated by the action through the chain "call action - operation - call event - event" (or through the chain "send action - signal - signal event - event") is not received/consumed by another SC.

Figure 2 shows a portion of an SC with three states and two transitions among states. The unit of translation is a state with its outgoing transitions: the right portion of the figure is the GSPN translation of the state Nl of the depicted SC and of its two outgoing transitions.

Assuming the reader knows the English-based semantics of SC, we only comment on the choices underlying the translation. We first comment the net assuming that all actions are asynchronous, that, at picture level, amounts to disregarding the portions in dotted boxes. Net objects are denoted as $name|label$ where $name$ is the name of the object and $label$ is the label, and τ labels are omitted. Places with a name starting with mbx represents event queues (observe that no policy is associated to the place, but actually no policy is specified in the SC definition either, apart for the choice of the term "queue"). These places are labeled as e_ev, where ev is the name of the corresponding event. Transitions that represents actions are labeled with the name of the corresponding action. Actions may or may not generate events, and this is represented by the presence of the dotted arc from action transitions to event places. Places labeled as ini_Nl, ini_Ml and ini_Ol represent the initial place of the SC states Nl, Ml and Ol, respectively. Observe that tokens are added to event places due to the execution of an action, while tokens are removed from an event place due to a change of state (an SC transition).

A slightly more complex translation is needed for actions that are synchronous (that is to say $actx.isAsynchronous = false$). Indeed a synchronous action is blocked until the generated event is not removed from the queue, but the problem we have encountered is that $isAsynchronous$ is an action attribute, but there is no way to determine whether an event has been generated by a synchronous action. We have therefore translated synchronous actions as depicted in the dotted box portion, where each transition of label $actx$ is substituted by a sequence transition - place - transition, transitions being labeled S_actx and E_actx, moreover for each event evw generated by the action an acknowledge place of label ack_evw is added: the place is part of the input set of the transition of label E_actx, so that the action can complete only when an acknowledge is received. The problem is now to decide which transitions are going to put tokens into the acknowledge place: since an SC that consumes an event does not know whether it was generated by a synchronous action or not the only possibility is that each transition that consumes an event evw (that is to say transitions of label e_evw) puts a token into the place of label ack_evw.

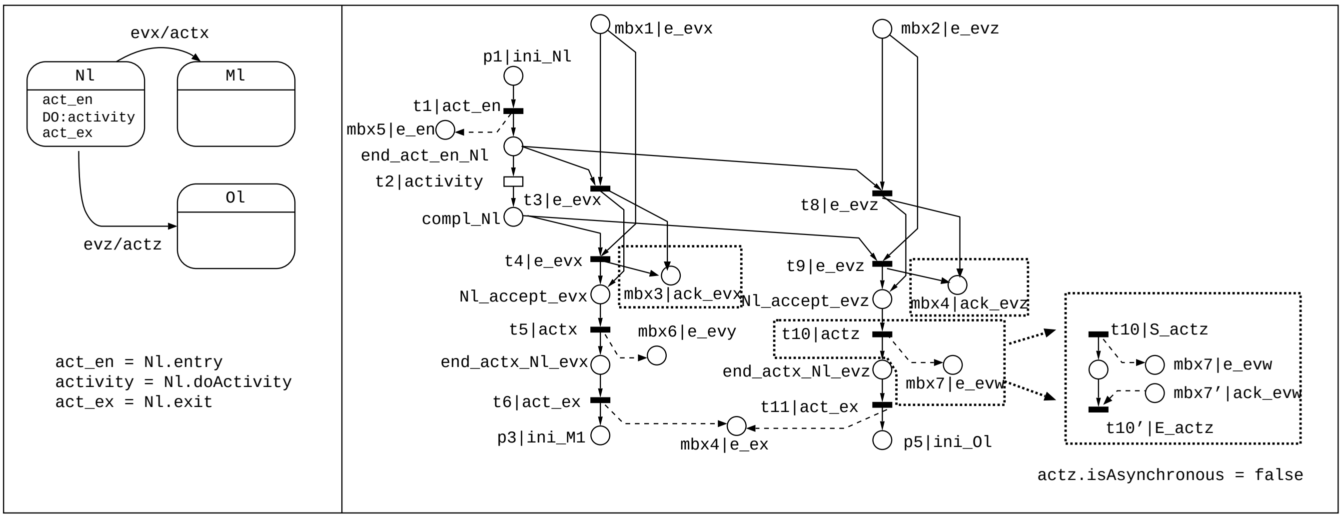

Figure 2: Translation of a SC into labeled GSPNs.

The final model of the SC is obtained by composing the unit models over places whose labels correspond to event names, event acknowledges, if any, and state names. The place corresponding to the initial SC state is the only one marked. A formal definition of the translation can be found in [18].

2.2 From UML SD to LGSPN model

UML SDs, as well as Collaboration Diagrams, are used to describe how a set of objects of the system interact in order to carry out a specific task and they are characterized by the semantics defined in the UML *Collaboration* package. While Collaboration Diagrams emphasize the set of participants, their roles and their relationships within a collaboration, SDs show the temporal execution of communication patterns. It is worth to notice that, although we focus on the translation of SDs, the formalization of the *Collaboration* package allows to obtain the translation of both Collaboration Diagrams and SDs. With reference to the metamodel of Figure 1, the SD sd_i of Figure 3 represents an interaction instance i of the $Interaction$ metaclass.

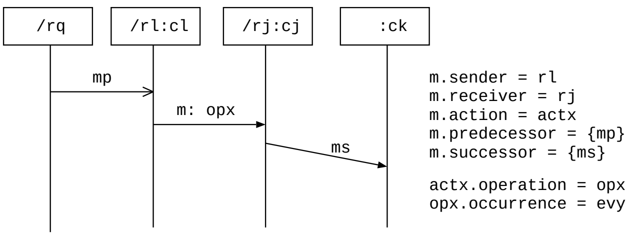

Figure 3: Example of a Sequence Diagram.

The set of messages belonging to an interaction are partially ordered based on which execution thread they belong to. Within each thread of control messages are sent in sequential order while messages belonging to different threads may be sent in parallel or in an arbitrary order. In Figure 3 one thread of control is represented. A partial order relation among messages is derived from the association *predecessor* (or its inverse association *successor*) defined on the $Message$ metaclass: it associates to each message the set of messages that must be completed before the former may be executed.

In SDs, since the vertical dimension represents time, the *predecessor* association is established between messages corresponding to successive arrows in the vertical sequence, e.g., in Figure 3 mp (ms) is the unique predecessor (successor) of the message m.

Note that the definition of *predecessor* relation as given in [19] is ambiguous, since the meaning of *message completion* has not been specified. Among all reasonable interpretations we adopt the following one: a completion of a message means the termination of the action related to the sending of the message and performed by the sender. In case of asynchronous communication, the termination of this action consists in simply sending the message, while in case of synchronous communication it means the resumption of the sender execution that was blocked after the sending of the message.

Messages in a SD are distinguished according to whether their are *synchronous or asynchronous* (graphically represented as filled solid arrowhead and as stick arrowhead respectively) and to whether they are *delayed or instantaneous* (slanted downward arrow or horizontal one respectively). In Figure 3 mp is asynchronous-not delayed, m is synchronous-not delayed and ms is synchronous-delayed.

Since *isAsynchronous* is an attribute only of the action in the metamodel, then we can assume that the following assumption holds:

Assumption 3. Let $m \in i.message$ be a message, if $m.action.isAsynchronous = true$ then the corresponding arrow in the SD has a stick arrowhead, else the arrow has a filled solid arrowhead.

The message $m \in i.message$ is exchanged between two generic objects belonging to the classes $cl \in rl.base$ and $cj \in rj.base$ of the $Classifier$ metaclass, respectively. Such classes play the corresponding roles $rl = m.sender$ and $rj = m.receiver$, of the $ClassifierRole$ metaclass, in the communication. Looking at Figure 3, participants in the communication are graphically represented with rectangular boxes together with their lifelines. Boxes are annotated with either the name of role played, or with the name of the *base* class or with both of them.

To each message m an action $actx$ is associated, i.e., $actx = m.action$, which causes the message to be sent: note that there is enough information to identify the state machine that performs the action, thanks to the $m.sender.available$ $Contents.behavior$ information. Action $actx$ can be either a call action or a send action. Its execution causes either a operation $opx = actx.operation$, as the message m of Figure 3, or a signal $sigx = actx.signal$ which in its turn raises a set of events (call events or signal events, respectively) $evy \in E$, where $E = opx.occurrence$ or $E = sigx.occurrence$. Signals are considered asynchronous while operations can be either synchronous or asynchronous depending on the action they are caused by.

Given a UML SD sd_i, the approach taken for its translation into a LGSPN model consists of the following steps:

1. model each message of sd_i with a LGSPN subsystem;
2. compose the LGSPN subsystems obtained in the previous step to take into account the partial order relation that exists between messages belonging to the same interaction. In a certain sense this operation will consist in "connecting" those subsystems modeling messages which are in precedence relation;
3. define the initial marking on the resulting LGSPN representing sd_i.

2.2.1 LGSPN representation of a single message

Let us start to give a representation into GSPN formalism of a single message of a SD sd_i. Given a message $m \in i.message$ the set of its predecessor messages is given by $PR_m = \{p \in i.message \mid p \in m.predecessor\}$ and the set of successor messages is $S_m = \{s \in i.message \mid s \in m.successor\}$, while the action is $actx = m.action$.

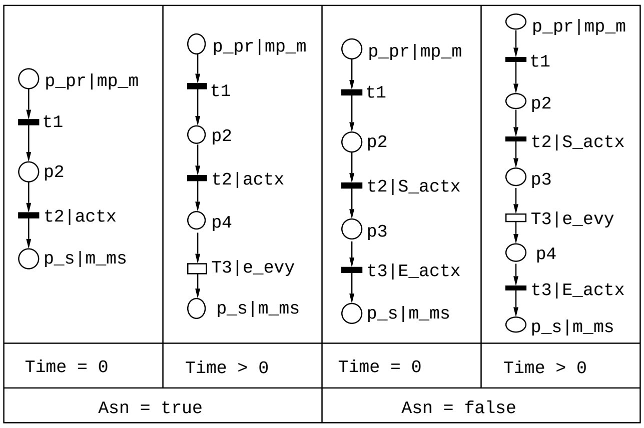

Figure 4: Translation of the message m into a LGSPN.

We denote as $Asn = actx.isAsynchronous$ the boolean variable that indicates whether $actx$ is asynchronous or not, and $Time = m.receiveTime - m.sendTime \in \mathbb{R}_0^+$ the variable that indicates whether m is instantaneous or delayed. Depending on the values of Asn and of $Time$ a different translation is required for the corresponding message. Figure 4 shows the four possible translations into LGSPN of message m.

Let us consider the simplest case (asynchronous and no delay): place p_{pr} is labeled with mp_m if mp is the predecessor of m (place p_{pr} is duplicated as many times as needed if there is more than one predecessor), transition t_2 of label $actx$ represents the action $m.action$, and the sink place p_s is labeled with m_ms if ms is the successor message (again, place p_s is duplicated as many times as needed if there is more than one successor).

If action $actx$ is synchronous, instead, the transition of label $actx$ is duplicated to match the start and the end of the action. If a message is delayed it means that the effect of the message is delayed, that is to say the event raised by the message is not "readily consumed". To model this, the translation has an additional timed transition of label e_evy, if evy is the name of the generated event (remember that in SC translation this label is used for transitions that "consume" events of type evy). Obviously, different assignments to the rate parameter are possible depending on the modeling context.

To formally define the LGSPN representing a message m we need to introduce the following functions:

$$\chi(X) = \begin{cases} 0 & \text{if } X = \emptyset \\ 1 & \text{if } X \neq \emptyset \end{cases}, \quad \Theta(Asn) = \begin{cases} 0 & \text{if } Asn = false \\ 1 & \text{if } Asn = true \end{cases}$$

where X is a subset of class instances,

$$\Delta(Time) = \begin{cases} 0 & \text{if } Time = 0 \\ 1 & \text{if } Time > 0 \end{cases}$$

and the corresponding complementary functions: $\overline{\chi}(X) = 1 - \chi(X)$, $\overline{\Theta}(Asn) = 1 - \Theta(Asn)$, $\overline{\Delta}(Time) = 1 - \Delta(Time)$. Moreover, for abuse of notation, we assume that $0 \cdot X = \emptyset$ and $1 \cdot X = X$.

The LGSPN $\mathcal{LS}_m = (\mathcal{S}_m, \psi_m, \lambda_m)$ is then characterized by the sets of transitions and places:

$$\begin{aligned} T_m &= \{t_1, t_2\} \cup \overline{\Theta}(Asn)\{t_3\} \cup \Delta(Time)\{T_3\}, \\ P_m &= \{p_2\} \cup \overline{\Theta}(Asn)\{p_3\} \cup \Delta(Time)\{p_4\} \\ &\quad \cup \{\overline{\chi}(PR_m)\{p_1\} \cup \chi(PR_m) \bigcup_{pr \in PR_m} \{p_{pr}\}\} \\ &\quad \cup \{\overline{\chi}(S_m)\{p_5\} \cup \chi(S_m) \bigcup_{s \in S_m} \{p_s\}\}. \end{aligned}$$

For brevity, we leave out the definition of its priority, arcs, weight and initial marking functions; we give, instead, the definition of the labeling functions:

$$\psi_m(p) = \begin{cases} mp\_m & \text{if } p \in \bigcup_{pr \in PR_m} p_{pr} \ \wedge \ mp = pr.name \\ m\_ms & \text{if } p \in \bigcup_{s \in S_m} p_s \ \wedge \ ms = s.name \\ \tau & \text{otherwise} \end{cases}$$

$$\lambda_m(t) = \begin{cases} actx & \text{if } t = t_2 \ \wedge \ \Theta(Asn) = 1 \\ S\_actx & \text{if } t = t_2 \ \wedge \ \Theta(Asn) = 0 \\ E\_actx & \text{if } t = t_3 \\ e\_evy & \text{if } t = T_3 \\ \tau & \text{otherwise} \end{cases}$$

where, S, E, e are constant characters, $m = m.name$, $actx = m.action.name$ and $evy = m.action.operation.occurrence.name$ (or $evy = m.action.signal.occurrence.name$).

2.2.2 LGSPN representation of the SD

Let sd_i be a SD, and $i.message$ be the set of messages exchanged in the interaction. For each message belonging to the interaction a LGSPN is obtained by following the indications given at the previous step. To obtain the GSPN translation of the whole the sd_i, each LGSPN $\mathcal{LS}_m = (\mathcal{S}_m, \psi_m, \lambda_m)$, $m \in i.message$ is considered as a black box characterized by *interface places*. Interface places are either source or sink places, i.e., ${}^\bullet p = \emptyset \vee p^\bullet = \emptyset$.

Note that there exist interface places with τ label, this is the case of LGSPNs modeling messages without predecessors (successors) as message mp (ms) depicted in Figure 3.

The LGSPN $\mathcal{LS}_{sd_i} = (\mathcal{S}_{sd_i}, \psi_{sd_i}, \lambda_{sd_i})$ representing the SD sd_i is obtained by composition of the LGSPNs $\mathcal{LS}_m$, $m \in i.message$ over the interface places. Formally, let

$$L_P = \bigcup_{m \in i.message} \psi_m(P_m) \setminus \{\tau\}$$

be the set of no τ-labels of the interface places of the LGSPNs representing messages, and J be the number of messages of sd_i ($J = |i.message|$). Then $\mathcal{LS}_{sd_i}$ is defined as follows:

$$\mathcal{LS}_{sd_i} = \underset{\emptyset, L_P}{\overset{j \in i.message}{|\,|}} \mathcal{LS}_j$$

where $|\,|_{\emptyset, L_P}$ is the superposition operator over the set of place labels L_P (see the appendix A for its definition) and $\mathcal{LS}_j$ is the LGSPN of the j-th message of sd_i.

3. ANALYSIS

This section describes two different ways to create a final analysable model of the system, and the relative objectives. By final analysable model we mean a GSPN model that includes the behaviour of the SCs and of their interactions represented by the SD and on which we can compute logical properties and performance results: we have therefore to define how the GSPN components are composed, what is the initial marking, the performance indices and the rate parameters. In this paper we do not deal with the computation of a significant value for the rate parameter, and how the UML diagrams can be annotated to carry timing information, a topic that is instead extensively treated, for example, in some of the papers surveyed in [3].

We assume that there are K SCs $\{sm_1, ..., sm_K\}$ and a single SD sd_i. Let $\{\mathcal{LS}_{sm1}, ..., \mathcal{LS}_{smK}\}$ be the LGSPN models of the K SCs produced according to the translation in [18] and $\mathcal{LS}_{sdi}$ the LGSPN model of the SD produced according to the translation of the previous section.

The full case. The first technique, that is called "full", is based on the construction of a complete SC model by superposition over event and event acknowledge places of the K SCs, and on the composition of the resulting LGSPN with the one representing the SD.

Let Ev_k be the set of events produced/consumed by sm_k and Lev_P^k the set of labels of event and event acknowledge places, $Lev_P^k = \{e\_evx, \forall evx \in Ev_k\} \bigcup \{ack\_evx, \forall evx \in Ev_k\}$, and $Lev_P = \bigcup_{k \in \{1,...,K\}} Lev_P^k$, then the complete model sc of the K SCs is given by the LGSPN

$$\mathcal{LS}'_{sc} = \underset{\emptyset, Lev_P}{\overset{j=1,..,k}{|\,|}} \mathcal{LS}_{smj}$$

Observe that this has exactly the same effect as a superposition over the all set of labels $L_P = \bigcup_{k \in \{1,...,K\}} L_P^k$, since the only place labels that appear in more than one LGSPN are those associated with event and event acknowledge places.

$\mathcal{LS}'$ can contain acknowledge places that are sinks (indeed all transitions that represent the consumption of an event send an acknowledge back since it is not defined if the event is synchronous or asynchronous), but if the event is generated by an asynchronous action no acknowledge is ever consumed and therefore the corresponding places should be removed. Let P_{ack} be the set of sink places with label of type ack_evx, then the model sc is:

$$\mathcal{LS}_{sc} = \mathcal{LS}'_{sc} \setminus P_{ack}$$

where $A \setminus B$ removes from net A all places in B and their incidence arcs.

Let Lev_T be the set of transition labels that represents actions and events consumption in the SD sd_i:

$$\begin{aligned} Lev_T \quad = \quad & \{e\_evx, \forall evx \in E_{sdi}\} \bigcup \{acty, \forall acty \in AA_{sdi}\} \\ & \bigcup \{S\_acty, E\_acty \; \forall acty \in AS_{sdi}\} \end{aligned}$$

where E_{sdi} is the set of events of sd_i, AA_{sdi} and AS_{sdi} are the set of asynchronous and synchronous actions of sd_i. The final LGSPN model $\mathcal{LS}$ is obtained by superposition of $\mathcal{LS}_{sc}$ and $\mathcal{LS}_{sdi}$ over the set of transitions of labels in the set Lev_T, that is to say:

$$\mathcal{LS} = \mathcal{LS}_{sc} \underset{Lev_T, \emptyset}{|\,|} \mathcal{LS}_{sdi}$$

Again, according to the labeling produced by the translation, by using the whole set of transition labels we would get the same resulting net.

Note that a transition that is produced by the superposition of two transitions of different priority is assigned a priority equal to the lower of the twos (so the superposition of an immediate with a timed produces a timed transition).

Observe that in the model produced according to the "full" technique the presence of sd_i only constraints the sc behaviour for what concerns the actions modeled in $\mathcal{LS}_{sdi}$, while all others interactions among and inside K SCs are still freely permitted.

On the "full" model we can define and compute the following properties and performance indices:

1. From the initial marking it is possible to reach one (or more than one) state in which sink place(s) of $\mathcal{LS}_{sdi}$ is (are) marked. Let M^{end} be one such state.

2. Compare the actions that are included in the shortest path of $\mathcal{LS}$ from the initial marking M^0 to M^{end} with the actions that are modeled in $\mathcal{LS}_{sdi}$.

3. Compute transient measures for entering specific states (for example M^{end}).

The interpretation of the results obtained depends on the hypothesis we are making on the reciprocal role of sc and sd_i. Coming back to the two questions Q1 and Q2 stated in the introduction, if we assume for Q1 that the set of the K sm_k includes SCs that are not involved in the sd_i interaction, then it is highly probable that states like M^{end} are not deadlock states (since the system may be able to evolve, even if the only marked place of sd_i is a sink place). Moreover the shortest path to M^{end} may well contain actions that are not considered in the sd_i interaction, but that belong to the sc description. Finally, the stochastic measures refer to the (potentially) many possible ways of reaching M^{end} taking stochastically into account the whole behaviour of the system.

Question Q2 is answered by point 2 above: if the shortest path includes actions not in sd_i then sd_i only describes a subset of the interactions, and, viceversa, if no marking M^{end} is reachable from M^0, then sd_i describes an interaction that can never take place in the sc description of the system.

The constrained case. How can we compute measures like mean time to M^{end}, *conditioned* on the fact that the path that leads to M^{end} includes only actions that are in sd_i, and in the same order (that we shall refer to as "compatible path")? It is necessary to be able to associate rewards with path (instead that to states and/or transition firings), and anyhow the resulting measure refer to a system that can fully evolve (since all interactions are allowed, not only whose in sd_i). As far as we know there is no Petri net tool that has this option. The other possibility is to constraint the net behaviour by modifying the net structure, as follows.

Let $\mathcal{LS}$ be the LGSPN system obtained according to the "full" technique, and let $\overline{Act}$ be the set of actions that are in sc, but not in sd_i. $\mathcal{LS}$ should not allow $\overline{Act}$ actions to happen. A naive solution could be to remove the corresponding transitions from $\mathcal{LS}$, but in this manner the net will certainly deadlock since, from a Statechart point of view, this is equivalent to say that the SC of Figure 2 moves out of state Nl by consuming event evx, but that state Ml is never reached, since action $actx$ has been cancelled.

It is therefore necessary to remove from the net all the sequence of places and transitions from the action back to the choice of executing that action, that, according to the proposed translation, amounts to follow the backward path from each transition of label $actx$ up to the first transition with more than one input arc (that is going to be labeled e_evx) and to remove the path from the net, including the first and last transition in the path.

The resulting model allows only and exactly only the actions of sd_i, in the same order as they are executed in $\mathcal{LS}_{sdi}$, so the answer to questions Q1 and Q2 is that only the components and the behaviour that are described in the SD are taken into account by the analysis.

Observe although that the net built with the "constrained" method does not allow even local actions, that is to say actions of a SC sm_k that produce events that are consumed by sm_k itself. If it is possible to discriminate between local actions and actions that causes an interaction, then the manipulation of $\mathcal{LS}$ could take place only for $\overline{Act}$ actions that are not local. This implies that the SD is not the pattern of action execution, but it is interpreted as the *only* possible pattern of interaction of the components (answer to Q2).

4. EXAMPLE

As an example we consider the *watchdog* mechanism which is a software fault tolerance provision for error detection. Once initialized from the application which uses it, the watchdog receives periodic signals of life from the application. If the watchdog does not receive any signal from the application within a fixed time-interval, either due to a transmission delay of the signal itself, or due to a simple delay of the application in executing its activity or actually due to a fault that caused an error to occur in the application, the watchdog sends a notification message to the *backbone*. The backbone is a safety recovery software mechanism whose task is to take the corrective action in case of occurrence of exceptions in the system. As effect of reception of the notification message from the watchdog, the backbone provides to terminate the watchdog and to reset the application.

Assuming that each element is represented by a class, the behavior of each class is modeled by the SCs depicted in the upper part of Figure 5. In the same figure are also represented the corresponding translations to LGSPNs. The labels of the places/transitions are annotated in italics font, while their names are annotated in boldface font; for readability purposes, only some names of places are shown.

A possible scenario is represented by the SD of Figure 6, upper part, and the corresponding translation to LGSPN is given in the lower part. The global GSPN model is derived by transition-based composition of the LGSPN of the SCs, obtained by place superposition of the LGSPNs of Figure 5, with the LGSPN of the SD, shown in Figure 6. Due to the watchdog definition, and to the particular SD chosen, the LGSPN is the same for the "full" technique and for the "constrained" one.

The reachability set of the model is characterized by 76 Tangible Markings and 7 of them are deadlock markings.

Correctness analysis. As a first analysis we can verify that the scenario represented by the SD of Figure 6 is "admissible" in the sense that there exists, at least, a firing sequence that, starting from the initial state of the model brings the model to a marking in which place **endSD** is marked, stating that the SD has been executed. Using the GreatSPN-to-PROD translator [8], the GSPN model has been fed to the PROD [26] model checker to find out the states in the Reachability Graph (RG) in which the (sink) place **endSD** is marked (meaning that the whole SD has been executed). Four states have been identified, and they are all deadlock states: three of them represent states of the system in which there are messages that have not been processed by the receiver, i.e., some places representing events are not empty. This is possible since some messages of the SD are asynchronous messages, so that the execution of the SD does not imply that all the produced events have been consumed.

Performance analysis. Since the model is open, i.e., its RG is not strongly connected, computation of metrics are carried out in transient state. By using GreatSPN [10] it is possible to compute the mean time to absorption for the four deadlock states identified previously. This provides information on the behaviour of the system when behaving in a manner "consistent" with the SD.

More interesting measure can be computed, although in a less straightforward manner. As an example we have computed the probability distribution function (Pdf) of the time to reset after a fault occurrence (TTR) that is marked in the SD of Figure 6 with a curly bracket on the time axis t. On the GSPN model, the metric to be computed is translated into the crossing time of a token from the place **startft** (transition labeled as *affect* being enabled) to the place **endSD**.

To obtain the performance result we have first checked how many reachable states enable transition *affect*, and we found a 22 of them, but they all share the characteristic

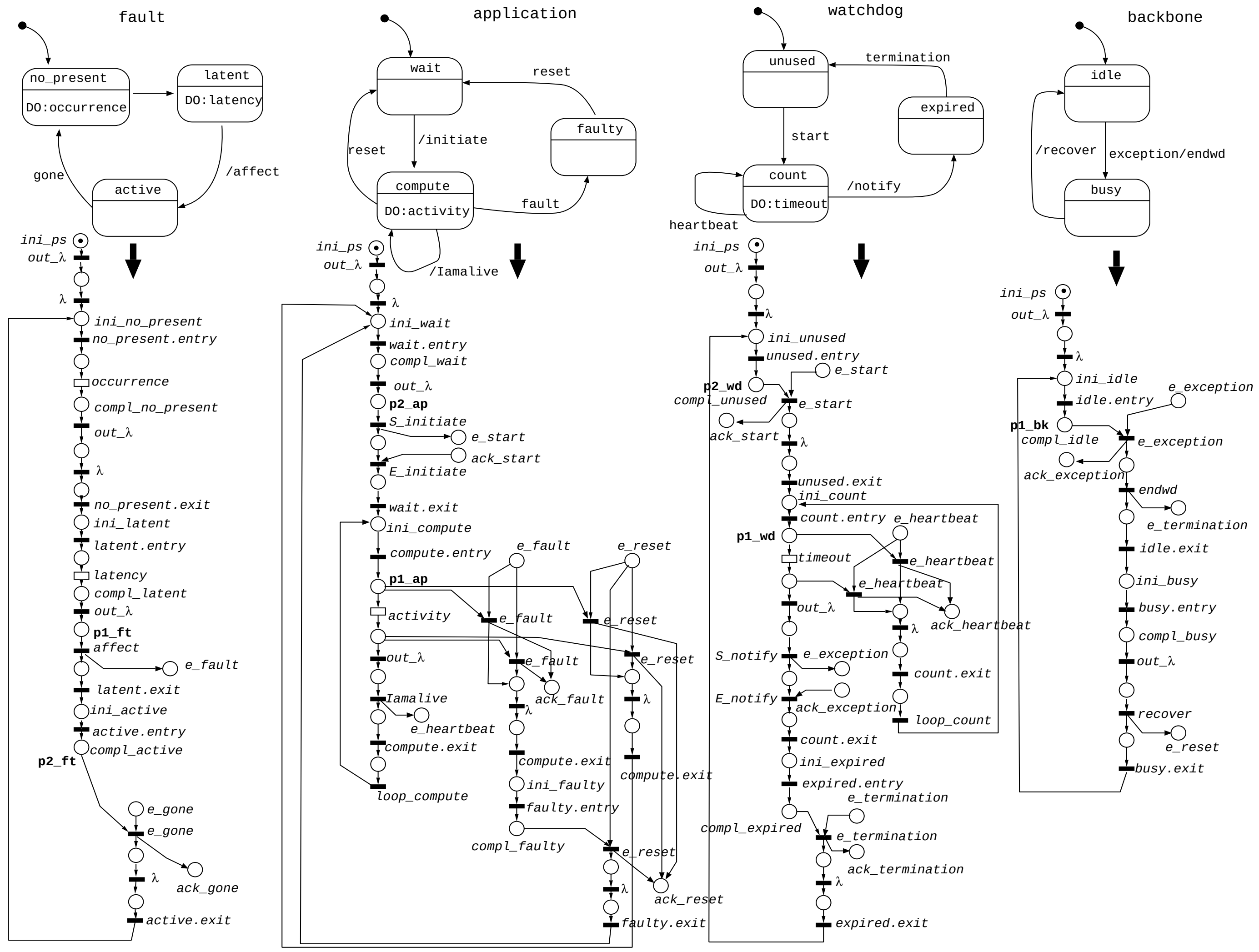

Figure 5: Translation of the SCs into LGSPNs.

that they uniquely lead with a sequence of immediate firings to the following marking, that is therefore taken as initial marking: $M0$ = [**p1_ft, p1_ap, p1_wd, p1_bk, startft**]. In this state message $m9$ is the first message that is sent. The first four places of $M0$ identify the states of the participants involved in the interaction, in particular, place **p1_ft** states that a $fault$ is going to occur, place **p1_ap** states that the *application* is executing its own activity, place **p1_wd** states that the *watchdog* is counting and place **p1_bk** states that the *backbone* is idle. The last place, instead, identifies the execution step of the SD corresponding to the sending of the message $m9$. The absorbing state is the deadlock marking $D4$ = [**p2_ft, p2_ap, p2_wd, p1_bk, endSD**]. Figure 7 shows the curves of the Pdf of the TTR computed under different transmission rate assumptions. In particular, all the timed transition rates of the model have been set to $1.0/ms.$ except for the rate parameter of transition **T2**, which models the transmission delay of message $m10$, that ranges from $1/ms.$ to $4/ms.$ As expected, the higher the transmission rate of the notification message to the backbone mechanism the shorter is the time to reset the application from its error state.

5. CONCLUSION

This paper presents a translation of Sequence Diagrams of UML into LGSPN, and how it can be used in conjunction with a (previously defined) translation of Statecharts into GSPN to provide a "complete" model of the system behaviour. The translation, and the relationships between SC and SD, are based on the UML metamodel, that should allow a reuse for the translation for other collaboration or state machine diagrams. The paper also discusses the type of analysis that can be performed, depending on the different hypothesis on the role that SD and SC play in the description of the system.

For what concerns future work we plan to study how the translation should be changed to move from classes to objects: the use of SDs at instance level supposes that each object, participating to the interaction represented by the SD, has got an identity and that communication between objects is carried out by specifying the identities of the sender and of the receiver. To keep the identities of the objects is not a trivial task in GSPN and we envision the need to move to colored extensions of Petri nets. The translation should also

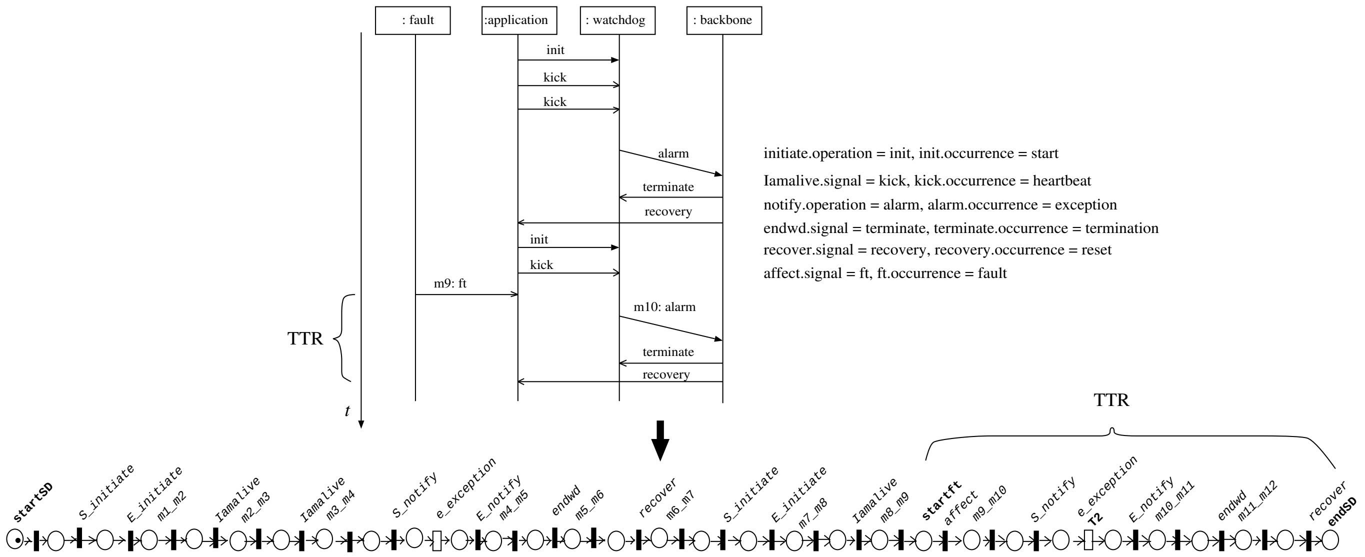

Figure 6: Translation of the SD into LGSPN.

be extended to other SD and SC features, and the problem of extracting temporal information from SD and SC, and how they can be eventually annotated for performance evaluation will also be considered in the near feature.

Concerning the analysis, we plan to consider multiple SDs and how their interaction with SCs impact on the system's performance; in case, by considering UML Use Case Diagrams to capture dependencies among the SDs at an higher level view.

As any automatically produced model also the LGSPNs produced from SC and SD contain a number of unnecessary elements: additional work is needed to define equivalent models [20] of reduced size in terms of vanishing markings and in terms of net elements (places/transitions). Petri nets, even after reduction has taken place, can suffer the so called "state space explosion problem". In our context the problem could be not so bad due to the fact that most places are 1-bounded, but it may nevertheless happen that the state space is too big for the computation of performance results using analytical solvers: in these cases simulation is indeed an option. Moreover, it should be observed that the correctness analysis uses model checkers that are less affected by the state explosion problem.

Finally, a necessary condition for what is written in this paper to eventually become useful modeling practice is to have it implemented: an interesting feature of UML is that there is a standard exchange format for the diagrams, called XMI, and we plan to start from there the implementation of the translation.

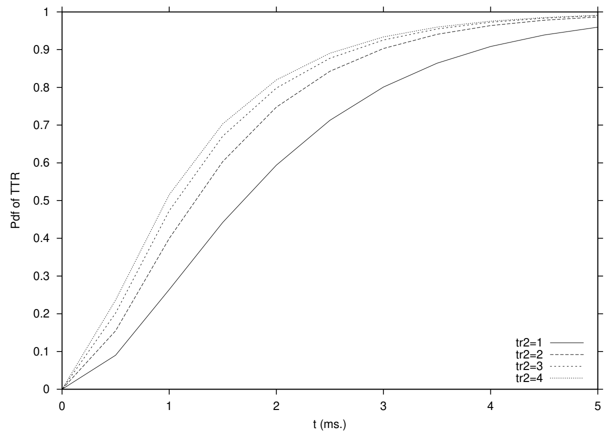

Figure 7: Comparison of the Pdf of TTR under different transmission rates.

6. REFERENCES

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modeling with Generalized Stochastic Petri Nets.* J. Wiley, 1995.

[2] F. Andolfi, F. Aquilani, S. Balsamo, and P. Inverardi. Deriving performance models of software architectures from Message Sequence Charts. In *Proc. of the* 2^{nd} *Int. Workshop on Software and Performance*, pages 47–57, Ottawa, Canada, Sept. 2000. ACM.

[3] S. Balsamo and M. Simeoni. On transforming UML models into performance models. In *ETAPS01: Workshop on Transformations in UML*, 2001.

[4] E. Best, R. Devillers, and J. Hall. The Petri box calculus: a new causal algebra with multilabel communication. In G. Rozenberg, editor, Vol. 609 of *LNCS*, pages 21–69. Springer Verlag, 1992.

[5] A. Bondavalli, I. Maizik, and I. Mura. Automated Dependability Analysis of UML Designs. In ICS Press, editor, *in Proc. ISORC'99 - 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing*, Saint Malo, France, 1999.

[6] J. Cardoso and C. Sibertin-Blanc. Ordering actions in Sequence Diagrams of UML. In *Proc. of* 23^{th} *Int. Conf. on Information Technology Interfaces*, Pula, Croatia, 2001.

[7] V. Cortellessa and R. Mirandola. Deriving a Queueing Network based Performance Model from UML Diagrams. In *Proc. of the* 2^{nd} *Int. Workshop on Software*

and Performance, pages 58–70, Ottawa, Canada, Sept. 2000. ACM.

[8] S. Donatelli and L. Ferro. Validation of GSPN and SWN models through the PROD tool. In *Proc. of the* 12^{th} *Int. Conf. on Modeling Techniques and Tools*. Vol. 2324 of LNCS, Springer and Verlag, April 2002.

[9] S. Donatelli and G. Franceschinis. The PSR methodology: integrating hardware and software models. In *Proc. of the* 17^{th} *Int. Conf. in Application and Theory of Petri Nets, ICATPN '96*, Osaka, Japan, June 1996. Springer Verlag. LNCS, Vol 1091.

[10] GreatSPN: `http://www.di.unito.it/~greatspn`.

[11] `http://ls4-www.informatik.uni-dortmund.de/QPN`.

[12] P. King and R. Pooley. Using UML to Derive Stochastic Petri Net Models. In *Proc.* 15^{th} *UK Performance Engineering Workshop* , pages 45–46, Bristol, UK, July 1999.

[13] I. Kruger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[14] J. Lilius and X. Li. Timing analysis of UML sequence diagrams. *Tech. report no.281 - Turku Centre for Computer Science, Finland*, May 1999.

[15] J. Lilius and I. Paltor. The production cell: an exercise on the formal verification of a UML model. *Tech. report no.288 - Turku Centre for Computer Science, Finland*, May 1999.

[16] J. Lilius and I. Paltor. The Semantics of UML State Machines. *Tech. report no.273 - Turku Centre for Computer Science, Finland*, May 1999.

[17] J. Merseguer, J. Campos, and E. Mena. Performance evaluation for the design of agent-based systems: A Petri net approach. In M. Pezzé and S. M. Shatz, editors, *Proc. of the Workshop on Software Engineering and Petri Nets, within the* 21^{st} *Int. Conf. on Application and Theory of Petri Nets*, pages 1–20, Aarhus, Denmark, June 2000. University of Aarhus.

[18] J. Merseguer, S. Bernardi, J. Campos, and S. Donatelli. A Compositional Semantics for UML State Machines aimed at Performance Evaluation. (submitted for publication).

[19] *OMG Unified Modeling Language Specification: version 1.4*. Object Management Group Inc., Sept. 2001. http://www.omg.org.

[20] L. Pomello, G. Rozenberg, and C. Simone. A survey of equivalence notions for net based systems. *LNCS; Adv. in Petri Nets 1992*, 609:410–472, 1992.

[21] pUML Group. http://www.cs.york.ac.uk/puml.

[22] S. Bernardi and S. Donatelli and A. Horváth. Implementing Compositionality for Stochastic Petri Nets. *Int. Journal of Software Tools for Technology Transfer (STTT)*, 3(4), August 2001.

[23] R.Taylor. A general purpose algorithm for analyzing concurrent programs. *Comm. of ACM*, (26), May '83.

[24] A. Tsiolakis. Integrating Model Information in UML Sequence Diagrams. Electronic Notes in Theoretical Computer Science, July 2001.

[25] UltraSAN: `http://www.crhc.uiuc.edu/UltraSAN`.

[26] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD Reference Manual. Tech. Report Series B, number 13, Helsinki University of Technology, August 1995.

APPENDIX

A. COMPOSITION OF LGSPNS

Given two LGSPN ordinary systems $\mathcal{LS}_1 = (\mathcal{S}_1, \psi_1, \lambda_1)$ and $\mathcal{LS}_2 = (\mathcal{S}_2, \psi_2, \lambda_2)$, the LGSPN ordinary system:

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

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 $F \in \{\Pi(), W()\}$, the initial marking function and the labeling functions are respectively equal to:

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

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

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

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

The relation being associative with respect to place superposition, we use also as an n-operand by writing $\mathcal{LS} = ||_{\emptyset, L_P}^{k=1,..,K} \mathcal{LS}_k$.