UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

EXTRACTION OF PI-CALCULUS SPECIFICATIONS FROM UML SEQUENCE AND STATE DIAGRAMS

Katerina Korenblat and Corrado Priami

February 2003

Technical Report # DIT-03-007

http://www.omnys.it/degas

DEGAS IST-2001-32072

Design Environments for Global ApplicationS

Consortium: UNITN (I) University of Trento (Coordinator), IMM (DK) Institute of Mathematical Modelling - Technical University of Denmark, DIPISA (I) Dipartimento di Informatica - Università di Pisa, UEDIN (UK) University of Edinburgh - MTCI (I) Motorola Technology Center Italy - OMNYS (I) Omnys Wireless Technology

Extraction of π-calculus specifications from UML sequence and state diagrams

---

Author(s): Katerina Korenblat and Corrado Priami

Participant(s): UNITN

Workpackage: WP4 Extraction, Reflection and Integration

Document number: WP4-UNITN-I01-Int-001

Security: Pub **Nature:** R **Version:** 0.1 **Pages:** 17

Extraction of π-calculus specifications from UML sequence and state diagrams*

Katerina Korenblat and Corrado Priami
Dipartimento di Informatica e Telecommunicazioni
University of Trento - Povo, Italy
pokozy@science.unitn.it, priami@dit.unitn.it

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. An implicit result of the paper is also the definition of a formal semantics for UML sequence diagrams.

1 Introduction

The Unified Modeling Language (UML) [1] 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 a description of different aspects of a complex (software) system. In this paper we restrict our attention on specifications including only sequence and state diagrams. Such a choice is often sufficient for specifying the behaviour of a whole system and can be considered as a first step in handling multi-diagram UML specifications.

To implement a formal analysis of a UML specification, we propose to translate it to some formal notation. The target formalism of the translation that we select is process algebras [4]. 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

*This work is partially supported by the DEGAS (Design Environment fore Global Applications) project IST-2001-32072 funded by the FET Proactive Initiative on Global Computing

actions, ii) $-|-$ that is the parallel composition of processes, iii) $-+-$ that denotes a nondeterministic choice. We can view process algebras from different levels of abstraction. A common interpretation is seeing these calculi as specification languages that must be refined towards a real code. The theory of behavioral analysis developed for process algebras (see e.g., [4]) postulates that refined descriptions are still expressed in the same calculus but through different programs. Then, some relations (usually a bisimulation) are established between the two descriptions of the system to ensure that an implementation behaves according to its specification. We are proposing here a different use of process algebras. We interpret these calculi as an intermediate language into which UML specifications can be translated. A state diagram based approach to translation from UML to process algebras was presented in [2] and [3]. In those works states are represented as processes and transitions are represented as actions along communication channels.

In this paper we focus on a sequence diagram based approach, where objects are considered as π-calculus [5] processes and messages as communications among these processes. A state diagram of an object is used for choosing the feasible sequences of the messages occurring in the sequence diagram. This is needed because sequence diagrams only show possible behaviour, thus exposing sample computations. 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.

We now briefly discuss the motivations for the present work. We rely on a standard Unified Modelling Language (UML) to ease formal methods into the software production process. The challanges 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 interacts with UML only 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. Finally, in section 5 we discuss the joint translation of sequence and state diagrams.

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 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 labeled 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 modeled element can proceed during its lifetime as a reaction to discrete events. A state diagram is a graph that represents a state machine.

The formal operational semantics of a state diagram is defined in terms of a Kripke structure [6]. Given a sequence $s_1[t_1\rangle \ldots [t_n\rangle s_n$ of states (s_i) and transitions (t_i) in the Kripke structure of a state diagram S, we call a *trace* of S the sequence of transitions $\langle t_1, \ldots, t_n\rangle$. A concatenation of two traces γ_1 and γ_2 is denoted by $\gamma_1 \circ \gamma_2$.

3 The π-Calculus

In this section we briefly recall the π-calculus [5], 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 $(\nu v_1)(\nu v_2)P$.

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 a commutative monoid,
- $\varepsilon.P \equiv P$,
- $[x = x]P \equiv P$,
- $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P, (\nu x)(R \mid S) \equiv (\nu x)R \mid S$ 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. 1. 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.

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

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

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

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

Table 1: Late transition system for the π-calculus.

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. 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 in the UML model, we translate this message as 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. For example, in Fig. 2 a simple sequence diagram with a single condition x is translated to a summation of two subprocesses representing two possible valuations of x.

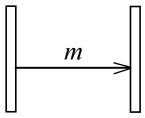
Nested messages initialized by a message with a condition including its return will be discarded if the condition is not satisfied. We translate an explicit return of the message as a usual message with the name $return^{\langle message\ name \rangle}$. For example, we translate a message $[x]m$ with an explicit return (Fig. 3) as a sequence of messages $[x]m$ followed by $return^m$.

A branching of several messages is translated as a parallel composition of these messages synchronized before continuation because all of branched messages have to be delivered. In more details, 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. Note that if the branching messages have alternative conditions

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

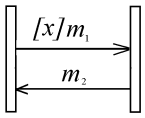
We now formally define a translation function from UML sequence diagrams to the π-calculus. Given a message m, we define a set of nested messages $nest(m)$ in a case of an explicit return as a set of messages that become enable by the sending of m before the return of m (including the return), oras an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

Fix a sequence diagram Sq with a set of objects $\mathcal{O}$, a set of



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

Figure 1: Translation of a message.



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

Figure 2: Translation of a condition.

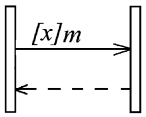
messages $\mathcal{M}es$ and a set of conditions $\mathcal{C}$. Let $\rho : \mathcal{C} \cup \{\Lambda\} \rightarrow Bool$ be an evaluation function of conditions that returns $true$ for the special condition Λ. Given a message m, we define c^m as a condition corresponding to m, or as Λ if there is no condition on m. Given an object O and a message m related with O, we define the function $tr_\rho : \mathcal{O} \times \mathcal{M}es \rightarrow \mathcal{P}$[1] as

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

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 any set $M_i = \{m_i^1, \ldots, m_i^{k_i}\}$ represent branching messages and $m_i^j \in \mathcal{M}es$ for each $j \in \{1 \ldots k_i\}$ and $i \in \{1 \ldots n\}$. We define the translation function $seq_\rho : \mathcal{O} \times \mathcal{M}es^* \rightarrow \mathcal{P}$ as follows:

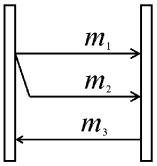
$seq_\rho(O, \langle\ \rangle) = \mathbf{0}$, and

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



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

Figure 3: Translation of a return.



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

Figure 4: Translation of a branching construction.

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

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

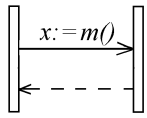
where $rest(\langle M_i, \ldots, M_n\rangle)$ is a subsequence of $\langle M_i, \ldots, M_n\rangle$ starting after returns of all messages from M_i.

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

In the proposed translation we always compare names with constant values. Name matching can be implemented as presented in Fig. 6. The idea of such strong translation is to construct an additional subprocess for the valuation of the variable.

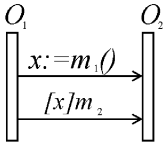
As final remarks, we give some notes for simplifying the result of the translation.

- *Repeating conditions.* We leave in a process only the first instance of a repeating condition.
- *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 and can be skipped together with a corresponding sending



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

Figure 5: Translation of an assignment.



$$\begin{aligned} P^{O_1} &= \overline{m_1}.\, return^{m_1}(x).\, [x = true]\overline{m_2} \\ P^{O_2} &= m_1.\, val(x).\, \overline{return}^{m_1}\langle x\rangle.\, m_2 \\ Val &= \overline{val}\langle true\rangle + \overline{val}\langle false\rangle \\ P &= P^{O_1} \mid P^{O_2} \mid Val \end{aligned}$$

Figure 6: Translation of a condition using name matching.

of the synchronizing message. In the case of a single active branched message we can translate it as usual sequence message. For an example see the end of the following subsection.

4.1 An Example

To illustrate our translation consider the sequence diagram in Fig. 7 representing a slight variant of the Phone system in [3]. 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 $\rho_2 : \rho_2(c_1) = false$, $\rho_2(c_2) = true$.

The result of the translation is shown below.

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

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

$$\begin{aligned} Receiver^{\rho_1} &= connect.\, \overline{return}^{connect}\langle busy\rangle.\, [busy = true]\varepsilon \\ Receiver^{\rho_2} &= connect.\, \overline{return}^{connect}\langle busy\rangle.\, [busy = false]call.\, \overline{answer}.\, talking_1. \\ &\quad \overline{talking}_2.\, \overline{return}^{call} \end{aligned}$$

$$System = (\nu V)(Phone^{\rho_1} \mid Receiver^{\rho_1} \mid Caller^{\rho_1}) + (Phone^{\rho_2} \mid Receiver^{\rho_2} \mid Caller^{\rho_2})$$

where V is the set of channels of $System$.

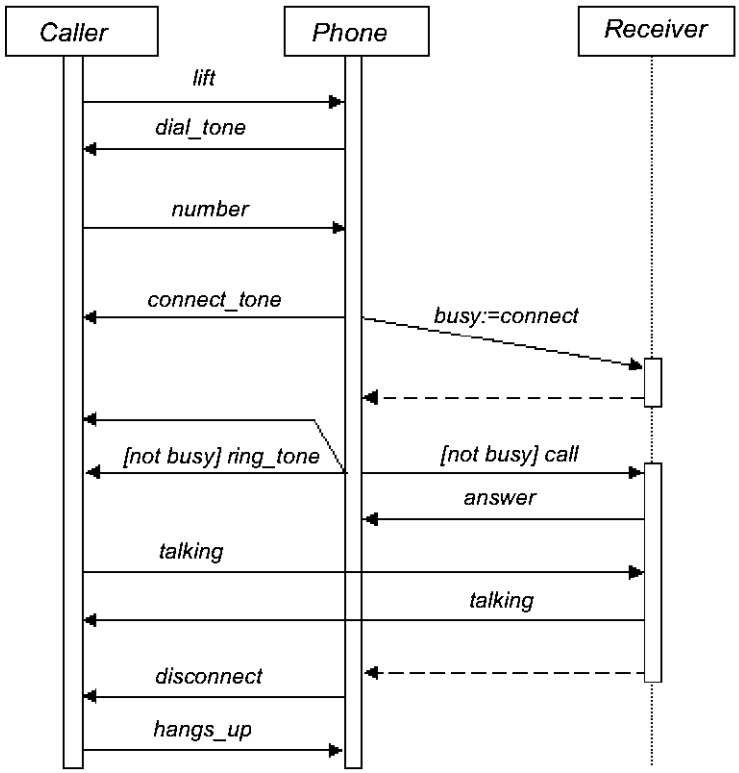


Figure 7: Sequence diagram of the *Phone system*.

Now we illustrate the simplification technique of empty branched subprocesses described in the previous section. In $Phone^{\rho_1}$ we can 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 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 di-

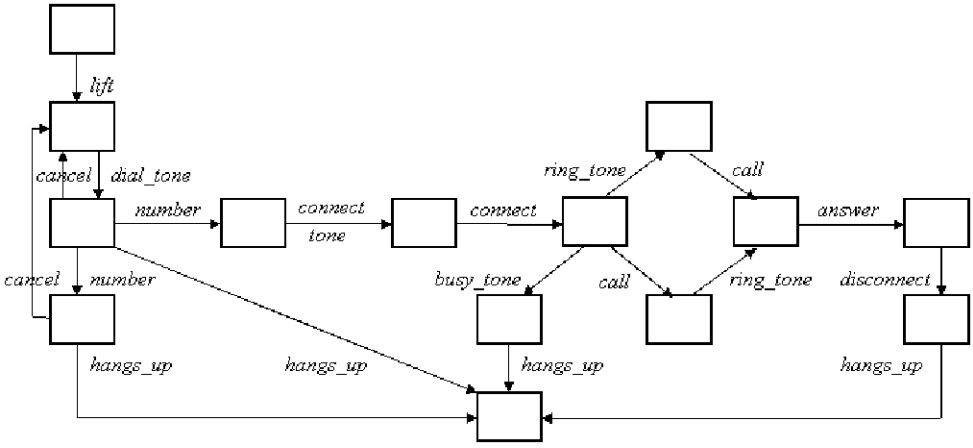


Figure 8: State diagram of the object *Phone* of the *Phone system*.

agrams). 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 [2] for producing translation to Performance Evaluation Process Algebra (PEPA).

The last approach requiresa 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 spec-

ification of all objects. This is why we prefer a sequence diagram based approach.

5.1 Sequence Diagram Based Translation

In the previous section we considered the case of nonskipped messages. However, in a general case a message in a sequence diagram can be skipped. It can be useful, for example, to present several 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 have to be skipped in an allowed behaviour. To illustrate such a situation we add to the Phone system on Fig. 7 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

$$\langle ..dial\_tone,\ timeout\_tone,\ number.. \rangle$$

besides the correct sequences

$\langle ..dial\_tone,\ timeout\_tone,\ hangs\_up \rangle$ and $\langle ..dial\_tone,\ number.. \rangle$.

To consider only correct sequences of messages we need some additional information which can be taken from other types of diagrams. For example, such information can be obtained from state diagrams which describe behaviour of objects.

UML gives instruments for the description of a system but it does not guarantee completeness and accordance of the model. So to make a correct translation we have to formalize our assumptions about a system. We consider a system made up of one sequence diagram and a number of state diagrams (one for each object of the sequence diagram) with the following requirements:

- Messages in a sequence diagram have unique names. It is necessary for correct correspondence between messages in the sequence diagram and transitions in the state diagrams.

- A name of a transition corresponding to a transmission of a message coincides with a name of this message.
- Each communication is presented in the state diagram of an object explicitly.

The behavior of a single object is described by a state diagram and communication between objects is described by a sequence diagram. Here we fix the simplest for translation assumption about state diagrams. In alternative, we can assume, for example, that a message absent in the state diagram can be skipped only with the immediately previous message.

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.

The first step of the translation is to rename transitions in such a way that there are no two transitions with the same name available in one state. For this purpose we enumerate such transitions by adding of superscripts to their names.

Given a state diagram S, we define an *external* trace of S as a restriction of a trace of S on a set of transitions corresponding to messages. A set of external traces of S is denoted by $\mathcal{T}(S)$.

Now we modify the translation of sequence diagrams proposed in section 4 by taking information from state diagrams to single out correct execution sequences. The difference from the original algorithm is the definition of the function seq_ρ. For readability we define seq_ρ assuming no branching of messages. Hereafter, we write *head* for a trace in a state diagram that reaches the current state. Fix an evaluation ρ and an object O with a sequence of sent/received messages $\langle m_1, \ldots, m_n \rangle$, and a state diagram S^O. Beginning from $seq_\rho(O, \emptyset, \langle m_1, \ldots, m_n \rangle)$ we calculate the translation function $seq_\rho : \mathcal{O} \times \mathcal{T}(S^O) \times \mathcal{M}es^* \to \mathcal{P}$ as follows:
$seq_\rho(O, \gamma, \langle\,\rangle) = \mathbf{0}$ where $\gamma \in \mathcal{T}(S^O)$, and
$seq_\rho(O, head, \langle m_i, \ldots, m_n \rangle) =$

$$\sum_{\{j \;|head \circ m_i^j \in \mathcal{T}(S^O)\}} tr_\rho(O, m_i).seq_\rho(O, head \circ m_i^j, \langle m_{i+1}, \ldots, m_n \rangle) + skip,$$

where $skip = seq_\rho(O, head, \langle m_{i+1}, \ldots, m_n \rangle)$, and it is present if

there exists a maximal trace $head \circ t$ for some $t \neq m_k$ or there exists a trace $head \circ m_k$ for some $k > i$.

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. 7) and the state diagram for the object *Phone* (Fig. 8).

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:

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

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. 7) and the state diagram for the object *Phone* (Fig. 8):

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

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.

6 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 first 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.

Finally, we have implicitly defined formal semantics of UML sequence diagrams based on the operational semantics of the π-calculus.

References

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

[2] C. Canevet, S. Gilmore, J. Hillston, and P. Stevens. Performance modelling with UML and stochastic process algebras. To appear in Proceedings of UK PEW, 2002.

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

[4] R. Milner. *A* Calculus of Communicating Systems. Prentice-Hall, 1989.

[5] R. Milner, J. Parrow and D. Walker. A calculus of mobile processes. Inform. and Comput. 100 (1), 1992.

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