Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 101 (2004) 51–72
www.elsevier.com/locate/entcs

Toward Extracting π-calculus from UML Sequence and State Diagrams[1]

Katerina Pokozy-Korenblat[2] and Corrado Priami[3]

Department of Informatics and Telecommunications
University of Trento
Povo, Italy

Abstract

We propose an automatic translation of UML specifications made up of sequence and state diagrams into π-calculus processes. The central point of the proposed translation is the coherence of the two types of diagrams. We show the feasibility of the approach on case studies.

Keywords: process algebras, π-calculus, UML, sequence diagram, state diagram, specification.

1 Introduction

The Unified Modelling Language (UML) [3] is a standard notation used to capture high-level design of software systems. It gives structured, semi-formal, graphical methods for specification which are however not strong enough for verification and validation of systems. UML provides the user with different kinds of diagrams, each of them is natural for describing different aspects of complex (software) systems. In this paper we restrict our attention on specifications including only sequence and state diagrams. Such a choice is often sufficient for specifying the communication level of systems and can be considered as a first step in handling multi-diagram UML specifications.

[1] This work is partially supported by the DEGAS (Design Environment for Global Applications) project IST-2001-32072 funded by the FET Proactive Initiative on Global Computing
[2] Email: pokozy@science.unitn.it
[3] Email: priami@dit.unitn.it

1571-0661
doi:10.1016/j.entcs.2004.09.007

To implement a formal analysis of a UML specification, we propose to translate it to the formal notation of process algebras [13]. Process algebras are foundational calculi used to describe the concurrent and distributed structure of systems. They are made up of a few operators such as: i) $a.-$ that describes sequential composition of actions, ii) $-|-$ that is the parallel composition of processes, iii) $-+-$ that denotes a nondeterministic choice. We view here process algebras as an intermediate language into which UML specifications can be translated for the following analysis. Note that process algebras is a natural formalism for representing communications that is very important in a translation of sequence diagrams and also for composing independent parts of the model that is essential for translation of UML specifications contains several entities. Furthermore, process algebras gives wide possibilities for performance analysis that is critical for communication intensive part of a model that is usually specified by sequence diagrams.

A number of synthesis techniques for building models from interaction diagrams has been developed. In [15] a way of representation of collaboration diagrams in terms of Colored Petri Nets was proposed. Considering a specification consisting of state and interaction diagrams, a state diagram based approach to translation was presented in [5] and [8]. In those works states are represented as processes and transitions are represented as actions along communication channels. However, when we focus on a communication level such an approach is not effective enough. It would be more natural to use messages as a basic element of translation since that better corresponds to the nature of process algebras. In this paper we focus on a sequence diagram based approach, where objects are considered as π-calculus processes and messages as communications between these processes. A state diagram of an object is used for choosing the feasible sequences of the messages occurring in the sequence diagram.

Note that an outcome of our proposal is also the definition of a formal semantics for UML sequence diagrams based on the structural operational semantics of the π-calculus. Many different approaches to define a formal semantics of UML, and in particular, of sequence diagrams and their counterpart from the telecommunication industry message sequence charts have been proposed. One of the approaches generates statechart model (e.g. [19], [4]). In [18] a semantics in terms of labelled transition systems and parallel composition is presented. A process algebras approach presented in [12] permits to compose several message sequence charts. Alur et al. [1] gives the semantics of message sequence charts in terms of partial order of events occurring in the system. A way of formalizing of collaboration diagrams by means of graph transformation rules and graph processes is presented in [10]. An integrated

graph semantics given in [9] allows to specify interaction diagrams in a universal ways together with other types of diagrams. A trace-based semantics was proposed in [2]. In [17] a translation of sequence diagrams into process algebras where messages are represented as actions, is considered. The work focuses on a translation of a set of simple sequence diagrams which correspond to separate scenarios. In contrast we represent a message as a communication between processes and focus on translation of constructions (e.g. branching, assignment) allowing to express complicated behaviour in a single diagram. Specifics and strong side of our process algebra based semantics is that in obtained formal model messages between objects are presented in a natural for communications way that is essential, for example, for security analysis. Note however that defining of formal semantics is not our main concern.

We now briefly discuss the motivations for the present work. We rely on a standard Unified Modelling Language (UML) to use formal methods in the software production process. The challenges we approach in this task are the definition of techniques to extract specifications into process calculi from the possibly excessive or incomplete information in the UML description. The final goal is to have a design environment in which the user only interacts with UML in order to perform formal analysis of his/her applications.

The paper is structured as follows. In section 2 there is a description of those aspects of UML we are interested in. In section 3 an overview of the π-calculus is presented. A translation from sequence diagrams to π-calculus is given in section 4. In section 5 we discuss the joint translation of sequence and state diagrams. Finally, some remarks on the ways of using of the obtained π-calculus specification are given.

2 Brief UML description

UML is a semi-formal modelling language which is a standard for high-level specification of software systems. There are many different types of UML diagrams which are used to specify different aspects of software systems. In this short presentation we focus on sequence and state diagrams.

A **sequence diagram** shows how objects interact with one another by representing examples of executions. A sequence diagram has two dimensions: the vertical dimension represents time and the horizontal one represents different objects. Objects can communicate by exchanging messages represented by arrows. To show different kinds of communications the following variations of notation are considered in this paper.

- *Stick arrowhead* is used for synchronous communication. In the case of

$$Act : \mu.P \xrightarrow{\mu} P \qquad Ide : \frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\mu} P'}{Q(\tilde{y}) \xrightarrow{\mu} P'}, Q(\tilde{x}) = P$$

$$Par : \frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q}, bn(\mu) \cap fn(Q) = \emptyset \qquad Sum : \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'}$$

$$Res : \frac{P \xrightarrow{\mu} P'}{(\nu x)P \xrightarrow{\mu} (\nu x)P'}, x \notin n(\mu) \qquad Open : \frac{P \xrightarrow{\overline{x}\langle y\rangle} P'}{(\nu y)P \xrightarrow{\overline{x}(y)} P'}, y \neq x$$

$$Close : \frac{P \xrightarrow{\overline{x}(y)} P', Q \xrightarrow{x(w)} Q'}{P|Q \xrightarrow{\tau} (\nu y)(P'|Q'\{y/w\})}, \ y \notin fn(Q) \qquad Com : \frac{P \xrightarrow{\overline{x}\langle y\rangle} P', Q \xrightarrow{x(w)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{y/w\}}$$

Table 1
Late transition system for the π-calculus.

nested control flow the entire nested sequence have to be completed before the outer level sequence resume.

- *Dashed arrow with stick arrowhead* is used for returning message.

A message is labelled at least with the message name; one can also include arguments and a condition which acts as a guard for sending the message. Furthermore a message can be associated with an assignment that associates with the assigned variable the value returned after the message.

Messages can be combined in a branching construction which is shown by multiple arrows leaving a single point and means alternative or concurrency of those messages depending of their conditions.

A **state diagram** describes the sequences of states and transitions through which the modelled element can proceed during its lifetime as a reaction to discrete events. A state diagram is a graph that represents a state machine.

The semantics of a state diagram can be defined in terms of a Kripke structure [11]. Given a state s in a state diagram S, we denote as $(s^\bullet)^\bullet$ a set of pairs of transitions t_1t_2 such that there exists a sequence $s_1[t_1\rangle s_2[t_2\rangle$ of states (s_i) and transitions (t_i) in the Kripke structure of S.

3 The π-Calculus

In this section we briefly recall the π-calculus [13], a model of concurrent communicating processes providing the notion of *naming*.

Let $\mathcal{N}$ be a countable infinite set of *names* ranged over by $a, b, \ldots$ with $\mathcal{N} \cap \{\tau\} = \emptyset$. We also assume a set $\mathcal{A}$ of *agent identifiers* ranged over by $A, A_1, \ldots$. *Processes* (denoted by $P, Q, R, \ldots \in \mathcal{P}$) are built from names according to the

syntax

$$P ::= \mathbf{0} \mid \pi.P \mid P + P \mid P|P \mid (\nu x)P \mid [x = y]P \mid A(y_1, \ldots, y_n)$$

where π may be $x(y)$ for *input*, $\overline{x}\langle y\rangle$ for *output* (where x is the *subject* and y the *object*), ε for *empty string*, or τ for *silent* moves. Hereafter, the trailing $\mathbf{0}$ will be omitted.

The prefix π is the first atomic action that the process $\pi.P$ can perform. The input prefix binds the name y in the prefixed process. Intuitively, some name y is received along the link named x. The output prefix does not bind the name y which is sent along x. The silent prefix τ denotes an action which is invisible to an external observer of the system. Summation denotes nondeterministic choice. The operator | describes parallel composition of processes. The operator (νx) acts as a static binder for the name x in the process P that it prefixes. In other words, x is a unique name in P which is different from all the external names. Finally, matching $[x = y]P$ is an `if-then` operator: process P is activated if $x = y$. $A(y_1, \ldots, y_n)$ is the definition of constants (hereafter, $\tilde{y}$ denotes $y_1, \ldots, y_n$). Each agent identifier A has a unique defining equation of the form $A(y_1, \ldots, y_n) = P$, where the y_i are distinct and $fn(P) \subseteq \{y_1, \ldots, y_n\}$ (see below for the definition of free names fn).

A parallel composition of processes $P_1, \ldots, P_n$ is written as $\prod_{i=1\ldots n} P_i$. For a set of names $V = \{v_1, \ldots v_n\}$ we use the notation $(\nu V)P$ for $(\nu v_1) \ldots (\nu v_n)P$ and $(\nu v_1, v_2)P$ for $(\nu v_1)(\nu v_2)P$.

We use here a late version of the π-calculus, although early semantics could apply as well. The late operational semantics for the π-calculus is defined in the SOS style, and the labels of the transitions are τ for silent actions, $x(y)$ for input, $\overline{x}y$ for free output, and $\overline{x}(y)$ for bound output. We will use μ as a metavariable for the labels of transitions (it is distinct from π, the metavariable for prefixes, though it coincides in two cases). We recall the notion of free names $fn(\mu)$, bound names $bn(\mu)$, and names $n(\mu) = fn(\mu) \cup bn(\mu)$ of a label μ.

μ	*Kind*	$fn(\mu)$	$bn(\mu)$
τ	Silent	$\emptyset$	$\emptyset$
$\overline{x}y$	Free Output	$\{x, y\}$	$\emptyset$
$x(y), \overline{x}(y)$	Input and Bound Output	$\{x\}$	$\{y\}$

Functions fn, bn and n are extended to processes in the obvious way. Below we assume that the *structural congruence* $\equiv$ on processes is defined as the least congruence satisfying the following clauses:

- P and Q α-equivalent (they only differ in the choice of bound names) implies $P \equiv Q$,

- $(\mathcal{P}/_{\equiv}, +, \mathbf{0})$ and $(\mathcal{P}/_{\equiv}, |, \mathbf{0})$ are commutative monoids,
- $\varepsilon.P \equiv P$,
- $[x = x]P \equiv P$,
- $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P, (\nu x)(R \mid S) \equiv (\nu x)R \mid S$ if $x \notin fn(S)$, $(\nu x)(R \mid S) \equiv R \mid (\nu x)S$ if $x \notin fn(R)$, and $(\nu x)P \equiv P$ if $x \notin fn(P)$.

A *variant* of $P \xrightarrow{\mu} Q$ is a transition which only differs in that P and Q have been replaced by structurally congruent processes, and μ has been α-converted, where a name bound in μ includes Q in its scope.

We report the late transition system for the π-calculus in Tab. 2. The transition in the conclusion of each rule, as well as in the axiom, stands for all its variants.

4 Translation of Sequence Diagrams

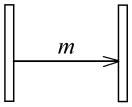
In this section we restrict our attention to UML sequence diagrams. Note that the semantics of UML allows a message in a sequence diagram to be skipped. For simplicity in this section we consider the case of non-skipping messages as it is the common practice of designers (we will deal with skipping of messages later in the paper). Moreover we assume that names of messages are unique, otherwise we rename them before translation.

First we consider sequence diagrams without conditions on messages. We will represent an object from a sequence diagram as a process in the π-calculus and compose all processes arising from the given sequence diagram via parallel composition.

A message between two objects is represented as a communication between the corresponding processes. For each message we create a private channel in the π-calculus representation and translate the message as a synchronization on this channel. In our translation a self-call is represented by the silent action τ. Note that for an assignment construction associated to a self-call it can be essential to express the fact that we assign the variable (e.g. for analyzing of security properties). In such a case self-call can be translated as a communication with an additional internal object.

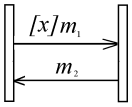
As far as sequence diagrams show how an object interacts with others, it is natural to consider an object as a sequence of sending and receiving of messages. To translate an object we produce sequentially for each of its sent/received message an input/output of a signal along the corresponding channel.

Given two objects connected by a message m, we translate this message as



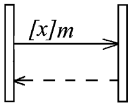
$$\Longrightarrow \quad (\nu m)(\overline{m} \mid m)$$

Fig. 1. Translation of a message.



$$\Longrightarrow \quad (\nu m_1,\ m_2)(([x = true]\ \overline{m}_1.\ m_2 \mid m_1.\ \overline{m}_2) + ([x = false]\ m_2 \mid \overline{m}_2))$$

Fig. 2. Translation of a condition.



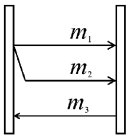
$$\Longrightarrow \quad (\nu m,\ return^m)([x = true]\overline{m}.\ return^m \mid m.\ \overline{return}^m)$$

Fig. 3. Translation of a return.

an input on the channel m in the object receiving the message and an output on the same channel in the object sending the message (see Fig. 1).

Consider now a message with a condition. The message is sent if its condition is satisfied, or it is skipped otherwise. Given a sending message $[x]m$, we obtain an output on the channel m prefixed by the matching $[x = true]$ if the condition is satisfied, or a skipping of the message prefixed by the matching $[x = false]$ otherwise. For a receiving message $[x]m$ we obtain an input on m or a skipping of it, depending on the value of x. Nested messages initialized by a message with a condition including its return will be discarded if the condition is not satisfied. To illustrate the translation of conditions, consider a simple sequence diagram with a single condition x from Fig. 2. It is translated to a summation of two subprocesses representing two possible valuations of x.

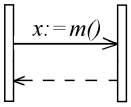
We translate an explicit return of the message as a usual message with the name $return^{\langle message\ name \rangle}$. For example, we translate a message $[x]m$ with an explicit return (Fig. 3) as a sequence of messages $\langle m, return^m \rangle$.



$$\Longrightarrow \quad (\nu m_1,\ m_2,\ m_3,\ syn)\ (\overline{m}_1.\ syn \mid \overline{m}_2.\ syn \mid \overline{syn}.\ \overline{syn}.\ m_3 \mid m_1 \mid m_2 \mid \overline{m}_3)$$

Fig. 4. Translation of a branching construction.

A branching of several messages is translated as a parallel composition of these messages synchronized before continuation because all of branched



$$\Longrightarrow \quad (\nu m,\ return^m)(\overline{m}.\ return^m(x) \mid m.\ \overline{return}^m\langle x\rangle$$

Fig. 5. Translation of an assignment.

messages have to be delivered. In more detail, for a sending object we introduce an input on the special channel *syn* after any branched messages. Then we construct a continuation process that is the translation of the remaining messages prefixed by sending of a *syn* signal for any message in the branching construction. Finally, we compose the translation of the branching structure and the continuation process by parallel composition. In the receiving process we have a parallel composition of branched messages and continuation without synchronization. We use the *syn* channel to force the continuation process starting after the delivery of all the branching messages. We illustrate our translation on a simple sequence diagram presenting two parallel messages followed by a reply (see Fig. 4). We use a channel *syn* to force m_3 occurring after m_1 and m_2.

An assignment construction is generally used for binding an identifier that stores the return value of a message. It can be translated as a message with an explicit return which transfers not a signal but a required variable (see Fig. 5). In other words, we use a real communication rather than a simple synchronization.

We now formally define a translation function from UML sequence diagrams to the π-calculus. Given a message m, we define a set of nested messages $nest(m)$ in case of an explicit return as the set of messages that become enabled by the sending of m before the return of m (including the return), or as an empty set, otherwise.

By abuse of notation we use the metavariable for processes $\mathcal{P}$ to denote prefixes possibly prefixed by a matching.

Fix a sequence diagram *SD* with a set of objects $\mathcal{O}$, a set of messages $\mathcal{M}es$ and a set of conditions $\mathcal{C}$. Let $\rho : \mathcal{C} \cup \{\Lambda\} \rightarrow Bool$ be an evaluation function that returns *true* for the special condition Λ. Given a message m, we define c^m as a condition corresponding to m, or as Λ if there is no condition on m. Given an object O and a message m in O, we define the function $tr_\rho : \mathcal{O} \times \mathcal{M}es \rightarrow \mathcal{P}$

as

$$tr_\rho(O,m) = \begin{cases} [c^m = true]\,\overline{m}, & \text{if } O \text{ sends } m \text{ and } \rho(c^m) = true; \\ \lceil c^m = false\rceil\varepsilon, & \text{if } O \text{ sends } m \text{ and } \rho(c^m) = false; \\ \overline{m}, & \text{if } O \text{ sends } m \text{ and } c^m = \Lambda; \\ m, & \text{if } O \text{ receives } m \text{ and } \rho(c^m) = true; \\ \varepsilon, & \text{if } (O \text{ receives } m \text{ and } \rho(c^m) = false) \text{ or} \\ & (\exists m' \mid m \in nest(m') \text{ and } \rho(c^{m'}) = false). \end{cases}$$

For a given assignment construction $x := m()$ we translate the return of m in a special way. We redefine the function $tr_\rho(O, return^m)$ in case of assignment as $return^m(x)$ for a sending object, or as $\overline{return}^m\langle x\rangle$ for a receiving object.

Fix an evaluation ρ and an object O defined by a sequence of set of sent/ received messages $M^O = \langle M_0, M_1, \ldots, M_n\rangle$, where a set $M_i = \{m_i^1, \ldots, m_i^{k_i}\}$ represents branching messages and $m_i^j \in \mathcal{M}es$ for each $i \in \{1 \ldots n\}$ and $j \in \{1 \ldots k_i\}$. We define the translation function $seq_\rho : \mathcal{O} \times \mathcal{M}es^* \times \mathcal{P} \to \mathcal{P}$ as follows:

$seq_\rho(O, \langle\,\rangle, tail) = tail$, and

$seq_\rho(O, \langle M_i, \ldots, M_n\rangle, tail) =$

$$= \begin{cases} tr_\rho(O, m_i^1).seq_\rho(O, \langle M_{i+1}, \ldots, M_n\rangle, tail), & \text{if } k_i = 1 \\ \prod_{j=1\ldots k_i} tr_\rho(O, m_i^j).seq_\rho(O, \langle M_{i+1}, \ldots, M_n\rangle\,|_{nest(m_i^j)}, syn^{M_i}.tail) \mid \\ \mid \underbrace{\overline{syn}^{M_i}. \ldots .\overline{syn}^{M_i}}_{k_i}.seq_\rho(O, rest(\langle M_i, \ldots, M_n\rangle), tail), & \text{if } k_i > 1 \end{cases}$$

where $rest(\langle M_i, \ldots, M_n\rangle)$ is a subsequence of $\langle M_i, \ldots, M_n\rangle$ starting after returns of all messages from M_i.

Eventually, for a fixed evaluation ρ we obtain $P^\rho = \prod_{O\in S} seq_\rho(O, M^O)$. And the overall translation of SD is $P = (\nu V)(\sum_{\rho\in\mathcal{C}\cup\{\Lambda\}\times Bool} P^\rho)$, where $V = \{v \mid v \in M^O \vee v = syn^{M_i}\}$.

As final remarks, we give some notes for simplifying the result of the translation.

- *Repeating conditions.* We leave in a process only the first instance of a repeating condition if these instances do not divided by an assignment that can change the condition value.
- *Empty branched subprocesses.* In a translation of branching construction because of the false value of a condition we can obtain a parallel subprocess containing only conditions and receiving of a synchronizing message. Such

processes are nonessential for the translation and can be skipped together with a corresponding sending of the synchronizing message. In the case of a single active branched message we can translate it as usual sequence message. An illustration of this point you can see the end of the next subsection.

4.1 An Example

To illustrate our translation consider the sequence diagram in Fig. 6 representing a slight variant of the Phone system in [8]. In this example we have two conditions $c_1 = [busy]$ and $c_2 = [not\ busy]$. Thus we obtain two possible valuations $\rho_1 : \rho_1(c_1) = true$, $\rho_1(c_2) = false$ and $\rho_2 : \rho_2(c_1) = false$, $\rho_2(c_2) = true$. Obviously only these two valuation are available. The result of the translation is shown below.

$Caller^{\rho_1} = \overline{lift}.dial\_tone.\overline{number}.connect\_tone.busy\_tone.disconnect.\overline{hangs\_up}$
$Caller^{\rho_2} = \overline{lift}.dial\_tone.\overline{number}.connect\_tone.ring\_tone.\overline{talking}_1.talking_2.$
$disconnect.\overline{hangs\_up}$

$Phone^{\rho_1} = (\nu syn_1)lift.\overline{dial\_tone}.number.(\overline{connect\_tone}.syn_1 \mid \overline{connect}.$
$return^{connect}(busy).syn_1 \mid \overline{syn}_1.\overline{syn}_1.[busy = true]\overline{busy\_tone}.$
$\overline{disconnect}.hangs\_up)$
$Phone^{\rho_2} = (\nu syn_1, syn_2)lift.\overline{dial\_tone}.number.(\overline{connect\_tone}.syn_1 \mid \overline{connect}.$
$return^{connect}(busy).syn_1 \mid \overline{syn}_1.\overline{syn}_1.Calling)$
$Calling = [busy = false]\overline{ring\_tone}.syn_2 \mid [busy = false]\overline{call}.answer.return^{call}.$
$syn_2 \mid \overline{syn}_2.\overline{syn}_2.\overline{disconnect}.hangs\_up$

$Receiver^{\rho_1} = connect.\overline{return}^{connect}\langle busy\rangle.\varepsilon$
$Receiver^{\rho_2} = connect.\overline{return}^{connect}\langle busy\rangle.call.\overline{answer}.talking_1.\overline{talking}_2.$
$\overline{return}^{call}$

$System = (\nu V)(Phone^{\rho_1} \mid Receiver^{\rho_1} \mid Caller^{\rho_1}) + (Phone^{\rho_2} \mid Receiver^{\rho_2} \mid$
$\mid Caller^{\rho_2})$ where V is the set of channels of $System$.

Now we illustrate the simplification technique of empty branched subprocesses described in the previous section. In $Phone^{\rho_1}$ we translate the branching construction $\{busy\_tone,\ ring\_tone,\ call\}$ as a single message $busy\_tone$ because conditions on two other messages are false on a valuation ρ_1, and we obtain its translation $[busy = true]\overline{busy\_tone}$ instead of $[busy = true]syn_2 \mid [busy = true]syn_2 \mid [busy = true]\overline{busy\_tone}.syn_2 \mid \overline{syn}_2.\overline{syn}_2.\overline{syn}_2$ as it would be done in the general case.

5 Joint Translation of Sequence and State Diagrams

Each type of UML diagrams has its own most natural way of translation to the π-calculus. To translate a whole system composed of different diagrams

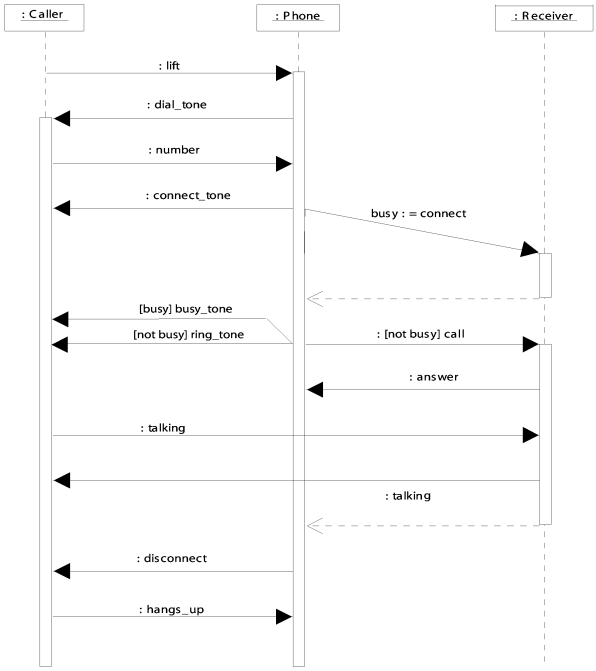


Fig. 6. Sequence diagram of the *Phone system*.

we choose a driving type of diagrams (here a sequence diagram), and we take necessary additional information about the system from other types of diagrams (here state diagrams). For a joint translation of sequence and state diagrams we have two different approaches which are distinguished in the choice of the driving type of diagrams:

- *Sequence diagram based translation.* According to our translation, each valuation is considered and translated separately so that the obtained π-calculus representation is a summation of subprocesses for all possible valuations. Additional information from state diagrams helps to determine feasible computations. The problem here is to represent more detailed information from state diagrams because it is not understandable how to translate internal actions of a state diagram not represented in the sequence diagram. It is possible to ignore this detailisation and obtain a translation at a communication level.
- *State diagram based translation.* A state diagram (for each object) is translated as a process parameterized by state name and representing a transition from state to state. Additional information from a sequence diagram is necessary for composing the processes corresponding to different state diagrams. Such approach was used in [5] for producing translation to Per-

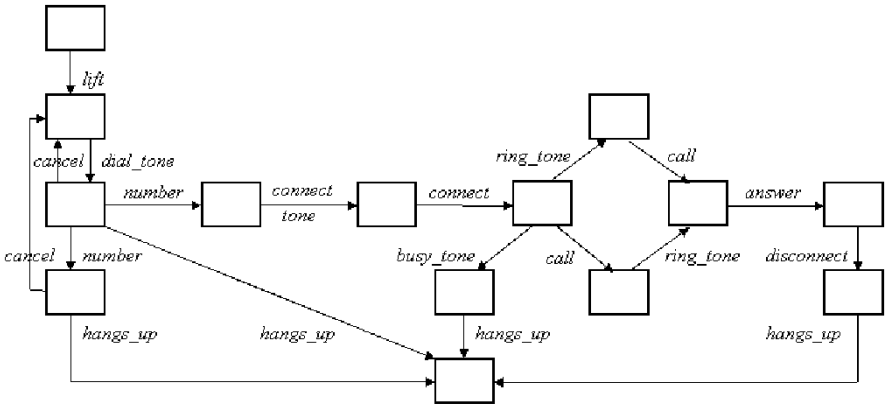


Fig. 7. State diagram of the object *Phone* of the *Phone system*.

formance Evaluation Process Algebra (PEPA).

The last approach requires a very detailed description of each object of a system. However, sometimes we would prefer to have a high-level specification which does not contain such a detailed specification of all objects. Thats why we choose a sequence diagram based approach in our translation.

5.1 Sequence diagram based translation

In the previous section we considered the case of nonskipped messages. However, in a general case messages in a sequence diagram can be skipped. It can be useful, for example, to present alternative use cases in one sequence diagram. If we consider a separate sequence diagram in this paradigm we obtain all possible subsequences of a given sequence of messages. So, we can see only an order of messages but have no information which message has to be skipped in an allowed behaviour.

To illustrate such a situation we add to the Phone system on Fig. 6 a message *timeout_tone* from *Caller* to *Phone* between messages *dial_tone* and *number*. It can be interpreted as follows. If a *Caller* does not dial a number, after some waiting period it receives a *timeout_tone* and finishes the call. After the *timeout_tone*, the *Caller* cannot dial a number, but a sequence diagram does not give us such information, and we have a sequence of messages $\langle \ldots dial\_tone,\ timeout\_tone,\ number \ldots \rangle$ besides the correct sequences $\langle \ldots dial\_tone,\ timeout\_tone,\ hangs\_up \rangle$ and $\langle \ldots dial\_tone,\ number \ldots \rangle$.

To consider only correct sequences of messages we need some additional information which can be taken from other types of diagrams. Namely we need an information about branching points in a behaviour model of the system. By default we consider messages in a sequence diagram sequentially and suppose that for each two sequential messages there is an execution way con-

cluding those two messages. In a point when we skip a message we have two alternatives: to execute messages of the sequence diagram one after another, or to jump to some other message of the diagram. We will call such a situation by a branching point.

In our translation information about branching points will be obtained from state diagrams which describe behaviour of single objects. We restrict a state diagram on a set of transitions representing communications and call it a *restricted* state diagram. We can talk about branching point for a restricted state diagram if for a given state we have outgoing transitions corresponding to more then one set of branching messages. A *skeleton* of a state diagram is a restriction of a restricted state diagram on a set of transitions which are belong to some branching point.

For sequence and state diagrams in isolation we can present semantics in terms of π-calculus or some other formalism. However, correlation between different diagrams is not specified formally in UML. So we have to make some assumptions on a relation between these diagrams to be correct in the translation of the whole specification. These assumptions are concern to an agreement on names presenting the same communications in different diagrams.

Given UML specification consisting of a sequence diagram SD and state diagrams $OD_1, \ldots OD_k$ corresponding to objects $O_1, \ldots O_k$ of SD. We will say that a specification is *coherent* if there is the following relation between messages in sequence diagram and transitions in state diagrams:

(i) Names of messages in SD are unique (it is necessary for correct correspondence between messages in the sequence diagram and transitions in the state diagrams).

(ii) A message m in SD relating objects O_i and O_j and a transition m in OD_i (OD_j) represent the same event in the specified system.

(iii) A message m in SD can be skipped iff there is a path in the corresponding state diagram in which a transition m is skipped.

(iv) A message m' follows m iff there exists a pair of sequential transitions $m,\ m'$ in a skeleton of the corresponding state diagram.

The last point means that we don't require all messages to be presented in a state diagram, and a message in a sequential part of the diagram can be skipped. Note that a state diagram can include more detailed information than a corresponding sequence diagram. But in this translation we use state diagrams only for additional information about available sequences of messages. So internal actions of a state diagram are nonessential to our translation.

For readability we present here a translation of a simplified version of sequence diagrams which do not contain a branching construction. Fix an

object O with a sequence of sent/received messages $M^O = \langle m_1, \ldots, m_n \rangle$. The first step of the translation is to calculate branching points. We define a set of *starting states of a message* m_{i+1} recursively as a set of states obtained from starting messages of m_i after execution of a transition corresponding to m_i. A *branching point* $B(m_i)$ contains a pair (m_i, m_{i+1}) and further for each starting state s of m_i is constructed in the following way: $\forall tt' \in (s^\bullet)^\bullet$

$$B(m_i) = \begin{cases} B(m_i) \cup (m(t), m(t')), & \text{if } m(t) = m_i \text{ and } \exists t'' \neq t'.\ tt'' \in (s^\bullet)^\bullet; \\ B(m_i) \cup (skip\_m_i, m(t)), & \text{if } m(t) \neq m_i. \end{cases}$$

Then we modify the translation of sequence diagrams proposed in section 4 by using information about branching points to single out correct execution sequences. The difference from the original algorithm is the definition of the function seq_ρ. Given an evaluation ρ, we calculate the translation function $seq_\rho : \mathcal{O} \times \mathcal{M}es^* \to \mathcal{P}$ as follows: $seq_\rho(O, \langle\,\rangle) = \mathbf{0}$, and

$$seq_\rho(O, \langle m_i, \ldots, m_n \rangle) = \sum_{\{j \ \mid (m_i, m_j) \in B(m_i)\}} tr_\rho(O, m_i).seq_\rho(O, \langle m_j, \ldots, m_n \rangle) + \sum_{\{j \ \mid (skip\_m_i, m_j) \in B(m_i)\}} seq_\rho(O, \langle m_j, \ldots, m_n \rangle).$$

Let us illustrate this approach with the Phone System which was already considered in section 4. Now we represent it by the sequence diagram (Fig. 6) and the state diagram for the object *Phone* (Fig. 7). An assumption about skipping of messages intuitively means that the *Caller* can hang up after any of his actions. The result of the translation of an object *Phone* on the valuation ρ_1 is:

$$\begin{aligned} Phone^{\rho_1} = {} & lift.\overline{dial\_tone}.(hangs\_up + number.hangs\_up + number. \\ & (\overline{connect\_tone}.syn_1 \mid \overline{connect}.return^{connect}(busy).syn_1 \mid \overline{syn}_1. \\ & \overline{syn}_1.[busy = true]\overline{busy\_tone}.hangs\_up)) \end{aligned}$$

A separate problem is a translation of a part-described system. If one of the objects in a sequence diagram related by the message is not described by a state diagram, we can extract information from an existing state diagram. To illustrate this moment let us consider the translation of the object *Caller* from the *Phone System* based on the sequence diagram (Fig. 6) and the state diagram for the object *Phone* (Fig. 7):

$$\begin{aligned} Caller^{\rho_1} = {} & \overline{lift}.dial\_tone.(\overline{hangs\_up} + \overline{number}.\overline{hangs\_up} + \overline{number}. \\ & connect\_tone.busy\_tone.\overline{hangs\_up}) \end{aligned}$$

Here we suppose that for the receiving of a message *number* there are three cases: "ignore the message and finish", "finish after the message" and "continue after the message", then we have the same cases for the sending of this message.

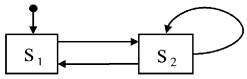


Fig. 8. An abstract level state diagram.

5.2 Compositional translation

To extract the case of one sequence diagram and a set of state diagrams for some of its objects, presented in section 4 we need a possibility to compose several sequence diagrams. We will do it by adding an abstract level state diagram which shows relations between different sequence diagrams.

This information can be obtained also from state diagrams of objects. We can construct an abstract level state diagram from the set of state diagrams of objects as follows. If in a state diagram of an object there are two sequential transitions corresponding to messages from different sequence diagrams, we produce a transition between states presenting those sequence diagrams in an abstract level state diagram.

In this section we propose a translation of a coherent specification consisting of an abstract level state diagram, a set of sequence diagrams corresponding to some of its states and a set of state diagrams corresponding to objects of sequence diagrams. The natural way of translation of an abstract state diagram is to correspond to each state an agent starting with an action which describes the state and prolonging as an agent for one of the states succeed to the considered one. If there is a sequence diagram corresponding to a given state, an action describing the state is a pair of control signals for initializing and finishing of the agent corresponding to the sequence diagram. Otherwise we suppose a silent action. A sequence diagram is translated as it was shown in section 4 except of the fact that the result of this translation has to be reexecutable because it can be used as a part of abstract state diagram translation. So we have to present a sequence diagram as a recursive agent that initializes a separate agent for each of its objects.

Fix an abstract level state diagram AD with a set of states S, an initial state s^{init} and a set of sequence diagrams $SD(s)$ for some of states s from S. For each state s of AD we construct an agent $s()$ as follows:

$$s() = \begin{cases} \sum_{\{s' | s[\rangle s'\}} \overline{ini\_s}.\, end\_s.\, s'(), & \text{if } \exists SD(s) \text{ and } \exists\{s' \mid s[\rangle s'\}; \\ \overline{ini\_s}.\, end\_s, & \text{if } \exists SD(s) \text{ and } \neg\exists\{s' \mid s[\rangle s'\}; \\ \sum_{\{s' | s[\rangle s'\}} \tau.\, s'(), & \text{if } \neg\exists SD(s) \text{ and } \exists\{s' \mid s[\rangle s'\}; \\ \tau, & \text{if } \neg\exists SD(s) \text{ and } \neg\exists\{s' \mid s[\rangle s'\}. \end{cases}$$

Fix a sequence diagram $SD(s)$ with a set of objects $O_1, \ldots O_k$ and an object

O_j that receives the last message in $SD(s)$. Given an evaluation ρ of conditions of $SD(s)$, we translate a sequence diagram as follows:

$SD^s() = (\nu V) \sum_{\rho \in \mathcal{C} \cup \{\Lambda\} \times Bool} SD^s_\rho();$

$SD^s_\rho() = (\nu V')ini\_s.\overline{ini\_O^\rho_1}. \ldots \overline{ini\_O^\rho_k}.end\_O^\rho_j.\overline{end\_s}.SD^s_\rho();$

$O_i() = \prod_{\rho \in \mathcal{C} \cup \{\Lambda\} \times Bool} O^\rho_i();$

$O^\rho_i() = ini\_O^\rho_i.seq_\rho(O^\rho_i, M^{O^\rho_i}).O^\rho_i() \ \forall i \in \{1 \ldots k\} \ \& \ i \neq j;$

$O^\rho_j() = ini\_O^\rho_j.seq_\rho(O^\rho_j, M^{O^\rho_j}).\overline{end\_O^\rho_j}.O^\rho_j().$

The overall translation of the system is $P = (\nu V'')s^{init}() \mid \prod_{s \in S}(SD^s() \mid \prod_{O \in SD^s} O())$

Here $V' = \{ini\_O^\rho_1 \ldots, ini\_O^\rho_k, end\_O^\rho_j\}$, $V'' = \{ini\_s_i, end\_s_i \mid \exists SD(s_i)\}$ and V is a set of internal variables of a given sequence diagram as it was defined in the translation of sequence diagrams in section 4.

As an example of the proposed compositional translation consider a system consisting of an abstract level state diagram in Fig. 8 and a sequence diagram in Fig. 1 corresponding to the state s_2. The result of the translation is:

$s_1() = \tau.s_2()$

$s_2() = \overline{ini\_s_2}.end\_s_2.s_1() + \overline{ini\_s_2}.end\_s_2.s_2()$

$SD^{s_2}() = (\nu\, ini\_O_1,\ ini\_O_2,\ end\_O_2,\ m)\ ini\_s_2.\overline{ini\_O_1}.\overline{ini\_O_2}.end\_O_2.$
$\overline{end\_s_2}.SD^{s_2}()$

$O^{s_2}_1() = ini\_O_1.\overline{m}.\overline{end\_O_1}.O^{s_2}_1()$

$O^{s_2}_2() = ini\_O_2.m.O^{s_2}_2()$

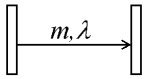
$P = (\nu\, ini\_s_1,\ ini\_s_2,\ end\_s_1,\ end\_s_2)\ s_1() \mid SD^{s_2}() \mid O^{s_2}_1() \mid O^{s_2}_2()$

6 Using the translation

6.1 Analysis of process algebras specifications

The aim of the translation of UML specification to π-calculus

is applying analysis techniques available for process algebras to the UML specification. Here we describe how we can make a performance analysis of system specified in UML using the result of our translation. For this type of analysis we add to the messages in a sequence diagram rates yielding a translation in the stochastic π-calculus (see, e.g. [16]) to express those quantitative measures. Stochastic π-calculus differs from the classical π-calculus in section 3 only for the prefix that has the form $\langle \pi, \lambda \rangle$, where λ is the unique parameter of an exponential distribution. In Fig. 9 you can see the translation of a message extended by rate to the stochastic π-calculus process. The exponential distribution enables us to recover a continuous time Markov chain from which performance measures are computed by using standard numerical techniques [14].



$$\Longrightarrow \quad (\nu m)(\langle \overline{m}, \lambda \rangle \mid \langle m, \lambda \rangle)$$

Fig. 9. Translation of a stochastic message.

6.2 WEB based micro-business case study

Here we apply our translation to a real case study that was developed in the context of the DEGAS project [6]. This case study is a WEB based service that is distributed to the users of a community and that enables the realization of micro-business based on peer-to-peer authentication and communication paradigm. Sequence diagrams are used to detail communication intensive parts of the system.

We present here a translation of a part of the specification that describes the e-commerce process as it seen from a buyer. After a seller has been selected the buyer has to handshake with it, and to authenticate. Then the buyer will search the selling list to get the available items, with which he can prepare its basket to be bought and make an e-commerce transaction. The specification is composed from an e-commerce abstract level state diagram (Fig. 10) and sequence diagrams describing handshaking (Fig. 12, above), authentication (Fig. 12, below) and searching of the seller list (Fig. 11). The result of the translation is shown in Appendix.

For the given case study the performance aspect is critical. Performance issues arise because the communication links are subject to unpredictable throughput variations. Extending the sequence diagrams with quantitative measures on arrows, we can analyze, for example, dependency of request time from number of requests which can appear concurrently in the system as we present in [7].

7 Conclusion

In this paper we discussed a translation to the π-calculus of non homogeneous UML specifications. We proposed an approach based on sequence and state diagrams. As a prolongation of this investigation it would be interesting to extract this translation for other types of UML diagrams to present more complex software systems. Furthermore, we are currently implementing our translation and integrating it with open-source UML tools.

This paper is a step towards the use of formal methods in the current practice of software development. The main contribution of our extraction of process algebra specifications from UML diagrams is the hiding of formal details from the designers when performing analysis.

Finally, we have implicitly defined formal semantics of UML sequence di-

agrams based on the operational semantics of the π-calculus.

As a last remark, assuming the semantic model of sequence diagrams presented in [18], we claim that a sequence diagram is behaviourally equivalent to the π-calculus process obtained by our translation.

References

[1] Alur, R., Etessami, K., Yannakakis M.: Inference of message sequence charts. Proc. ICSE'00, Limerick, Ireland (2000)

[2] Aredo, D.B.: Semantics of UML Sequence Diagrams in PVS. Proc. UML2000

[3] Booch, G., Rumbaugh, J. and Jacobson, I.: UML notation guide, version 1.1. Rational Software Corporation, Santa Clara, CA (1997)

[4] Broy, M. et al.: From MSCs to statecharts. Distributed and parallel embedded systems. Kluwer Academic Publisher (1999)

[5] Canevet, C., Gilmore, S., Hillston, J., and Stevens, P.: Performance modelling with UML and stochastic process algebras. Proc. UK PEW (2002)

[6] Caraguili C., Piazza D., Mura I. et al.: Specification in UML of case studies. DEGAS project deliverable 24, http://www.omnys.it/degas/ (2002)

[7] Caraguili C., Priami C. et al.: Static and Dynamic Analysis of case studies. DEGAS project deliverable 25, http://www.omnys.it/degas/ (2003)

[8] Dumond, Y., Girardet, D. and Oquendo, F.: A Relationship Between Sequence and Statechart Diagrams. Proc. <<UML>>2000, York, UK (2000)

[9] Gogolla M., Ziemann P. and Kuske S.: Towards an Integrated Graph Based Semantics for UML, Proc. GT-VMT 2002, Barcelona, Spain (2002)

[10] Heckel R. and Sauer S.: Strengthening UML Collaboration Diagrams by State Transformations. Proc. FASE 2001

[11] Latella, D., Majzik, I., Massink, M.: Toward a formal operational semantics of UML statechart diagrams. Proc. FMOODS'99, Florence, Italy (1999)

[12] Mauw, S. and Reniers M.A.: Operational semantics for MSC'96. Computer networks (Amsterdam, Netherlands), Vol. 31, N17 (1999)

[13] Milner, R.: Communicating and Mobile Systems: the π-calculus. Cambridge Univ. Press (1999)

[14] Nottegar, C., Priami, C. and Degano, P.: Performance Evaluation of Mobile Processes via Abstract Machines. IEEE Trans. on Software Engineering,Vol. 27/10 (2001)

[15] Pettit IV, R.G. and Gomaa, H.: Validation of Dynamic Behavior in UML Using Colored Petri Nets. Proc. UML2000

[16] Priami, C.: Stochastic π-calculus. The Computer Journal. Vol. 38, N 6 (1995)

[17] Roubtsova, E. and Kuiper R. Process semantics for UML component specifications to assess inheritance. Proc. GT&VMT'02, Elsevier Journal, Electronic Notes in Theoretical Computer Science 72(4), Barcelona, Spain (2002)

[18] Uchitel, S. and Kramer J.: A Workbench for Synthesising Behaviour Models from Scenarios. Proc. ICSE'01, Toronto, Canada. ACM Press, (2001)

[19] Whittle, J. and Schumann J.: Generating Statechart designs from scenarious. ICSE'00, Limerick, Ireland (2000)

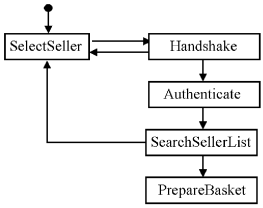


Fig. 10. E-Commerce State Diagram.

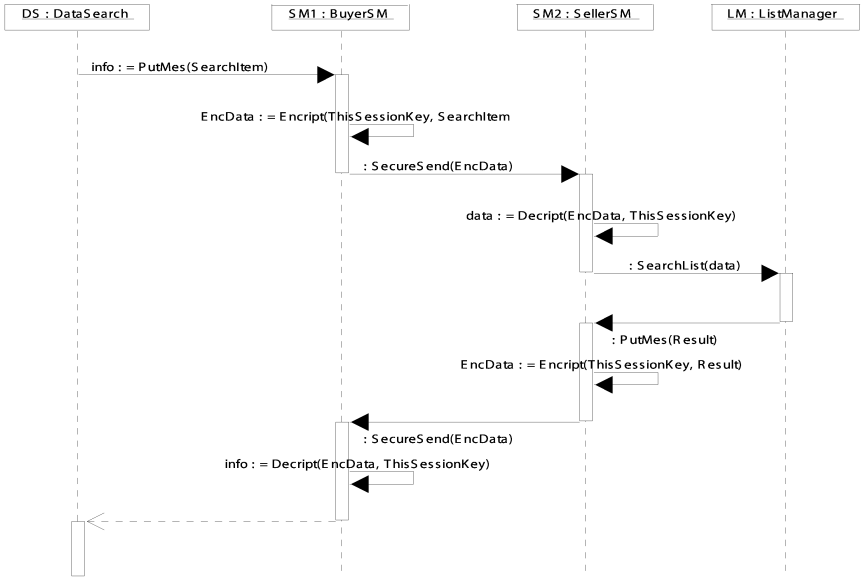


Fig. 11. SearchSellerList sequence diagram.

8 Appendix

Result of the translation of the UML case study specification. $SelectSeller() =$

$\tau.Handshake()$

$Handshake() = \overline{ini\_Handshake}.end\_Handshake.SelectSeller() +$
$\overline{ini\_Handshake}.end\_Handshake.Authenticate()$

$Authenticate() = \overline{ini\_Authenticate}.end\_Authenticate.SearchSellerList()$

$SearchSellerList() =$
$\overline{ini\_SearchSellerList}.end\_SearchSellerList.PrepareBasket() +$
$\overline{ini\_SearchSellerList}.end\_SearchSellerList.SelectSeller()$

$PrepareBasket() = \tau.\mathbf{0}$

$SD^{SearchSellerList}() = (\nu V^{SearchSellerList})ini\_SearchSellerList.\overline{ini\_DS}.$
$\overline{ini\_SM1}.$
$\overline{ini\_SM2}.$
$\overline{ini\_LM}.end\_DS.$
$\overline{end\_SearchSellerList}.SD^{SearchSellerList}()$

$$DS^{SearchSellerList}(SearchItem, info) =$$
$$ini\_DS.\overline{PutMes1}\langle SearchItem\rangle.return^{PutMes1}(info).end\_DS.$$
$$DS^{SearchSellerList}(SearchItem, info)$$
$$SM1^{SearchSellerList}(EncData) =$$
$$ini\_SM1.PutMes1(SearchItem).\tau.\overline{SecureSend1}\langle EncData\rangle.$$
$$SecureSend2(EncData).\tau.\overline{return^{PutMes1}}\langle info\rangle.$$
$$SM1^{SearchSellerList}(EncData)$$
$$SM2^{SearchSellerList}(EncData, data, Result) =$$
$$ini\_SM2.SecureSend1(EncData,).\tau.\overline{SearchList}\langle data\rangle.$$
$$PutMes2(Result).\tau.\overline{SecureSend2}\langle EncData,\rangle.$$
$$SM2^{SearchSellerList}(EncData, data, Result)$$
$$LM^{SearchSellerList}(data, Result) =$$
$$ini\_LM.SearchList(data).\overline{PutMes2}\langle Result\rangle.$$
$$LM^{SearchSellerList}(data, Result)$$
$$SD^{Authenticate}() = (\nu V^{Authenticate})ini\_Authenticate.\overline{ini\_SM1}.\overline{ini\_SM2}.\overline{ini\_BM}.end\_SM1.\overline{end\_Authenticate}.SD^{Authenticate}()$$
$$SM1^{Authenticate}(EncData,) =$$
$$ini\_SM1.\tau.\overline{SecureSend}(EncData,).return^{SecureSend}.\overline{end\_SM1}.$$
$$SM1^{Authenticate}(EncData,)$$
$$SM2^{Authenticate}(Buyer, BuyerData) =$$
$$ini\_SM2.SecureSend(EncData,).\tau.\overline{RetriveLogPwd}(Buyer).$$
$$return^{RetriveLogPwd}(BuyerData).\tau.\overline{return}^{SecureSend}.$$
$$SM2^{Authenticate}(Buyer, BuyerData)$$
$$BM^{Authenticate}(Buyer, BuyerData) =$$
$$ini\_BM.RetriveLogPwd(Buyer).\tau.\overline{return}^{RetriveLogPwd}.$$
$$BM^{Authenticate}(Buyer, BuyerData)$$
$$SD^{Handshake}() = (\nu V^{Handshake})ini\_Handshake.\overline{ini\_SelM}.\overline{ini\_SM1}.\overline{ini\_SM2}.\overline{ini\_BM}.end\_SM1.\overline{end\_Handshake}.SD^{Handshake}()$$
$$SelM^{Handshake}(\ldots) =$$
$$ini\_SelM.RetrieveURL(Seller).\overline{return}^{RetrieveURL}(Seller\_URL).$$
$$RetrieveKey1(Seller).\overline{return}^{RetrieveKey1}(LastSessionKey).$$
$$StoreKey(ThisSessionKey,).\overline{return}^{StoreKey}.SelM^{Handshake}(\ldots)$$
$$SM1^{Handshake}(\ldots) =$$
$$ini\_SM1.\overline{RetrieveURL}(Seller).return^{RetrieveURL}(Seller\_URL).$$
$$\overline{RetrieveKey1}(Seller).return^{RetrieveKey1}(LastSessionKey).\tau.$$
$$\overline{SecureSend1}(EncData,).SecureSend2(EncData,).\tau.\tau.$$
$$\overline{StoreKey}(ThisSessionKey,).return^{StoreKey}.\overline{end\_SM1}.SM1^{Handshake}(\ldots)$$
$$SM2^{Handshake}(\ldots) =$$
$$ini\_SM2.SecureSend1(EncData,).\overline{RetrieveKey2}(Buyer).$$
$$return^{RetrieveKey2}(LastSessionKey).\tau.\tau.\overline{StoreKey}(ThisSessionKey,).\tau.$$
$$\overline{SecureSend2}(EncData,).SM2^{Handshake}(\ldots)$$
$$BM^{Handshake}(\ldots) =$$
$$ini\_BM.RetrieveKey2(Buyer).\overline{return}^{RetrieveKey2}(LastSessionKey).$$
$$StoreKey(ThisSessionKey,).BM^{Handshake}(\ldots)$$
$$P = (\nu V')(SelectSeller() \mid SD^{Handshake}() \mid$$

$SelM^{Handshake}(Seller, Seller\_URL, LastSessionKey, ThisSessionKey, EncData,) \mid$
$SM1^{Handshake}(Seller, Seller\_URL, LastSessionKey, ThisSessionKey) \mid$
$SM2^{Handshake}(LasSessionKey, ThisSessionKey, EncData, Buyer) \mid$
$BM^{Handshake}(LasSessionKey, ThisSessionKey, Buyer) \mid$
$SD^{Authenticate}() \mid SM1^{Authenticate}(EncData,) \mid$
$SM2^{Authenticate}(Buyer, BuyerData) \mid$
$BM^{Authenticate}(Buyer, BuyerData) \mid$
$SD^{SearchSellerList}() \mid DS^{SearchSellerList}(EncData) \mid$
$SM1^{SearchSellerList}(EncData, data, Result) \mid$
$SM2^{SearchSellerList}(EncData, data, Result) \mid$
$LM^{SearchSellerList}(data, Result))$,

where $V = \{ini\_Handshake, end\_Handshake, ini\_SearchSellerList, end\_SearchSellerList, ini\_Authenticate, end\_Authenticate\}$ and $V^{\langle name \rangle}$ is defined as a set of all messages of the sequence diagram $\langle name \rangle$; ini_O for each object $\langle name \rangle$ and end_O for the object of $\langle name \rangle$ receiving the last message.

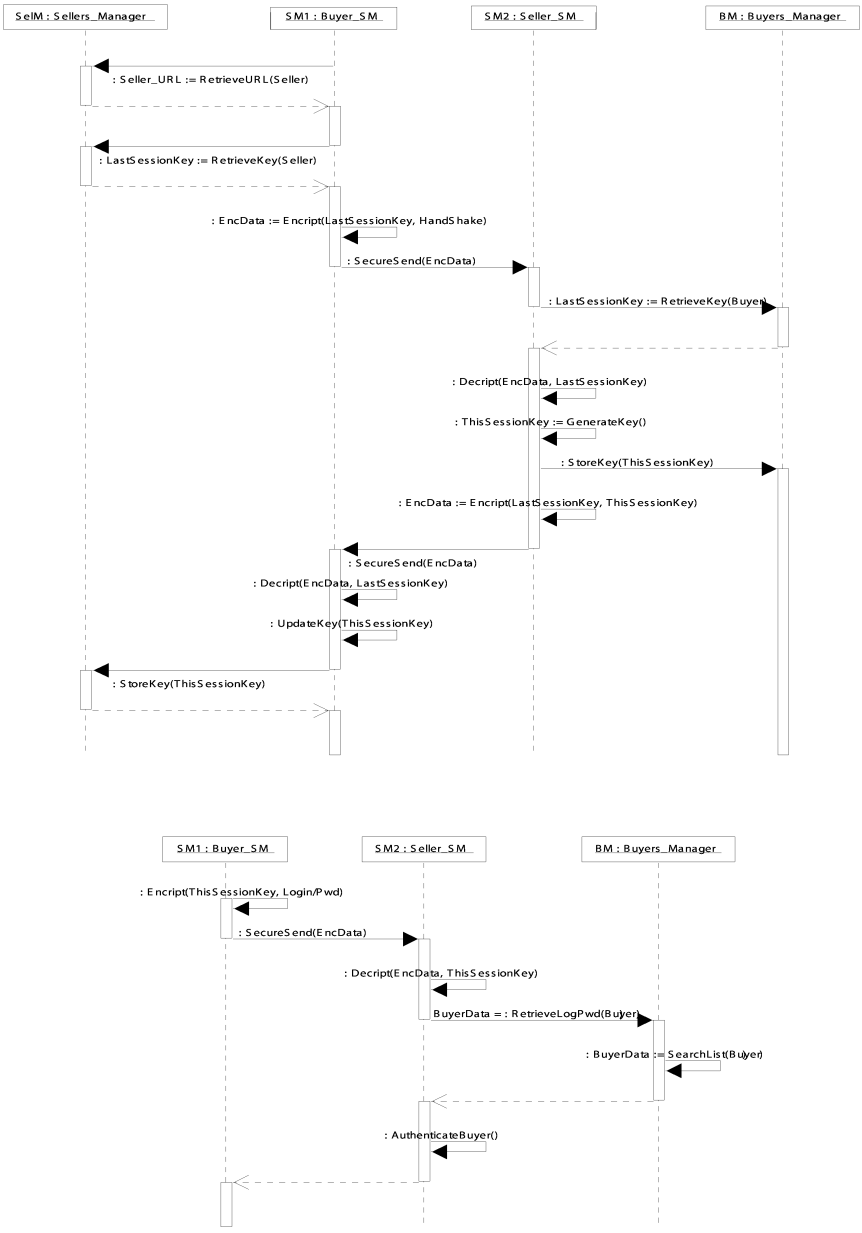


Fig. 12. Handshake (above) and Authenticate (below) sequence diagrams.