Software Performance Modeling based on UML and Petri nets

José Javier Merseguer Hernáiz

TESIS DOCTORAL

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Advisors: Dr. Javier Campos Laclaustra
Dra. Susanna Donatelli

January 2003

Acknowledgements

I am indebted and very grateful to Javier Campos, he gave me the "opportunity", and every day his support at once. I would also like to thank Susanna Donatelli, she gave crucial ideas to channel this thesis in the "right" moment.

I would like to thank Simona Bernardi, Juan Pablo López Grao and Eduardo Mena who coautored the works that support this thesis. I would be very grateful if they consider this work as theirs.

I would like to thank the members of the Petri net group of the Universidad de Zaragoza, particularly, I want to mention special gratefulness to Carlos "Charlie" José Pérez, Laura Recalde and Diego Rodríguez, who solved me a lot of problems, not only in the Petri net world.

I am particularly grateful to Isidro Ramos, mainly because he shown me the "path of the research" in the "Wednesday afternoon meetings", some years ago when I was grant holder in his research group. Also for the facilities offered during six months in the final period of the thesis. I want to remember my workmates in that period, summer 2002, specially Pedro Alberti, who "introduced" me and my computer in the Linux world, that was fantastic, I have forgotten "panic windows messages".

I cannot forget mentioning my friends José Romero and Jorge Ferrer, researchmates in the Universidad Politécnica de Valencia, I want to give them encouragement to succesfully finish their thesis.

The environment provided by the following projects made writing this book possible: The projects TAP 0679/98 and TIC 04334-C03-02 financed by the Spanish Ministerio de Ciencia y Tecnología; acción integrada Hispano-Italiana HI 1998-0196, financed by the Spanish Ministerio de Educación y Ciencia; project UZ-00-TEC-03, financed by the University of Zaragoza and projects PI423-06 and P084/2001 financed by the Aragonese Diputación General de Aragón.

Lastly, I would like to thank my family and my friends, the most important people for me, for their support.

Preface

Today complex distributed software systems, most of them based on the Internet, are simultaneously used by hundred-thousands of people sometimes in risk real time operations such as auctions or electronic commerce. For this kind of systems, performance, dependability or responsiveness requirements of high quality become a must. Nevertheless, systems are deployed without the performance expected by clients, being a common practice among developers to test the performance of their systems only when they have been implemented. It is common to believe that powerful hardware at minor extra cost will solve performance problems when the software is deployed. In our opinion, the necessity to predict the performance of software systems is doubtless. Moreover, we consider that predictions in the early stages of the software life cycle promote major benefits. Thus, if performance objectives are not meet, it will be easier and less expensive to take the appropriate design decisions to solve them.

To face these problems *software engineering* and *performance evaluation* are the scientific fields where answers should be found: The first as the discipline that studies the systematic representation of software requirements and the second as the discipline that proposes formalisms to represent and analyze performance models. Both fields have met a common place, the *software performance engineering*, that is proposed as a method for constructing responsible software systems that meet performance objectives. In this work, to achieve this goal we propose to combine the use of the Unified Modeling Language with the Petri net formalism. The Unified Modeling Language has been adopted by the software engineering community as a "de facto" standard, while the stochastic extension of the Petri net formalism has been proved in the last decades as one of the most successful paradigm to evaluate performance issues in different engineering fields.

The main contributions of this work are the definition of a compositional semantics in terms of stochastic Petri nets for the UML state machines aimed at performance evaluation and a performance extension of the UML notation. They together are used to evaluate performance in the early stages of the software development process. The semi-automatic generation of performance models from UML designs allows that as a "by-product" of the software life cycle performance measures can be estimated; moreover, it implies that the training of the software engineers in performance formalisms can be relaxed. The approach has been applied to predict estimates in different real software systems, obtain interesting results in the field of the wireless systems.

Contents

Chapter 1

Introduction

Today the number of advances in the vast amount of technologies related with computer sciences has improved the development of complex distributed software systems, most of them based on the Internet, maybe in wireless environments, accessing complex and heterogeneous data repositories and with the most sophisticated user interfaces. Not only the "big" companies, even small vendors can participate in the "race", sometimes exhibiting more successful results than the bigger ones. These software products are used simultaneously by hundreds or thousands or hundred-thousands of people, sometimes increasing risks as in real time operations such as auctions, electronic commerce or financial systems. Considering these facts, the deployed software should satisfactorily fulfill an important number of requirements: Performance, dependability, responsiveness, security or scalability are examples of qualities that good software products must meet. *Software engineering* has been recognized as one of the most useful disciplines in the computer science field to face all these stuff. Concerning the performance requirements of the deployed systems, subject of this thesis, the results cannot be considered extremely satisfactory. Actually, the developers are not so worry about them, arising performance problems sometimes because it is considered that powerful hardware at minor extra cost will solve them, sometimes because real scenarios of usage (with respect the number of users or the operation mode) are not tested, sometimes because the training of software engineers in performance assessment issues (models and techniques) are not adequate or even they lack in their curricula. *Performance evaluation* is the field that studies and proposes the application of performance models and techniques for the different engineering disciplines to meet its performance objectives.

The following "war stories", taken from [SS01], depict situations that unfortunately cannot be considered strange in the world of the computer systems. They show the importance of the evaluation of the performance characteristics of the software systems.

At the Olympic Games in Atlanta, the information system developed by IBM to evaluate individual competition results was tested by one hundred and fifty users. The

system crashes during the competition since it was used by more than one thousand people. As a result some matches were delayed and IBM suffered deep-cutting image losses.

The luggage processing system of the Denver airport was planned for the United Airlines terminals, during the development it was enlarged to support all the airport terminals but without considering the new system's workload. As a result of the inadequate performance characteristics of the system, the airport was opened 16 months later, a loss of 160.000 US$ per day was recorded.

As a conclusion, the necessity to predict the performance of software systems is doubtless, since even nowadays, systems are deployed without the performance expected by clients, being a common practice among developers to test the performance of their systems only when they have been implemented. Performance requirements of software systems must be considered within the software development process to meet performance properties as soon as possible.

It should not be forgotten that performance requirements meeting for a software system ultimately depends on the hardware platform that will support its executions. Therefore, the performance requirements for a system should be consider as a whole, taking into account at least the hardware, the software and the middleware. Nevertheless, such an ambitious approach is out of the scope of this work, that focuses on the software part of the system. It causes that in this work an "infinite hardware resources" hypothesis is assumed. It will be desirable (necessary) that the results obtained in this work will be integrated or extended by an approach of that general form.

This thesis arises in an environment where there exists a strong experience in the analysis of queuing network models for the computation of performance measures (which is a well-known branch of the performance evaluation field to be applied in software systems), as well as knowledges in the modeling of software systems. Both areas share common interests in the *software performance engineering* field.

The software performance engineering (SPE) is proposed as a "method (a systematic and quantitative approach) for constructing software systems to meet performance objectives, taking into account that SPE augments others software engineering methodologies but it does not replace them" [Smi90]. We propose to meet these goals by combining the use of the the Unified Modeling Language and the Petri net formalism. In the following we briefly recall both disciplines.

The Unified Modeling Language (UML) is a semi formal language developed by the Object Management Group [Obj01] to specify, visualize and document models of software systems and non-software systems too. It is gaining widespread acceptance as an effective way to describe the behavior of systems. As such, it has also attracted the attention of researchers that are interested in deriving, automatically, performance evaluation models from system's descriptions. Actually, in the last years UML has become the standard notation to model software systems, widely accepted by the software engineering community. Unfortunately, UML lacks of the necessary expressiveness to accurately describe performance features.

Petri nets (PNs) were proposed in the literature for the modeling of concurrent

systems and the analysis of its qualitative properties [Pet81, Sil85]. The stochastic interpretation of the PN formalism [AMBC+95] has been shown as a suitable tool for the performance evaluation of complex distributed systems. As an alternative to stochastic PNs (SPNs), several performance-oriented formalisms can be considered as underlying mathematical tools for computing performance indices of interest: Among them, Markov chains (MCs) [Çin75], queuing networks paradigm (QNs) [Kan92], and stochastic process algebras (SPAs) [HHM95]. The main drawback of plain MCs is their poor abstraction level (each node of the underlying graph model represents a state of the system). Concerning SPAs, even if they are compositional by nature, thus specially adequate for the modeling of modular systems, the present lack of efficient analysis algorithms and software tools for performance evaluation would make them difficult to use in real applications. The use of SPNs rather than QNs models is justified because SPNs include an explicit primitive for the modeling of synchronization mechanism (a synchronizing transition) therefore they are specially adequate for the modeling of distributed software design. Even more, a vast amount of literature exists concerning the use of PNs for both validation of logical properties of the system (e.g., liveness or boundedness) and quantitative analysis (performance evaluation).

The reasons to combine both formalisms (SPNs and UML) in a proposal for SPE are given by the advantages and disadvantages they provide. A disadvantage of SPNs is that they do not show the system load (message size, guards probability, activities duration) as clear as UML diagrams do, since this information is hidden in the definition of the parameters of the net transitions. Besides, much work has been done in the software engineering area in developing methodologies [RBP+91, JCJO92] from which UML takes profit and SPNs do not. SPNs have advantage over UML because they can be used to obtain performance figures, due to the underlying mathematical model; even more, there exist software tools that automatically obtain them [CFGR95, ZFGH00]. Moreover, PNs express concurrency unambiguously, UML does not. Finally, it can be said that the UML diagrams are considered in this work as the documentation part of the system, being useful for the system analyst to express in an easy way the system requirements, including performance requirements. While SPNs are considered as the underlaying mathematical tool which represents the performance model of the system.

The main objective of this work is to provide software engineers with the tools to achieve performance estimates in the early stages of the software life cycle. Thus, if performance objectives are not meet, it will be easier and less expensive to take the appropriate design decisions to solve them. Performance evaluation often requires deep knowledge in queueing theory, to avoid this necessity, it is a challenge a hight degree of integration among the common software engineering practices and the performance evaluation techniques. So, it is our objective that performance models should be obtained as a "by-product" of the software life cycle, avoiding the software engineer to perform tasks for which a strong mathematical background is required.

In the following, we recall the work developed in this thesis to summarize the objectives and explain some details of them.

The study of the software time efficiency (response time, delays, throughput)

requires from the software designer the most accurately possible description of the system load and the routing rates. As previously discussed, software engineers are not familiar with the notation of the performance theory, moreover this notation is too far from the artifacts they use to model software systems. Our proposal to describe the load and the routing rates in software systems considers UML as a modeling language and augment it (using the tagged values mechanism) to introduce these abilities at software engineer level. At the same time, it tries to be adequate to generate performance models and to fit in the performance evaluation process that we propose. Then we have investigated the role of all the UML behavioral diagrams with performance evaluation purposes: Use case diagrams, interaction diagrams, statechart diagrams and activity diagrams. These subset of UML diagrams is powerful enough to describe the dynamics and the load of a wide range of distributed software systems.

The use case diagrams are shown as the tool to characterize the actors of the system by the usage they perform of it. Interaction diagrams allow to describe the load of the system when the participants exchange messages among them. The statechart diagrams are a tool where the routing rates and the duration of the activities of the system at high level of description can be modeled. While activity diagrams became useful for a detailed and accurately modeling of an internal process measuring its basics activities.

Since these diagrams lack of formal semantics then they cannot be analyzed as a performance model to obtain estimates. Therefore we propose a translation (an interpretation) from them into a formalism with well-known properties and able to represent performance models capable to be analyzed using the theory in performance evaluation, i.e. SPNs. The approach adopted in this work for the formalization of the statecharts and the activity diagrams consists in translating the two UML notations into SPNs separately, starting from the metamodels of the UML *State Machines* and *Activity Graphs* packages which describe informally the semantics underlying the statecharts and the activity diagrams, 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 statecharts and the activity diagrams, since the two metamodels contain common metaclasses. The formalization of the sequence diagrams being in the scope of our proposal [BDM02] has not been developed in this thesis. The elements of the use case diagram do not require of a formalization.

Given a "flat" UML state machine, the approach taken for its translation into a SPN model, takes the following steps:

step 1 Each simple state is modeled by a SPN representing the basic elements of states and transitions.

step 2 The initial pseudostate (if it exists) and the final states are translated. The SPN subsystems produced in this step are composed with those of the previous step to produce a SPN model of the entire state machine.

If the state machine is not "flat":

step 3 A SPN model for each concurrent region in a compound state will be obtained proceeding as in step two.

step 4 For each composite state a SPN model is obtained either composing its concurrent regions if any or proceeding as in step 2.

If the system is described through a set of state machines, the final step composes them:

step 5 Compose the SPN subsystems of the state machines and define the initial marking.

For the translation of a UML activity graph a new perspective has been explored. As an initial premise, we assume that the activity graph 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 can then be divided in three phases, which are presented in the subsequent paragraphs.
Pre-transformations. Before translating the activity graph, we need to apply some simplifications: Suppression of decisions, merges, forks and joins; deducting and making explicit the implicit control flow.
Translation process. Following three steps:

step 1 Translation of each diagram element into the corresponding stochastic Petri net model.

step 2 Superposition of models corresponding to the whole set of each kind of diagram elements.

step 3 Working out the complete model for the diagram itself by superposition of the models obtained in the last step.

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 model.

The definition of a notation for some UML diagrams to describe system load as well as their translation into SPN models are the basis to define a process to obtain performance models that represent software systems as a "by-product" of the software life-cycle. This process tries to bridge the gap among the software engineering practices and the performance evaluation practices. Additionally, we have explored some ideas related with the "design patterns" to complement our approach of performance evaluation process.

Finally, the proposal to evaluate performance of software systems has been applied to three distributed systems. Two of them were software retrieval systems, it has allowed to carry out a performance comparison among them.

The work proposed will allow to obtain: Compositional semantics in terms of SPNs for the UML state machines and the activity graphs representing our interpretation of

both tools; the definition of the performance annotations to augment the UML with performance features as well as to define the role of each UML diagram; the definition of a process that using the augmented UML designs and the translation procedures generates a performance model.

This thesis comprises nine chapters including this one, the balance is as follows:

In chapter 2 the context of the work is clearly stated by defining the Petri net formalism, revising the Unified Modeling Language and highlighting the basic principles for the software performance engineering. The related work in the software performance modeling field using UML is studied in order to compare the advantages/disadvantages given by our proposals of annotated UML and software performance process. Also, the pattern-based approaches for SPE are studied and compared with ours. Finally, the related work is completed by revising some significant works that give formal semantics either to UML state machines or UML statechart diagrams or related formalisms such as classical Harel statecharts. It will allow to compare them with our proposal of compositional semantics for the UML state machines.

In chapter 3 our proposal of performance annotated UML is presented. We visit each of the UML behavioral diagrams (use cases, interactions, statecharts and activity diagrams) to find the role that each one can play in the performance process. Then we define for each diagram the feature/s (system load, system usage or routing rate) that it is able to model as well as its representation by means of the tagged value UML extension mechanism.

In chapter 4 we give a translation from the elements in the UML state machines metamodel into the SPNs formalism. The elements taken into account are those that conform the "flat" state machines. The translation being compositional gives semantics to the UML state machines metamodel. Since the translation consider the time expressed in the activities, the SPN model obtained becomes a performance model.

In chapter 5 we give the translation into SPNs for the UML state machine elements that were not considered in the previous chapter. Then leading compositional semantics for composite state machines.

In chapter 6 the extension of the UML state machines, i.e. the activity graphs, is considered. Then, we extend the compositional translation for the elements in the activity graph package that are not present in the UML state machines package.

In chapter 7 we present our process to obtain performance models from UML annotated diagrams that represent software systems and we explore how to evaluate performance parameters from them. The most remarkable feature of this process is that the performance models are obtained as a "by-product" of the software life cycle, i.e. semi-automatically from the UML models. We remark that the only contribution of the process is that it makes use of both the UML annotated diagrams and the translation proposed in the previous chapters, apart of that any proposal of software process should be valid. The process is presented using the Antarctica software retrieval service (designed with mobile agents within a wireless environment) as a running example. Finally, we explore the use of the design patterns approach in the context of the software performance.

In chapter 8 our proposal to evaluate performance of software systems is applied to another software retrieval system that does not use mobile agents. It will allow us to compare some system performance parameters among different software retrieval systems as well as the impact of the mobile and intelligent agents technology. Finally, an Internet protocol is analyzed to test the use of the activity diagrams in our context.

In chapter 9 we present the conclusions of this work as well as the proposed future work.

Chapter 2

Context and State of the Art

Along this chapter and before to present our proposal to evaluate performance of complex distributed software systems, we want to give a brief introduction of the formalisms and notations related to our work. We start with the description of the Petri net formalism, from its basic formulation until the stochastic interpretation, to finalize with a discussion about the analysis of the formalism. Second, the Unified Modeling Language is addressed by describing the semantics of the diagrams related to our approach. Third, the software performance engineering field is contextualized to be aware with its principles and objectives and to know the techniques and models used in this area. Finally, we enumerate the most relevant works in the software performance engineering as well as in the field of the formal semantics for the UML models. Their main features are described and we compare them with our work.

2.1 Petri nets

We assume that the reader is familiarized with the basic concepts of Petri nets, for an introduction see [Mur89, Sil85, DHP+93, Pet81]. In this section, we introduce the notations and concepts related to Petri nets which are of interest in this work.

2.1.1 Place/Transition nets

A place/transition (P/T) net is a mathematical tool aimed to model a wide-range of concurrent systems. In this work, we propose P/T nets to model concurrent and distributed software systems.

A P/T net is graphically represented by a directed graph that comprises a set of *places* P drawn as circles, a set of *transitions* T drawn as bars and arcs that connect transitions to places and places to transitions. A formal definition of a P/T net is thus the following [Sil85]:

Definition 2.1. *A P/T net is a 4th-tuple* $\mathcal{N} = \langle P, T, I, O \rangle$ *such that,*
$P = \{p_1, p_2, \ldots, p_n\}$

$T = \{t_1, t_2, \ldots, t_m\}$
$P \cap T = \emptyset$
$I : P \times T \longrightarrow \mathbb{N}$ *is the input function*
$O : T \times P \longrightarrow \mathbb{N}$ *is the output function.*

Places in a P/T net may contain *tokens* drawn as black dots. The *state* of a P/T net is defined by the number of tokens contained in each place and it is denoted by $M : P \longrightarrow \mathbb{N}$, the net *marking*. The number $M(p)$ means the local state of place p.

Definition 2.2. *A P/T net system (P/T system) is a pair* $\mathcal{S} = \langle \mathcal{N}, M_0 \rangle$ *where* $M_0 : P \longrightarrow \mathbb{N}$ *is the initial marking.*

A transition t is *enabled* in a given marking M_i if and only if for all "input" place p, $M(p) \geqslant I(p,t)$. The evolution of a P/T system is given by the transition *firing rule*: A transition can fire in a given marking M_i producing a new marking M_j, denoted by $M_i \xrightarrow{t} M_j$, if it is enabled in M_i. The new marking is obtained by $M_j(p) = M_i(p) + O(t,p) - I(p,t) \;\; \forall p \in P$.

The firing rule gives to a P/T system dynamic behavior that allows to model concurrent evolutions in discrete systems.

2.1.2 Stochastic Petri nets

The original definition of the Petri nets was as a causal model, explicitly neglecting time. But being a goal in this work the performance evaluation of software systems, the modeling of time constraints is a must.

The introduction of time into the Petri net model allows to attach it either to places [Sif78] or transitions [Ram74]. Following the second approach, timed Petri net models focused on performance evaluation were formerly introduced in [RH80] as a deterministic approach and in [MF76] associating with each transition a maximum and a minimum firing time.

Time interpretation entails difficulties in the definition of the model. Issues such as firing policy, consistency among timing and priority, server semantics and duration of the activities (deterministic or non deterministic) should be defined.

Atomic firing policy establishes that the input tokens of an enabled transition remain in their places until the transition fires, then changing in zero time. Also, it can be interpreted that the input tokens are retained when the transition is enabled until its firing, then leading a three phases policy.

A consistent conflict resolution can be established associating priority zero to timed transitions and priority > 0 to untimed ones, then solving conflicts between immediate transitions associating probabilities to them. Conflicts between timed transitions can be solved by a race policy, i.e. shortest firing delay "wins the race".

The number of concurrent activities modeled by an enabled timed transition means the number of concurrent servers dedicated to the transition. A timed transition should define a "k-server" semantics, then modeling k concurrent activities, being $k \in [1, \infty]$.

As non deterministic model, stochastic Petri nets (SPNs) were initially proposed in [BT81, Mol82, FN85] associating with each transition a random firing time with negative exponential distribution. Formally:

Definition 2.3. *A SPN is a pair $\langle \mathcal{S}, w \rangle$ where $\mathcal{S}$ is a P/T system and $w : T \longrightarrow (0, \infty)$ associates to each transition a negative exponential distributed random variable with parameter w, i.e. its random firing time.*

In this work an extension of the SPNs is considered, the class of Generalized Stochastic Petri Nets (GSPNs) [AMBC84, AMBCC87], that are defined as follows:

Definition 2.4. *A GSPN system is a 8th-tuple $\langle P, T, \Pi, I, O, H, W, M_0 \rangle$ where,*
P, T, I, O, M_0 as in Def. 2.2
$\Pi : T \longrightarrow \mathbb{N}$ is the priority function that maps transitions onto priority levels
$H : P \times T \longrightarrow \mathbb{N}$ is the inhibition function
$W : T \longrightarrow \mathbb{R}$ is the weight function that assigns rates to timed transitions and weights to immediate transitions.

Immediate transitions (those that fire in zero time) are drawn as bars and timed transitions (those that have associated an exponentially distributed random firing $W(t)$) as boxes. Immediate transitions have priority over timed transitions defined by $\Pi(t)$. Conflicts between immediate transitions with equal priority are resolved by $W(t)$. Inhibitor arcs are drawn as "normal" arcs but with a circle joined to the inhibited transition, meaning that the target transition cannot fire until the marking in the inhibitor place is less than $H(p)$.

2.1.3 Labeled Petri nets

Since in this work we give *compositional* semantics, in terms of Petri nets, to a number of UML diagrams, it is necessary to introduce an operator to *compose* the GSPNs models that represent modules of the system. Previously, to properly define such operator, an extension of the GSPN formalism is proposed [BDM02]:

Definition 2.5. *A labeled GSPN system (LGSPN) is a triplet $\mathcal{LS} = (S, \psi, \lambda)$ where,*
S is a GSPN system as in Def. 2.4
$\psi : P \longrightarrow L^P \cup \tau$ is a labeling function for places
$\lambda : T \longrightarrow L^T \cup \tau$ is a labeling function for transitions
L^T, L^P and τ are sets of labels
τ-labeled net objects are considered to be "internal", not visible from the other components.

The previous definition differs from the original given in [DF96] since 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.

Composition will be carry out in this work over LGSPN using the operator defined in [BDM02] as follows:

Definition 2.6 (Place and transition superposition of two ordinary labeled GSPNs). *Given two LGSPN ordinary systems* $\mathcal{L}S_1 = (S_1, \psi_1, \lambda_1)$ *and* $\mathcal{L}S_2 = (S_2, \psi_2, \lambda_2)$*, the LGSPN ordinary system* $\mathcal{L}S = (S, \psi, \lambda)$*:*

$$\mathcal{L}S = \mathcal{L}S_1 \underset{L_T, L_P}{||} \mathcal{L}S_2$$

resulting from the composition over the sets of (no τ*) labels* $L_T \subseteq L^T$ *and* $L_P \subseteq L^P$ *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* $\mathcal{L}S_1$ *that are labeled* l *and* $P_1^{E_P}$ *(*$T_1^{E_T}$*) be the set of all places (transitions) in* $\mathcal{L}S_1$ *that are labeled with a label in* E_P *(*E_T*). Same definitions apply to* $\mathcal{L}S_2$*. 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 $\cup$ *is the union over sets.*
Functions $F \in \{\Pi(), W()\}$ *are equal to:*

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

The initial marking function is equal to:

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

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

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

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

The relation being associative with respect to place and transition superposition, we use also as an n-operand by writing,

$$\mathcal{LS} = \overset{k=1,..,K}{\underset{L_T,L_P}{||}} \mathcal{LS}_k.$$

The previously defined operator was redefined in [LGMC02a] in order to consider the simplification of the nets, as follows:

Definition 2.7 (Place and transition superposition and simplification of two ordinary labeled GSPNs). *Given two LGSPN ordinary systems $\mathcal{LS}_1 = (S_1, \psi_1, \lambda_1)$ and $\mathcal{LS}_2 = (S_2, \psi_2, \lambda_2)$, the LGSPN ordinary system $\mathcal{LS} = (S, \psi, \lambda)$:*

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

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

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

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

2.1.4 High level stochastic Petri nets

Stochastic Petri nets lead to models whose size is too large when modeling realistic systems. High level Petri nets [JR91], have been proposed as a more adequate tool to develop "compact" or "folded" models. In this work, we use a class of stochastic high level Petri nets, the stochastic well-formed nets [CDFH93] (SWNs), which are the stochastic extension of the family of well-formed colored nets [CDFH90].

In this work functions are defined to obtain GSPNs from UML diagrams. But sometimes the performance models obtained are not precise enough to express same system properties (usually instance based properties) or these models can be compacted into simple ones. In such cases, we propose to color the original GSPN in order to obtain a SWN by exploiting the analyst knowledge in the "problem domain". It must be clear from the very beginning that this work does not formalize a translation from UML diagrams into SWNs but GSPNs.
Formally a SWN is as follows [CDFH93]:

Definition 2.8. *A* $SWN = \langle P, T, C, J, W^-, W^+, W^h, \Phi, \Pi, \theta, M_0 \rangle$ *is made of* P, T *and* Π *as in Def. 2.4*
C *a family of basic classes:* $C = \{C_1, \ldots, C_n\}$ *with* $C_i \cap C_j = \emptyset$ *(* $I = \{1, \ldots, n\}$ *a ordered set of indexes)*
$J : P \cup T \longrightarrow \mathrm{Bag}(I)$*, where* $\mathrm{Bag}(I)$ *is the multiset on* I*.* $C(r) = C_{J(r)}$ *denotes the color domain of node* r
$W^-, W^+, W^h \;\; : W^-(p,t), W^+(p,t), W^h(p,t) \in [C_{J(t)} \longrightarrow \mathrm{Bag}(C_{J(p)})]$ *the input, output and inhibition functions are arc expressions*
$\Phi(t) : C_{J(t)} \longrightarrow \{True, False\}$ *is a standard predicate associated with the transition* t*. By default it is assumed* $\forall t \in T$ *the standard predicate* $\Phi(t) = True$
$\theta(t) : \tilde{C}(t) \times \mathrm{Bag}(\tilde{C}(p_1)) \times \mathrm{Bag}(\tilde{C}(p_2)) \times \ldots \mathrm{Bag}(\tilde{C}(p_{|P|})) \longrightarrow I\!R$
$M_0 : M_0(p) \in \mathrm{Bag}(C(p))$ *is the initial marking of* p*.*

2.1.5 Petri nets analysis

As we stated, a goal of this work is to obtain performance models (in terms of GSPNs or SWNs) from UML models describing software systems. Performance models can be used to estimate some quantifiable *performance measures* in the first stages of the life-cycle to evaluate alternatives for some system parameters. With this purpose a performance model can be simulated or analyzed, but only the second approach has been followed in the examples developed in this work (cfr. chapter 8), it does not mean that our proposal forgets simulation techniques.

Responsiveness and utilization performance measures can be calculated, either operationally or stochastically, see [Cam98b] for a survey. As an example, it can be calculated for places, the average steady-state marking; for transitions, the average steady-state enabling degree, utilization or the throughput.

The throughput of a transition, denoted by $\chi(t)$, is defined as the number of firings per unit of time. This performance measure is the only one used in this work (cfr. chapter 8). We refer to [Cam98b] for its calculation.

Traditionally, techniques for the analysis (computation of performance measures or validation of logical properties) of Petri nets are classified in three complementary groups [CTM98]: enumeration, transformation, and structural analysis:

- *Enumeration* methods are based on the construction of the reachability graph (coverability graph in case of unbounded models), but they are often difficult to apply due to their computational complexity, the well-know *state explosion problem*.

- *Transformation* methods obtain a Petri net from the original one belonging to a subclass easier to analyze but preserving the properties under study, see [Ber87].

- *Structural analysis* techniques are based on the net structure and its initial marking, they can be divided into two subgroups: *Linear programming* techniques, based on the state equation and *graph based* techniques, based on "ad hoc" reasoning, frequently derived from the firing rule.

(c_1)	$\overline{\boldsymbol{\mu}}[p] = \mathbf{m_0}[p] + \sum_{t_i \in {}^\bullet p} \lvert f_i \rvert \boldsymbol{\sigma}[t_i] - \sum_{k_j \in p^\bullet} \lvert g_j \rvert \boldsymbol{\sigma}[k_j],$	$\forall p \in P : \mathbf{Post}[p, t_i] = f_i$, $\mathbf{Pre}[p, k_i] = g_j$;
(c_2)	$\sum_{t_i \in {}^\bullet p} \lvert f_i \rvert \boldsymbol{\chi}[t_i] \geq \sum_{k_j \in p^\bullet} \lvert g_j \rvert \boldsymbol{\chi}[k_j],$	$\forall p \in P : \mathbf{Post}[p, t_i] = f_i$, $\mathbf{Pre}[p, k_i] = g_j$;
(c'_2)	$\sum_{t_i \in {}^\bullet p} \lvert f_i \rvert \boldsymbol{\chi}[t_i] = \sum_{k_j \in p^\bullet} \lvert g_j \rvert \boldsymbol{\chi}[k_j],$	$\forall p \in P$ bounded;
(c_3)	$\frac{\boldsymbol{\chi}[t_i]}{r_i} = \frac{\boldsymbol{\chi}[t_j]}{r_j},$	$\forall t_i, t_j \in T$: behav. free choice;
(c_4)	$\lvert f \rvert \boldsymbol{\chi}[t] \, \overline{\mathbf{s}}[t] \leq OUT(p,t) \, \overline{\boldsymbol{\mu}}[p],$	$\forall t \in T, \forall p \in {}^\bullet t : \mathbf{Pre}[p, t] = f$;
(c_5)	$\alpha_f \, \boldsymbol{\chi}[t] \, \overline{\mathbf{s}}[t] \geq OUT(p,t) \, \overline{\boldsymbol{\mu}}[p] - \lvert cd(t) \rvert (\alpha_f - 1),$	$\forall t \in T$ persistent, age memory or immediate: ${}^\bullet t = \{p\}$, $\mathbf{Pre}[p, t] = f$, $A(f) = 1$;
(c'_5)	$\boldsymbol{\chi}[t] \, \overline{\mathbf{s}}[t] \geq k \frac{OUT(p,t)\overline{\boldsymbol{\mu}}[p] + \lvert cd(t) \rvert (1 - k\alpha_f)}{OUT(p,t) + \lvert cd(t) \rvert (1 - k\alpha_f)},$	$\forall t \in T$ persistent, age memory or immediate: ${}^\bullet t = \{p\}$, $\mathbf{Pre}[p, t] = f$, $A(f) = 1$, $\wedge \, k \in \mathbb{N} : k\alpha_f \leq \mathbf{b}[p] \leq (k+1)\alpha_f$;
(c_6)	$\alpha_f \boldsymbol{\chi}[t] \, \overline{\mathbf{s}}[t] \geq OUT(p,t)\overline{\boldsymbol{\mu}}[p] + \lvert cd(t) \rvert (1 - \alpha_f) - OUT(p,t)\mathbf{b}[p] f_q,$ where $f_q = \lvert cd(t) \rvert - \frac{OUT(q,t)\overline{\boldsymbol{\mu}}[q] + \lvert cd(t) \rvert (1-\alpha_g)}{OUT(q,t)\mathbf{b}[q] + \lvert cd(t) \rvert (1-\alpha_g)},$	$\forall t \in T$ persistent, age memory or immediate: ${}^\bullet t = \{p, q\}$, $\mathbf{b}[p] \leq \mathbf{b}[q]$, $\mathbf{Pre}[p, t] = f, \mathbf{Pre}[q, t] = g, A(f) = A(g) = 1$;
(c_7)	$\alpha_1 \boldsymbol{\chi}[t] \, \overline{\mathbf{s}}[t] \geq OUT(p1,t)\overline{\boldsymbol{\mu}}[p1] - \lvert cd(t) \rvert (-\alpha_1 + 1) - OUT(p_1,t)\mathbf{b}[p_1] \max_{1 \leq j \leq n} f_j,$ where $f_j = 1 - \frac{OUT(p_j,t)\overline{\boldsymbol{\mu}}[p_j] + \lvert cd(p_j) \rvert (1-\alpha_j)}{\mathbf{b}[p_j]/\lvert cd(p_j) \rvert - \alpha_j + 1},$	$\forall t \in T$ persistent, age memory or immediate: ${}^\bullet t = \{p_1, \ldots, p_n\}, \mathbf{b}[p_1] \leq \mathbf{b}[p_j], j \in \{2, \ldots, n\}$, $\mathbf{Pre}[p_i, t] = f_i$, $A(f_1) = 1$;
(c_8)	$\overline{\boldsymbol{\mu}}, \boldsymbol{\chi}, \boldsymbol{\sigma} \geq \mathbf{0}$	

Table 2.1: Linear programming problem.

A complementary classification of the techniques for the quantitative analysis of the SPNs based on the quality of the results obtained can be: exact, approximation and bounds.

- *Exact* techniques are mainly based on algorithms for the automatic construction of the infinitesimal generator of the isomorphic Continuous Time Markov Chain (CTMC). Refer to [Bal98] for numerical solutions of GSPN systems. In general, these techniques suffer the referred state explosion problem. In [Don94] the model is decomposed for the exact computation of the steady state distribution of the original model. Others techniques based on *tensor algebra* to obtain exact solutions are proposed in [CDS99].

- *Approximation* techniques do not obtain the exact solution but an approximation. Some of them substitute the computation of the isomorphic CTMC by the solution of smaller components [CCJS94]. In [PJ02] techniques based on *divide and conquer* strategies are presented for the approximated computation of the throughput.

- Finally, *bounds* are techniques that offer the further results from the reality. Nevertheless, they can be useful in the early phases of the software life-cycle. Since our work is proposed to calculate performance estimates in these preliminary stages, bound seems to be a good alternative. In the following, we explore them.

Performance bounds

Performance bounds [Cam98a] are useful in the preliminary stages of the software life-cycle, in which many parameters are not known accurately. Several alternatives for those parameters should be quickly evaluated, and rejected those that are clearly bad. The benefits of the bounds come from the fact that they require much less computation effort than exact and approximation techniques.

As in this thesis we propose a process to estimate performance parameters in the early stages of the software life-cycle and the complexity of this kind of systems is a reality, the use of bounds becomes interesting in some situations to avoid the state explosion problem.

Bounds for GSPNs can be computed from the solution of proper linear programming problems (LPP), therefore they can be obtained in polynomial time on the size of the net model, and they depend only on the mean values of service time associated to the firing of transitions and the routing rates associated with transitions in conflict and not on the higher moments of the probability distribution functions of the random variables that describe the timing of the system.

The idea, valid for GSPNs as well as for SWNs, is to compute vectors that maximize or minimize the throughput of a transition or the average marking of a place among those verifying the operational laws and other linear constraints. For the case of SWNs the LPP appear in table 2.1, for details about how to obtain each constraint refer to [CAC+93].

2.2 The Unified Modeling Language

In this section we assume that the reader knows the basis of the Unified Modeling Language [BJR99] (UML). Then, we just summarize the semantics and the most important concepts given in [Obj01] for the subset of diagrams that are related with our proposal. For details in the descriptions or more advanced issues refers to [Obj01].

The UML is a semi formal language developed by the Object Management Group [Obj01] to specify, visualize and document models of software systems and non-software systems too. UML has gained widespread acceptance in the software development process for the specification of software systems based on the object-oriented paradigm.

UML provides several types of diagrams which allow to capture different aspects and views of the system. A UML model of a system consists of several diagrams which represent the functionality of the system, its static structure, the dynamic behavior of each system component and the interactions among the system components.

UML defines twelve types of diagrams, divided into three categories: static diagrams, behavioral diagrams and diagrams to organize and manage application modules.

- Static diagrams are intended to model the structure (logical and architectural) of the system. They are: Class diagram, object diagram, component diagram and deployment diagram.
- Behavioral diagrams are intended to describe system dynamics, and they are of five kinds: Sequence diagram, collaboration diagram, use case diagram, statechart diagram and activity diagram.
- Diagrams to organize modules allow to reduce complexity of the system. There exist packages, subsystems and models.

Behavioral diagrams constitute a major aim in this work since the most performance issues of systems can be represented by means of them.

The *sequence diagram* (cfr. [Obj01] section 3.60) specifies a set of partially ordered messages, each specifying one communication, e.g. signals or operation invocations, as well as the roles to be played by the sender and the receiver. They represent patterns of interaction among the objects. The *collaboration diagram* (cfr. [Obj01] section 3.65) is a non temporal view of the sequence diagram. The *use case diagram* (cfr. [Obj01] section 3.54) is used to requirements gathering. The *statechart diagram* (cfr. [Obj01] section 3.74) describes possible sequences of states and actions through which the modeled element can proceed during its lifetime as a result of reacting to discrete events (e.g., signals, operation invocations). They allow to specify the behavior of individual entities of the system, such as the class objects. The *activity diagram* (cfr. [Obj01] section 3.84) is a special case of statechart in which the states represent the execution of actions.

The semantics of the behavioral UML diagrams, is specified in the *Behavioral Elements* package (cfr. [Obj01] section 2.8), see Figure 2.1, which is decomposed

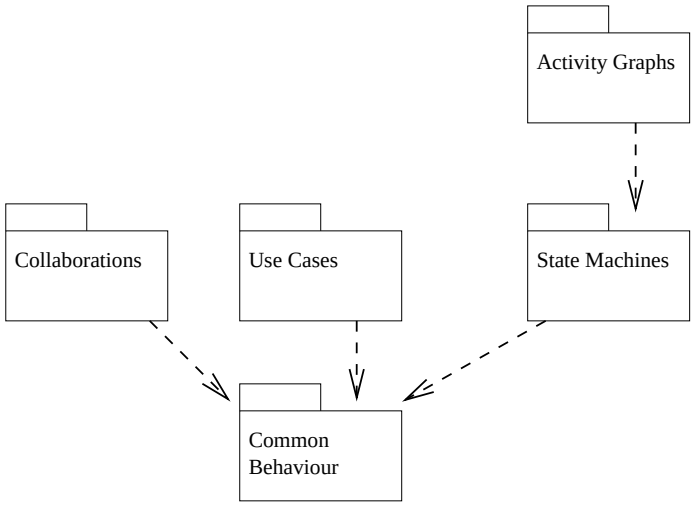


Figure 2.1: Behavioral elements package.

into the following subpackages: Common behavior, collaborations, use cases, state machines and activity graphs.

Actually each behavioral diagram maps into one of the behavioral package in the following fashion:

- The use case diagram maps into the use case package.
- The collaborations diagram maps into the collaboration diagram.
- The sequence diagram maps into the collaboration diagram.
- The statechart diagram maps into state machines package.
- The activity diagram maps into activity graphs package.

In the following sections we describe each one of the behavioral packages paying special attention to the state machines package since it is the core of this work.

2.2.1 Common behavior package

The common behavior package (cfr. [Obj01] section 2.9) specifies the core concepts to support the rest of the packages in the behavioral package.

The most important elements in this package to understand our work are the following: signal, action, instance, link and object.

An *action* specifies an executable statement that results in a change in the state of the model. It can be carried out by sending a message to an object or modifying a link or a value of an attribute. Among the different kind of actions must be remarked the following:

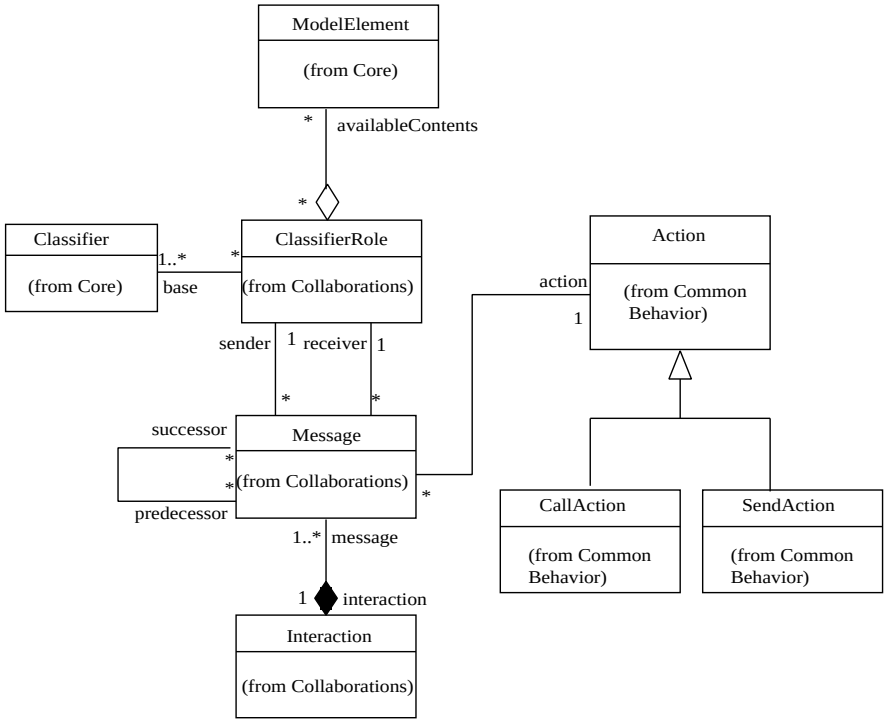


Figure 2.2: Partial view of the UML Collaborations metamodel.

- A *call action* is the invocation of an operation on an instance.
- A *create action* creates an instance of some classifier.
- A *destroy action* destroys an object.
- A *send action* is the (asynchronous) sending of a signal.

An *instance* is an entity with a state that reflects the effects of its operations. Connections between instances are *links*, i.e. instances of an association. A *signal* is a specification of an asynchronous stimulus communicated between instances. An *object* is an instance of a class and may originate from several classes. It is structured and behaves according to its class.

2.2.2 Collaborations package

The collaborations package (cfr. [Obj01] section 2.10) provides the means to define communication patterns performed by instances to carry out a specific task, i.e. *interactions*. Moreover, this package allows the structural description of the participants in the interactions, *collaboration*. Figure 2.2 represents a partial view of the package.

An interaction is defined in the context of a collaboration. It specifies the communication patterns between the roles in the collaboration. More precisely, it contains a set of partially ordered messages, each specifying one communication. A collaboration may be attached to an operation or a classifier, like a use case, to describe the context in which their behavior occurs; that is, what roles instances play to perform the behavior specified by the operation or the use case. A collaboration is used for describing the realization of an operation or a classifier.

A collaboration may be expressed at different levels of granularity. A coarse-grained collaboration may be refined to produce another collaboration that has a finer granularity.

Collaborations can be used for expressing several different things, like how use cases are realized, actor structures of ROOM [SGW94], OOram [Ree] role models, and collaborations defined as in Catalysis [DW98].

The most important elements together with the interactions and collaborations in this package related to our work are the ClassifierRole and the Message.

A *classifier role* specifies a restricted view of a classifier, being a specific role played by a participant in a collaboration. A *message* defines a particular communication between instances that is specified in an interaction.

2.2.3 Use cases package

The purpose of a *use case* (cfr. [Obj01] section 2.11) is to define a piece of behavior of a model element without revealing its internal structure. Each use case specifies a service the model element provides to its users; that is a specific way of using the element. A use case describes the interactions between the users and the model element as well as the responses performed by the model element, as these responses are perceived from the outside of the model element. A use case also includes possible variants of this sequence (for example, alternative sequences, exceptional behavior, error handling, etc.).

A use case can be used to specify the requirements of a system, subsystem or class and for the specification of their functionality. Moreover, the use cases state how the users should interact so the entity will be able to perform its services. Use cases specifying class requirements are mapped onto operations of the classes.

An actor is a specific user of the use case that communicates exchanging message instances that are expressed by *associations* between the actor and the use case.

A use case may be related to other use cases by extend, include and generalization relationships:

- An *include* relationship defines that a use case contains the behavior defined in another use case.
- An *extend* relationship defines that instances of a use case may be augmented with some additional behavior defined in an extending use case.
- A *generalization* relationship implies that the child use case contains all the elements defined in the parent use case and may define additional behavior.

2.2.4 State machines package

The state machines package (cfr. [Obj01] section 2.12) (UML SMs) is a subpackage of the behavioral elements package. It specifies a set of concepts that can be used for modeling discrete behavior trough finite state-transition systems. These concepts are based on concepts defined in the foundation package as well as concepts defined in the common behavior package. This enables integration with the other subpackages in behavioral elements. Moreover, they provide the semantic foundation for activity graphs. This means that activity graphs are simply a special form of state machines.

UML SMs incorporate several concepts similar to those defined in ROOMcharts, a variant of statechart defined in the ROOM [SGW94] modeling language. Actually, UML SMs are defined as an object-based variant of Harel statecharts [Har87, HG96, HN96], major differences are identified in [Obj01], in the following we remark the more interesting:

- Harel statecharts specify behaviors of processes, however UML SMs specify behavior of individual entities or interactions,
- UML SMs do not support event conjunction,
- UML SMs support the notion of synchronous communication between SMs,
- UML SMs transitions are not based on the zero-time assumption.

UML SMs can be used to specify behavior of various elements that are being modeled. For example, they can be used to model the behavior of individual entities (e.g., class instances) or to define the interactions (e.g., collaborations) between entities.

A deep description of the UML SMs, as usual in the UML Semantics Specification [Obj01], is given in three sections: Abstract Syntax, Well-formedness rules and Semantics. In the following, we briefly outline them:

- **Abstract syntax.** The abstract syntax for UML SMs is expressed graphically in Figure 2.3, which covers all the concepts of the state machine graphs such as state, transitions, events.
- **Well-formedness rules.** The *static semantics* of the UML SMs, except for multiplicity and ordering constraints, are defined as a set of invariants of an instance of the metaclass. These invariants have to be satisfied for the construct to be meaningful. The rules thus specify constraints over attributes and associations defined in the metamodel. Each invariant is defined by an Object Constraint Language (cfr. [BJR99] Chapter 7 or [Obj01] Chapter 6) expression together with an informal explanation of the expression.
- **Semantics.** The *execution semantics* of UML SMs is defined using natural language. For convenience, the semantics is described in terms of the operations of a hypothetical machine that implements a state machine specification. This is for reference purposes only. Individual realizations are free to choose any form that achieves the same semantics. The key components of this hypothetical machine are:

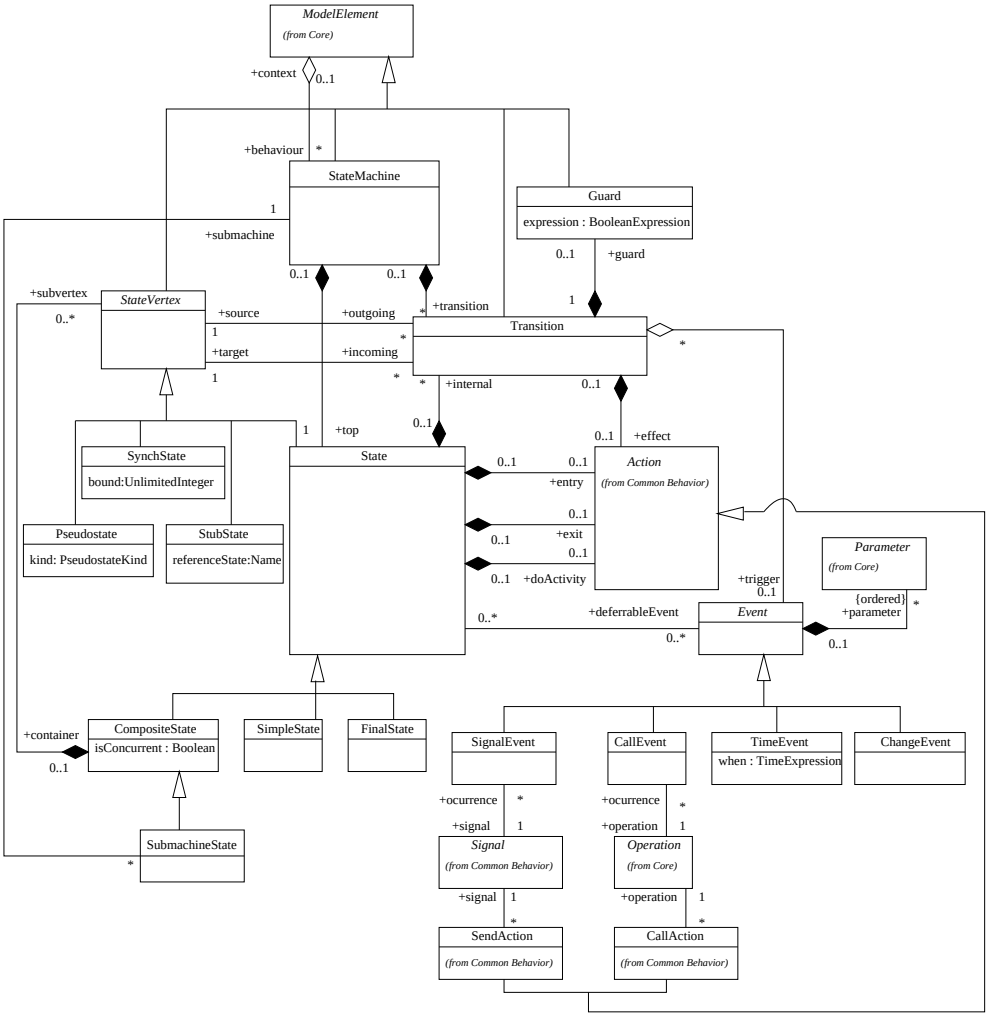


Figure 2.3: UML State Machines metamodel.

- an *event queue* which holds incoming event instances until they are dispatched
- an *event dispatcher mechanism* that selects and de-queues event instances
- an *event processor* which processes dispatched event instances. The semantics of event processing is based on the *run-to-completion* assumption, that means that an event can only be dequeued and dispatched if the processing of the previous current event is fully completed (cfr. [Obj01] section 2.12.4.7).

Elements of the state machines

The translation for the UML SMs proposed in chapters 4 and 5 is based in the UML SMs metamodel, i.e. the abstract syntax, therefore this translation takes as input the elements in Figure 2.3 to obtain LGSPNs in a compositional manner. In the following we stress some important elements (classes and relations) of the UML SMs metamodel.

A *state* models a situation during which some invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some activity.

Some of the associations involving the class state are heavily related with the class *action* (cfr. section 2.2.1). These associations are the following:

- $entry : State \longrightarrow Action$, is a relation that associates to s an optional *Action* that is executed whenever s is entered regardless of the transition taken to reach s. If defined, entry actions are always executed to completion prior to any internal activity or transitions performed within s.
- $exit : State \longrightarrow Action$, is a relation that associates to s an optional *Action* that is executed whenever s is exited regardless of which transition was taken out of s. If defined, exit actions are always executed to completion only after all internal activities or transitions actions have completed execution.
- $doActivity : State \longrightarrow Action$, is a relation that associates to s an optional *Action* that is executed while being in s. The execution starts when s is entered, and stops either by itself, or when s is exited, whichever come first.

Moreover, in order to understand the semantics of a state, the following associations must be remarked:

- $internal : State \longrightarrow Transition$, is a relation that associates to s a set of transitions that, if triggered, occur without causing a state change. This means that the entry or exit actions of s will not be invoked.
- $outgoing : StateVertex \longrightarrow Transition$, is a relation that, given a state $s \in State$ playing a *StateVertex* role, associates a set of transitions that, if triggered, occur

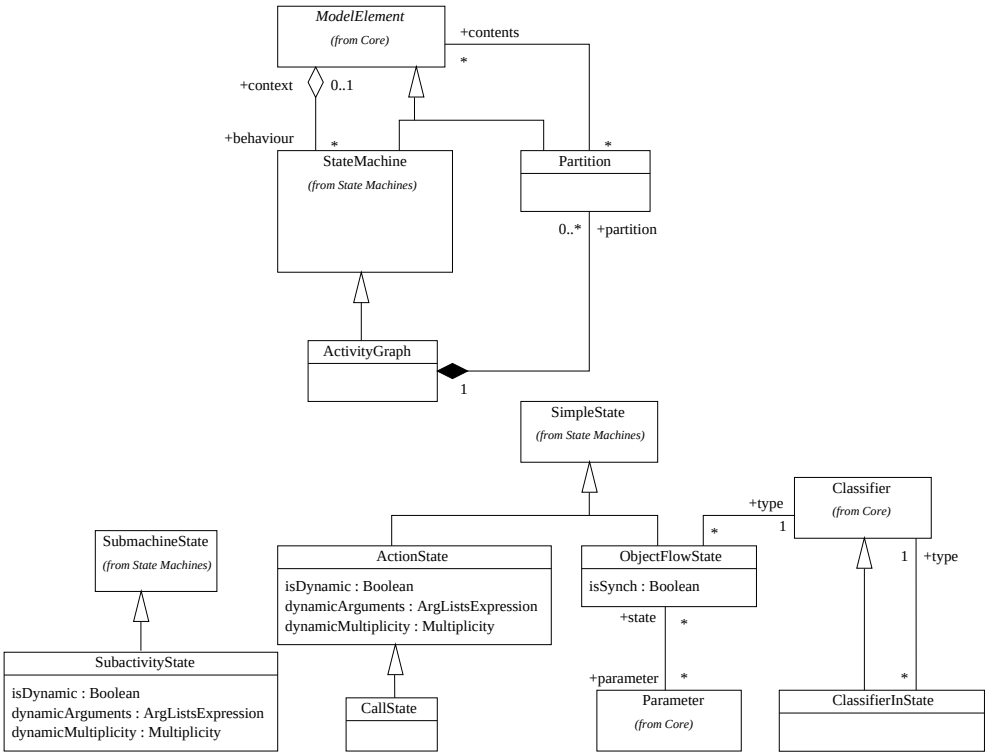


Figure 2.4: Partial view of the UML Activity Graphs metamodel.

causing a state change. This means that the entry or exit actions of s are invoked.

- $deferrableEvent : State \longrightarrow Event$, is a relation that, given a state $s \in State$, associates a set of events that can be retained in it.

2.2.5 Activity graphs package

The activity graphs package (cfr. [Obj01] section 2.13) has few semantics of its own, it defines an extended view of the state machine package, being a special case of a state machine that is used to model processes involving one or more classifiers. A partial view of the package is given in Figure 2.4.

Activity graphs can be attached to packages, classifiers (including use cases) and behavioral features. In this work, we propose them to carry out a detailed description of the performance features of an action in a state machine.

Most of the states in an activity graph are *action states* that represent atomic actions, that are specialized into *call states* to represent the execution of a call action as its entry action. *Subactivity states* represent a nested activity that has some duration

and internally consists of a set of actions or more subactivities. The generation of an object by an action in an action state may be modeled by an *object flow state*. The states can be organized in *partitions* according to a criteria.

Transitions, that are inherited from the state machines package, are triggered by:

- the completion of an action,
- the availability of an object in a certain state,
- the occurrence of a signal, or
- the satisfaction of some condition.

The translation given in chapter 6 for the elements in the activity graph metamodel, see Figure 2.4, follows the same pattern as given in the previous section for the UML SMs, i.e. these elements are mapped in a compositional manner into LGSPNs.

2.3 Software Performance Engineering

The term software performance engineering (SPE) was first introduced by C.U. Smith in 1981 [Smi81]. Several complementary definitions have been given in the literature to describe the aim of the SPE. Among them we remark the followings:

- In [Smi90], the SPE is proposed as a method (a systematic and quantitative approach) for constructing software systems to meet performance objectives, taking into account that SPE augments others software engineering methodologies but it does not replace them.
- SPE is defined in [SS01] as a collection of methods for the support of the performance-oriented software development of application systems throughout the entire software development process to assure an appropriate performance-related product quality.
- Finally, in [Smi01] new perspectives for SPE are devised, then proposing that SPE must provide principles, patterns [MCM00a, GM00] and antipatterns [SW00] for creating responsive software, the data required for evaluation, procedures for obtaining performance specifications and guidelines for the types of evaluation to be conducted at each development stage.

It is important to remark that the previous definitions emphasize that SPE cannot be placed outside the context of software engineering. It contrasts with other engineering fields, such as telecommunication, where performance practices have been applied successfully in "isolation", i.e. not explicitly while developing the engines. Moreover SPE, as pointed out in [SS01], reuses and enlarges concepts and methods from many others disciplines such as: Performance management, performance modeling, software engineering, capacity planning, performance tunning and software quality assurance.

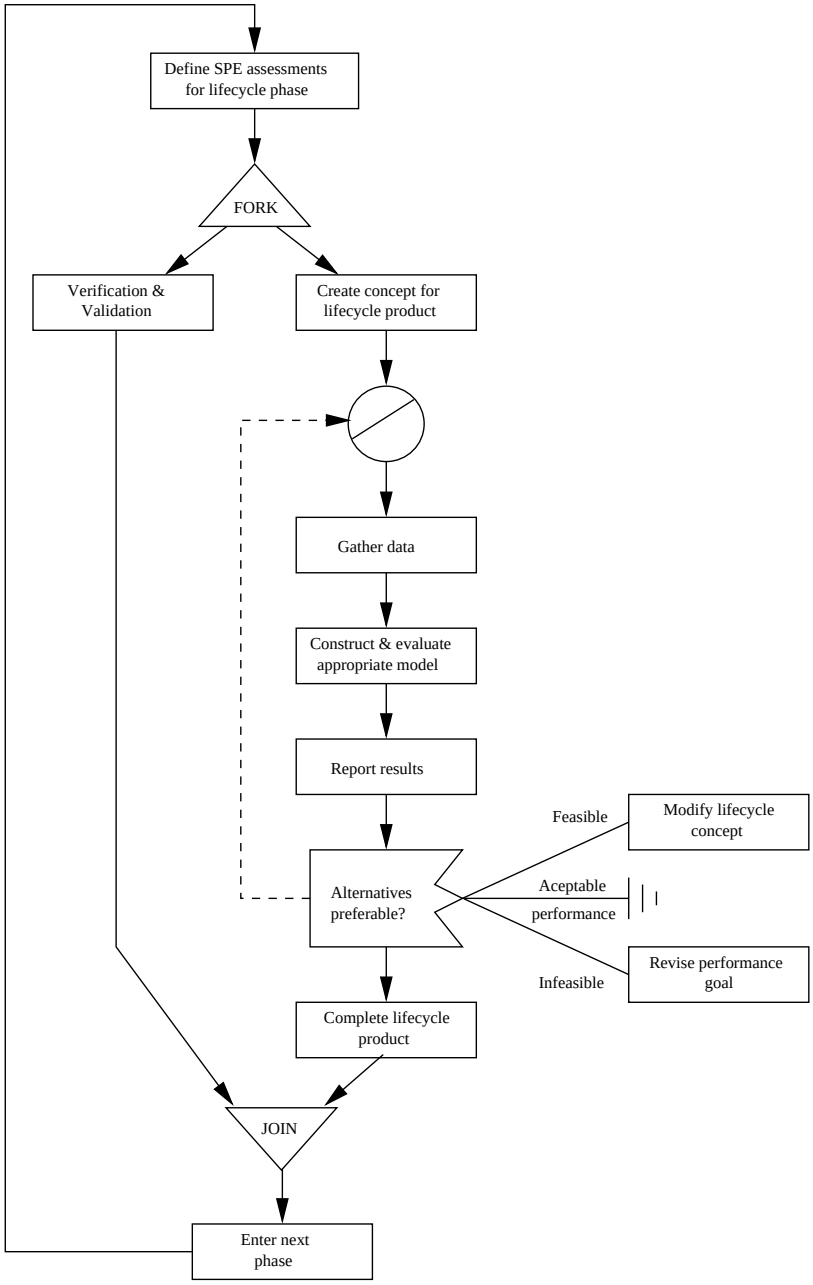


Figure 2.5: Software Performance Engineering Process (taken from [Smi90]).

In this work, we emphasize the use of the SPE in the early phases of the life-cycle effectively avoiding the "fix-it-later" approach, as well as in [Smi90], that proposes evaluating performance of software systems in the early stages of the development process. Thus, if performance problems are detected, it will be easier and less expensive to take the appropriate design decisions to solve them. Although our proposal focuses only in these uses of the SPE, in our opinion the use of good SPE practices must be extended along the complete development process, then facilitating to meet performance objectives in each stage. In this way the best choices of software architecture, design and implementation can be considered.

The SPE process that appears in Figure 2.5 was formerly proposed in [Smi90] and still remains as reference for a very general proposal to establish the basic steps that a SPE process should consider. In the following we recall this proposal.

Firstly, it must be defined which goals or quantitative values are of interest, obviously it changes from one stage of the software life cycle to other and also among different kind of systems. Business systems define performance objectives in terms of responsiveness as seen by the system users while reactive systems take into account event responses or throughput. The *concept* for the life cycle product also depends on the stage of the life cycle and it refers to the software architecture, the design, the algorithms, the code and so on. Responsive architectures and designs are conceived by applying SPE principles, patterns and antipatterns. Subsequently, data *gathering* is accomplished by defining the proper scenarios interpreting how the system will be typically used and its possible deviations (defining upper and lower bounds when uncertainties). The construction and evaluation of the performance model associated with the system is one of the fundamentals in SPE. Later in this section we explore common types of performance models proposed in the literature. In our approach the performance model is defined in terms of GSPNs or SWNs, this kind of models can be evaluated either by analysis or simulation. If the results obtained do not agree with those predicted in the original concept, alternatives for improvement are considered. If a feasible alternative exists the concept is modified, if none performance objectives should be revised. Validation and verification of the performance model are on-going activities of the SPE process.

Performance measures in SPE typically include resource utilizations, waiting times, execution demands and response time. In section 2.1.5 these performance measures were interpreted from a Petri net point of view in terms of places and transitions.

The common paradigms of stochastic models used in SPE are recalled in the following paragraphs.

Queuing models Queuing network modeling is defined in [LZSS84] as a particular approach to computer system modeling in which the computer system is represented as a *network of queues* which is evaluated *analytically*. A network of queues is a collection of *service centers*, which represent system resources, and *customers*, which represent users or transactions. Analytic evaluation involves using software to solve efficiently a set of equations induced by the network of queues and its parameters.

Queuing networks models have been extended to represent finite capacity queue and memory constraints, simultaneous resource possession and synchronization and concurrency constraints. Moreover, the "layered queuing network"(LQN) model [RS95] allows client-server communication in concurrent and distributed software systems. In a LQN model a server may become client (customer) to other servers while serving its own client requests.

Stochastic process algebras Process algebras are abstract languages used for the specification and design of concurrent systems, where systems are modeled as collections of entities, called agents, which execute atomic actions. Stochastic process algebras [HR98] (SPAs) incorporate the features of the process algebras to performance modeling, being the most important, the *compositionality* of the models. SPAs associate a random variable, representing duration, with each action.

Stochastic Petri nets The stochastic extension of the Petri net formalism has been discussed in section 2.1.2. The performance models in this work are expressed in terms of stochastic Petri nets.

2.4 Related work

In this section we revise the work in the literature related with ours, remarking its features and comparing them with our proposals. We have tried to carry out an exhaustive revision by focusing on all the related areas that have arisen while developing the work. Then we first revise the most important works in methods and methodologies in the software performance evaluation, second the pattern based approach recently emarged is analyzed, third the profile adopted for the annotation of UML design is addressed and finally the attempts to formalize some UML diagrams are studied. In the following, we briefly put in context each one.

As we commented, the term software performance engineering was coined by C.U. Smith in [Smi81]. In this thesis SPE is proposed to evaluate performance of software systems in the early stages of the development process. One of the preliminary works in SPE was [Smi90], many authors point it as the beginning of this field as it is understood today. Nevertheless, it is important to remember that other works also were relevant in the origins of the SPE, such as [GCD73, RWS+78, SB75]. A good review of the evolution of the SPE can be found in [Smi01]. In sections 2.4.1, 2.4.2 and 2.4.3, the most important works in software performance engineering in the last five or six years are analyzed and compared. Moreover, for a good survey of the different approaches for performance evaluation based on UML diagrams we recommend [BS01]. Some of these approaches augmented the notation of the UML with performance features, this topic is addressed in section 2.4.4. The precise UML group [pUM] has as goals to state the UML as a precise (i.e. well defined) modelling language, in this context several works have arisen, some of them trying to give formal semantics to the state machines and to the activity diagrams; since in this work we give formal

semantics to these diagrams to construct a performance model, the work in this area is revised in section 2.4.5.

2.4.1 Preliminary works in SPE

In this section we review the works that we consider as preliminary to this thesis. They share similarities such as: SPE is driven by models, they were proposed before or in the first specific conference of the SPE [SWC98], they began to devise UML as the design notation. Summarizing, they were the firsts attempts to position SPE as we understand it today. Concretely, we focus on three works: the Permabase project [WLAS97, UH97], the works developed at Carleton University [HRW95, WHSB98, FW98, Woo00, WHSB01] and the works by Pooley and King [KP99, Poo99, PK99].

The Permabase project started late 1995 was jointly developed by the University of Kent, the BT Labs and ERA Technology. The goal of the project was to define an architecture that enables early performance predictions of object-oriented software designs combined with descriptions of the network configuration and system workload. The architecture is composed by a kernel called Composite Modeling Data Structure, by a modeling tool (Rose [Rat01]), by an execution environment and workload tool (Configurator [Pen96]) and by engine to simulate the models (SES Workbench [SES01]). The UML diagrams developed with Rose are augmented with performance annotations.

Concerning to our work, the main differences are: Since our approach is also based on UML, it can be used within the object oriented paradigm, but the Permabase project is constrained to this paradigm. The performance annotations for the UML models are shortly described then taking into account only the load of the messages, the time of the methods and parameters for the hardware. It can be noted that our approach is more elaborated in this aspect (cfr. chapter 3). The tool used for obtaining results allows only the simulation of the models, our approach is feasible also with simulation and analytical techniques. As our approach, Permabase has been applied in real projects (video library retrieval system, voice-conferencing system and cache/server example), but it has not been possible to find documentation about them but just parts of some involved diagrams.

The works developed at the Carleton University have in common that they use the layered queuing network [RS95] (LQN) formalism to represent the performance model of the software system. In the following we revise the most important ones.

One of the earliest works was [HRW95]. Although some deficiencies can be identified, such as it does not use the UML, we highlight it because important features for the SPE were devised, such as the prediction in the early stages or the automatic generation of the performance model. The process to automatically generate performance models for distributed and concurrent software systems is proposed in five steps: a prototype of the system is created using an object-oriented environment called MLog; the prototype is executed to record "angio events" and "angio traces"(containing behavior information and dynamic details such as data dependent branching or task

identity); the traces are reduced to "workthreads" using resource functions; the performance model in terms of QNs or LQNs is constructed and evaluated; finally if the results do not match with the objectives the design must be improved. The process was applied to an online transaction processing system obtaining a LQN model.

The differences with our approach are the following. We consider the final system designs to obtain the performance model but this approach works with prototypes; we use UML in the design instead MLog; our formal performance model is in terms of Petri nets, they use QN or LQN; they introduce concepts such as "angio traces" and "angio events" to represent ways to execute the system that are not common in the software vocabulary.

In [WHSB98, WHSB01] a wideband approach for the performance prediction of software systems is presented, meaning by wideband that it can be applied to a wide range of design descriptions and parameters, throughout the entire life-cycle. The application domain of the proposal is restricted to the real time field using ROOM [SGW94] as a design methodology. The performance model is created in terms of LQNs with the information obtained from the design models, the execution traces ("angio traces") and the testing of the deployed system. The Performance Analysis by Model Building tool (PAMB) assists this approach, and takes the following steps: Define scenarios and create the design using ObjectTime; execute the scenarios to automatically obtain a "sub-model" for each one; enter the performance measures, create an experiment and run the LQN solver to obtain results.

In our opinion the wideband approach is a good receipt of the strategies that should be followed in the SPE process. But compared with our approach it lacks of the following. It is not clear how the system load or routing rates are obtained, nevertheless they claim that data gathering is automated by a tool. It is not explained how the performance model is obtained but we give the functions that perform it. The design models proposed are not enough to express all the system load. The use of ROOM instead UML is a disadvantage in our opinion. Similarly to our proposal the performance models can be analyzed or simulated.

Finally, we review the works of Pooley and King. They follow the approach to annotate the UML diagrams to obtain by a translation procedure performance models in terms of stochastic Petri nets or stochastic process algebra. In [PK99] some UML diagrams are revised to find its role in the SPE. For the use cases, they consider that actors may represent system workload. In the implementation diagrams they identify servers and jobs. Sequence diagrams are considered as discrete event simulators, while the collaboration diagram may allow steady state analysis. A combination of the state and collaboration diagrams is pointed as the best approach to simulate UML. In [KP99], GSPN models are produced starting from UML diagrams: the main difference with our work is that the construction is described only at an intuitive level, through an example, and no systematic approach to the translation is given.

2.4.2 Approaches for SPE based on UML

In this section we focus on the contemporary works to this thesis. They explicitly consider UML as the design language for the SPE process and are more mature than those considered in section 2.4.1 since they are later in time, some of them were presented in [WGM00] and the others even in posterior relevant conferences or journals. Concretely, we focus on four works that in our opinion describe perfectly the state of the art in the SPE field between the conferences [WGM00] and [IBB02], they are the works of Cortellessa [CM00, CDI01], the software architecture approaches [ABI01, AABI00], the work of De Miguel [dMLH+00] and finally the research at Carleton University [PS02].

Cortellessa et al. propose methods that from software models are capable to generate systematically and automatically performance models in terms of extended queuing networks [Lav83] (EQN) or layered queuing networks [RS95] (LQN). In [CM00] the methodology to obtain EQN is proposed while in [CDI01] the LQN approach is developed. Since both approaches share commonalities, we are going to study and compare with ours the first one [CM00]. The performance model generated consists of two parts, the software model that is based on execution graphs [Smi90] (EG) and the machinery model based on EQN. The combination of the EGs and the EQNs gives a complete parametrized performance model. The methodology proposed uses the use case diagram to derive the user profile, the sequence diagrams to derive an EG and the deployment diagram to identify hardware/software relationships that improve the accuracy of the performance model. The steps of the methodology are the following: The user profile is obtained from the use case diagram; for each use case a set of sequence diagrams are developed and processed to obtain a meta-EG, it is carried out by means of a given algorithm; the deployment diagram is used to obtain the EQN model of the hardware platform and to tailor the meta-EG obtaining a EG-instance; numerical parameters are assigned to the EG-instance; finally, the EG-instance is combined with the EQN model using the SPE approach [Smi90]. Concerning our approach, firstly we must recognize that we own to this work the annotations performed in the use case diagram, we adopted them because in our opinion it is a clear and nice way to define the role or the profile of the user in the system from a performance viewpoint. Another important feature is that the use of the deployment diagram allows to avoid the infinite resource assumption that is present in our work since hardware resources can be modeled. On the other hand, since our approach makes use of the statecharts and the activity diagrams the performance modeling power in our proposal is increased, allowing to model low level description of activities as well as reactive object behavior. The performance model in terms of queuing networks instead of PN is a difference that has been commented previously.

In the field of software architectures [SG96] (SA) the works in [ABI01, AABI00] are representatives for the generation of performance models in terms of queuing networks (QN). [ABI01] proposes a method to generate the performance model from a labeled transition system (LTS) that specifies the global dynamic behavior of a SA. Although a LTS is not a UML diagram, the statecharts have been formalized using LTS in [US94] then the equivalence among LTS and statecharts can be assumed. The

methodology starts with the description of a SA by a LTS that is analyzed by a two-phases algorithm. The first phase examines the paths in the LTS to single out the pairs of interacting components, the second phase analyzes and compares the pairs to derive the QN topology. After, quantitative parameters are defined to obtain the complete QN model. Our approach is more general in the sense that it is not restricted to only one specification tool, i.e. LTS or statecharts. In [AABI00] the QN model is obtained exclusively from sequence diagrams, newly the approach is too restricted in its modeling power if it is compared with ours. In an high abstraction level this approach is quite similar to the previous one, the change is that the algorithm deals with the sequence diagram instead with the LTS.

De Miguel developed in [dMLH+00] a UML profile for hard real time systems using the UML extension mechanisms (stereotypes, tagged values and constraints). It was motivated by the lack (or poor way) of UML to represent specific real time issues such as: system load, temporal restrictions, quality of service (QoS), resource consumption or scheduling. This profile was included in a UML based CASE tool and to identify the extensions it takes care of the UML semantics, the automatic generation of analysis and simulation models and the software implementation of scheduling analysis theorems. Concretely, UML is extended by means of seven constraints (periodic timing, aperiodic timing, access protocol, execution time, remote message, priority and location) and nine new or redefined stereotypes (processor, network, cyclic, sporadic server, concurrent, activity diagram, activity state, call state and object flow state). As an example of constraint, the "execution time" identifies the maximum execution time of some UML elements ($Utilization\ Time \leq Time$), this constraint can be attached to operations, call states or action states. The profile was applied to an air traffic control system using class diagrams and activity diagrams to develop it. In our opinion, this approach is close related with that presented in section 2.4.4 than with ours, then some of the comments that will be given still remain valid in this context. This proposal differs from ours because it is centered in hard real time domain. The major criticism from our viewpoint is that it is omitted how the scheduling and simulation models are obtained from the extended UML diagrams. Another point is that statecharts are discarded because in their opinion they are more complex than activity diagrams. Obviously we do not share this viewpoint since activity diagrams have all the expressivity of the statecharts plus that given by the semantics of the activity graph package.

Petriu and Shen present in [PS02] a graph-grammar based method to automatically obtain performance models in terms of LQN [RS95] from UML descriptions augmented with the UML performance profile (cfr. section 2.4.4). The work develops an algorithm that using as input an XML file with the UML description obtains the corresponding LQN model. Following the SPE approach [Smi90], from each performance annotated scenario a LQN is obtained and after these models are merged. Each LQN is obtained in two steps: The structure is obtained from the collaboration and deployment diagrams; and the entries, phases, visit ratio parameters and execution time demands are obtained from the activity diagrams by a translation process, this process is studied and compared in section 2.4.5 since we have dedicated that sec-

tion to compare our translation of state machines and activity diagrams with others. With respect to our approach, it is similar in the sense that from the activity diagram the performance model is obtained, but we use also the use case diagram and the statechart diagram and they use the deployment diagram. To represent scenarios we propose the sequence diagram while they discard it considering that it is not well defined in UML, in our opinion this misspecification of the sequence diagram can be avoided with a coherent interpretation.

2.4.3 Patterns and antipatterns in SPE

Design patterns [GHJV95] are *"descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context"*, while antipatterns [BMMI98], being conceptually similar to patterns, document common mistakes made during software development as well as their solutions.

Although the use of the design patterns and antipatterns with reusability or maintainability purposes is "classic" in the literature, the SPE field has recently payed attention to them. To our knowledge only the works [MCM00a, SW00, GM00, VGNP00, PS97] have addressed some issues and possibilities in this context. In our opinion it is a new field where the SPE community should invest a lot of work, in the following years, to take profit of some well-known and documented design problems and solutions from the performance engineering point of view. Section 7.3 recalls our proposal given in [MCM00a].

Smith and Williams suggest in [SW00] the use of antipatterns in the SPE field. Concretely they propose performance antipatterns to document performance problems and their causes found in different contexts but with the same underlying pathology. To illustrate the use of the performance antipatterns, that work discusses performance problems associated with the "god" or "blob" class [BMMI98] antipattern. A "god" class is one that performs most of the work of the system creating performance problems due to the excessive message traffic. They show that distributing intelligence uniformly across the top-level classes provokes a performance gain such that $T_s = M_s \times O$, where T_s is the processing time saved, M_s is the number of messages saved and O is the overhead for message. Moreover, they document and give solutions for three new performance antipatterns: the Excessive Dynamic Allocation, Circuitous Treasure Hunt and One Lane Bridge.

Concerning our approach, the antipattern proposal can be seen as "the other side of the coin". While we propose to enhance the pattern language with new sections that reflect the performance objectives and workload of well-known designs, they propose to identify performance problems also in well-know designs and give solutions to them. In our opinion both approaches are complementary and the combined use of performance patterns and antipatterns should lead to good software performance practices. On the other hand, although their proposal has been applied to solve (document) performance problems in four antipatterns, ours has been applied in a real project, the ANTARCTICA software retrieval system (cfr. chapter 7).

Gomaa and Menascé in [GM00] propose "component interconnection patterns"

for the performance evaluation of distributed software architectures. A component interconnection pattern defines and encapsulates the way client and server communicate with each other. In a high abstraction level their proposal shares similarities with ours: They start with a UML design model of the component interconnection pattern, then provide a performance annotation of the design using XML and finally map the design to a performance model to analyze the performance of the architecture. It must be taken into account that their proposal is completely oriented to model distributed systems using the concepts of components and connectors. In this sense our proposal is more general since it offers the possibility to model these concepts, as well as to represent the performance annotations given for these concepts. More concretely, they use the class diagram and the collaboration diagram to model the interconnection patterns. Although our proposal does not use the class diagram since static features have not been addressed yet, it offers performance extensions for all the UML behavioral diagrams, then gaining modeling power. The use of the XML language to annotate performance features seems a good election, however the OMG group has adopted the tagged value approach [Obj02] as our work did. Finally, the performance models are obtained in terms of queuing networks models but it is not given the mapping function among the two models, then being not possible to compare it with our translation procedure.

The Maisa (Metrics for Analysis and Improvement of Software Architectures) project [VGNP00] proposes an approach where patterns and antipatterns embedded in the design are discovered by a mining tool directly from the activity diagrams. A comparison with our work is not realized since in [VGNP00] the proposal is just positioned itself.

In [PS97] a pattern language different of that considered in this work is extended to give a set of patterns for improving the capacity of reactive systems. These patterns identify some causes that limit the efficiency of distributed layered client-server systems. Since the patterns are defined just for a subclass of reactive systems and using a different language, this proposal is difficult to be compared with ours. But the idea of extending a pattern language with performance features is also present.

2.4.4 UML profile for schedulability, performance and time specification

In chapter 3 we motivate the necessity to annotate the UML diagrams with performance parameters that express the load and the routing rates in the system, at the same time we recall our proposal of performance annotated UML that was given in [MCM00b, LGMC02a]. These concerns have provoked in the SPE community the adoption by the OMG in March 2002 of the "UML profile for schedulability, performance and time specification" [Obj02], that was presented in [IBB02]. In our opinion, the adoption of this profile by the SPE community is of major importance since it will allow to avoid individual efforts and will permit to share research results while "speaking" the same language. This profile reuses much of the effort carried out by the SPE community in this area, for a summary of the proposals to annotate UML

diagrams refer to [BS01]. Taking into account these considerations, it makes sense that in this section we compare our proposal of annotations only with the standard profile. Nevertheless, we want to highlight the work in [CM00] where we have taken the annotations for the use case diagram, and others good works previous to the adoption of the profile such as [dMLH+00, GM00, MGD01], the first two already revised from the methodological viewpoint and the third one as a contributor of the profile.

The UML profile for schedulability, performance and time specification [Obj02] was adopted by the OMG with different purposes:

- to encompass different real-time modeling techniques,
- to annotate UML real-time models to predict timeliness, performance and schedulability characteristics based on analyzing these software models.

Obviously, part of the work developed in this thesis is involved with the second objective, but real-time modeling techniques are out of the scope of this work. Although the profile is oriented to real time systems, the annotations proposed in the performance sub-profile remain valid for more general purposes, even to specify distributed software systems in non real time environments. Here on we will study the performance modeling sub-profile, that extends the UML metamodel with stereotypes, tagged values and constraints to attach performance annotations to a UML model.

The sub-profile for performance modeling provides facilities for:

1. capturing performance requirements within the design context,
2. associating performance-related QoS characteristics with selected elements of a UML model,
3. specifying execution parameters which can be used by modeling tools to compute predicted performance characteristics,
4. presenting performance results computed by modeling tools or found in testing.

In order to meet these objectives the performance sub-profile extends the UML metamodel with the following abstractions. The QoS requirements are placed on *scenarios*, which are executed by *workloads*. The workload is *open* when its requests arrive at a given rate and *closed* when has a fixed number of potential users executing the scenario with a "think time" outside the system. The scenarios are composed by *steps*, i.e. elementary operations. *Resources* are modeled as servers and have *service time*. *Performance measures* (utilizations, response times, ...) can be defined as required, assumed, estimated or measured values. These concepts are represented with stereotypes and tagged values. Concretely, the "performance values" are described by tagged values as in our proposal. The sub-profile allows to map these concepts either into a collaboration or into an activity graph. It allows to construct UML performance designs by means of these two kind of diagrams.

Concerning our proposal, the first and second objectives are met in chapter 3 when defining the performance annotations, since they allow both to annotate performance

requirements in the design models and also QoS features. The fourth objective is accomplished in our proposal as a result of the process given in chapter 7.

The main difference with our proposal is that we identify the role of each UML diagram (use cases, interactions, statecharts and activity diagrams) in the performance process and define the performance annotations for them, while the sub-profile defines a set of performance concepts and maps them into the concepts of each diagram, but only two diagrams (collaborations and activity) have been considered. Finally, both approaches obtain a set of annotated UML diagrams that should be the input to create a performance model in terms of some queuing or simulation model.

2.4.5 Formal semantics for the UML state machines and the activity diagrams

As a very important part of our proposal to study performance of software systems, in chapters 4 and 5 this thesis proposes to (semi)automatically obtain a performance model in terms of GSPNs from UML state machines (UML SMs). The performance model is created by functions that map UML state machines abstractions into GSPNs submodels and working out them taking advantage of the compositional properties of PNs. In short, we propose a translation (an interpretation) of the UML SMs into a mathematical formalism, i.e. PNs. Several works can be found in the literature that translate UML SMs, or UML statecharts (its syntactical form) or Harel statecharts (from which UML SMs are just an object-oriented variant, cfr. section 2.2.4) into some mathematical formalisms. As an example, several semantics for Harel statecharts have been proposed in the literature [HN96, PS91, SSBD99, US94, Lev97, Mar92, MSPT96, DJHP97], but none of them in the context of performance evaluation. Also, a lot of works have been devoted to give formal semantics to UML SMs or UML statecharts as [PL99, LMM99, EW00], but in general with the aim to validate qualitative properties. To our knowledge there does not exist another work that formalizes a translation from UML SMs into the Petri net formalism with quantitative purposes. Translations from UML SMs to Petri nets for the validation of qualitative properties are given in [BP01, SS00], the closer approach to our was given in [KP99] but as we commented in section 2.4.1 it is not possible to stablish a comparison with it since no formalization is proposed in that work, just intuitive ideas are presented. Nevertheless, we do not want to finish the related work of this thesis without at least comment how other authors have addressed the problem of formalizing UML SMs. Therefore, from the plethora of studies remarked before, we want to highlight the work in [PL99] and briefly explain how the authors formalized the UML SMs.

In [PL99], as in our work, a complete formalization of the UML SMs package is given. Their approach is proposed in two parts: firstly the structure of the UML SM is formalized in a rewriting system terms fashion (needed to define the transition selection algorithm) and secondly operational semantics in terms of an execution algorithm compliant with the run to completion step are attached to the formal model. In our approach only the first step is necessary, i.e. to formalize the structure

of the SM as a Petri net, since the operational semantics is implicit in our target formalism, so taking care that the underlaying operational semantics is compliant with the run to completion step, our approach gains in simplicity and in our opinion is more "homogeneous" since it does not need an additional algorithm, the token game abstracts it. In the first step in [PL99], the SM is defined as a set of states of different kinds, then defining a *state configuration* as a term over a signature where the states are operator symbols; the elements of a state (entry, exit, activity, ...) are functions over a given language; each transition is a triplet (s, t, s') where s, s' are terms of the signature and t is a transition name; the attributes of a given transition (trigger, guard, ...) are defined as functions over the corresponding sets. In the second step, a set of definitions (active and enable transitions, conflict transitions, ...) based on the structure proposed in the first step are formalized to be used in an algorithm that implements the run to completion step.

Concerning activity diagrams, the disquisition performed at the beginning of this section remains valid, i.e. there exist a number of works that translate activity diagrams into some mathematical formalisms [PdS01, EW01, PS02]. From them, we have considered interesting to remark the work developed in [PS02], where activity diagrams are translated into layered queue networks (LQN) detailed features 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 (cfr. chapter 6) that is based in the composition of the submodels obtained for each abstraction.

Chapter 3

UML Diagrams for Performance Evaluation: the pa-UML proposal

The study of the software time efficiency (response time, delays, throughput) requires from the software designer the most accurately possible description of the system load and the routing rates. In general, software engineers are not familiar with the notation of performance modeling, moreover this notation is too far from the artifacts they use to model software systems.

In the last years, the software performance engineering community [SWC98, WGM00, IBB02] has dedicated great efforts to incorporate to the software specification languages the abilities to describe system load in understandable terms for software engineers. Then several proposals have been given, for a nice summary see [BS01]. Most of the proposals have considered to augment the language notation, in different ways, to describe the system load. Also, it has been a must for the community to consider as an important feature that the resulting language should be able to (semi)automatically generate a performance model from which performance predictions can be computed.

In this chapter we present our proposal to describe the load in software systems, we will refer to it as "pa-UML" to reflect in some manner that it proposes **p**erformance **a**nnotations in the **UML** [Obj01] language. The proposal takes into account the previous principles, the ability to describe system the load and the routing rates at a software engineer level and the adequacy to generate performance models, and tries to be adequate for the performance evaluation process that we give in chapter 7.

Our proposal considers UML as a specification language instead of the notation of well-established methodologies such as OMT [RBP+91], OOSE [JCJO92] or Fusion [CAB+94] because UML has become a standard among the software engineering community in the last years as a universal language to model software systems. More-

over in the first specific conference on software performance [SWC98], it was "unanimously" decided that UML should be the language of reference for future work. It has been clearly confirmed in the subsequent conferences [WGM00, IBB02].

Unfortunately, UML is not an exception among the software notations, with respect to the lack of the necessary expressiveness to accurately describe the system load and routing rates (consider that the proposal in [Obj02] was not developed when we developed pa-UML [MCM00b]). Moreover, the language lacked also the ability to identify which of its diagrams are relevant for performance evaluation purposes and which of the elements in these diagrams are suitable to describe performance aspects. These tasks are realized in this chapter by exploring the relevant UML diagrams and by identifying the model elements of interest to which associate performance parameters. We will use a UML extension mechanism, the *tagged values*, to associate system load to the selected model elements.

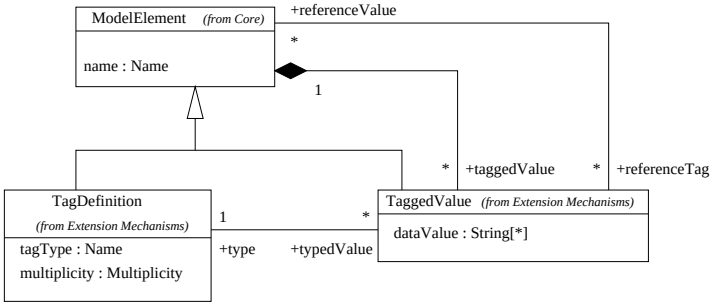
The chapter is organized as follows. Section 3.1 identifies the UML diagrams relevant for performance evaluation and explains how tagged values are used to introduce system load. Sections 3.2, 3.3, 3.4 and 3.5 describe respectively the use case diagrams, the interaction diagrams, the statechart diagrams and the activity diagram. For each diagram a short description taken from the UML document [Obj01] is given, it puts the reader in context but if more information is needed, we refer to the manual; after that, the role of the diagram concerning performance goals is analyzed; and finally, the performance annotations suggested for the diagram are given. The chapter ends by giving some conclusions in section 3.6.

3.1 UML diagrams for performance evaluation

Another way to classify UML diagrams, different from that given in section 2.2, establishes that UML comprises nine different kinds of diagrams that can be grouped in three categories: structural diagrams, behavioral diagrams and implementation diagrams. All of them can be related with performance aspects, it depends on the particular use the modeler assigns to each one in the description of her/his system and on the kind of systems to deal with. In order to have a complete performance description of the system, the UML structural and behavioral models should be used at "modeling" level and implementation diagrams play when configuring the hardware platform for the system.

Among the structural UML diagrams we have focused in this work in the use case diagrams since they will allow to model the usage of the system for each actor. But the most important effort has been devoted to the behavioral diagrams, because traditionally the performance of a system has been studied from its dynamic view. Four different diagrams are proposed by UML to describe behavioral aspects: interaction diagrams (sequence diagrams and collaboration diagrams, both based on the collaboration package), statechart diagrams (based on the state machines package) and activity diagrams (based on the activity graph package, a specialization of the state machines package).

The interaction diagrams will allow to describe the load of the messages sent




```
TaggedValue.name = performance annotation
TaggedValue.dataValue = {a concrete annotation}
TaggedValue.type.name = performance annotation
TaggedValue.type.multiplicity = 1
TaggedValue.type.tagType = [system usage, system load, routing rate]
```

Figure 3.1: Metamodel of the annotations proposed in pa-UML.

among the participants in the system, statecharts will be identified to be used to describe the routing rates in the system and the load of the activities at high level. Activity diagrams will be of interest to assign load to the basic actions in the system.

UML implementation diagrams (component and deployment) can be of interest to evaluate some performance parameters in the latest stages of the design. In these diagrams performance aspects depending on implementation issues such as which database management system or which compiler is going to be used can be modeled. These diagrams are not subject of research in this work since, as we explain in chapter 7, it is focused in the early stages of the development process. Nevertheless, they can be useful to avoid the unlimited resources assumption made in our work. As an example of its application, in [CM00] the deployment diagram is used to obtain a performance model in terms of extended queuing networks formalism for specific hardware platforms.

The UML diagrams studied in this work with performance evaluation purposes are: the use case diagrams, the interaction diagrams, the statecharts and the activity diagrams, that all together will allow to model a wide range of distributed software systems, those that we are focused on.

Fortunately, UML is a language that provides mechanisms to increase the modeling power. Then we propose the use of one of the UML extensions to describe the system load and the routing rates, the extension mechanism selected is the tagged values. The use of the tagged values is proposed in such a way that it neither conflicts with nor contradicts the standard UML semantics. In the following, it is explained how the tagged values are proposed to annotate performance aspects in the UML diagrams.

Figure 3.1 shows the part of the UML metamodel concerning the definition of the tagged values. A tagged value allows information to be attached to any model

	ANNOTATION	KIND*	REFERENCED VALUE
USE CASE DIAGRAM	Probability that an actor executes a use case	A	Association link
SEQUENCE DIAGRAM	Probability of success of a message	B	Message
	Message size	C	Message
STATECHART DIAGRAM	Activity duration	C	Action (with doActivity role)
	Probability of success of a message	B	Transition[1]
	Message size	C	Event
ACTIVITY DIAGRAM	Activity duration	C	Timed transition
	Probability to take the transition	B	Transition (timed or immediate)

*A=System usage, B=Routing rate, C=System load.
[1]It is not associated to an event since the transition can be automatic.

Table 3.1: Summary of the annotations proposed.

element in conformance with its tag definition. On the other hand, a tag definition specifies the tagged values that can be attached to a kind of model element.

Then we introduce the tag definition named performance annotation, which also will be the name of the tagged values. The type of the tag definition can be either system usage or system load or routing rate. The multiplicity of the tag definition will be 1 specifying that a concrete tagged value can have exactly one data value. The last assumption does not restrict the number of performance annotations in a concrete model element since the cardinality of the role +taggedValue for any model element is *, see Figure 3.1. Finally, each concrete performance annotation in a UML diagram will be specified as a tagged value with the attribute dataValue meaning the annotation with the form {a concrete annotation}, i.e. a quantitative annotation inside braces, e.g. {100K}.

For an example see in Figure 3.2 the annotation {p1} attached to the association between the actor1 and the use case UseCase1. The annotation can be attached to an association element because the association metaclass is a specialization of the metaclass ModelElement as required in the metamodel in Figure 3.1. This association element could have multiple tagged values of the type performance annotation, but only one in the example. Then it is interpreted that TaggedValue.name = performance annotation, TaggedValue.dataValue = {p1}, TaggedValue.type.tagType = routing rate, TaggedValue.referencedValue = the referenced association. For each new performance

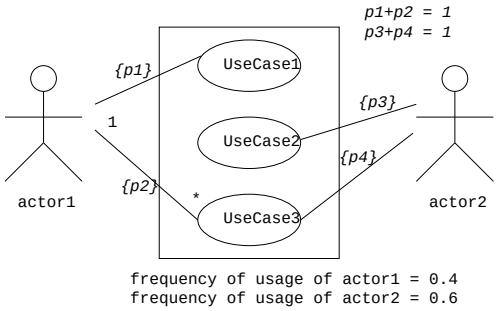


Figure 3.2: Use case diagram with performance annotations.

annotation in this association element one tagged value is created.

The set of tagged values is open-ended in our proposal. Any value that conforms with a performance annotation is valid.

In the following sections the pa-UML proposal for the use case diagrams, the interaction diagrams, the statechart diagrams and the activity diagrams is presented making use of the extension described. This notation will be used in chapter 7 in conjunction with the performance process introduced in that chapter to model performance in a software system. Table 3.1 summarizes for each kind of diagram the annotations proposed in pa-UML as well as the element that affects the annotation.

Before describing which model elements in each diagram should be annotated, we want to stress that it is of special interest for us to introduce the minimal set of annotations possible in each diagram. It is motivated because we recognize that the UML notation is complex and large enough to be increased and also we recognize that the success of a modeling language should be based in its simplicity of use and construction.

3.2 Use case diagrams

In UML a use case diagram shows actors and use cases together with their relationships. The relationships are associations between the actors and the use cases, generalizations between the actors, and generalizations, extends and includes among the use cases [Obj01].

A use case represents a coherent unit of functionality provided by a system, a subsystem or a class as manifested by sequences of messages exchanged among the system (subsystem, class) and one or more actors together with actions performed by the system (subsystem, class). The use cases may optionally be enclosed by a rectangle that represents the boundary of the containing system or classifier [Obj01].

In the example use case diagram in Figure 3.2 appears: two actors, three use cases and four associations relationships between actors and use cases, like that represented by the link between actor1 and use case UseCase1.

Role of the use case diagram concerning performance

We propose the use case diagram with performance evaluation purposes to show the use cases of interest to obtain performance figures. Among the use cases in the diagram a subset of them will be of interest and therefore marked to be considered in the performance evaluation process.

For us the existence of a use case diagram is not mandatory to obtain a performance model. Since as it will be explained in chapter 7 a performance model for the whole system can be obtained from the statecharts that describe the system. The role of the use case diagram is to show the use cases that represent executions of interest in the system. A performance model can be obtained for each concrete execution as it will be proposed in chapter 7.

Each use case of interest should be detailed by means of a sequence diagram, which are studied in the next section.

Performance annotations

The performance annotations for the use case diagram do not constitute a novelty of this work since they have been taken from [CM00]. The proposal consists in the assignment of a probability to every edge that links a type of actor to a use case, i.e. the probability of the actor to execute the use case. The assignment induces the same probability to the execution of the corresponding set of sequence diagrams that describes it. Since we propose to describe the use case by means of only one sequence diagram, we can express formally our case as follows.

Let suppose to have a use case diagram with m users and n use cases. Let $p_i (i = 1, \ldots, m)$ be the i-th user frequency of usage of the software system and let P_{ij} be the probability that the i-th user makes use of the use case $j (j = 1, \ldots, n)$. Assuming that $\sum_{i=1}^{m} p_i = 1$ and $\sum_{j=1}^{n} P_{ij} = 1$, the probability of a sequence diagram corresponding to the use case x to be executed is:

$$P(x) = \sum_{i=1}^{m} p_i \cdot P_{ix}$$

The previous formula is important because it allows to assign a "weight" to each particular execution of the system.

In Figure 3.2 the performance annotations introduced are: the frequencies of usage of the system for each actor, 0,4 and 0,6, the probabilities p1, p2, p3 and p4 attached to each association between an actor and a use case.

The relationships between the actors themselves, and between the use cases themselves are not considered with performance evaluation purposes.

3.3 Interaction diagrams

The description of behavior involves two aspects: 1) the structural description of the participants and 2) the description of the communication patterns. The structure of

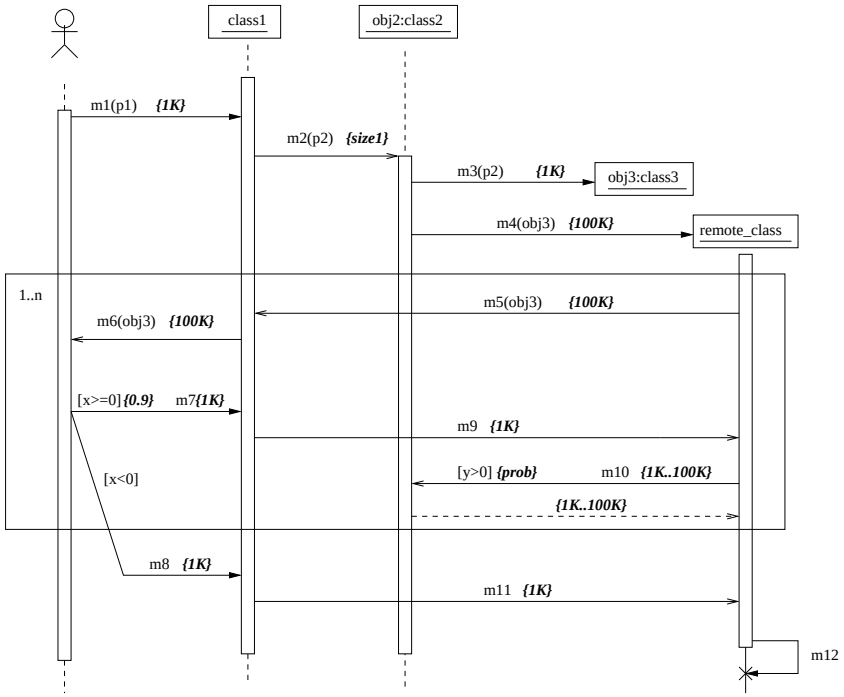


Figure 3.3: Sequence diagram with performance annotations.

instances playing roles in a behavior and their relationships is called a *collaboration*. The communication pattern performed by instances playing the roles to accomplish a specific purpose is called an *interaction*. The two aspects of behavior are often described together on a single diagram, but some times it is useful to describe the structural aspects separately [Obj01].

Interaction diagrams come in two forms based on the same underlying information, specified by a collaboration and possibly by an interaction, but each form emphasizes a particular aspect of it. The two forms are *sequence diagrams* and *collaboration diagrams*. A sequence diagram shows the explicit sequence of communications and it is better for real-time specifications and for complex scenarios. A collaboration diagram shows an interaction organized around the roles in the interaction and their relationships [Obj01].

An interaction is defined in the context of a collaboration. It specifies the communication patterns between the roles in the collaboration. More precisely, it contains a set of partially ordered messages, each specifying one communication. A collaboration may be attached to an operation or a classifier, like a use case, to describe the context in which their behavior occurs; that is, what roles instances play to perform the behavior specified by the operation or the use case. A collaboration is used for describing the realization of an operation or a classifier [Obj01].

Sequence diagrams A sequence diagram presents two dimensions: 1) the vertical dimension represents time and 2) the horizontal dimension represents different instances. An instance is shown as a vertical dashed line called the "lifeline". The lifeline represents the existence of the instance at a particular time. An object symbol is drawn at the head of the lifeline [Obj01].

An activation (focus control) shows the period during which an instance is performing an action. It is shown as a tall thin rectangle whose top is aligned with its initiation time and whose bottom is aligned with its completion time. A stimulus is a communication between two instances that conveys information with the expectation that an action will ensure. A stimulus will cause an operation to be invoked, raise a signal, or cause an instance to be created or destroyed. It is shown as a horizontal solid arrow from the lifeline of one instance to the lifeline of another, labeled with the name of the operation or the signal. Solid arrowhead means procedure call, stick arrowhead means asynchronous communication, while a dashed arrow with stick arrowhead means return from procedure call [Obj01].

Role of the sequence diagram concerning performance

It is worth to notice that, although in this thesis we focus on the sequence diagram to study the relevance of the interactions among objects concerning performance evaluation, the proposals for this diagram are valid also for the case of the collaboration diagram, since as it has been explained they are based on the same underlaying information. By proposals we understand the performance annotations that will be given for the diagram later in this section as well as the role of the diagram in the performance evaluation process that will be given in chapter 7.

As it is explained, a sequence diagram represents messages sent among objects. Also, we have remarked that for our purposes a sequence diagram should detail the functionality expressed in one of the use cases in the use case diagram by focusing in the interactions among its participants. We consider that it is the adequate tool to characterize some aspects of the system load when modeling the execution of interest that describes the corresponding use case. In the following, we remark the relevant elements and constructions of the sequence diagram from the performance point of view to model the load of the system.

Objects can reside in the same machine or in different machines in the case of distributed systems. In the first case it can be assumed that the time spent to send the message is not significant in the scope of the modeled system. Of course the actions taken as a response of the message can spend computation time, but it will be modeled in the statechart diagram. For the second case, those messages that travel through the net, it is considered that they spend time, then supposing a load for the system that should be modeled.

To each message in the diagram a condition can be attached, representing the possibility that the message could be dispatched. Even multiple messages can leave a single point each one labeled by a condition. From the performance point of view it can be considered that routing rates are attached to the messages. See as an example messages m7, m8 and m10 in Figure 3.3.

A set of messages can be dispatched multiple times if they are enclosed and marked as an iteration. This construction also has its implications from the performance point of view. See as an example messages m5, m6, m7, m9 and m10 in Figure 3.3.

In chapter 7 it will be shown how the information contained in the sequence diagram is useful to create a performance model that describes a particular execution of the system.

Performance annotations

UML proposes a notation to deal with time based on the use of time constraints. This restrictions are expressed as time functions on message names, e.g., {(messageOne.receiveTime - messageOne.sendTime) < 1 sec.}. An open-ended set of functions can be defined by the user, as example UML proposes the functions sendTime (the time at which a stimulus is sent by an instance) and receiveTime (the time at which a stimulus is received by an instance).

Without the purpose of slighting the proposal of UML and without renouncing to it for other diagrams, we consider more realistic in this diagram to annotate the message size instead time constraints. In this way, if for instance we consider systems where the messages travel through the net, performance parameters for different net speeds could be calculated or even to make different modeling assumptions for the net.

Each message in the diagram will be annotated with its size, in Figure 3.3 message m1 is labeled with {1 K}. It is also possible to annotate the size with a range in the UML common way, like the message m10 with label {1K..100K}. If the message size is unknown, the annotation is a label representing a performance parameter, e.g. message m2 is annotated with the label {size1}, also in Figure 3.3, then leaving open the possibility to model different sizes. The annotation will be strictly informative if the objects that exchange the message reside on the same machine. But if they reside on different computers, the annotation will be used to model system load, concretely the time spent by the message traveling through the net. A simple assumption can be that the time spent is the size of the message multiplied by the speed of the net, we have adopted it in this work for the sake of simplicity. This time can be assumed as the mean for all possible executions, then it can be modeled by an exponential distribution. But more complex assumptions can be made [DFJR98, BC98].

Each condition associated to a message will have an annotation that represents a routing rate, it expresses the event probability success. See, for instance, the probability {0.9} associated in Figure 3.3 to the condition x >= 0 in message m7. A range is accepted too. Sometimes, it is possible that the probability is unknown when modeling. Also, it could be that the probability a message occurs is a parameter subject to study. In the example, the condition y > 0 associated to the message m10 is a parameter subject of study. In such situations, we will annotate an identifier, corresponding to the unknown probability.

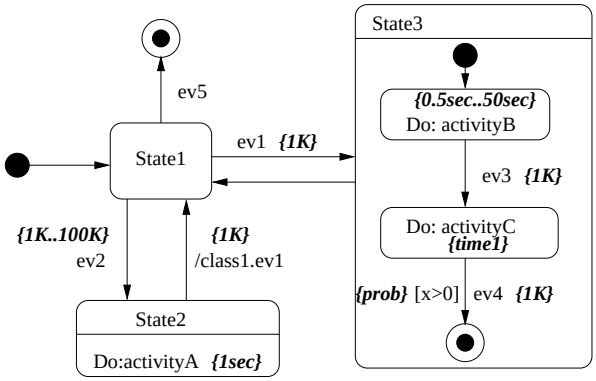


Figure 3.4: Statechart with performance annotations.

3.4 Statechart diagrams

A UML statechart diagram can be used to describe the behavior of a model element such as an object or an interaction. Specifically, it describes possible sequences of states and actions through which the element can proceed during its lifetime as a result of reacting to discrete events. A statechart maps into a state machine that differs from classical Harel state machines in a number of points that can be found in section 2-12 of [Obj01].

A state in a statechart diagram is a condition during the life of an object or an interaction during which it satisfies some condition, perform some action, or waits for some event. A simple transition is a relationship between two states indicating that an object in the first state will enter the second state. An event is a noteworthy occurrence that may trigger a state transition [Obj01].

A composite state is decomposed into two or more concurrent substates (regions) or into mutually exclusive disjoint substates [Obj01].

Role of the statechart diagrams concerning performance

Sequence diagrams show how objects interact for a particular execution of the system. But to take a complete view of the system behavior, it is necessary to understand the life of the objects involved in it, since the statechart diagram is a tool that can be used with these purposes. It is proposed to capture performance requirements at this level of modeling. Then, for each class with relevant dynamic behavior a statechart diagram must be specified.

In a statechart diagram the elements relevant from the performance evaluation point of view are described in the following.

Activities represent tasks performed by an object in a given state. Such activities consume computation time that must be measured and annotated.

Guards show conditions in a transition that must hold in order to fire the corre-

sponding event. Then they can be considered as routing rates as in the case of the sequence diagram.

Events labeling transitions correspond to events in the sequence diagram showing the server or the client side of the object. Then the considerations given in the sequence diagram for the exchange of messages among objects still remain valid for this diagram.

Chapter 7 describes how the information related to the load of the system collected in all the statechart diagrams is used to create a performance model that describes all the possible executions of the system.

Performance annotations

We consider that the UML proposal based on time constraints is more appropriate for this diagram. The annotations for the duration of the activities will show the time needed to perform them. If it is necessary, a minimum and a maximum values could be annotated. If different durations should be tested for a concrete activity then a variable can be used. See, for example, labels {1sec}, {0.5sec..50sec} and {time1} in Figure 3.4.

The annotations for the load of the messages will be attached to the transitions (outgoing or internal) as explained for the case of the sequence diagrams. See, for example, label {1K} associate to transition outgoing State1 with event ev1 or label {1K..100K} associate to transition exiting also from State1 but with event ev2 in Figure 3.4.

The probability of event success represents routing rates as in the case of the sequence diagrams, then it will be annotated in the same way and the same considerations must be taken into account. See, for instance, label {prob} joined to condition [x > 0] in Figure 3.4.

The information provided by the last two kind of annotations, message load and probability of event success, is relevant only to obtain a performance model that represents the whole system. When it is desired to obtain a performance model that represents a concrete execution of the system this information is useless, in this case the message load and the probability of event success is given by the corresponding sequence diagram.

3.5 Activity diagrams

Activity diagrams represent UML activity graphs, which are just a variant of UML state machines (see [Obj01], section 3.84). In fact, a UML activity graph is a specialization of a UML state machine, as it is expressed in the UML metamodel (see Figure 2.2). The main goal of activity diagrams is to stress the internal control flow of a process in contrast to statechart diagrams, which represent UML SMs and are often driven by external events.

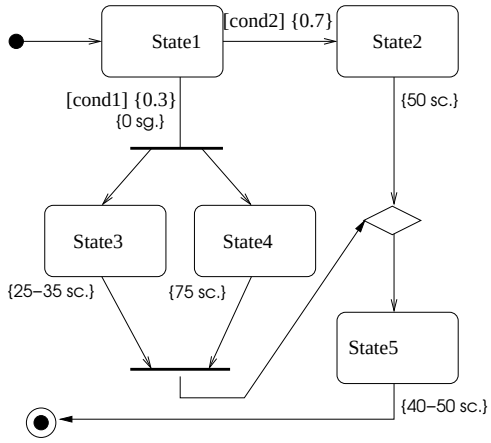


Figure 3.5: Activity diagram with performance annotations.

Role of the activity diagrams concerning performance

In this work the activity diagram is proposed as a tool to specify activities modeled in statechart diagrams. It allows to describe detailed views of the performance parameters of the system when they are known for the very basic actions.

According to UML specification (cfr. [Obj01], section 3.84), almost every state in an activity diagram should be an action or subactivity state, so almost every transition should be triggered by the ending of the execution of the entry action or activity associated to the state. Anyway, UML is not strict at this point, so elements from state machines package could occasionally be used. But it must be noted that in this work when we refer to activity diagrams we only focus in those elements proper of the activity graph package.

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

The performance model obtained from an activity diagram in terms of LGSPNs as proposed in chapter 6 can be used with performance evaluation purposes with two goals: A) just to obtain performance measures of the model element they describe or B) to compose this performance model with the performance models of the statecharts that use the activity modeled in order to obtain a final performance model of the system described by the referred statecharts.

Performance annotations

Annotations will be attached to transitions instead of states as in statecharts, in order to allow the assignment of different action durations depending on the decision.

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

The format suggested is the same as for the previous diagrams making use of the tagged values, {n sec.}, {n-m sec.} and { P(k)} for timed transitions and {P(k)} for not-timed transitions. If no probability P(k) is provided we will assume an equiprobable sample space, i.e., identical probability for each 'brother' transition to be triggered. As it is shown, we allow time expressed in terms of an estimated value or a range of them. We have discarded the usage of packet size annotations as proposed for statecharts due to the fact that activity diagrams are commonly used to model internal control flow. Figure 3.5 shows some examples of annotations (in braces).

Time annotations will be allocated wherever an action is executed (outgoing transitions of such states, including outgoing transitions of decision pseudostates with an action state as input) and probability annotations wherever a decision is taken, i.e. next to guard conditions. It must be noticed that there is a special case where the performance annotation is attached to the state instead of the outgoing transition: when the control flow is not shown because it is implicit in the action-object flow. We do so because we do not want to have performance annotations applied to it, as it has a different semantics.

3.6 Conclusions

The achievement of a performance model for a software system requires the description of the load and the routing rates that characterize it. We have identified the necessity to provide the software engineers artifacts with the notations to accomplish these tasks. The most of the proposals given have in common the use of the UML language, the incorporation to it of annotations that describe performance aspects and the ability to obtain (semi)automatically performance models.

In this chapter we have presented our approach to annotate system load and routing rates in the UML proposal, leading the pa-UML notation. It takes into account the principles required by the software performance community. At the same time we have identified the UML diagrams relevant in the performance evaluation process. Then we have investigated the role of the use case diagrams, interaction diagrams, statechart diagrams and activity diagrams with performance evaluation purposes. These subset of UML diagrams is powerful enough to describe the dynamics and the load of a wide range of software systems, as we can discover through the non trivial examples that we present in the following chapters.

The use case diagrams have been shown as the tool to characterize the actors of

the system by the usage they perform of it. Interaction diagrams allow to describe the load of the system when the participants exchange messages among them. The statechart diagram is a tool where the routing rates and the duration of the activities of the system can be modeled at high level of description. While activity diagrams became useful for a detailed and accurately modeling of an internal process measuring its basics activities.

Chapter 4

UML flat State Machines Compositional Semantics

In this chapter, we give a formal semantics in terms of Labelled Generalized Stochastic Petri Nets (LGSPNs) [DF96] to the simple states, the final states, the initial pseudostates and the outgoing transitions of the state machines package of UML [Obj01]. The previous features conform what we call a "flat UML state machine". This work was developed in [MBCD02]. The features considered for the simple states are: Entry actions, exit actions, activities, internal transitions, deferred events, incoming and outgoing transitions. Therefore all the characteristics for the simple states, final states and initial pseudostates, as given in the state machine package of UML, are considered. Chapter 5 gives formal semantics to the rest of the elements of the state machines package: Composite states, submachine states, pseudostates (except the initial pseudostate), synchronous states and stub states.

In order to give a formal semantics to the UML state machines package in terms of LGSPNs, we propose a translation from the package elements to the LGSPN elements. It must be clearly stated how we are going to refer to the elements in both formalisms, i.e., which are the input and the output models. The input model, UML state machines package, has been summarized in section 2.2.4 and its elements appear in the metamodel in Figure 2.3. The output model will be a LGSPN system as given in [DF96] (see section 2.1.3).

Therefore, the contribution of this chapter is to define the translation of a number of elementary state machine elements into LGSPNs components. Moreover it is given how from these components the LGSPN model for a state machine is obtained. Finally, it is formalized how to obtain a LGSPN model from a set of UML state machines (described by its corresponding LGSPN models). In the translation process several decisions will be taken, since it implies an interpretation of the "non formally defined" UML concepts.

We assume that a system is described by a set of UML state machines, and we show how a LGSPN model can be generated by composing [DF96] the LGSPN models

of the single UML state machines. The LGSPN models are defined compositionally starting from the LGSPN components of each state together with its transitions. Also these components are defined in terms of smaller LGSPNs that represent the basic elements of a state machine: entry and exit actions, do activities, deferred events, internal and outgoing transitions. The translation is performed taking into account that the operational semantics of the LGSPN system must guarantee the "run to completion assumption" of UML, that means that an event can only be dequeued and dispatched if the processing of the previous current event is fully completed.

Given a UML state machine (cfr. section 2.2.4, chapter 2), the approach taken for its translation into a LGSPN model consists of the following steps:

step 1 Each simple state $s \in SimpleState$ is modelled by a LGSPN representing the basic elements of states and transitions. Section 4.7 discusses this step.

step 2 The initial pseudostate (if it exists) and the final states are translated. Sections 4.8 and 4.9 discuss respectively these translations. The LGSPNs produced in this step are composed with those of the previous step to produce a LGSPN model of the entire UML state machine. Section 4.10 discusses this step.

If the system is described through a set of UML state machines, the final step composes them:

step 3 Compose the Petri net subsystems of the state machines and define the initial marking. Section 4.11 discusses this step.

To obtain a LGSPN component that interprets a simple state, see step 1, a set of relations and functions that relate a simple state with its equivalent LGSPN component are given in sections 4.2, 4.3, 4.4, 4.5 and 4.6. Each section deals with a feature of the simple states and begins with the UML description of the feature and when neccesary our interpretation. They have two main parts: An "Informal explanation" of the translation and its formalization, in the paragraph "Formal translation". Moreover, this set of relations and functions can be seen as the formal specification to implement a software tool, that carry out the translation process.

4.1 An example of "flat" UML state machine

A state machine $sm \in StateMachine$ is basically characterized by states and transitions. In the following we describe the informal semantics given by the UML state machines package to the elements in the metamodel of Figure 2.3, considering only those elements that are used for the definition of a "flat" state machine.

The state machine sm in Figure 4.1 is composed of three simple states $M, N, K \in SimpleState$, an initial pseudostate, represented by a black dot, $ps \in Pseudostate$, a final state, represented by a bull eye $f \in FinalState$, and six outgoing transitions. The state machine sm starts its execution by firing transition $tr0 \in ps.outgoing$ which means the arrival of the stereotyped event $\ll create \gg$, $create = trigger(tr0) \in Event$; as a consequence, action $act0 = effect(tr0) \in Action$ is performed. When

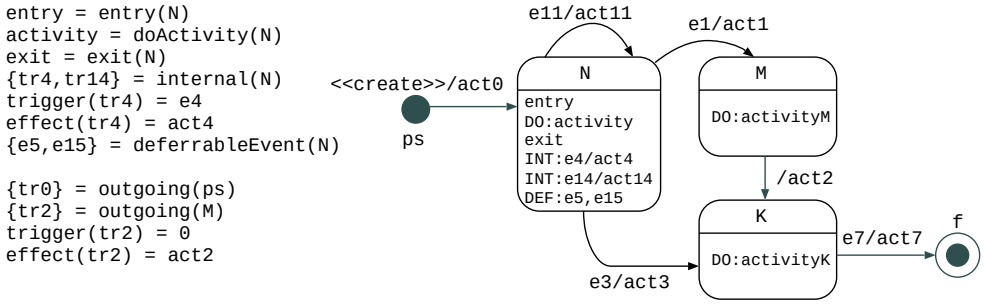


Figure 4.1: An example of "flat" state machine.

in state N, the entry action $entry = entry(N) \in Action$ is executed first; after, the execution of the activity $activity = doActivity(N) \in Action$ begins. During the activity execution the outgoing transitions for state A can ocurr, or the internal transitions as $tr4 \in internal(N)$ or a deferred event such as $e5 \in deferrableEvent(N)$. Outgoing transitions provoke an exit from the state and an entry into the target state (possibly the source state itself – self-loop outgoing transitions, such as $tr11 \in outgoing(N)$). Internal transitions instead do not provoke a change of state and no entry or exit action is executed. Deferred events are not triggered in the present state, they are retained by the state machine. Completion of activity means the generation of the "completion event" for the state. When an outgoing transition as $tr2$ from M to K has no trigger it means that it fires when the "completion event" for its source state $source(tr2) = M$ is generated (they are called immediate outgoing transitions). After visiting states M and K or just K depending on the triggered events, sm completes when it arrives to its final state f.

4.2 Entry actions and activities

The semantics for the entry actions, given in section 2.12 of [Obj01], highlights that: "Whenever a state is entered, it executes its entry action before any other action is executed. If defined, the activity associated with a state is forked as a concurrent activity at the instant when the entry action is completed".

The semantics for the activities in a state, given in section 2.12 of [Obj01] remarks that: "The activity represents the execution of a sequence of actions, that occurs while the state machine is in the corresponding state. The activity starts executing upon entering the state, following the entry action. If the activity completes while the state is still active, it raises a completion event. In case where there is an outgoing completion transition the state will be exited. If the state is exited as a result of the firing of an outgoing transition before the completion of the activity, the activity is aborted prior to its completion".

In this section we define a system that interpretates in terms of LGSPNs the entry

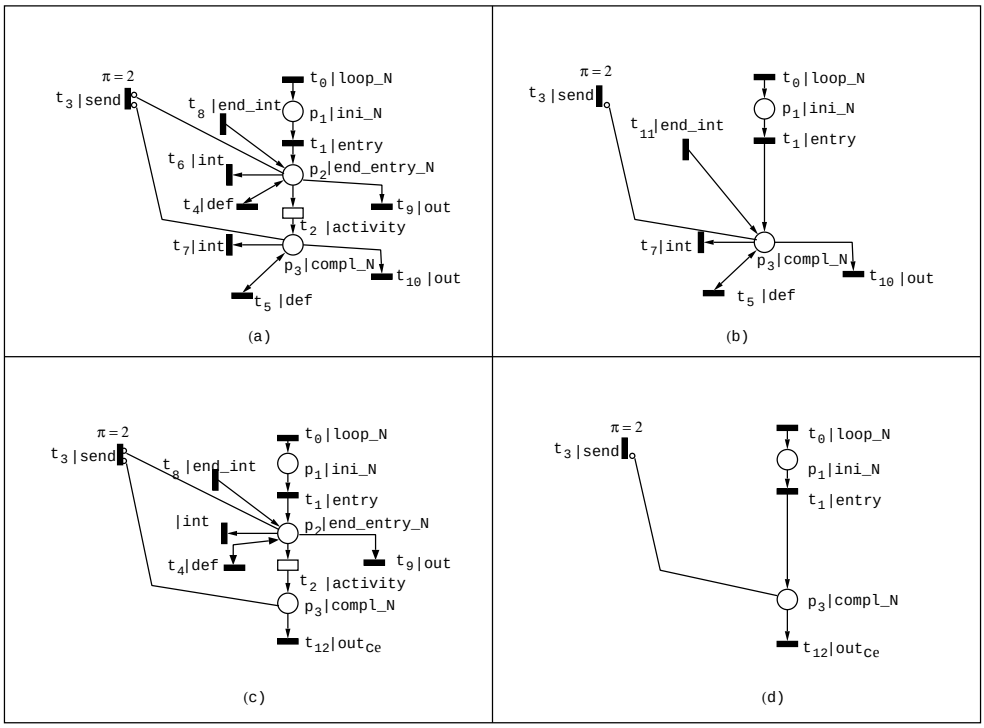


Figure 4.2: Different labelled "basic" systems for a simple state N: (a) with activity and no immediate outgoing transition; (b) no activity and no immediate outgoing transition; (c) with activity and immediate outgoing transition; (d) no activity and immediate outgoing transition.

actions and the activities. The LGSPN system will be composed with other systems that interpretate the rest of the elements in a simple state, for this reason we refer to this system as the "basic" system for a simple state.

4.2.1 Informal explanation

For each simple state in a UML state machine is created a "basic" LGSPN system, named $\mathcal{LS}_i^B$. The formal definition of $\mathcal{LS}_i^B$ will be presented in section 4.2.2. For the sake of simplicity, instead of using the name of the state to name the system, we use a subindex i that represents the number of the state. Therefore $i \in 1..n$ where n is the number of simple states in the state machine. Obviously, the order of enumeration of the states is irrelevant. The $\mathcal{LS}_i^B$ system is the LGSPN interpretation of the entry action and the activity for a given state i. An informal explanation of this interpretation is given in this section in order to easily understand the formal definition given in section 4.2.2.

In order to obtain the $\mathcal{LS}_i^B$ system, the UML metamodel must be taken into account. A different system will be obtained depending on whether an activity exists or not and whether an immediate outgoing transition exists or not in the state. Concretely, four situations can arise:

- Case A. The state has activity and no immediate outgoing transition.
- Case B. The state has not activity and no immediate outgoing transition.
- Case C. The state has activity and immediate outgoing transition.
- Case D. The state has not activity but immediate outgoing transition.

Figure 4.2 shows the four possible configurations for the $\mathcal{LS}_i^B$ system, which are described in detail in the following paragraphs.

Case A: Figure 4.2(a) depicts the $\mathcal{LS}_i^B$ system when there is not an immediate outgoing transition but there is activity in the state i. The following places, transitions and arcs must be created to obtain it:

- Transition t_1 represents outgoing self-loop transitions (see formalization in Definition 4.4), therefore it may exist or not depending whether there exist self-loop transitions. This transition puts a token in place p_1, labelled with the name of the state preceded by *ini* (see Definition 4.1). It means in the UML state machine that the state has been entered and its *entry action*, if it exists, is going to be executed.
- The entry action of a state in the state machine is represented in the $\mathcal{LS}_i^B$ by an immediate transition (see Definition 4.5). The transition is immediate due to the fact that the duration of the entry action is considered immediate in the scope of the modeled system. This transition has an input arc from the place previously described. The firing of the transition represents the execution of entry action, being an action, can belong to any of the subclasses of class *Action*. In particular, if an action belongs to either the subclass *SendAction* or to the subclass *CallAction* it will generate the corresponding events. The interpretation of these particular cases will be given when transitions are translated, now for simplicity we assume that actions do not belong to any of these subclasses. Also, the transitions has just an output arc to a new place, created by the function in Definition 4.2; a token in this place represents the end of the entry action as well as the begining of the activity.
- The activity of a state in the state machine, is represented in the $\mathcal{LS}_i^B$ by a timed transition (see formalization in Definition 4.6). Since it is a timed transition, it must be associated to an exponentially distributed random variable. How to calculate its rate will be explained in 7.2.2 (first approach), when the process is introduced. This transition has just an input arc from the place that represents the end of the entry action and just an output arc to a new place, created by

the function formalized in Definition 4.3, that represents the completion of the activity. This transition may:

- fire, then placing a token in the completion place. In the state machine it means that the "completion event" for this state has been raised, therefore, the activity has been successfully completed.
- be disabled by the firing of any of the transitions created by any of the following functions[1]: $\Lambda^*_{df}(i)$, $\Lambda^*_{ie}(i)$, $\Lambda^*_{e}(i)$. They represent the arrival of a deferred event, an internal event or an event in an outgoing transition respectively. Then the activity is aborted.

- *Interface transitions.* The following transitions are created to compose the $\mathcal{LS}^B_i$ system with the systems for the deferred events, internal transitions and outgoing transitions.
  - Deferred events: If there exist deferred events in the state then three interface transitions in the $\mathcal{LS}^B_i$ system will be created to compose it with the system for the deferred events (see Definition 4.20). They are the following: A transition labeled *send* (created in Definition 4.9) and two transitions labeled *def* (created in Definition 4.8). In Figure 4.2, see transitions $t3|send$, $t4|def$ and $t5|def$.
  - Internal transtions: If there exist internal transitions in the state then three interface transitions in the $\mathcal{LS}^B_i$ system will be created to compose it with the system for the internal transitions (see Definition 4.30). They are the following: A transition labeled *end_int* (created in Definition 4.10) and two transitions labeled *int* (created in Definition 4.12). In Figure 4.2, see transitions $t8|end\_int$, $t6|int$ and $t7|int$.
  - Outgoing transtions: The interface transitions in the $\mathcal{LS}^B_i$ system to compose it with the system for the outgoing transitions (see Definition 4.44) are the following:
    - Two interface transitions labelled *out* (created in Definition 4.13). In Figure 4.2 see transitions $t9|out$ and $t10|out$.

Case B: Figure 4.2(b) depicts the $\mathcal{LS}^B_i$ system when there is not an immediate outgoing transitions neither activity for the state i. The created places are those that represent the state and the completion event. The created transitions are those that represent the outgoing self-loop transition and the entry action. The created interface transitions are those connected to the completion event place as explained in case A, also a new *end_int* interface transition is created with an output arc to the completion event place.

Case C: Figure 4.2(c) depicts the $\mathcal{LS}^B_i$ system when there is an immediate outgoing transition and activity for the state i. The places, transitions and arcs created are the

[1] Defined in 4.2.2.

same as in the case A except the input and output arcs connected to the completion event place which are removed. Moreover a new *interface transition*, out_{ce}, is created with an input arc from the completion event place.

Case D: Figure 4.2(d) depicts the $\mathcal{LS}_i^B$ system for this case when there is an immediate outgoing transition but not activity for the state i. The places, transitions and arcs created here are the same as in case B except the input and output arcs connected to the completion event place which are removed. Moreover a new *interface transition*, out_{ce}, is created with an input arc from the completion event place.

Finally, it must be remarked that the $\mathcal{LS}_i^B$ system will be composed with the systems created in sections 4.3, 4.4 and 4.5 for the deferred events, the internal transitions and the outgoing transitions, in order to obtain a system that interpretates the whole simple state (see Section 4.7).

4.2.2 Formal translation: The $\mathcal{LS}_i^B$ system

In this section we give the formal definitions to obtain a LGSPN system for the entry actions and the activities of a given state i, the $\mathcal{LS}_i^B$ system. The system is obtained mapping abstractions in the UML metamodel, see Figure 2.3 onto the elements of the $\mathcal{LS}_i^B$ system. Therefore it is necessary to clearly stablish how we are going to refer to these abstractions in mathematical terms.

The abstractions in the UML metamodel are named as follows:

- We refer to a class by its name, e.g. the class Action is *Action*. The following exceptions arise: For the classes State, SimpleState, CompositeState, FinalState, shallowHistory, Pseudostate, deepHistory Pseudostate and initial Pseudostate, being their names too long, we use the following symbols to name respectively each one of them, $\Sigma, \Sigma_{ss}, \Sigma_{cs}, \Sigma_{fs}, \Sigma_{sh}, \Sigma_{dh}, \Sigma_{ini}$.
- We adopt a functional notation to indicate the image of an element (or of a set of elements) belonging to the domain of a certain relation, in the following we give an example. Being *doActivity* an association among the classes Σ_{ss} and *Action*, then the image of an instance s of the class Σ_{ss}, through the relation *doActivity* is denoted as *doActivity*(s).

The elements of the resulting LGSPN (the $\mathcal{LS}_i^B$ system) are named as usally (cfr. Definition 2.5), i.e. $\mathcal{LS}_i^B = (S_i^B, \psi_i^B, \lambda_i^B)$, with $S_i^B = \langle P_i^B, T_i^B, I_i^B, O_i^B, H_i^B, \Pi_i^B, W_i^B, {M_0}_i^B \rangle$ the GSPN system and ψ_i^B, λ_i^B the labeling functions.

Let us assume that:

Case A. $P_i^B = \{p_s, p_{en}, p_{ce}\}, T_i^B = \{t_l, t_{en}, t_{do}, t_{df}, t'_{df}, t_{sdf}, t'_{end}, t_{int}, t'_{int}, t_{out}, t'_{out}\}$.

Case B. $P_i^B = \{p_s, p_{ce}\}, T_i^B = \{t_l, t_{en}, t_{df}, t_{sdf}, t_{end}, t_{int}, t_{out}\}$.

Case C. $P_i^B = \{p_s, p_{en}, p_{ce}\}, T_i^B = \{t_l, t_{en}, t_{do}, t_{ce}, t'_{df}, t_{sdf}, t'_{end}, t'_{int}, t'_{out}\}$.

Case D. $P_i^B = \{p_s\}, T_i^B = \{t_l, t_{en}, t_{ce}, t_{sdf}\}$.

Before to define the $\mathcal{LS}_i^B$ system (see Definition 4.14) let us introduce the relations between the elements in a given flat UML state machine (**SM**) and the places and transitions for the LGSPN system.

Definition 4.1. *Let us define a function,* $\Psi_s : \Sigma_{ss} \longrightarrow \bigcup_{j=1}^{n} P_j^B$*, from the set of simple states of* **SM** *to the set of places in* $\bigcup_{j=1}^{n} S_j^B$*, such that,*

$$\forall s \in \Sigma_{ss} : \Psi_s(s) = p_s.$$

Definition 4.2. *Let us define a partial function,* $\Psi_{en} : \Sigma_{ss} \hookrightarrow \bigcup_{j=1}^{n} P_j^B$ *, from the set of simple states of* **SM** *to the set of places in* $\bigcup_{j=1}^{n} S_j^B$*, such that,*

$$\forall s \in \Sigma_{ss}, dA(s)^2 \neq \emptyset : \Psi_{en}(s) = p_{en}.$$

Definition 4.3. *Let us define a function,* $\Psi_{ce} : \Sigma_{ss} \longrightarrow \bigcup_{j=1}^{n} P_j^B$ *, from the set of simple states of* **SM** *to the set of places in* $\bigcup_{j=1}^{n} S_j^B$*, such that,*

$$\forall s \in \Sigma_{ss} : \Psi_{ce}(s) = p_{ce}.$$

Definition 4.4. *Let us define a partial function,* $\Lambda_l : \Sigma_{ss} \hookrightarrow \bigcup_{j=1}^{n} T_j^B$*, from the set of simple states of* **SM** *to the set of transitions in* $\bigcup_{j=1}^{n} S_j^B$*, such that,*

$$\forall s \in \Sigma_{ss} : \exists t \in out(s)^3, target(t) = source(t) = s : \Lambda_l(s) = t_l.$$

Definition 4.5. *Let us define a function,* $\Lambda_{en} : \Sigma_{ss} \longrightarrow \bigcup_{j=1}^{n} T_j^B$*, from the set of simple states of* **SM** *and the set of transitions in* $\bigcup_{j=1}^{n} S_j^B$*, such that,*

$$\forall s \in \Sigma_{ss} : \Lambda_{en}(s) = t_{en}.$$

Definition 4.6. *Let us define a partial function,* $\Lambda_{do} : \Sigma_{ss} \hookrightarrow \bigcup_{j=1}^{n} T_j^B$*, from the set of simple states of* **SM** *and the set of transitions in* $\bigcup_{j=1}^{n} S_j^B$*, such that,*

$$\forall s \in \Sigma_{ss}, dA(s) \neq \emptyset : \Lambda_{do}(s) = t_{do}.$$

Let us introduce the following notation before the Definitions to map the interface transitions.

Definition 4.7. *Let us denote by* Σ_{ss}° *and* $\Sigma_{ss}^{\bullet}$ *the following sets,*

$\Sigma_{ss}^{\circ} = \{s \mid s \in \Sigma_{ss} \ \wedge \ \exists t \in \mathrm{T}_{SM} : source(t) = s, \ trigger(t) = \emptyset\}$
$\Sigma_{ss}^{\bullet} = \{s \mid s \in \Sigma_{ss} \ \wedge \ (\forall t \in \mathrm{T}_{SM} : source(t) = s) \rightarrow trigger(t) \neq \emptyset\}$
Therefore, $\Sigma_{ss}^{\circ} \cup \Sigma_{ss}^{\bullet} = \Sigma_{ss} \ \wedge \ \Sigma_{ss}^{\circ} \cap \Sigma_{ss}^{\bullet} = \emptyset$

[2]Let us use in the following, $dA(s)$ as a shorthand notation of $doActivity(s)$.
[3]Let us use in the following, $out(s)$ as a shorthand notation of $outgoing(s)$.

Definition 4.8. *Let us define a partial function*[4], $\Lambda^*_{df} : \Sigma_{ss} \hookrightarrow (\bigcup_{j=1}^{n} T_j^B \times \bigcup_{j=1}^{n} T_j^B) \cup \bigcup_{j=1}^{n} T_j^B$, *from the set of simple states of* **SM** *to the set of transitions in* $\bigcup_{j=1}^{n} S_j^B$, *such that,*

$$\forall s \in \Sigma_{ss}, dE(s)^5 \neq \emptyset : \Lambda^*_{df}(s) = \begin{cases} (t_{df}, t'_{df}) & \text{if } s \in \Sigma^{\bullet}_{ss} \ \wedge \ dA(s) \neq \emptyset \\ t'_{df} & \text{if } s \in \Sigma^{\bullet}_{ss} \ \wedge \ dA(s) = \emptyset \\ t_{df} & \text{if } s \in \Sigma^{\circ}_{ss} \ \wedge \ dA(s) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 4.9. *Let us define a partial function,* $\Lambda^*_{sdf} : \Sigma_{ss} \hookrightarrow \bigcup_{j=1}^{n} T_j^B$, *from the set of simple states of* **SM** *to the set of transitions in* $\bigcup_{j=1}^{n} S_j^B$, *such that,*

$$\forall s \in \Sigma_{ss}, dE(s) \neq \emptyset : \Lambda^*_{sdf}(s) = t_{sdf}.$$

Definition 4.10. *Let us define a partial function,* $\Lambda^*_{eie} : \Sigma_{ss} \hookrightarrow \bigcup_{j=1}^{n} T_j^B$, *from the set of simple states of* **SM** *to the set of transitions in* $\bigcup_{j=1}^{n} S_j^B$, *such that,*

$$\forall s \in \Sigma_{ss}, int(s)^6 \neq \emptyset : \Lambda^*_{eie}(s) = \begin{cases} t_{end} & \text{if } s \in \Sigma_{ss} \ \wedge \ dA(s) \neq \emptyset \\ t'_{end} & \text{if } s \in \Sigma^{\bullet}_{ss} \ \wedge \ dA(s) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 4.11. *Let us define a function,* $\Lambda^*_{ce} : \Sigma^{\circ}_{ss} \longrightarrow \bigcup_{j=1}^{n} T_j^B$, *between the simple states with immediate outgoing transitions of the* **SM** *to the set of transitions in* $\bigcup_{j=1}^{n} S_j^B$, *such that,*

$$\forall s \in \Sigma^{\circ}_{ss} : \Lambda^*_{ce}(s) = t_{ce}.$$

Definition 4.12. *Let us define a partial function,* $\Lambda^*_{ie} : \Sigma_{ss} \hookrightarrow (\bigcup_{j=1}^{n} T_j^B \times T_j^B) \cup T_j^B$, *from the set of simple states of* **SM** *to the set of transitions in* $\bigcup_{j=1}^{n} S_j^B$, *such that,*

$$\forall s \in \Sigma_{ss}, int(s) \neq \emptyset : \Lambda^*_{ie}(s) = \begin{cases} (t_{int}, t'_{int}) & \text{if } s \in \Sigma^{\bullet}_{ss} \ \wedge \ dA(s) \neq \emptyset \\ t'_{int} & \text{if } s \in \Sigma^{\bullet}_{ss} \ \wedge \ dA(s) = \emptyset \\ t_{int} & \text{if } s \in \Sigma^{\circ}_{ss} \ \wedge \ dA(s) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 4.13. *Let us define a partial function,* $\Lambda^*_{e} : \Sigma_{ss} \hookrightarrow (T_i^B \times T_i^B) \cup T_i^B$, *from the set of simple states of* **SM** *to the set of transitions in* $\bigcup_{j=1}^{n} S_i^B$, *such that,*

[4]Functions named usign an * create interface transitions.
[5]Let us use in the following, $dE(s)$ as a shorthand notation of $deferrableEvent(s)$.
[6]Let us use in the following, $int(s)$ as a shorthand notation of $internal(s)$.

$$\forall s \in \Sigma_{ss}, out(s) \neq \emptyset : \Lambda_e^*(s) = \begin{cases} (t_{out}, t'_{out}) & \text{if } s \in \Sigma_{ss}^{\bullet} \ \wedge \ dA(s) \neq \emptyset \\ t'_{out} & \text{if } s \in \Sigma_{ss}^{\bullet} \ \wedge \ dA(s) = \emptyset \\ t_{out} & \text{if } s \in \Sigma_{ss}^{\circ} \ \wedge \ dA(s) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 4.14. *The system* $\mathcal{LS}_i^B = (S_i^B, \psi_i^B, \lambda_i^B)$ *associated to the entry actions and the activities of a given state* i *of a flat UML state machine* **SM** *with* $S_{\mathrm{i}}^{\mathrm{B}} = \langle P_{\mathrm{i}}^{\mathrm{B}}, T_{\mathrm{i}}^{\mathrm{B}}, I_{\mathrm{i}}^{\mathrm{B}}, O_{\mathrm{i}}^{\mathrm{B}}, H_{\mathrm{i}}^{\mathrm{B}}, \Pi_{\mathrm{i}}^{\mathrm{B}}, W_{\mathrm{i}}^{\mathrm{B}}, M_{0\mathrm{i}}{}^{\mathrm{B}} \rangle$ *is defined as follows:*

$P_i^B = \Psi_s(i) \cup \Psi_{en}(i) \cup \Psi_{ce}(i)$,
$T_i^B = \Lambda_l(i) \cup \Lambda_{en}(i) \cup \Lambda_{do}(i) \cup \Lambda_{df}^*(i) \cup \Lambda_{sdf}^*(i) \cup \Lambda_{eie}^*(i) \cup \Lambda_{ie}^*(i) \cup \Lambda_e^*(i)$,

$$I_i^B(t) = \begin{cases} p_s & \text{if } t = t_{en} \\ p_{en} & \text{if } t \in \{t_{do}, t_{df}, t_{int}, t_{out}\} \\ p_{ce} & \text{if } t \in \{t'_{df}, t'_{int}, t'_{out}, t_{ce}\} \\ \emptyset & \text{if } t \in \{t_l, t_{sdf}, t_{end}, t'_{end}\} \end{cases}$$

$$O_i^B(t) = \begin{cases} p_s & \text{if } t = t_l \\ p_{en} & \text{if } t = \{t_{df}, t_{end}\} \\ p_{en} & \text{if } t = t_{en} \ \wedge \ dA(i) \neq \emptyset \\ p_{ce} & \text{if } t = t_{en} \ \wedge \ dA(i) = \emptyset \\ p_{ce} & \text{if } t \in \{t_{do}, t'_{df}, t'_{end}\} \\ \emptyset & \text{if } t \in \{t_{sdf}, t_{int}, t'_{int}, t_{out}, t'_{out}, t_{ce}\} \end{cases}$$

$$H_i^B(t) = \begin{cases} p_{en} \cup p_{ce} & \text{if } t \in \Lambda_{sdf}^*(i) \ \wedge \ dA(i) \neq \emptyset \\ p_{ce} & \text{if } t \in \Lambda_{sdf}^*(i) \ \wedge \ dA(i) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$\Pi_i^B(t) = \begin{cases} 0 & \text{if } t = t_{do} \\ 2 & \text{if } t = t_{sdf} \\ 1 & \text{otherwise} \end{cases} \qquad W_i^B(t) = \begin{cases} r_{do} & \text{if } t = t_{do} \\ 1 & \text{otherwise} \end{cases}$$

$$\psi_i^B(p) = \begin{cases} ini\_st & \text{if } p = p_s \\ end\_entry\_st & \text{if } p = p_{en} \\ compl\_st & \text{if } p = p_{ce} \\ & \text{where } st = name(i) \end{cases}$$

$$\lambda_i^B(t) = \begin{cases} loop\_st & \text{if } t = t_l \\ \lambda & \text{if } t = t_{en} \ \wedge \ entry(i) = \emptyset \\ ent & \text{if } t = t_{en} \ \wedge \ entry(i) \neq \emptyset \ \wedge \ ent = name(a) \ \wedge \ a = entry(i) \\ act & \text{if } t = t_{do} \ \wedge \ act = name(a') \ \wedge \ a' = dA(i) \\ send & \text{if } t = t_{sdf} \\ def & \text{if } t \in \{t_{df}, t'_{df}\} \\ end\_int & \text{if } t \in \{t_{end}, t'_{end}\} \\ int & \text{if } t \in \{t_{int}, t = t'_{int}\} \\ ini\_st\_st & \text{if } t = t_{ini} \\ out & \text{if } t \in \{t_{out}, t'_{out}\} \\ out_{ce} & \text{if } t = t_{ce} \\ & \text{where } st = name(i) \end{cases}$$

4.3 Deferred events

"A state may specify a set of event types that may be *deferred* in that state. An event instance that does not trigger any transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead, it remains in the event queue while another non-deferred message is dispatched instead. This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a transition.". The previous paragraph is a citation from section 2.12 of [Obj01]. As an example, in Figure 4.1 the event $ev6$ in the state A is a deferred event.

It is interesting to remember that "strictly speaking", the term *event* is used to refer to the type and not to an instance of the type. However, if the meaning is clear from the context, the term is also used to refer to an event instance.

In this section we will define a labelled system, named $\mathcal{LS}_i^d$, that interpretates in terms of Petri nets the deferred events in a simple state i.

4.3.1 Informal explanation

For each state with deferred events in a flat UML state machine **SM**, a $\mathcal{LS}_i^d$ system is created (see Definition 4.19 for its formalization and Figure 4.3(b) for an example). This system will be composed with the systems $\mathcal{LS}_i^B$ (basic LGSPN system), $\mathcal{LS}_i^t$ (LGSPN system for the internal transitions) and $\mathcal{LS}_i^g$ (LGSPN system for the outgoing transitions) in order to give a Petri net interpretation of the simple state i, see Section 4.7.

For practical pourposes, in order to explain how the Petri net interpretation of the deferred events works without taking into account any other feature of the state machine, we have created the $\mathcal{LS}_i^D$ system. This system is the superposition of the

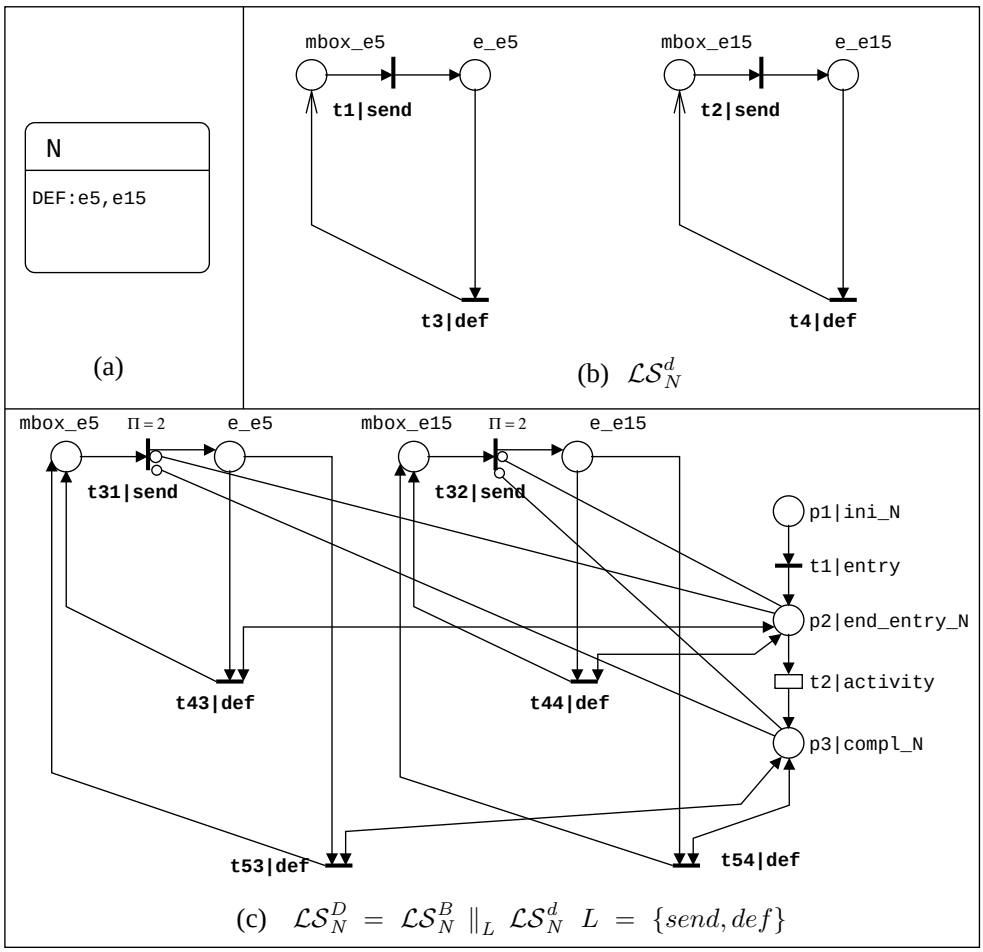


Figure 4.3: Translations of the deferred events of the state N.

$\mathcal{LS}_i^B$ system and the $\mathcal{LS}_i^d$ system using the labels $\{send, def\}$. Also for practical pourposes, in order to clarify the $\mathcal{LS}_i^D$ system, the transitions $\{loop\_i, int, end\_int, out\}$ of the $\mathcal{LS}_i^B$ are not considered (hided) in $\mathcal{LS}_i^D$. This system is formally defined in Definition 4.20.

In the following, we describe informally the previous systems using Figure 4.3 as an example. The formal definitions are given in the next section.

Step A. The $\mathcal{LS}_i^d$ system

The $\mathcal{LS}_i^d$ system represents the translation of all deferred events in the state i. The example in Figure 4.3(b) represents the translation of the deferred events $e5$ and $e15$

for the state N (extracted from Figure 4.1).

The elements of the $\mathcal{LS}_i^d$ system are the following:

- There are as many *event* places, labelled *e_event* as deferred events exist in the state. A token in one of these places represents an instance of an event of the type *event*.
- There are as many immediate *interface* transitions labelled *def* as deferred events exist in the state. Each of them has an input arc from an *event* place. Also, each of them has an output arc to the *mbox_event* place (these transitions are formally defined in Definition 4.17). In the example, two *def* transitions have been created, one has $e5$ as input place and the other $e15$, and *mbox_e*5 and *mbox_e*15 as output places, respectively.
- There are as many *mbox_event* places as deferred events exist in the state. They represent in the $\mathcal{LS}_i^d$ the places to store the deferred events previously to insert them in the event queue of the **SM**. Each of them has an input arc from a *def* transition and an output arc to the *send* transition (these places are formally defined in Definition 4.15). In the example these places are *mbox_e*5 and *mbox_e*15.
- There are as many immediate *interface* transitions labeled *send* as deferred events exist in the state. They are used in the $\mathcal{LS}_i^d$ to transfer the instances of the deferred events to the event queue. Therefore, each one has an input arc from the *mbox_event* place and an output arc to the place *e_event*, which represents the event queue of a deferred *event* in **SM** (this transition is formally defined in Definition 4.18). In the example, two *send* transitions have been created, one has *mbox_e*5 and *mbox_e*15 as input places, and $e5$ and $e15$ as output places.

Step B. The $\mathcal{LS}_i^D$ system

Systems $\mathcal{LS}_i^B$ and the $\mathcal{LS}_i^d$ are composed using superposition of transitions leading a new system named $\mathcal{LS}_i^D$. We use this system to easily explain the Petri net interpretation of the deferred events. The superposition is performed using the interface transitions labelled *def* and *send*. The composition of the $\mathcal{LS}_N^d$ in Figure 4.3(b) and the $\mathcal{LS}_N^B$ in Figure 4.2(a) is shown in Figure 4.3(c). Remember that the transitions $\{loop\_i, int, end\_int, out\}$ are not taken into account in order to simplify the explanation. In the following, the new system is commented stressing the composition over the event $e5$.

Transitions $t43|def$ and $t53|def$ are the superposition of transitions $t4|def$ (in $\mathcal{LS}_N^B$) and $t3|def$ (in $\mathcal{LS}_N^d$) and $t5|def$ (in $\mathcal{LS}_N^B$) and $t3|def$ (in $\mathcal{LS}_N^d$), respectively. Obviously, the firing of any of these transitions represents the arrival of the deferred event in **SM**.

One of these transitions, $t43|def$ in this example, has two input arcs: One of them from the place that represents the end of the entry action (*end_entry_N*) and the other from the place that represents the triggered event. Also, it has two output arcs,

one to the *end_entry_N* place and the other to the place *mbox_e*5. The meaning of this transition in **SM** is that a deferred event can be accepted while the activity is in execution, the consequences are:

1. A token is removed from place *end_entry_st*, which means that the activity is aborted.

2. A token is added to place *mbox_event*, which means that an event named *event* is stored in a buffer until the current state will be exited.

3. Finally, a token is inserted in the place *end_entry_st*, which means that the activity is newly executed.

The other transition that represents the event, $t53|def$ in this example, has also two input arcs: One of them from the *compl_N* place and the other from the place that represents the triggered event. Also, it has two output arcs: one to the *compl_N* place and the other to place *mbox_e*5. The meaning of this transition is that a deferred event can be accepted after the activity execution has finished. The consequence is that a token is added to place *mbox_event*, which means that an event named *event* is stored in a buffer until the current state will be exited and a token is placed newly in the *compl_state* place, therefore, events can be newly accepted.

Transition $t31|send$ is the superposition of transitions $t3|send$ (in $\mathcal{LS}_N^B$) and $t1|send$ (in $\mathcal{LS}_N^d$). This transition transfers tokens from place *mbox_e*5 to place *e*5. If there exist a token in place *end_entry_N*, or in place *completion_event_N*, or in any of the places in the set[7] $\{\Psi_{acc}(N)\}$ the transition cannot fire. It means that the tokens are transferred only when the current state is completed. The priority of this transitions must be greater than that of any transition created by $\Lambda_{ex}(exit(N))$.

4.3.2 Formal translation

Step A. The $\mathcal{LS}_i^d$ system

Before to define the system $\mathcal{LS}_i^d = (S_i^d, \psi_i^d, \lambda_i^d)$, with $S_i^d = \langle P_i^d, T_i^d, I_i^d, O_i^d, H_i^d, \Pi_i^d, W_i^d, {M_0}_i^d \rangle$ let us introduce the relations between the elements in the flat state machine **SM** and the places and transitions for the system.

Let us assume that $P_i^d = \{p_1, \ldots, p_k\}, T_i^d = \{t_1, \ldots, t_k\}, k = 2 * |dE(i)|$.

Definition 4.15. *Let us define a relation,* $\Psi_m \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^d$*, between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^d$*, such that,*

$$\Psi_m{}^{8} = \{(s, p_l) \mid s \in \Sigma_{ss}, dE(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{d}}, l = 1 \ldots |dE(s)|\}.$$

[7] Each place in this set means that an internal transition in the state N has been accepted.

[8] We will use the functional notation $\Psi_m(s) = \{p_1, \ldots, p_k\}$ to denote the set of places related with the state s throught the relation Ψ_m.

Definition 4.16. *Let us define a relation,* $\Psi_d \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^d$ *, between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^d$*, such that,*

$$\Psi_d = \{(s, p_l) \mid s \in \Sigma_{ss}, dE(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{d}}, l = |dE(s)| + 1 \ldots 2 * |dE(s)|\}.$$

Definition 4.17. *Let us define a relation,* $\Lambda_{df} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} T_j^d$*, between the set of simple states of* **SM** *and the set of transitions in* $\bigcup_{j=1}^{n} S_j^d$*, such that,*

$$\Lambda_{df} = \{(s, t_l) \mid s \in \Sigma_{ss}, dE(s) \neq \emptyset, t_l \in T_{\mathrm{s}}^{\mathrm{d}}, l = 1 \ldots |dE(s)|\}.$$

Definition 4.18. *Let us define a relation,* $\Lambda_{sdf} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} T_j^d$*, between the set of simple states of* **SM** *to the set of transitions in* $\bigcup_{j=1}^{n} S_j^d$*, such that,*

$$\Lambda_{sdf} = \{(s, t_l) \mid s \in \Sigma_{ss}, dE(s) \neq \emptyset, t_l \in T_{\mathrm{s}}^{\mathrm{d}}, l = |dE(s)| + 1 \ldots 2 * |dE(s)|\}.$$

Definition 4.19. *The system* $\mathcal{LS}_i^d = (S_i^d, \psi_i^d, \lambda_i^d)$ *associated to the deferred events of a given state* i *of a flat UML state machine* **SM** *with* $S_{\mathrm{i}}^{\mathrm{d}} = \langle P_{\mathrm{i}}^{\mathrm{d}}, T_{\mathrm{i}}^{\mathrm{d}}, I_{\mathrm{i}}^{\mathrm{d}}, O_{\mathrm{i}}^{\mathrm{d}}, H_{\mathrm{i}}^{\mathrm{d}}, \Pi_{\mathrm{i}}^{\mathrm{d}}, W_{\mathrm{i}}^{\mathrm{d}}, M_{0\mathrm{i}}^{\mathrm{d}} \rangle$ *is defined as follows:*

$P_i^d = \Psi_m(i) \cup \Psi_d(i)$,
$T_i^d = \Lambda_{sdf}(i) \cup \Lambda_{df}(i)$,

$$I_i^d(t) = \begin{cases} p_m \in \Psi_m(i) & \text{if } t \in \Lambda_{sdf}(i) \\ p_d \in \Psi_d(i) & \text{if } t \in \Lambda_{df}(i) \end{cases} \quad O_i^d(t) = \begin{cases} p_d \in \Psi_d(i) & \text{if } t \in \Lambda_{sdf}(i) \\ p_m \in \Psi_m(i) & \text{if } t \in \Lambda_{df}(i) \end{cases}$$

$$H_i^d(t) = \emptyset, W_i^d(t) = 1, \Pi_i^d(t) = 1 : \forall t \in T_i^d$$

The following restriction must be fulfilled, $\forall p \in P_i^d : |{}^\bullet p| = |p^\bullet| = 1$.

$$\psi_i^d(p) = \begin{cases} mbox\_name(ev) & \text{if } p \in \Psi_m(i) \\ e\_name(ev) & \text{if } p \in \Psi_d(i) \\ & \text{where } ev \in dE(i) \end{cases}$$

The following restriction must be fulfilled, $\forall ev \in dE(i) : \exists \{l_1, l_2\} \in \psi_i^d(p) : l_1 = mbox\_name(ev) \wedge l_2 = e\_name(ev)$.

$$\lambda_i^d(t) = \begin{cases} send & \text{if } t \in \Lambda_{sdf}(i) \\ def & \text{if } t \in \Lambda_{df}(i) \end{cases}$$

Step B. The $\mathcal{LS}_i^D$ system

Definition 4.20. *The system* $\mathcal{LS}_i^D$ *is the superposition of* $\mathcal{LS}_i^B$ *and* $\mathcal{LS}_i^d$ *over the set* L *of labels,*

$$\mathcal{LS}_i^D = \mathcal{LS}_i^B \underset{L}{\parallel} \mathcal{LS}_i^d \qquad L = \{send, def\}$$

but for function $F_i^D \in \{W_i^D(), \Pi_i^D()\}$ we define,

$$F_i^D(t) = \begin{cases} F_i^B(t) & \text{if } t \in T_i^B \backslash T_{i,L \cup L'}^B \\ max\{F_i^B(t_1), F_i^d(t_2)\} & \text{if } t = (t_1, t_2) \ \wedge \ \lambda_i^B(t_1) = \lambda_i^d(t_2) \in L \end{cases}$$

where $L' = \{int, end\_int, out, out_{ce}, ini\_i\_i\}$

4.4 Internal transitions

In a state i, it may exist a set of internal transitions, $internal(i)$. "They are a set of transitions that, if triggered, occur without exiting or re-entering this state. Thus, they do not cause a state change. This means that the entry or exit condition of the state will not be invoked". The previous paragraph is taken from section 2.12 of [Obj01].

In an internal transition t may appear: A guard, $guard(t)$, they are not considered in this work; an event which triggers the transition, $trigger(t)$; and an action as an effect, $effect(t)$. An internal transition can be triggered both, while the activity is in execution, therefore it is aborted, or when the activity has finished. In both cases, the activity is restarted when the action of the internal transition ends.

In this section we will define a labelled system, named $\mathcal{LS}_i^t$, that interpretates in terms of Petri nets the internal transitions in a simple state i.

4.4.1 Informal explanation

For each state in a flat UML state machine **SM** with internal transitions a $\mathcal{LS}_i^t$ system is created (Definition 4.29 constitutes its formalization), see Figure 4.4(b) for an example. This system will be composed in section 4.7 with the systems $\mathcal{LS}_i^B$(basic LGSPN system), $\mathcal{LS}_i^d$ (LGSPN system for the deferred events) and $\mathcal{LS}_i^g$ (LGSPN system for the outgoing transitions) in order to give a Petri net interpretation of the simple state i.

For practical pourposes, in order to explain how the Petri net interpretation of the internal transitions works without taking into account any other feautre of the state machine, we have created the $\mathcal{LS}_i^I$ system. This system is the superposotion of the $\mathcal{LS}_i^B$ system and the $\mathcal{LS}_i^t$ system using the labels $\{int, end\_int\}$. Also for practical pourposes, in order to clarify the $\mathcal{LS}_i^I$ system, the transitions $\{loop\_i, send, def, out\}$ of the $\mathcal{LS}_i^B$ are not considered (they are hided) in $\mathcal{LS}_i^I$.

In the following, we describe informally the previous systems using Figure 4.4 as an example. The formal definitions are given in the next section.

Step A. The $\mathcal{LS}_i^t$ system

The $\mathcal{LS}_i^t$ system represents the translation of all the internal transitions in the state i. The example in Figure 4.4(b) represents the translation of the internal transitions $e4/act4$ and $e14/act14$ for the state N (extracted from Figure 4.1).

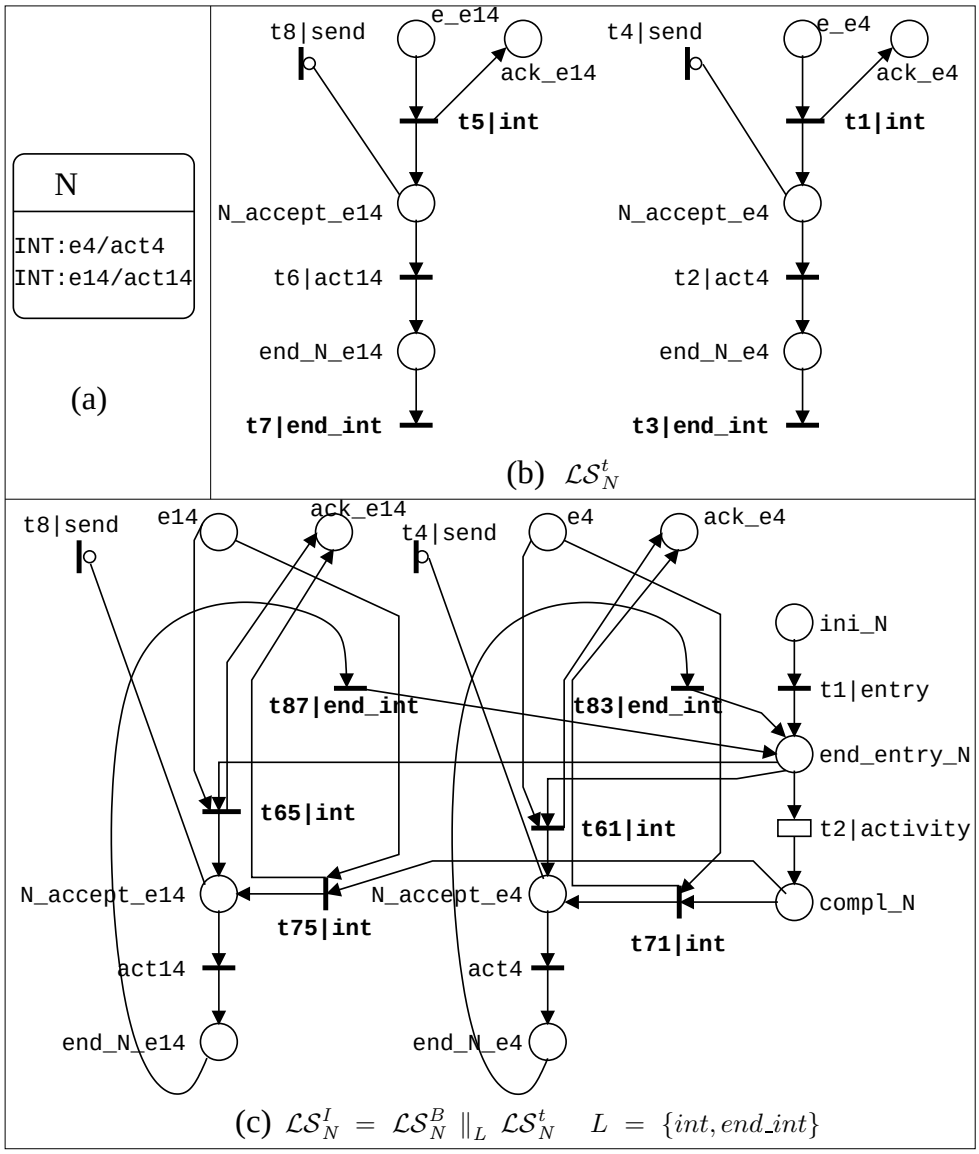


Figure 4.4: Translation of the internal transitions of the state N.

The elements of the $\mathcal{LS}_i^t$ system are the following:

- There are as many *event* places, labelled *e_event* as internal transitions exist in the state. A token in one of these places represents an instance of an event of the type *event*. Places labelled with event names represent event queues (observe that no policy is associated to the place, apart for the choice of the term "queue"). These places are formally stated in Definition 4.21.
- There are as many immediate *interface* transition labelled *int* as internal transitions. Each one has an input arc from the *event* place, the event is obviously that which triggers the internal transition in the **SM**, (see Definition 4.26).
- There are as many *acknowledge* places, labelled *ack_event* as internal transitions exist in the state. A token in one of these places represents that an instance of an event of the type *event* has been consumed, (see Definition 4.22).
- There are as many *st_accept_event* places as internal transitions. Each one represent that an event has been accepted. It has an input arc from the *int* transition, (see Definition 4.23).
- There are as many immediate transition, labeled *action*, as internal transitions. Each one represent the execution of the internal action in the **SM**. It has an input arc from the *state_accept_event* place, (see Definition 4.27).
- A *end_action_st_event* place to represent in the $\mathcal{LS}_i^t$ that the action of the internal transition has been completed. It has an input arc from the *action* transition, (see Definition 4.24).
- An immediate *interface* transition labelled *end_int*. It has an input arc from the *end_action* place, (see Definition 4.28).
- There are as many immediate transition, labelled *send*, as internal transitions. Each one has an inhibitor arc from the *state_accept_event* place to represent that the *send* transition can fire only if an internal transition has not been accepted, (see Definition 4.25).

Step B. The $\mathcal{LS}_i^I$ system

The $\mathcal{LS}_i^B$ and the $\mathcal{LS}_i^t$ systems are composed using superposition of transitions in a new one named $\mathcal{LS}_i^I$. We use this system to explain the Petri net interpretation of the internal transitions. The superposition is performed using the interface transitions labeled *int* and *end_int*. The composition of the $\mathcal{LS}_N^t$ in Figure 4.4(b) and the $\mathcal{LS}_N^B$ in Figure 4.2(a) is shown in Figure 4.4(c), we remark that transitions $\{loop\_i, send, def, out\}$ from $\mathcal{LS}_N^B$ have been hided. In the following the new system is commented stressing the composition over the translation of the internal transition $e4/act4$.

Transitions $t61|int$ and $t71|int$ are the superposition of the transitions $t6|int$ (in $\mathcal{LS}_N^B$) and $t1|int$ (in $\mathcal{LS}_N^t$) and $t7|int$ (in $\mathcal{LS}_N^B$) and $t1|int$ (in $\mathcal{LS}_N^t$), respectively.

Obviously, the firing of any of these transitions represents the trigger in the **SM** of the transition caused by event $e4$.

Transition $t61|int$ has two input arcs. One of them from the place that represents the end of the entry action (end_entry_N) and the other from the place that represents the triggered event. Also, it has two output arcs. One to the place named N_accept_e4 and the other to the place ack_e4. The meaning of this transition in the **SM** is that an internal transition can be accepted while the activity is in execution, the consequences are:

1. A token is removed from place *end_entry_state*, which means that the activity is aborted and any other internal transition, nor deferred event, nor outgoing transition can be accepted.

2. A token is removed from place *e_event*, which means that an instance of an *event* has been accepted.

3. A token is added to place *state_accept_event*, which means that the action of the internal transition can be carried out.

4. A token is added to place *ack_event*, which means the acknowledge of the arrival of the *event* to the action which generates it, but actually there is no way to determine wheter the event has been generated by a synchronous or asynchronous action, the only possibility is that each transition that consumes an *event* puts a token into the place of label *ack_event*.

The other transition that represents the arrival of the event, $t71|int$, also has two input arcs. One of them from the place that represents the triggered event, it has the same meaning as in the previous case; the other removes a token from the completion event place ($compl\_N$), which means that any other internal transition, neither deferred event, nor outgoing transition can be accepted. Moreover, this transition has the same output arcs as $t61|int$ with the same meaning.

The immediate transition in $\mathcal{LS}_i^I$ that represents the action of the transition of the **SM** has just an input arc from place *state_accept_event* and an output arc to place *end_entry_state*. The meaning of the arc from place *state_accept_event* to the transition is that the action is executed immediately after the internal event has been accepted.

Transition $t83|end\_int$ is the superposition of transitions $t8|end\_int$ (in $\mathcal{LS}_N^B$) and $t3|end\_int$ (in $\mathcal{LS}_N^t$). Obviously, the firing of this transition means that when the action has been performed the activity must be newly executed and any other event can be accepted.

4.4.2 Formal translation

Step A. The $\mathcal{LS}_i^t$ system

Before to define the $\mathcal{LS}_i^t$ system let us introduce the relations between the elements in a flat state machine **SM** and the places and transitions for the system. As in the

previous sections, the notation for the LGSPN is the following, $\mathcal{LS}_i^t = (S_i^t, \psi_i^t, \lambda_i^t)$, with $S_i^t = \langle P_i^t, T_i^t, I_i^t, O_i^t, H_i^t, \Pi_i^t, W_i^t, {M_0}_i^t \rangle$.

Let us assume that $P_i^t = \{p_1, \ldots, p_k\}, T_i^t = \{t_1, \ldots, t_k\}, k = 4 * |int(s)|$.

Definition 4.21. *Let us define a relation,* $\Psi_{ie} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^t$*, between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^t$*, such that,*

$$\Psi_{ie} = \{(s, p_l) \mid s \in \Sigma_{ss}, int(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{t}}, l = 1 \ldots |int(s)|\}.$$

Definition 4.22. *Let us define a relation,* $\Psi_{iak} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^t$*, between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^t$*, such that,*

$$\Psi_{iak} = \{(s, p_l) \mid s \in \Sigma_{ss}, int(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{t}}, l = h + 1 \ldots 2h, h = |int(s)|\}.$$

Definition 4.23. *Let us define a relation,* $\Psi_{acc} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^t$*, between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^t$*, such that,*

$$\Psi_{icc} = \{(s, p_l) \mid s \in \Sigma_{ss}, int(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{t}}, l = 2h + 1 \ldots 3h, h = |int(s)|\}.$$

Definition 4.24. *Let us define a relation,* $\Psi_{eia} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^t$*, between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^t$*, such that,*

$$\Psi_{eia} = \{(s, p_l) \mid s \in \Sigma_{ss}, int(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{t}}, l = 3h + 1 \ldots 4h, h = |int(s)|\}.$$

Definition 4.25. *Let us define a relation,* $\Lambda_{si} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} T_j^t$*, between the set of simple states of* **SM** *and the set of transitions in* $\bigcup_{j=1}^{n} S_j^t$*, such that,*

$$\Lambda_{si} = \{(s, t_l) \mid s \in \Sigma_{ss}, int(s) \neq \emptyset, t_l \in T_{\mathrm{s}}^{\mathrm{t}}, l = 1 \ldots h, h = |int(s)|\}.$$

Definition 4.26. *Let us define a relation,* $\Lambda_{ie} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} T_j^t$*, between the set of simple states of* **SM** *and the set of transitions in* $\bigcup_{j=1}^{n} S_j^t$*, such that,*

$$\Lambda_{ie} = \{(s, t_l) \mid s \in \Sigma_{ss}, int(s) \neq \emptyset, t_l \in T_{\mathrm{s}}^{\mathrm{t}}, l = h + 1 \ldots 2h, h = |int(s)|\}.$$

Definition 4.27. *Let us define a relation,* $\Lambda_{ia} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} T_j^t$*, between the set of simple states of* **SM** *and the set of transitions in* $\bigcup_{j=1}^{n} S_j^t$*, such that,*

$$\Lambda_{ia} = \{(s, t_l) \mid s \in \Sigma_{ss}, int(s) \neq \emptyset, t_l \in T_{\mathrm{s}}^{\mathrm{t}}, l = 2h + 1 \ldots 3h, h = |int(s)|\}.$$

Definition 4.28. *Let us define a relation,* $\Lambda_{eie} : \Sigma_{ss} \hookrightarrow \bigcup_{j=1}^{n} T_j^t$*, between the set of simple states of* **SM** *and the set of transitions in* $\bigcup_{j=1}^{n} S_j^t$*, such that,*

$$\Lambda_{eie} = \{(s, t_l) \mid s \in \Sigma_{ss}, int(s) \neq \emptyset, t_l \in T_{\mathrm{s}}^{\mathrm{t}}, l = 3h + 1 \ldots 4h, h = |int(s)|\}.$$

Definition 4.29. *The system* $\mathcal{LS}_i^t = (S_i^t, \psi_i^t, \lambda_i^t)$ *associated to the internal transitions of a given state i of a flat UML state machine* **SM** *with* $S_{\mathrm{i}}^{\mathrm{t}} = \langle P_{\mathrm{i}}^{\mathrm{t}}, T_{\mathrm{i}}^{\mathrm{t}}, I_{\mathrm{i}}^{\mathrm{t}}, O_{\mathrm{i}}^{\mathrm{t}}, H_{\mathrm{i}}^{\mathrm{t}}, \Pi_{\mathrm{i}}^{\mathrm{t}}, W_{\mathrm{i}}^{\mathrm{t}}, {M_0}_{\mathrm{i}}^{\mathrm{t}} \rangle$ *is defined as follows:*

$P_i^t = \Psi_{ie}(i) \cup \Psi_{acc}(i) \cup \Psi_{eia}(i)$,
$T_i^t = \Lambda_{si}(i) \cup \Lambda_{ie}(i) \cup \Lambda_{ia}(i) \cup \Lambda_{eie}(i)$,

$$I_i^t(t) = \begin{cases} p_{ie} \in \Psi_{ie}(i) & \text{if } t \in \Lambda_{ie}(i) \\ p_{acc} \in \Psi_{acc}(i) & \text{if } t \in \Lambda_{ia}(i) \\ p_{eia} \in \Psi_{eia}(i) & \text{if } t \in \Lambda_{eie}(i) \end{cases}$$

$$O_i^t(t) = \begin{cases} \{p_{acc}, p_{iak}\} \in \Psi_{acc}(i) & \text{if } t \in \Lambda_{ie}(i) \\ p_{eia} \in \Psi_{eia}(i) & \text{if } t \in \Lambda_{ia}(i) \end{cases}$$

$$H_i^t(t) = \begin{cases} p_{acc} \in \Psi_{acc}(i) & \text{if } t \in \Lambda_{si}(i) \\ \emptyset & \text{otherwise} \end{cases}$$

$$W_i^t(t) = 1, \Pi_i^t(t) = 1 : \forall t \in T_i^t$$

The following restrictions must be fulfilled, $\forall p \in (\Psi_{acc}(i) \cup \Psi_{eia}(i)) : |^\bullet p| = |p^\bullet| = 1$ and $\forall p \in \Psi_{ie}(i) : |^\bullet p| = 0 \ \wedge \ |p^\bullet| = 1$ and $\forall p \in \Psi_{iak}(i) : |^\bullet p| = 1 \ \wedge \ |p^\bullet| = 0$.

$$\psi_i^t(p) = \begin{cases} e\_ev & \text{if } p \in \Psi_{ie}(i) \\ ack\_ev & \text{if } p \in \Psi_{iak}(i) \\ st\_accept\_ev & \text{if } p \in \Psi_{acc}(i) \\ end\_st\_ev & \text{if } p \in \Psi_{eia}(i) \\ & \text{where } ev = name(e) \ \wedge \ e \in trigger(int(i)) \ \wedge \ st = name(i) \end{cases}$$

this function is injective and the following restrictions must be fulfilled,

$\forall t \in \Lambda_{ie}(i) : \psi_i^t(I_i^t(t)) = e\_event, \psi_i^t(O_i^t(t)) = \{st\_accept\_event, ack\_event\} \Longrightarrow \exists \mid t_{in} \in int(i) : trigger(t_{in}) = ev \ \wedge \ name(ev) = event$

$\forall t \in \Lambda_{ia}(i) : \psi_i^t(I_i^t(t)) = st\_accept\_ev, \psi_i^t(O_i^t(t)) = end\_st\_ev \Longrightarrow \exists \mid t_{in} \in int(i) : trigger(t_{in}) = e \wedge name(e) = ev \wedge ((effect(t_{in}) = a \wedge name(a) = act) \vee (effect(t_{in}) = \emptyset \ \wedge \ act = \lambda))$.

$$\lambda_i^t(t) = \begin{cases} send & \text{if } t \in \Lambda_{si}(i) \\ int & \text{if } t \in \Lambda_{ie}(i) \\ (1) & \text{if } t \in \Lambda_{ia}(i) \\ end\_int & \text{if } t \in \Lambda_{eie}(i) \end{cases} \quad (1) = \begin{cases} name(a) \Longrightarrow \exists \mid t_{in} \in int(i) : effect(t_{in}) = a \\ \vee \\ \lambda \Longrightarrow \exists t_{in} \in int(i) : effect(t_{in}) = \emptyset \end{cases}$$

Step B. The $\mathcal{LS}_i^I$ system

Definition 4.30. *The system $\mathcal{LS}_i^I$ is the superposition of $\mathcal{LS}_i^{BI}$ and $\mathcal{LS}_i^t$ over the set L of labels,*

$$\mathcal{LS}_i^I \ = \ \mathcal{LS}_i^B \underset{L}{\parallel} \mathcal{LS}_i^t \qquad L \ = \ \{int, end\_int\}$$

4.5 Outgoing transitions and exit actions

In this section, we deal with the translation of the exit actions and the external transitions together due to practical pourposes. Concretely, because exit actions make sense in the context of outgoing transitions. An exit action is executed as a consecuence of the trigger of an outgoing transition.

Concerning exit actions, in section 2.12 of [Obj01] the following is stated: "Whenever a state is exited, it executes its exit action as the final step prior to leaving the state and upon exit, the activity is terminated before the exit action is executed".

Concerning outgoing transitions, in section 2.12 of [Obj01] it is stated: "If the state is exited as a result of the firing of an outgoing transition before the completion of the activity, the activity is aborted prior to its completion."

In an outgoing transition t the following elements may appear: A guard, $guard(t)$ (they are not considered in this work); an event which triggers the transition, $trigger(t)$; and an action as an effect, $effect(t)$. Upon the arrival of the event, the outgoing transition can be triggered both, while the activity is in execution, therefore it is aborted, or when the activity has finished. In both cases, the object transits to the *target* state of the transition.

In this section we will define a labelled system, named $\mathcal{LS}_i^g$, that interpretates in terms of Petri nets the exit action and the outgoing transitions in a simple state i.

4.5.1 Informal explanation

For each state in a given flat UML state machine **SM** with outgoing transitions a $\mathcal{LS}_i^g$ system is created (Definition 4.43 will present its formalization), see Figures 4.6(b) and 4.7(b) for an example. This system will be composed later in section 4.7 with the $\mathcal{LS}_i^B$, $\mathcal{LS}_i^d$ and $\mathcal{LS}_i^t$ systems in order to give a Petri net interpretation of the simple state i.

For practical pourposes, in order to explain how the PN interpretation of the outgoing transitions works without taking into account any other feautre of the state machine, we have created the $\mathcal{LS}_i^O$ system. This system is the superposition of $\mathcal{LS}_i^B$ system and $\mathcal{LS}_i^g$ system using the labels $\{out, out_{ce}, loop\_i\}$. In order to clarify the $\mathcal{LS}_i^O$ system, the interface transitions $\{send, def, int, end\_int\}$ of $\mathcal{LS}_i^B$ are not considered (they are hided) in $\mathcal{LS}_i^O$. Formal definitions of this system will came in Definition 4.44.

In the following, we describe informally the previous system using Figure 4.6 and 4.7 as an example.

Step A. The $\mathcal{LS}_i^g$ system

Depending on whether an outgoing transition is a self-loop transition (transition with the same *source* and *target* state) or not and whether the transition is immediate (transition without *trigger*) or not a different translation for the transition is proposed. Figure 4.5 shows the four cases that can arise, supposing evx is the *trigger* of the transition and $actx$ its *effect*.

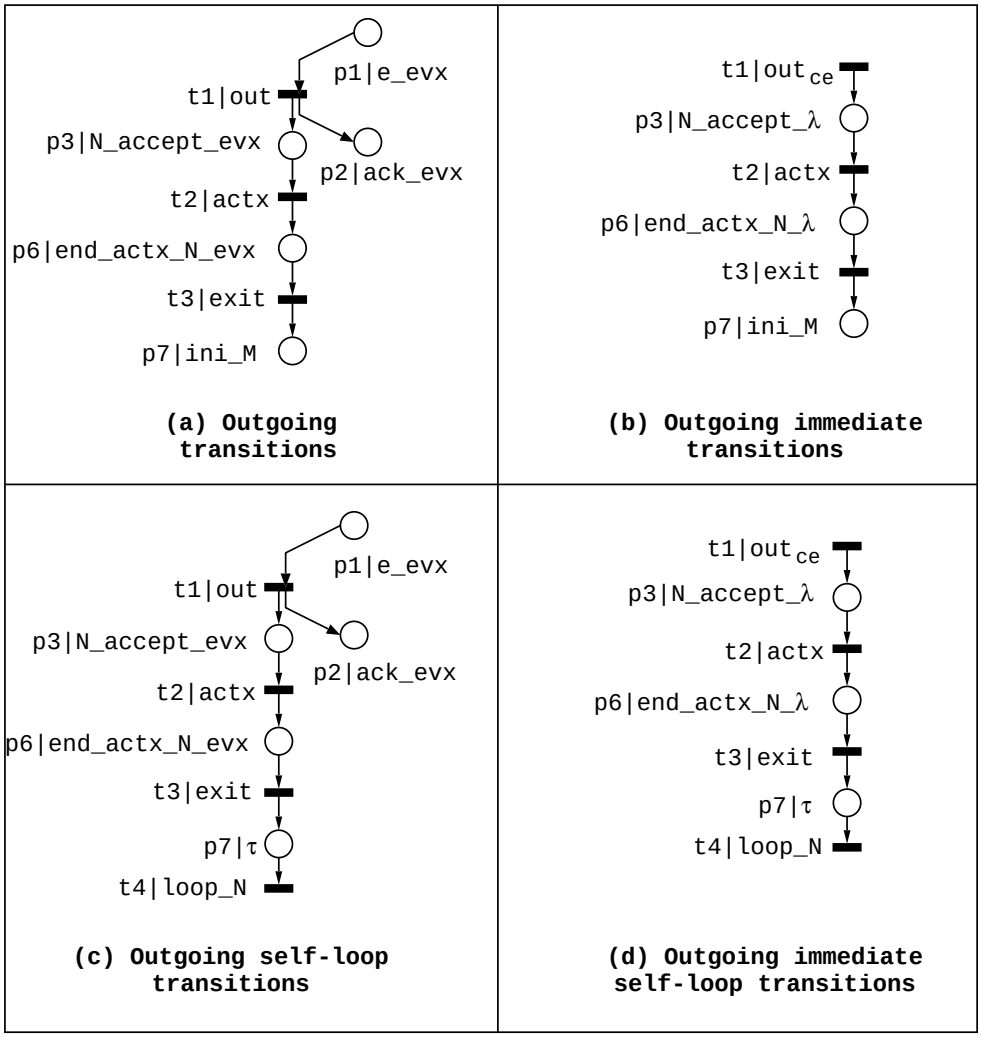


Figure 4.5: Translation of outgoing transitions.

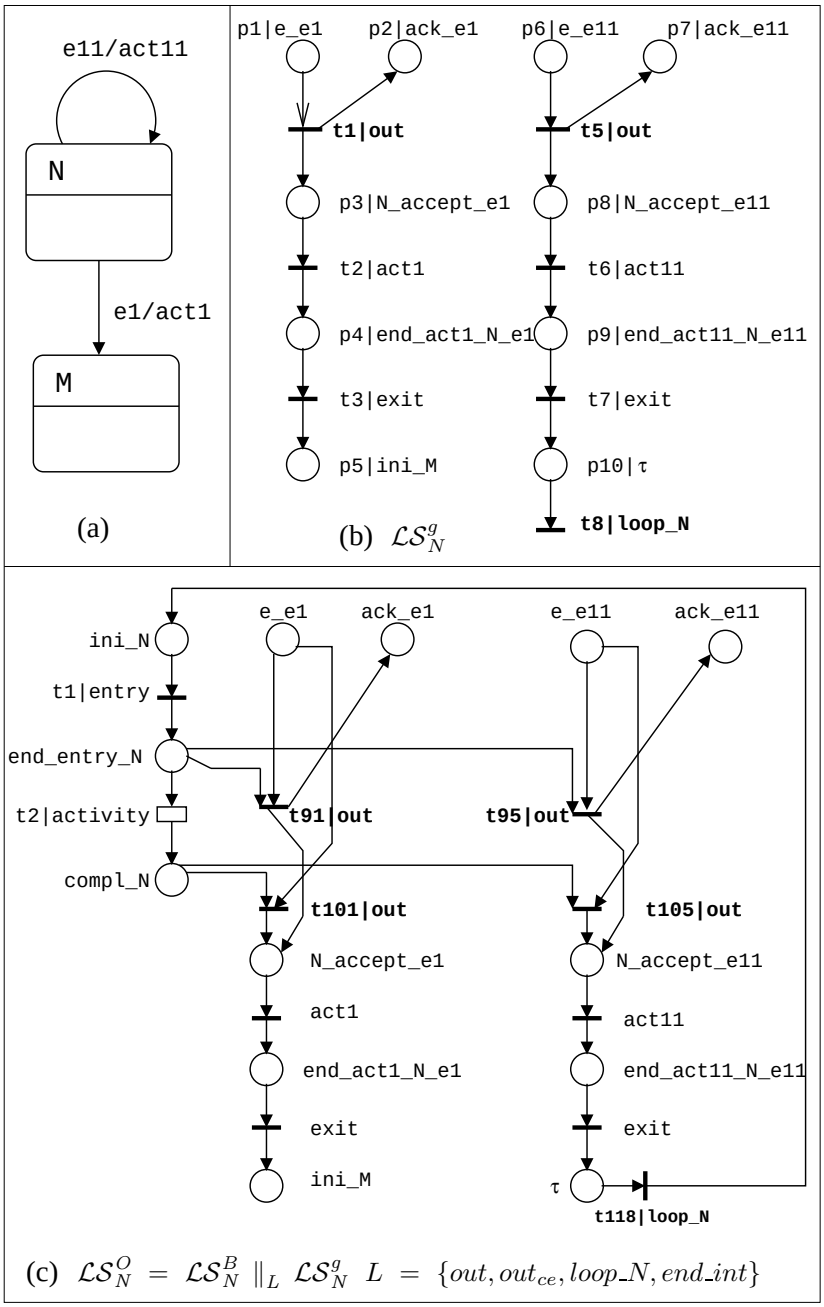


Figure 4.6: Translation of the outgoing transitions of the state N.

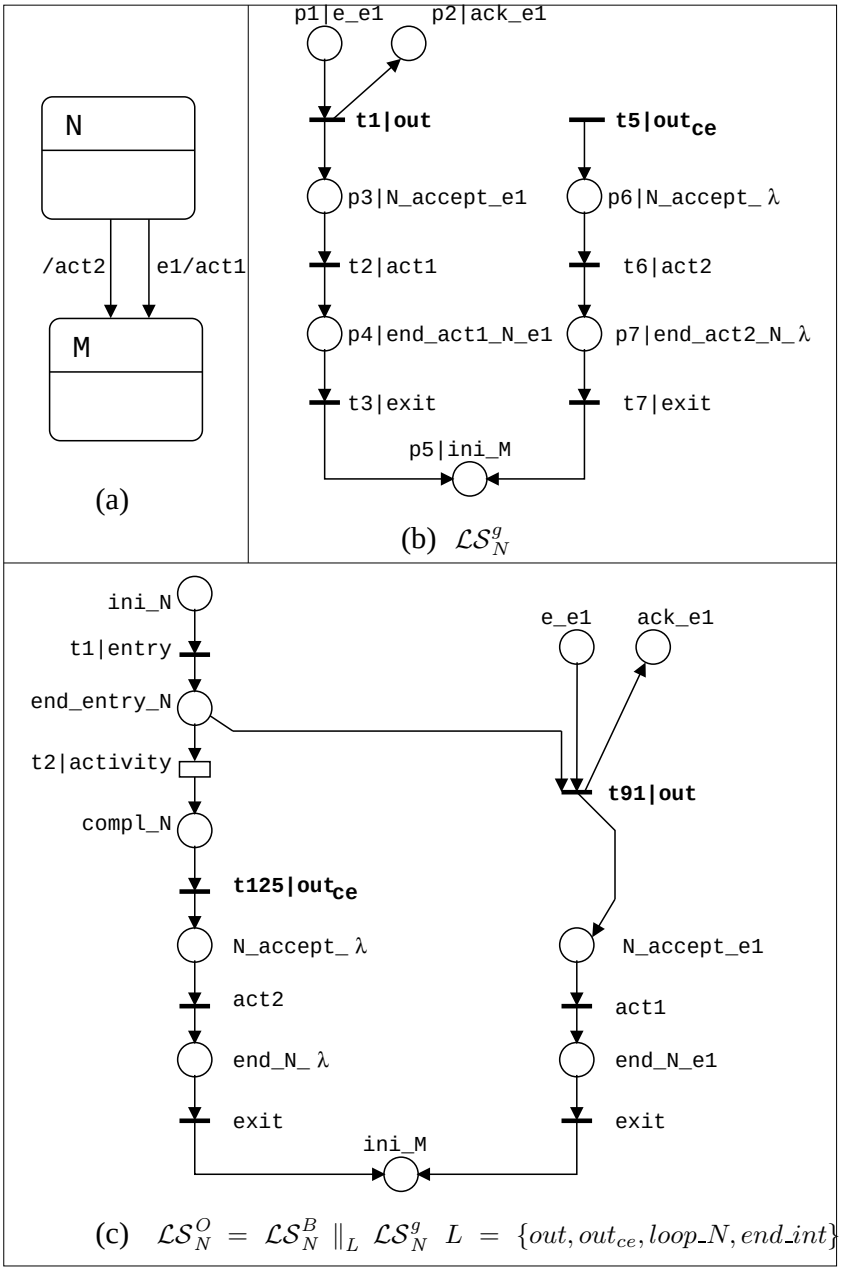


Figure 4.7: Translation of the outgoing transitions of the state N (with an immediate outgoing transition.

The $\mathcal{LS}_i^g$ system represents the translation of all the outgoing transitions in the state i. In Figure 4.6(b), we represent the translation of the outgoing transitions $e1/act1$ and $e11/act11$ for the state N in Figure 4.6(a) (extracted from Figure 4.1). The example in Figure 4.7(b) represents the translation of the outgoing transitions $e1/act1$ and $/act2$ for the state N in Figure 4.7(a) (also extracted from Figure 4.1, but we have added the transition $/act2$).

The elements of the $\mathcal{LS}_i^g$ system are the following:

- There are as many *event* places, labelled *e_event* and representing event queues, as outgoing transitions with *trigger* exist in the state. A token in one of these places represents an instance of an event of the type *event*. These places are formally introduced in Definition 4.37.

- There are as many immediate *interface* transition labelled *out* as outgoing transitions with *trigger*. Each of them has an input arc from an *e_event* place and an output arc to the *ack_event* place, the *event* is obviously that which triggers the transition in the **SM**. Definition 4.39 includes their formalization.

- There are as many *acknowledge* places, labelled *ack_event* as outgoing transitions exist in the state. A token in one of these places represents that an instance of an event of the type *event* has been consumed, (see Definition 4.38).

- If the state has an immediate outgoing transition, there is in the system an immediate *interface* transition labelled out_{ce}, (see Definition 4.39).

- There are as many *state_accept_event* places as outgoing transitions with *trigger*. Each one represents in the $\mathcal{LS}_i^g$ that the *event* has been accepted. Each place has an input arc from an *out* transition, (see Definition 4.33).

- There is a *state_accept_*λ place if an immediate outgoing transition exists. It represents in the $\mathcal{LS}_i^g$ that the *completion event* has been created. This place has an input arc from an out_{ce} transition, (see Definition 4.33).

- There are as many immediate transition, labelled *action* or λ, as outgoing transitions. Each one represents either the execution of the action of the outgoing transition, if it exists, or nothing. Each one has an input arc from the *state_accept_event* or *state_accept_*λ place, (see Definition 4.40).

- There are as many *end_action_state_event* places as outgoing transitions. Each one represents in the $\mathcal{LS}_i^g$ that the action of the outgoing transition has been completed. Each place has an input arc from an *action* or λ transition if there does not exist any action for the transition, (see Definition 4.34).

- There is a *end_state_*λ place if an immediate outgoing transition exists. It represents in the $\mathcal{LS}_i^g$ that the action of the outgoing transition has been completed. Each place has an input arc from an *action* or λ transition if there does not exist any action for the transition, (see Definition 4.34).

- There are as many immediate transition, labelled *exit* or λ, as outgoing transitions. Each of them represents in the $\mathcal{LS}_i^g$ the execution of the exit action of the state. Each transition has an input arc from the *end_action_state_event* place or the *end_action_state_*λ place, (see Definition 4.41).

- There is one place labelled *ini_target* for each state that is target of any outgoing transition. If two outgoing transitions have the same *target* only one place is created. Even if there exist self transitions, no place will be created for state i. These places become interface places to compose the system of the state i with the system of the *target* state, (see Definition 4.35).

- There are as many places labelled τ and as many transitions labelled *loop_state* as loop-transitions in the **SM**. The places are just buffers for the transitions. The transitions are used as interface transitions to compose this system with the "basic" system for the state i representing a re-entrance in the state i, (see Definition 4.42 and Definition 4.36, respectively).

Step B. The $\mathcal{LS}_i^O$ system

The $\mathcal{LS}_i^B$ and the $\mathcal{LS}_i^g$ systems are composed using superposition of transitions in a new one named $\mathcal{LS}_i^O$ in order to easily explain the PN interpretation of the outgoing transitions. The superposition is performed using the interface transitions labeled *out*, out_{ce} and *loop_i*.

The composition of the $\mathcal{LS}_N^g$ in Figure 4.6(b) and the $\mathcal{LS}_N^B$ in Figure 4.2(a) is shown in Figure 4.6(c). We remark that transitions $\{send, def, int, end\_int\}$ have been hided. In the following, the new system is commented stressing the composition over the translation of the outgoing transition $e1/act1$. The composition of the $\mathcal{LS}_N^g$ in Figure 4.7(b) and the $\mathcal{LS}_N^B$ in Figure 4.2(c) is shown in Figure 4.7(c), this system includes an immediate outgoing transition, it will not be described because its interpretation is straightforward considering the previous one.

Transition $t91|out$ ($t101|out$) is the superposition of the transitions $t9|out$ in $\mathcal{LS}_N^B$ and $t1|out$ in $\mathcal{LS}_N^g$ ($t10|out$ in $\mathcal{LS}_N^B$ and $t1|out$ in $\mathcal{LS}_N^g$). Obviously, the firing of this transition represents the arrival of the transition's event, $e1$ ($e11$), in **SM**.

Transition $t91|out$ has two input arcs: One of them from the place that represents the end of the entry action (*end_entry_st*) and the other from the place that represents the triggered event. Also, it has an output arc to a new place, named the *st_accept_ev*. The meaning of this transition is that an event can be accepted while the activity is in execution. The consequences of its firing are:

1. A token is removed from place *end_entry_st*, which means that the activity is aborted and **SM** cannot accept internal transitions, nor outgoing transitions nor deferred events.

2. A token is added to place *st_accept_ev*, which means that the *effect* of the outgoing transition can be carried out.

Transition $t101|out$ also has two input arcs: One of them from the place that represents the completion event of the state ($compl\_st$) and the other from the place that represents the triggered event. Also, it has an output arc to the place st_accept_ev. The meaning of this transition in the **SM** is that the event that *triggers* the outgoing transition can be accepted after the activity execution has finished, the consequences are the same as those caused by $t91|out$ but the activity is not aborted because it was completed.

Transitions $act1$ and $exit$ represent the execution of the action associated to the outgoing transition and the execution of the exit action in the source state of the outgoing transition.

The place ini_M represents in the **SM** the entrance in the state M as a consequence of the execution of the event $e1$.

The *interface* transition $t118|loop\_N$ represents in the **SM** that the state N is exited and newly entered, as a consequence of the execution of the event $e11$.

4.5.2 Formal translation

Step A. The $\mathcal{LS}_i^g$ system

Before to define the $\mathcal{LS}_i^g$ system let us introduce the relations between the elements in a given flat UML state machine **SM** and the places and transitions for the system. As in previous sections, the notation for the LGSPN system is $\mathcal{LS}_i^g = (S_i^g, \psi_i^g, \lambda_i^g)$, with $S_i^g = \langle P_i^g, T_i^g, I_i^g, O_i^g, H_i^g, \Pi_i^g, W_i^g, {M_0}_i^g \rangle$.

Definition 4.31. *Let us define a partial function,* $targets : \Sigma_{ss} \hookrightarrow \Sigma_{ss} \cup \Sigma_{fs}$, *such that,* $\forall s \in \Sigma_{ss}, out(s) \neq \emptyset : targets(s) = \{s' : \exists t \in out(s) \ \wedge \ target(t) = s' \ \wedge \ s' \neq s\}$

Definition 4.32. *Let us define a partial function,* $OUT_{loop} : \Sigma_{ss} \hookrightarrow \mathrm{T}_{SM}$, *such that,* $\forall s \in \Sigma_{ss}, out(s) \neq \emptyset : OUT_{loop}(s) = \{tr \in out(s) : source(tr) = target(tr) = s\}$

Let us assume that $P_i^g = \{p_1, \ldots, p_l\}$,
$l = (4 * |out(s)|) + |targets(s)| + |OUT_{loop}(s)| \ \wedge \ \nexists t \in out(s) : trigger(t) = \emptyset$
$\vee \ l = (4 * |out(s)|) + |targets(s)| + |OUT_{loop}(s)| - 1 \ \wedge \ \exists \mid t \in out(s) : trigger(t) = \emptyset.$
Let us assume that $T_i^g = \{t_1, \ldots, t_k\}, k = (3 * |out(s)|) + |OUT_{loop}(s)|$.

Definition 4.33. *Let us define a relation,* $\Psi_{ace} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^g$, *between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^g$, *such that,*

$$\Psi_{ace} = \{(s, p_l) \mid s \in \Sigma_{ss}, out(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{g}}, l = 1 \ldots h, h = |out(s)|\}.$$

Definition 4.34. *Let us define a relation,* $\Psi_{ea} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^g$, *between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^g$, *such that,*

$$\Psi_{ea} = \{(s, p_l) \mid s \in \Sigma_{ss}, out(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{g}}, l = h + 1 \ldots 2h, h = |out(s)|\}.$$

Definition 4.35. *Let us define a relation,* $\Psi_{oi} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^g$, *between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^g$, *such that,*

$$\Psi_{oi} = \{(s, p_l) \mid s \in \Sigma_{ss}, out(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{g}}, l = 2h + 1 \ldots q, q = |targets(s)| + 2h\}$$

Definition 4.36. *Let us define a relation,* $\Psi_b \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^g$*, between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^g$*, such that,*

$$\Psi_b = \{(s, p_l) \mid s \in \Sigma_{ss}, out(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{g}}, l = q+1 \ldots r, r = |OUT_{loop}(s)| + q\}$$

Definition 4.37. *Let us define a relation,* $\Psi_o \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^g$*, between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^g$*, such that,*

$$\begin{aligned}\Psi_o = \{(s, p_l) \mid s \in \Sigma_{ss}, out(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{g}}, l = r+1 \ldots t\} \ \wedge \\ ((t = |out(s)| + r \wedge \nexists t \in out(s) : trigger(t) = \emptyset) \\ \vee \ (t = |out(s)| + r - 1 \wedge \exists \mid t \in out(s) : trigger(t) = \emptyset))\end{aligned}$$

Definition 4.38. *Let us define a relation,* $\Psi_{oak} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} P_j^g$*, between the set of simple states of* **SM** *and the set of places in* $\bigcup_{j=1}^{n} S_j^g$*, such that,*

$$\begin{aligned}\Psi_{oak} = \{(s, p_l) \mid s \in \Sigma_{ss}, out(s) \neq \emptyset, p_l \in P_{\mathrm{s}}^{\mathrm{g}}, l = t+1 \ldots r\} \ \wedge \\ ((r = |out(s)| + t \wedge \nexists t \in out(s) : trigger(t) = \emptyset) \\ \vee \ (r = |out(s)| + t - 1 \wedge \exists \mid t \in out(s) : trigger(t) = \emptyset))\end{aligned}$$

Definition 4.39. *Let us define a relation,* $\Lambda_e \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} T_j^g$*, between the set of simple states of* **SM** *and the set of transitions in* $\bigcup_{j=1}^{n} S_j^g$*, such that,*

$$\begin{aligned}\Lambda_e = \{(s, t_l) \mid s \in \Sigma_{ss}, out(s) \neq \emptyset, t_l \in T_{\mathrm{s}}^{\mathrm{g}}\} \ \wedge \\ t_l = t_{ce} \cup \{t_2, \ldots, t_h\}, h = |out(s)| \qquad \text{if } s \in \Sigma_{ss}^{\circ} \\ t_l = \{t_1, \ldots, t_h\}, h = |out(s)| \qquad \text{if } s \in \Sigma_{ss}^{\bullet}\end{aligned}$$

Definition 4.40. *Let us define a relation,* $\Lambda_a \subseteq \Sigma_{ss} \times T$*, between the set of simple states of* **SM** *and the set of transitions in* $\bigcup_{j=1}^{n} S_j^g$*, such that,*

$$\Lambda_a = \{(s, t_l) \mid s \in \Sigma_{ss}, out(s) \neq \emptyset, t_l \in T_{\mathrm{s}}^{\mathrm{g}}, l = h+1 \ldots h, h = |out(s)|\}$$

Definition 4.41. *Let us define a relation,* $\Lambda_{ex} \subseteq \Sigma_{ss} \times T$*, between the set of simple states of* **SM** *and the set of transitions in* S*, such that,*

$$\Lambda_{ex} = \{(s, t_l) \mid s \in \Sigma_{ss}, out(s) \neq \emptyset, t_l \in T_{\mathrm{s}}^{\mathrm{g}}, l = 2h+1 \ldots 3h, h = |out(s)|\}$$

Definition 4.42. *Let us define a relation,* $\Lambda_{ol} \subseteq \Sigma_{ss} \times \bigcup_{j=1}^{n} T_j^g$*, between the set of simple states of* **SM** *and the set of transitions in* $\bigcup_{j=1}^{n} S_j^g$*, such that,*

$$\begin{aligned}\forall s \in \Sigma_{ss}, OUT_{loop}(s) \neq \emptyset : \Lambda_{ol}(s) = \{t_{3h+1}, \ldots, t_k\}, k = |OUT_{loop}(s)| + 3h \\ \Lambda_{ol} = \{(s, t_l) \mid s \in \Sigma_{ss}, out(s) \neq \emptyset, t_l \in T_{\mathrm{s}}^{\mathrm{g}}, l = 3h+1 \ldots k, k = |OUT_{loop}(s)| + 3h\}\end{aligned}$$

Definition 4.43. *The system* $\mathcal{LS}_i^g = (S_i^g, \psi_i^g, \lambda_i^g)$ *associated to the outgoing transitions and exit actions of a given state* i *of a flat UML state machine* **SM** *with* $S_{\mathrm{i}}^{\mathrm{g}} = \langle P_{\mathrm{i}}^{\mathrm{g}}, T_{\mathrm{i}}^{\mathrm{g}}, I_{\mathrm{i}}^{\mathrm{g}}, O_{\mathrm{i}}^{\mathrm{g}}, H_{\mathrm{i}}^{\mathrm{g}}, \Pi_{\mathrm{i}}^{\mathrm{g}}, W_{\mathrm{i}}^{\mathrm{g}}, M_{0\mathrm{i}}^{\mathrm{g}} \rangle$ *is defined as follows:*

$P_i^g = \Psi_o(i) \cup \Psi_{oak}(i) \cup \Psi_{ace}(i) \cup \Psi_{ea}(i) \cup \Psi_{ib}(i)$,
$T_i^g = \Lambda_e(i) \cup \Lambda_a(i) \cup \Lambda_{ex}(i) \cup \Lambda_{ol}(i)$,

$$I_i^g(t) = \begin{cases} \emptyset & \text{if } t = t_{ce} \\ p_o \in \Psi_o(i) & \text{if } t \in (\Lambda_e(i) - t_{ce}) \\ p_{ee} \in \Psi_{ee}(i) & \text{if } t \in \Lambda_a(i) \\ p_{ace} \in \Psi_{ace}(i) & \text{if } t \in \Lambda_{ex}(i) \\ p_{ib} \in \Psi_{ib}(i) & \text{if } t \in \Lambda_{ol}(i) \end{cases}$$

$$O_i^g(t) = \begin{cases} p_{ace} \in \Psi_{ace}(i) & \text{if } t = t_{ce} \\ p_{ace} \in \Psi_{ace}(i) \cup p_{oak} \in \Psi_{oak}(i) & \text{if } t \in (\Lambda_e(i) - t_{ce}) \\ p_{ea} \in \Psi_{ea}(i) & \text{if } t \in \Lambda_a(i) \\ p_{ib} \in \Psi_{ib}(i) & \text{if } t \in \Lambda_{ex}(i) \end{cases}$$

$$H_i^g(t) = \emptyset, \Pi_i^g(t) = 1, W_i^g(t) = 1 : \forall t \in T_i^g$$

the following restrictions must be fulfilled, $\forall p \in (\Psi_{ace}(i) \cup \Psi_{ee}(i) \cup \Psi_{ea}(i)) : |^\bullet p| = |p^\bullet| = 1$ and $\forall p \in \Psi_o(i) : |^\bullet p| = 0 \wedge |p^\bullet| = 1$ and $\forall p \in \Psi_{oak}(i) : |^\bullet p| = 1 \wedge |p^\bullet| = 0$ and $\forall p \in \Psi_{oi}(i) : |^\bullet p| \geq 1 \wedge |p^\bullet| = 0$ and $\forall p \in \Psi_b(i) : |^\bullet p| = 1 \wedge |p^\bullet| = 1$.

$$\psi_i^g(p) = \begin{cases} e\_ev & \text{if } p \in \Psi_o(i) \\ ack\_ev & \text{if } p \in \Psi_{oak}(i) \\ st\_accept\_ev & \text{if } p \in \Psi_{ace}(i) \\ end\_act\_st\_ev & \text{if } p \in \Psi_{ea}(i) \\ ini\_st' & \text{if } p \in \Psi_{oi}(i) \wedge i \neq i' \\ \tau & \text{if } p \in \Psi_b(i) \wedge i = i' \\ & \text{where:} \\ & st = name(i) \wedge st' = name(i') \wedge \\ & \exists \mid t_{out} \in out(i) : (target(t_{out}) = i') \wedge \\ & \quad ((ev = \lambda \text{ if } trigger(t_{out}) = \emptyset) \vee \\ & \quad (ev = name(evt) \text{ if } trigger(t_{out}) = evt)) \wedge \\ & \quad ((act = \lambda \text{ if } effect(t_{out} = \emptyset) \vee \\ & \quad (act = name(a) \text{ if } effect(t_{out} = a)) \end{cases}$$

this function is injective and the following restrictions must be fulfilled,

$\psi_i^g(O_i^g(t_{ce})) = st\_accept\_\lambda$.

$\forall t \in \Lambda_e(i) - t_{ce} : \psi_i^g(I_i^g(t)) = e\_ev, \psi_i^g(O_i^g(t)) = \{st\_accept\_ev, ack\_ev\} \Longrightarrow \exists \mid t_{out} \in out(i) : trigger(t_{out}) = evt \wedge name(evt) = ev$.

$\forall t \in \Lambda_a(i) : \psi_i^g(I_i^g(t)) = st\_accept\_ev, \psi_i^g(O_i^g(t)) = end\_act\_st\_ev \Longrightarrow (\exists \mid t_{out} \in out(i) : trigger(t_{out}) = e \ \wedge \ name(e) = ev \ \wedge \ ((effect(t_{out}) = a \ \wedge \ name(a) = act) \ \vee \ (effect(t_{out}) = \emptyset \ \wedge \ act = \lambda))) \ \vee \ (\exists \mid t_{out} \in out(i) : trigger(t_{out}) = \emptyset \ \wedge \ ev = \lambda \ \wedge \ ((effect(t_{out}) = a \ \wedge \ name(a) = act) \ \vee \ (effect(t_{out}) = \emptyset \ \wedge \ act = \lambda)))$.

$$\forall t \in \Lambda_{ex}(i) = \begin{cases} \psi_i^g(I_i^g(t)) = end\_act\_st\_ev, \psi_i^g(O_i^g(t)) = ini\_st' \Longrightarrow \\ \qquad (\exists \mid t_{out} \in out(i) : (e_1) \wedge (e_3)) \ \vee \ (\exists \mid t_{out} \in out(i) : (e_2) \wedge (e_3)) \\ \vee \\ \psi_i^g(I_i^g(t)) = end\_act\_st\_ev, \psi_i^g(O_i^g(t)) = \tau \Longrightarrow \\ \qquad (\exists \mid t_{out} \in OUT_{loop}(i) : (e_1)) \ \vee \ (\exists \mid t_{out} \in OUT_{loop}(i) : (e_2)) \end{cases}$$

$(e_1) = (trigger(t_{out}) = evt \ \wedge \ name(evt) = ev)$.
$(e_2) = (trigger(t_{out}) = \emptyset \ \wedge \ ev = \lambda)$.
$(e_3) = (st' = name(s') \ \wedge \ s' = target(t_{out}))$.

$$\lambda_i^g(t) = \begin{cases} out_{ce} & \text{if } t = t_{ce} \\ out & \text{if } t \in \Lambda_e(i) - t_{ce} \\ (e_4) & \text{if } t \in \Lambda_a(i) \\ (e_5) & \text{if } t \in \Lambda_{ex}(i) \\ loop\_st & \text{if } t \in \Lambda_{ol}(i) \ \wedge \ st = name(i) \end{cases}$$

$$(e_4) = \begin{cases} name(a) & \text{if } exit(i) \neq \emptyset \ \wedge \ a = exit(i) \\ \vee & \\ \lambda & \text{if } exit(i) = \emptyset \end{cases}$$

$$(e_5) = \begin{cases} name(a) \Longrightarrow \exists \mid t_{out} \in out(i) : effect(t_{out}) = a \\ \vee \\ \lambda \Longrightarrow \exists t_{out} \in out(i) : effect(t_{out}) = \emptyset \end{cases}$$

Step B. The $\mathcal{LS}_i^O$ system

Definition 4.44. *The $\mathcal{LS}_i^O$ system is the superposition of $\mathcal{LS}_i^B$ and $\mathcal{LS}_i^g$ over the set L of labels,*

$$\mathcal{LS}_i^O \ = \ \mathcal{LS}_i^B \underset{L}{\parallel} \mathcal{LS}_i^g \qquad L \ = \ \{out, out_{ce}, loop\_i\}$$

4.6 Actions

Actions are specifications of executable states and can be realized by:

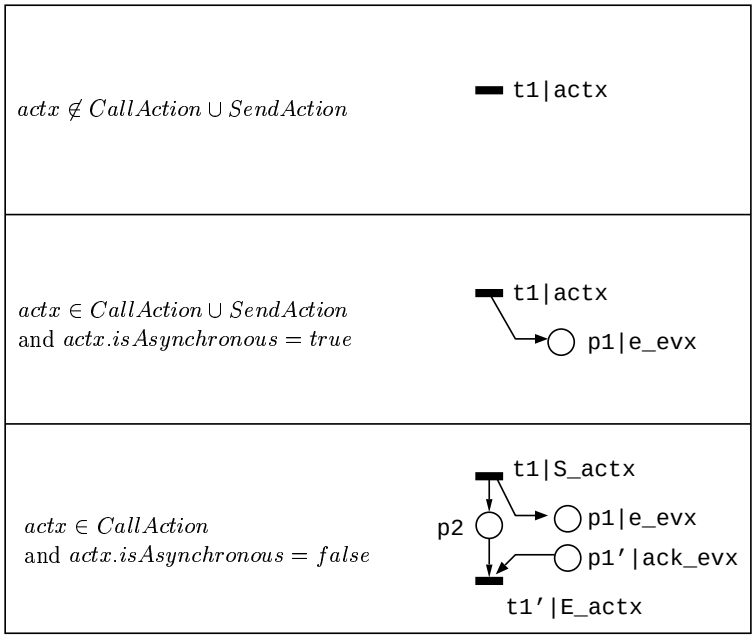


Figure 4.8: Translation of the different types of actions.

A) modifying a link or a value (actions that do not belong to subclass *CallAction* or *SendAction*),

B) generating one or more events (actions belonging to the subclass *CallAction* or *SendAction*). Moreover, they are characterized by the attribute *isAsynchronous* that allows to specify if the dispatched stimulus is asynchronous or not, where *synchronous* means that the action will not be completed until the event eventually generated by the action is not consumed by the receiver.

In the following, we show how actions must be translated depending on whether they belong to the case A or B. Figure 4.8 shows the different translations of an action:

A) Figure 4.8(a) shows how an action, $actx$, belonging to case A is translated just as a transition labelled with its name.

B.1) Figure 4.8(b) shows how an action, $actx$, belonging to case B and being *asynchronous* is translated. A transition is created to represent the action. Moreover, as many places are created as events exist in the set $operation(signal(actx))$ if $actx \in CallAction$ or in the set $ocurrence(signal(actx))$ if $actx \in SendAction$. Each place has an input arc from the transition. In Figure 4.8(b) $ocurrence(signal(actx)) = \{evx\}$

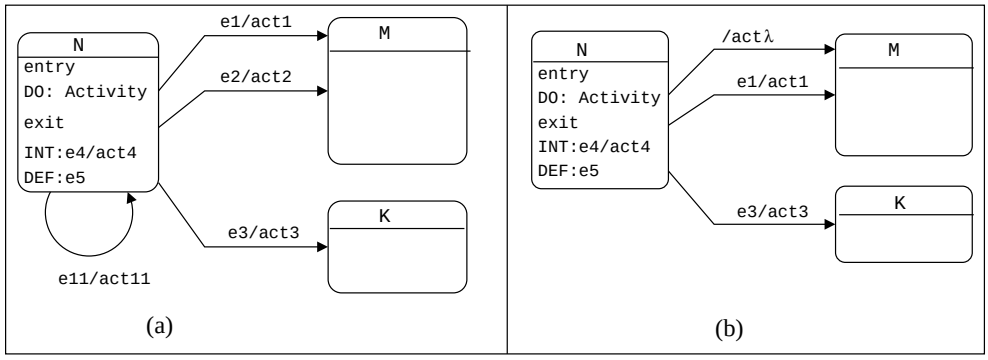


Figure 4.9: Examples of simple states (a)From Fig. 4.1 but adding transition $e2/act2$, (b)From Fig. 4.1 but adding transition $/act\lambda$.

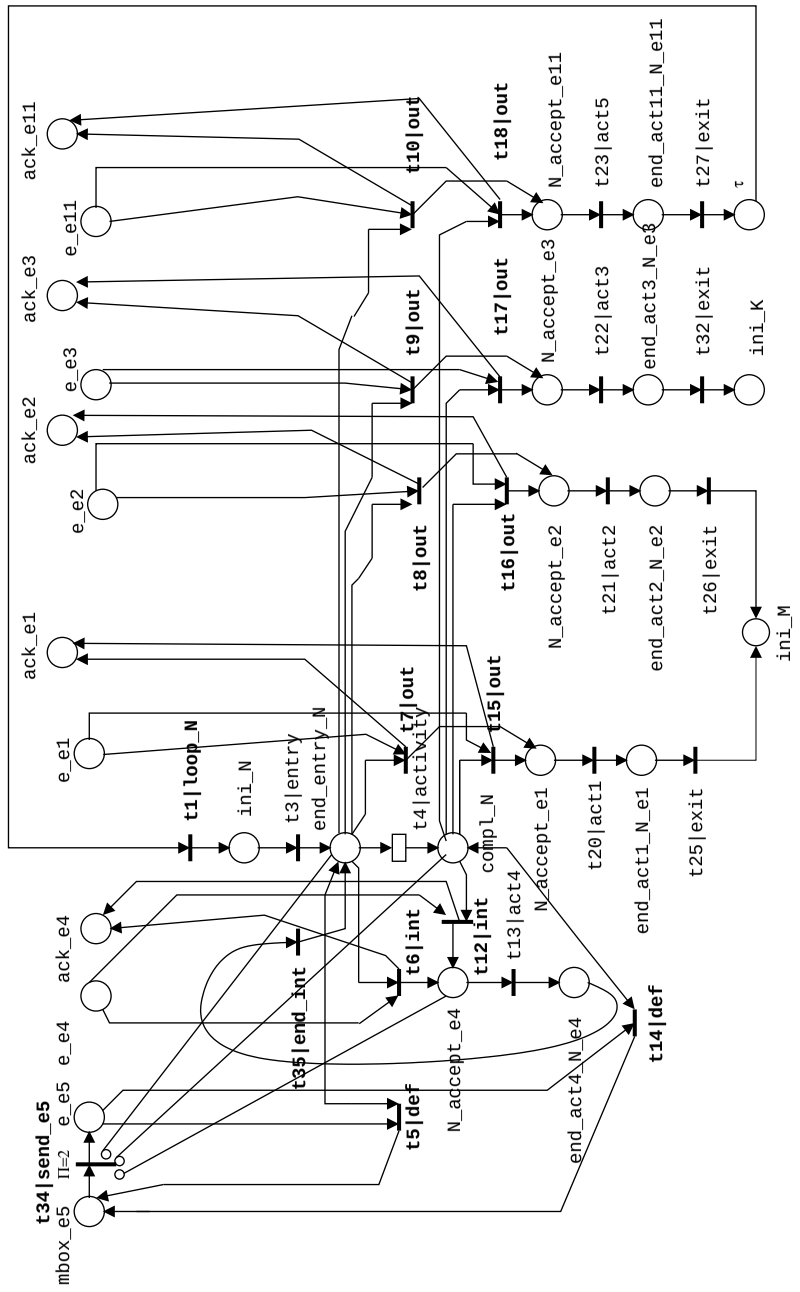
B.2) Figure 4.8(c) shows how an action, *actx*, belonging to case B and being *synchronous* is translated. Note that only transitions belonging to class *CallAction* can be *synchronous*. Transition $t1|S\_actx$ represents the start of the action and transition $t2|E\_actx$ the end. As in the previous case, as many places are created as events are generated. Moreover, for each event place an acknowledge place has been added, in this place a token will be added when the target object executes the corresponding *operation* letting to finish the action. Place $p2$ acts as a buffer.

Finally, we recall that different kinds of actions can appear in a state: *entry actions, exit actions, doActivities* and the *effect* of a transition. For the sake of simplicity, we have proposed in the previous sections to translate them as if they belong to the case A. On the contrary, if they belong to cases B.1 or B.2, it is straightforward to devise its formalization taking into account the description given in this section.

4.7 The model of a simple state

The labelled systems obtained in the previous sections can be composed using the operator in Definition 2.6 (cfr. chapter 2). The resulting labelled system $\mathcal{LS}_i$ interprets a simple state i together with its outgoing transitions. Obviously, the interest of this system is not to perform any kind of analysis but to establish the fine grain unit to compose state machines (together with the labelled systems for the initial pseudostate and the final states).

According to the translations defined up to now, given a state i with internal transitions, deferred events, and outgoing transitions, we get four labelled systems (one for each feature and the "basic" system) that need to be combined to get a model of the state i.



$$\mathcal{LS}_N = ((\mathcal{LS}^t_N \mid\mid_{Lev^P} \mathcal{LS}^d_N) \mid\mid_{Lev^P} \mathcal{LS}^g_N) \mid\mid_{Ltr^T} \mathcal{LS}^B_N$$

Figure 4.10: Translation of the simple state N in Figure 4.9(a).

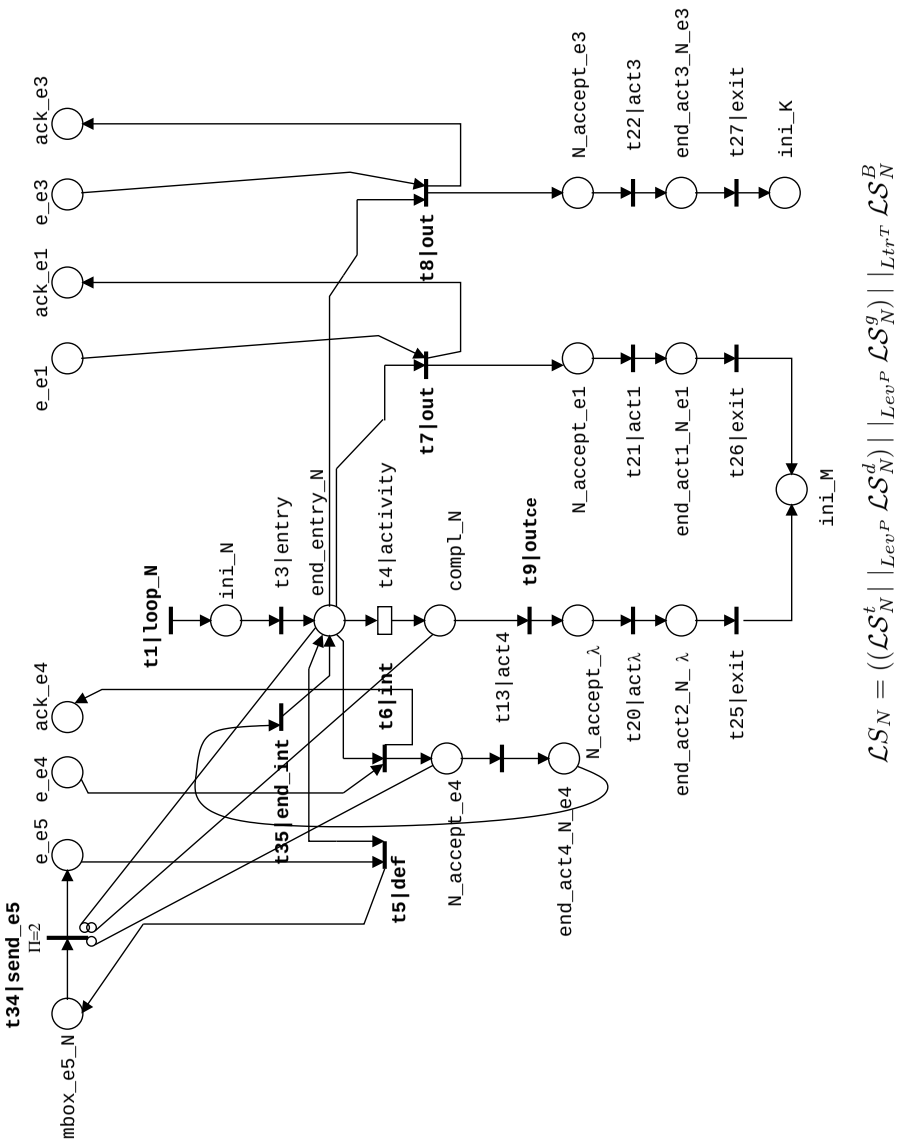


Figure 4.11: Translation of the simple state N in Figure 4.9(b).

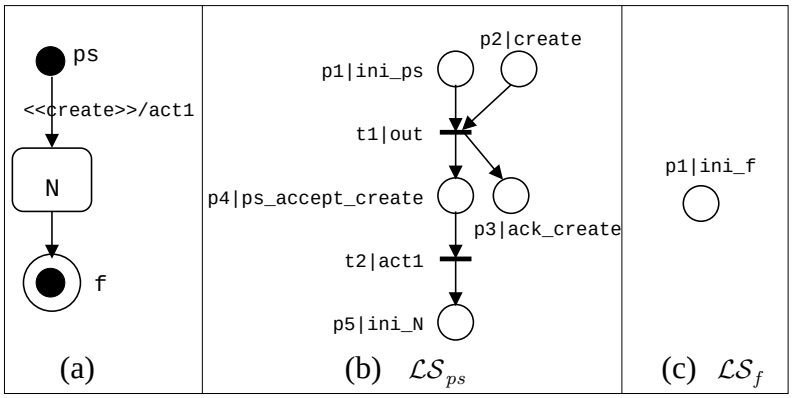


Figure 4.12: Translation of the initial pseudostate and the final state.

Definition 4.45. *The system* $\mathcal{LS}_i = (S_i, \psi_i, \lambda_i)$ *that represents the state* i *is defined as follows,*

$$\mathcal{LS}_i = ((\mathcal{LS}_i^t \underset{Lev^P}{||} \mathcal{LS}_i^d) \underset{Lev^P}{||} \mathcal{LS}_i^g) \underset{Ltr^T}{||} \mathcal{LS}_i^B$$

where Lev^P is the set of labels of event and event acknowledge places, $Lev^P = \{e\_evx, \forall evx \in Ev\} \bigcup \{ack\_evx, \forall evx \in Ev\}$ with Ev is the set of events produced/consumed by i . Moreover, $Ltr^T = \{int, end\_int, def, send, out, out_{ce}, loop\_i\}$.

Figures 4.10 and 4.11 show two examples of the system that models a simple state. The main difference among them is that the first system has not an immediate outgoing transitions while the second one has.

4.8 Initial pseudostates

In a flat UML state machine at most one initial pseudostate can appear, let us name it *ps* by convention. An initial pseudostate is graphycally depicted by a black dot and represents the starting point of the state machine. In the metamodel, it belongs to the class Pseudostate that is a subclass of the class StateVertex.

From the well-formedness rules of the state machines package [Obj01] is interesting to note that:

- An initial vertex can have at most one outgoing transition and no incoming transitions,
  $(self.kind = \#initial)$ *implies*
  $((self.outgoing \rightarrow size \leq 1)$ *and* $(self.incoming \rightarrow isEmpty))$
- An initial transition at the top most level either has no trigger or it has a trigger with the stereotype “create”,

$$
\begin{array}{l}
self.source.oclIsKindOf(Pseudostate)\ implies \\
\quad (self.source.oclAsType(Pseudostate).kind\ =\ \#initial)\ implies \\
\quad\quad (self.source.container\ =\ self.stateMachine.top)\ implies \\
\quad\quad\quad ((self trigger \rightarrow isEmpty)\ or \\
\quad\quad\quad\quad (self.trigger.stereotype.name\ =\ 'create'))
\end{array}
$$

In this section we will define a labelled system, named $\mathcal{LS}_{ps}$, that interpretates in terms of Petri nets an initial pseudostate ps.

4.8.1 Informal explanation

For the initial state of a given flat UML state machine **SM** the labelled system $\mathcal{LS}_{ps}$ is created (Definition 4.52 will present its formalization). Figure 4.12 depicts an example. This system represents the interpretation of the initial pseudostate and its outgoing transition. An informal explanation of this interpretation is given in this section by using the proposed example in order to easily understand the formal definition give in the next section.
The elements of $\mathcal{LS}_{ps}$ are the following:

- A token in place $p1$ represents a resource waiting for an instance event of type "create". A formal definition will be given in Definition 4.46.
- The places labelled *create* and *ack_create* represent, respectively, the queue of the event that creates instances and the acknowledge of its arrival. They will be formalized in Definition 4.47.
- The firing of the immediate transition, labeled *out*, represents the arrival of the event that fires the outgoing transition in **SM**, (see Definition 4.50).
- The place, labelled *ps_accept_create* represents in the $\mathcal{LS}_{ps}$ that the event has been accepted, (see Definition 4.48).
- The immediate transition $t2$ represents the execution of the action if it is labelled with the *effect* or nothing if it is labelled λ, (see Definition 4.51).
- After completion of the action, a token in place $p5$ means the completion of the initial pseudostate therefore the entry in state N, (see Definition 4.49).

4.8.2 Formal translation

The $\mathcal{LS}_{ps}$ system

Before to define the $\mathcal{LS}_{ps}$ system let us introduce the relations between the initial pseudostate of a given flat UML state machine **SM** and the places and transitions for the system. As in previous sections, $\mathcal{LS}_{ps} = (S_{ps}, \psi_{ps}, \lambda_{ps})$, with $S_{ps} = \langle P_{ps}, T_{ps}, I_{ps}, O_{ps}, H_{ps}, \Pi_{ps}, W_{ps}, M_{0\,ps} \rangle$.

Let us assume that $P_{ps} = \{p_i, p_c, p_{cak}, p_a, p_{i'}\}, T_{ps} = \{t_{ine}, t_{ina}\}$.

Definition 4.46. *Let us define a function,* $\Psi_i : ps \longrightarrow P_{ps}$*, from the initial pseudostate of* **SM** *to the set of places in* S_{ps}*, such that,*

$$\Psi_i(ps) = p_i.$$

Definition 4.47. *Let us define a partial function,* $\Psi_c : ps \hookrightarrow P_{ps} \times P_{ps}$*, from the initial pseudostate of* **SM** *to the set of places in* $S_{ps} \times S_{ps}$*,such that,*

$$\Psi_c(ps) = \begin{cases} \{p_c, p_{cak}\} & \text{if } \exists \mid t \in \mathrm{T}_{SM} : t \in out(ps) \ \wedge \ trigger(t) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 4.48. *Let us define a function,* $\Psi_a : ps \longrightarrow P_{ps}$*, from the initial pseudostate of* **SM** *to the set of places in* S_{ps}*, such that,*

$$\Psi_a(ps) = p_a.$$

Definition 4.49. *Let us define a function,* $\Psi_{i'} : ps \longrightarrow P_{ps}$*, from the initial pseudostate of* **SM** *to the set of places in* S_{ps}*, such that,*

$$\Psi_{i'}(ps) = p_{i'}.$$

Definition 4.50. *Let us define a function,* $\Lambda_{ine} : ps \longrightarrow T_{ps}$*, from the initial pseudostate of* **SM** *to the set of transitions in* S_{ps}*, such that,*

$$\Lambda_{ine}(ps) = t_{ine}.$$

Definition 4.51. *Let us define a function,* $\Lambda_{ina} : ps \longrightarrow T_{ps}$*, from the initial pseudostate of* **SM** *to the set of transitions in* S_{ps}*, such that,*

$$\Lambda_{ina}(ps) = t_{ina}.$$

Definition 4.52. *The system* $\mathcal{LS}_{ps} = (S_{ps}, \psi_{ps}, \lambda_{ps})$ *for an initial pseudostate with* $S_{\mathrm{ps}} = \langle P_{\mathrm{ps}}, T_{\mathrm{ps}}, I_{\mathrm{ps}}, O_{\mathrm{ps}}, H_{\mathrm{ps}}, \Pi_{\mathrm{ps}}, W_{\mathrm{ps}}, M_{0\mathrm{ps}} \rangle$ *is defined as follows:*

$P_{ps} = \Psi_i(ps) \cup \Psi_c(ps) \cup \Psi_a(ps) \cup \Psi_{i'}(ps)$,
$T_{ps} = \Lambda_{ine}(ps) \cup \Lambda_{ina}(ps)$,

$$I_{ps}(t) = \begin{cases} \{p_i, p_c\} & \text{if } t = t_{ine} \\ p_a & \text{if } t = t_{ina} \end{cases}$$

$$O_{ps}(t) = \begin{cases} \{p_{cak}, p_a\} & \text{if } t = t_{ine} \\ p_{i'} & \text{if } t = t_{ina} \end{cases}$$

$$H_{ps}(t) = \emptyset, \Pi_{ps}(t) = 1, W_{ps}(t) = 1 : \forall t \in T_{ps}$$

$$\psi_{ps}(p) = \begin{cases} ini\_ps & \text{if } p = p_i \\ create & \text{if } p = p_c \\ ack\_create & \text{if } p = p_{cak} \\ ps\_accept\_create & \text{if } p = p_a \\ ini\_st & \text{if } p = p_{i'} \\ & \text{where:} \\ & st = name(state) \wedge \\ & \exists \mid t_{out} \in out(i) : target(t_{out}) = state \end{cases}$$

$$\lambda_{ps}(t) = \begin{cases} out & \text{if } t = t_{ine} \\ (e_1) & \text{if } t = t_{ina} \end{cases}$$

$$(e_1) = \begin{cases} name(a) \Longrightarrow \exists \mid t_{out} \in out(i) : effect(t_{out}) = a \\ \vee \\ \lambda \Longrightarrow \exists \mid t_{out} \in out(i) : effect(t_{out}) = \emptyset \end{cases}$$

4.9 Final states

In the UML metamodel, a final state is a special kind of state that can appear in a flat UML state machine meaning that the entire state machine has completed. The final state is graphycally represented by a bull eye and cannot have any outgoing transition.

In our interpretation we do not consider a final state as a processing state, as in the UML metamodel. For us, it is just a quiescent state meaning that the entire state machine has completed. Therefore, we do not allow entry actions, nor exit actions, nor activities, nor internal transitions nor deferred events. Moreover, we consider that it is an error in the UML metamodel to allow to process in the final states since it makes no sense to wait for internal transitions or deferred events after completion; on the other hand, entry actions, exit actions or activities can be modeled by means of other constructs without loosing modeling capabilities.

Although we allow that in a "flat" **SM** several final states can appear, from the above it can be assumed that actually there are not differences between them because they represent the same final quiescent state. The fact that a **SM** can be completed in different ways is represented by the transitions arriving to final states and not by the final states themselves, meaning each one a different way to complete it. Let us denote all the final states f by convention.

According to our interpretation, we propose to translate all the final states of a flat UML state machine in only one labelled system $\mathcal{LS}_f$ which is composed by only one place labelled ini_f. Figure 4.12(c) shows an example of the $\mathcal{LS}_f$ system. The $\mathcal{LS}_f$ system will be composed, using the operator defined in Chapter2 (Definition 2.6),

with the systems that model the simple states, then it will represent in the new net the finalization of the state machine. Section 4.10 shows how this composition is performed.

4.10 The model of a state machine

The labelled system $\mathcal{LS}_{sm}$that interpretates the a whole flat UML state machine (Definition 4.53 below) is obtained by composing the systems that interpretate the simple states ($\mathcal{LS}_i$ systems) the system for the initial pseudostate ($\mathcal{LS}_{ps}$) and the system for the final states ($\mathcal{LS}_f$).

Let Ev be the set of events produced/consumed by the flat UML state machine sm and Lev^P the set of labels of event and event acknowledge places, $Lev^P = \{e\_evx, \forall evx \in Ev\} \bigcup \{ack\_evx, \forall evx \in Ev\}$. Let $States$ be the set of the simple states of sm, and $Lstate^P$ the set of labels of places representing the entrance into states, $Lstate^P = \{ini\_target, \forall target \in States\}$, then the complete model for the i simple states that compose sm is:

$$\mathcal{LS}_{states} = \underset{Lev^P \cup Lstate^P}{\overset{i \in States}{||}} \mathcal{LS}_i$$

and the labelled system for the whole state machine is given in the following definition.

Definition 4.53. *The system $\mathcal{LS}_{sm} = (S_{sm}, \psi_{sm}, \lambda_{sm})$ that represents the flat UML state machine sm is defined as follows,*

$$\mathcal{LS}_{sm} = (\mathcal{LS}_{states} \underset{Lini^P}{||} \mathcal{LS}_{ps}) \underset{\{ini\_f\}}{||} \mathcal{LS}_f$$

where $Lini^P = \{ini\_state : state = name(st), st = target(t_{ini}), t_{ini} \in out(ps)\}$.

Figure 4.13 shows the $\mathcal{LS}_{sm}$ system that represents the interpretation of the of the flat UML state machine sm in Figure 4.1.

4.11 The model of a UML system

This section explains how to create an analysable model for a system assuming it is described as a set of flat UML state machines. By analysable model we mean a labelled system that includes the behaviour of the state machines that describe it on which we can compute logical properties and/or performance results: we have therefore to define how the labelled system components are composed, what is the initial marking and the performance indices.

We assume that the system is described for k state machines $\{sm_1, \ldots, sm_k\}$ which interact by exchanging synchronous and asynchronous messages through actions of the type *CallAction* and *SendAction*. Let $\{\mathcal{LS}_{sm_1}, ..., \mathcal{LS}_{sm_k}\}$ be the labelled systems of the k state machines produced according to the Definition 4.53.

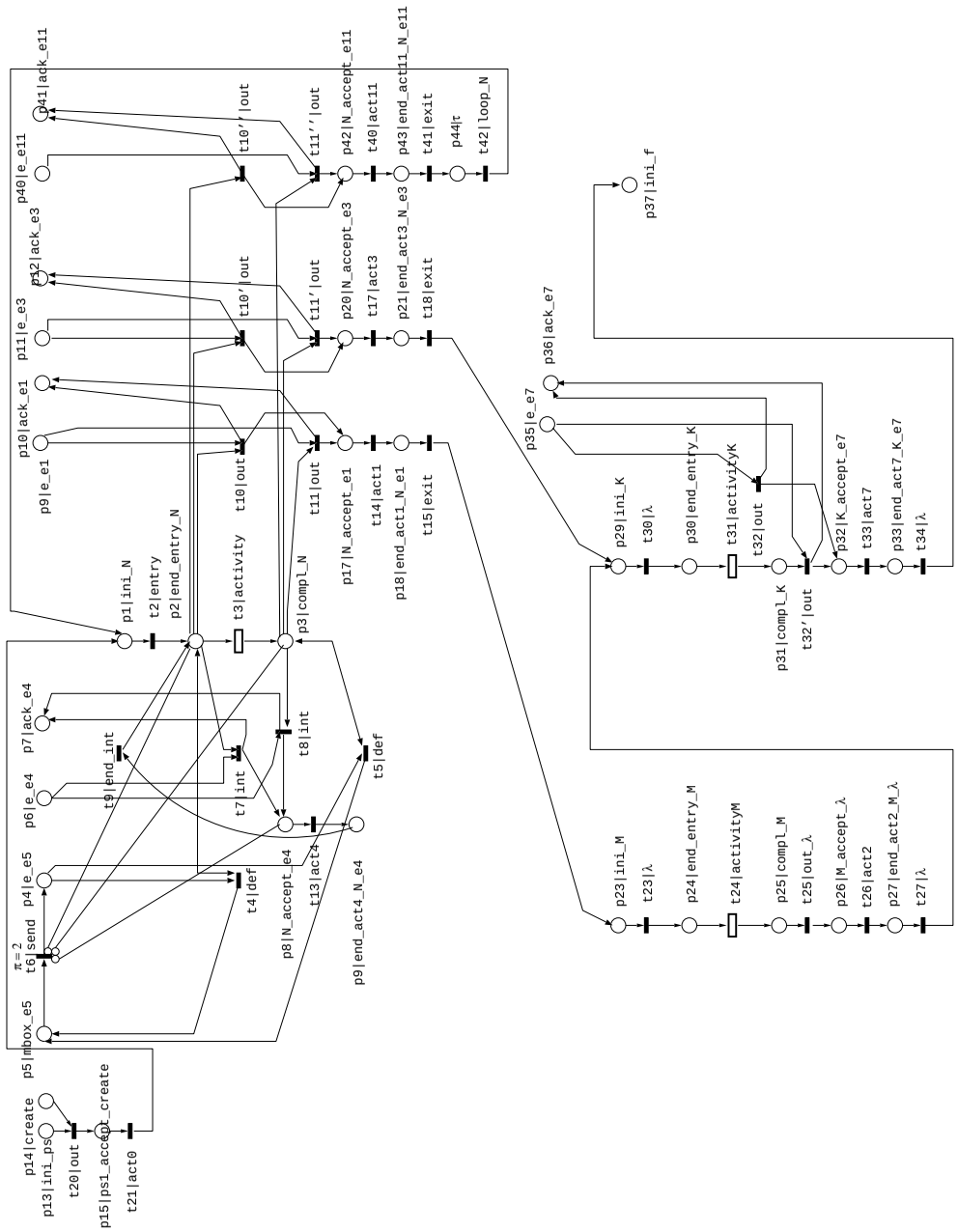


Figure 4.13: Labelled system of the state machine in Figure 4.1 (the internal transition $e14/act14$ and the deferred event e15 do not appear to gain readability).

A complete formal model for the system is obtained by superposition over event and event acknowledge places of the k state machines.

Let Ev_j be the set of events produced/consumed by sm_j and Lev_j^P the set of labels of event and event acknowledge places, $Lev_j^P = \{evx, \forall evx \in Ev_j\} \bigcup \{ack\_evx, \forall evx \in Ev_j\}$, and $Lev^P = \bigcup_{j\in\{1,\ldots,k\}} Lev_j^P$, then the complete model of the k state machines is given by the labelled system

$$\mathcal{LS}' = \overset{j=1,..,k}{\underset{Lev^P}{||}} \mathcal{LS}_{sm_j}$$

$\mathcal{LS}'$ 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 is given in the next definition.

Definition 4.54. *The system $\mathcal{LS}(S, \psi, \lambda)$ modelling k "flat" state machines is defined as follows:*

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

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

From the $\mathcal{LS}$ system a GSPN model that represent the UML system can be obtained as it will be explained in chapter 7 section 7.2.2.

4.12 Conclusions

In this chapter we have given a formal semantics in terms of labeled generalized stochastic Petri nets (LGSPNs) to a subset of UML elements that conform what we call "flat" state machines. In the following we briefly comment the translation given for each element.

The most relevant element formalized in this chapter has been the "simple state". For a simple state we have proposed a "basic" LGSPN system that interpretes its entry action and activity. This "basic" system also offers interface transitions to compose it with the Petri net systems for its deferred events, its internal transitions and its outgoing transitions.

The main characteristic of the deferred events system is that a *mbox_event* place stores tokens meaning event instances that are dispached to the corresponding *event* place when the "basic" system is exited. The internal transitions provoke the restart of the exponentially distributed firing time of the transition that represents the activity of the simple state.

The outgoing transitions and the exit actions of a simple state are translated into a LGSPN system that has a different configuration depending on the existence of self-loop transitions and/or immediate outgoing transitions.

A section is devoted to study how different LGSPN models are obtained from the different kind of actions: call actions, send actions and the rest of actions, taking into account if the call action is synchronous or not.

Translations for the initial pseudostates and the final states into LGSPN models are proposed. These ones composed with the models for the simple states give a LGSPN system for a entire "flat" UML state machine. Finally, an analysable model for a software system assuming it is described as a set of "flat" UML state machines is obtained by the superposition of the models of the state machines.

Chapter 5

UML Composite State Machines Compositional Semantics

In this chapter, we give formal semantics in terms of Labeled Generalized Stochastic Petri Nets (LGSPNs) [DF96] to a number of elements the UML state machines package [Obj01]. Concretely, those elements that were not previously discussed in chapter 4: Composite states, submachine states, history pseudostates, fork and join pseudostates, junction and choice pseudostates, synchronous states and stub states.

This formal semantics will be obtained in the same way that we proposed in chapter 4 for the elements of the "flat" state machines: by translating each element of the UML meta model, the input model, into a LGSPN model, the output model. As in the previous chapter the translation forces to perform an interpretation of the "non formally defined" UML concepts. Also, it will be studied how the LGSPN models obtained from the previous elements can be composed to achieve a LGSPN system representing a state machine. Moreover, it will be detailed how a LGSPN that represents a software system described by means of a number of a state machines can be obtained, obviously by composing the models of the individual state machines.

Therefore, the contribution of this chapter is to define the translation of a number of elements in the UML meta model into LGSPN models and to define the rules to compose them to obtain a LGSPN that represents a software system. The difference with respect to the previous chapter is that now we are not restricted to "flat" state machines.

5.1 Composite states

Composite states can be concurrent or not, in this section we study composite states without concurrency, the next section explores this quality of composite states. Now

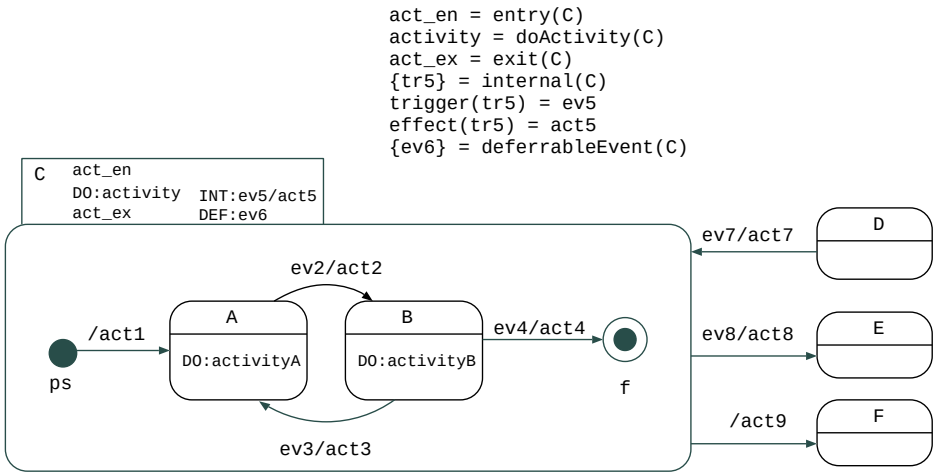


Figure 5.1: A composite state.

we succinctly recall how a composite state is defined in the UML meta model. "A composite state $cs \in CompositeState$ is a state that contains other state vertices. The association between the composite and the contained vertices is a composition association. Hence, a state vertex can be a part of at most one composite state. Any state enclosed within a composite state is called a *substate* of that composite state. It is called a *direct substate* when it is not contained by any other state; otherwise it is referred to as a *transitively nested substate*" [Obj01] section 2.12.

Furthermore, it is interesting to highlight from the UML Well-FormednessRules for the composite states that:

- A composite state can have at most one initial vertex,

  $$self.subvertex \rightarrow select(v \mid v.oclIsKindOf(Pseudostate)) \rightarrow select(p : Pseudostate \mid p.kind = \#initial) \rightarrow size \leq 1$$

- The substates of a composite state are part of only that composite state, $self.subvertex \rightarrow forAll(s \mid (s.container \rightarrow size = 1)$ **and** $(s.container = self))$

5.1.1 Informal interpretation

In this section our interpretation of a non concurrent composite state is given. Forces, compositionality [US94] is a challenge in the sense that we try to interpret the concepts of the composite states as homogeneously as possible with the interpretation of the simple states given in chapter 4, with the goal of obtaining compositional models.

UML allows three ways to enter in a composite state: explicitly, using a history pseudostate, or by default. An explicit entry is graphically indicated by an incoming

transition that goes to a direct substate of the composite state or to a transitively nested substate. In our interpretation this way to enter a composite state should be forbidden because it provokes crossing boundaries and hence models without compositional properties as it is discussed in [Sim00]. Later in this section we come back to this discussion. The entry in a composite state by means of a history (shallow or deep) vertex will be discussed in section 5.4 where the reasons that motivate our rejection to this kind of entry are given. Last, the default entry in a composite state is graphically indicated by an incoming transition that terminates on the outside edge of the composite state, see transition $ev7/act7$ in Figure 5.1 as an example. In this case the default transition (the transition exiting from the initial pseudostate) is taken and if there is a guard on this transition it must be enabled (true). A disable default transition is an ill-defined execution state and its handling is not defined.

Concerning the termination of a composite state, UML allows either explicit or default exit. Explicit exit is represented by a transition that, outgoing one of its substates (direct or nested), crosses the boundaries of the composite state to reach an outside state. Obviously this kind of exiting is forbidden in our interpretation for the same reason as the explicit entry: the drawback over the compositionality. It will be discussed later in this section. Default exit is represented by any transition that arrives to a final state inside the composite state at the top most level, see for an example Figure 5.1 where transition $ev4/act4$ arrives to final state f. When this kind of exit occurs it is mandatory, in our interpretation, that the composite state has an immediate outgoing transition, that will be taken after the final state is reached. See in Figure 5.1 the transition $/ev9$ outgoing the state C which will be taken after the state f is reached. Later in this section, this interpretation is discussed in detail. Finally, it must be noticed that the exit is possible from a substate using an outgoing transition of the composite state since they are inherited, this interpretation will be explained in depth in this section.

In the following, we are going to describe the behaviour of the composite state C in Figure 5.1. It will allow us to introduce our interpretation for the composite states. The interpretation is "informally" explained in this section and its formalization given in the next section. The composite state C can be entered by default by the transition $ev7/ac7$. On entering, the entry action act_en is executed and, when it completes, the activity starts its execution. In the meantime, the deferred events and the internal transitions belonging to the composite state, $ev6$ and $ev5/act5$ in the example, can be accepted causing the finalization of the activity. If a deferred event or an internal transition is accepted, it performs its execution model as explained in chapter 4. When the activity has completed, the default transition $/act1$ is taken. The execution of the "flat" state machine enclosed in the composite state is interpreted as given in chapter 4. But it must be taken into account that each substate apart from its deferred events, internal transitions and outgoing transitions inherits those of its container states in a transitively fashion. If the enclosed state machine is not "flat", i.e., it contains same composite state, then these states are interpreted as we are explaining in this paragraph, therefore providing a recursive interpretation. On completion of the enclosed state machine, i.e., the arrival to the final state f, the composite state

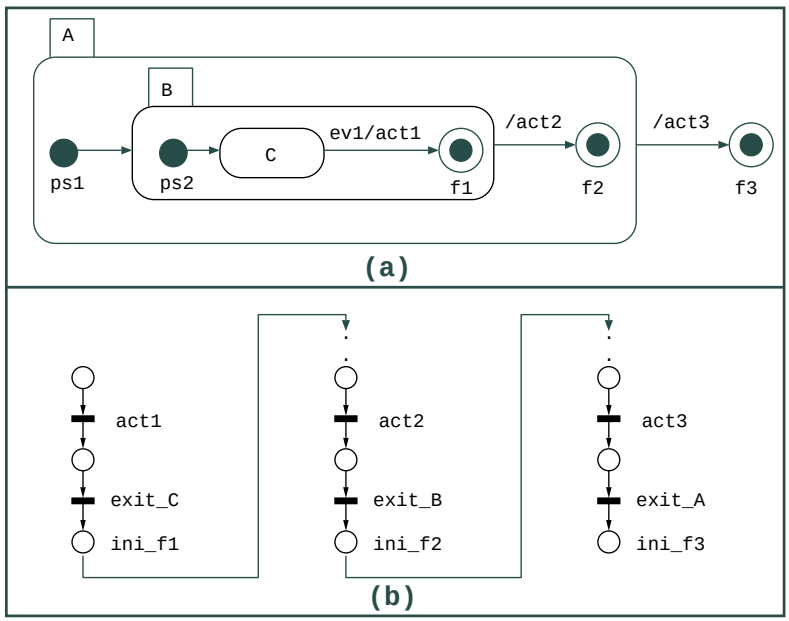


Figure 5.2: Default exit for composite states A and B.

must take its immediate outgoing transition $/act9$, then $act9$ is executed and after the exit action, act_ex, of the composite state C is executed. When the state A or B is active, transition $ev8/act8$ can be taken on arrival of event $ev8$, in this case the execution sequence is as follows: $act8$, exit activity of A or B and the exit of the composite state.

From the explanations given above to describe the behavior of the composite state C in Figure 5.1 it is obvious that our interpretation of a composite state differs from that of UML in a number of points. The restrictions that we impose are convenient to achieve compositional semantics and some of them come from [Sim00] while others differ considerably. In the following we give "informally" our interpretation of a composite state:

1. Transitions crossing boundaries of composite states are not allowed in any clean compositional model. This restriction is largely discussed in [Sim00] and we assume it in order to ensure encapsulation in hierarchical state machines. It must be noticed that by forbidding crossing boundaries then the explicit entry and the explicit exit in a composite state are not possible. We recognize that by doing so, it is not possible to properly represent more than one distinct accept states to indicate distinct outcomes in a composite state [SG99]. Nevertheless, we assume this drawback in our interpretation to guarantee compositional models.

2. In our interpretation, a composite state must have exactly one initial vertex. This fact differs from the UML standard since it allows at most one initial vertex. By forcing the existence of an initial state the default entry is guaranteed (remember that it is the only one that we allow) and by forcing the existence of only one the indeterminism upon entrance is avoided. The transition outgoing from an initial vertex, that must be unique since we do not allow guards, does not have trigger. Remember that in the UML model this kind of transitions can be triggered only by an event stereotyped *create* and obviously the object is created by the transition exiting from the initial state at the top most level. This transition will be taken always upon entry in the composite state independently of which incoming transition has fired the entry in the composite state. Its related action will be performed after the execution of the entry action and the activity of the composite state. Concerning the activity of the composite state, it can be aborted by any of the outgoing transitions of the composite state, the same as for the simple states.

3. Concerning final states, in our interpretation a composite state can have zero or more final states as in the UML standard. The discussion presented in chapter 4 when our interpretation of final states in the context of "flat" state machines was introduced is completely valid in the context of composite states and it differs from that given in UML. We recall the main conclusions but applied to this context:

   – a final state in a composite state means that the entire state has completed;
   – a final state is not considered as a processing state;
   – although we allow several final states inside a composite state, actually there are not differences between them because they represent the same final quiescent state. Therefore they will be represented by the same LGSPN model.

4. For us, like in the UML standard, states inside a composite state inherit outgoing transitions from its composite state in a transitively manner, except immediate outgoing transitions and those transitions fired by an event that also fires any outgoing transition, internal transition or deferred event in the substate. By doing so the following interpretations are gained:

   – It is ensured that a composite state behaves as a reactive state instead as a lock-in process [Sim00]. Note that if the substates of a composite state react only to the transitions exiting its own boundaries then the enclosed state machine completes only when it arrives to any of its final states showing a lock-in process behaviour. Reactive behavior is highly desirable because we have interpreted simple states in chapter 4 also as reactive states, therefore the same behavior is expected for every kind of states.
   – Moreover, when the enclosed state machine is interpreted as the description of the activity performed by the composite state then the possibility

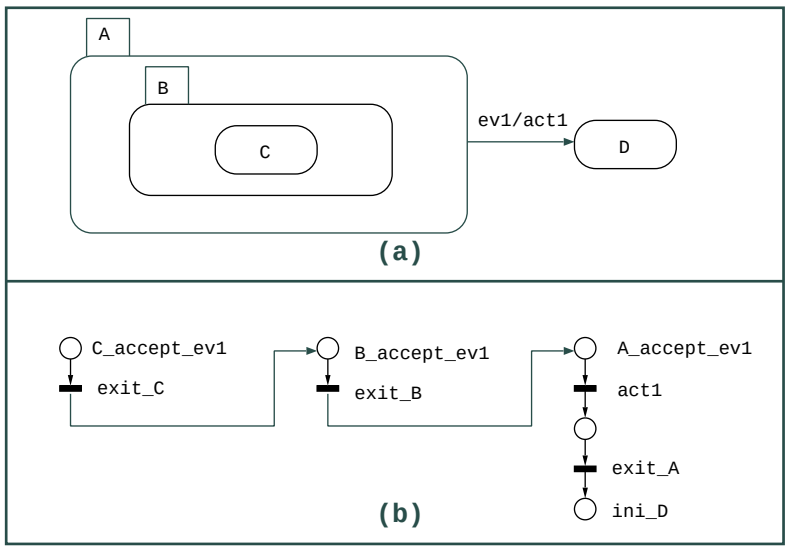


Figure 5.3: Exit of the composite state A from the substate C.

to accept exits from the boundaries means that this activity could be interrupted by the outgoing transitions of its composite state, which also is compliant with the interpretation given in chapter 4 for the activities inside simple states. Note that different interpretations can be associated with the composite state.

- "Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty" [Obj01]. Avoiding the inheritance of transitions triggered by an event that triggers a deferred event or an internal or outgoing transition is important because it implies that "conflicting transitions" cannot occur in our models.
- The fact that immediate outgoing transitions are not inherited is a drawback for our interpretation, since the rest of the transitions are inherited. Furthermore, it unables the reactive behavior of the composite state in some manner [Sim00] because the enclosed substates do not react to the free transition. But if this kind of inheritance were allowed then it would not be possible the interpretation given previously in point 3 (where we associated immediate outgoing transitions of composite states to the default exit). So, we recognize an asymmetry in the interpretation of the inheritance of the outgoing transitions.

5. The exit of a composite state in our interpretation can be either by default or by firing any of the transitions in its boundaries. In the following, both possibilities

are discussed:

- The existence of a final state in a composite state means that default exit is possible and it implies that the composite state must have an immediate outgoing transition. This transition will be immediately taken after the composite state reaches one of its final states. The execution model that we propose is as follows: after the final state is reached the action of the immediate outgoing transition is performed, if it exists, and after the exit action of the composite state, also if it exists. Notice that this interpretation of the immediate outgoing transitions in the boundaries of a composite state differs from that given by us in chapter 4 for immediate outgoing transitions in the boundaries of a simple state, but actually it is not desirable since an homogeneous interpretation would be nice. But if we interpret that this transition is taken immediately after the activity has complete then the enclosed state machine is never executed, which makes no sense.

  For an example in which several default exits are nested see Figure 5.2 (a). In Figure 5.2 (b) the part of the LGSPN model that represents the exits is shown. In it can be observed that the sequence transition action and exit action is successively performed from the innermost state to the outermost.

- On the other hand, as the outgoing transitions are inherited by the substates, it is possible to exit a composite state from any of its substates. The example in Figure 5.3 is going to be used to explain our interpretation of execution model for this kind of exit, for simplicity the example shows just the exit. Each exit action is executed innermost to outermost until the top most state is reached, where first the action of the transition is executed and after the exit action of the state. Although in the UML interpretation this point is not very clear, we deduce that first the action of the transition is executed and after every exit action (innermost to outermost) until the top most state is reached, it obviously differs from our interpretation.

6. Finally, it must be noticed that in our interpretation each substate inherits the internal transitions as well as the deferred events from its composite state in a transitively manner and as it has been explained for the outgoing transitions in the point 4 of this relation. Moreover, they are interpreted as it was explained in chapter 4 for the simple states.

The LGSPN in Figure 5.4 represents the state machine in Figure 5.1 upon our interpretation. It has been obtained by applying the formalization given in chapter 4 for a "flat" state machine with slight differences that are formalized in the next section. This differences are enumerated in the following:

- To represent the default entry, the following decisions have been taken: it has been added the arc from transition $t2|activity$ to place $p12|ini\_ps$. Notice that

the place labelled $compl\_C$ has been removed and also the arc from it to the transition labeled out_{ce}. It interprets the point number 3 in the above enumeration list, in this way when the activity of the composite state finishes the immediate outgoing transition is not taken but the default transition.

- To represent the default exit, an arc from place $p33|ini\_f$ to transition $t10|out_{ce}$ has been added. It interprets the first part in point number 5 of the above enumeration list, therefore it makes possible to take the immediate outgoing transition after the final state has been reached.

- To represent the exit from a substate (say A) taking an outgoing transition of the composite state (say $ev8/act8$), the following decisions have been taken: the transition that represents the action $act8$ and the place that represents its finalization $end\_act8\_A\_ev8$ are removed from the net that represents the enclosed state (say A). Finally, an arc is added from the transition that represents the exit action ($t27|\lambda$) to the place that represents the acceptation of the event in its container state ($p6|C\_accept\_ev8$). It interprets the second part in the point number 5 of the above enumeration list.

5.1.2 Formal translation

For the sake of clarity, the formal translation is divided in two parts. Firstly, the case of a composite state that encloses a "flat" state machine is formalized (cfr. Definition 5.1). In the second part the previous definition is used to recursively define the general case, i.e a composite state that encloses either a "flat" state machine or a state machine with composite states (cfr. Definition 5.2).

Part I

To obtain the LGSPN system $\mathcal{LS}_C^{flat} = (S_C^{flat}, \psi_C^{flat}, \lambda_C^{flat})$ that represents a composite state C that encloses a "flat" state machine, the following facts must be taken into account:

- The system for the initial state in C, $\mathcal{LS}_{ps}$, is obtained as given in Definition 4.52.

- The system for the final states in C, $\mathcal{LS}_f$, is obtained as given in section 4.9. Moreover, a transition labeled out_{ce} and an arc from the place labeled ini_f to this transition must be added.

- Each simple state $s \in subvertex(C)$ inherits all the deferred events, internal transitions and outgoing transitions from C, even those that have been inherited by C, except those (deferred, internal or outgoing) whose trigger event is present in its own list of deferred, internal or outgoing. Formally,

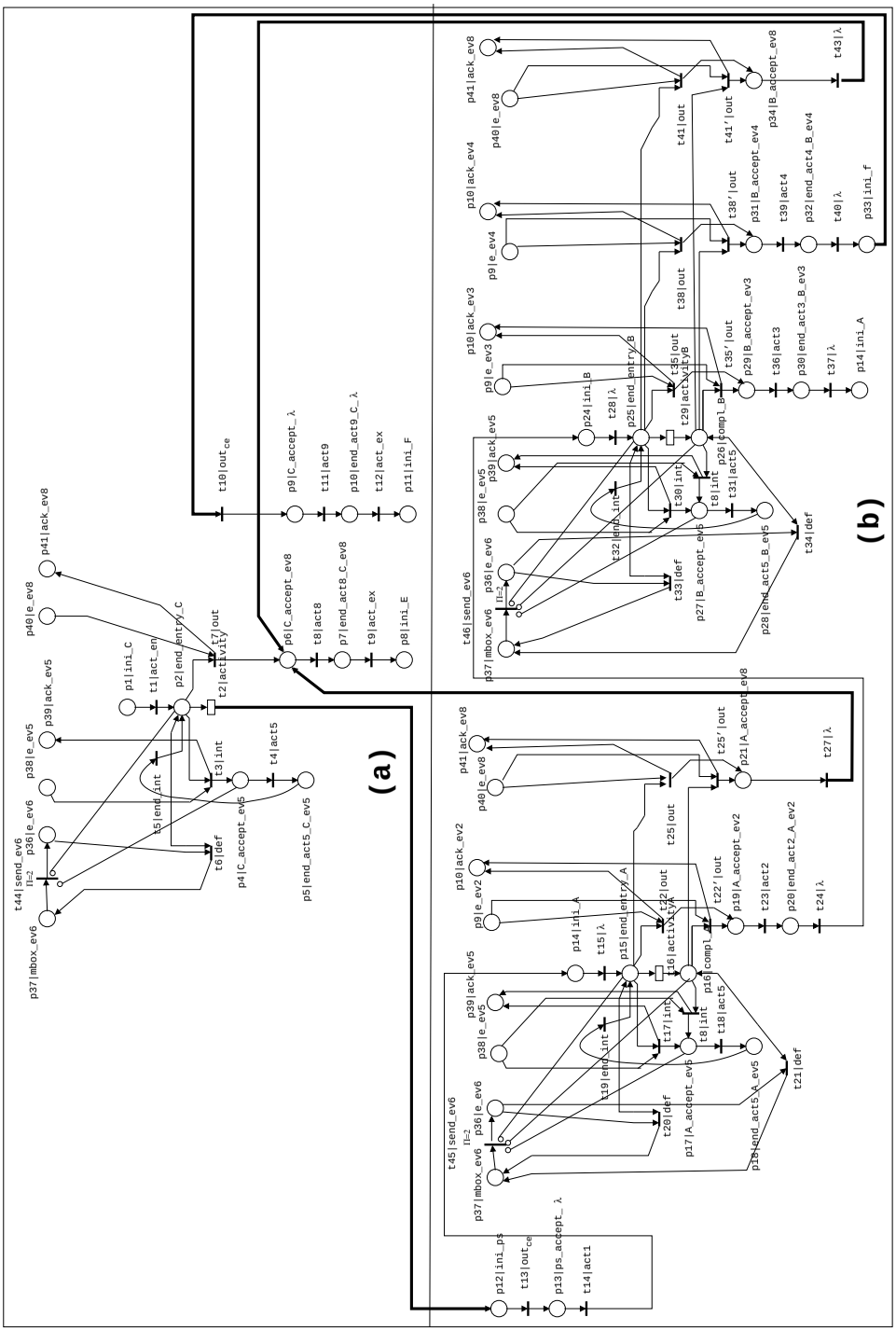


Figure 5.4: Labeled system of the composite state C in Figure 5.1.

$$out(s) = \{t \in Transition \mid (t \in outgoing(s)) \vee (t \in (out(C) - \{t' \mid trigger(t') \in Triggers(s)\})\}$$
$$int(s) = \{t \in Transition \mid (t \in internal(s)) \vee (t \in (int(C) - \{t' \mid trigger(t') \in Triggers(s)\})\}$$
$$def(s) = \{e \in Event \mid (e \in deferrableEvent(s)) \vee (e \in (def(C) - \{e' \mid e' \in Triggers(s)\})\}$$
$$Triggers(s) = \{e \in Event \mid (e \in deferrableEvent(s)) \vee (\exists t \in (outgoing(s) \cup internal(s)) \ : \ trigger(t) = e)$$

By doing so, it is guaranteed that when two events (that trigger outgoing transitions, internal transitions or deferred events) are in conflict then the priority policy decides for the innermost one.

$\mathcal{L}S_s$ system is obtained as given in Definition 4.45. But taking into account that in $\mathcal{L}S_s^g$ the transitions that represent an action of an inherited transition (say $act8$ in Figure 5.4) and the place that represents its acceptation (say A_accept_ev8) must be removed and a new place representing the acceptation in the composite state (say C_accept_ev8) and a new arc to it from the place of the exit action must be added.

- Once defined the systems for the initial pseudostate, the final states and the simple states, then a labeled system $\mathcal{L}S'_C$ that represents the enclosed "flat" state machine can be obtained by applying Definition 4.53. Figure 5.4(b) shows $\mathcal{L}S'_C$ for the enclosed state machine of state C in Figure 5.1.

- To represent the composite state itself (with its deferred events, internal transitions and outgoing transitions) a labeled system $\mathcal{L}S''_C$ is obtained by applying Definition 4.45 to the composite state supposing that it has an immediate outgoing transition (even in the case it is not true). Moreover, the place labelled $compl\_C$ must be renamed to ini_ps being ps the name of the initial pseudostate in C, and removing all the arcs exiting from this place. Figure 5.4(a) shows $\mathcal{L}S''_C$ for the enclosed state machine of state C in Figure 5.1.

Definition 5.1. *The system $\mathcal{LS}_C^{flat} = (S_C^{flat}, \psi_C^{flat}, \lambda_C^{flat})$ that represents the composite state C which encloses a "flat" state machine is defined as follows,*

$$\mathcal{LS}_C^{flat} = \mathcal{L}S'_C \mid\mid_L \mathcal{L}S''_C$$

$$L = \{ini\_ps\} \cup \{out_{ce}\} \cup Laccept$$

$$Laccept = \{C\_accept\_evx, \forall evx \in out(C)\}$$

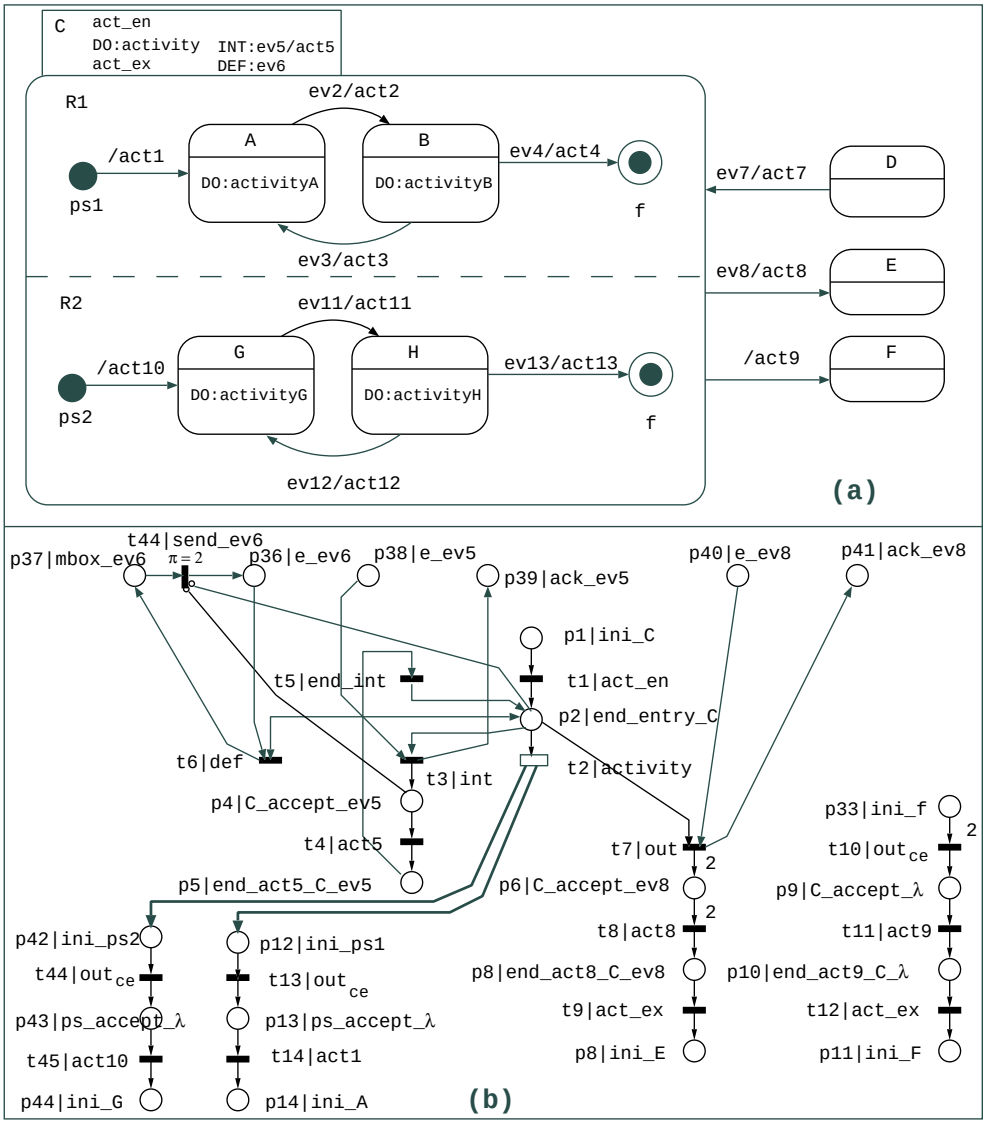


Figure 5.5: (a) A concurrent composite state. (b) $\mathcal{LS}''_C$ (a part of the labelled system for the concurrent composite state C).

Part II

The labeled system, $\mathcal{LS}_C$, that interpretates a composite state C, cfr. Definition 5.2, is the labeled system defined in 5.1 if C has a "flat" state machine enclosed or the labeled system defined in Definition 5.2 composed with the labeled systems for the composite states in $CompositeStates_C$ otherwise.

Let Ev be the set of events produced/consumed by the composite state C and Lev^P the set of labels of event and event acknowledge places, $Lev^P = \{e\_evx, \forall evx \in Ev\} \bigcup \{ack\_evx, \forall evx \in Ev\}$.

Let be $States_C = \{s \in State, \forall s\ container(s) = C\}$ and let $CompositeStates_C = \{s \in CompositeState, \forall s\ container(s) = C\}$, obviously $CompositeStates_C \subset States_C$.

Let $Lstate^P$ be the set of labels of places representing the entrance into the states in $States_C$, $Lstate^P = \{ini\_target, \forall target \in States_C\}$, and let $Lstcom^P$ be the set of labels of places representing the entrance into the composite states in $CompositeStates_C$, $Lstcom^P = \{ini\_target, \forall target \in CompositeStates_C\}$.

Definition 5.2. *The system $\mathcal{LS}_C = (S_C, \psi_C, \lambda_C)$ that represents the composite state C is defined as follows,*

$$\mathcal{LS}_C = \begin{cases} \mathcal{LS}_C^{flat} \quad \textit{(Def. 5.1)} & \text{if } C \text{ has a "flat" state machine enclosed} \\ \mathcal{LS}_C^{flat} \underset{Lev^P \cup Lstate^P}{||} \mathcal{LS}^{Compo} & \text{otherwise} \end{cases}$$

where

$$\mathcal{LS}^{Compo} = \underset{Lev^P \cup Lstcom^P}{\overset{j \in CompositeState_C}{||}} \mathcal{LS}_j$$

When the composite state has not a "flat" state machine enclosed, therefore it has composite states enclosed, then we understand by $\mathcal{LS}_C^{flat}$ a labeled system as in Definition 5.1 which does not take into account the composite states.

Figure 5.4 shows the $\mathcal{LS}_C$ system that represents the interpretation of the composite state C in Figure 5.1 which encloses a "flat" state machine.

5.2 Concurrent states

"Composite states in UML have a derived boolean attribute that indicates whether it is a substate of a concurrent state. If it is true then this composite state is a direct substate of a concurrent state" (section 2.12 in [Obj01]).

It is interesting to emphasize the following two properties of the concurrent states from the Well-FormednessRules:

- There have to be at least two composite substates in a concurrent composite state,

$(self.isConcurrent)$ **implies**
$(self.subvertex \rightarrow select$
$(v \mid v.oclIsKindOf(CompositeState)) \rightarrow size \geq 2)$

- A concurrent state can only have composite states as substates,

$(self.isConcurrent)$ **implies**
$(self.subvertex \rightarrow forAll(s \mid s.oclIsKindOf(CompositeState)))$

5.2.1 Informal interpretation

In the following, we are going to describe the behavior of the concurrent composite state C in Figure 5.5(a). It will allow us to introduce our interpretation for the concurrent composite states. In this subsection we give our "informal" interpretation of the concurrent composite states. As in the previous section first we study an example, after that the interpretation is "informally" explained and finally its formalization is given in the next subsection. The concurrent composite state C in Figure 5.5(a) can be entered only by default through the transition $ev7/ac7$. On entering, the entry action act_en is executed and when it finishes, the activity starts its execution. In the meantime, the deferred events and the internal transitions belonging to the composite state, $ev6$ and $ev5/act5$ in the example, can be accepted causing the finalization of the activity. If a deferred event or an internal transition is accepted it performs its execution model as explained in chapter 4. The deferred events, internal transitions and outgoing transitions are inherited by each substate. So far the interpretation is the same as for non concurrent composite states. But when the activity has completed, the two regions $R1, R2$ must be entered, region $R1$ taking its default transition $/act1$ and region $R2$ its own one $/act10$, from now onwards each region is executed concurrently. The execution of each region continues like a non concurrent composite state, as explained in section 5.1.1, until completion. On completion of one of the regions, say $R1$, i.e. the arrival to the final state, it must wait until completion of the other region, $R2$. It means that when a region has reached a final state it is mandatory for the concurrent state to complete using default exit, therefore inherited transitions cannot be taken. When $R1$ and $R2$ have reached their final states, the concurrent composite state must take the immediate outgoing transition $/act9$, then $act9$ is executed and after the exit of the composite state C, act_ex. The other possibility to exit C is to take transition $ev8/act8$ when the state is in any of the following configurations $A-G$, $A-H$, $B-G$ or $B-H$. In this case the execution sequence is as follows in our interpretation: the exit activity of each enclosed state in the active configuration is performed, possibly concurrently, after that the action $act8$ is executed and finally the exit of the concurrent composite state.

As in the case of non-concurrent composite states, our interpretation differs from that of UML in a number of points. In the following we give "informally" our interpretation of concurrent composite state by focusing in its own aspects and assuming that the interpretation given in section 5.1.1 for the composite states is valid for each region of the concurrent state:

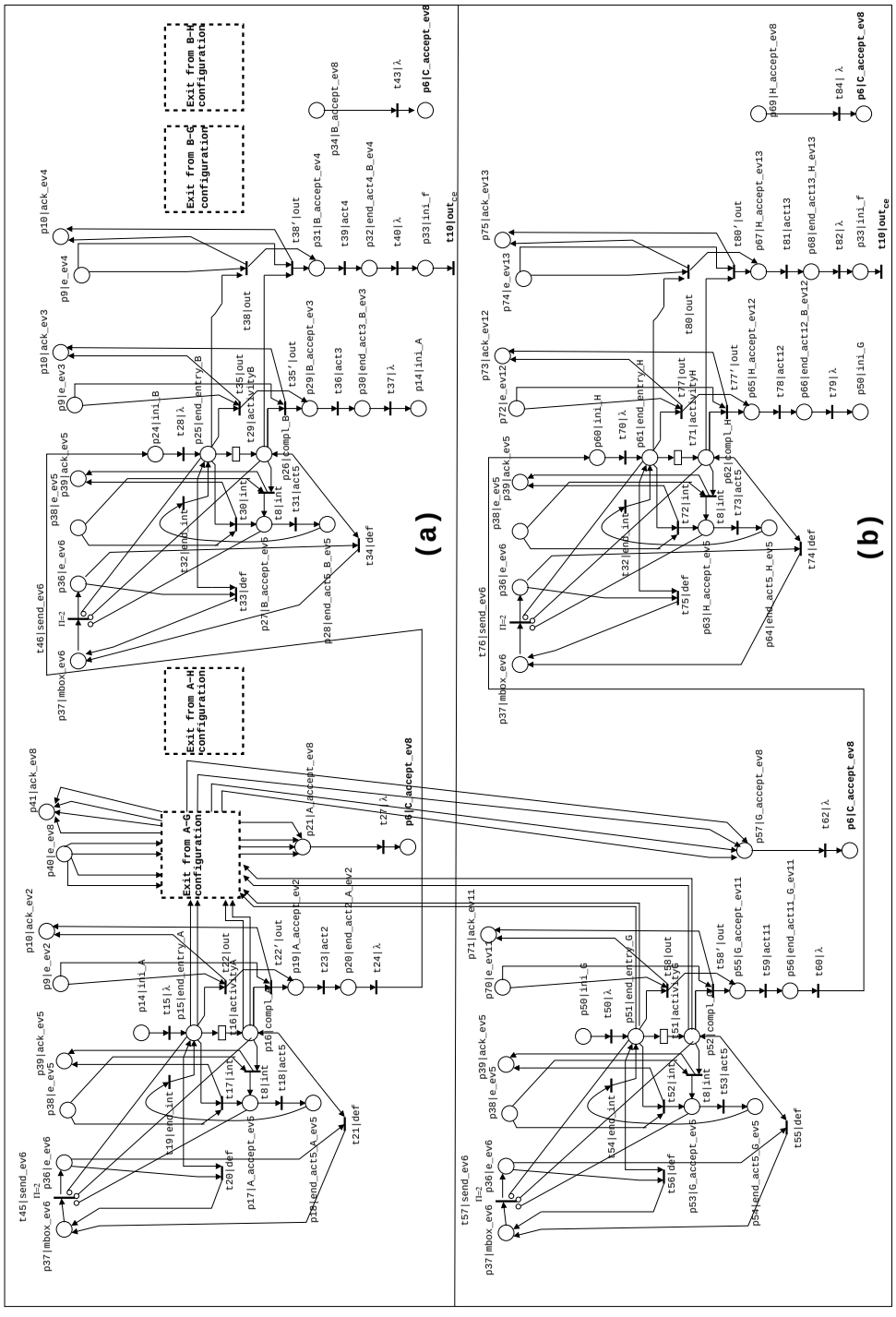


Figure 5.6: Partial labeled system of the concurrent composite state C in Figure 5.5.

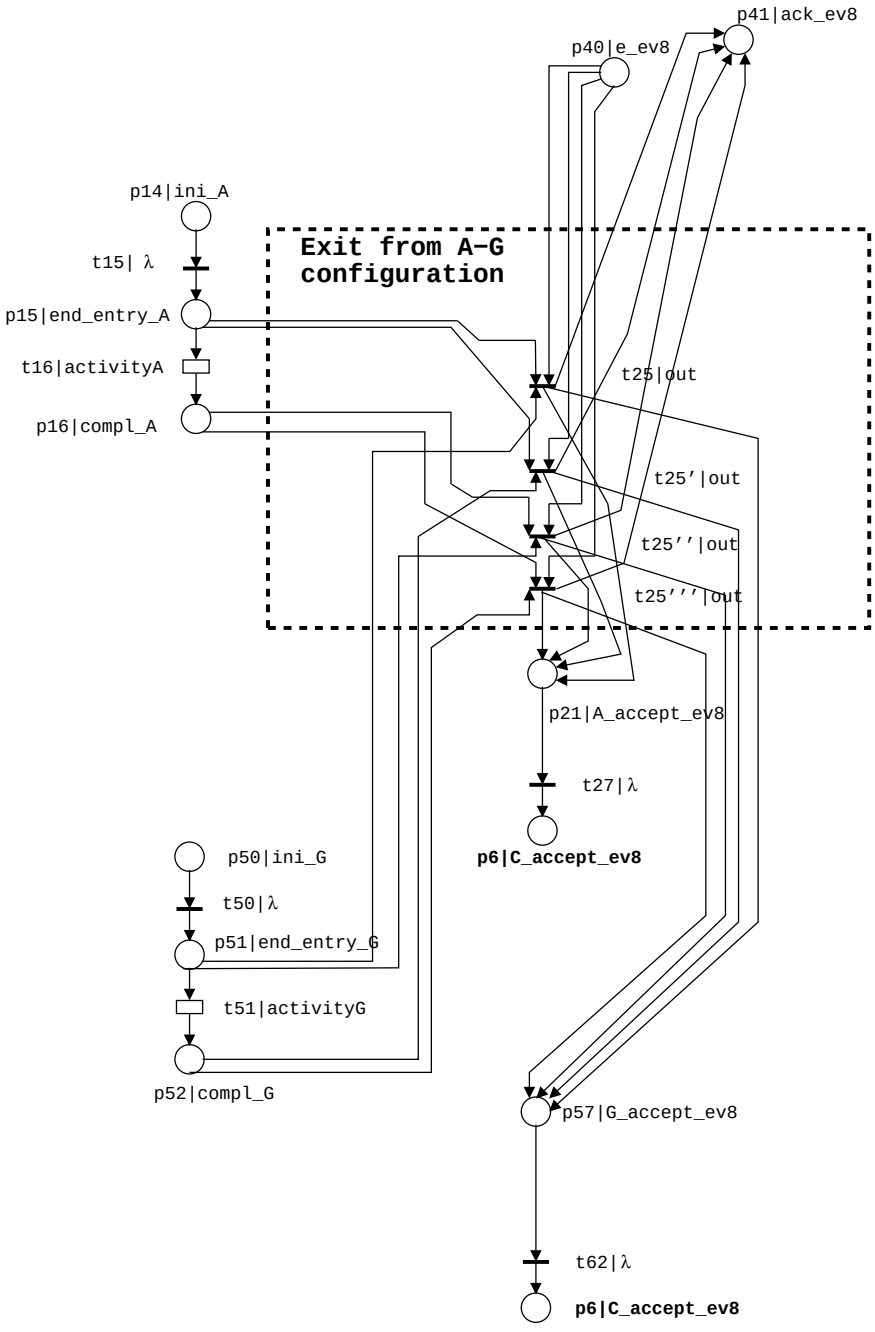


Figure 5.7: Exiting from configuration A-G of the concurrent composite state C in Figure 5.5.

1. In UML the regions that conform a concurrent state are considered each one a composite state. This means that a region can have its own entry, exit, do Activity, deferred events and internal transitions. For us this modeling power is outlandish and it can be achieved by using the existing concepts. Therefore for the sake of simplicity, in our interpretation regions are not proper composite states. They are just a conventional notation to refer sets of orthogonal states that can be concurrently executed in a composite state, but without additional semantics. Implications of this decision are clear in the execution model of a concurrent state. The entry and exit of a concurrent composite state do not take care about new actions nor activities. Moreover, upon this interpretation the previous well-formed rules should be rewritten to:
   - There have to be at least two regions in a concurrent composite state.
   - A concurrent state can only have regions as substates.
2. "Whenever a concurrent composite state is entered, each one of its regions is also entered, either by default or explicitly. If the transition terminates on the edge of the composite state, then all the regions are entered using default entry. If the transition explicitly enters one or more regions (in case of fork), these regions are entered explicitly and the others by default" [Obj01]. In order to be congruent with the interpretation given in section 5.1.1, concurrent composite states can be entered only by default, i.e. entering transitions must terminate on the edge of the composite state. Once the composite state has been entered, the default entry of each region is taken.
3. "When exiting from a concurrent state, each of its regions is exited. After that the exit actions of the regions are executed" [Obj01]. Obviously, in our interpretation exit actions for regions are not executed. Depending on the kind of exit the following execution models are performed in our interpretation:
   - Upon exit taken an inherited transition, the actions of the substates in the active state configuration are executed, after the action of the transition and finally the exit action of the concurrent composite state. Exiting taken an inherited transition is possible only in an stable state configuration, the possible state configurations are the Cartesian product of the states in the orthogonal regions ($A-G$ $A-H$ $B-G$ or $B-H$ in the example in Figure 5.5). In Figure 5.6 the four possible exits are abstracted in the four dotted boxes, for clarity only in the first box, the entries and the exits have been represented arriving to the box. Figure 5.7 shows the details of the exit of this configuration, which is the same for all the possible exits with inherited transitions. The exit is represented by four transitions in the LGSPN ($t25, t25', t25''$ and $t25'''$) representing each one an stable configuration.
   - Upon default exit, the action of the transition is executed and after that, the exit action of the concurrent composite state. In our interpretation if a

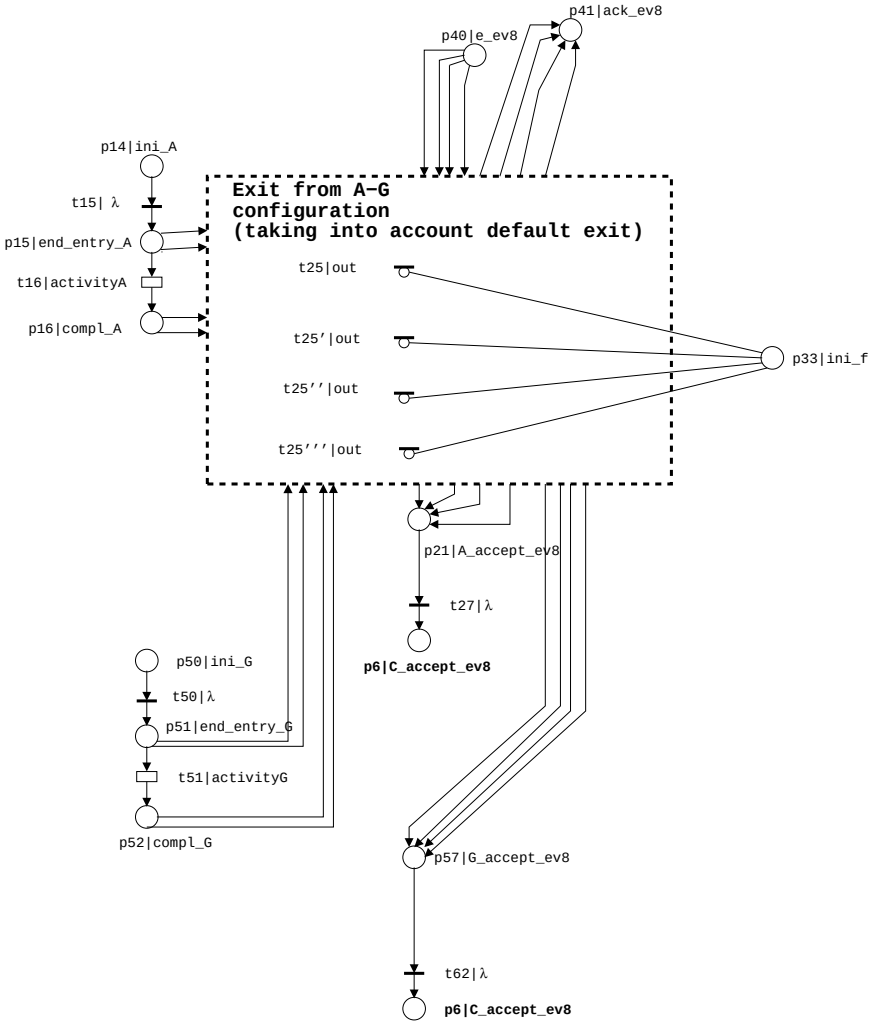


Figure 5.8: Default exit of the concurrent composite state C in Figure 5.5.

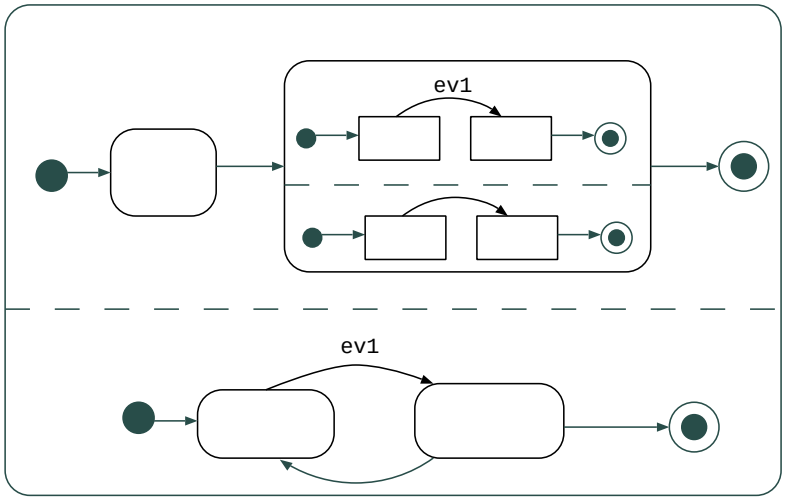


Figure 5.9: A concurrent composite state with conflicting transitions.

region reaches its final state it implies that the concurrent composite state will be exited by default, then it is expected that the rest of the regions reach their own final states. When all the regions have reached their final state the immediate outgoing transition must be taken. This implies that when any region has reached its final state, no of the other regions can accept inherited outgoing transitions. This behavior is achieved in our LGSPN model by inhibiting the inherited transitions by the place that represents the final states, see Figure 5.8.

4. "Only transitions that occur in mutually orthogonal regions may be fired simultaneously" [Obj01]. An example of this kind of transitions is represented in Figure 5.9 by transition $ev1$. In our interpretation only one of these transitions can fire. It is motivated because an event is represented by a token which is used to fire only one of the two conflicting transitions with the same probability.

   Summarizing, the firing policy in our interpretation is as follows:

   – Conflicts among transitions at different levels in the state hierarchy are not possible since we do not allow to inherit confliting transitions, see point 4 in section 5.1.1. Therefore our policy consists in the selection of the innermost transition.

   – The conflict among transitions in orthogonal regions is probabilistically resolved (all transitions have the same probability).

The LGSPN in Figures 5.5(b) and 5.6 represents the concurrent composite state in Figure 5.5(a) upon our interpretation. It has been obtained by applying the for-

malization given in section 5.1.2 for the composite states with slight differences that are formalized in the next section. These differences are enumerated in the following:

- In order to represent the default entry, arcs must be added from transition $t2|activity$ to each place that represents the enter in a region, places $p42|ini\_ps2$ and $p12|ini\_ps1$ in the example.

- For the default exit, the arc from the place that represents the final state of the regions to the transition that represents the completion of the composite state is labelled with the number of concurrent regions. In the example, the arc from $p33|ini\_f$ to the transition $t10|out_{ce}$. Therefore, there are as many tokens as regions to fire the default transition, which means that the state is not exited until all regions complete.

- In order to represent the exit taking a transition on the boundaries of the concurrent composite state, the following decisions have been taken: the arcs from $t7$ to $p6$ and from $p6$ to $t8$ are labelled with the number of concurrent regions, which means that the state is not exited until all regions complete; for each substate, the transitions in the set $out(substate)$ (see section 5.1.2) that represent inherited transitions are removed and substituted by the transitions represented in the box of the Figure 5.7, these transitions are inhibited by the place that represents the final state, as can be seen in Figure 5.8.

5.2.2 Formal translation (sketch)

In order to translate a concurrent composite state C into a LGSPN, Definition 5.2 is used with the following modifications:

- In the system $\mathcal{LS}''_C$ as many places labeled ini_ps must be created (where ps is the name of the initial state in each region) as concurrent regions.

- Also in the system $\mathcal{LS}''_C$ the arc entering the transition out_ce must be labelled with the number of regions. The arcs entering and exiting places that represent the acceptation of outgoing transitions are also labelled with the number of regions.

- In the systems $\mathcal{LS}_s$ (system for each substate) the out transitions that represent inherited transitions are removed. For each stable configuration (cartesian product of the states in orthogonal regions) a system as that presented in Figure 5.7 must be created, taking into account that each transition has an inhibitor arc from the place that represent the final state.

Taking into account the modifications introduced for $\mathcal{LS}''_C$ and $\mathcal{LS}_s$ systems, Definition 5.2 gives the LGSPN model for a concurrent composite state C.

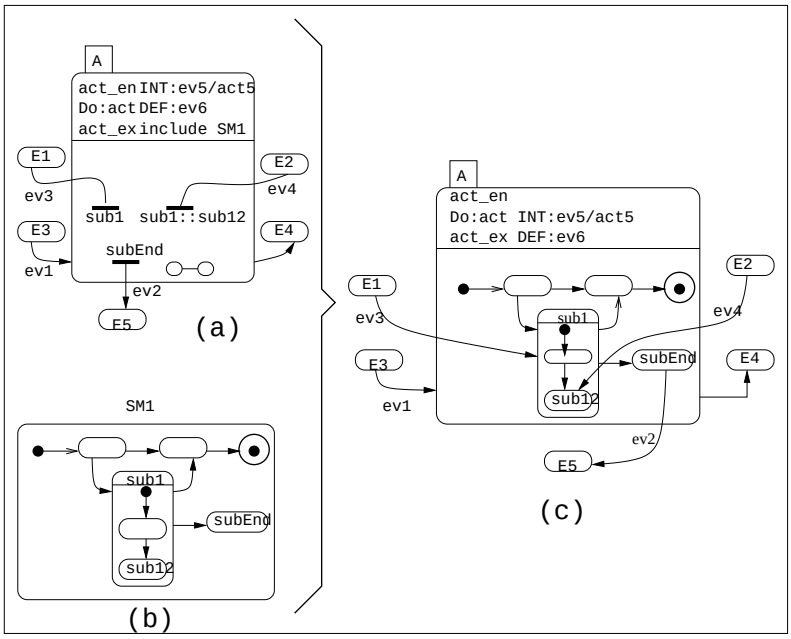


Figure 5.10: (a)A submachine state. (b)The referenced state machine. (c)The state machine inside the composite state.

5.3 Submachine states and stub states

"A submachine state is a syntactical convenience that facilitates reuse and modularity. It is a shorthand that implies a macro-like expansion by another state machine and is semantically equivalent to a composite state. The state machine that is inserted is called the *referenced* state machine while the state machine that contains the submachine state is called the *containing* state machine. The same state machine may be referenced more than once in the context of a single containing state machine. In effect, a submachine state represents a "call" to a state machine "subroutine" with one or more entry and exit points. The entry and exit points are specified by stub state. It may have entry and exit actions, internal transitionss, and activities" (taken from section 2.12 in [Obj01]).

"A stub state can appear within a submachine state and represents an actual subvertex contained within the referenced state machine. It can serve as a source or destination of transitions that connect a state vertex in the containing state machine with a subvertex in the referenced state machine" (section 2.12 in [Obj01]).

From the Well-FormednessRules for the submachine states:

- Only stub states are allowed as substates of a submachine state.

$$self.subvertex \rightarrow forAll(s \mid s.oclIsKindOf(StubState))$$

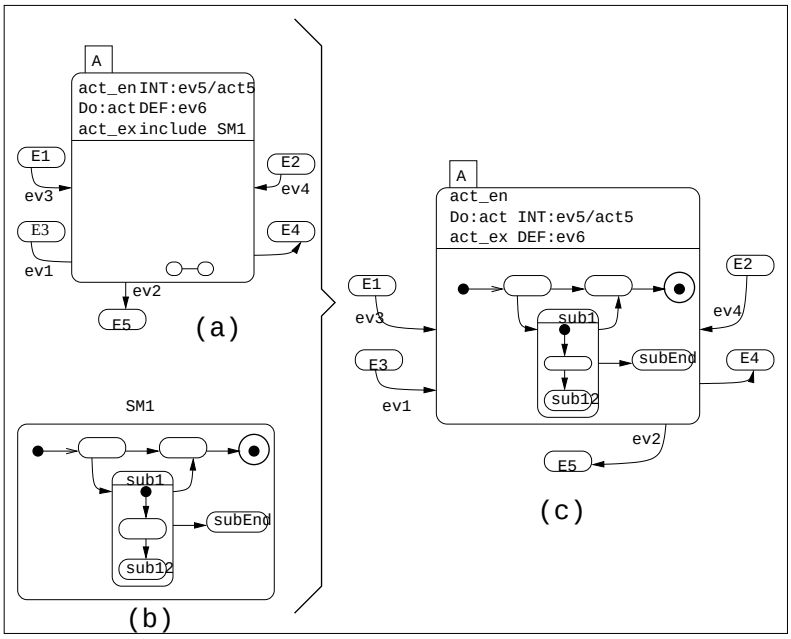


Figure 5.11: (a)A submachine state. (b)The referenced state machine. (c)The state machine inside the composite state.

- Submachine states are never concurrent.

  $self.isConcurrent = false$

Figure 5.10 shows an example of the combined use of the submachine states togheter with the stub states. In Figure 5.10(a) the submachine state A is depicted where the states that act as entry points ($sub1$ and $sub1 :: sub12$) and exit points ($subEnd$) can be identified as stub states. Figure 5.10(b) shows the referenced state machine $SM1$ where the states corresponding to the stub states can be seen. Finally, in Figure 5.10(c) the referenced state machine inside the submachine state linking by the stub states has been depicted.

The interpretation given for the composite states affects in different ways to the interpretation of the submachine states since these are just an specialization of the first ones. The implicit entry and exit were forbidden for the composite states. It is the same case for the submachines states since they are semantically equivalent. Therefore the default entry is the only entry addmited and the default exit and the exit from the boundaries of the submachine states the only exits allowed. This means that any of the states in the referenced state machine can be used neither for the entry nor the exit, therefore stub states make no sense in our interpretation. This restrictions convert submachine states in our interpretation in states that reference

a state machine that is entered by its initial state and exited by default or taking any transition in its boundaries. Figure 5.11 shows an example of the use of the submachine states with the restrictions of our interpretation.

To translate submachine states into LGSPNs we propose that, previously, the referenced state machine is enclosed into the state, as in Figure 5.11(c). Then the translation can be performed as if the state was a composite state and its formal translation used. In this way the modeller can use the syntactical convenience of the submachine states and when the analysis must be performed the substitution can be done.

5.4 History pseudostates

Two kind of history pseudostates are allowed in UML, *deepHistory* and *shallowHistory*. They represent different ways to enter in a composite state. These are the sentences of UML that explain them: "*deepHistory* is used a shorthand notation that represents the most recent active configuration of the composite state that directly contains this pseudostate; that is, the state configuration that was active when the composite state was last exited. A composite state can have at most one deep history vertex. A transition may originate from the history conector to the *default* deep history state. This transition is taken in case the composite state had never been active before or the most recently active configuration was the final state". Which concerns *shallowHistory* the difference is that "it represents the most recent active configuration of its containing state (but not the substates of that substate)".

Furthermore, it is interesting to highlight from the Well-FormednessRules for the composite states that:

- A composite state can have at most one deep history vertex,

$$self.subvertex \rightarrow select(v \mid v.oclIsKindOf(Pseudostate)) \rightarrow \\ select(p : Pseudostate \mid p.kind = \#deepHistory) \rightarrow size \leq 1$$

- A composite state can have at most one shallow history vertex,

$$self.subvertex \rightarrow select(v \mid v.oclIsKindOf(Pseudostate)) \rightarrow \\ select(p : Pseudostate \mid p.kind = \#shallowHistory) \rightarrow size \leq 1$$

The use of history pseudostates presents a major drawback from the compositionality point of view: It implies to cross the boundaries of the composite state that includes the history pseudostate since the transition that indicates its entrance must arrive to it, in other case (if the transition stops on the boundary of the composite state) it is not possible to distinguish it from the transitions that indicate the default entry. As it was discussed in section 5.1, boundary crossing is not allowed in any clean compositional model because it violates encapsulation.

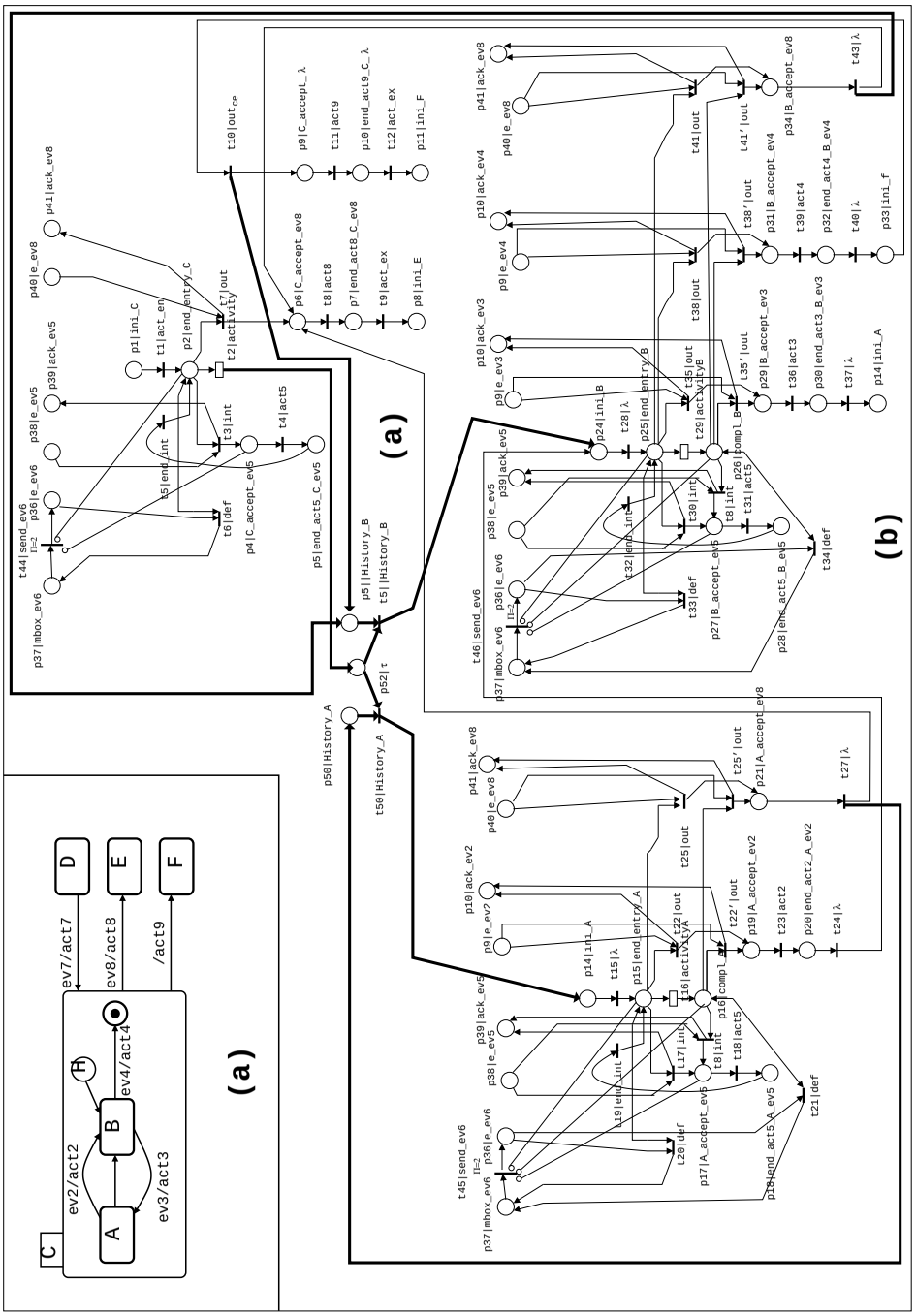


Figure 5.12: A composite state with history pseudostate.

Moreover, as it is discussed in [SG99], the history pseudostates correspond to a *"repeated duplication of the entire composite state, once for each alternative remembered substate. It blinds most developers to the real complexity of what they have created"*.

From the point of view of the compositionality, the violation of the encapsulation makes impossible to formalize a composite state that includes a history pseudostate since the expected behaviour of the composite state does not depend on its own. Nevertheless, the use of this kind of pseudostates could be necessary to model interrupts and co-routines [SG99]. Therefore we propose an example, see Figure 5.12, that suggests the translation of a composite state with a history pseudostate into the LGSPN formalism. This example does not try to promote the use of history states, on the contrary we adhere to the advise of discouraging the use of the history pseudostates given in [SG99]. On the other hand, the example does not intent any kind of formalization, it is given just at descriptive level, but it could be useful when the use of an history state is interesting for a modeler.

The example in Figure 5.12(a) is the same as in Figure 5.1 but the initial pseudostate has been changed by a shallow history state that enters substate B and therefore the composite state C. For simplicity we have avoided to rewrite the activities, entry actions etc, but the translation takes them into account, so we refer to Figure 5.1 for details. The elements modified with respect to the LGSPN in Figure 5.4 to obtain the new translation are in bold face in Figure 5.12, they are the following:

- A new place for each state ($History\_A, History\_B$). They have a token when the corresponding state must be entered.
- A new transition for each state ($History\_A, History\_B$)). They represent the entrace in the corresponding state.
- A new place (τ) to distinguis between different entrances.
- Arcs to connect the history places and the history transitions.
- Arcs to connect the place with the history places.
- Arcs to connect the transitions that represent the exit of the states with the history places.
- The arc from $t2|activity$ to $p12|ini\_ps$ has been removed to avoid immediate entrance in state A.
- A token is added to the place that represents the history default state ($History\_B$)).

5.5 Fork and join pseudostates and join

UML proposes to use a join pseudostate to "merge several transitions emanating from source vertices in different orthogonal regions" and to use a fork pseudostate to "split

an incoming transition into two or more transitions terminating on orthogonal target vertices" (see section 2.12 in [Obj01]). To ultimately understand the role of this kind of pseudostates in UML, the following Well-FormednessRules are given:

- A join vertex has at least two incoming transitions and exactly one outgoing transition.

  $(self.kind = \#join)$ **implies**
  $((self.outgoing \rightarrow size = 1)$ **and** $(self.incoming \rightarrow size \geq 2))$

- A fork vertex must have at least two outgoing transitions and exactly one incoming transition.

  $(self.kind = \#fork)$ **implies**
  $((self.incoming \rightarrow size = 1)$ **and** $(self.outgoing \rightarrow size \geq 2))$

- A fork segment should not have guards or triggers.

  $(self.source.oclIsKindOf(Pseudostate)$
  and not $oclIsKindOf(self.stateMachine, ActivityGraph))$ **implies**
  $((self.source.oclAsType(Pseudostate).kind = \#fork)$ **implies**
  $((self.guard \rightarrow isEmpty)$ **and** $(self.trigger \rightarrow isEmpty)))$

- A join segment should not have guards or triggers.

  $(self.target.oclIsKindOf(Pseudostate)$
  and not $oclIsKindOf(self.stateMachine, ActivityGraph))$ **implies**
  $((self.target.oclAsType(Pseudostate).kind = \#join)$ **implies**
  $((self.guard \rightarrow isEmpty)$ **and** $(self.trigger \rightarrow isEmpty)))$

- A fork segment should always target a state.

  $(self.stateMachine \rightarrow notEmpty$
  and not $oclIsKindOf(self.stateMachine, ActivityGraph))$ **implies**
  $self.source.oclIsKindOf(Pseudostate)$ **implies**
  $((self.source.oclAsType(Pseudostate).kind = \#fork)$ **implies**
  $((self.target.oclIsKindOf(State)))$

- A join segment should always originate from a state.

  $(self.stateMachine \rightarrow notEmpty$
  and not $oclIsKindOf(self.stateMachine, ActivityGraph))$ **implies**
  $self.target.oclIsKindOf(Pseudostate)$ **implies**
  $((self.target.oclAsType(Pseudostate).kind = \#join)$ **implies**
  $((self.source.oclIsKindOf(State)))$

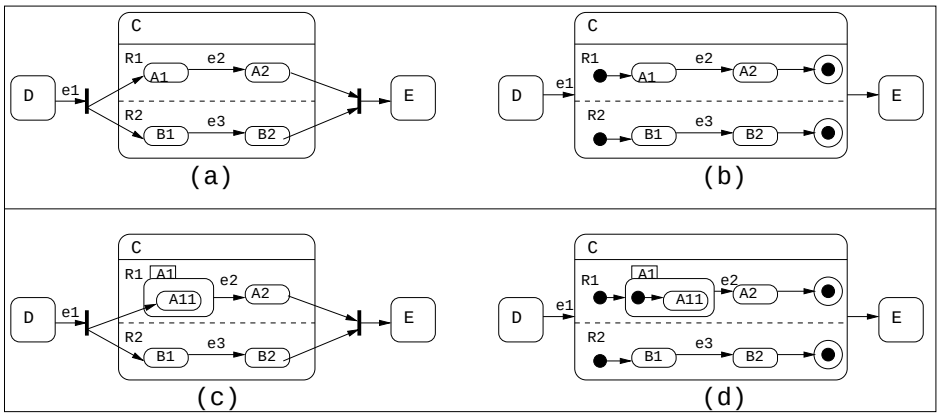


Figure 5.13: Fork and join pseudostates.

- All transitions outgoing a fork vertex must target states in different regions of a concurrent state.

  $(self.kind = \#fork$
  and not $oclIsKindOf(self.stateMachine, ActivityGraph))$ **implies**
  $self.outgoing \rightarrow forAll(t1, t2 \mid t1 \neq t2$ **implies**
  $(self.stateMachine.LCA(t1.target, t2.target).$
  $container.isConcurrent)$

- All transitions incoming a join vertex must originate in different regions of a concurrent state.

  $(self.kind = \#join$
  and not $oclIsKindOf(self.stateMachine, ActivityGraph))$ **implies**
  $self.incoming \rightarrow forAll(t1, t2 \mid t1 \neq t2$ **implies**
  $(self.stateMachine.LCA(t1.source, t2.source).$
  $container.isConcurrent)$

As in the case of history pseudostates, the use of fork and join pseudostates presents a major drawback from the compositionality point of view since it implies crossing boundaries. Nevertheless, fork and join pseudostates as defined in UML are actually syntactical conventions but they do not add new semantics. Fork is proposed to enter orthogonal regions, by means of a triggered event, in composite states and the join to exit these regions by means of completion. This behaviour can be obtained using some of the UML elements studied so far, without using fork and join pseudostates. Figure 5.13 will help us to understand why fork and join are just syntactic sugar and crossing boundaries avoided.

In Figure 5.13(a) a fork substate can be seen with an incoming transition (as it is mandatory) and two outgoing transitions arriving states in orthogonal regions (more

than two outgoing transitions are allowed but our example does not loose generality). The incoming transition is triggered by an event (it is not mandatory) but the outgoing transitions must not have event labels nor guards. The join pseudostate has two incoming transitions from orthogonal regions and an outgoing transition that must target a state, none of the transitions can be labelled with triggers or guards.

Figure 5.13(b) has the same meaning as Figure 5.13(a) but it does not use fork nor join pseudostates. From Figure 5.13, it is easy to infer a set of rules that transform a model that uses forks and joins into a model that eliminates them and therefore it can be translated applying the formalization given in this chapter. In this way the modeler can use fork and join as a syntactical convenience when desired but before to translate the model into a LGSPN the following rules must be applied:

- A transition must be added: It must exit from the state origin of the fork and must target the concurrent state which is the least common ancestor of the regions entered. It will be labelled with the trigger of the fork if it exists.
- Each region entered by the fork must add a default entry for each entered state at any level. See Figures 5.13(c) and (d).
- Each region exited by the join must add a default exit for each state exited at any level. See Figures 5.13(c) and (d).
- A transition must be added: It must exit from the concurrent state which is the least common ancestor of the regions entered and must target the target of the join.
- The fork and join pseudostate and its outgoing and incoming transitions must be eliminated.

5.6 Junction and choice pseudostates

In UML, junction pseudostates "are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transitions paths between states. They realize static conditional branch" [Obj01]. However choice pseudostates "when reached, result in the dynamic evaluation of the guards of its outgoing transitions" [Obj01]. Moreover the following Well-FormednessRules are given in UML:

- A junction vertex must have at least one incoming and one outgoing transition.

  $(self.kind = \#junction)$ **implies**
  $((self.incoming \rightarrow size \geq 1)$ **and** $(self.outgoing \rightarrow size \geq 1))$

- A choice vertex must have at least one incoming and one outgoing transition.

  $(self.kind = \#choice)$ **implies**
  $((self.incoming \rightarrow size \geq 1)$ **and** $(self.outgoing \rightarrow size \geq 1))$

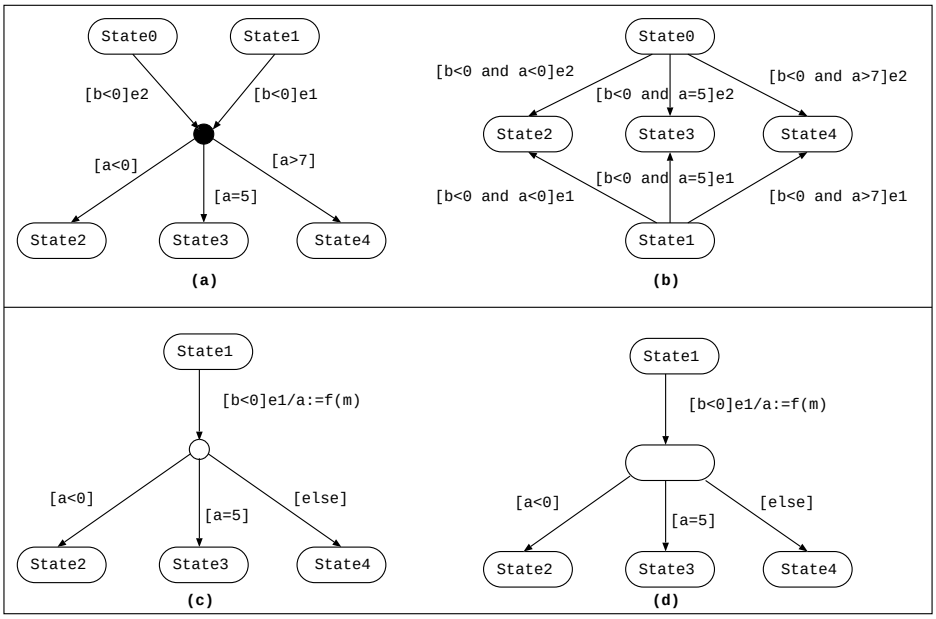


Figure 5.14: Fork and join pseudostates.

Junction pseudostates are defined as "semantic-free", but it is very common to use them as a syntactical convenience to reuse guards. In Figure 5.14(a), State2 is reached from State0 when the guard $[b < 0]$ and the guard $[a < 0]$ are true. It must be taken into account that the guards are evaluated in State0 performing "static conditional branch". This behaviour is the same as in Figure 5.14(b) but without reusing guards. Therefore the purpose of the junction pseudostate is just guard conjunction. As we have decided that guards are out of the scope of our work, as it was explained in chapter 4, then the use of the junction pseudostates makes no sense in our models.

Choice pseudostates differ from junction pseudostates only because the guards of the outgoing transitions are evaluated "at the time the choice point is reached". In Figure 5.14(c), if the guard $[b < 0]$ is evaluated to true then the action $a := f(m)$ is performed and the choice point reached, then the guard $[a < 0]$ is evaluated and if it is true State2 is reached. This behaviour implies that the meaning of the choice pseudostate is not guard conjunction but a simple state without procesing capabilities where evaluate guards, Figure 5.14(d) shows its equivalent. Newly this model element is discarded in our models since guards are not allowed.

5.7 Synchronous states

A synchronous state "is used for synchronizing two concurrent regions of a state machine. It is different from a state in the sense that it is not mapped to a boolean

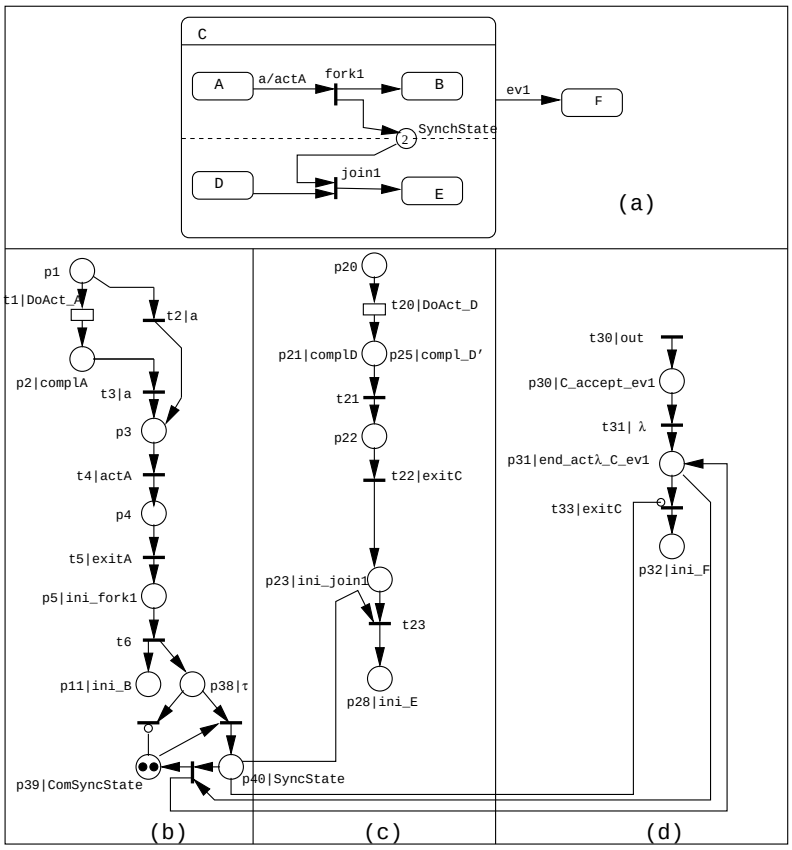


Figure 5.15: Synchronous state: limited bound to two.

value (active, not active), but an integer. It is used in conjunction with forks and joins to ensure that one region leaves a particular state or states before another region can enter a particular state or states. The integer that it is mapped to, the bound attribute, means the difference between the number of times the incoming and outgoing transitions are fired" [Obj01]. The Well-FormednessRules in UML are the following:

- The value of the bound attribute must be a positive integer, or unlimited. $(self.bound > 0)$ **or** $(self.bound = unlimited)$

- All incoming transitions to a synchronous state must come from the same region and all outgoing transitions must go to the same region.

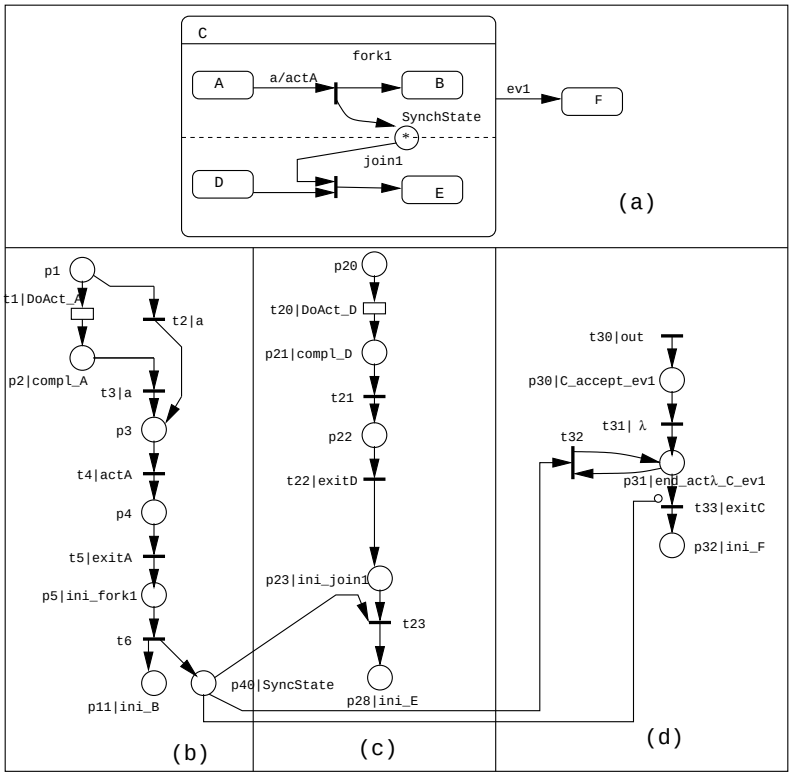


Figure 5.16: Synchronous state: unlimited bound.

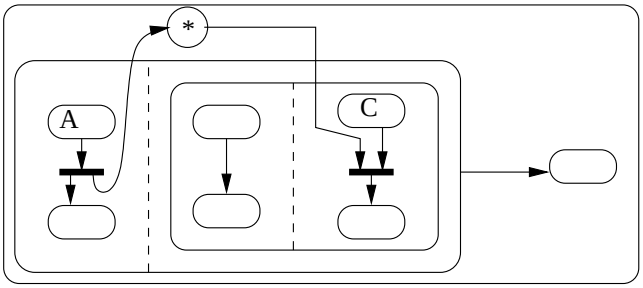


Figure 5.17: Synchronizing no sibling regions.

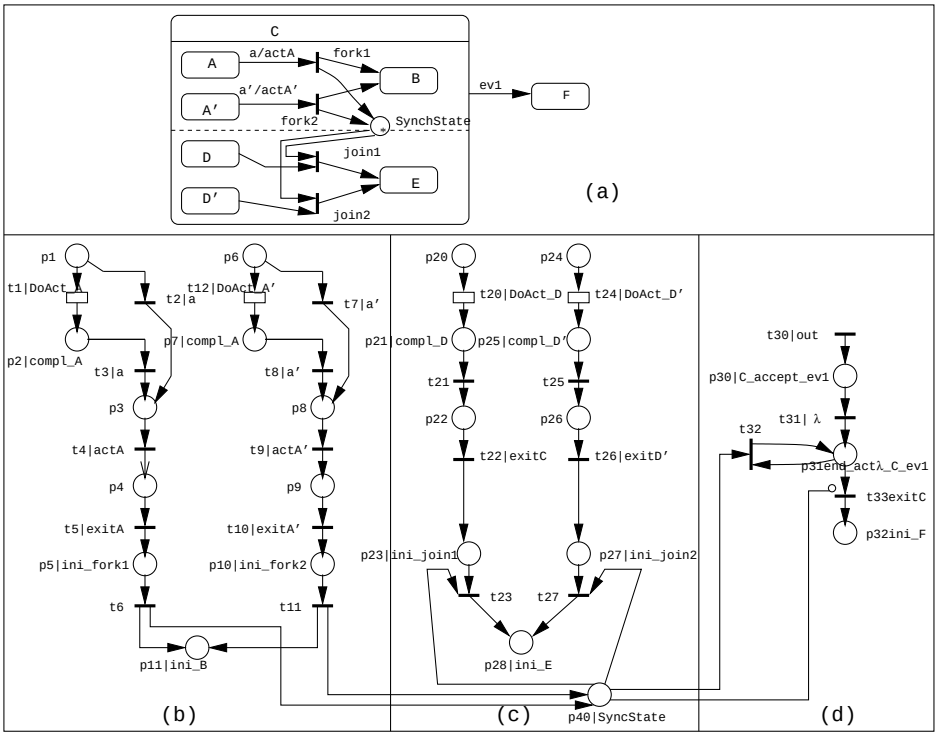


Figure 5.18: Synchronous state with multiple joins and forks.

5.7.1 Informal interpretation

In the following, the most remarkable points of the interpretation of the synchronous states are broached stressing how the LGSPN obtained by the translation given in the next section solves each one of them:

1. "When the source region reaches a synchronization fork, the target states of that fork become active, including the synch state" [Obj01]. The fact that the synchronization fork is reached is represented in our LGSPN by a token in the state ini_fork (see in Figure 5.16(b) place $p5$). Also it can be observed that a token in this place is used to fire the transitions that enter the target state ini_B and the synch state $SynchState$.

2. "When the target region reaches the corresponding synchronization join, it is prevented from continuing unless all the states leading into the synchronization join are active, including the synch states" [Obj01]. See in Figure 5.16(c) that this behaviour is obtained in our model by transition $t23$ whose input places are $SynchState$ and ini_join1.

3. Since the synchronized regions do not need to be siblings, they only must have a

common ancestor state, models such that depicted in Figure 5.17 are allowed in UML. But allowing to synchronize not siblings regions the problem of boundary crossing arises newly, violating encapsulation with the consequent drawback over the compositionality. Therefore we deal only with the case of synchronising siblings regions.

4. "Synch states keep count of the difference between the number of times their incoming and outgoing transitions are fired. When an incoming transition is fired, the count is incremented by one, unless its value is equal to the value defined in the bound atributte. In that case, the count is not incremented. When an outgoing transition is fired the count is decremented by one. An outgoing transition may fire only if the count is greater than zero, which prevents the count from becoming negative" [Obj01]. This behaviour is obtained in our models by two different realizations depending on if the value of attribute bound of the synchronous state, its size, is unlimited or has any integer value. See Figure 5.16(b,c) and Figure 5.15(b,c) respectively.

5. "A synchronous state may have multiple incoming and outgoing transitions, used for multiple synchronization points in each region" [Obj01]. Our interpretation, that is consistent with that of UML but without ambiguities, shows a clear "Petri net style": Any fork can increase the value of the count and any join decrease it. Figure 5.18 shows clearly this behaviour through transitions $t6, t11$ that increase this counter by means of increasing the number of tokens in place $p40|SyncState$ and through transitions $t23, t27$ that decrease it.

6. "The count is automatically set to zero when its container state is exited" [Obj01]. This behaviour is obtained in our models by two different realizations depending on if the size of the synchronous state is unlimited or has any integer value. See transitions $t32, t33$ in Figure 5.16(d) for the first case and transitions $t33, t40$ in Figure 5.15(d) for the second case.

Summarizing, in our interpretation synchronous states are allowed with one or multiple incoming forks and with one or multiple outgoing joins but taking into account that all of them must belong to siblings regions. Moreover, it must be remembered that outgoing transitions from a fork and outgoing and incoming trasitions from and to a join can not be labelled with guards nor events nor actions.

5.7.2 Formal translation

In this section the formal translation of the synchronous states is given. Since different systems are obtained for the synchronous states depending on their size (bounded/unbounded), we propose the translation in two different subsections. Finally, a new subsection is introduced to interpret the synchronous states (bounded/unbounded) in the context of the composite states.

A. UnBounded synchronous states

To obtain a labeled system $\mathcal{LS}^{u}_{syn}$ that represents an unbounded synchronous state *syn* (cfr. Definition 5.7) it is necesary to compose the labeled systems that interpretate its incoming forks, $\mathcal{LS}^{Fu}_{syn}$(Definition 5.4), and its outgoing joins, $\mathcal{LS}^{Ju}_{syn}$(Definition 5.6).

The following systems must be formalized: The labelled system, $\mathcal{LS}^{fu}_{fork}$, that interpretates one incoming fork (Definition 5.3) and the labelled system, $\mathcal{LS}^{ju}_{join}$, that interpretates one outgoing join (Definition 5.5).

Definition 5.3. *The system* $\mathcal{LS}^{fu}_{fork} = (S^{fu}_{fork}, \psi^{fu}_{fork}, \lambda^{fu}_{fork})$ *that interpretates one incoming fork of an unbounded synchronous state is defined as follows,*

For the LGSPN system $S^{fu}_{fork} = \langle P^{fu}_{fork}, T^{fu}_{fork}, I^{fu}_{fork}, O^{fu}_{fork}, H^{fu}_{fork}, \Pi^{fu}_{fork}, W^{fu}_{fork}, M_0{}^{fu}_{fork}\rangle$ we define,

$P^{fu}_{fork} = \{p1, p2\} \cup \{p\}_{ini},\ \ T^{fu}_{fork} = \{t1\},$

the functions $I^{fu}_{fork}(), O^{fu}_{fork}(), H^{fu}_{fork}(), W^{fu}_{fork}(), \Pi^{fu}_{fork}()$ are:
$I^{fu}_{fork}(t1) = \{p1\}, O^{fu}_{fork}(t1) = \{p2\} \cup \{p\}_{ini}, H^{fu}_{fork}(t1) = \emptyset, \Pi^{fu}_{fork}(t1) = 1, W^{fu}_{fork}(t1) = 1$

The labelling function for places is:

$$\psi^{fu}_{fork}(p) = \begin{cases} ini\_fork & \text{if } p = p1 \ \wedge\ fork = name(fork) \\ synch & \text{if } p = p2 \ \wedge\ \exists t \in out(fork) : target(t) = synchst \ \wedge \\ & synchst \in SynchState \ \wedge\ synch = name(syncst) \\ ini\_state & \text{if } p \in \{p\}_{ini} \ \wedge\ \exists t_{st} \in out(fork) : target(t_{st}) = st \ \wedge \\ & st \in State \ \wedge\ state = name(st) \end{cases}$$

The labelling function for transitions is $\lambda^{fu}_{fork}(t1) = \tau$.

Definition 5.4. *The system* $\mathcal{LS}^{Fu}_{syn} = (S^{Fu}_{syn}, \psi^{Fu}_{syn}, \lambda^{Fu}_{syn})$ *that represents all the incoming forks of the unbounded synchronous state syn is defined as follows,*

$$\mathcal{LS}^{Fu}_{syn} = \mathop{||}_{\{syn\}\cup Lini}^{fork \in Fork\_syn} \mathcal{LS}^{fu}_{fork}$$

Where $Fork\_syn = \{fork, \forall fork \in Pseudostate : \exists t \in Transition, sync = target(t) \ \wedge\ fork = source(t)\}$ and $Lini = \{ini\_st, \forall st \in State : \exists t \in Transition, source(t) \in Fork\_syn \ \wedge\ st = target(t)\}$

Definition 5.5. *The system* $\mathcal{LS}^{ju}_{join} = (S^{ju}_{join}, \psi^{ju}_{join}, \lambda^{ju}_{join})$ *that interpretates one outgoing join of an unbounded synchronous state is defined as follows,*

For the LGSPN system $S^{ju}_{join} = \langle P^{ju}_{join}, T^{ju}_{join}, I^{ju}_{join}, O^{ju}_{join}, H^{ju}_{join}, \Pi^{ju}_{join}, W^{ju}_{join}, M_0{}^{ju}_{join}\rangle$ we define,

$P^{ju}_{join} = \{p1, p2\} \cup \{p\}_{ini},\ \ T^{ju}_{join} = \{t1\},$

the functions $I^{ju}_{join}(), O^{ju}_{join}(), H^{ju}_{join}(), W^{ju}_{join}(), \Pi^{ju}_{join}()$ are:
$I^{ju}_{join}(t1) = \{p1, p2\}, O^{ju}_{join}(t1) = \{p\}_{ini}, H^{ju}_{join}(t1) = \emptyset, \Pi^{ju}_{join}(t1) = 1, W^{ju}_{join}(t1) = 1$
The labelling function for places is:

$$\psi^{ju}_{join}(p) = \begin{cases} synch & \text{if } p = p1 \ \wedge\ \exists t \in in(join) : source(t) = synchst \ \wedge \\ & synchst \in SynchState \ \wedge\ synch = name(synchst) \\ ini\_join & \text{if } p = p2 \ \wedge\ join = name(join) \\ ini\_state & \text{if } p \in \{p\}_{ini} \ \wedge\ \exists t_{st} \in out(join) : target(t_{st}) = st \ \wedge \\ & st \in State \ \wedge\ state = name(st) \end{cases}$$

The labelling function for transitions is $\lambda^{ju}_{join}(t1) = \tau$.

Definition 5.6. *The system $\mathcal{LS}^{Ju}_{syn} = (S^{Ju}_{syn}, \psi^{Ju}_{syn}, \lambda^{Ju}_{syn})$ that represents all the outgoing joins of the unbounded synchronous state syn is defined as follows,*

$$\mathcal{LS}^{Ju}_{syn} = \mathop{||}_{\{syn\} \cup Lini}^{join \in Join\_syn} \mathcal{LS}^{ju}_{join}$$

Where $Join\_syn = \{join, \forall join \in Pseudostate : \exists t \in Transition, syn = source(t) \ \wedge\ join = target(t)\}$ and $Lini = \{ini\_st, \forall st \in State : \exists t \in Transition, join \in Join\_syn \ \wedge\ st = target(t) \wedge \}$

Definition 5.7. *The system $\mathcal{LS}_{syn} = (S_{syn}, \psi_{syn}, \lambda_{syn})$ that represents an unbounded synchronous state syn is defined as follows,*

$$\mathcal{LS}^{u}_{syn} = \mathcal{LS}^{Fu}_{syn} \mathop{||}_{\{syn\}} \mathcal{LS}^{Ju}_{syn}$$

The system in Definition 5.7 represents a synchronous state *syn* with unbounded size that synchronizes several siblings regions in a composite state. But upon exiting the composite state it is possible that the place *syn* has tokens and they must be removed. Figure 5.16(d) shows the elements that must be added to each outgoing transition of the composite state, they are: a new transition with an input and an output arc from and to the place that represents the end of the action of the transition and an input arc from the *syn* place. Finally an inhibitor arc to the place that represents the exit of the composite state to the *syn* place must be added.

B. Bounded synchronous states

To obtain a labeled system $\mathcal{LS}^{b}_{syn}$ that represents a bounded synchronous state *syn* (Definition 5.12), it is necesary to compose the labeled systems that interpretate its incoming forks $\mathcal{LS}^{Fb}_{syn}$ (Definition 5.9) and its outgoing joins $\mathcal{LS}^{Jb}_{syn}$ (Definition 5.11).

Definitions 5.8 and 5.10 formalize the following labeled systems: $\mathcal{LS}^{fb}_{fork}$ that interpretates one incoming fork (Definition 5.8) and $\mathcal{LS}^{jb}_{join}$that interpretates one outgoing join (Definition 5.10).

Definition 5.8. *The system $\mathcal{LS}^{fb}_{fork} = (S^{fb}_{fork}, \psi^{fb}_{fork}, \lambda^{fb}_{fork})$ that interpretates one incoming fork of a bounded synchronous state is defined as follows,*

For the LGSPN system $S^{fb}_{fork} = \langle P^{fb}_{fork}, T^{fb}_{fork}, I^{fb}_{fork}, O^{fb}_{fork}, H^{fb}_{fork}, \Pi^{fb}_{fork}, W^{fb}_{fork}, M_0{}^{fb}_{fork}\rangle$ we define,

$P^{fb}_{fork} = \{p1, p2, p3, p4\} \cup \{p\}_{ini}$, $T^{fb}_{fork} = \{t1, t2, t3\}$,

the functions $I^{fb}_{fork}(), O^{fb}_{fork}(), H^{fb}_{fork}(), W^{fb}_{fork}(), \Pi^{bf}_{fork}()$ are:

$$I^{fb}_{fork}(t) = \begin{cases} p1 & \text{if } t = t1 \\ p4 & \text{if } t = t2 \\ p3, p4 & \text{if } t = t3 \end{cases} \qquad O^{fb}_{fork}(t) = \begin{cases} p4, p5 & \text{if } t = t1 \\ \emptyset & \text{if } t = t2 \\ p2 & \text{if } t = t3 \end{cases}$$

$$H^{fb}_{fork}(t) = \begin{cases} p3 & \text{if } t = t2 \\ \emptyset & \text{otherwise} \end{cases} \qquad \Pi^{fb}_{fork}(t1) = 1, W^{fb}_{fork}(t1) = 1$$

The labelling function for places is:

$$\psi^{fb}_{fork}(p) = \begin{cases} ini\_fork & \text{if } p = p1 \ \wedge \ fork = name(fork) \\ synch & \text{if } p = p2 \ \wedge \ \exists t \in out(fork) : target(t) = synchst \ \wedge \\ & synchst \in SynchState \ \wedge \ synch = name(syncst) \\ count\_synch & \text{if } p = p3 \\ \tau & \text{if } p = p4 \\ ini\_state & \text{if } p \in \{p\}_{ini} \ \wedge \ \exists t_{st} \in out(fork) : target(t_{st}) = st \ \wedge \\ & st \in State \ \wedge \ state = name(st) \end{cases}$$

The labelling function for transitions is $\lambda^{fb}_{fork}(t) = \tau, \forall t \in T^{fb}_{fork}$.

Definition 5.9. *The system $\mathcal{LS}^{Fb}_{syn} = (S^{Fb}_{syn}, \psi^{Fb}_{syn}, \lambda^{Fb}_{syn})$ that represents all the incoming forks of a bounded synchronous state syn is defined as follows,*

$$\mathcal{LS}^{Fb}_{syn} = \mathop{||}_{\{syn\}\cup\{count\_syn\}\cup Lini}^{fork \in Fork\_syn} \mathcal{LS}^{fb}_{fork}$$

Where $Fork\_syn$ and $Lini$ are defined as in Definition 5.4.

Definition 5.10. *The system $\mathcal{LS}^{jb}_{join} = (S^{jb}_{join}, \psi^{jb}_{join}, \lambda^{jb}_{join})$ that interpretates one outgoing join of a bounded synchronous state is defined as follows,*

For the LGSPN system $S^{jb}_{join} = \langle P^{jb}_{join}, T^{jb}_{join}, I^{jb}_{join}, O^{jb}_{join}, H^{jb}_{join}, \Pi^{jb}_{join}, W^{jb}_{join}, M_0{}^{jb}_{join} \rangle$ we define,

$P^{jb}_{join} = \{p1, p2, p3\} \cup \{p\}_{ini},\ T^{jb}_{join} = \{t1\},$

the functions $I^{jb}_{join}(), O^{jb}_{join}(), H^{jb}_{join}(), W^{jb}_{join}(), \Pi^{jb}_{join}()$ are:

$I^{jb}_{join}(t1) = \{p1, p2\},\ O^{jb}_{join}(t1) = \{p3\} \cup \{p\}_{ini},$

$H^{jb}_{join}(t1) = \emptyset,\ \Pi^{jb}_{join}(t1) = 1,\ W^{jb}_{join}(t1) = 1$

The labelling function for places is:

$$\psi^{jb}_{join}(p) = \begin{cases} synch & \text{if } p = p1 \ \wedge\ \exists t \in in(join) : source(t) = synchst \ \wedge \\ & synchst \in SynchState \ \wedge\ synch = name(synchst) \\ ini\_join & \text{if } p = p2 \ \wedge\ join = name(join) \\ count\_synch & \text{if } p = p3 \\ ini\_state & \text{if } p \in \{p\}_{ini} \ \wedge\ \exists t_{st} \in out(join) : target(t_{st}) = st \ \wedge \\ & st \in State \ \wedge\ state = name(st) \end{cases}$$

The labelling function for transitions is $\lambda^{jb}_{join}(t1) = \tau$.

Definition 5.11. *The system $\mathcal{LS}^{Jb}_{syn} = (S^{Jb}_{syn}, \psi^{Jb}_{syn}, \lambda^{Jb}_{syn})$ that represents all the outgoing joins of the bounded synchronous state syn is defined as follows,*

$$\mathcal{LS}^{Jb}_{syn} = \mathop{||}_{\{syn\} \cup \{count\_syn\} \cup Lini}^{join \in Join\_syn} \mathcal{LS}^{jb}_{join}$$

Where $Join\_syn$ and $Lini$ are defined as in Definition 5.6.

Definition 5.12. *The system $\mathcal{LS}^{b}_{syn} = (S^{b}_{syn}, \psi^{b}_{syn}, \lambda^{b}_{syn})$ that represents a bounded synchronous state syn is defined as follows,*

$$\mathcal{LS}^{b}_{syn} = \mathcal{LS}^{Fb}_{syn} \mathop{||}_{\{syn\} \cup \{count\_syn\}} \mathcal{LS}^{Jb}_{syn}$$

The system in Definition 5.12 represents a synchronous state *syn* with bounded size that synchronizes several siblings regions in a composite state. But upon exiting the composite state it is possible that: a) the place *syn* has tokens and they must be removed, b) the place *count_syn* must have as many tokens as the size of the state. Figure 5.15(b,d) shows the elements that must be added to each outgoing transition of the composite state, they are: a new transition with an input and an output arc from and to the place that represents the end of the action of the transition and an input arc from the *syn* place and an output arc to the *count_syn* place. Finally an inhibitor arc to the place that represents the exit of the composite state to the *syn* place must be added.

C. The model of a composite state with synchronous states

Definition 5.13. *The system* $\mathcal{LS}_C = (S_C, \psi_C, \lambda_C)$ *that represents the composite state* C *that can contain* k *synchronous states is redefined from* Definition 5.2 *as follows,*

$$\mathcal{LS}_C = \begin{cases} \mathcal{LS}_C^{flat} \quad \textit{(Def. 5.1)} & \text{if } C \text{ has an enclosed ``flat'' state machine} \\ \mathcal{LS}_C^{flat} \underset{Lev^P \cup Lstate^P}{||} \mathcal{LS}^{Compo} \underset{Lstate^P}{||} \mathcal{LS}_C^{SYN} & \text{otherwise} \end{cases}$$

where

$$\mathcal{LS}_C^{SYN} = \mathcal{LS}_C^{BOUND} \underset{Lstate^P}{||} \mathcal{LS}_C^{UNBOUND}$$

$$\mathcal{LS}_C^{BOUND} = \overset{syn \in C}{\underset{Lstate^P}{||}} \mathcal{LS}_{syn}^{b} \qquad \mathcal{LS}_C^{UNBOUND} = \overset{syn \in C}{\underset{Lstate^P}{||}} \mathcal{LS}_{syn}^{u}$$

The labels not defined are the same as in Definition 5.2.

5.8 The model of a state machine

As well as in UML, in our interpretation a state machine is make up of just one composite state, the *top* state. The *top* state cannot have containing states nor source transitions. This interpretation allow us to say that a state machine *sm* with top state *top* is represented by the LGSPN obtained by applying Definition 5.13 to *top*, i.e. $\mathcal{LS}_{top}$.

5.9 The model of a UML system

In order to create an analysable model for a system assuming that it is described as a set of k state machines, the discussion given in section 4.11 for the "flat" state machines is valid, but assuming that the labelled systems for the k state machines are not produced according to the Definition 4.53 but as given in section 5.8. From this system a LGSPN model that represent the UML system can be obtained as it was explained in chapter 7 section 7.2.2.

5.10 Conclusions

In this chapter we have given a formal semantics in terms of LGSPNs to a set of UML elements not dealed so far. In the following we briefly comment each of them.

Composite states have been translated taking into account that outgoing transitions are inherited by the substates avoiding lock-in process interpretation; internal

transitions and deferred events are also inherited. Only the default entry is allowed since the other kinds of entries violate encapsulation and therefore compositionality cannot be used. Default exit and exit taken inherited transitions are valid but the other kinds are not due to the same reasons given for the forbidden entries. The inheritance policy described causes that in case of potencial conflicting transitions in the UML model our LGSPN model selects the innermost transition.

For the concurrent composite states, as well as for the non-concurrent, only default entry and default exit and exit taking inherited transitions are allowed. Regions are not interpreted as composite states since the additional modelling power is outlandish. The firing policy causes that conflicting transitions in mutually orthogonal regions are fired with the same probability.

Submachine states are just a syntactical convenience to abstract the state machine that they represent. They must be substituted by the referenced state machine and translated to the corresponding LGSPN.

Motivated by [Sim00] we discourage the use of the history pseudostates since they create complex models. Moreover, it is not possible to formalise them upon our perspective of compositionality since they provoke crossing boundaries. Nevertheless, we have given an example of translation at descriptive level.

Fork and join pseudostates are just syntactical conventions. They present the drawback of crossing boundaries, but they are equivalent to other concepts whose translation has been given. We have proposed a set of rules to convert fork and join into tractable structures.

The main motivation of junction pseudostates is to reuse guards while choice pseudostates are quiescent vertices where evaluate guards. Since we have discarded the use of guards these model elements are out of the scope of our work.

Synchronous states to synchronize not sibling regions are not allowed in our interpretation since crossing boundaries should be necessary. We have proposed two different translations for the synchronous states, the first one for synchronous states whose size is limited and the second one for unlimed synchronous states. It is motivated because each case corresponds with a different synchronization policy.

By composing the translation of the elements presented so far, a LGSPN model for an entire state machine is obtained. Moreover, by composing the models that represent several state machines it can be obtained a LGSPN model that represents a software system described by the mentioned state machines.

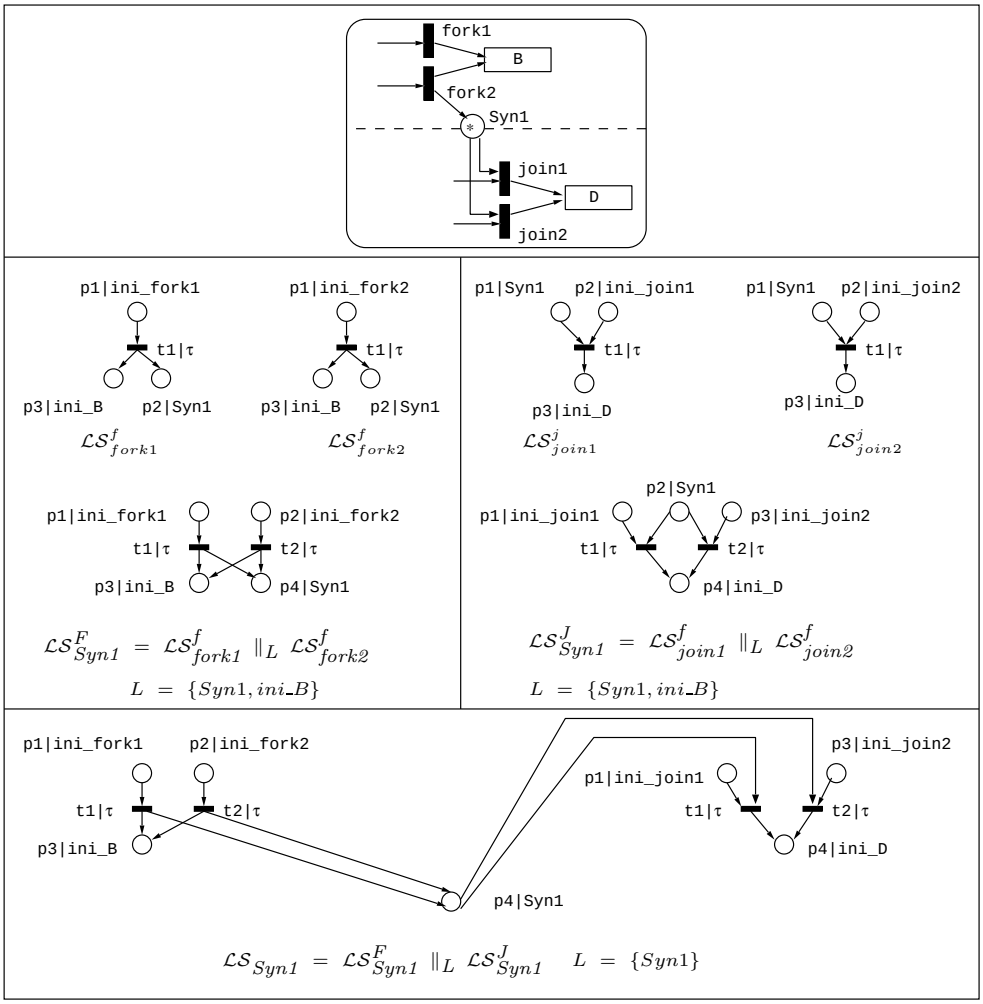


Figure 5.19: Summary of the translation of synchronous states: unlimited bound.

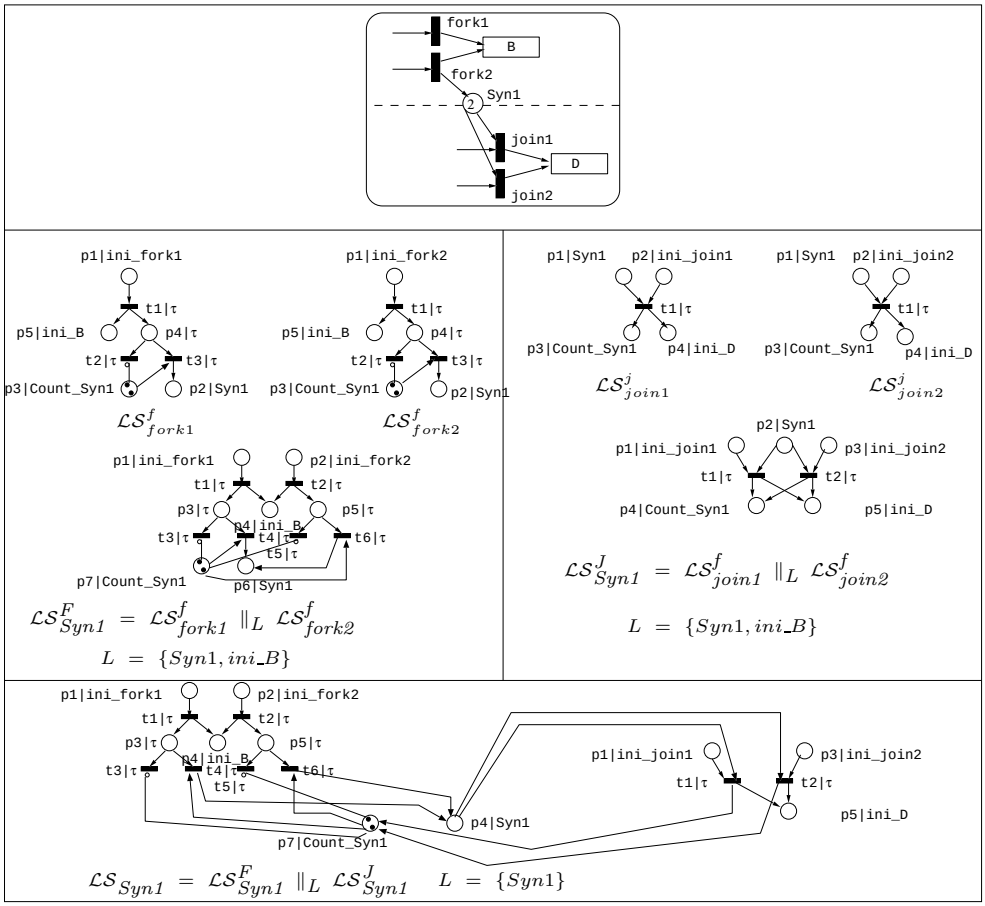


Figure 5.20: Summary of the translation of synchronous states: limited bound.

Chapter 6

UML Activity Graphs Compositional Semantics

Chapter 3 stated our proposal of extension of UML semantics for some diagram types and in chapters 4 and 5 a complete method to translate state machines into GSPN models was presented. Here we will focus on activity graphs (AG) (the package that gives semantics to activity diagrams (AD)). A semantics for them will be presented in terms of labelled generalized stochastic Petri nets (LGSPNs), while we explain their link with other UML diagrams such as statecharts so as to amplify the expressivity at system description. This work has been developed in [LGMC02a, LGMC02b].

Activity diagrams represent UML activity graphs and are just a variant of UML state machines (cfr. [Obj01], 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 2.4). The main goal of ADs is to stress the internal control flow of a process in contrast to statechart diagrams, which represent UML SMs and are often driven by external events.

It must be noted that in this chapter we only focus in those elements that are proper of ADs. Remember that when we studied the role of the AD in chapter 3, we remarked that almost every state in an AD should be an action or subactivity state, so almost every transition should be triggered by the ending of the execution of the entry action or activity associated to the state. Anyway, UML is not strict at this point, so elements from state machines package could occasionally be used when modeling activities using the AD. Also, we recall that in chapter 3 we decided to disallow other states than action, subactivity or call states, and thus to accept only the use of external events by means of call states and control icons involving signals, i.e. signal sendings and signal receipts. So an activity will not ever be interrupted when it is described by an AD.

The performance model obtained from an activity diagram in terms of LGSPNs can be used with performance evaluation purposes with two goals: A) just to obtain performance measures of the model element they describe or B) to compose this

performance model with the performance models of the statecharts that the modeled activity uses in order to obtain a final performance model of the system described by the referred statecharts.

The chapter is organized as follows. Section 6.1 enumerates the main rules of the translation method. Section 6.2 analyzes the translation of each element in the activity graph package into an LGSPN model. Section 6.3 discusses how the stochastic Petri net model for the whole activity diagram can be obtined and how this model can be composed with the ones obtained for the state machines. Finally, section 6.4 summarizes the work realized in this chapter.

6.1 Translation rules and formal definitions

A brief description of each AG element and their translation to LGSPNs is presented in the next section. Section 6.3 illustrates the method to compose those LGSPNs to obtain the whole model for a concrete AD according to our proposed semantics. We must note that, in the following, we suppose that every object derived from ModelElement metaclass has an unique name within its namespace, although it could be not explicitly shown in the model.

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

step 1 The translation of each outgoing and self-loop transition. It is applicable to action, subactivity and call states, and to fork pseudostates. Depending on the kind of transition, a different rule must be applied (see Figures 6.1 and 6.3).

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

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

The formal definition of one of the LGSPN systems shown in Figure 6.1 is stated below. The rest of the cases in Figures 6.1 and 6.3 are straightforward derived from this one, so they will not be explicitly illustrated.

A system for an outgoing timed transition *ott* of an action state *AS* (see Figure 6.1, case 1.a) is an LGSPN $\mathcal{LS}^{ott}_{AS} = (S^{ott}_{AS}, \psi^{ott}_{AS}, \lambda^{ott}_{AS})$ characterized by the set of transitions $T^{ott}_{AS} = \{t_1, t_2\}$, and the set of places $P^{ott}_{AS} = \{p_1, p_2, p_3\}$. The input and output functions are respectively equal to:

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

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

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

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

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

The rate r_{ott} is equal to $1/n$ if the time annotation attached to the AD transition is expressed in the format $\{n\ sec.\}$, or equal to $2/n+m$ if it is expressed in the format $\{n-m\ sec.\}$.

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

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

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

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

6.2 Translating activity graph elements

The following subsections are devoted to translate each package element into an LGSPN; the composition of these nets (section 6.3) results in a stochastic Petri net system that will be used to obtain performance parameters for the modeled element or to combine it with the stochastic Petri net of the statechart that uses the modeled action.

6.2.1 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' (cfr. [Obj01], section 3.85). According to this definition and the translation of simple states in chapter 4 we should interpret the action as atomic and therefore represent it by an immediate transition within the LGSPN corresponding to the state. However, if we considered every action immediate (for action states), then most of the activities modelled by ADs would be immediate too, when they are expected to have a concrete duration. So we will distinguish between timed and not-timed transitions (in ADs) to determine the type of transition needed –timed or immediate– and its associated rate in the resulting LGSPN.

Translating an action state into LGSPN formalism takes the three steps expressed in section 6.1. Given an action state AS let q be the number of outgoing timed transitions OT_i of the state (which do not end in a join pseudostate), q' the number of outgoing not-timed transitions ON_j (which do not end in a join pseudostate), r the number of outgoing timed transitions OTJ_m that end in a join pseudostate, r' the number of outgoing not-timed transitions OTN_n that end in a join pseudostate, s the number of self-loop timed transitions ST_k and s' the number of self-loop not-timed transitions SN_l.

Then for each outgoing or self-loop transition t, we have an LGSPN $\mathcal{LS}^t_{AS} = (S^t_{AS}, \psi^t_{AS}, \lambda^t_{AS})$ as shown in Figure 6.1, cases 1.a-1.f. This fact results in a set of $q + q' + r + r' + s + s'$ LGSPN models that must 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. Note that the operators given in Definitions 2.6 and 2.7 are used:

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

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

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

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

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

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

6.2.2 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.

The translation of a subactivity state into an LGSPN takes the three steps expressed in section 6.1. 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 6.1, 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 the state SS, $\mathcal{LS}_{SS} = (S_{SS}, \psi_{SS}, \lambda_{SS})$. The LGSPNs corresponding to each set of kind of transitions are now obtained by superposition:

$$\mathcal{LS}^O_{SS} = \mathop{||}_{Lstvertex^P, end\_AG}^{i=1,\ldots,q} \mathcal{LS}^{O_i}_{SS} \quad \mathcal{LS}^S_{SS} = \bigsqcup_{end\_AG, out\_\lambda, ini\_SS}^{j=1,\ldots,s} \mathcal{LS}^{S_j}_{SS}$$

$$\mathcal{LS}^{OJ}_{SS} = \mathop{||}_{end\_AG}^{k=1,\ldots,r} \mathcal{LS}^{OJ_k}_{SS}$$

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

$$\mathcal{LS}_{SS} = ((\mathcal{LS}^{OJ}_{SS} \mathop{||}_{end\_AG} \mathcal{LS}^S_{SS}) \mathop{||}_{end\_AG} \mathcal{LS}^O_{SS}) \mathop{||}_{ini\_SS} \mathcal{LS}^B_{SS}$$

6.2.3 Call states

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

Besides, the CallAction may be synchronous or not depending on the value of its attribute *isAsynchronous*, where *synchronous* means that the action will not be completed until the event eventually generated by the action is not consumed by the receiver. In that case, we need a new place and transition in the corresponding LGSPN to model the synchronization (see Figure 6.1, cases 3.a, 3.c and 3.e).

To translate a call state, steps to follow are similar to those described in section 6.1. Given a call state CS,

- If it verifies $S.entry.IsAsynchronous = false$ (i.e., its associated CallAction is a synchronous call) we will 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 it verifies $S.entry.IsAsynchronous = true$ (i.e., its associated CallAction is an asynchronous call) we will define u' as the number of outgoing transitions

OA_j of the state (which do not end in a join pseudostate), v' the number of outgoing transitions OJA_l that end in a join pseudostate, and w' the number of self-loop transitions SA_n.

Also let evx be an event generated by the call action, $evx = S.entry.operation{\rightarrow}ocurrence$. Considering this, we have one LGSPN for each outgoing or self-loop transition t, $\mathcal{LS}^t_{CS} = (S^t_{CS}, \psi^t_{CS}, \lambda^t_{CS})$, as shown in Figure 6.1, cases 3.a-3.f. Therefore, we have either $u+v+w$ or $u'+v'+w'$ LGSPN models that need to be combined to get a model of the state CS, $\mathcal{LS}_{CS} = (S_{CS}, \psi_{CS}, \lambda_{CS})$. The LGSPNs corresponding to each set of kind of transitions are now obtained by superposition:

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

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

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

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

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

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

6.2.4 Decisions

Decisions are preprocessed before the AD translation, as it will be mentioned in section 6.3.1. They are substituted by equivalent outgoing transitions on action states (as shown in Figure 6.2), preserving the properties inherent in performance annotations. Therefore, they do not have to be translated.

6.2.5 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.

The translation of a merge pseudostate M depends on the kind of target element of its outgoing transition. Figure 6.3 (cases 5.a and 5.b) shows the direct translation of the model $\mathcal{LS}_M$, according to the condition expressed below.

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

6.2.6 Concurrency support items

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

Given a join pseudostate J, it is translated into the labelled system $\mathcal{LS}_{J}$, shown in Figure 6.3, 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 6.3, we have a basic LSGPN $\mathcal{LS}^{B}_{F} = (S^{B}_{F}, \psi^{B}_{F}, \lambda^{B}_{F})$ (case 4.a in the Figure) and one LGSPN (case 4.b) for each outgoing transition t, $\mathcal{LS}^{t}_{F} = (S^{t}_{F}, \psi^{t}_{F}, \lambda^{t}_{F})$. 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 obtained by superposition:

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

- And the final LGSPN $\mathcal{LS}_{F}$ is composed following the expression:

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

6.2.7 Initial and final states

Initial pseudostates and final states are elements inherited from UML state machines semantics. However, unlike it happened on UML state machines (cfr. chapter 4) the initial pseudostate is not translated into an LGSPN model when translating an AG, 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 chapter 4.

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 6.3, case 6.a.

6.2.8 Signal sending and signal receipt

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

As these symbols are control icons, there is not a metaclass corresponding to these elements in UML metamodel. Therefore, 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 6.3, 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 6.3, 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 by adding new places in the LGSPN to consider that possibility as well.

6.3 The system translation process

In the previous section we have presented our method to translate every AG 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 AGs and the way to integrate the resulting LGSPN with the ones obtained from the translation of UML state machines in chapter 5.

6.3.1 Translating activity diagrams into GSPN

As an initial premise we assume that every AG 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 AG can then be divided in three phases, which are presented in the subsequent paragraphs.

Pre-transformations

Before translating the AG into an LGSPN model, we need to apply some simplifications to the diagram in order to properly use the translations given in section 6.2. These simplifications are merely syntactical so the system behaviour is not altered. Most relevant ones are:

- Suppression of decisions. Figure 6.2 shows a particular case of this kind of transformation. New decisions could be found in any branch of the chaining tree, but the Figure has been simplified for the sake of simplicity.
- Suppression of merges / forks / joins chaining, bringing them together into a unique merge / fork / join pseudostate.
- Deducting and making explicit the implicit control flow in action-object flow relationships, where aplicable.
- Avoidance of bad design cases (e.g., when the target of a fork pseudostate is a join pseudostate).

Translation process

Once pre-transformations have been 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 6.1.

step 2 Superposition of the LGSPNs corresponding to the whole set of each kind of diagram elements:

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

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

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

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

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

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

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

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

Post-optimizations

Contrasting with pre-transformations, which are mandatory, post-optimizations are optional. Their objective is just to eliminate some spare places and transitions in the resulting LGSPN so as to make it more attractive without altering its semantics. One example of this 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 and input places) in case of no conflict.

6.3.2 Composing the whole system

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

Let d be the number of AGs 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 AGs and $Lend\_AG^P$ is the set of final places of those nets. Now, we can merge the referred LGSPNs by superposition (of places):

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

Now let $\mathcal{LS}''_{sc}$ be the LGSPN corresponding to the translation of the set of SCs in the model. $\mathcal{LS}''_{sc}$ was previously obtained by composition (superposition of places) of the nets obtained for each SC and subsequent removal of sink *acknowledge* places (cfr. chapter 4).

Then let T_act be the set of transitions in $\mathcal{LS}''_{sc}$ labelled *activity* (cfr. chapter 4) which represent activities that are described with activity graphs. $\mathcal{LS}_{sc}$ is 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 graph AG' that represents the activity, $top = AG'.top.name$. Similarly, outgoing places (labelled $compl\_A$) will be now labelled end_AG'.

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

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

The resulting net $\mathcal{LS}_{sc-ad}$ often represents the whole system behavior.

A sample case of the translation of a very simple system is illustrated in Figure 6.4. A state machine and an AG models for the system are presented on the left side of the Figure. We have obviated some diagrams of the system description so only part of the resulting LGSPN is included on its right side. That fact results in the lack of tokens in the initial marking of the net.

The state machine in Figure 6.4 represents the life-cycle of an object from the class *car wash machine*, that can be either working or inactive (i.e, waiting for a new car to be washed). The activity performed by the machine when it is working is described by the AG below. As it is shown, the machine works in a different way depending on the amount of money spent by the driver, and can do some tasks simultaneously.

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

6.4 Conclusions

The main contributions of this chapter can be summarized as follows. We have given a translation of the activity graph (that models a doActivity) into a labelled generalized stochastic Petri net model. In this way, it can be composed with any other labelled generalized stochastic Petri net model that represents a state machine that uses the doActivity. A formal semantics for the activity graphs is achieved in terms of stochastic Petri nets. They allow to check logical properties as well as to compute performance indices. Obviously, this formal semantics represents a particular interpretation of the "informally" defined concepts of the UML activity graphs package. Our interpretation is focused on the basis that the activity diagram is meant for the description of the doActivities in a state machine. A very simple example of the application of the proposed translation has been developed, it models a car wash machine.

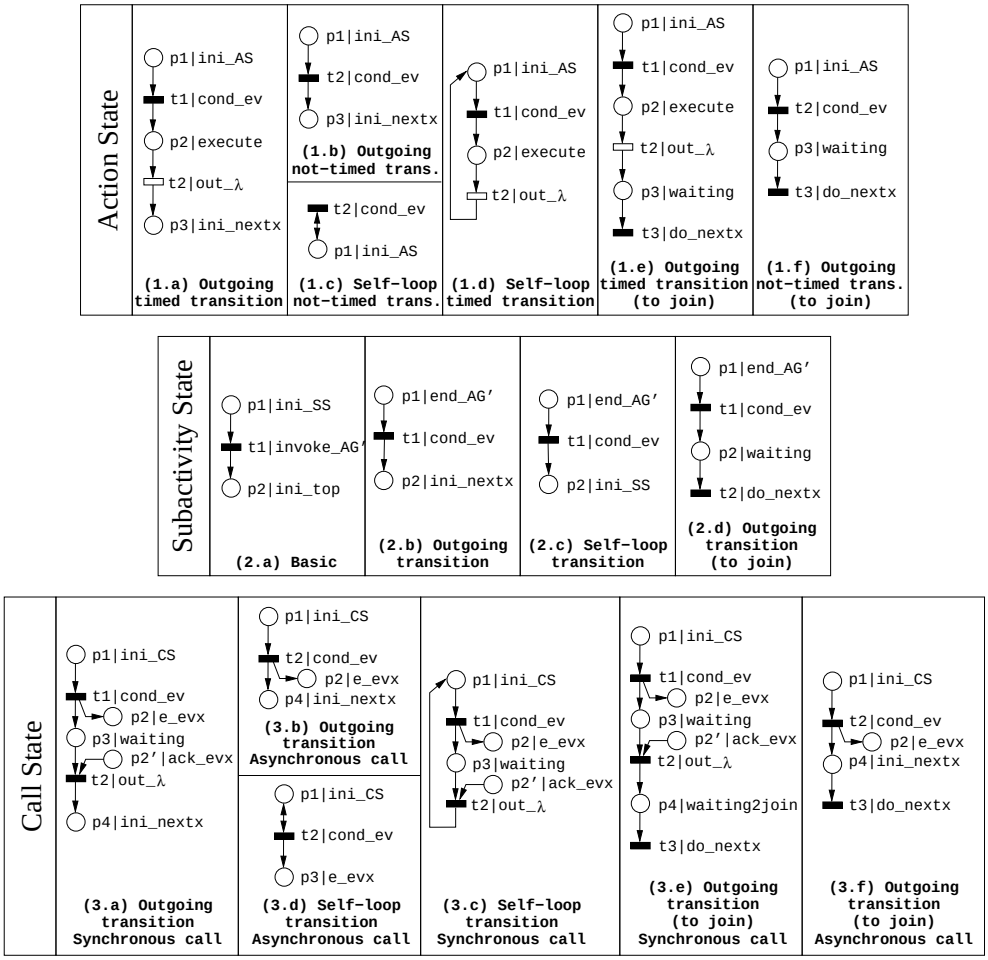


Figure 6.1: Action, subactivity and call states to LGSPN

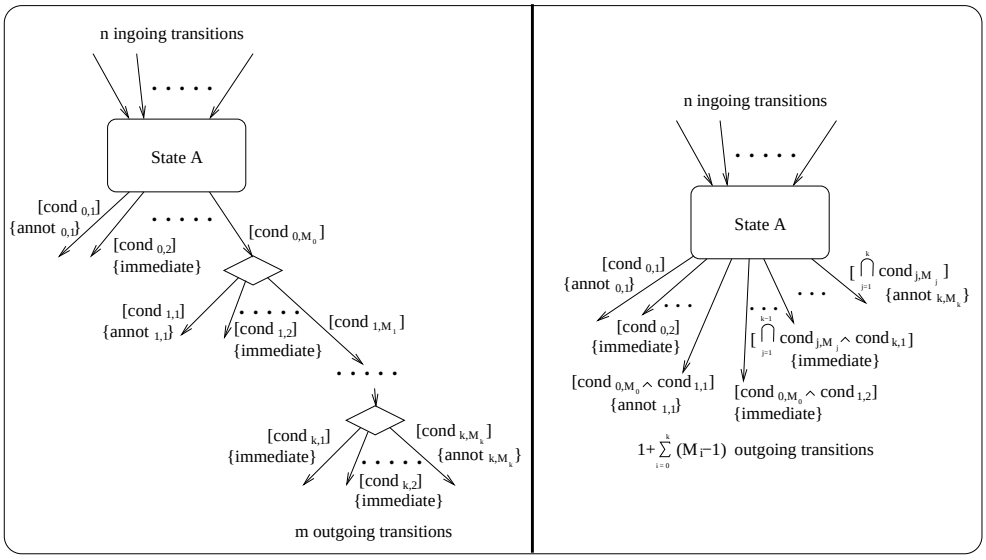


Figure 6.2: Decision to LGSPN (pre-transformation)

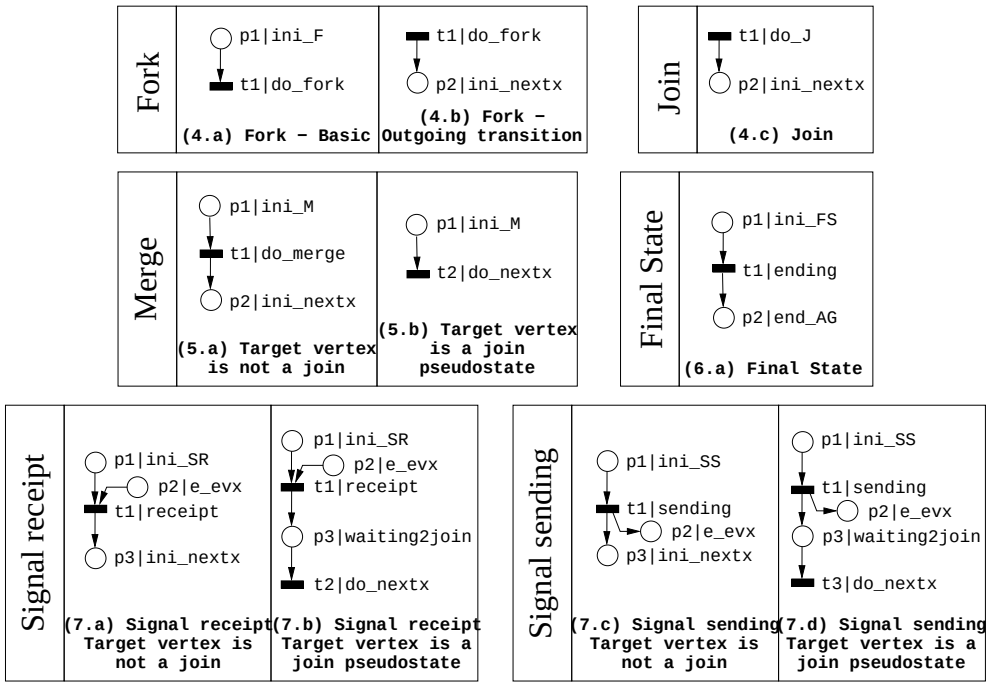


Figure 6.3: Fork, Join, Merge, Final State, Signal Sending and Signal Receipt to LGSPN

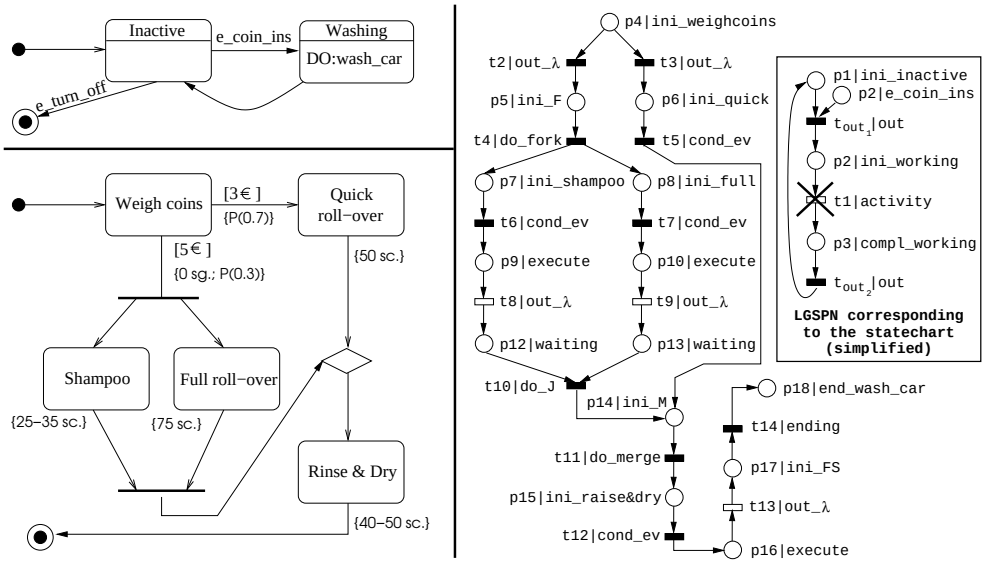


Figure 6.4: Car wash machine example

Chapter 7

Software Performance Process

Once the UML diagrams related with performance modeling have been identified and a notation to describe system load and routing rates has been introduced, it seems necessary to study how these diagrams should be used to obtain a performance model from the information they contain.

In this chapter we present a process to obtain performance models that represent software systems and we explore how to evaluate performance parameters from them. These issues have been developed in [MCM00b, MCM01b]. The process makes use of the notation given in chapter 3 to model the system and also it profits from the translation of these diagrams into stochastic Petri nets introduced in chapters 4 and 5. As we will see, an interesting characteristic of the process is that the formal model, in terms of stochastic Petri nets, can be obtained semi-automatically inside the software development process from the pa-UML models. In this way, without much effort, the process allows to obtain a performance model as a by-product of the software life-cycle.

Moreover, to complement our approach of performance evaluation process, we present the ideas given in [MCM00a]. They explore the use of the "design patterns" [GHJV95] in the performance evaluation process.

The chapter is organized as follows. Section 7.1 presents an example of software system that will be used in later sections to model and evaluate performance aspects. The importance of the example comes from its implementation domain, mobile agents, it is a research area where performance aspects are of special interest. In section 7.2 we present our proposal to evaluate performance of software systems and in section 7.3 the complementary approach based on "design patterns". These two sections use the previously mentioned software system as a running example. Some conclusions are presented in section 7.4.

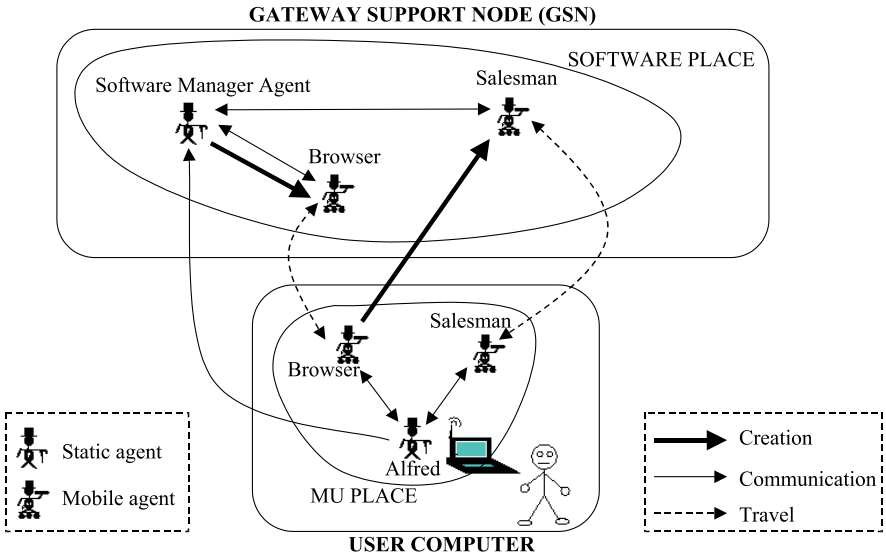


Figure 7.1: Architecture for the ANTARCTICA SRS.

7.1 Example: software retrieval service

Before to show our proposal of a process to evaluate performance of software systems, we present an example of software system. It will be used as a running example in the next section to illustrate the different steps that conform the process.

The example we are going to deal with belongs to the ANTARCTICA[1] system [VGGI98, GIM+01] that has been developed by the Interoperable Database Group [Gru].The ANTARCTICA system has been designed to provide mobile computer users with different software services that enhance the capabilities of their computers. The use of the mobile agents [HCK97, PS98, KRR98] technology plays a prominent role in the ANTARCTICA system.

Software design and implementation using mobile agents are nowadays involved in a skepticism halo. There are researchers who question its utility because it could be a new technology that does provide new skills but it could introduce new problems, such as the inappropriate use of the net resources. Security and performance are the most critical aspects for this new kind of software. Therefore the design of software systems that make use of mobile agents technology is of special importance for us since the study of performance in this research field becomes crucial.

From the different services developed in the ANTARCTICA project we concentrate on the Software Retrieval Service [MIG00b, MIG00a, MIG00c], ANTARCTICA SRS. The goal of the ANTARCTICA SRS is to provide mobile computer users with a service to select and download software in an easy and efficient way. Efficient be-

[1] Autonomous ageNT bAsed aRChitecture for cusTomized mobIle Computing Assistance.

cause the system optimizes battery consumption and wireless communication costs. Then the aim of this service is to propose an alternative method to the current web-based software retrieval systems like [Inc99d, Inc99a, Inc99b, Inc99c]. In chapter 8 a performance comparison among the ANTARCTICA SRS and an approach that is not based on agents will be presented. The ANTARCTICA SRS provides several interesting features:

- The system manages the knowledge needed to retrieve software without user intervention, using an ontology.
- The location and access method to remote software is transparent to users.
- There is a "catalog" browsing feature to help user in software selection.
- The system maintains up to date the information related to the available software.

Some of the advantages of the use of mobile agents, related to accessing remote information, are the following:

- They encapsulate communication protocols.
- They do not need synchronous remote communications to work.
- They can act in an autonomous way and carry knowledge to perform local interactions at the server system instead of performing several remote procedure calls.
- They can make use of remote facilities and perform specific activities at different locations.

System description and modeling assumptions

The ANTARCTICA SRS is situated in a concrete server called the GSN[2]. Agents are executed in contexts denominated *places* [MBB+98]. Mobile agents can travel from one place to another. The service incorporates two places: one place on the user computer called the *Mobile User place*, and other situated on the GSN, called the *Software place* (see Figure 7.1).

The procedure that the ANTARCTICA SRS supports for the software retrieval process is the following: the user sends requests for software to an agent (*Alfred*). The request is sent to the GSN and an agent (the *browser*) is created. The user receives the visit of the browser, which helps her/him to select the most appropriate software by browsing a catalog customized to that concrete user. The user can request more detailed information until s/he finally selects a piece of software. Then a new agent arrives to the user computer (the *salesman*) with the selected piece of software.
In the following such agents are described, grouped in two categories:

[2]The *Gateway Support Node* is the proxy that provides services to computer users.

1. *The user agent.* Alfred is an efficient majordomo that serves the user and is in charge of storing as much information about the user computer, and the user her/himself, as possible.

2. *Information exploitation.* The software manager agent creates and provides the browser agent with a catalog of the available software, according to the needs expressed by Alfred (on behalf of the user), i.e., it is capable to obtain customized metadata about the underlying software. For this task, the software manager consults an ontology. The software itself can be either stored locally on the GSN or accessible through the web in external data sources. Thus, the GSN can have access to a great number of distinct software for different systems, with different availability, purpose, etc. The goal of the browser agent is to interact with the user in order to refine a catalog of software until the user finally chooses a concrete piece of software. When this is done, the salesman agent carries the program selected by the user to her/his computer, performs any electronic commerce interaction needed (which depends on the concrete piece of software), and installs the program, whenever possible.

The following modeling assumptions must be taken into account:

- It must be considered that the user spends some time reading the catalog presented by Alfred, an exponentially distributed random variable with rate $\lambda_{observe}$ ($\lambda_{observe}$ is obtained as the inverse of the time in seconds) will be used to model several kinds of users.

- The number of catalogs that the user must navigate until s/he finds the software is difficult to estimate (it depends on her/his experience), the probability that the user finds the software by selecting n catalogs models different kind of users, from naive users, those who need to visit many catalogs to find the software, to expert users, those who find the software visiting very few catalogs.

- Whenever the user requests a catalog or a concrete piece of software, the software manager consumes time consulting an ontology to create the catalog or to find the piece of software. These activities are modeled by variables with rates $\lambda_{findCat}$ and $\lambda_{findFile}$

- Some of the messages sent among the browser, the software manager, the salesman and Alfred travel through the net. A variable with rate λ_{m_i} models the time spent by the message i navigating through the net. Notice that local messages do not consume net resources.

- As the browser is an intelligent agent, sometimes it does not ask for information to the software manager, but if it does it, then it will performed using remote procedure call (RPC) or traveling through the net; the size of the catalog could be parameterized.

The ANTARCTICA SRS was proposed in [MIG00b] using different technologies, namely CORBA [Obj99], HTTP and mobile agents. Some performance tests were

applied to different implementations, in order to select the best way of accessing remote software. Conclusions were the following:

- Time corresponding to CORBA and mobile implementations are almost identical for a wide range of files to be downloaded.
- Mobile agent approaches are fast enough to compete with client/server approach.

Although considering the importance and the relevance of the results of the work [MIG00b], we would like to stress the enormous cost of implementing different prototypes in order to evaluate the performance of the different alternatives.

7.2 A process to evaluate performance of software systems

In chapter 2 we analyzed the requirements that a software performance process should satisfy. In this section we present our proposal to evaluate performance of software systems. It tries to be compliant with the principles given in chapter 2.

"Large software systems are the most complex artifacts of human civilization" [Bro87], even more if they are distributed systems, the kind of systems for which we are interested to evaluate performance aspects. In our opinion complex systems need formal models to be properly modeled and analyzed, also if the kind of analysis to accomplish is with performance evaluation purposes. Then the use of formalisms in our proposal is a must.

Two kind of techniques based on models have been proposed and largely discussed in the literature to evaluate performance: simulation and analytic techniques [Jai91]. But there is a gap in the connection of these techniques with the classical proposals for software development [You89, PJ80, RBP+91, JCJO92, CAB+94, Mes00] or more recent design techniques such as design patterns [GHJV95]. Neither of these proposals for software development deal with performance analysis, at most they propose to describe some kind of performance requirements. So, it could be argued that there does not exist an accepted *process* to model and study system performance in the software development process. Moreover, this lack implies, as it was identified in chapter 3, that there is not a well-defined language or *notation* to annotate system load, system delays and routing rates in the context of software modeling. Nevertheless, in the last years a research community has emerged concerned with this kind of problems as it was commented in chapter 2 when the state of the art was analyzed. Works that are not based in the UML language also address the problem of the software performance process, such as [MJP02].

We consider that a proposal to evaluate performance in software systems must accomplish with both, the notation and the process. The notation should be expressive enough to identify all the relevant performance parameters in the system but introducing the minimal set of elements to describe them. In our opinion, the

performance notation should be part in some sense of the notation used by the software designer to model software systems, then trying to avoid the introduction of new models that represent the load orthogonally to those that describe the dynamics and the functionality of the system. The notation that we propose to be used in the process is pa-UML, the extension given in chapter 3 to deal with performance features at the design stage. Remember that we identify UML as a de facto standard among the software engineering community, but unfortunately it lacks of the necessary expressiveness to accurately describe the system load, which is needed to obtain performance figures. To bridge the gap, pa-UML was introduced.

The process should propose the steps to follow to obtain a performance model and it should give trends to analyze this model. In our opinion, a process to evaluate software performance should be "more or less transparent" for the software designer. By "transparent" we mean that the software designer should be concerned as less as possible to learn new processes since the analysis and design task already implies the use of a process. Therefore, ideally the process to evaluate performance of software systems should not exist, it should be integrated in the daily practices of the software engineer. We have tried that our proposal of process can be used together with any software life-cycle, and the performance model can be semi-automatically obtained as a "by-product" of the software life-cycle. Obviously, the proposal is not absolutely "transparent" for the software engineer but it goes in this direction. Another requirement for the process is that the performance model should be obtained in the early stages of the software life-cycle. In this way proper actions to solve performance problems take less effort and less economical impact.

Concerning the steps of the process, we propose the following ones:

1. Model software requirements using the desired software life cycle paradigm, meaning statecharts the description of the behavior of the active classes of the system, meaning activity diagrams the refinement of the activities in the statecharts and the sequence diagrams concrete executions of interest in the context of the system. While developing the models, the pa-UML proposal must be used to describe performance requirements or parameters.

2. Apply the translation functions and compositions rules given in chapters 4, 5 and 6 to the pa-UML models to obtain a performance model in terms of stochastic Petri nets. The functions and compositions rules can be used in two different ways depending if it is desired to obtain a performance model for the whole system or for a concrete execution, section 7.2.2 gives the steps to follow for both approaches. Translation and composition rules can be embeded in an augmented CASE tool.

3. Define the parameters to be computed and the experiments to test the system. Apply (analytical or simulation) techniques to solve the formal model. This phase can be made easier by integrating the augmented CASE tool with stochastic Petri net analysers.

In the following, some aspects of the previous steps are detailed. In the first step the UML diagrams that must be taken into account are at least those included in the

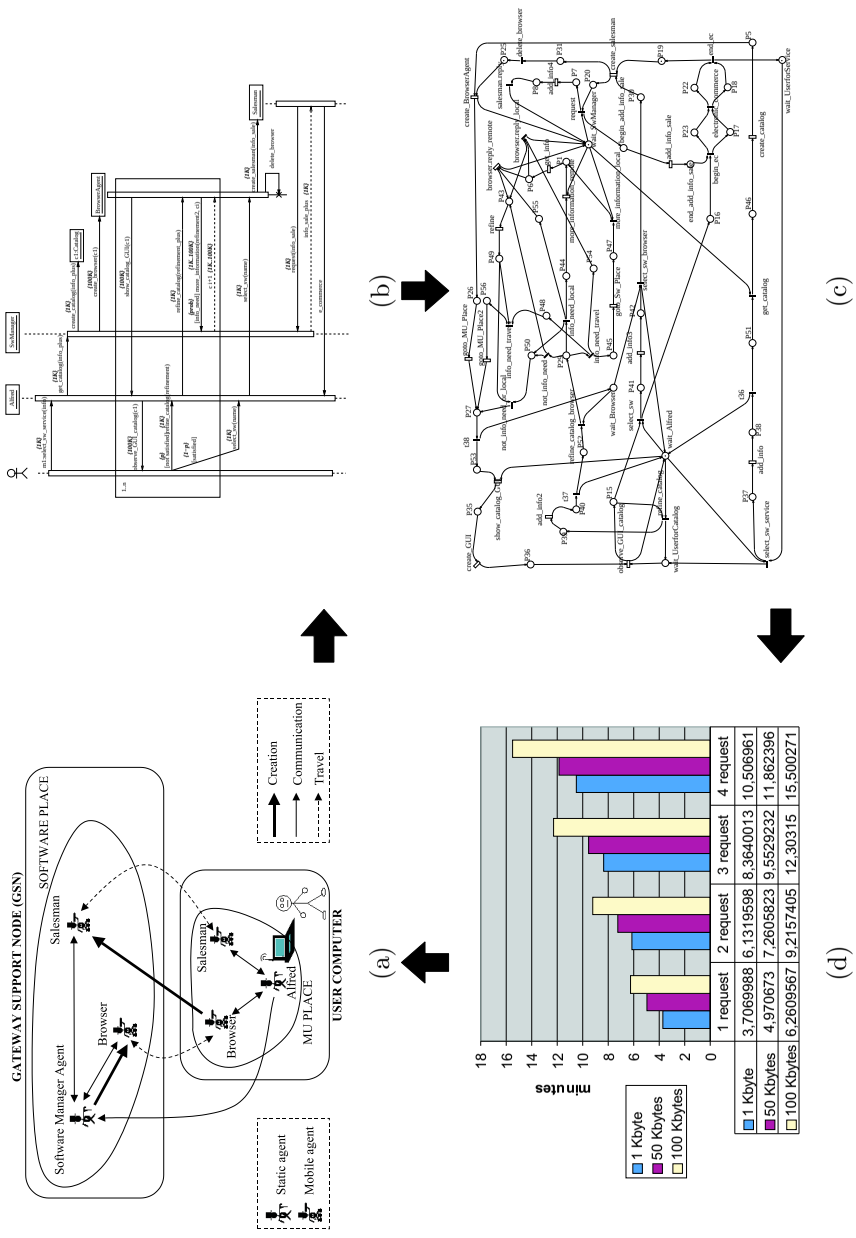


Figure 7.2: An overview of the process

pa-UML proposal: the use case diagram, the statechart diagram, the activity diagram and the sequence diagram which allow to model a wide range of distributed software systems driven by events. Implementation diagrams have not been considered yet in the process, but as explained in chapter 9 they can be useful to model system resources, then avoiding the unlimited resources assumption made in our work. A set of modeling assumptions that parametrize the system should be introduced, see section 7.1 for different modeling assumptions in the context of the ANTARCTICA SRS.

Once the pa-UML models have been developed, it should be nice to obtain performance indices for the modeled system from them. As UML lacks of the necessary formal semantics to apply (simulation or analytical) techniques to obtain performance figures, we need to provide the language with it. We use Petri nets with this purpose, Petri nets are a widely used formal paradigm for the modeling and validation of concurrent systems and provided with a stochastic time interpretation [AMBC+95], they are suitable for performance evaluation. Therefore in the second step, we propose an automatic translation from the pa-UML diagrams into stochastic Petri nets or what it is the same to provide pa-UML diagrams with a unique and consistent interpretation. This interpretation avoids ambiguities introduced by UML in some specification aspects, for example concurrency, which is fundamental for performance evaluation. From stochastic Petri nets models, performance indices may be computed by applying techniques already developed in the literature. In this way, our proposal can deal with simulation and analytic techniques since stochastic Petri nets are a formalism that admit them, nevertheless the examples developed in our works have explored only the second ones. The techniques that we will use in our examples to analyze the Petri nets models are those implemented in the *GreatSPN* tool [CFGR95].

Finally, in the third step performance parameters to be computed must be identified (througput, response time, delays) togheter with the transitions and/or places in the stochastic Petri net that are involved in their computation. By asigning different values to the parameters identified in the modeling assumptions proposed for the system in the first step, the software analyst can obtain the testbed to be aplied to the performance model. To modify a parameter of the model it must be taken into account that each one will be related with a place and/or a transition or with a set of places and/or transitions. Then to assign a value to a parameter of the model consists in the addition of tokens to a place/s, to modify the probability of an immediate transition or to change the rate parameter of a timed transition.

We think that an important issue of the approach is that the obtention of a formal performance model does not imply an additional effort for the software engineer (aside to describe the performance requirements using pa-UML) since it is obtained as a "by-product" of the software life-cycle. Therefore the software engineer does not need to know how to model complex systems using stochastic Petri nets since the nets that represents the statatecharts and the activity diagrams, *component* nets, are obtained automatically by applying the formalization given in chapters 4 and 5. After, by means of a set of rules, given in section 7.2.2, the sequence diagram will be used to compose the component nets, obtaining a net that represents the performance model.

This net will be marked and some of its transitions modified according the third step, gaining the performance model.

Another advantage of the approach is that it can be integrated in any CASE tool that supports UML notation. The pa-UML annotations should be introduced as tagged values as explained in chapter 3. The stochastic Petri nets can be automatically obtained by implementing the formalization given in chapters 4 and 5 and the rules in section 7.2.2. We suggest that the implementation of the translation takes as input model the UML model in XMI [OMG99] format, the standard metadata exchange format, since the most case tools try to be compliant with it, then an added value is gained.

Figure 7.2 shows graphically the steps of the proposed approach. The interest of the figure is to show a general view of the approach, it is not of interest the concrete information it contains, therefore it does not matter that the details are not legible. Figure 7.2(a) represents the problem domain (any software system), Figure 7.2(b) means that the software engineer has modeled the UML diagrams for the system using the performance annotations proposed in pa-UML (step 1), Figure 7.2(c) reflects that the formal model in terms of stochastic Petri nets has been obtained (steps 2 and 3), Figure 7.2(d) is an abstraction of the performance results (step 3) obtained for the system that will be used by the software engineer to take decisions in the problem domain in order to obtain the best configuration. Therefore, the approach allows the software engineer to perform what-if analysis, for example in the ANTARCTICA SRS: What if the service is attended by two majordomos?, what if there exist five or fifty users connected to the system?.

In the following subsections the steps of our approach are applied to the example presented in section 7.1. Section 7.2.1 is devoted to the first step, the modeling using pa-UML, it will let us to remember the role of each diagram in the performance process. The section 7.2.2 details the second step of the process explaining the two different ways to obtain a performace model but applying only one of the proposals to two different cases. Finally, step three is accomplished in section 7.2.3. How to perform the analysis for the ANTARCTICA SRS is explained as a guide for the analysis of others systems.

7.2.1 Modeling the system using pa-UML

In this section we address the first step of the approach, also the pa-UML models for the running example are going to be developed.

The first step proposes to model software requirements using the desired software life cycle paradigm with UML notation and the performance annotation given in the pa-UML proposal. We suggest that the first pa-UML diagram to model should be the use case diagram, in this way the scenarios (use cases) of interest for the system are specified. As it was mention in chapter 3 for each use case that describes (a part of) the system, the probability of its usage by each actor must be annotated. Each use case j with probability to occur must be described by a sequence diagram with performance purposes; probability to occur means that there exist at least one actor

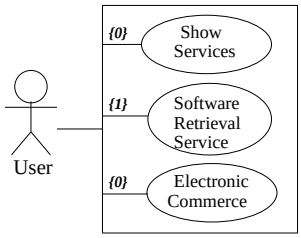


Figure 7.3: Use Cases

i in the system such that its probability to make use of the use case P_{ij} is greater than zero. Each sequence diagram in the previous set will be modeled taking into account that it represents a concrete execution of interest in the context of the system by means of the exchange of messages among objects. Therefore, they are useful from the performance evaluation point of view only when figures for the particular execution they represent are of interest. If it is desired to obtain figures for the behavior of the whole system then sequence diagrams are not necessary. The figures can be obtained from the information available in the statecharts. In the sequence diagrams the routing rates for the messages and its size will be annotated. After, a statechart diagram for each active class in the system should be modeled, meaning its behavior. Each statechart will be annotated with the duration of the activities, the routing rates for the messages and the size of them. Last, if it is of interest to detail some of the activities in the statecharts, then an activity diagram for each activity will be modeled. They will be annotated with the duration of the fine grain activities and the probabilities in the transitions.

The models developed following this first step do not express concurrency satisfactorily. For example, in the running example that we are treating we have not decided a priori how many requests should attend a majordomo. Moreover, we cannot parametrize the system to answer questions such as how many concurrent users can use the system, etc. These reasons, among the previously explained, provoke the necessity to obtain a formal model that solves the ambiguities. The next step (section 7.2.2 proposes the translation of the models obtained in this step into stochastic Petri nets.

Use Cases

Figure 7.3 shows the use cases needed to describe the dynamic behavior of the ANTARCTICA SRS system and the unique actor that interacts with the system, the "user". The system deals with three different use cases: "show services", "software retrieval service" and "e-commerce". Among the uses cases that describe the system, it is of interest to evaluate performance figures only for the software retrieval service use case, since it is the unique that has an association link with an actor with probability to be used, concretely 1. The use cases are described in the following.
Show services use case description.

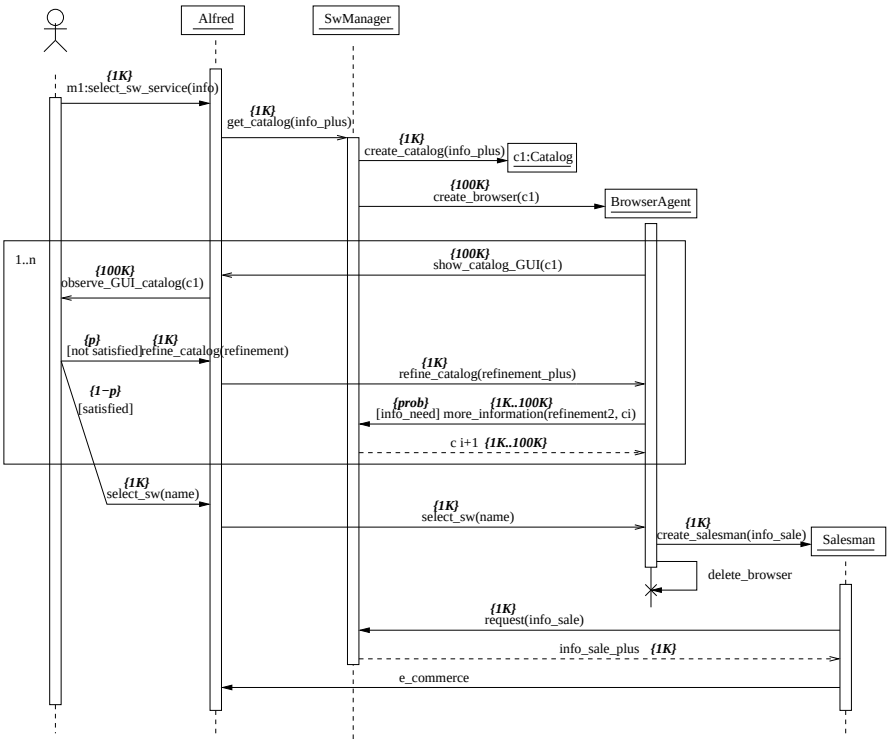


Figure 7.4: Sequence diagram for the Software Retrieval Service use case

- Principal event flow: the use case goal is to show to the user the available services that the system offers. The Software Retrieval Service is one of those services and it is also described as a use case.
- Probability to be executed by the user: 0 (since it is not of interest to evaluate performace figures in this use case).

Software retrieval service use case description.

- Principal event flow: the user requests the system for the desired software. The browser gets a catalog and the majordomo, Alfred, shows it to the user, who selects the software s/he needs.
- Exceptional event flow: if the user is not satisfied with the catalog presented, s/he can ask for a refinement. This process could be repeated as many times as necessary until the user selects a concrete piece of software.
- Probability to be executed by the user: 1.

Electronic commerce use case description.

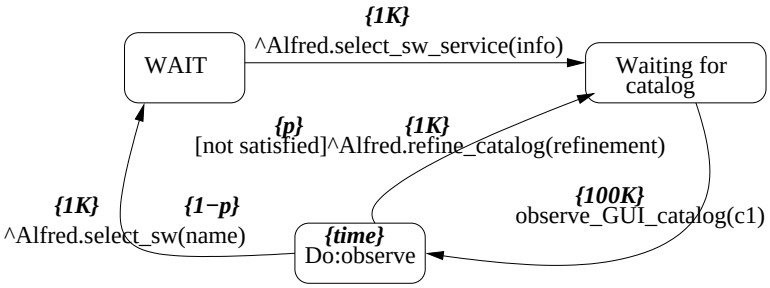


Figure 7.5: Statechart diagram for the user

- Principal event flow: the goal is to provide the user an e-commerce activity and the download of the software selected in the software retrieval service use case.
- Probability to be executed by the user: 0.

Sequence diagrams

In a sequence diagram two kind of messages were distinguished: Those that do not consume time and those that do. In a mobile agent system, messages sent among objects on the same computer are considered of the first kind and messages sent among objects on different computers as the second kind.

The annotations proposed in chapter 3 to model the load of the messages and its routing rates are used in the sequence diagram that models the running example, see Figure 7.4. For an example of the first kind of annotations see message m1 that is labeled with {1 Kbyte}. For this message the annotation is interpreted as a tagged value in the following way, TaggedValue.name = performance annotation, TaggedValue.dataValue = {1 Kbyte}, TaggedValue.referencedValue = m1. The second kind of annotations is illustrated by the message that has associated the refine_catalog event whose probability of success is equal to {1-p}.

Statechart diagrams

The statecharts modeled for the ANTARCTICA SRS (Figures 7.5 to 7.9) show the guards attached to the transitions, but only with documentary purposes since they will not be translated to a guard language in the formalization proposed in chapters 4 and 5.

Three kinds of annotations were proposed for the statecharts in chapter 3: the duration of the activities, the size of the messages and the routing rates. As an example of the first kind see label {time} associated to the activity observe in Figure 7.5. This annotation means a variable that will allow to parametrize the system to study it taking into account users that spend different amounts of time observing a catalog. The annotation is interpreted as follows, TaggedValue.name = performance annota-

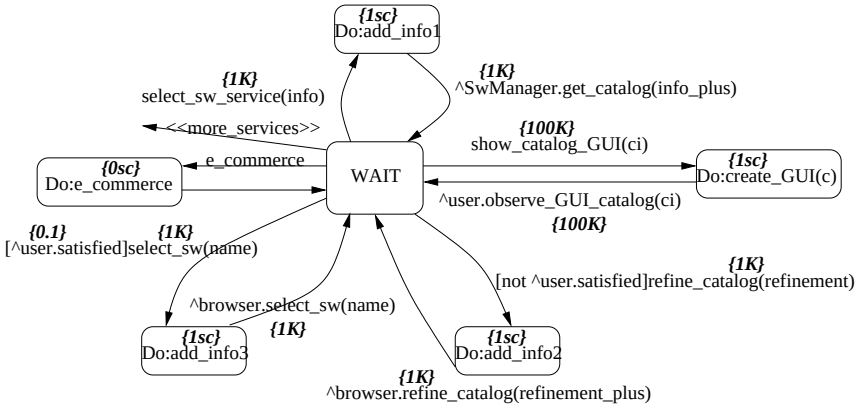


Figure 7.6: Statechart diagram for Alfred

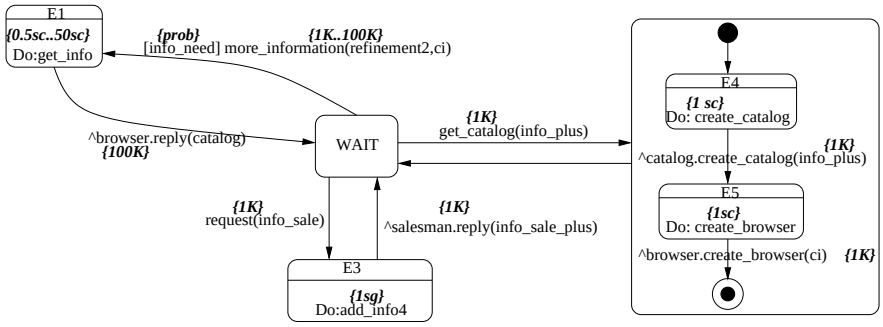


Figure 7.7: Statechart diagram for the Software Manager

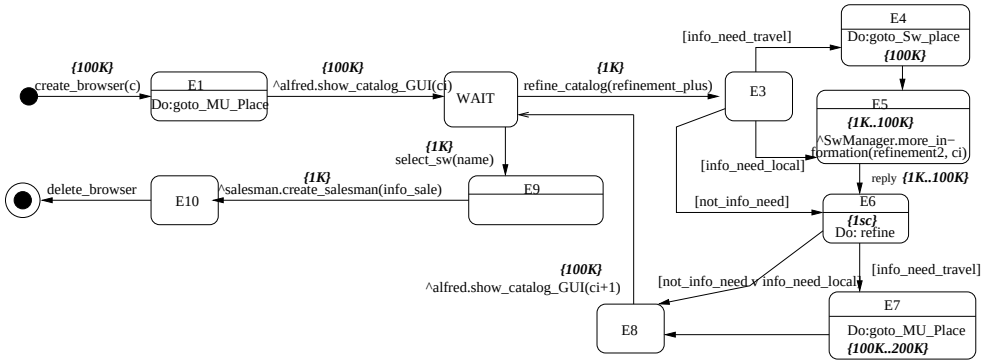


Figure 7.8: Statechart diagram for the Browser

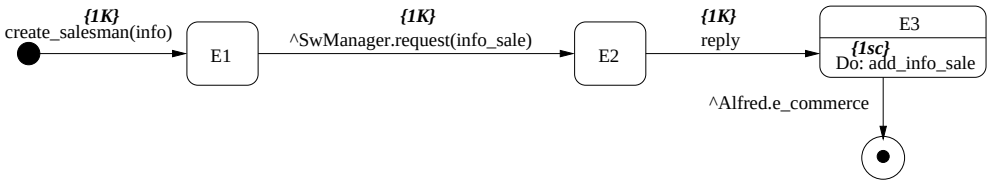


Figure 7.9: Statechart diagram for the Salesman

tion, TaggedValue.dataValue = {time}, TaggedValue.referencedValue = act, act.name = observe.

We now present the statechart diagrams for our example using the pa-UML notation.

User statechart diagram. In Figure 7.5 the behavior of a user is represented. The user is in the wait state until s/he activates a select_sw_service event. This event sets the user in the waiting_for_catalog state. The observe_GUI_catalog event, sent by Alfred, allows the user to examine the catalog to look for the desired software. If it is in the catalog, the users selects the select_sw event, otherwise s/he selects the refine_catalog event.

Alfred statechart diagram. The example supposes that Alfred is always present in the system, no creation event is relevant for our purposes (Figure 7.6). Alfred's behavior is typical for a server object behavior. It waits for an event requesting a service (select_sw_service, show_catalog_GUI, refine_catalog or select_sw). For each of these requests it performs a concrete action, and when it is completed, a message is sent to the corresponding object in order to complete the task. After the message is sent, Alfred returns to its wait state to serve another request. The stereotyped transition $\ll$ *more_services* $\gg$ means that Alfred may attend other services that are not of interest here.

Software manager statechart diagram. Like Alfred, the Software Manager behaves as an server object. It is waiting for a request event (more_information, get_catalog, request) to enable the actions to accomplish the task. Figure 7.7 shows its statechart diagram; it is interesting to note the actions performed to respond the get_catalog request. First, an ontology is consulted and, after that, two different objects are created, those involved in task management.

Browser statechart diagram. The statechart diagram in Figure 7.8 describes the Browser's life. It is as follows: once the Browser is created it must go to the MU_Place, where it invokes Alfred's shows_catalog_GUI method to visualize the previously obtained catalog. At this state it can attend two different events, refine_catalog or select_sw. If the first event occurs there are two different possibilities: first, if the Browser has the necessary knowledge to solve the task, a refinement action is directly performed; second, if it currently has not this background, the Browser must obtain

information from the Software Manager, by sending a more_information request or by traveling to the software place. If the select_sw event occurs, the Browser must create a Salesman instance and die.

Salesman statechart diagram. The Salesman's goal is to give e-commerce services, as we can see in Figure 7.9. After its creation it asks the Software Manager for sale information. With this information the e-commerce can start. This is a complex task that must be described with its own use case and sequence diagram and it is not developed in this work.

7.2.2 Modeling with Petri nets

At this point, we have modeled the ANTARCTICA SRS system with pa-UML notation, taking into account the load of the system in the use cases, sequence diagrams and the statechart diagrams. So, a pragmatic model of the system has been obtained. But this representation is not precise enough to express the requirements proposed for the system since it is desired to predict the system behavior in different ways. First, we want to study how the system works with only one user served by one majordomo. On the other hand, it is also of our interest to know the system behavior when several users are served by only one majordomo, or by several majordomos. In order to obtain answers to these questions, performance (analytical or simulation) techniques should be applied to the developed pa-UML diagrams, but as we have explained, there is a lack in this field because no performance model exist for UML. Moreover, the pragmatic model is not expressive enough to describe system concurrency as needed to represent for example the existence of several majordomos (since UML models concurrency in a very poor way). To solve these lacks, our proposal chooses stochastic Petri nets as formal model, because it has the remarked capabilities and also there are well-known techniques to analyze system performance of such models models. Thus, without increasing the modeling effort, it could be possible to avoid the necessity of implementing the system for predicting performance figures.

Two different approaches to create a final analyzable model of the system in terms of stochastic Petri nets can be devised. The first approach will be used when performance figures for the whole system must be computed and the second one when performance figures for a concrete execution of the system must be obtained.

First approach

The method to create a stochastic Petri net that represents the behavior of the whole system (i.e. all the possible execution paths in the system) is formalized in chapters 4 and 5 for the case of "flat" state machines and "non-flat" state machines respectively. In the following we describe informally the proposed strategy, considering that there do not exist activity diagrams. If there exist activity diagrams we refer to chapter 6.

- First, for each statechart a "component" net (a stochastic Petri net) is obtained by applying the definitions in chapters 4 and 5 to each element (action, activity,

transition, ...) of the statechart.

- When each component net has been obtained they must be composed to create another stochastic Petri net that represents the whole system, the "system" net. The composition is performed by applying Definition 4.54 for the case of "flat" state machines and following the steps in section 5.9 for "non-flat" state machines. But this net cannot be analyzed yet since the issues in the following steps must be considered:

1 If the analysis will be performed in the steady-tate, then it must be guaranteed that the "system" net is live and cyclic. To achieve it, each component net must be live. A component net is live when its corresponding statechart has not sink states, i.e. each state can be reached from any state. For example, nets that do not agree with this requirement are those obtained from a statechart with final state at the top most level. In this case, a transition must be created with an input arc from the place that represents the final state of the statechart and an output arc to the place that represents the initial state of the statechart. As an example we are going to describe how to perform it in the component Petri net for the salesman, see Figure 7.12. It must be created a new transition with an arc from place P157|ini_f, that represents the final state of the salesman, and an arc to place P132|ini_ps, that represents the initial state. The addition of this new transition and arcs means in the statechart that when an object (resource) is destroyed, immediately a new one of the same kind is created. It must be remembered that out proposal assumes the hypothesis of "infinite resources" in the system. Obviously, these changes must be realized in the component net when it belongs to the "system" net, but we have illustrated them over an isolated component net for simplicity.

   As a conclusion, statecharts with sink places cannot be analyzed, they mean that the system cannot evolution at a certain point.

2 The initial marking for the system net must be defined. The tokens should represent the population (resources) of the system. We suggest that the places that represent the initial pseudostates at the top most level in each statechart will be marked placing as many tokens as instances (resources) of the class should be considered. For those statecharts without initial pseudostate, a state of the diagram should be chosen for this purpose, then marking its corresponding place in the net.

   For the ANTARCTICA SRS users/requests are represented by tokens in the place P1|ini_wait, see Figure 7.10; instances of Alfred are represented by tokens in the place P30|ini_wait, see Figure 7.13; instances of software manager are represented by tokens in the place P164|ini_wait, see Figure 7.14 and finally salesmen are modeled by tokens in place P132|ini_ps in Figure 7.12.

   Since the resources of the system should be changed to represent different tests for the system, the initial marking can change for different experiments (one user and one majordomo), (one majordomo and several user, etc.).

3 For the transitions created by Definition 4.6, i.e., those that represent activity duration, it must be calculated the rate r_{do} of the exponentially distributed random variable that models its duration. As an example, see in Figure 7.7 the activity create_browser in the state E5 with a duration of {1 sec.}. The time is converted into msec., 1000 msec., and the inverse 0.001 is the rate of the transition T160|create_browser in the net of the Figure 7.14.

4 Some of the immediate transitions may be in conflict because they represent outgoing transitions exiting from the same state in a statechart either labeled with the same event or they are immediate outgoing transitions. A probability to occur must be assigned to each one of them. As an example, see in Figure 7.8 the immediate transitions exiting from the state E3. They are represented in Figure 7.11 by transitions t77|info_nt, t|info_nl and t79|not_info_n. Normally, these situations are related with a modeling assumption, in this case these transitions model assumption 4 in section 7.1.

5 For each message sent among remote objects, the time spent by the message traveling through the net must be modeled with a timed transition. For example, the message create_salesman sent by the browser to the salesman (see Figure 7.4) is modeled in the system net (see Figure 7.15) by a new transition T173 and a new place P196. The place is connected to the transition by an output arc. The place has an input arc from the transition t69|S_createS that represents the departure of the message and the new transition has an output arc to the place P134|e_createS that represents the arrival of the message. The arc that connects the transition t69|S_createS to the place P134|e_createS must be removed. The rate of the transition is calculated depending on the speed of the net and the size of the message. Being 1 Kbyte the size of the create_salesman message and 100 Kby/sec. the speed of the net, it will arrive in 0.01 seconds supposing that the net has not fails. Then, in this example the rate of the transition T173 will be 100. By modifying the speed of the net, several tests can be created.

   It could be argued that exponential assumption is not realistic for the modeling of network delays, and that heavy tailed distributions would be better. However, a performance model must many times lose in accuracy of the representation of reality in order to be able to be analyzed.

Therefore the information contained in the statecharts is enough to create a performance model, the "system" net, for the whole system. Nevertheless, we have modeled a sequence diagram for the ANTARCTICA SRS that represents the complete execution of the system with illustrative purposes, since it is easier to understand the behavior of the system from only one diagram, the sequence diagram, than from several diagrams, the statecharts. But we want to remark that in this case the sequence diagram does not represent a particular execution but a complete execution of the system. It is important to emphasize that because the second approach uses the sequence diagram as a tool to represent particular executions of the system.

Second approach

A performance model, in terms of Petri nets, to compute indices for particular executions of the system can also be obtained, in contrast to the complete execution of the system described by the first approach.

A concrete execution of the system can be modeled in UML by means of a sequence diagram. The LGSPN that represents the execution of interest of the system is obtained from the "component" nets, as in the first approach, but also using the target sequence diagram to compose them. In this case the performance annotations modeled in the sequence diagram (load and routing rates) will be of interest instead of the annotations in the statecharts.

This approach is part of the work developed in [BDM02] but it is out of the scope of this work.

Once the LGSPN that represents the performance model of the system or a concrete execution of it has been obtained (either by using the first or the second approach) then it could be necessary to transform it into a stochastic well formed net [CDFH93]. It occurs when different kinds of resources in the net should be synchronized. An example will be seen for the ANTARCTICA SRS later in this section. Also, it must be decided which places and/or transitions are of interest to compute performance indices, as we will see in the next section when performance results are presented.

Returning to the ANTARCTICA SRS system, the design proposed in [MIG00b] deals with one user and one majordomo. Petri nets allow to represent cases such as:

1. One user and one majordomo.
2. Several users served by one majordomo.
3. Many users served by many majordomos, once per request.

In the following, we model the first two proposed systems, which are illustrative enough to apply the proposal and to introduce the stochastic well formed nets. For the first system –one user and one majordomo– GSPN have the expressive power to accomplish the task. To study the second system, several users served by one majordomo, stochastic well-formed colored Petri nets [CDFH93] are of interest since each request is associated to a browser along the software retrieval process. Once the systems are modeled, we use analytic techniques implemented in GreatSPN [CFGR95] tool to obtain the target performance requirements.

Petri net model for a system with one majordomo and one user

Figures 7.10, 7.13, 7.14, 7.11 and 7.12 model the *component* nets of the system, i.e. the LGSPNs for the user, Alfred, the software manager, the browser agent and the salesman, respectively. They have been obtained by applying the definitions given in chapters 4 and 5 for each element in a statechart (activities, outgoing transitions, ...).

In the following, we remark some important aspects of these nets.

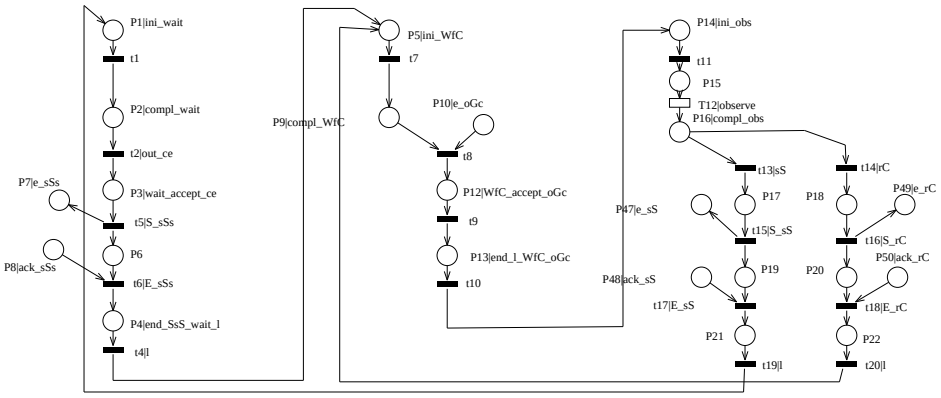


Figure 7.10: User LGSPN component

User component net. The number of tokens in place P1|ini_wait models how many users supports the system. This parameter cannot be modeled in the UML diagrams. The assumption, $\lambda_{examine}$ (cfr. section 7.1) is modeled in transition T12|observe. The choice between transitions t13 and t14 models the assumption about the probability that the user finds the software by selecting n catalogs.

Browser component net. Despite the difficult readability of this net, it must be pointed out that, among others, it reflects the important assumption that states that sometimes the browser does not ask for information to the software manager, but if it does it, then it will performed using RPC or traveling through the net. Transitions t77, t78 and t79, are involved in its modeling.

Alfred component net. It is important to note that all the transitions labeled add_infoX model the activities performed by Alfred to manage the information.

Software manager component net. The third modeling assumption, that expresses that the software manager consumes time consulting an ontology to create the catalog or to find the piece of software, is modeled by the transition T135|get_info. The number of tokens in place P164|ini_wait models how many browsers the software manager can attend. This parameter cannot be modeled in the UML diagrams.

From these component nets and applying the steps for the first approach, a performance model has been obtained that represents the execution of the whole system. The performance model is represented by the LGSPN shown in Figure 7.15.

Petri net model for a system with one majordomo and several users

The goal, as in the previous case, is to obtain the LGSPN that model the components and the execution of the system. In order to model with LGSPNs the situation

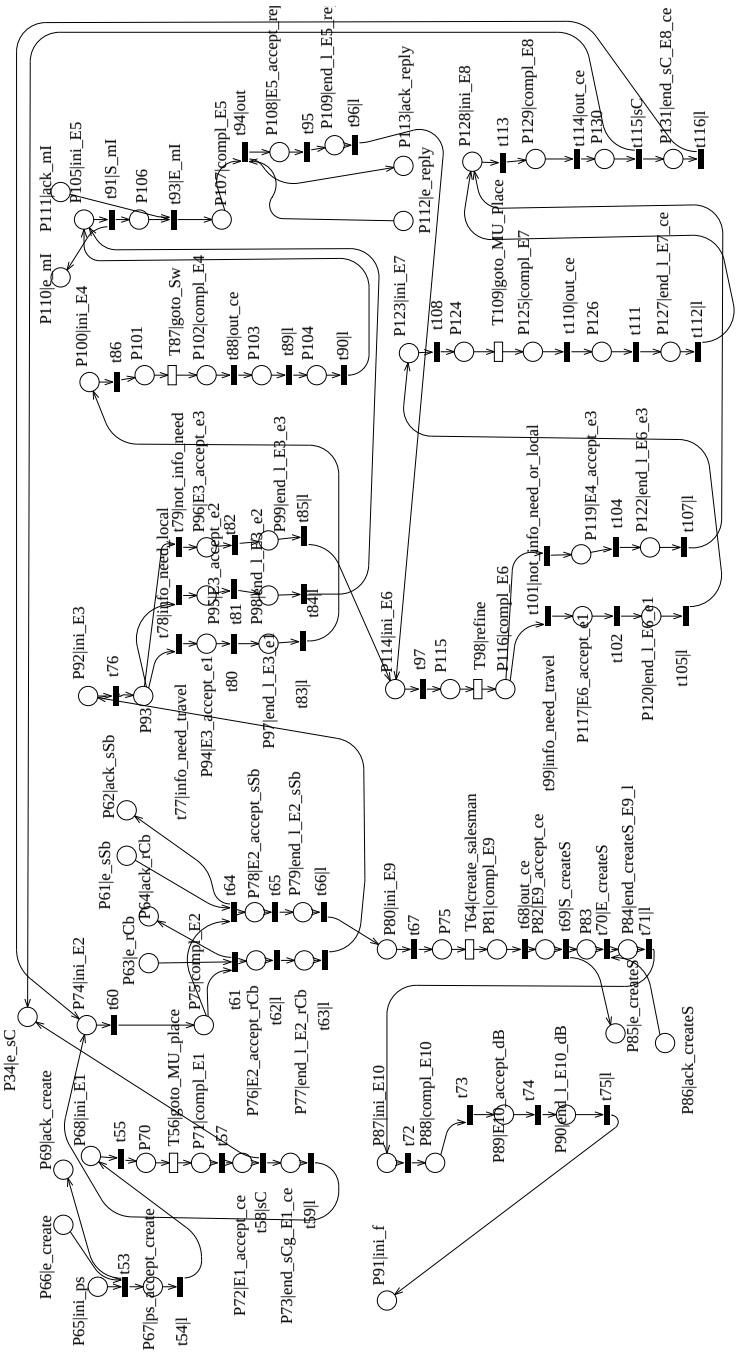


Figure 7.11: Browser LGSPN component

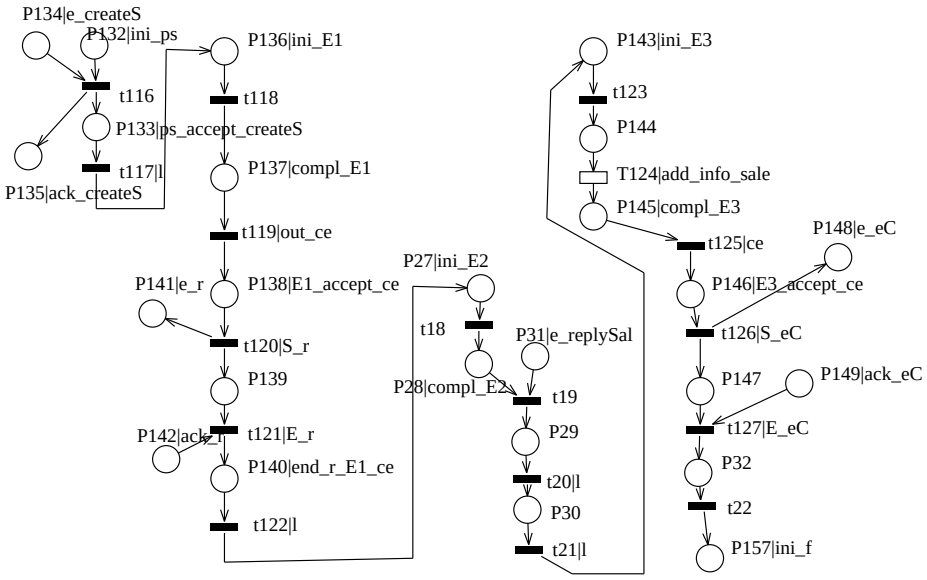


Figure 7.12: Salesman LGSPN component

of several users being served by one majordomo, it is necessary to mark some places in the component nets with several tokens.

Let us begin with the component nets. As in the previous system, the component nets are those obtained by applying the definitions given in chapters 4 and 5 for each element in a statechart. The LGSPNs for Alfred and the software manager will be the same as in the previous system, because only one instance of each is present in the system. On the contrary, the system will have as many instances of users, browsers and salesmen as required, suppose five for the example. For instance, in the net for the user, each token in the place wait_UserforService will represent a request to the system.

Moreover the system must distinguish between different tokens (they represent different requests), then we add a color domain for the requests, thus leading to stochastic well-formed colored Petri nets [CDFH93].

Now, pay attention on Figure 7.16, which represents the well-formed colored Petri net for the user. The R color means that the system deals with one to five requests and the initial marking m1 in place wait_for_service denotes that all class instances will be used. Moreover, all the places in the net have color R and the arcs are labeled with the identity function (<x>), in this way only one request could be fired once a time.

Figure 7.17 shows the well formed colored Petri net for the browser. Initial marking

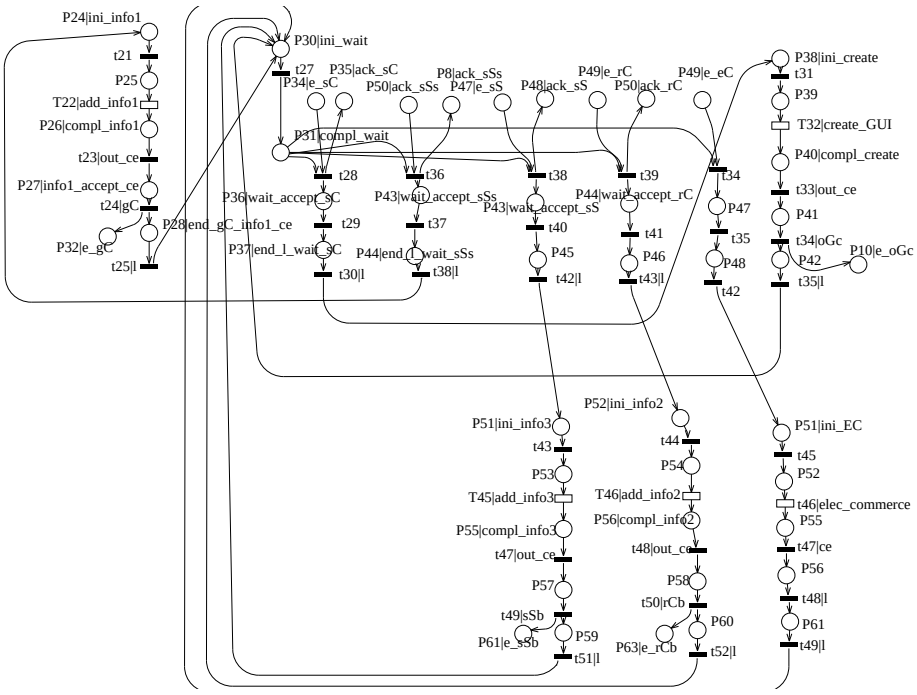


Figure 7.13: Alfred LGSPN component

m1 in place P1 shows that a maximum of five browsers could be created, one for each users request. Salesman well-formed colored Petri net (see Figure 7.18) has been designed in the same way.

Now, we are going to focus on the complete well-formed colored net for the system, see Figure 7.19. The same steps given when the first approach was presented must be taken into account. In addition, the following rules will be applied concerning the colors:

Rule 1. *All colors and markings defined for the component nets will appear in the net system.*

Rule 2. *The places with color and/or markings in the components nets will appear in the net system in the same way.*

Rule 3. *The arcs labeled in the component nets will appear in the net system in the same way.*

Rule 4. *Conflicting arcs are those that appear labeled in a component net but not in the component net which it is synchronized. When conflicting arcs appear, the net system must have the two arcs labeled, preserving in this way the richest semantic.*

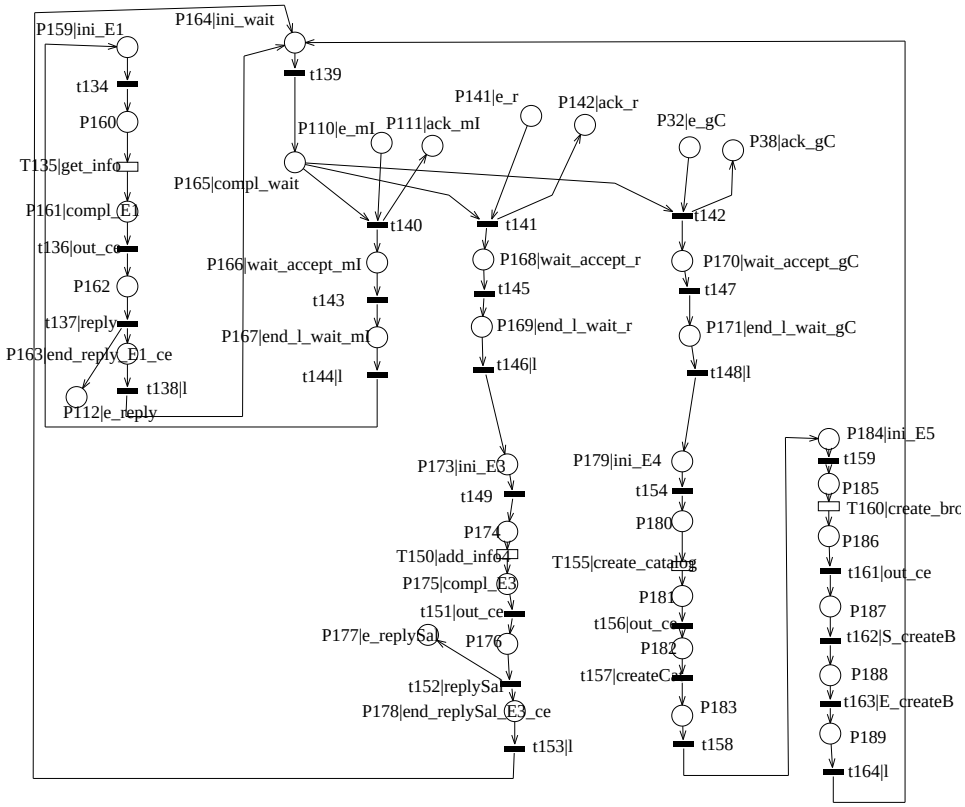


Figure 7.14: Software manager LGSPN component

As an example of rule 3 see outgoing arcs for the synchronized transitions select_sw and alfred.select_sw in Figures 7.13 and 7.16 respectively.

We remark that the complete well-formed colored net for the system describes concurrency at the same level as the complete net for the system given in the previous section. Moreover, it introduces a new level of concurrency. The use of colored tokens models concurrent user requests of a complete service, as it can be seen in the select_sw_service transition, that can fire several tokens from place wait_UserforService representing several user requests.

7.2.3 Performance results

In the last step of the process, the LGSPN or the SWN nets models that represent the execution of the complete system or the execution of a concrete scenario of interest are the input models to obtain performance figures. It will be achieved by applying analytical or simulation techniques to solve the formal model. The places and/or

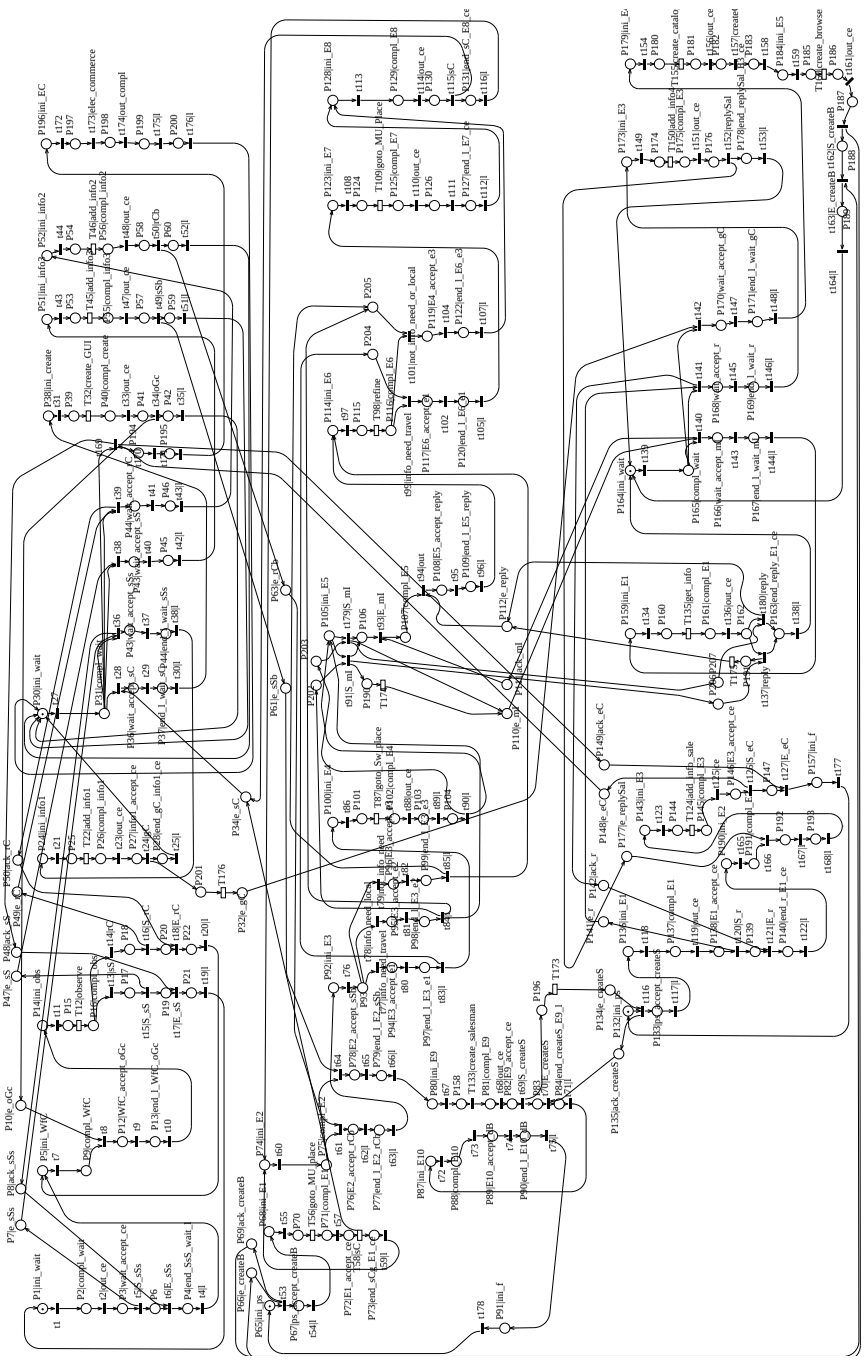


Figure 7.15: The LGSPN for the whole system

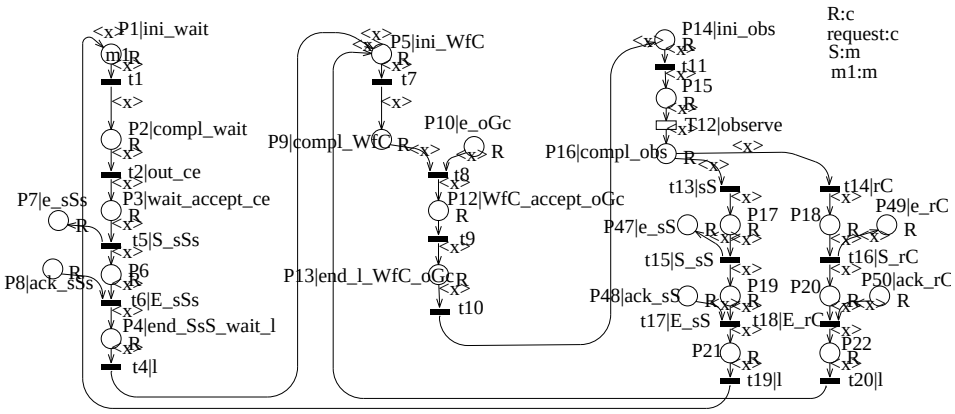


Figure 7.16: User SWN component

transitions that compute parameters of interest for the system must be identified (response time, delays, throughput). The modeling assumptions for the system must be taken into account in order to study the parameters.

The results for the ANTARCTICA SRS have been obtained from the complete nets that model the examples; the complete net that models the case in which the system is used by one user, who is attended by only one majordomo (Figure 7.15) and the complete net that models the case in which the system is used by several users, which are attended by only one majordomo (Figure 7.19).

In the case of the ANTARCTICA SRS, it is of interest to study the system *response time* in the presence of a user request. To obtain the response time, first the throughput of the t6|E_sSs transition, that represents the end of the select_sw_service message in the net system, will be calculated by computing the steady state distribution of the isomorphic *Continuous Time Markov Chain* (CTMC) with *GreatSPN* [CFGR95]; finally, the inverse of the previous result gives the system response time. Moreover it is important to know which are *the bottlenecks of the system and identify their importance*. There are two possible parts which can decrease system performance. First, the trips of the Browser from the "user place" to the "software place" (and way back) in order to obtain new catalogs. Second, the user requests for catalog refinements, because s/he is not satisfied with it.

In order to study the system response time and the two possible bottlenecks, the modeling assumptions described in section 7.1 must be considered. Then we have developed a test taking into account the following possibilities:

1. When the browser needs a new catalog (under request of the user) there are several possibilities:

   - The browser has enough information to accomplish the task or it needs to ask for the information. It is measured by the t79|not_info_need tran-

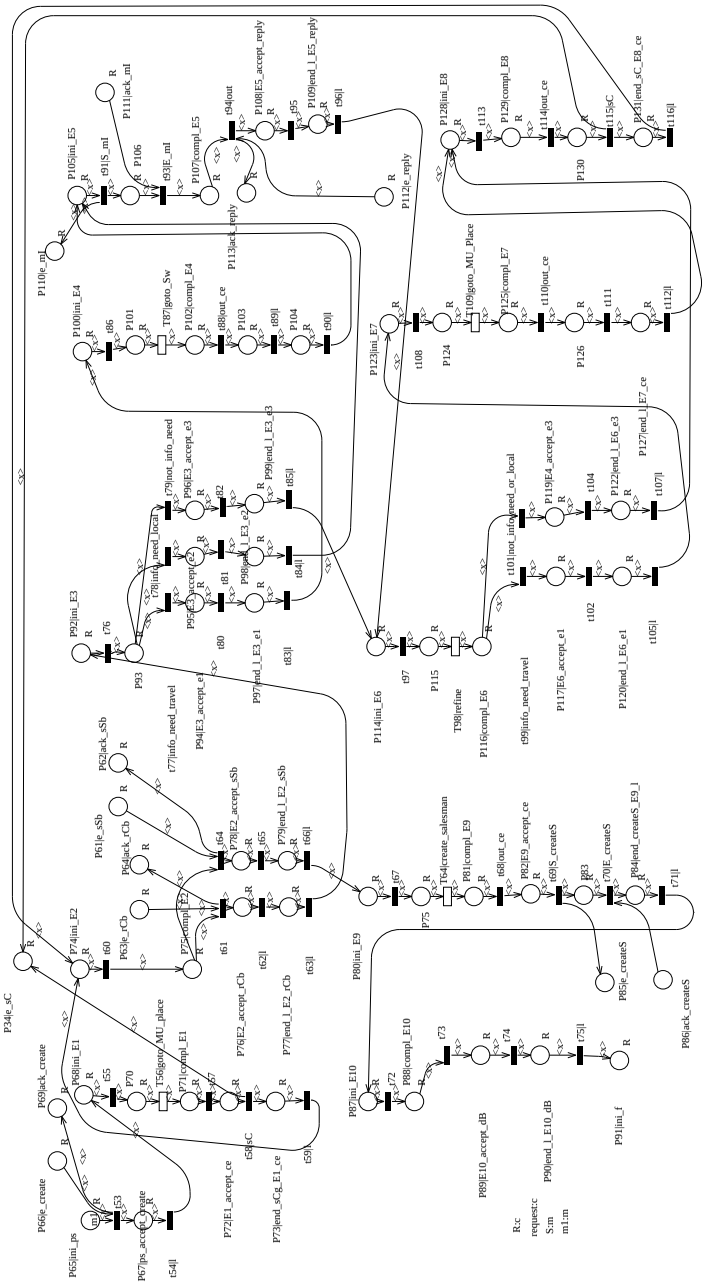


Figure 7.17: Browser SWN component

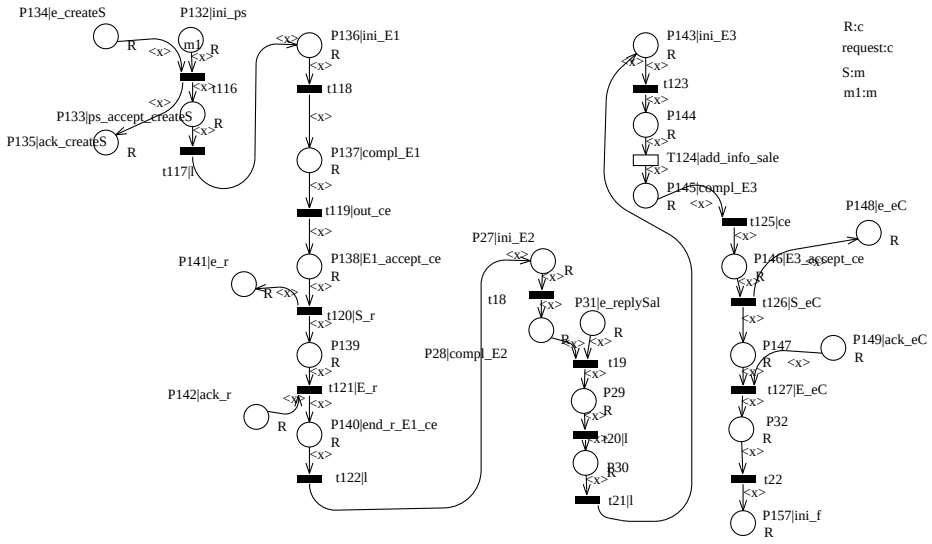


Figure 7.18: Salesman SWN component

sition. We have considered an "intelligent browser" which does not need information the 70% of the times that the user asks for a refinement.

- When the browser needs information to perform the task, it may request it by a *remote procedure call* (RPC) (represented in the net system by the t78|info_need_local transition) or it may travel through the net to the Software_place (represented in the net system by the t77|info_need_travel transition) to get the information and then travel back to the MU_Place. In this case, we have considered two scenarios. First, a probability equal to 0.3 to perform a RPC, so a probability equal to 0.7 to travel through the net. Second, the opposite situation, a probability equal to 0.7 to perform a RPC, therefore a probability equal to 0.3 to travel through the net.

2. To test the user refinement request, we have considered two different possibilities. An "expert user" requesting a mean of 10 refinements, and a "naive user" requesting a mean of 50 refinements. Transitions t14|rC that represents the message refine_catalog and transition t15|sS that represents the message select_sw model this behavior.

3. The size of the catalog obtained by the browser can also decrease the system performance. We have used five different sizes for the catalog: 1 Kbyte, 25 Kbytes, 50 Kbytes, 75 Kbytes and 100 Kbytes.

4. The speed of the net is very important to identify bottlenecks. Also the time

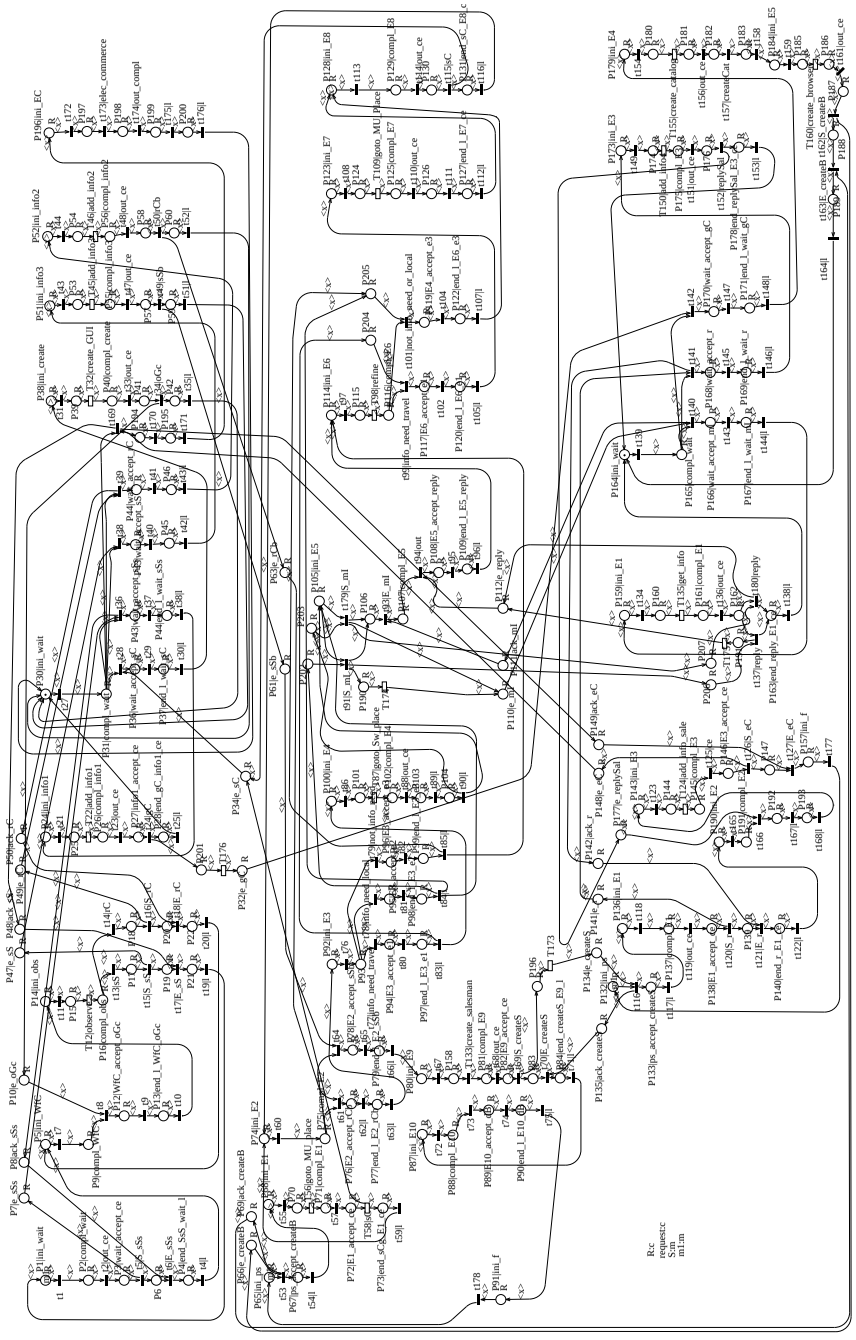


Figure 7.19: The SWN for the whole system.

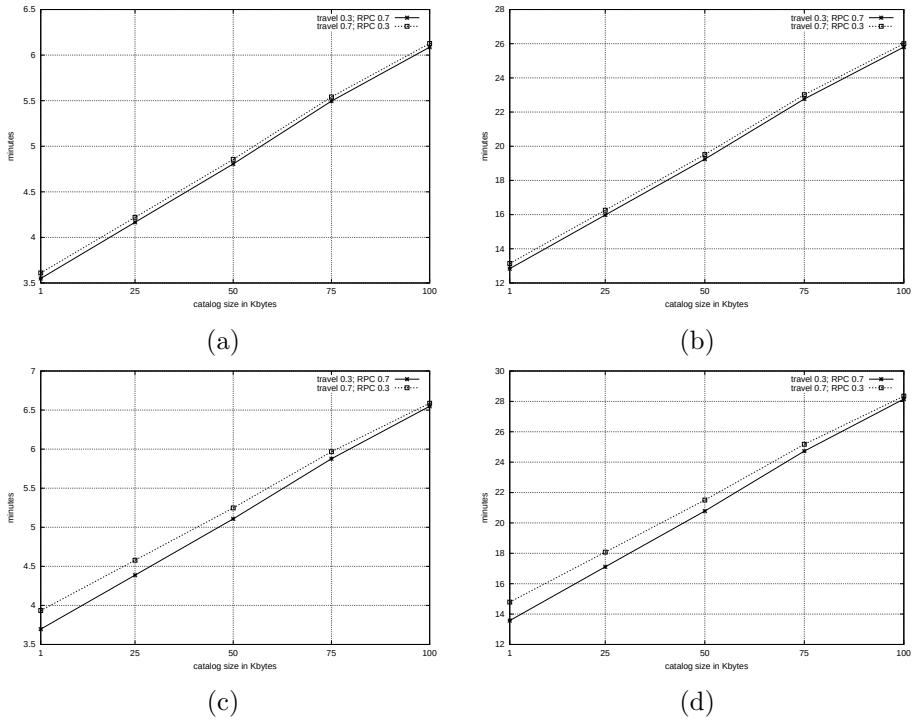
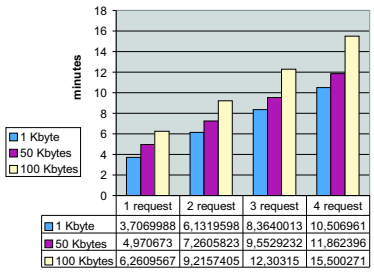


Figure 7.20: Response time for a different scenarios with an "intelligent browser". (a) and (b) represent a "fast" connection speed, (c) and (d) a "slow" connection speed; (a) and (c) an "expert user" and (b) and (d) a "naive user".



	1 request	2 request	3 request	4 request
1 Kbyte	3,7069988	6,1319598	8,3640013	10,506961
50 Kbytes	4,970673	7,2605823	9,5529232	11,862396
100 Kbytes	6,2609567	9,2157405	12,30315	15,500271

Figure 7.21: Response time for an "intelligent Browser", an "expert user", a "fast" connection and also different number of request.

spent by sending remote messages depends on the speed of the net. It must be measured as explained in step 5 in the previous section. We have considered two cases: a net with a speed of 100 Kbytes/sec. ("fast" connection speed) and a net with a speed of 10 Kbytes/sec. ("slow" connection speed). The messages sent between Alfred and the software manager, the browser and the software manager, and the browser and the salesman are those that travel through the net. They are represented by the transitions get_catalog, more_information, reply and create_salesman.

5. The time spent reading the catalog is measured by the transition T12|observe. We have supposed an "expert user" who reads the catalog in ten seconds.

One user and one majordomo
Figure 8.13(a) shows system response time (in minutes), for the net in Figure 7.15, supposing "fast" connection speed, "expert user" and an "intelligent" Browser. One of the lines represents a probability equal to 0.7 to travel and 0.3 to perform a RPC, the other line represents the opposite situation. We can observe that there are small differences between the RPC and travel strategies. Such a difference is due to the round trip of the agent. As the agent size does not change, this difference is not relevant for the global system performance. Thus, we show that the use of mobile agents for this task does not decrease the performance.

Figure 8.13(b) shows system response time (in minutes), supposing "fast connection", "intelligent" Browser, "naive user". The lines have identical meaning than in Figure 8.13(a). The two solutions still remain identical.

Someone could suspect that there exist small differences because of the net speed. So, we have decreased the net speed to 10 Kbytes/sec., (Figures 8.13(c) and 8.13(d)). It can be seen how the differences still remain non significant.

Several user requests served by one majordomo
Figure 7.21 represents a test for an "intelligent Browser", an "expert" user, a probability for RPC equal to 0.7 and equal to 0.3 to travel. Now, we have tested the system for a different number of requests ranging from 1 to 4, thus the colored model in Figure 7.19 has been used. Observe that when the number of requests is increased, the response time for each request increases, i.e., tasks cannot execute completely in parallel. Alfred and the Software Manager are not duplicated with simultaneous requests. Thus, they are the bottleneck for the designed system with respect to the number of concurrent requests of the service and the impact of such bottlenecks can be evaluated using our approach.

7.3 On the use of the design patterns in the software performance process

In this section we explore the use of the design patterns [GHJV95] and its possible benefits in the process to evaluate performance for software systems. As we explained

in [MCM00a], we do not pretend to introduce a new orthogonal proposal to that explained in section 7.2 and developed in [MCM00b, MCM01b]. On the contrary, our idea is to begin to study how new approaches in software design such as design patterns [GHJV95], can be integrated into the software performance process. The novelty of these disciplines (both new software design techniques and software performance) causes that less effort has been dedicated to merge them. Therefore we consider important to begin to consider this task opening possibilities to future research. Among the wide range of design techniques, we explore the convenience of the design patterns mainly motivated by their success in the last years. Since benefits from design patterns come from their ability to achieve software reuse, in the same way, we claim for reuse in performance modeling.

7.3.1 Basics on design patterns

The idea of the design patterns was formulated initially in the architecture[3] field in [AIS+77] as *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"*. This way to use and document designs was rapidly accepted in most of the engineering fields and concretely in the software design field was formulated in [GHJV95] as *"descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context"*. We assume that the reader is familiar with the patterns language as proposed in [GHJV95], where twenty three design patterns, that solve a wide range of software design problems, are proposed. This language describes each pattern using the "sections" that appear in table 7.1.

7.3.2 Patterns and software performance

It is widely recognized that software design is a hard task that requires a significant amount of effort and experience. Most of the "classical" paradigms for the software development process have recognized and identified as a goal the necessity of *reuse* in all the stages of the development process. So, software reuse [JGJ97] has become a must to increase software productivity. The object-oriented approach is the representative paradigm to develop software, if it is considered that software reuse is a must. This is because the concepts underlying object-oriented paradigm promote reuse in all stages of the software life cycle better than the rest of the paradigms. The proposal of the design patterns [GHJV95] has been successfully applied as a way to reuse at the design stage in the context of the object-oriented paradigm.

We have considered that software performance must be accomplished in the early stages of the software life-cycle when proper actions to solve performance problems take less effort and less economical impact. Then the design stage where patterns are

[3]The art and science of designing and making buildings. Not the hardware or software architecture field.

Section name	Section description
Pattern name	Conveys the essence of the pattern succinctly.
Intent	The problem that the pattern addresses.
Motivation	A scenario that illustrates how the pattern solves the problem.
Applicability	Situations in which the pattern can be applied.
Structure	Representation of the classes in the pattern (using OMT notation).
Participants	Classes/objects participating in the design pattern and their responsibilities.
Collaborations	How the participants collaborate.
Consequences	Trade-offs of using the pattern.
Implementation	Techniques to be used when implementing the pattern.
Sample code	Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.
Known uses	Examples of the pattern found in real systems.
Related patterns	Patterns closely related to this one.

Table 7.1: Sections of the design patterns language.

used seems to be a good point to address performance evaluation. So, we consider to take into account to bring together software reuse and software performance.

The language proposed in [GHJV95] to describe design patterns deals with the structural and behavioral aspects of the software and also gives trends to accomplish the implementation phase. In the next section, we propose the enrichment of this language with performance skills, describing for each pattern its performance goals. In spite of design patterns use the OMT [RBP+91] notation to describe the structure and behavior of the design, we will use pa-UML notation.

7.3.3 Adding performance skills to design patterns: leading performance patterns

The three steps process presented in section 7.2 still remains valid to be combined with a pattern approach. It is necessary to identify where and how to use patterns inside the process.

The first step of the process proposes to *model software requirements using the desired software life-cycle with pa-UML*. Then design patterns can be used in this stage. If design patterns include the ability to express performance parameters as

well as performance assumptions, it will be achieved a model expressed in pa-UML with all the required information to accomplish the second step of the proposal.

In this section, we give a proposal of enlargement of the design patterns language in order to describe performance skills. First, we propose the improvement of the "Collaborations" and "Participants" sections. Later, new sections for the design patterns language are proposed.

The "Collaborations" section of the language is enhanced as follows. Currently, this section is described in some patterns using a sequence diagram, the rest of the patterns describe it in a textual way. We propose the use of a sequence diagram in all the patterns to describe the section. In this way, a sequence diagram will be used to annotate the message load among objects and the probability for the guards success using pa-UML.

The "Participants" section, which describes the classes/objects participating in the pattern, must be also enhanced. A statechart for each participant must be modeled. These diagrams represent the life of objects. The statechart will be annotated with the events load, the probabilities of the guards and the time to perform the actions using pa-UML.

The meaning of the annotations in the diagrams, the techniques to obtain them, the distinction between messages sent among objects residing on the same machine or on different machines together with the problems encountered to develop the process were largely explained in section 7.2, so we do not extend here on them.

Now, we are going to extend the design patterns language with new sections. These sections taken from [Smi90] are those that we consider necessary to reach a complete description of the pattern performance skills:

- **Performance goals**: the pattern performance objectives will be expressed: response time, delays, throughput or utilization.

- **Workload definitions**: such as request arrival rates or the number of concurrent users.

The sections proposed in [GHJV95] together with the enhancement of the "Collaboration" and "Participants" sections and the new sections proposed, define a "performance pattern". A "performance" pattern should be the *description of the relevant performance characteristics of a general design problem that can be customized in a particular context.*

The task of identifying "performance patterns" in the real word is not an easy task. Experience obtained from the modeling process in the performance field is the best feedback to detect and propose them.

The catalog presented in [GHJV95] contains twenty three design patterns. It would be very interesting to introduce performance skill in all of them. So, they could become "performance patterns".

7.4 Conclusions

In this chapter we have presented a process to evaluate performance of software systems. The gap among the software engineering practices and the performance evaluation practices causes that there does not exist a well-accepted process to evaluate performance of software systems.

The process tries to exploit the software life-cycle by annotating the designs using the proposal given in chapter 3, pa-UML. After that, the automatic translation process given in chapters 4 and 5 gains a stochastic Petri net that represents the behavior of the whole system. The steps proposed in section 7.2.2 give the appropriate decisions to convert this net in the performance model of the system, this is why we consider the process as semi-automatic instead automatic. Finally, the experiments to test the system are developed an applied to the stochastic Petri net.

As relevant features of our proposal we want to underline that it tries to obtain the performance model as a by-product of the software life-cycle. Then no additional efforts should be made by the software designer aside to annotate the system load. Any CASE tool that supports UML notation can be integrated in the process, since the annotations are introduced as tagged values. The UML diagrams can be translated into the standard format XMI, which can be used as input language to apply the translation rules give in chapters 4 and 5 to obtain the stochastic Petri nets. It is also important to remark that "what-if" analysis is possible to accomplish since performance results can be obtained in the early stages of the software life-cycle when the economical impact and the effort to change system requirements is less important.

An example from the mobile agents research area has been used as a running example to explain the process. We believe that the interest of the example is true since the performance is a subject of study in this area.

The proposal to include patterns in software performance prediction has been applied only to the ANTARCTICA SRS system in [MCM00a], but we do not include it here since much of it repeats the diagrams developed previously for the system. Being applied to only one experiment is not enough for us to conclude about its applicability. Nevertheless, in our opinion, the main benefit of the use of design patterns in the software performance process should be the possibility to reuse performance aspects for specific software designs. In short, a "performance" pattern should be the description of the relevant performance characteristics of a general design problem that can be customized in a particular context. To our knowledge only the works [GM00, SW00, VGNP00] have studied how to combine patterns and software performance evaluation.

During the process, several modeling assumptions have been taken. Among them, that which refers to model the time spent by the messages traveling through the net with an exponential distribution could cause some critics. However, to lose representation reality to obtain an analyzable model can be necessary many times. Anyhow, the possibility of representing network delays with non-exponential distributions could be considered in the future if simulation techniques are used instead of the analytic approach followed here.

Chapter 8

Additional Experiences in the use of the Software Performance Process

Our proposal to evaluate performance of software systems was applied in chapter 7 to the ANTARCTICA SRS, that was introduced as a running example, to obtain performance indices for a distributed software system. Once the applicability of the approach has been showed, we propose in this chapter to intensify its use. It will be performed by realizing an analysis of another software retrieval system, that also is a distributed system; by realizing a performance comparison of both systems (the ANTARCTICA SRS and the new one) and finally by analyzing a software Internet protocol.

For each one of the two new proposed systems we will apply: (a) the performance annotations (introduced in chapter 3) to the UML diagrams that describe the new system; (b) the automatic translation for the statecharts and the activity diagrams proposed in chapters 4, 5 and 6; (a) and (b) guided by the process given in chapter 7.

The software retrieval system that is going to be analyzed in this chapter is actually a simplification of a family of software retrieval systems, that we will call later on *Tucows-like* [Inc99d] systems. The ANTARCTICA SRS was proposed as an agent-based alternative to this kind of systems, then both proposals share the same goal: to retrieve software in an easy and efficient way. Since Tucows-like systems are not designed using mobile agents, it will be of interest to compare both designs in order to know how much time the network connection needs to be open to retrieve a software product and also the impact of the mobile agents in the design of a software retrieval system.

The performance comparison among the ANTARCTICA SRS and the Tucows-like systems will be important both to discuss performance criteria about the use of the mobile agents paradigm in the context of software retrieval systems and to

show new perspectives in the use of our proposal. All in all, we will able to reason about the applicability of our proposal beyond an isolate but complex example. The analysis of the Tucows-like system and the performance comparison were developed in [MCM01a, MCM03].

The second system analyzed in this chapter is a well-know protocol to manage mail in remote hosts, the POP3 mail protocol. By introducing this example we intend to test completely our proposal, since two important features have not been used in the previous examples. Therefore, we will test the translation proposed for the activity diagrams in chapter 6 and also we want to test the second approach in the step of the stochastic Petri net modeling presented in section 7.2.2 of chapter 7, i.e. to obtain a performance model for a particular scenario of the system. This example was developed in [LGMC02c].

The chapter is organized as follows. Section 8.1 presents the requirements and modeling assumptions for the Tucows-like systems, while they are modeled and analyzed by means of our approach in section 8.2, where we present the results for the same experiments realized for the ANTARCTICA SRS in chapter 7. The performance comparison of both systems is realized in section 8.3, where interesting conclusions about the use of the mobile agents in the design of software retrieval systems are obtained. The POP3 protocol will be studied and performance results for it obtained in section 8.4. Finally, section 8.5 summarizes the contributions of this chapter.

8.1 Tucows-like software retrieval systems

There exist a variety of software retrieval systems, as the popular web sites Tucows.com [Inc99d], Download.com [Inc99a] or Gamecenter.com [Inc99c], that provide Internet users with facilities to retrieve and install software. These systems allow users to find software in two different ways, by using a keyword-based search engine and by navigating through categories especially designed to make this task easier.

The software architecture of these kind of systems for the navigation facility shares commonalities. Therefore, it is possible to model how these kind of systems work, making a number of assumptions, without losing reality with respect to performance aspects. We refer to these kind of systems as Tucows-like systems.

The keyword-based search engine helps users that know some features of the wanted software. This search facility will not be considered in this work since it has not its counterpart in the ANTARCTICA SRS (defined in chapter 7) and moreover it can not be used by naive users that do not known the concrete software that they need.

The navigation facility consists of several web pages residing on a server and organized as categories linked between them in a way that guides the user to find the software. For instance, a number of these systems present an initial web page where the categories correspond to different operating systems, say Windows 2000, Windows 95/98, Linux or Unix. The user selects the desired category and a new web page with several topics like multimedia, browsers or Internet tools is loaded; in this way the user can continue the search of the software. As we saw in chapter 7, the

ANTARCTICA SRS offers a mechanism to retrieve software similar to the navigation facility, but it makes use of mobile agents to perform the task, therefore a performance comparison can be realized between the two systems.

Our goal of study in depth the application of our proposal is accomplished by evaluating performance indices for a Tucows-like system in order to compare them with those obtained for the ANTARCTICA SRS in chapter 7. Basically, taking advantage of our proposal that consists of the description of the functional and performance requirements by means of the pa-UML proposal, given in chapter 3; the automatic translation of the obtained models in a labelled generalized stochastic Petri net (LGSPN) that represents the behavior of the whole system, by applying the transformations given in chapters 4 and 5; and the iteration of the process presented in chapter 7 to semi-automatically obtain the LGSPN system that represents the performance model of the system and the possibility of realize the tunning of the system if it is desired.

In the following, we describe the navigation facility of the Tucows-like systems. The next section is devoted to model and analyze the system, according to our approach, by performing the same experiments applied to the ANTARCTICA SRS, while in section 8.3 the desired comparison with the systems will be performed .

System description and modeling assumptions

In a Tucows-like system, the *user* navigates, with the help of a *browser*, different HTML pages (representing software categories and descriptions of concrete pieces of software) until s/he finds a piece of software that satisfies her/his needs. Then, that piece of software is downloaded.

In short, the process of selecting software by navigating HTML pages behaves as follows: The user "clicks" on a category, then the browser requests the *web server* for the corresponding HTML page. The web server returns the HTML page to the browser, which presents it to the user. After reading this page, the user can "click" on another link in order to access a new web page with other categories or a list of software under the current category. This process is repeated until the user finds a software that fulfills her/his needs. Then the browser requests the selected software, which is downloaded into the user computer.

In the following we give the modeling assumptions taken into account for the Tucows-like system. It is easy to see that each one has its counterpart in the ANTARCTICA SRS, in this way the experiments proposed for the ANTARCTICA SRS can be reproduced in the context of the Tucows-like system:

- It must be considered that the user spends some time *reading* the information presented by the system. An exponentially distributed random variable with rate $\lambda_{examine}$ ($\lambda_{examine}$ is obtained as the inverse of the time in seconds) will be used to model several kinds of users.

- The number of HTML pages that the user must navigate until s/he finds the software is difficult to estimate (it depends on her/his experience). The probability that the user finds the software by selecting n categories models different

kinds of users, from naive users, those who need to visit many categories to find the software, to expert users, those who find the software visiting very few categories.

- Whenever the user requests an HTML page or a concrete piece of software, the web server must perform the corresponding activities to find the page or the piece. The time consumed by these activities will be modeled by variables with rates $\lambda_{findHTML}$ and $\lambda_{findFile}$.
- The browser, on the client machine, sends messages through the net to the web server on the server machine and vice versa. A variable with rate λ_{m_i} models the time spent by the message i navigating through the net. Notice that local messages sent among the user and the browser do not consume net resources.

As in the case of the ANTARCTICA SRS (chapter 7), it is assumed that the network delays are modeled by exponentially distributed random variables. As we discussed for the ANTARCTICA SRS, it could be argued that exponential assumption could be not realistic in some circumstances for the modeling of network traffic, and that heavy tailed distributions would be better. The reasons given previously remain valid in this context and it is interesting to remember that the possibility of representing delays with non-exponential distributions could be considered if simulation techniques are used instead of the analytic approach followed in our examples, which is not incompatible with the process presented in chapter 7.

8.2 Analysis of the Tucows-like systems

Once the requirements of the Tucows-like system have been proposed, we start its performance analysis using our proposal. Therefore, in this section we apply the three steps given in chapter 7. First, a pragmatic modeling is given using the pa-UML notation to describe performance requirements. Second the performance model in terms of LGSPNs is obtained for two cases of interest: a Tucows-like system with one user and one browser and a Tucows-like system with several users and several browsers. And finally, some tests based on the modeling assumptions are defined to compute performance figures over the performance models.

8.2.1 Modeling the system using pa-UML

In this section we accomplish the first step of the process presented in chapter 7, then we model the dynamic view of the Tucows-like system using pa-UML notation as proposed in chapter 3.

Use Cases

The use case diagram in Figure 8.1 shows the actor that interacts with the system, the “user”, and the two possible use cases for the system: the “navigation facility”

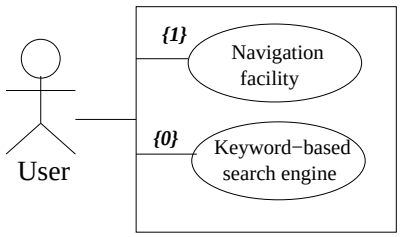


Figure 8.1: Use cases for the Tucows-like system.

and the "keyword-based search engine". It is important to remember that in this diagram we follow the notation given in [CM00] where the tagged values mean the probability that the user executes the scenario attached to the link where the probability is annotated. We assume that p=1 in the link among the user and the navigation facility use case because we are not interested in computing indices for the keyword-based search engine, in this way all user executions correspond to the navigation facility. These use cases are described in the following.

Navigation facility use case description.

- Principal event flow: the user requests the system with the URL that contains the wanted software. The browser sends the request to the web server, who searches and selects the software on behalf the browser that finally obtains it to install it in the client machine.
- Exceptional event flow: if the user is not satisfied with the information presented in the HTML page, i.e. it does not contain an entry with the wanted software, s/he can ask for a new one. This process could be repeated as many times as necessary until the user selects an entry in the HTML page that corresponds with a software .
- Probability to be executed by the user: 1.

Keyword-based search engine use case description.

- Principal event flow: in this case the system offers a widget where the user asks directly the browser for the wanted software using a simple query language. It is an alternative given by the Tucows-like systems to retrieve remote software.
- Probability to be executed by the user: 0.

Sequence diagram

As proposed in our approach, each use case relevant to compute performance indices must be detailed. Then, the sequence diagram in Figure 8.2 shows a detailed description of the "navigation facility" use case. It shows the messages sent among

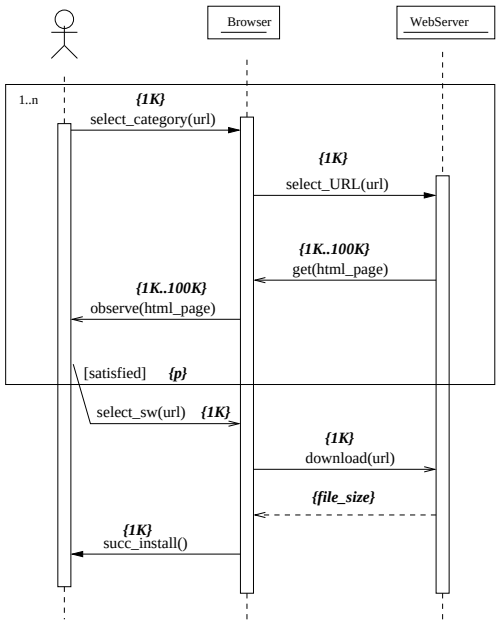


Figure 8.2: Sequence diagram for the navigation facility use case.

the objects in the system with the purpose to retrieve the piece of software that the user needs. Being in a distributed system, two different kinds of messages were distinguished, those that travel through the net (sent between the browser and the web server) and those that do not (sent between the user and the browser). As it is known, this feature will be relevant in the final performance model in order to calculate the rate of the transitions that represent messages sent through the net, taking into account the assumptions considered.

The sequence diagram begins with a select_category(url) message, being its size {1 Kbyte}, sent by the user to the browser. It represents the "click" performed by the user in the browser to select a category in an HTML page. The rest of the diagram describes the steps explained in the description of the system for selecting software.

Statechart diagrams

In order to get a complete description of the Tucows-like system dynamics and its load, we are going to develop a statechart for each class with relevant dynamic behavior.

As in the ANTARCTICA SRS, the guards in the diagrams will be used to assign probabilities to the corresponding transitions in the performance model, i.e. routing rates, since a guard language is not introduced. Remember that the annotations in these diagrams correspond to the duration of the activities, the size of the messages and the routing rates. In the following each statechart is described.

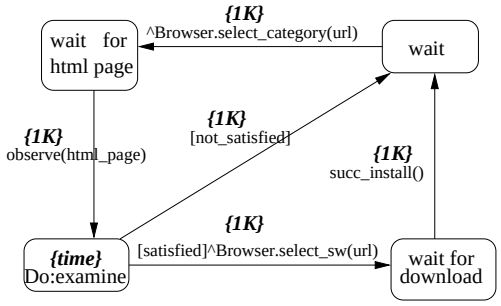


Figure 8.3: Statechart diagram for the user.

User statechart diagram.
Figure 8.3 shows this statechart, i.e. the behavior of a user of the system is represented. The user is in the wait state until s/he activates the select_category event. This event sets the user in the wait for HTML page state. The observe event, sent by the browser, allows the user to perform the examine activity that has associated the label {time}. This label models the time that the user spends reading the HTML page. This activity will be translated in the performance model in a timed transition, and by modifying its rate different kinds of users can be modeled, according with the $\lambda_{examine}$ assumption pointed out in the previous section. Once the activity is performed two situations can arise:

- If the requested software is not present in the current HTML page the user returns to the wait state.
- In other case, the user sends the select_sw(url) message to the browser, where url means the web address where the software is located in the server, and enters in the wait for download state. When the browser fulfills the necessary activities to complete the download, it sends to the user the succ_install() message and the user returns to the wait state.

Browser statechart diagram.
Figure 8.4 shows the browser's statechart. The browser behaves as a server object: it is waiting for user's requests, represented by select_category and select_sw events.

When a select_category event arrives requesting a URL, the browser sends to the web server the select_URL message and waits for a new HTML page. When the web server obtains it, then it triggers the get event attaching the new HTML page, whose estimated size is {20K..30K}. Since this message is sent through the net, it will be interpreted in the performance model as a transition with rate $\lambda_{m_{get}}$, as we pointed out in the previous section. After that, the HTML page is shown to the user.

When a select_sw event arrives requesting a URL that contains a piece of software, a download message with the URL is sent to the web server. The browser waits for the reply message that contains the requested file with size file_size, it will be translated in the performance model in a transition with rate $\lambda_{m_{reply}}$. Finally, the

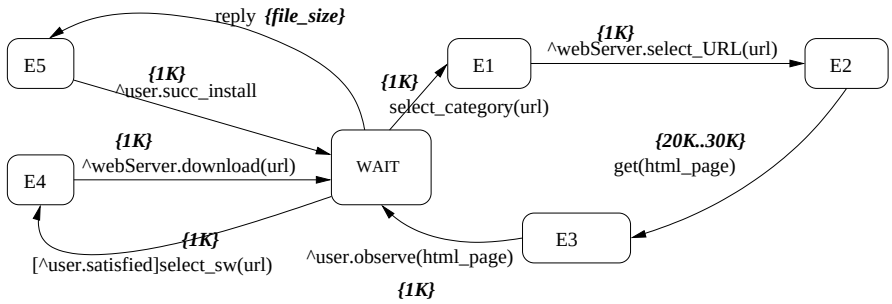


Figure 8.4: Statechart diagram for the browser.

file is installed (succ_install).

Web server statechart diagram.
As the browser, the web server behaves as a server object. It is waiting for a request (select_URL and download) from the browser. For each request, the web server performs the corresponding actions to serve it (find_html_page and find_file). When the actions are completed, it sends the corresponding message to the browser. Figure 8.5 shows the web server's statechart diagram.

8.2.2 Modeling with LGSPNs

It is well-known that the pa-UML models designed in the previous step do not represent accurately some system features such as concurrency. As an example, we could not distinguish in these diagrams, a Tucows-like system that represents one user and one browser from the other that represents several users and several browsers, the two scenarios that we propose as examples.

Then, in this section we accomplish the second step of our process given in chapter 7, that achieves a performance model in terms of LGSPNs for each one of the previous scenarios. Concretely, we are going to apply the "first approach" given for this step, that allows to obtain performance indices for the whole system.

First step of the first approach

We begin with the first step of the selected approach, therefore we are going to apply to each statechart the translations proposed in chapter 4, since all of them are "flat", to obtain a component LGSPN from each one. In the following we comment the relevant aspects of these component nets.

Figures 8.6, 8.7 and 8.8 depict the component nets of the system, i.e. the LGSPN for the user, the browser and the web server, respectively.

User component net.
The number of tokens in the place P1|ini_wait models how many concurrent users

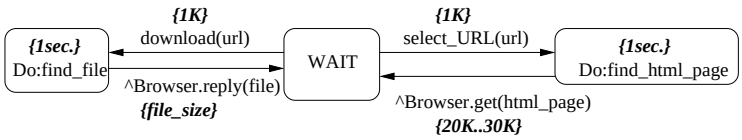


Figure 8.5: Statechart diagram for the web server.

supports the system. This parameter cannot be modeled in the UML diagrams.

The firing of the transition t3|S_sC models the dispatch of the select_category message to the browser to specialize the current HTML page, since the transition t4|E_sC models its acknowledge. The firing of the transition t11|examine models the time spent by the user reading the information presented in the new HTML page. After the end of the reading, a choice will determine whether the user is satisfied with any of the products shown (firing of the immediate transition t13|out_ce), or not (firing of the immediate transition t12|out_ce).

The firing of the immediate transition t21|out models the arrival of the message succ_install to confirm that the retrieval of the software has been successfully completed.

Browser component net.
The number of tokens in the place P1|ini_wait models how many concurrent browsers can access to the system. This parameter cannot be modeled in the UML diagrams.

The firing of the transition t4|out models the arrival of the message select_category from the user requesting for a specialization of the category that s/he has examined. The request will be sent to the web server by firing the transition t23|S_sURL. The firing of the transitions t27|out and t32|S_observe model, respectively, the arrival of the message get, then obtaining a new HTML page with new categories, and the dispatch to the user of the message observe to read the HTML page.

The firing of the transition t2|out models the arrival of select_sw messages from the user requesting a concrete piece of software. The request will be sent to the web server by firing the transition t13|S_download. The firing of the transition t3|out models the arrival of the message reply that obtains the file requested. Finally, the firing of the transition t8|S_sI models the message succ_install which is the advertisement to the user that the retrieve of the software has been successfully completed.

Each place labeled e_eventname or ack_eventname will be superposed in the complete net with the places in the user component net with the same name.

Web server component net.
The number of tokens in the place P9|ini_wait models how many concurrent processes the web server has launched to attend browser's requests. This parameter could not be modeled in the UML diagrams.

The firing of the transition t9|out models the arrival of the message select_URL to request for a new HTML page, since the firing of the transition t8|out models the

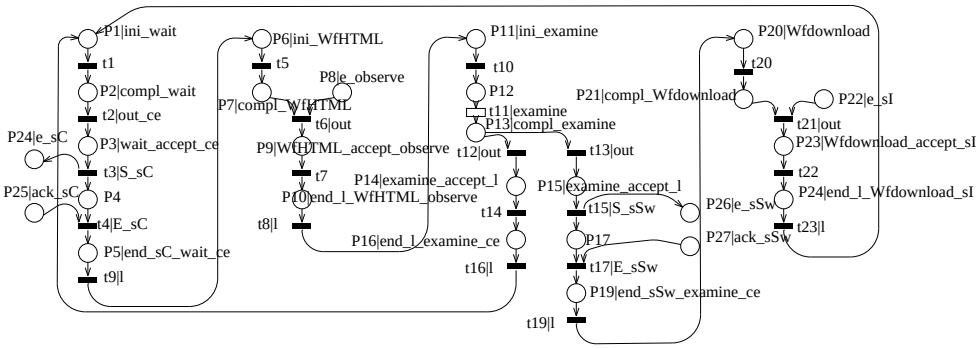


Figure 8.6: User LGSPN component

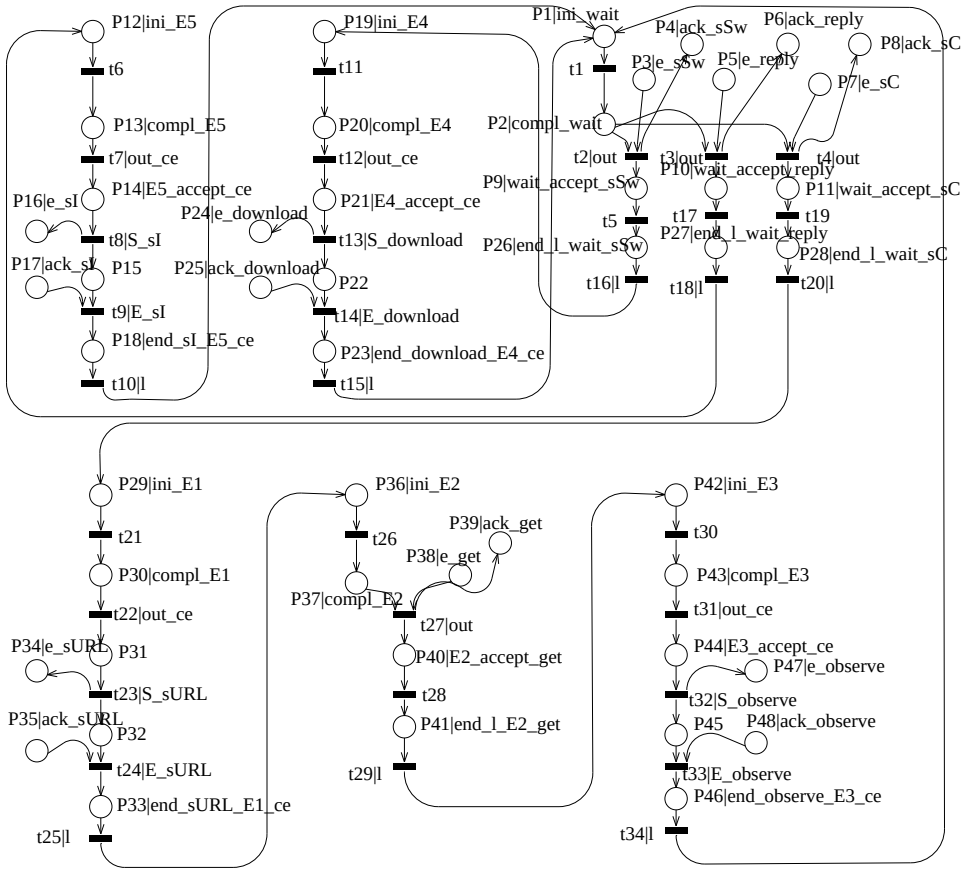


Figure 8.7: Browser LGSPN component.

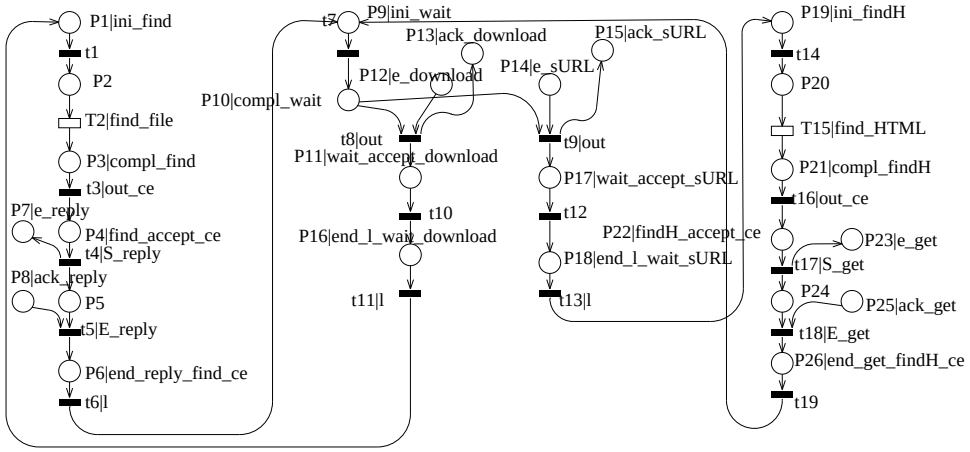


Figure 8.8: Web server LGSPN component.

arrival of the download message to request for a concrete piece of software. Transition T15|find_HTML models the completion of the search for a new HTML page, since transition T2|find_file models the completion of the search for a requested piece of software. The firing of the transition t17|get models the dispatch of the HTML page to the browser. Finally, the firing of the transition t4|reply models the dispatch of the file to the browser.

Places labeled e_eventname and ack_eventname will be superposed in the complete net with the places in the browser component net with the same name.

These component nets are useful for the system with one user and one browser. But for the system with several users and browsers, we will convert them in a stochastic well-formed net (SWN), as we will explain.

Second step of the first approach

When the component nets have been obtained, the first step of the "first approach" has been completed. In the following we comment how to apply the second step of the "first approach" to obtain the LGSPN for the system, the "system" net. This net represents the execution of the whole system and its performance model.

As the statecharts modeled are "flat", Definition 4.54 must be applied. This definition performs a composition of the component nets to obtain a first version of the net for the whole system. But remember that five sub-steps must be taken into account:

1. The statecharts have not sink states, then the "system" net is live.
2. Tokens in places P9|ini_wait, P110|ini_wait and P113|ini_wait, Figures 8.9 and 8.12, represent instances of users, browsers and web server

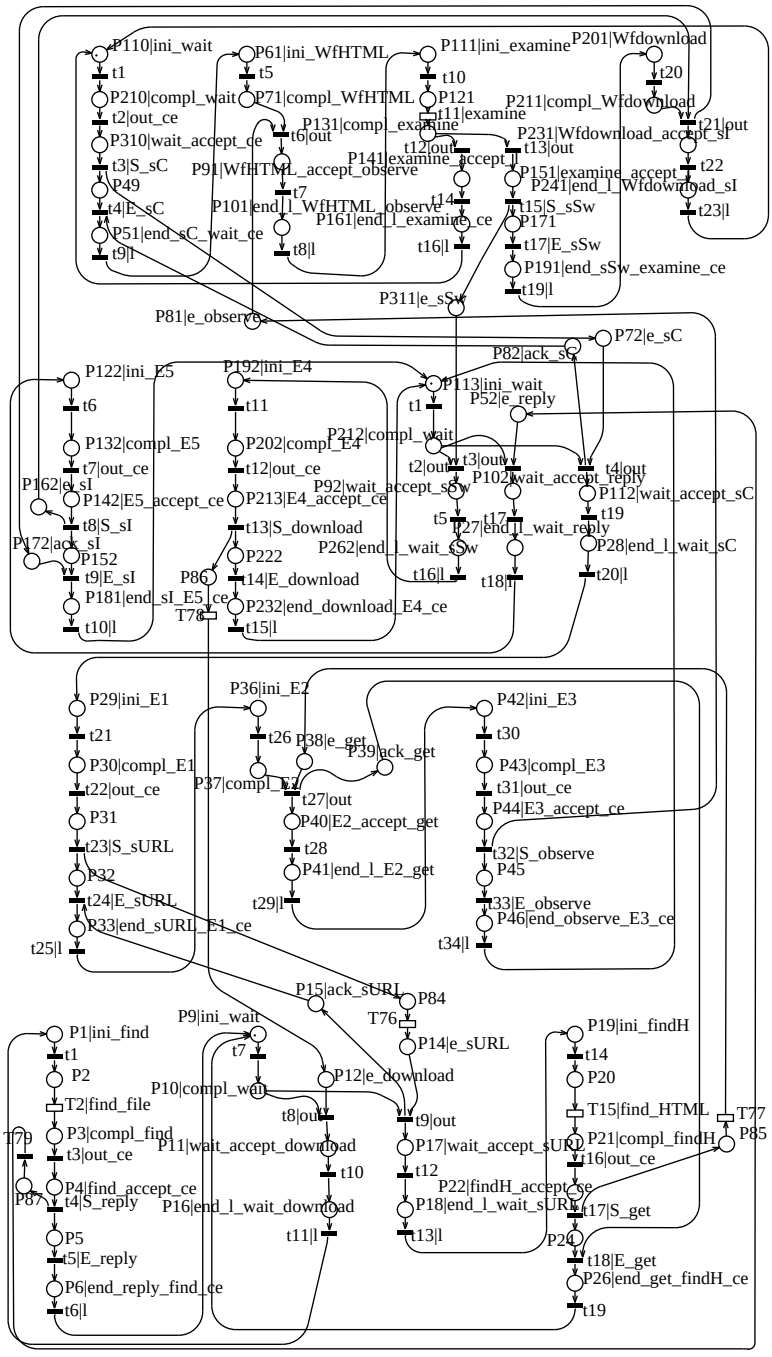


Figure 8.9: The LGSPN for the whole system.

process, respectively. They must be populated according with the experiment, then the number and color of these tokens will change from the first experiment (one user and one browser) to the second (several users and browsers).

3. The rate of the transitions t11|examine (t111|examine), T2|find_file and T15|find_HTML that represent activities must be calculated as it was explained in chapter 7. The first transition models the first modeling assumption (cfr. Section 8.1), since the second and the third refer to the third modeling assumption (cfr. Section 8.1).

4. It must be assigned probability to transitions t12|out and t13|out in Figure 8.9 (t121|out and t131|out in Figure 8.12). They refer to the second modeling assumption (cfr. Section 8.1), then they model different kind of users. Depending on the experiment, these values should change.

5. For each message exchanged between the browser and the web server (select_URL, get, download and its reply), a new place and a new timed transition must be created to represent the network delay, in the way explained in chapter 7. These places are respectively P84, P85, P86 and P87 in Figure 8.9 (P103, P104, P102 and P91 in Figure 8.12). The transitions are T76, T77, T78 and T79 in Figure 8.9 (T77, T78, T76 and T79 in Figure 8.12). The rate of the timed transitions refer to the fourth modeling assumption (cfr. Section 8.1).

In the following we comment the particularities of each one of the systems proposed, since the nets change from one to another.

LGSPN model for a system with one user and one browser

The LGSPNs in Figures 8.6, 8.7 and 8.8, that have been obtained by applying directly the translations proposed in chapter 4, model the component nets for the user, the browser and the web server, respectively. Then, it is not necessary to perform any change in them.

When these nets are composed, using Definition 4.54 over the set of places e_event and ack_event, the "system" net is obtained. According to sub-step 2 (defined in the first approach of section 7.2.2), the places P9|ini_wait, P110|ini_wait and P113|ini_wait must be populated with one token each one to represent one user, one browser and one web server process. Figure 8.9 represents the "system" net for this case.

LGSPN model for a system with several users and several browsers

The component LGSPNs for the user and the browser, that have been obtained by applying directly the translations proposed in chapter 4, must be converted into SWN. In this way when a colored token in the user net matches with a colored token in the browser net, it means that this user request will be served by this browser instance and never by another one. This behavior is achieved by marking, in the user net, place P110|ini_wait with the number of concurrent users that can access to the system; and by marking in the browser net, place P113|ini_wait with the number of concurrent browsers that can access to the system. The color is the same in both

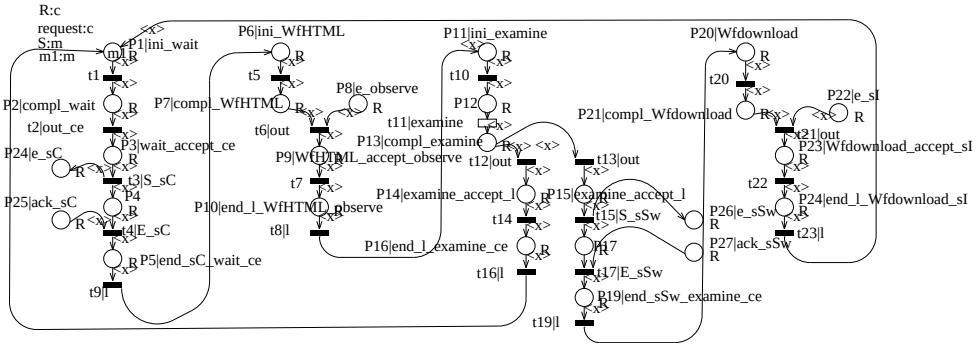


Figure 8.10: User colored LGSPN component

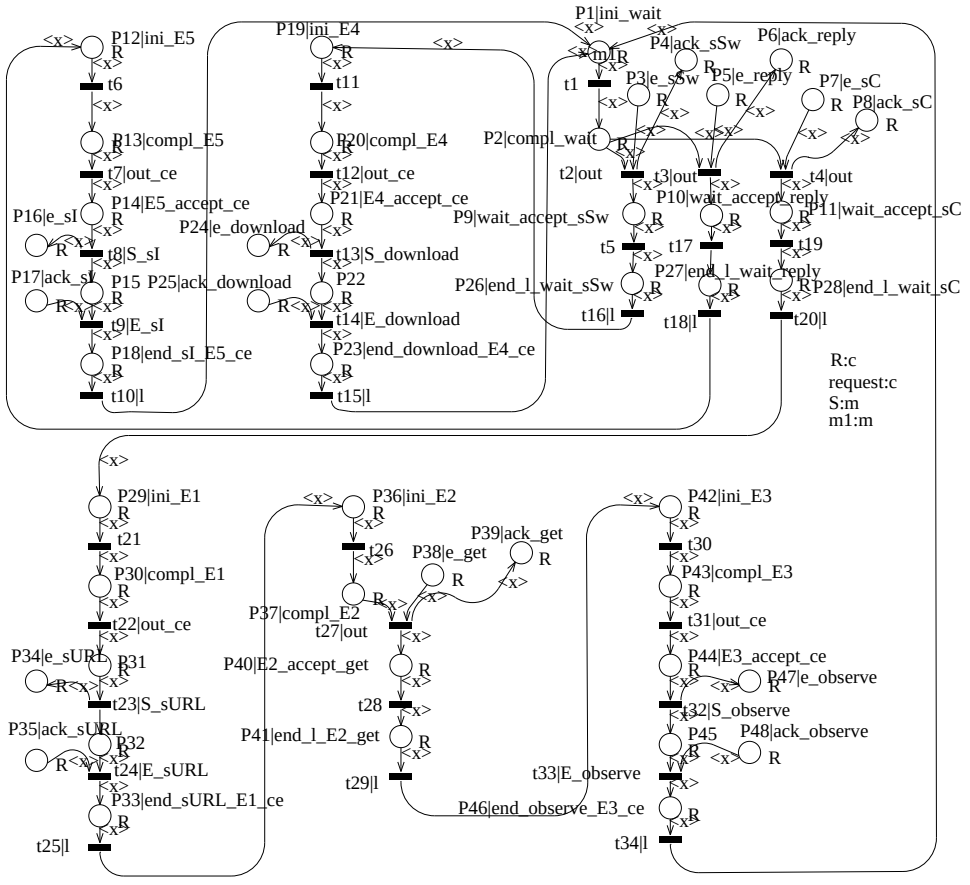


Figure 8.11: Browser colored LGSPN component.

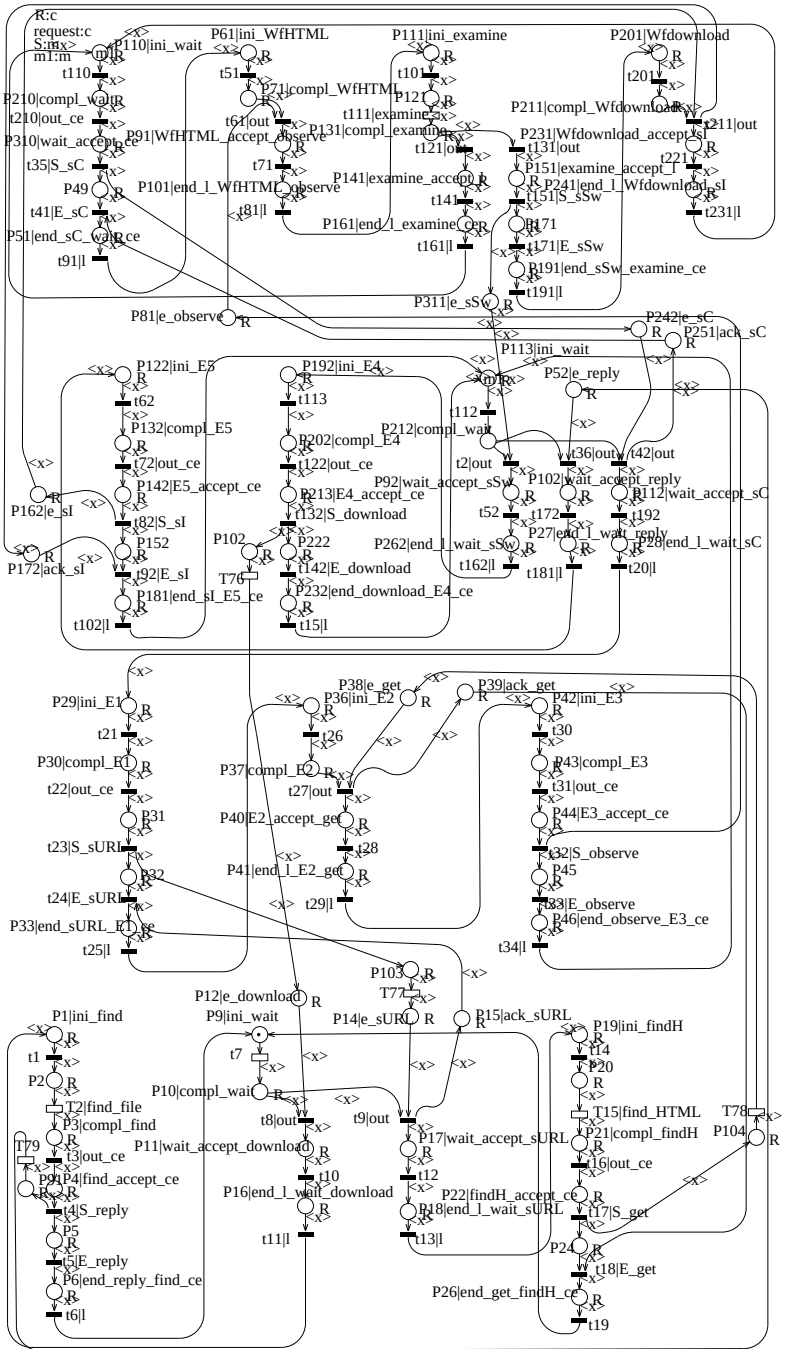


Figure 8.12: The colored LGSPN for the whole system.

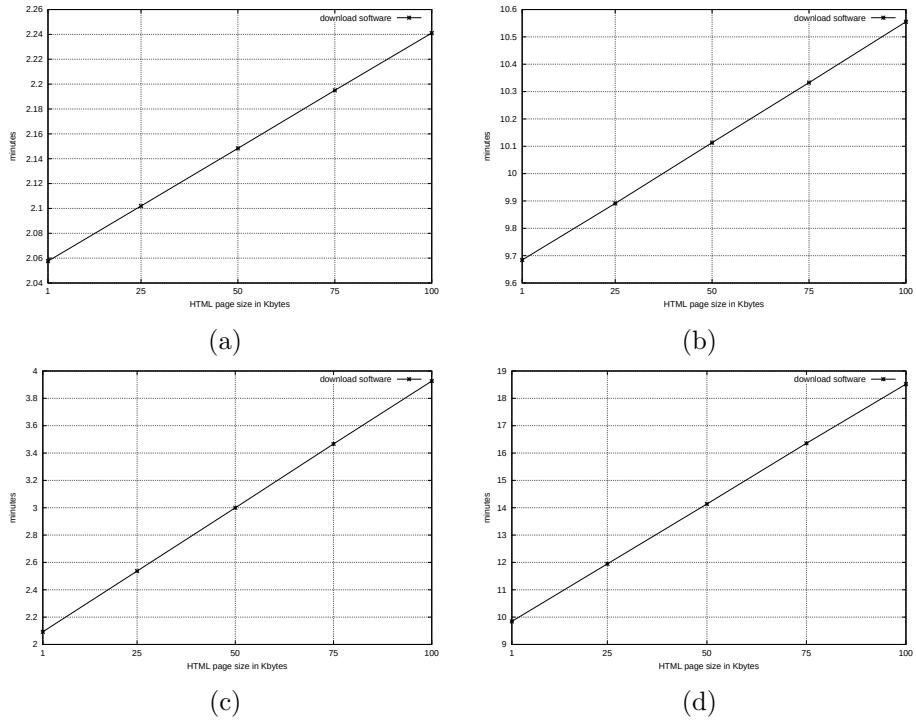


Figure 8.13: Response time of the Tucows-like system for different scenarios. (a) and (b) represent a "fast" connection speed, (c) and (d) a "slow" connection speed; (a) and (c) an "expert user" and (b) and (d) a "naive user".

places, then identifying each browser with a concrete user. Figures 8.10 and 8.11 represent these component nets.

On the other hand, the LGSPN for the web server remains as in the previous case since only one web server process is active.

In order to create the "system" net the rules one to four given in chapter 7 must be taken into account. The final net is shown in Figure 8.12.

8.2.3 Performance results

In this section we realize the last step of the process presented in chapter 7. As for the ANTARCTICA SRS, it is of our interest to obtain the system *response time* in the presence of a user/s request/s. Therefore the results do not correspond to a concrete scenario but to the whole system, then they have been obtained from the "system" nets in Figures 8.9 and 8.12 that represent the performance models of the two cases under study (one user and one browser and several users and browsers).

The transition whose throughput must be computed is t21|out since when it fires

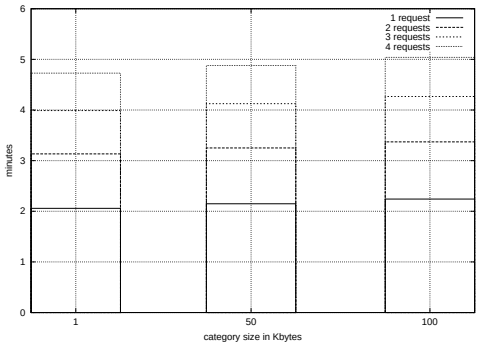


Figure 8.14: Response time of the Tucows-like system for "fast" connection speed, "expert users" and different number of requests.

a complete download has been performed. From this value, the response time is calculated in the same way as it was explained for the case of the ANTARCTICA SRS.

In the following, we describe the tests performed by giving concrete values to each one of the modeling assumptions presented in section 8.1. Obviously, each one has its counterpart in the ANTARCTICA SRS, then we have used the same values:

1. We have considered an "expert" user that spent 10 sec. reading an HTML page. Transition t11|examine (t111|examine) model this behavior.

2. We have tested an "expert" user that request a mean of 10 categories until s/he finds the desired software and a "naive" user that tests a mean of 50 categories. Transitions t12|out and t13|out (t121|out and t131|out) represent this behavior.

3. We have taken into account two cases for the net speed: a net with a speed of 100 Kbytes/sec. ("fast" connection speed) and a net with a speed of 10 Kbytes/sec. ("slow" connection speed). They are modeled by transitions T76, T77, T78 and T79 (T77, T78, T76 and T79).

One user and one browser
Results have been obtained from the net in Figure 8.9. Although the tests performed are the same as in the ANTARCTICA SRS, only one curve for each experiment is obtained. It is a consequence of considering the system designed without using mobile agents, then it makes no sense to change the probability to perform either RPC or travel as in the ANTARCTICA SRS.

It can be observed in Figure 8.13 that the results follow the same trends as in the ANTARCTICA SRS but smaller response times are obtained. Then it can be concluded that with the assumptions given, the Tucows-like approach behaves better with respect to response time than the ANTARCTICA SRS.

Several users request several browsers
In this case the results have been obtained from the performance model given in Figure 8.12. They are presented in Figure 8.14 for a test that represents"fast" connection speed, three different sizes of categories (1Kbyte, 50 Kbytes and 100 Kbytes) and several "expert" users (until four users requesting for pieces of software). It can be observed that when the number of requests is increased, the response time is also increased. Nevertheless this increment is not proportional, which allows to think that some parallelism among the tasks can be achieved.

It could be argued that all the tests performed are favorable to the Tucows-like approach since all the scenarios presented are faster in the Tucows version. Actually, the ANTARCTICA SRS has been designed for wireless connection systems, then high speed connection time is a drawback for this approach. In the next section a more realistic performance comparison is realized.

8.3 Performance comparison

As we pointed out in the introduction of the chapter, it is of our interest to compare the proposed designs for the ANTARCTICA SRS and the Tucows-like systems in order to know how much time the network connection needs to be open to retrieve a software product and also the impact of the mobile and intelligent agents in the design of a software retrieval system. In this section we consider scenarios closely related to the wireless environments, i.e. slow network connection speed, in contrast to the tests performed in section 7.2.3 for the ANTARCTICA SRS and in section 8.2.3 for the Tucows-like system. In this way the comparison can be performed in the field where the ANTARCTICA SRS has been proposed.

The results presented in this section have been obtained from the SWNs which model the Tucows-like system and the ANTARCTICA SRS (Figures 8.12 and 7.19 respectively).

8.3.1 Study of the network connection time

So, we want to study how much network connection time is necessary to download a software product, *network time*. As in the previous analysis, by computing the steady-state distribution of the isomorphic *Continuous Time Markov Chain* (CTMC) with *GreatSPN* [CFGR95], the throughput of the target transitions is obtained. The inverse of the previous result gives the network time. Remember that these transitions are t6|E_sSs in the ANTARCTICA SRS and t21|out in the Tucows-like system.

To study the network time in both systems, we have developed a test taking into account the following scenarios:

1. Two different kinds of users have been considered: a user who spends 10 sec. to study the information presented by the system (web page or a software catalog) and a user who spends 60 sec. in that task (modeling the information processing speed of the user).

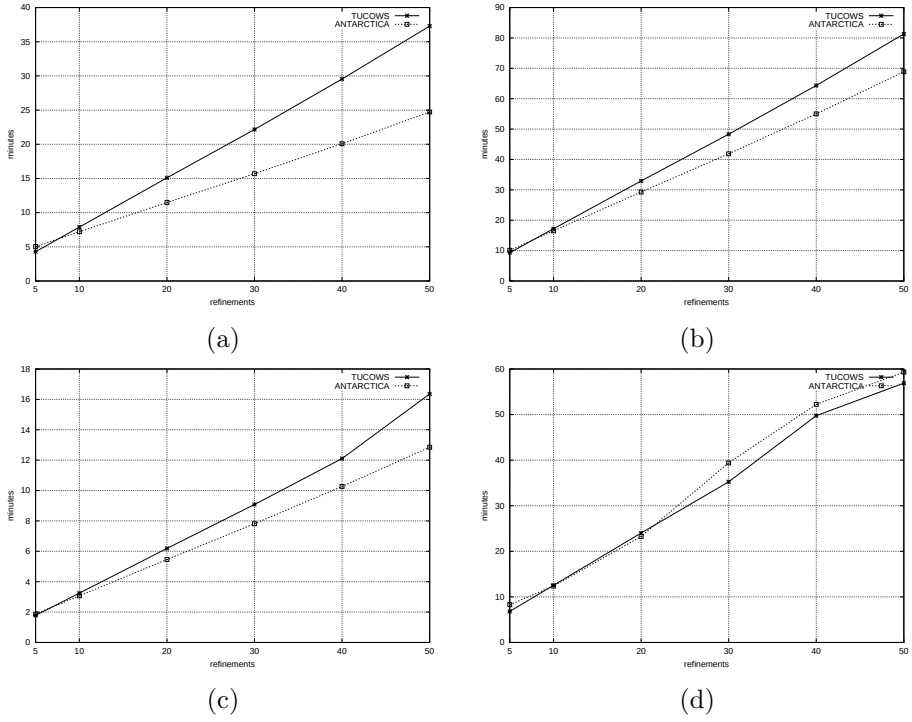


Figure 8.15: Network time for different scenarios: (a) and (b) represent a net speed of 1 K/sec., (c) and (d) represent a net speed of 5 K/sec., (a) and (c) represent a "user delay" of 10 sec., (b) and (d) represent a "user delay" of 60 sec. The intelligence of the ANTARCTICA's browser has been set to 70%.

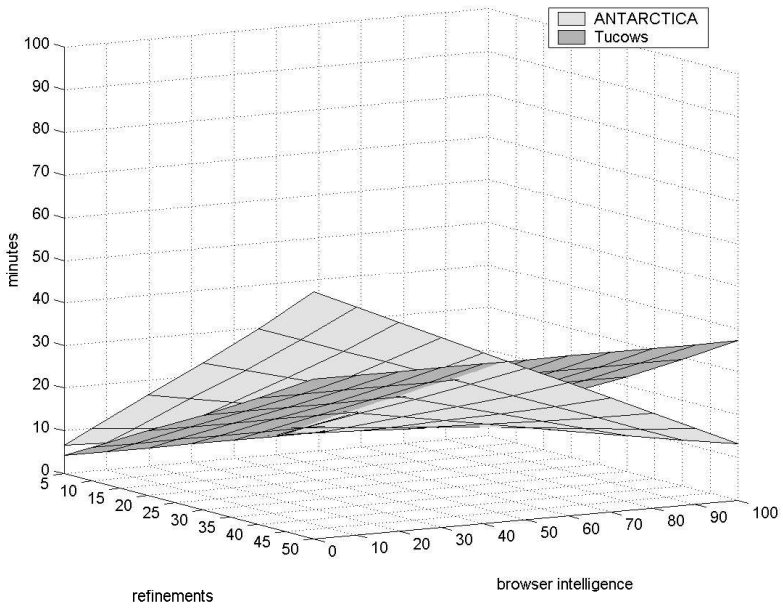
2. To test the *user refinement request*, we have considered six different possibilities. A user requesting a mean of 5, 10, 20, 30, 40 and 50 refinements[1] (modeling different expertise of the user).

3. We have considered two cases for the *net speed*: 1 K/sec. and 5 K/sec. By considering these low speed values we want to compare the performance of both approaches in a wireless computing environment (*real* GSM network speed is around 800 bytes/sec.).

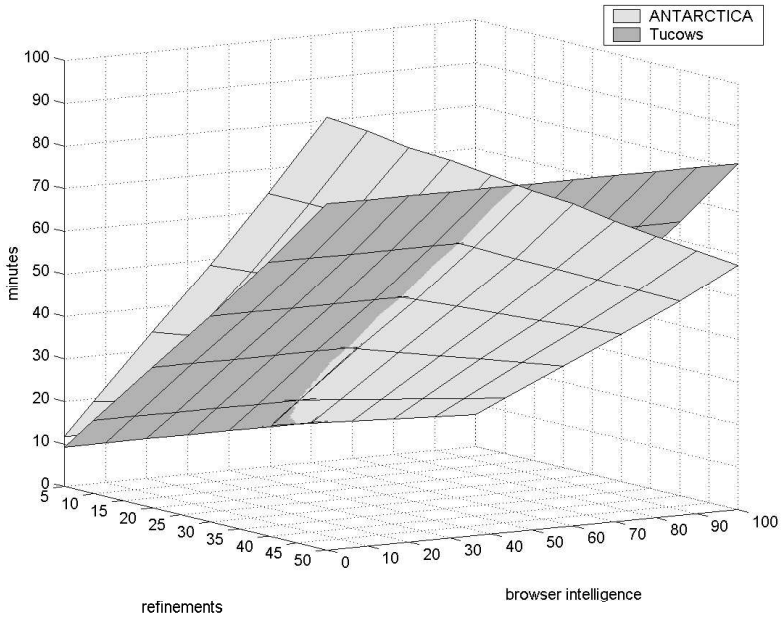
For the ANTARCTICA SRS, we have also considered:

1. A browser which does not need to ask for information to the software manager agent the 70% of the times that the user asks for a refinement. When the browser needs information, it requests the information by a *remote procedure call* (RPC).

[1]We mean by refinement a category selection in a Tucows-like system and a catalog refinement in the ANTARCTICA SRS.



(a)



(b)

Figure 8.16: (a) and (b) represent the same scenarios than Figures 8.15.a and 8.15.b, respectively, but varying the intelligence of the ANTARCTICA SRS browser.

2. *The size of the catalog* obtained by the browser is 50 K.

Figure 8.15 shows network time (in minutes) for the Tucows-like system and the ANTARCTICA SRS in different scenarios. Concretely in Figure 8.15.b we can observe that when the net speed is 1 K/sec., the user is naive and performs 50 refinements (the worst case), then the ANTARCTICA SRS is more than ten minutes faster than the Tucows-like system. The same results are obtained if the user is expert, see Figure 8.15.a. However, when the net speed is increased to 5 Kbyte/sec. (see Figure 8.15.d), the differences decrease, and if the user performs more than thirty refinements the ANTARCTICA SRS behaves worse. In conclusion, we can say that the ANTARCTICA approach behaves much better than a Tucows-like system for low network speed. Differences between the two approaches become less significant for a higher network speed. Taking this analysis as basis we could estimate which approach is better for a given situation.

8.3.2 Comments on the impact of the mobile agents

The study of the impact of the mobile and intelligent agents in the design of a software retrieval system in our context can be enunciated as "how much intelligent the browser agent in the ANTARCTICA SRS must be in order to obtain the same or better results than the Tucows-like system".

About the intelligence of the ANTARCTICA SRS browser agent, Figure 8.16 gives us interesting results. This figure shows the same scenarios as Figure 8.15.a and 8.15.b, but varying the intelligence of the ANTARCTICA SRS browser, from a browser that needs to ask for information the 100% of the times (dummy browser) to a browser that needs to ask for information the 0% of the times (oracle browser). When the intelligence of the browser is less than 40 (it does not need to ask for information the 40% of the times) the ANTARCTICA SRS behaves worse than the Tucows-like system. However, when the intelligence of the browser does not need to ask for information the 40% of the times or more, then ANTARCTICA SRS obtains similar or better results than a Tucows-like system.

8.3.3 Conclusions

As a conclusion of the tests performed in the previous sections, we can affirm that the ANTARCTICA SRS behaves better than the Tucows-like system when the speed of the net is slow, less than 1 Kbyte/sec.. So, the ANTARCTICA SRS is more appropriate than the Tucows-like system to retrieve software efficiently in wireless environments. It must be also taken into account that the intelligence of the browser agent is a must, because if this agent does not solve the requests of the user at least the forty per cent of the times, then the ANTARCTICA SRS cannot obtain better results than the Tucows-like system.

8.4 The POP3 mail protocol

This example has as a novelty value with respect to the other examples: it incorporates the use of the activity diagram in the pa-UML stage of our proposal to model the details of a concrete activity. Also, a performance model will be obtained not only for a complete execution of the system but for a particular execution of it, which also constitutes a novelty in this work, leading the "second approach" of the "modeling with stochastic Petri nets" stage of our approach as described in chapter7.

The Post Office Protocol Version 3 [MR94] (POP3 protocol) specifies an *Internet standard track protocol for the Internet community.* In short, the POP3 protocol *permit a client host to dynamically access a mail on a server host in a useful fashion.*

In the following the POP3 protocol is modeled and analyzed using our proposal. Since the previous examples were developed following in detail each step of the proposal, this one will be addressed assuming that the reader is already familiarized with the proposal.

8.4.1 Modeling the system

It must be assumed the existence of a user in a client host trying to check her/his mail that resides in a remote host, the POP3 server host.

The behavior of the system is rather intuitive. Upon the arrival of the user's request to check her/his mail; first, the client host tries to establish a TCP connection with the server via port 110. If succeeds (reception of a greeting message), both (client and server hosts) begin the authentication (authorization) phase. The client host sends the username and his/her password through a USER and PASS command combination. For the sake of simplicity, usage of the APOP command has not been contemplated here.

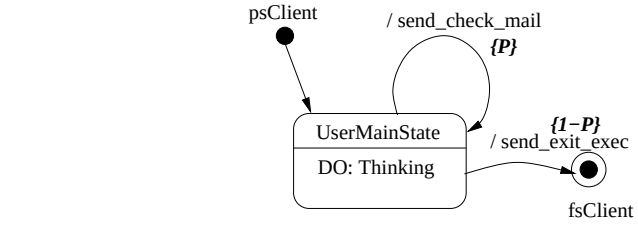
If the server host has answered with a positive status indicator ("+OK") to both messages, then the POP3 session enters the transaction state (phase). Otherwise (e.g., the password doesn't match the one specified for the username), it returns to the beginning of the authorization phase.

In the transaction phase, the client host checks for new mail using the LIST command. If there is any, the client host obtains every e-mail by means of the RETR and DELE commands. It must be noted that, for simplicity, potential errors have not been considered here; thus, no negative status messages ("-ERR") are modeled.

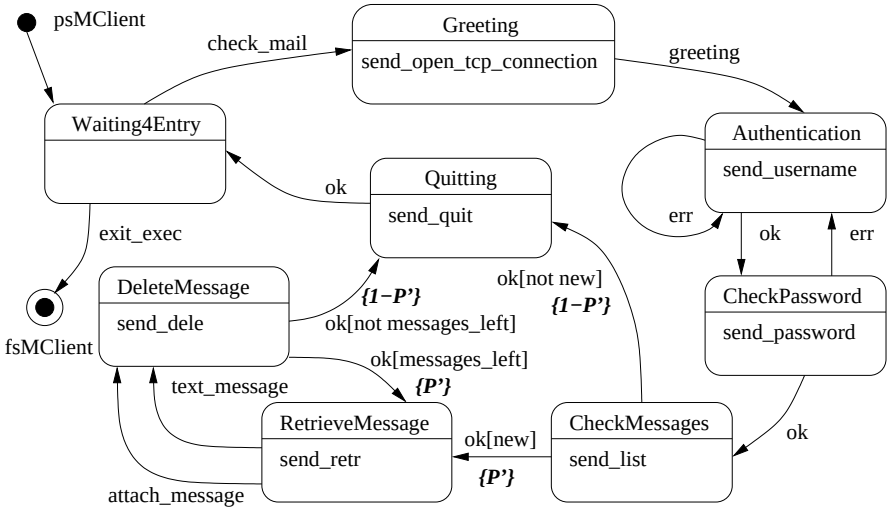
Once every e-mail has been downloaded, the mail client host issues a QUIT command to end the interaction. This provokes the POP3 server host to enter the update state and release any resource acquired during the transaction phase. The protocol is ended with a goodbye ("+OK") message.

pa-UML models

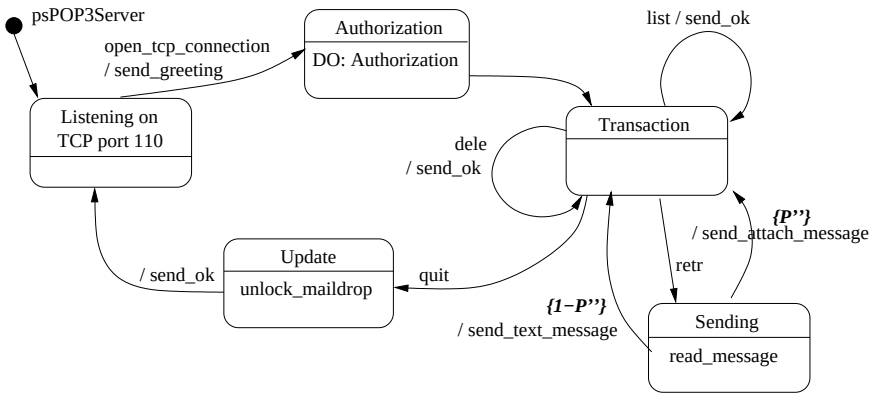
The behavior of the three participant objects, the user, the client host and the server, host has been modeled through the statecharts in Figures 8.17(a),(b) and (c), respectively.



(a)



(b)



(c)

Figure 8.17: Statecharts for: (a) the user, (b) the client host, (c) the server host.

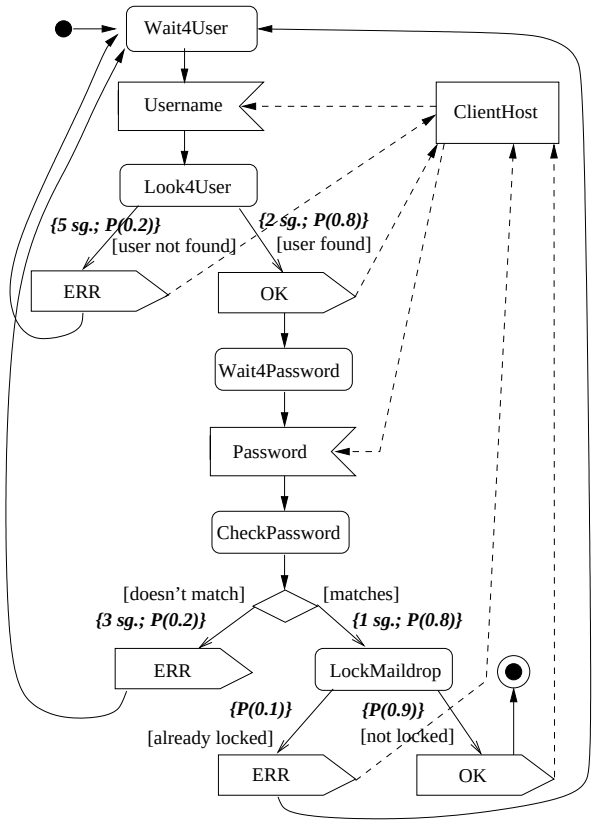


Figure 8.18: Activity diagram for the authorization activity in the server host.

- The statechart of the user shows that the system can be parametrized to obtain different performance figures varying the probability of execution of the check mail event (send_check_mail).

- The statechart of the client host introduces the following performance parameters: first, the probability to find new messages in the transaction phase (ok[new]); second, the probability to find a new message while the retrieving process (ok[messages_left]).

- The statechart of the server host with respect to performance allows to model different distributions in the number of messages (text or attach) (send_text_message and send_attach_message). Moreover, the activity Authorization is rather relevant to the system performance. Therefore, it is necessary to model the actions performed within. Here we will use an activity diagram (see Figure 8.18), although it may be more useful in cases where there is not such a strong external event dependence (e.g., 'internal' operations). The activity

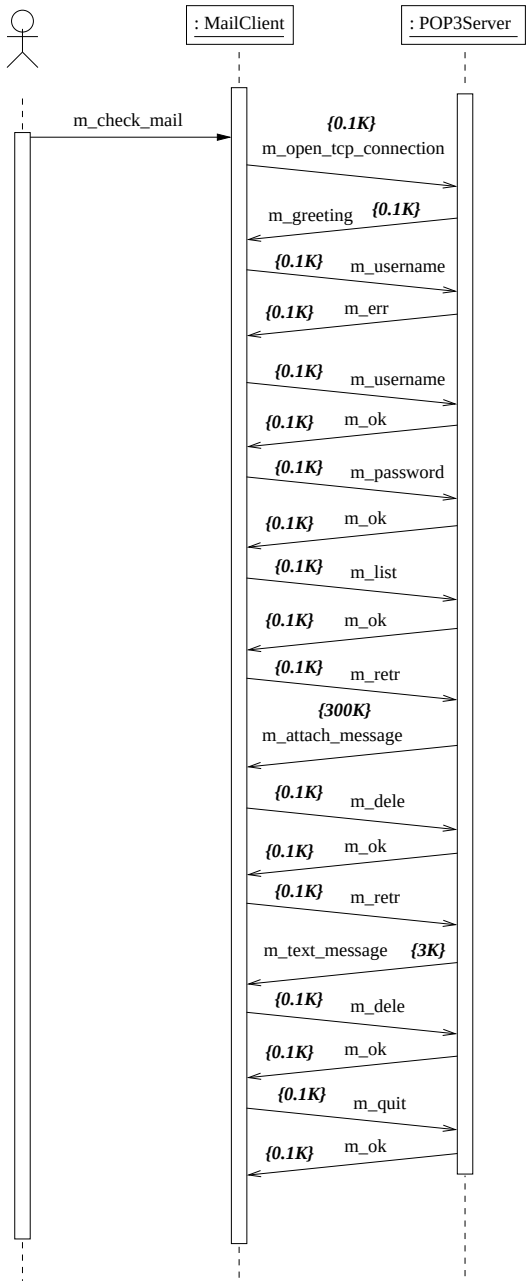


Figure 8.19: Sequence diagram for a particular execution of the POP3 protocol system.

could have been described extending the statechart but, in general, activity diagrams provide some additional expressiveness [LGMC02a] for certain tasks.

Finally, we use a sequence diagram to obtain performance analytical measures in a certain context of execution. Figure 8.19 shows an example of interaction between both the server and the client hosts. Some results for this particular scenario will be obtained in section 8.4.2. As we said, the analysis performed for the ANTARCTICA SRS and the Tucows systems do not make use of this feature of our approach.

LGSPN models

From the statecharts in Figures 8.17(a),(b) and (c), the component LGSPNs for the user, the client host and the server host can be obtained. Since all the statecharts are "flat" (i.e. they do not include any of the features that were developed in chapter 5), then these component nets are obtained from the translation given in chapter 4. Figures 8.20, 8.22, 8.23 show respectively the component nets for the user, the client host and the server host.

The activity diagram that represents the specification of the server::Authorization activity (see Figure 8.18) must be also translated into a component LGSPN. In this case the translation proposed in chapter 6 is of interest. Figure 8.24 shows the component LGSPN obtained for this activity diagram.

Following our proposal, all these component nets must be composed in order to obtain a LGSPN that represents a performance model. For this system we are going to use the first and the second approaches proposed in section 7.2.2, i.e. applying the first approach a LGSPN will be obtained that represents a performance model for the whole system, while applying the second approach it will be obtained a LGSPN that represents a performance model for a particular execution of the system, concretely the execution performed by the sequence diagram in Figure 8.19. These nets are not included in figures since the high number of places, states and transitions make too difficult to read them, it is necessary the GreatSPN front-end (or a similar one) to manage them.

8.4.2 Performance results

Once the final LGSPN models are obtained, performance estimates can be computed leading the third step of the approach. The results presented in this section have been obtained from the LGSPN that represents the whole system behavior and from the LGSPN that represents the concrete scenario. Figure 8.21 shows some results for both cases.

The analysis performed for the first approach (whole system behavior) is shown in Figure 8.21.a. The graph represents the effective transfer rate of the client when checking mail (maximum transfer rate: 56 Kbps). Note that higher amounts of data minimizes the relative amount of time spent by protocol messages.

On the other hand, the analysis performed for the second approach (execution of the sequence diagram in Figure 8.19) is shown in Figure 8.21.b. This graph represents

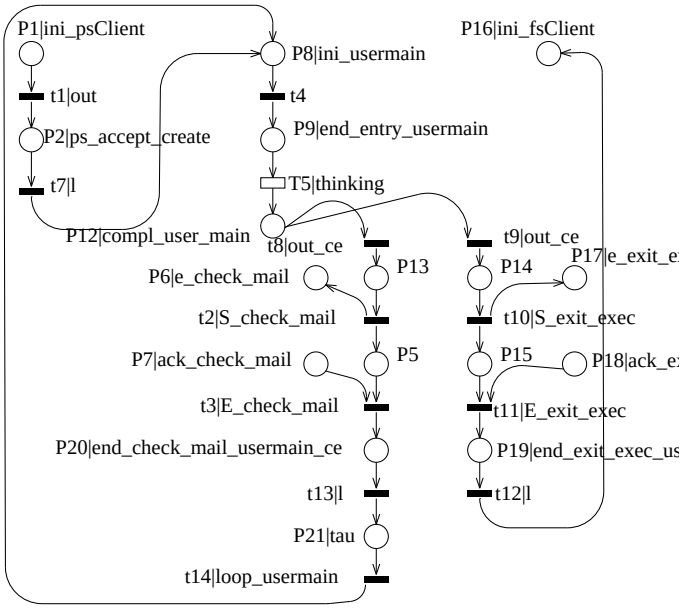
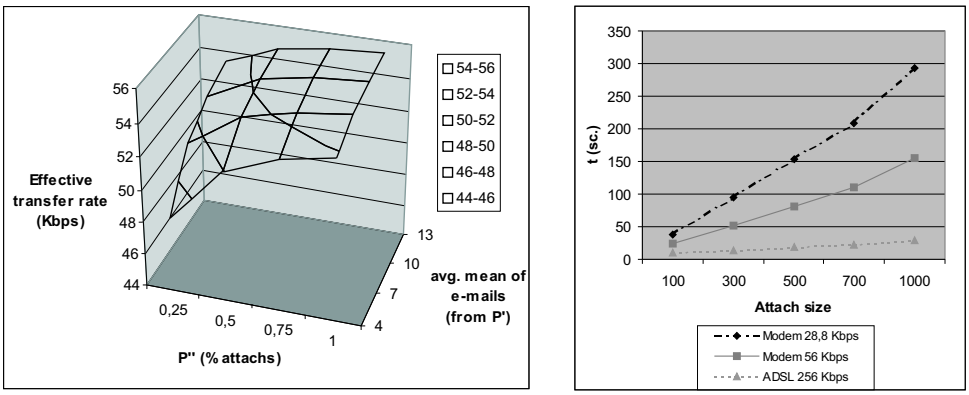


Figure 8.20: User LGSPN component.



(a) (b)

Figure 8.21: Results for the POP3 protocol.

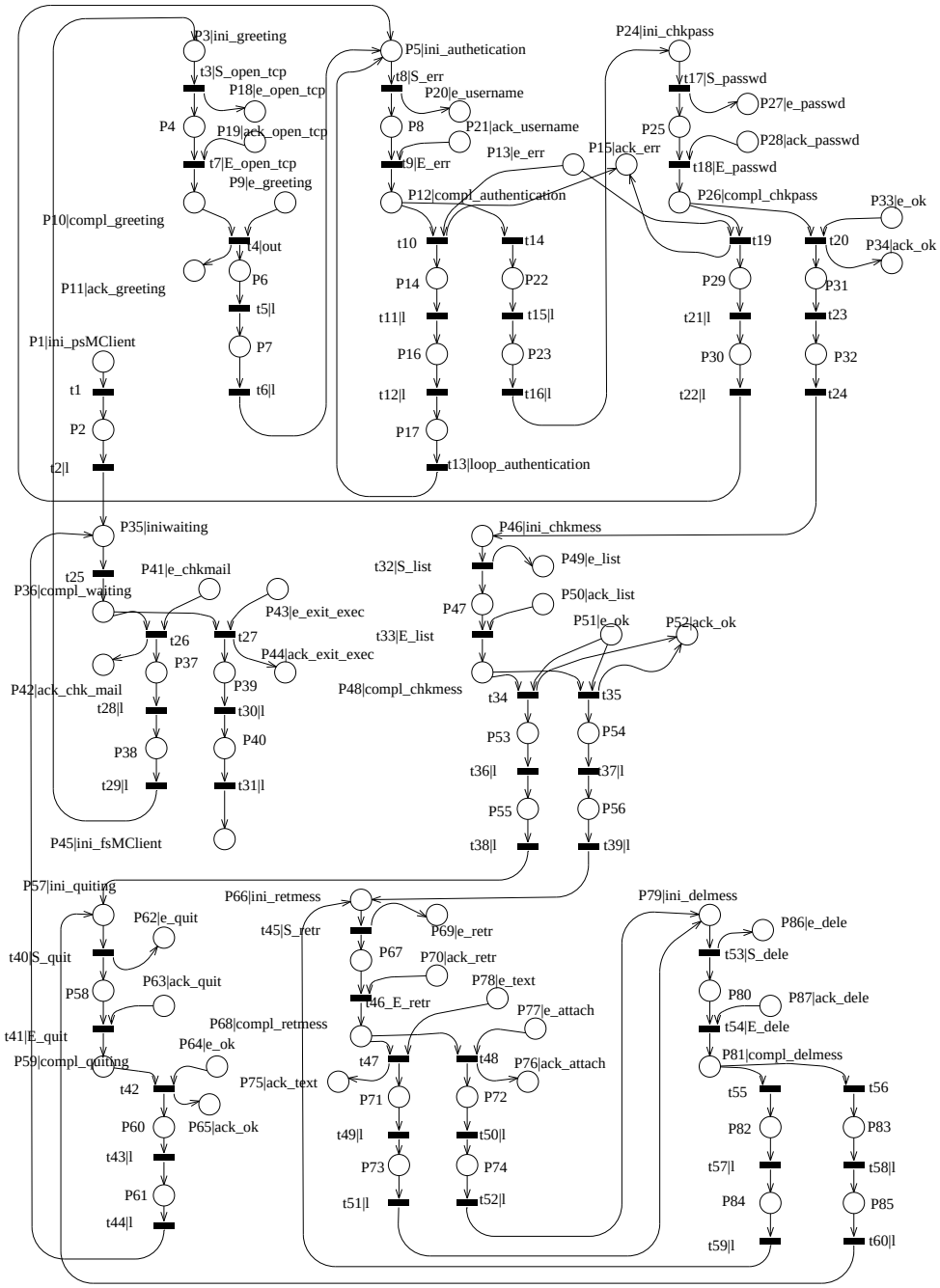


Figure 8.22: Client host LGSPN component.

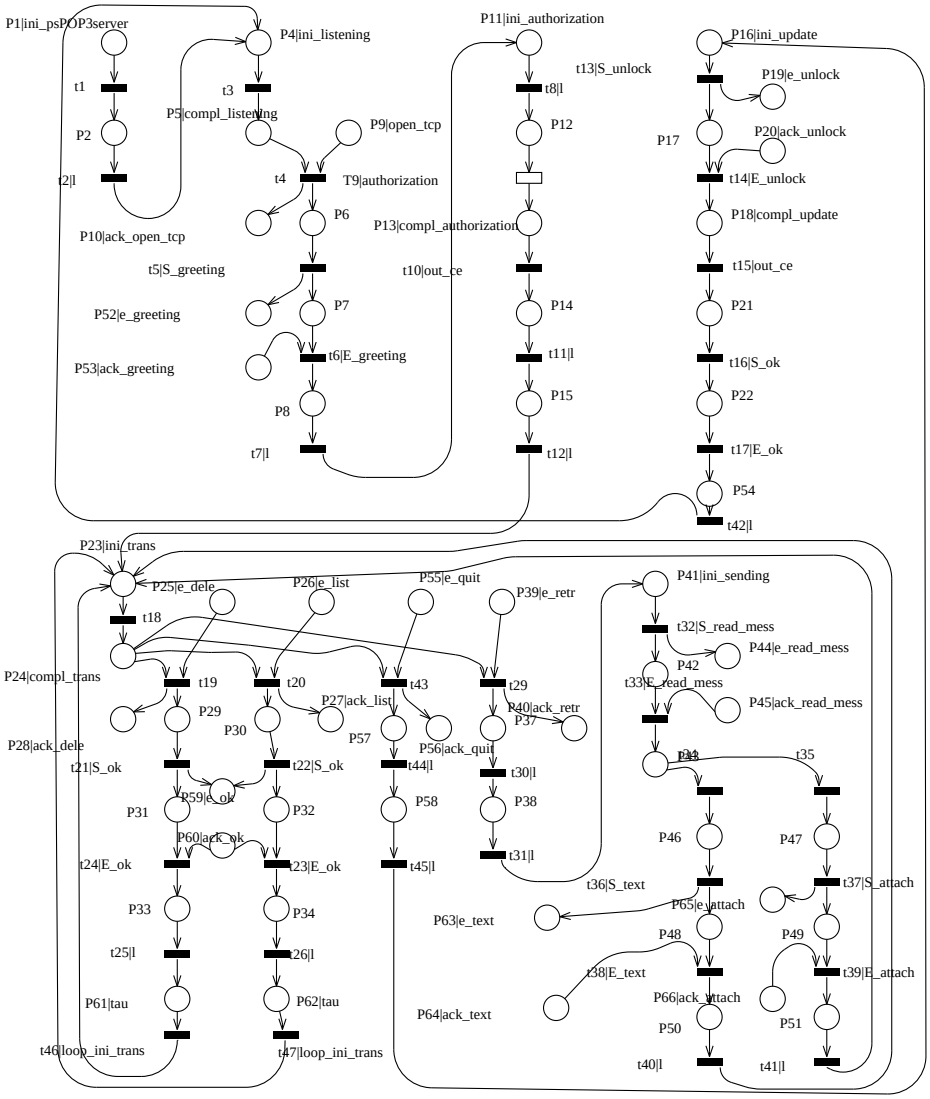


Figure 8.23: Server host LGSPN component.

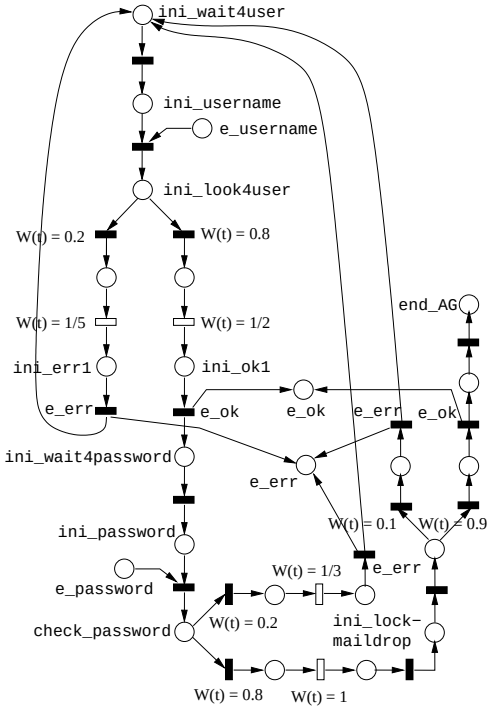


Figure 8.24: Authorization activity LGSPN component.

the time cost of executing the interaction in function of different attach file sizes and maximum network speeds. As it could be supposed, high performance networks respond in a better way to the increment of the file size. Obviously, the results are close to the reality.

8.5 Conclusions

In this chapter we have applied the software performance process given in this thesis to two software systems, therefore obtaining performance models for both systems.

The conclusions for this chapter can be summarized as follows. Firstly, our intention to completely apply the proposal developed in this work has been reached. We say "completely" because introducing the POP3 protocol, the translation of the activity diagrams and the performance model gain for a particular scenario have been applied.

Secondly, the performance analysis carry out to compare the software retrieval systems and to study the POP3 protocol have been interesting enough to obtain conclusions in both fields, that can be summarized as follows.

The performance comparison between the ANTARCTICA SRS and the Tucows-

like system has shown that:

- Software proposals for specific fields, such as ANTARCTICA SRS to retrieve software in wireless media, can be of importance. But performance analysis is necessary to find the benefits they can promote, and our proposal is right enough to be used with this purpose, at least to find the trends given by the systems. Concretely, through our proposal a threshold has been found over which the ANTARCTICA SRS behaves worst than conventional software retrieval systems.

- The use of mobile and intelligent agents in the field of software retrieval systems is of interest, but it is of importance to determine how much "intelligent" the agents should be in order to obtain acceptable results. Our approach has been shown suitable for this task.

The performance analysis of the POP3 mail protocol has been useful to confirm analytically the results obtained every day when checking mail: Higher amount of data minimize the relative amount of time spent by protocol messages and high performance networks behave better when file size is increased.

Finally, the use of the approach with exact techniques of analysis shows that in some cases they cannot scale fine when colored nets represent the system performance model. Therefore, the use of the approximate analysis techniques and bounds commented in chapter 7 could be of interest.

Chapter 9

Final conclusions and future work

At the beginning of this work, we formulated as a main objective "to provide software engineers with the tools to achieve performance estimates in the early stages of the software life cycle". It was also an objective that these tasks were accomplished by software engineers with nothing (or minimum) training in the underlaying mathematical formalisms that are commonly used to represent performance models (Markov chains, queuing networks, stochastic process algebra, stochastic Petri nets). Also, we cited that if we were able to obtain these performance models as a "by-product" of the software life-cycle, much effort would be avoided for the software and performance engineers.

Considering the previous premises, the novel contributions of this thesis to the *state of the art* can be summarized as follows:

1. A compositional semantics in terms of stochastic Petri nets for the UML state machines has been gained. This semantics actually represents our interpretation of the UML state machines. In the same fashion, semantics for the UML activity graphs has been obtained. Since the semantics is aimed at performance evaluation, it is possible to analyze system quantitative properties as well as to validate qualitative properties, when the system is described as a set of statecharts and its activities detailed as activity diagrams.

2. Quantitative and qualitatively properties can also be validated for particular executions of the system. That is possible when these executions are represented by sequence diagrams.

3. A language to describe performance features of software systems has been given by increasing the UML notation. It allows to model the dynamics and the load of a wide-range of complex software systems.

4. This language together with the translation of the UML behavioral diagrams allow software engineers to obtain performance models as a "by-product" of the software life cycle, since they are combined in a software performance engineering process that builds on them. A pattern-based extension of the process has also been proposed.

5. The application of the previous contributions to the study of performance issues in the field of the mobile and intelligent agents has given interesting results about the design of software retrieval systems in a wireless environment.

All these contributions have been obtained by the individual realizations of each chapter in the following manner.

In chapter 3, the role of the UML behavioral diagrams related to performance evaluation was studied. Then in this chapter, our contribution to augment the UML notation to represent performance parameters was presented as well as the possibilities and constraints of each diagram to represent system load, system usage or routing rates. Also, our functional interpretation of each diagram was given.

In chapters 4 and 5, the translation functions from the abstractions in the UML state machines metamodel into the GSPNs elements were given.

In chapter 6, the translation functions for the elements in the activity graph package into the GSPN formalism were given.

In chapter 7, the steps to obtain a performance model from the augmented UML designs were shown. Two approaches were given: First, when the system is represented by a set of statecharts (may be its actions refined with activity diagrams) then the performance model represents a complete execution of the system; second, when the system is modeled by a set of sequence diagrams and a set of statecharts (may be its actions refined with activity diagrams), then the performance model allows to obtain performance measures for the particular execution of the system represented by a sequence diagram. The complementary approach of the design patterns was also presented.

In chapter 8, the approach was applied to different real systems.

The work detailed in this thesis has been the subject of several publications in international workshops, conferences, book chapters and journals. In the following we enumerate them in a chronological order.

The article in [MCM00b] developed the preliminary ideas of the translation of the UML state machines and the sequence diagram into stochastic Petri nets, also with that pragmatic approach the Antarctica SRS was analyzed. The application of the design patterns to model software performance issues was given in [MCM00a], leading the "performance patterns" proposal. The work in [MCM01b] improved some aspects of the approach given in [MCM00b] concerning the number of elements considered and the notation. Different features of the Antarctica SRS were compared, using the previous proposals, with the Tucows-like system in [MCM01a] to obtain performance results in wireless media. The previous work was selected to submit an extended version to a special issue of a journal [MCM03]. In [MBCD02] the formalization of the translation of the "flat" UML state machines into stochastic Petri nets was

formulated. The concepts related with the composite states have not been published yet. The composition of "flat" UML state machines by means of the sequence diagram in a formalization using stochastic Petri nets has been presented in [BDM02]; the part of this work that accomplishes the formal translation from the sequence diagram into a stochastic Petri net is not included in this thesis. The work in [LGMC02c] presents the analysis of an Internet protocol using the entire approach (translation of the state machines, sequence diagrams and activity diagrams). The formalization of the activity diagrams together with the definition of its performance role have been established in [LGMC02b].

Now we present some issues that can be addressed in the future to improve our approach from three different points of view: related to the UML diagrams, related to the pattern approach and concerning the notation to represent performance parameters. They are the following:

1. UML diagrams related improvements. Although our proposal includes an important number of UML diagrams used to model performance requirements (as we shown in chapter 2 when the state of the art was revised), it is true that the study and integration in the proposal of the following ones would improve it. Concretely:

   - The "infinite hardware resources" hypothesis implicit in our proposal could be avoided by modeling different hardware aspects of the system. Then the component and the deployment diagrams play a prominent role, since they offer the possibility to model issues such as the distribution of the software components in the hardware platform, the distribution of the hardware platform itself or the operative system resources. Therefore by modeling consistently these features of the system with the rest of the proposal, this hypothesis would be overcame.
   - The class diagram can be exploited to represent some aspects of the load of the population in the system.
   - With respect to activity diagrams, conditional forks and more complex external event processing support, especially important to resolve the problem of 'uninterruptible' activities due to the use of action states, can be studied.

2. Design patterns related improvements. The "performance patterns" proposal is open to be improved from several perspectives. Performance features of the design patterns in the literature can be studied and augmented. Patterns to solve concrete and common performance problems in software design could be addressed.

3. Notation related improvements. The extension of the UML to represent more performance issues, is in our opinion open. We suggest that this work should be followed from the UML performance profile viewpoint [Obj02]. It does not mean that our proposal is not able to address new challenges but as we declare, we

advocate for the standards. Therefore we think that that work can be continued or even merged with ours to define role of performance of some other diagrams in that proposal.

Relevant Publications Related to the Thesis

[MCM00a] J. Merseguer, J. Campos, and E. Mena. A pattern-based approach to model software performance. In *Proceedings of the Second International Workshop on Software and Performance (WOSP2000)*, pages 137–142, Ottawa, Canada, September 2000. ACM.

[MCM00b] J. Merseguer, J. Campos, and E. Mena. Performance evaluation for the design of agent-based systems: A Petri net approach. In Mauro Pezzé and Sol M. Shatz, editors, *Proceedings of the Workshop on Software Engineering and Petri Nets, within the 21st International Conference on Application and Theory of Petri Nets*, pages 1–20, Aarhus, Denmark, June 2000. University of Aarhus.

[MCM01a] J. Merseguer, J. Campos, and E. Mena. Performance analysis of internet based software retrieval systems using Petri nets. In M. Meo, T. Dahlberg, and L. Donatiello, editors, *Proceedings of the 4th ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems, within the 7th International Conference on Mobile Computing and Networking*, pages 47–56, Rome, July 2001. ACM.

[MCM01b] J. Merseguer, J. Campos, and E. Mena. A performance engineering case study: Software retrieval system. In R. Dumke, C. Rautenstrauch, A. Schmietendorf, and A. Scholz, editors, *Performance Engineering. State of the Art and Current Trends*, Lecture Notes in Computer Science, (LNCS) Vol. 2047, pages 317–332. Springer-Verlag, Heidelberg, 2001.

[BDM02] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In *Proceedings of the Third International Workshop on Software and Performance (WOSP2002)*, pages 35–45, Rome, Italy, July 2002. ACM.

[MBCD02] J. Merseguer, S. Bernardi, J. Campos, and S. Donatelli. A compositional semantics for UML state machines aimed at performance evaluation. In A. Giua and M. Silva, editors, *Proceedings of the 6th International Workshop on Discrete Event Systems*, pages 295–302, Zaragoza, Spain, October 2002. IEEE Computer Society Press.

[LGMC02c] J.P. López-Grao, J. Merseguer, and J. Campos. Performance engineering based on UML and SPNs: A software performance tool. In *Proceedings of the Seventeenth International Symposium On Computer and Information Sciences (ISCIS XVII)*, pages 405–409, Orlando (Florida), USA, October 2002. CRC Press.

[MCM03] J. Merseguer, J. Campos, and E. Mena. Analysing internet software retrieval systems: Modeling and performance comparison. *Wireless Networks (WINET)*, 9(3):223–238, May 2003.

Bibliography

[AABI00] F. Andolfi, F. Aquilani, S. Balsamo, and P. Inverardi. Deriving performance models of software architectures from message sequence charts. In Woodside et al. [WGM00], pages 47–57. ISBN 1-58113-195-x.

[ABI01] F. Aquilani, S. Balsamo, and P. Inverardi. Performance analysis at the software architectural design level. *Performance Evaluation*, 45:147–178, July 2001.

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[AMBC84] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.

[AMBC+95] 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.

[AMBCC87] M. Ajmone Marsan, G. Balbo, G. Chiola, and G. Conte. Generalized stochastic Petri nets revisited: Random switches and priorities. In *Proceedings of the International Workshop on Petri Nets And Performance Models*, pages 44–53, Madison, WI, USA, August 1987. IEEE Computer Society Press.

[Bal98] G. Balbo. *Exponential Stochastic Petri Nets*, chapter 9, pages 304–341. In Balbo and Silva [BS98], September 1998.

[BC98] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '98 / PERFORMANCE '98. Performance Evaluation Review 26(1)*, pages 151–160, Madison, Wisconsin, USA, June 22-26 1998.

[BDM02] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In Inverardi et al. [IBB02], pages 35–45. ISBN 1-58113-563-7.

[Ber87] G. Berthelot. Transformations and decompositions of nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets 1986 - Part I*, volume 254 of *Lecture Notes in Computer Science*, pages 359–376. Springer-Verlag, Berlin, 1987.

[BJR99] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language*. Addison Wesley, 1999.

[BMMI98] W.J. Brown, R.C. Malveau, and H. W. McCormick III. *AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*. John Wiley and Sons, New York, 1998.

[BP01] L. Baresi and M. Pezzè. On formalizing UML with high-level Petri nets. In G.A. Agha, F. De Cindio, and G. Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets. State of the Art*, volume 2001 of *Advances in Petri Nets. Lecture Notes in Computer Science, (LNCS)*, pages 276–304. Springer-Verlag, Heidelberg, 2001.

[Bro87] F.P. Brooks. Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[BS98] G. Balbo and M. Silva, editors. *Performance Models for Discrete Event Systems with Synchronisations: Formalisms and Analysis Techiques*. Editorial KRONOS, 1998.

[BS01] S. Balsamo and M. Simeoni. On transforming UML models into performance models. In *Proceedings of the Workshop on Transformations in UML, ETAPS 2001*, 2001.

[BT81] A. Bertoni and M. Torelli. Probabilistic Petri nets and semi-Markov systems. In *Proceedings of the* 2^{nd} *European Workshop on Petri Nets*, pages 59–78, Bad Honnef, Germany, September 1981.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object Oriented Development. The Fusion Method*. Object Oriented. Prentice Hall, 1994.

[CAC+93] G. Chiola, C. Anglano, J. Campos, J. M. Colom, and M. Silva. Operational analysis of timed Petri nets and application to the computation of performance bounds. In *Proceedings of the* 5^{th} *International Workshop on Petri Nets and Performance Models*, pages 128–137, Toulouse, France, October 1993. IEEE-Computer Society Press.

[Cam98a] J. Campos. *Performance Bounds*, chapter 17, pages 587–635. In Balbo and Silva [BS98], September 1998.

[Cam98b] J. Campos. *Performance Measures and Basic Properties*, chapter 8, pages 283–302. In Balbo and Silva [BS98], September 1998.

[CCJS94] J. Campos, J. M. Colom, H. Jungnitz, and M. Silva. Approximate throughput computation of stochastic marked graphs. *IEEE Transactions on Software Engineering*, 20(7):526–535, July 1994.

[CDFH90] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On well-formed colored nets and their symbolic reachability graph. In *Proceedings of the* 11^{th} *International Conference on Applications and Theory of Petri Nets*, Paris, France, June 1990.

[CDFH93] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets for symmetric modelling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, November 1993.

[CDI01] V. Cortellessa, A. D'Ambrogio, and G. Iazeolla. Automated derivation of software performance models from case documents. *Performance Evaluation*, 45:81–105, July 2001.

[CDS99] J. Campos, S. Donatelli, and M. Silva. Structured solution of asynchronously communicating stochastic modules. *IEEE Transactions on Software Engineering*, 25(2):147–165, March 1999.

[CFGR95] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24:47–68, 1995.

[Çin75] E. Çinlar. *Introduction to Stochastic Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1975.

[CM00] V. Cortellessa and R. Mirandola. Deriving a queueing network based performance model from UML diagrams. In Woodside et al. [WGM00], pages 58–70. ISBN 1-58113-195-x.

[CTM98] J.M. Colom, E. Teruel, and Silva. M. *Logical Properties of P/T Systems and their Analysis*, chapter 6, pages 185–232. In Balbo and Silva [BS98], September 1998.

[DF96] S. Donatelli and G. Franceschinis. PSR Methodology: integrating hardware and software models. In J. Billington and W. Reisig, editors, *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, volume 1091 of *Lecture Notes in Computer Science*, Osaka, Japan, June 24-28 1996. Springer.

[DFJR98] J. Dilley, R. Friedrich, T. Jin, and J. Rolia. Web server performance measurement and modeling techniques. *Performance Evaluation*, (33):5–26, 1998.

[DHP+93] F. DiCesare, G. Harhalakis, J. M. Proth, M. Silva, and F.B. Vernadat. *Practice of Petri Nets in Manufacturing*. Chapman & Hall, London, 1993.

[DJHP97] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In *COMPOS*, pages 186–238, 1997.

[dMLH+00] M. de Miguel, T. Lambolais, M. Hannouz, S. Betge, and S. Piekarec. UML extensions for the specification of latency constraints in architectural models. In Woodside et al. [WGM00], pages 83–88. ISBN 1-58113-195-x.

[Don94] S. Donatelli. Superposed generalized stochastic Petri nets: Definition and efficient solution. In R. Valette, editor, *Application and Theory of Petri Nets 1994*, volume 815 of *Lecture Notes in Computer Science*, pages 258–277. Springer-Verlag, Berlin, 1994.

[DRSS01] R. Dumke, C. Rautenstrauch, A. Schmietendorf, and A. Scholz, editors. *Performance Engineering. State of the Art and Current Trends*, volume 2047 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2001.

[DW98] D. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[EW00] R. Eshuis and R. Wieringa. Requirements-level semantics for UML statecharts. In S.F. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV*, pages 121–140. Kluwer Academic Publishers, 2000.

[EW01] R. Eshuis and R. Wieringa. A real-time execution semantics for UML activity diagrams. In Heinrich Hußmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2029 of *Lecture Notes in Computer Science*. Springer, 2001.

[FN85] G. Florin and S. Natkin. Les réseaux de Petri stochastiques, 1985. Thesis de Doctorat d'Etat, Université Pierre et Marie Curie, Paris (in French).

[FW98] G. Franks and M. Woodside. Performance of multi-level client-server systems with parallel service operations. In Smith et al. [SWC98], pages 120–130. ISBN 1-58113-060-0.

[GCD73] R.M. Graham, G.J. Clancy, and D.B. DeVaney. A software design and evaluation system. *Communications of the ACM*, 16(2):110–116, 1973.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GIM+01] A. Goñi, A. Illarramendi, E. Mena, Y. Villate, and J. Rodríguez. ANTARCTICA: A multiagent system for internet data services in a wireless computing framework. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, Scottsdale, Arizona (USA), October 2001.

[GM00] H. Gomaa and D. Menascé. Design and performance modeling of component interconnection patterns for distributed software architectures. In Woodside et al. [WGM00], pages 117–126. ISBN 1-58113-195-x.

[Gru] Interoperable Database Group. `http://siul02.si.ehu.es/~jirgbdat`.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HCK97] C. Harrison, D. Chess, and A. Kershenbaum. Mobile agents: are they a good idea? In *Mobile Object Systems: Towards the Programmable Internet*, pages 46–48, 1997.

[HG96] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257. IEEE Computer Society Press, 1996.

[HHM95] H. Hermanns, U. Herzog, and V. Mertsiotakis. Stochastic process algebras as a tool for performance and dependability modelling. In *Proceedings of IEEE International Computer Performance and Dependability Symposium*, pages 102–113. IEEE CS-Press, April 1995.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of the statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[HR98] J. Hillston and M. Ribaudo. Stochastic process algebras: a new approach to performance modeling. In K. Bagchi and G. Zobrist, editors, *Modeling and Simulation of Advanced Computer Systems*. Gordon Breach, 1998.

[HRW95] C.E. Hrischuk, J.A. Rolia, and M. Woodside. Automatic generation of a software performance model using an object-oriented prototype. In Patrick W. Dowd and Gelenbe E., editors, *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, Durham, North Carolina, USA, January 1995. IEEE Computer Society.

[IBB02] P. Inverardi, S. Balsamo, and Selic B., editors. *Proceedings of the Third International Workshop on Software and Performance*, Rome, Italy, July 24-26 2002. ACM. ISBN 1-58113-563-7.

[Inc99a] CNET Inc., 1999. `http://www.download.com`.

[Inc99b] CNET Inc., 1999. `http://www.shareware.com`.

[Inc99c] CNET Inc., 1999. `http://www.gamecenter.com`.

[Inc99d] Tucows.Com Inc., 1999. `http://www.tucows.com`.

[Jai91] R Jain. *The Art of Computer Systems Performance Analysis*. Wiley, New York, April 1991.

[JCJO92] I. Jacobson, M. Christenson, P. Jhonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture Process and Organization for Business Success*. Addison-Wesley, 1997.

[JR91] K. Jensen and G. Rozenberg, editors. *High-Level Petri Nets: Theory and Application*. Springer-Verlag, Berlin, 1991.

[Kan92] K. Kant. *Introduction to Computer System Performance Evaluation*. Mc Graw-Hill, 1992.

[KP99] P. King and R. Pooley. Using UML to derive stochastic Petri nets models. In J. Bradley and N. Davies, editors, *Proceedings of the Fifteenth Annual UK Performance Engineering Workshop*, pages 45–56. Department of Computer Science, University of Bristol, July 1999.

[KRR98] E. Kovacs, K. Röhrle, and M. Reich. Mobile agents OnTheMove - integrating an agent system into the mobile middleware. In *Acts Mobile Summit*, Rhodos, Grece, June 1998.

[Lav83] S.S. Lavenberg. *Computer Performance Modeling Handbook*. Academic Press, New York, 1983.

[Lev97] F. Levy. *Verification of Temporal and Real-time Properties of Statecharts*. PhD thesis, University of Pisa-Genova-Udine, Pisa,Italy, February 1997.

[LGMC02a] J. P. López-Grao, J. Merseguer, and J. Campos. From UML activity diagrams to stochastic PNs: Application to software performance analysis. Technical report, Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, April 2002.

[LGMC02b] J.P. López-Grao, J. Merseguer, and J. Campos. On the use of formal models in software performance evaluation. In *Actas de las X Jornadas de Concurrencia*, pages 367–387, Jaca, Spain, June 2002. Universidad de Zaragoza.

[LGMC02c] J.P. López-Grao, J. Merseguer, and J. Campos. Performance engineering based on UML and SPNs: A software performance tool. In *Proceedings of the Seventeenth International Symposium On Computer and Information Sciences (ISCIS XVII)*, pages 405–409, Orlando (Florida), USA, October 2002. CRC Press.

[LMM99] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999*. Kluwer, 1999.

[LZSS84] E.D. Lazowska, J. Zahorjan, G. Scott, and K.C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *International Conference on Concurrency Theory*, pages 550–564, 1992.

[MBB+98] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF, the OMG mobile agent system interoperability facility. In *Proceedings of Mobile Agents '98*, September 1998.

[MBCD02] J. Merseguer, S. Bernardi, J. Campos, and S. Donatelli. A compositional semantics for UML state machines aimed at performance evaluation. In A. Giua and M. Silva, editors, *Proceedings of the 6th International Workshop on Discrete Event Systems*, pages 295–302, Zaragoza, Spain, October 2002. IEEE Computer Society Press.

[MCM00a] J. Merseguer, J. Campos, and E. Mena. A pattern-based approach to model software performance. In Woodside et al. [WGM00], pages 137–142. ISBN 1-58113-195-x.

[MCM00b] J. Merseguer, J. Campos, and E. Mena. Performance evaluation for the design of agent-based systems: A Petri net approach. In Mauro Pezzé and Sol M. Shatz, editors, *Proceedings of the Workshop on Software Engineering and Petri Nets, within the 21st International Conference on Application and Theory of Petri Nets*, pages 1–20, Aarhus, Denmark, June 2000. University of Aarhus.

[MCM01a] J. Merseguer, J. Campos, and E. Mena. Performance analysis of internet based software retrieval systems using Petri nets. In M. Meo, T. Dahlberg, and L. Donatiello, editors, *Proceedings of the 4th ACM International Workshop on Modeling, Analysis and Simulation of Wireless*

and Mobile Systems, within the 7th International Conference on Mobile Computing and Networking, pages 47–56, Rome, July 2001. ACM.

[MCM01b] J. Merseguer, J. Campos, and E. Mena. A performance engineering case study: Software retrieval system. In Dumke et al. [DRSS01], pages 317–332.

[MCM03] J. Merseguer, J. Campos, and E. Mena. Analysing internet software retrieval systems: Modeling and performance comparison. *Wireless Networks (WINET)*, 9(3):223–238, May 2003.

[Mes00] J. Meseguer. Rewriting logic and Maude: Concepts and applications. In Leo Bachmair, editor, *Rewriting Techniques and Applications, 11th International Conference, RTA 2000*, volume 1833 of *Lecture Notes in Computer Science*, pages 1–26, Norwich, UK, July 10-12 2000. Springer.

[MF76] P. M. Merlin and D. J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043, September 1976.

[MGD01] J.L. Medina, M. Gonzalez, and J.M. Drake. MAST real-time view: A graphic UML tool for modeling object-oriented real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 245–256, London, UK, December 2001. IEEE Computer Society Press.

[MIG00a] E. Mena, A. Illarramendi, and A. Goñi. Automatic ontology construction for a multiagent-based software gathering service. In *Proceedings of the Fourth International ICMAS'2000 Workshop on Cooperative Information Agents (CIA'2000), Springer series of Lecture Notes on Artificial Intelligence (LNAI), Boston (USA)*, July 2000.

[MIG00b] E. Mena, A. Illarramendi, and A. Goñi. Customizable software retrieval facility for mobile computers using agents. In *Proceedings of the 7th International Conference on Parallel and Distributed Systems (ICPADS'2000), Workshop International Flexible Networking and Cooperative Distributed Agents (FNCDA'2000)*, Iwate (Japan), July 2000. IEEE Computer Society.

[MIG00c] E. Mena, A. Illarramendi, and A. Goñi. A software retrieval service based on knowledge-driven agents. In *Cooperative Information Systems CoopIS'2000*, pages 174–185, Eliat, Israel, September 2000. Opher Etzion, Peter Scheuermann editors. Lecture Notes in Computer Science, (LNCS) Vol. 1901, Springer.

[MJP02] O. Munar, C. Juiz, and R. Puigjaner. Extending MASCOTS to a component-based software performance engineering methodology. In

Proceedings of the Seventeenth International Symposium On Computer and Information Sciences (ISCIS XVII), pages 410–414, Orlando (Florida), USA, October 2002. CRC Press.

[Mol82] M. K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, 31(9):913–917, September 1982.

[MR94] J. Myers and M. Rose. RFC 1725: Post Office Protocol - version 3, November 1994.

[MSPT96] A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of statecharts. In *International Conference on Concurrency Theory*, pages 687–702, 1996.

[Mur89] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Obj99] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, June 1999. Revision 2.3.

[Obj01] Object Management Group, http:/www.omg.org. *OMG Unified Modeling Language Specification*, September 2001. version 1.4.

[Obj02] Object Management Group, http:/www.omg.org. *UML Profile for Schedulabibity, Performance and Time Specification*, March 2002.

[OMG99] OMG. *XML standard metadata exchange format*. Object Management Group Inc., 1999. `http://www.omg.org`.

[PdS01] P. Pinheiro da Silva. A proposal for a LOTOS-based semantics for UML. Technical Report UMCS-01-06-1, Departament of Computer Science, University of Manchester, 2001.

[Pen96] Ian Pennock. *Configurator-User Guide, Issue 1.* N&S, AA&T, 1996.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[PJ80] M. Page-Jones. *The Practical Guide to Structured System Design*. Yourdon Press, 1980.

[PJ02] C. J. Pérez-Jiménez. *Técnicas de aproximación the throughput en redes de Petri estocásticas*. PhD thesis, University of Zaragoza, Spain, June 2002. In Spanish.

[PK99] R. Pooley and P. King. The unified modeling language and performance engineering. In *IEE Proceedings Software*. IEE, March 1999.

[PL99] I.P. Paltor and J. Lilius. Formalising UML state machines for model checking. In R.B. France and B. Rumpe, editors, *Proceedings of UML'99*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445, Fort Collins, CO, USA, October 28-30 1999. Springer.

[Poo99] R. Pooley. Using UML to derive stochastic process algebra models. In J. Bradley and N. Davies, editors, *Proceedings of the Fifteenth Annual UK Performance Engineering Workshop*, pages 23–34. Department of Computer Science, University of Bristol, July 1999.

[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Intl. Conf. TACS '91: Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 44–264. Springer, 1991.

[PS97] D. Petriu and G. Somadder. A pattern language for improving the capacity of layered client/server systems with multi-threaded servers. In *Proceedings of EuroPLoP*, Kloster Irsee, Germany, July 1997.

[PS98] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, 1998.

[PS02] 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, *Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002*, volume 2324 of *Lecture Notes in Computer Science*, pages 159–177, London, UK, April 14-17 2002. Springer.

[pUM] Precise UML Group. `http://www.puml.org/`.

[Ram74] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Petri Nets*. PhD thesis, MIT, Cambridge, MA, USA, February 1974.

[Rat01] Rational Software Corporation., 2001. `http://www.rational.com`.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, E. Frederick, and W. Lorensen. *Object Oriented Modeling and Design*. Prentice-Hall, 1991.

[Ree] T. Reenskaug. *Working with Objects: The OOram Software Engineering Method*.

[RH80] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering*, 6(5):440–449, September 1980.

[RS95] J. Rolia and K.C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):682–688, 1995.

[RWS+78] W.E. Riddle, J.C. Wileden, J.H. Sayler, A.R. Segal, and A.M. Stavely. Behavior modeling during software design. In *Proceedings of the 3rd International Conference on Software Engineering*, pages 13–22, Atlanta, Georgia, USA, May 1978. IEEE Computer Society.

[SB75] H.A. Sholl and T.L. Booth. Software performance modeling using computation structures. *IEEE Transactions on Software Engineering*, 1(4):414–420, 1975.

[SES01] SES/Workbench release 3.1. Scientific and Engineering Software Inc., 2001. `http://www.ses.com`.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[SG99] A. J. H. Simons and I. Graham. 30 things that go wrong in object modelling with UML 1.3. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 237–257. Kluwer, Dordrecht, 1999.

[SGW94] B. Selic, G. Guleckson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[Sif78] J. Sifakis. Use of Petri nets for performance evaluation. *Acta Cybernetica*, 4(2):185–202, 1978.

[Sil85] M. Silva. *Las Redes de Petri en la Automática y la Informática*. Editorial AC, Madrid, 1985. In Spanish.

[Sim00] A.J.H. Simons. On the compositional properties of UML statechart diagrams. In *Proceedings of the Rigorous Object-Oriented Methods,ROOM 2000*, January 2000.

[Smi81] C.U. Smith. Increasing information systems productivity by software performance engineering. In D.R. Deese, R.J. Bishop, J.M. Mohr, and H.P. Artis, editors, *Proceedings of the Seventh International Computer Measurement Group Conference*, pages 5–14, New Orleans, LA, USA, December 1-4 1981. Computer Measurement Group.

[Smi90] C. U. Smith. *Performance Engineering of Software Systems*. The Sei Series in Software Engineering. Addison–Wesley, 1990.

[Smi01] C.U. Smith. Origins of software performance engineering: Highlights and outstanding problems. In Dumke et al. [DRSS01], pages 96–118.

[SS00] J.A. Saldhana and S.M. Shatz. UML diagrams to object Petri net models: An approach for modeling and analysis. In *Twelfth International Conference on Software Engineering and Knowledge Engineering*, pages 103–110, Chicago, IL, USA, July 6-8 2000. Knowledge Systems Institute.

[SS01] A. Schmietendorf and A. Scholz. Aspects of performance engineering - An overview. In Dumke et al. [DRSS01], pages IX–XII.

[SSBD99] S. A. Seshia, R. K. Shyamasundar, A. K. Bhattacharjee, and S. D. Dhodapkar. A translation of statecharts to esterel. In *World Congress on Formal Methods (2)*, pages 983–1007, 1999.

[SW00] C.U. Smith and Ll.G. Williams. Software performance antipatterns. In Woodside et al. [WGM00], pages 127–136. ISBN 1-58113-195-x.

[SWC98] C.U. Smith, M. Woodside, and P. Clements, editors. *Proceedings of the First International Workshop on Software and Performance*, Santa Fe, New Mexico, USA, October 12-16 1998. ACM. ISBN 1-58113-060-0.

[UH97] P. Utton and B. Hill. Performance prediction: an industry perspective. In *Performance Tools 97 Conference*, St Malo, June 1997.

[US94] A.C. Uselton and S.A. Smolka. A compositional semantics for statecharts using labeled transition systems. In *Proceedings of the 5th International Conference on Concurrency Theory*, Lecture Notes in Computer Science, pages 2–17. Springer-Verlag, August 22-25 1994.

[VGGI98] Y. Villate, D. Gil, A. Goñi, and A. Illarramendi. Mobile agents for providing mobile computers with data services. In *Proceedings of the Ninth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 98)*, 1998.

[VGNP00] A.I. Verkamo, J. Gustafsson, L. Nenonen, and J. Paakki. Design patterns in performance prediction. In Woodside et al. [WGM00], pages 143–144. ISBN 1-58113-195-x.

[WGM00] M. Woodside, H. Gomaa, and D. Menascé, editors. *Proceedings of the Second International Workshop on Software and Performance*, Ottawa, Canada, September 17-20 2000. ACM. ISBN 1-58113-195-x.

[WHSB98] M. Woodside, C. Hrischuck, B. Selic, and S. Bayarov. A wideband approach to integrating performance prediction into a software design environment. In Smith et al. [SWC98], pages 31–41. ISBN 1-58113-060-0.

[WHSB01] M. Woodside, C. Hrischuck, B. Selic, and S. Bayarov. Automated performance modeling of software generated by a design environment. *Performance Evaluation*, 45:107–123, July 2001.

[WLAS97] G. Waters, P. Linington, D. Akehurst, and A. Symes. Communications software performance prediction. In *13th UK Workshop on Performance Engineering of Computers and Telecommunication Systems*, pages 38/1–38/9, Ilkley, July 1997. Demetres Kouvatsos Ed.

[Woo00] M. Woodside. Software performance evaluation by models. In G. Haring, C. Lindemann, and M. Reiser, editors, *Performance Evaluation*, volume 1769 of *Lecture Notes in Computer Science*, pages 283–304. Springer, 2000.

[You89] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989.

[ZFGH00] A. Zimmermann, J. Freiheit, R. German, and G. Hommel. Petri net modelling and performability evaluation with TimeNET 3.0. In *Proceedings of the 11th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 188–202. Lecture Notes in Computer Science, Vol. 1786, Springer, 2000.

Index