MiKO
Mikado Koncurrent Objects

Francisco Martins * Liliana Salvador † Luís Lopes †
Vasco T. Vasconcelos ‡

January 2005

1 A TyCO-calculus based model

In this section we introduce the syntax and operational semantics of MiKO. MiKO is a distributed higher order instance of TyCO-calculus [5] based on the MIKADO's membrane model [2, 3], distributing processes over a flat network of domains. The migration of code from one domain to another passes always through the domains's membranes. The calculus contains special primitives to handle migration between such domains.

Syntax

Figure 1 describes the syntax of the calculus. Consider a set of names $\mathcal{N}$ that do not possess the constants in, out, and mkdom and a set of labels $\mathcal{L}$, disjoint from $\mathcal{N}$, ranged over by l. To improve readability we use r–t to range over names of domains, a–c to range over names of channels, and x–z to ranged over unknowns.

The calculus is organised in two layers: *networks* and *processes*. A network consists of a set of *domains* and *network messages* running independently in parallel. A domain $x\{m\rangle P_1\}[P_2]$ is a named location x composed by a *membrane* $m\rangle P_1$ and a *contents* P_2. Membranes define the interaction between the domain and the outside world (either the contents part of the domain or the remainder of the network). The set of public methods m defines the interface with the domain and process P_1 controls the migration of code with the domain. The contents is the computation oracle of the domain. For the following we consistently denote membranes by R–T, processes by P and Q, and networks by H, L, and N. A network message $x\,!\,M$ denotes a message M heading to domain x. The remaining networks are standard in process calculi.

The syntax of membranes and contents is mostly the standard syntax of TyCO [4, 6] enriched with constructs $\mathsf{in}[P]$ that launches a process P in the contents part of the domain, $\mathsf{out}[x, M]$ that sends a message M to the domain x, and $\mathsf{mkdom}[x, m\rangle P, P]\ \mathsf{in}\ P'$ that creates a new domain x with guardian $m\rangle P$

*Universidade dos Açores, Departamento de Matemática
†Universidade do Porto, Departamento de Ciência de Computadores
‡Universidade de Lisboa, Departamento de Informática

$$
\begin{array}{rll}
N ::= & \mathsf{inaction} \mid x\{m\rangle P\}[P] \mid x\,!\,M \mid N \mid N \mid \mathsf{new}\ x\ N & \text{(networks)} \\
P ::= & \mathsf{inaction} \mid x\,!\,M \mid x?m \mid x?{*}m \mid P \mid P \mid \mathsf{new}\ x\ P \mid & \text{(processes)} \\
& A\vec{V} \mid \mathsf{in}[P] \mid \mathsf{out}[x, M] \mid \mathsf{mkdom}[x, m\rangle P, P]\ \mathsf{in}\ P' & \\
M ::= & l[\vec{V}] & \text{(messages)} \\
m ::= & \{l_i = A_i\}_{i \in I} & \text{(methods)} \\
A ::= & (\vec{x})P & \text{(abstractions)} \\
V ::= & x \mid A & \text{(values)}
\end{array}
$$

Figure 1: Syntax of MiKO

$$
\begin{array}{rll}
\mathsf{new}\ x\ N \mid L \equiv \mathsf{new}\ x\ (N \mid L) & x \notin \mathrm{fn(L)} & \text{(N-SRC)} \\
s\{m\rangle S\}[\mathsf{new}\ c\ P] \equiv \mathsf{new}\ c\ s\{m\rangle S\}[P] & c \notin \mathrm{fn(m}\rangle\mathrm{S)} & \text{(N-PSR)} \\
s\{m\rangle \mathsf{new}\ c\ S\}[P] \equiv \mathsf{new}\ c\ s\{m\rangle S\}[P] & c \notin (\mathrm{fn(P)} \cup \mathrm{fn(m)}) & \text{(N-MSR)}
\end{array}
$$

Figure 2: Structural congruence on networks

and contents P, visible in the membrane P' of the creator's domain. Messages are tuples of values sent on channels and interpreted as method invocations. The values that can be communicated in messages are either names or process abstractions.

Operational Semantics

The operational semantics of the calculus is presented with the help of congruence relations on networks and on processes. The bindings of the calculus are scope restriction $\mathsf{new}\ x\ P$ and abstraction $(\vec{x})P$, binding free occurrences of x and $\vec{x}$, in process P. The set of free names for networks N, processes P, and methods m is denoted by fn(N), fn(P) and fn(m), respectively.

Structural Congruence

The structural congruence relation on networks, $\equiv$, is the least congruence relation closed under the rules in figure 2 adding the commutative monoid rules for parallel composition, having inaction as the neutral element. The structural congruence between processes is that of TyCO [5] and therefore is omitted.

The following example illustrates the operational semantics of the calculus. Consider a client-server session manager: a client that establishes a session with a server and finishes the session at the end of the conversation. The protocol we envisage is very simple: (1) the client issues a connect request to the server; (2) the server replies with a session identifier; (3) the client issues a disconnect message to terminate the session.

First we comment on the server's membrane. The server's membrane must provide two public methods to handle connection and disconnection: *connect* and *disconnect*. The *connect* method creates a private channel *sessionID*, sends

it to the client via an `out` operation and launches an object at the server's contents to handle the session using the `in` process. The object that handles the session just implements the *quit* operation. A more elaborated session handler is presented in the mathServer example later in section 2. The *disconnect* method triggers the end of the session by selecting the *quit* operation from the handler. A possible implementation of the server's membrane is

```
server_memb = {
    connect (client, replyTo) =
        new sessionID
        out [client, enter [()replyTo ! sessionID]] |
        in [
            sessionID ? {
                quit() = inaction
            }
        ]
    disconnect (client, sessionID) =
        in [sessionID ! quit[]]
    }
    |>
    {
        inaction
    }
```

The client's membrane needs to offer a *connect* and *disconnect* methods, to interact with the server's counterpart. Moreover, it must provide a method (that we name *enter*) to receive the responses from the server. The methods *connect* and *disconnect* just invoke the server operations. Both client and server do not need a state, so the body of the membrane is the `inaction` process. Following is a possible implementation of the client's membrane.

```
client_memb = {
    connect (server, replyTo) =
        out [server, connect [myDomain, replyTo]]
    enter (x) =
        in [x []]
    disconnect (server, sessionID) =
        out [server, disconnect [sessionID] ]
    }
    |>
    {
        inaction
    }
```

An interaction between the client and the server may be written as

Reduction Rules

The reduction rules for networks and processes are given in figures 3 and 4.

The migration of messages between domains proceeds as depicted below

$\mathsf{client}_{\mathsf{cont}} =$
`new` connection
client ! connect [server, connection] |
connection ? (sessionID) myDomain ! disconnect [sessionID]

$$s\{m\rangle P\}[s\,!\,M \mid Q] \to s\{m\rangle m.M \mid P\}[Q] \qquad \text{(N-SEND)}$$

$$s\{m\rangle \mathsf{out}[r, l[\vec{V}]] \mid S\}[P] \to r\,!\,l[s, \vec{V}] \mid s\{m\rangle S\}[P] \qquad \text{(N-OUT)}$$

$$s\,!\,M \mid s\{m\rangle S\}[P] \to s\{m\rangle S \mid m.M\}[P] \qquad \text{(N-COM)}$$

$$s\{m\rangle \mathsf{in}[P] \mid S\}[Q] \to s\{m\rangle S\}[Q \mid P] \qquad \text{(N-IN)}$$

$$s\{m\rangle \mathsf{mkdom}[r, m'\rangle S, P] \text{ in } R \mid T\}[Q] \to \mathsf{new}\ r\ (r\{m'\rangle S\}[P] \mid s\{m\rangle R \mid T\}[Q]) \quad \text{if } r \notin (\mathrm{fn(T)} \cup \mathrm{fn(Q)}) \qquad \text{(N-MKD)}$$

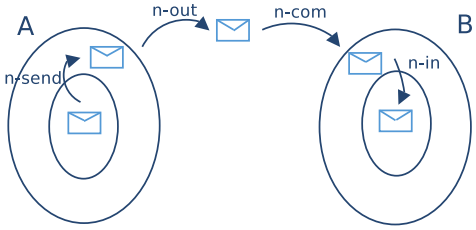
$$\frac{S \to T}{s\{m\rangle S\}[P] \to s\{m\rangle T\}[P]} \qquad \text{(N-MEMB)}$$

$$\frac{P \to Q}{s\{m\rangle S\}[P] \to s\{m\rangle S\}[Q]} \qquad \frac{N \to L}{N \mid H \to L \mid H} \qquad \text{(N-PROC, N-PAR)}$$

$$\frac{N \to L}{\mathsf{new}\ x\ N \to \mathsf{new}\ x\ L} \qquad \frac{N \equiv L \quad N \to H}{L \to H} \qquad \text{(N-SRES, N-STR)}$$

where $\{l_i = A_i\}_{i \in I}.l_j[\vec{V}] = A_j\vec{V}, j \in I$

Figure 3: Reduction rules on networks



The N-SEND rule delivers a message from the contents part of the domain to its membrane. The message M initiates the execution of a method from the domain's interface in its membrane. Rules N-OUT and N-COM route the message through the network. Upon reception, the message interacts with the target membrane by selecting a public method. Rule N-IN drives processes from membranes to contents of the domain.

A domain can only be created from the membrane of an existing domain. To create a domain, rule N-MKD, we provide a domain name (r), a guardian ($m'\rangle T$), a contents (Q) that is going to run on the domain, and a part of the membrane (R) of the creator domain where the new domain is visible. Rules N-MEMB, N-PROC, N-PAR, and N-RESC allow reductions inside the domain, the parallel composition of networks, and channel restriction, respectively. Finally, N-STRC brings structural congruence on networks to the reduction relation.

As for processes, figure 4, the reception of messages by objects (followed by the selection of the appropriate method and the instantiation of the method's

$$c?m \mid c!M \to m.M \qquad \text{(P-COM)}$$

$$c?*m \mid c!M \to c?*m \mid m.M \qquad \text{(P-COMR)}$$

$$((\vec{x})P)\vec{V} \to P[\vec{V}/\vec{x}] \qquad \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \text{(P-SUBS, P-PAR)}$$

$$\frac{P \to Q}{\mathsf{new}\ c\ P \to \mathsf{new}\ c\ Q} \qquad \frac{P \equiv Q \quad P \to R}{Q \to R} \qquad \text{(P-SRESC, P-STRC)}$$

Figure 4: Reduction Rules on Processes

body) constitutes the basic communication mechanism of the calculus, as captured by axiom M-COM.

The remaining rules are straightforward.

As an example of a reduction in MiKO, consider

$$\text{server}\{\mathsf{server}_{\mathsf{memb}}\}[\texttt{inaction}] \mid \text{client}\{\mathsf{client}_{\mathsf{memb}}\}[\mathsf{client}_{\mathsf{cont}}]$$

It reduces as follows:

1. The communication between $\mathsf{client}_{\mathsf{cont}}$ and $\mathsf{client}_{\mathsf{memb}}$ is made by the rule N-SEND. The message client ! connect [server, connection] is sent to the membrane and interacts with the *connect* method.

2. Membrane $\mathsf{server}_{\mathsf{memb}}$ sends messages to the network by the rule N-OUT. The *connect* method sends the message server ! connect [myDomain, replyTo] to the network towards the server domain.

3. Network messages interact with $\mathsf{client}_{\mathsf{memb}}$ by the rule N-COM. The message sent to the server enters in the membrane and interacts with the *enter* method.

4. $\mathsf{server}_{\mathsf{memb}}$ interacts with its contents $\mathsf{server}_{\mathsf{cont}}$ by the rule N-IN. The *connect* method when triggered launches the process replyTo ! sessionID to the server's contents.

5. By rule M-COM, the process launched in the contents reduces with the object connection ? (sessionID) myDomain ! disconnect [sessionID] and the $\mathsf{server}_{\mathsf{cont}}$ sends a disconnect message server!disconnect[sessionID] to the client by rules N-SEND and N-OUT.

6. The message enters in $\mathsf{client}_{\mathsf{memb}}$, reduces with the *disconnect* method and the process sessionID ! quit[] is launched to the contents, by rules N-COM, M-COM and M-IN.

2 The Type System

In this section, we define the syntax of types and the type inference system for MiKO.

$$\alpha ::= \{l_i \colon \vec{\alpha}_i\}_{i \in I} \mid t \mid \mu t.\alpha$$

Figure 5: Syntax of types

$$\frac{\Gamma \vdash_s m \colon \alpha \quad \Gamma \vdash_s P \quad \Gamma \vdash_s Q \quad \Gamma(s^{\mathsf{d}}) \approx \alpha}{\Gamma \vdash s\{m \rangle\!\rangle P\}[Q]}\ (\text{TN-NET}) \qquad \frac{\Gamma \vdash \vec{V} \colon \vec{\alpha} \quad \Gamma(s^{\mathsf{d}}).l \approx \vec{\alpha}}{\Gamma \vdash s\,!\,l[\vec{V}]}\ (\text{TN-MSG})$$

$$\frac{}{\Gamma \vdash \mathsf{inaction}}\ (\text{TN-INAC}) \qquad \frac{\Gamma \vdash N \quad \Gamma \vdash L}{\Gamma \vdash N \mid L}\ (\text{TN-PAR}) \qquad \frac{\Gamma, x^{\bullet} \colon \alpha \vdash N}{\Gamma \vdash \mathsf{new}\ x\ N}\ (\text{TN-RESC})$$

Figure 6: Typing networks

Syntax

We fix a countable set of type variables ranged over by t, and let α, β range over types. A sequence of types $\alpha_1 \ \ldots \ \alpha_n$ is abbreviated as $\vec{\alpha}$. The syntax of types is given in figure 5.

A type of the form $\{l_i : \vec{\alpha}_i\}_{i \in I}$ describes locations of objects or domains containing n methods labelled $l_1 \ldots l_n (n > 0)$ with types $\vec{\alpha}_1 \ldots \vec{\alpha}_n$. Types are interpreted as rational (regular infinite) trees. A type of the form $\mu t.\alpha$ (with $\alpha \neq t$) denotes the rational tree solution of the equation $t = \alpha$. If α is a type, denote by α^* its associated rational tree. An interpretation of recursive types as infinite trees naturally induces an equivalence relation $\approx$ on types, by putting $\alpha \approx \beta$ if $\alpha^* = \beta^*$.

Typing rules

Type assignments to names are formulae $x : \alpha$, for x a name and α a type. Typings, denoted by Γ are partial functions on finite domains from names to types $(x \colon \alpha)$. We write $\mathtt{dom}(\Gamma)$ for the domain of Γ. When $x \notin \mathtt{dom}(\Gamma)$, we write $\Gamma, x \colon \alpha$ for the type environment Γ' such that $\mathtt{dom}(\Gamma') = \mathtt{dom}(\Gamma) \cup \{x\}$, $\Gamma'(x) = \alpha$, and $\Gamma'(y) = \Gamma(y)$, for $y \neq x$. So, the expression $\Gamma(x)$ denotes the type α if $x \colon \alpha$ and $\Gamma(\vec{x})$ represents the sequence of types $\vec{\alpha}$. We denote by x^{d}, x^{c}, and $x^{\bullet}$, respectively, identifiers that are domains, channels, and that can be both.

The type system, described in figures 6–9 includes four kinds of judgements: (a) judgement $\Gamma \vdash N$ asserts that network N is well typed under typing assumptions Γ; (b) judgement $\Gamma \vdash_s P$ means that process P is running at domain s and is well typed under typing assumptions Γ; (c) judgement $\Gamma \vdash_s A \colon \vec{\alpha}$ denotes that abstraction A has a sequence of parameters of type sequence $\vec{\alpha}$; and finally (d) judgement $\Gamma \vdash_s \vec{V} \colon \vec{\alpha}$ assigns types to sequences of values.

Networks. To type a domain $s\{m \rangle\!\rangle S\}[P]$, rule TN-NET, one has to type the membrane $m \rangle\!\rangle S$ and the contents P. Domain s possesses the type of its interface

$$\text{(TP-OBJ)}\quad \frac{\Gamma \vdash_s m\colon \alpha \quad \Gamma(c^{\mathsf{c}}) \approx \alpha}{\Gamma \vdash_s c?m} \qquad \text{(TP-MSG)}\quad \frac{\Gamma \vdash_s \vec{V}\colon \vec{\alpha} \quad \Gamma(x^{\bullet}).l \approx \vec{\alpha}}{\Gamma \vdash_s x\,!\,l[\vec{V}]} \qquad \text{(TP-IN)}\quad \frac{\Gamma \vdash_s P}{\Gamma \vdash_s \mathsf{in}[P]}$$

$$\text{(TP-OUT)}\quad \frac{\Gamma \vdash_r (s\vec{V})\colon \vec{\alpha} \quad \Gamma(r^{\mathsf{d}}).l \approx \vec{\alpha}}{\Gamma \vdash_s \mathsf{out}[r, l[\vec{V}]]} \qquad \text{(TP-MKD)}\quad \frac{\Gamma, r^{\mathsf{d}}\colon \alpha \vdash r\{m \rangle S\}[P] \quad \Gamma, r^{\mathsf{d}}\colon \alpha \vdash_s R}{\Gamma \vdash_s \mathsf{mkdom}[r, m \rangle S, P]\ \mathsf{in}\ R}$$

$$\text{(TP-OBJR)}\quad \frac{\Gamma \vdash_s c?m}{\Gamma \vdash_s c?{*}m} \qquad \text{(TP-RESC)}\quad \frac{\Gamma, c^{\mathsf{c}}\colon \alpha \vdash_s P}{\Gamma \vdash_s \mathsf{new}\ c\ P} \qquad \text{(TP-INAC)}\quad \frac{}{\Gamma \vdash_s \mathsf{inaction}}$$

$$\text{(TP-APP)}\quad \frac{\Gamma \vdash_s A\colon \vec{\alpha} \quad \Gamma \vdash_s \vec{V}\colon \vec{\beta} \quad (\vec{\alpha} \approx \vec{\beta})}{\Gamma \vdash_s A\vec{V}} \qquad \text{(TP-PAR)}\quad \frac{\Gamma \vdash_s P \quad \Gamma \vdash_s Q}{\Gamma \vdash_s P \mid Q}$$

Figure 7: Typing processes

$$\text{(TA-METH)}\quad \frac{\forall i \in I, \quad \Gamma \vdash_s A_i\colon \vec{\alpha}_i}{\Gamma \vdash_s \{l_i = A_i\}_{i \in I}\colon \{l_i : \vec{\alpha}_i\}_{i \in I}}$$

Figure 8: Typing abstractions

$$\text{(TV-VAR)}\quad \Gamma, x^{\bullet}\colon \alpha \vdash_s x^{\bullet}\colon \alpha \qquad \text{(TV-ABS)}\quad \frac{\Gamma, \vec{x}^{\bullet}\colon \vec{\alpha} \vdash_s P}{\Gamma \vdash_s (\vec{x})P\colon \vec{\alpha}} \qquad \text{(TV-SEQ)}\quad \frac{\Gamma \vdash_s V_1\colon \alpha_1 \ \ldots\ \Gamma \vdash_s V_n\colon \alpha_n}{\Gamma \vdash_s \vec{V}\colon \vec{\alpha}}$$

where $\{l_i : \vec{\alpha}_i\}_{i \in I}.l_j = \vec{\alpha_j}, j \in I$

Figure 9: Typing values

m. Rule TN-MSG expresses that s is a domain that has a method with the same name as the message heading to it and with a compatible type.

Processes. When typing processes, we record (under the turnstile) the domain that hosts the process. The information is relevant when typing the `out` operation, since we stamp messages with the name of the sending domain. Rule TP-OBJ expresses the fact that c must be a channel and has type compatible with method m. Rule TM-MSG is similar to TP-MSG, but here x can be a channel or a domain, since we use the same constructor to send messages to membranes or interact with objects. To type the `in` constructor in domain s, we type P in s using rule TN-IN. Rule TP-OUT types the `out` constructor in the origin domain s and the sequence of values $s\vec{V}$ in the destination domain r, since abstractions run in the destination domain. (Names are network-wide, so may be typed in any domain.) Notice that one appends the origin domain s to the list of parameters. To type creation of domains, rule TM-MKD, the created domain $r\{m\Rightarrow S\}[P]$ must be itself well-typed as well as membrane R.

Values. When typing a method, by rule TM-METH, we type each abstraction with the type of the arguments of the method l_i. The remaining typing rules are direct.

Some properties of the type system

We now present some properties of the type system for MiKO. The following lemmas are proved by straightforward induction on the depth of the derivation of the judgements in the hypothesis.

Lemma 2.1 (Strengthening Lemma)

i. *If* $\Gamma, x\colon \alpha \vdash N$ *and* $x \notin \mathrm{fn(N)}$, *then* $\Gamma \vdash N$

ii. *If* $\Gamma, x\colon \alpha \vdash_s P$ *and* $x \notin \mathrm{fn(P)}$, *then* $\Gamma \vdash_s P$

Lemma 2.2 (Weakening Lemma)

i. *If* $\Gamma \vdash N$, *then* $\Gamma, x\colon \alpha \vdash N$

ii. *If* $\Gamma \vdash_s P$, *then* $\Gamma, x\colon \alpha \vdash_s P$

Lemma 2.3 (Substitution Lemma)

If $\Gamma, \vec{x}\colon \vec{\alpha} \vdash_s P$ *and* $\Gamma \vdash_s \vec{V}\colon \vec{\beta}$ *with* $\vec{\alpha} \approx \vec{\beta}$, *then* $\Gamma \vdash_s P[\vec{V}/\vec{x}]$.

Subject-reduction (theorem 2.1) expresses a consistency property between the operational semantics and the typing rules. We prove the theorem by induction on the depth of the derivations $N \to L$ and $P \to Q$, respectively.

Theorem 2.1 (Subject-Reduction)

i. *If* $\Gamma \vdash N$ *and* $N \to L$, *then* $\Gamma \vdash L$.

ii. *If* $\Gamma \vdash_s P$ *and* $P \to Q$, *then* $\Gamma \vdash_s Q$.

Example. Our second example illustrates a mathematical server, math server for short. A math server accepts computation requests from clients, compute the data, and reply back the answers. We build the math server on top of the first example we show. Therefore, the client establishes a session with the server and issues repeatedly computation requests. Upon finish it closes the session with the server.

The server's membrane now offer two additional methods: *eval* and *replyResult*. The *eval* method accepts computation requests and routes them to the respective session handler that sits on the computational area of the math server. The *replyResult* method delivers the computation results to the client. Our math server accounts for the number of concurrent sessions it handles, illustrating a membrane with a state associated. Therefore, the *connect* operation consumes one session, restored when a *disconnect* operation occurs. The state is recorded via a private replicated object *localStatus*, updated via an object that offers two called *connect* and *disconnect* to easily identify them with the domain public methods.

The math server membrane may be programmed as

```
mathServer_memb = {
    connect (client, replyTo) =
        controller ! connect [client, replyTo]
    eval (client, x) =
        in [x [] ]
    replyResult (client, x) =
        out [client, enter [x]]
    disconnect (client, sessionID) =
        controller ! disconnect [sessionID]
    }
    |
    new localStatus
    localStatus ?* (availSessions) =
        controller ? {
            connect (client, replyTo) =
                if availSessions > 0 then
                    sessionIDManager |
                    localStatus ! [availSessions - 1]
                else
                    localStatus ! [availSessions]
            disconnect (sessionId) =
                in [sessionID ! disconnect [] ] |
                localStatus ! [availSessions + 1]
            } |
    localStatus ! [5]
}
```

Notice that the *connect* and *disconnect* methods redirect their requests to update the number of sessions handled concurrently by the server. The number of simultaneous sessions starts with five. Then, whenever a connection request arrives, the server verifies if there are still resources available. If there are, a new session is created and the number of available sessions decreased, otherwise

the request is simply discarded. When disconnecting from the math server, the number of available resources is restored.

The sessionIDManager is responsible for generating a new session identifier and for setting a session handler at the contents part of the math server. Since the client may issue repeated requests, the session handler needs to be a persistent object. The object instantiates itself until a disconnect message is received. Every time a new session is created, the sessionHandler is launched in the math server's contents where it is responsible for computing the mathematical operations selected by the client and for sending the results back to it. The client sends a message to the server that, after being filtered by the membrane, selects the operation method in the correspondent object.

The sessionIDManager we foresee is

```
sessionIDManager =
    new sessionID
    out [client, enter [() replyTo ! [sessionID] ] ] |
    in [
        new sessionHandler
        sessionHandler ?* (self, client) =
            self ? {
                add (n, m, replyTo) =
                    mathServer ! replyResult [client, () replyTo ! [n + m]] |
                    sessionHandler ! [self, client]
                neg (n, replyTo) =
                    mathServer ! replyResult [client, () replyTo ! [0 - n]] |
                    sessionHandler ! [self, client]
                disconnect () =
                    inaction
            } |
        sessionHandler ! [sessionID, client]
    ]
```

The client's membrane provides the same *connect*, *enter*, and *disconnect* methods as in the first example. Additionally, there is a new method *eval* that sends a computation to be executed by the math server. As before, there is no status associated with the client's membrane. The client's membrane we provide is

```
    mathClient_memb = {
        connect (server, replyTo) =
            out [ server, connect [replyTo] ]
        enter (server, x) =
            in [ x [] ]
        eval (server, x) =
            out [server, eval [x]]
        disconnect (server, sessionID) =
            out [server, disconnect [sessionID] ]
        }
        |
        inaction
    }
```

To interact with the math server, we provide a process that establishes a session and then requests the addition of two values and the negation of the result of the previous operation. Finally, after receiving from the server the result of the negation, it closes the session with the math server.

The process is

```
mathClient_cont = {
    new replyTo
    client ! connect [ mathServer, replyTo ] |
    replyTo ? (sessionID) =
        new result
        client ! eval [ mathServer, () sessionID ! add [3, 4, result] ] |
        result ? (x) =
            client ! eval [ mathServer, () sessionID ! neg [x, result]] |
            result ? (x) =
                io ! printi [x] |
                client ! disconnect [mathServer, sessionID]
```

References

[1] *The π-calculus A Theory of Mobile Processes*. Cambridge University Press, 2001.

[2] G. Boudol. Core programming model, release 0. Mikado Deliverable D1.2.0, 2002.

[3] G. Boudol. A parametric model of migration and mobility, release 1. Mikado Deliverable D1.2.1, 2003.

[4] Vasco T. Vasconcelos. Core-tyco appendix to the language definition, version 0.2. Technical report, Faculty of Sciences of the University of Lisbon, 2001.

[5] Vasco T. Vasconcelos. Tyco gently. Technical report, Faculty of Sciences of the University of Lisbon, 2001.

[6] Vasco T. Vasconcelos and Rui Bastos. Core-tyco the language definition, version 0.1. Technical report, Faculty of Sciences of the University of Lisbon, 1998.

[7] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. *1st International Symposium on Object Technologies for Advanced Software*, 742:460–474, 1993.