Encoding π-calculus

David Guaspari

January 28, 2010

1 Introduction

This note describes an encoding of π-calculus in the process model of [1], hereafter called "our model." It assumes familiarity with that paper and with π-calculus. We consider the formulation of monadic π-calculus as given in, for example, [2]. The syntax of a π-calculus process term is:

$$P \ ::= \ \mathbf{0} \mid \Sigma_{i=1}^{n} \pi_i . P_i \mid P|Q \mid !P \mid (\nu x)P$$

where $n > 0$ and a prefix π_i is either the "get" operation $c(x)$ or the "put" operation $\overline{c}\,x$. We will model the basic π-calculus semantics that does not require communications to be chosen "fairly." Section 5 discusses this further.

Several basic differences between π-calculus and our model must be negotiated. The notion of location, fundamental in our model, doesn't exist in π-calculus. In our model, a process sends a message to a known recipient by labeling the message with the recipient's location, whereas a π-calculus process sends a message to a channel without knowing which other processes might be reading from it. Our model is asynchronous and processes can act only on local knowledge; whereas in π-calculus, communication is synchronous and rendezvous are effected by the environment, using global information.

Terminology and notation Unless otherwise indicated, "process" refers to a process in the sense of our model. "Target processes" will encode π-calculus process terms and "bookkeeping processes" will manage and constrain their interactions. A "Target component" is a component, in the sense of our model, that encapsulates a target process. Unexplained notation comes from [1].

2 Communication

It seems appealing, at first glance, to encode a π-calculus channel as a process, but that runs into problems because π-calculus semantics can require global decisions to choose which messages to transmit (and which processes will receive them); our model has no built-in magic to make such global decisions.

Instead, we introduce a central bookkeeping process *Comm*, at location l_{comm}, that manages all communication. A target process at location l says it's available to communicate on channel b by sending a message to *Comm* that says, roughly, "location l is willing to get from [put to] channel b." *Comm* decides which communications will occur and carries them out by sending a value to the process doing the "get" and an acknowledgement to the process doing the "put"—both of which block until they receive these messages. In general, target processes communicate directly only with bookkeeping processes, not with one another.[1]

Comm's decisions about which communications occur must be made nondeterministically, without ruling out any possibilities. In our model all nondeterminism comes from the environment, and processes—in particular, *Comm*—must be deterministic, but a simple trick can be used to achieve the desired effect.

π-calculus terms that can immediately engage in communications have the form

$$\pi_1.P_1 + \ldots + \pi_n.P_n$$

A process at location l that encodes this term will ask *Comm* to perform one of those communications by placing on its external part[2]

$$[l_{comm} : \langle l, [\pi_1, \ldots, \pi_n]\rangle]$$

a message of type

$$\mathit{Reqs} = \mathit{Loc} \times (\mathit{Prefix\ List})$$

We abuse notation by using expressions of the form $c(x)$ and $\bar{c}\,v$ to denote both π-calculus prefixes and NuPl terms of type *Prefix* that represent them. The message includes l in order to provide *Comm* with a return address.

Note that the list of π_i could contain repetitions—e.g., the process could be

$$c(x).P + c(x).Q$$

so that the corresponding *Reqs* is $\langle l, [c(x), c(x)]\rangle$. Accordingly, *Comm*'s replies will have type

$$\text{Value} \times \mathbb{N}$$

A reply will contain both a return value (or acknowledgement) and the index of the request chosen. In this case it would send to location l a message of the form $\langle v, 1\rangle$ or $\langle v, 2\rangle$. The requesting process will use the index to determine whether to continue as P or as Q.

In making communication synchronous, we will ensure

Invariant 1 *The external part of a target component contains at most one element (representing the list of communications for one guarded choice).*

[1]There's a tiny exception: one target process may send a special fire message to another in order to activate it; but such messages do not correspond to communications in the π-calculus execution being simulated.

[2]Strictly speaking, a process does not have an external part; only a component does. The official definitions will not be so loose.

2.1 A nondeterministic version of *Comm*

We will first describe *Comm* by allowing it to invoke a nondeterministic choice operator. As noted, our model has no such thing. We then show how to implement that choice operator with a trick that pushes the nondeterminism into the environment.

We define *Comm* with the operator *RecPr*, defined in [1], that encapsulates a recursive definition of a process in terms of an implicit state Σ and a *next* function

$$\mathit{next} : \mathit{Msg} \times \Sigma \to \Sigma \times \mathit{Ext}$$

Given an input from *Msg* and a state, *next* computes the next state and a list of messages to send.

In this nondeterministic version, the state of *Comm* is simply a finite partial function from locations to lists of prefixes:

$$\mathit{state}_C : \mathit{Loc} \xrightarrow{\mathit{fp}} \mathit{Prefix\ List}$$

$\mathit{state}_C[l]$ is defined when *Comm* has a pending request from l—the request to perform one of the communications in $\mathit{state}_C[l]$.

The only inputs to *Comm* are of type *Reqs*. Consider an input $\mathit{loc} : \vec{\pi}$, where $\vec{\pi} = [\pi_1, \ldots, \pi_n]$. The definition of $\mathit{next}(\mathit{loc} : \vec{\pi}, \mathit{state}_C)$ splits into cases, according to whether there exists a match—that is, whether there exist $i, j \in \mathbb{N}$, $l \in \mathit{Loc}$ such that π_i and $\mathit{state}_C[l][j]$ are matching prefixes (a put and a get on the same channel). Because a π-calculus process can't rendezvous with itself we should also insist that $l \neq \mathit{loc}$; invariant 2 guarantees that any match found will satisfy that constraint.

No match exists

$$\mathit{next}(\mathit{loc} : \vec{\pi}, \mathit{state}_C) = \langle \mathit{state}_C[\mathit{loc}\ := \ \vec{\pi}], \mathit{nil}\rangle$$

If there is no match, add $\vec{\pi}$ to state_C and send no messages.

A match exists If there is a match, nondeterministically choose such a triple, $\langle i', j', l'\rangle$.

$$\mathit{next}(\mathit{loc} : \vec{\pi}, \mathit{state}_C) = \langle \mathit{state}_C[l'\ := \ \mathit{nil}], \mathit{msgs}\rangle$$

where the list *msgs* consists of the two messages that carry out the rendezvous determined by $\langle i', j', l'\rangle$. The pending request $\mathit{state}_C[l']$ is deleted.

The definition of *msgs* splits into two cases (depending on who's sending and who's receiving):

If $\pi_{i'} = c(x)$ and $\mathit{state}_C[l'][j'] = \bar{c}\,v$, $\mathit{msgs} = [\mathit{loc} : \langle v, i'\rangle,\ l' : \langle \mathit{ack}, j'\rangle]$

If $\pi_{i'} = \bar{c}\,v$ and $\mathit{state}_C[l'][j'] = c(x)$, $\mathit{msgs} = [\mathit{loc} : \langle \mathit{ack}, j'\rangle,\ l' : \langle v, i'\rangle]$

As noted, the qualification $l \neq \mathit{loc}$ is redundant because of invariant 2.

Invariant 2 *Comm cannot receive a message from location l unless $\mathit{state}_C[l] = \mathit{nil}$.*

2.2 Encoding nondeterministic choice

The choice of the triple $\langle i', j', k' \rangle$ has to range nondeterministically over all possible matches. It suffices to implement the operation "choose nondeterministically from a list" because *Comm* can compute the list of all possible matching triples and request a choice from that. The (trivial) trick is to have the environment make the decision.

Making *one* such choice is easy. Introduce a new process, *Choose*, at a fresh location l_C. *Choose* communicates only with *Comm*. When *Comm* wants *Choose* to pick an element from the list $[x_1, \ldots, x_n]$ it places the n separate messages $l_C : x_1, \ldots, l_C : x_n$ on its external part. The action of *Choose* is simply to send back to *Comm* the first message it receives and to ignore subsequent messages.

If we wish *Choose* to make choices repeatedly, it must know when it's seeing a stale message (from a choice that has already been made, and which should be ignored) and when it's seeing a new message (which should be echoed). It suffices for *Comm* to keep an integer counter, *id*, and proceed as follows: when it wants a choice from $[x_1, \ldots, x_n]$ it increments *id* and then place the messages $l_C : \langle x_1, id \rangle, \ldots, l_C : \langle x_n, id \rangle$ on its external part. *Choose* keeps track of the counter values that it has seen; it echoes $\langle m, k \rangle$ if it has not seen k, and ignores $\langle m, k \rangle$ if it has. (An equally good alternative would be to fork off a new version of a one-time-only *Choose* every time a choice is wanted.)

This is not quite good enough, because the decisions that *Comm* makes are not independent of one another. Consider this scenario: $state_C$ is defined only at location l and $state_C[l] = [\bar{c}\, v]$. *Comm* receives the message $\langle l_1, [c(x)] \rangle$, determines the list of possibilities, $[\langle 1, 1, l \rangle]$ and asks *Choose* to pick one. Before *Choose* returns a result, *Comm* receives the message $\langle l_2, [c(y)] \rangle$, determines *the same* list of possibilities, because $state_C$has not changed, and asks *Choose* to choose from it. Then *Choose* will return $\langle 1, 1, l \rangle$ to *Comm* in response to both requests and *Comm* will execute both rendezvous, sending the value v both to l_1 and l_2—which a mistake. We therefore construct *Comm* so that after submitting one request for a choice, it waits for an answer before it submits the next one; and it queues any messages received from target processes while it's waiting. To provide one more illustration of programming in our model, section 6 shows how to do the (straightforward) bookkeeping.

3 Encoding π-calculus terms and programs

The preceding section has factored the model of communication from the rest of the semantics: to send or messages a target process sends an appropriate vector to *Comm* and waits for a response.

If P is a π-calculus process term, then its denotation will be a map

$$[\![P]\!] : \mathit{Loc} \to \mathit{Process}$$

Intuitively, for any location l, $[\![P]\!](l)$ represents the behavior of P if it is "installed" at location l.

Notation: For readability, we'll commonly write $[\![P]\!](l)$ as $[\![P, l]\!]$

We also define $\mathcal{M}(P) \in \mathit{System}$, a system (in the sense of [1]) whose runs simulate the executions of P as a stand-alone π-calculus program. One component of $\mathcal{M}(P)$ will be constructed from $[\![P]\!]$ and the others will be constructed from bookeeping processes.

Processes of the form $[\![P, l]\!]$ will have certain features in common:

- Each responds to a special message, called fire, that activates it.
- Each communicates only with three bookkeeping processes: *Comm*; the *location server LServer*; and the *name server NServer*.

Strictly speaking, *LServer* is a map whose inputs are finite sets of locations and whose outputs are processes. If L is a finite set of locations, then $\mathit{LServer}(L)$ is a process that, in response to an input returns a message with a name that differs from any location in L and any location it has previously returned. *NServer* does the corresponding thing for names. The description of $[\![P]\!]$ will treat "get a new location" and "get a new name" as basic operations, though they really involve sending a message to the appropriate server and waiting for a reply—analogous to, but simpler than, the interaction between *Comm* and *Choose*.

Note: If locations are modeled as atoms then, for technical reasons, *LServer* is limited to serving up an arbitrarily large but finite number of new locations. However, there is no reason why locations have to be atoms. Alternatively, one may adopt the less elegant solution of simulating the encoding described here at a single location. (Each process has a unique id that simulates its location; a message addressed to a process contain that id in its header; etc.)

$[\![P, l]\!]$ may create new processes. Our model requires that we know the level of each process being created. Accordingly, we introduce a map Λ from π-calculus terms to $\mathbb{N}$ and will guarantee invariant 3.

Invariant 3 *For any l, $[\![P, l]\!]$ is a process of level $\Lambda(P)$.*

The definition of Λ is simple:

$$\begin{aligned}
\Lambda(\mathbf{0}) &= 0 \\
\Lambda(\Sigma_{i=1}^{n} \pi_i.P_i) &= \max\{\Lambda(P_1), \ldots.\Lambda(P_n)\} \\
\Lambda(P|Q) &= \max\{\Lambda(P), \Lambda(Q)\} + 1 \\
\Lambda(!P) &= \Lambda(P) + 1 \\
\Lambda((\nu x)P) &= \Lambda(P)
\end{aligned}$$

Notation: This section will contain process definitions in the form "if the input satisfies condition ϕ, the transition is ... elsif ψ, the transition is ...." They should be understood as saying that, in all other cases, the transition is a no-op. Formally, this is a recursive definition:

$$P = \lambda m.\,\mathsf{if}\ \phi(m)\ \mathsf{then}\ \ldots\ \mathsf{elsif}\psi(m)\ \ldots\mathsf{else}\ \langle P, \mathit{nil}\rangle$$

We will use P and Q, with decorations, to vary over π-calculus process terms and boldface $\mathbf{P}$ and $\mathbf{Q}$, decorated, to vary over processes.

Note: The definitions in this section make use of primitive operations "get new name" and "get new location." Accordingly, the only messages a target process receives are fire and messages from *Comm*. Explicitly implementing the "new name" and "new location" operations by interaction with *NServer* and *LServer* would add trivial complications.

The base case The base case in the definition of $[\![P]\!]$ is trivial:

$$[\![\mathbf{0}, l]\!] = \mathit{null}$$

where *null* is the process that does nothing:

$$\mathit{null} = \lambda m.\, \langle \mathit{null}, \mathit{nil} \rangle$$

Guarded choice

$$[\![\pi_1.P_1 + \ldots + \pi_n.P_n, l]\!] = \lambda m.\mathsf{if}\ m = \mathsf{fire}\ \mathsf{then}\ \langle \mathbf{Q}, [l_{comm} : \langle l, [\pi_1 \ldots, \pi_n] \rangle] \rangle$$

where

$$\mathbf{Q} = \lambda m.\mathsf{let}\ \langle v, i \rangle = m\ \mathsf{in}\ \langle \mathbf{P}_i, [l : \mathsf{fire}] \rangle$$

and the definition of $\mathbf{P}_i$ depends on π_i:

$$\mathbf{P}_i = \mathsf{if}\ \pi_i = c(x)\ \mathsf{then}\ [\![P_i[x := v], l]\!]\ \mathsf{else}\ [\![P_i, l]\!]$$

That is, when it receives the fire message, the process sends its list of possible communications to *Comm* and becomes process $\mathbf{Q}$. $\mathbf{Q}$ waits for a reply from *Comm*, becomes the appropriate continuation process, and then activates that process by sending it the fire message.

What is the point of "fire"? A process is purely reactive; it acts only in response to an input. But the components that represent π-calculus terms cannot be purely reactive: they must initiate communications by sending requests to *Comm*. We can make a component (initially) "active" by creating it with a nonempty external part; the environment will deliver those messages. To do this, we could generalize the notion of "process message" to include both a process and an initial external part, generalize the notion of boot process correspondingly, etc. Instead, we construct each target process so that it responds appropriately to the fire message, and we have the boot process install the result of fireing the target process.

Digression: a fancy boot process The obvious strategy for encoding parallel composition or replication is to create new processes at new locations. Thus, $[\![P|Q, l]\!]$ will proceed by getting two new locations l_1 and l_2 and sending them, respectively, messages containing processes that encode P and Q. The effect of such "process messages" depends on the boot process to be applied. If we use the default boot process the newly installed processes will not be "active" because they're waiting for the fire message. Thus we must both install them and send the activating message. Without adding assumptions about the environment it will not suffice to have $[\![P|Q, l]\!]$ send both the processes and the fire messages—for there is no guarantee that either process will be installed before the fire message arrives.

The solution we choose is to introduce a fancier boot process that both installs a process and activates it by sending it the message fire. It's a simple modification of the boot process defined in [1]. (To improve readability the definitions write the constructor *pmsg* as μ.)

$$\mathit{boot}^+(n) = \mathit{RecPr}(\mathit{next}, \bot)$$

where

$$\begin{aligned}
\mathit{next}(\bot) = \lambda m. \quad & \textsf{if } m = \mathit{pmsg}(k, \mathbf{Q}) \text{ and } k < n \\
& \textsf{then let } \langle \mathbf{P}, e\rangle = \mathbf{Q}(\textsf{fire})\textsf{in}\langle \mathit{inl}(\mathbf{P}), e\rangle \\
& \textsf{else } \langle \bot, \mathit{nil}\rangle \\
\mathit{next}(\mathit{inl}(\mathbf{Q})) = \lambda m. \quad & \textsf{let } \langle \mathbf{Q}', e'\rangle = \mathbf{Q}(m) \textsf{ in } \langle \mathit{inl}(\mathbf{Q}'), e'\rangle
\end{aligned}$$

Parallel composition The process encoding $P|Q$ will create processes for P and Q at new locations, activate them, and then terminate. This definition clearly preserves invariant 3.

$$\begin{aligned}
[\![P|Q, l]\!] = \lambda m.\, & \textsf{if } m = \textsf{fire then} \\
& \quad \textsf{let } l_1,\, l_2 \text{ be new locations } \textsf{in} \\
& \quad\quad \langle \mathit{null}, [l_1 : \mathit{pmsg}(\Lambda(P), [\![P, l_1]\!]),\, l_2 : \mathit{pmsg}(\Lambda(Q), [\![Q, l_2]\!])]\rangle
\end{aligned}$$

One subtlety: we want l_1 and l_2 to be, in particular, different from l. We will guarantee that by insuring that the locations of all the target processes are either in the finite set with which *LServer* is seeded or are themselves returned from a call on *LServer*.

Replication Replication is just a variant of parallel composition. The process $[\![!P, l]\!]$ will get a new location l_1, install $[\![P, l_1]\!]$ there, and resend the fire message to itself (enabling it, when the message is delivered, to install further models of P at other locations). The definition clearly preserves invariant 3.

$$\begin{aligned}
[\![!P, l]\!] = \lambda m.\, & \textsf{if } m = \textsf{fire then} \\
& \quad \textsf{let } l_1 \text{ be a new location } \textsf{in} \\
& \quad\quad \langle [\![!P, l]\!], [l : \textsf{fire},\, l_1 : \mathit{pmsg}(\Lambda(P), [\![P, l_1]\!])]\rangle
\end{aligned}$$

The ν operator The ν operator is encoded straightforwardly by calling on *NServer*.

$$[\![(\nu x)P, l]\!] = \lambda m.\, \textsf{if } m = \textsf{fire then let } n \text{ be a new name } \textsf{in } \langle [\![P[x := n], l]\!], \mathit{nil}\rangle$$

4 Simulating a π-calculus program

In our model, what executes is not a process but a system. Thus, if P is a π-calculus process term, we define the system $\mathcal{M}(P)$ to represent execution of P as a stand-alone program, its elements interacting only with one another. We may assume that all the bound names of P are distinct.

Let N be the set of all names occurring in P and choose five distinct locations: l_{comm}, l_C, l_N, l_L, and l. $\mathcal{M}(P)$ will consist of the following five components, together with the boot process $\mathit{boot}^+(\Lambda(P)+1)$.

- $\langle l_{comm}, \mathit{Comm}, \mathit{nil}\rangle$
- $\langle l_C, \mathit{Choose}, \mathit{nil}\rangle$
- $\langle l_N, \mathit{NServer}(N), \mathit{nil}\rangle$
- $\langle l_L, \mathit{LServer}(\{l_C, l_{comm}, l_N, l_L, l_0\}), \mathit{nil}\rangle$
- $\langle l, [\![P, l]\!](\mathsf{fire})\rangle$

As discussed in the next section, our semantic model of P is the collection of *fair* runs of $\mathcal{M}(P)$.

5 Fairness, etc.

A correctness proof for this encoding would show that the collection of fair runs of $\mathcal{M}(P)$ simulates execution of P. Instead of proof, we offer hand-waving. However, one possible subtlety should be addressed—why we insist on the fair runs of $\mathcal{M}(P)$, those in which every message placed in an external part is eventually delivered. Without that assumption, the simulation is not guaranteed to make progress. With it—the hands wave—$\mathcal{M}(P)$ simulates the "standard" π-calculus semantics of P, not any kind of "fair" semantics.

To see why we need guaranteed message delivery, consider $P = !c(x).\mathbf{0}|!\overline{c}\,v.\mathbf{0}$. Execution of the π-calculus term P produces only one trace, an infinite sequence that repeatedly sends v on channel c. However, there is an unfair run of $\mathcal{M}(P)$ that does nothing but create new copies of $Q = !\overline{c}\,v.\mathbf{0}$ and never performs any communication at all: At each step the environment has the option to deliver a fire message that forks off another copy of Q, and chooses to do so instead of delivering any "get" or "put" message to *Comm.*

To see why fair runs of $\mathcal{M}(R)$ need not correspond to "fair" executions of R, consider $R = !c(x).P_1|!c(x).P_2|!\overline{c}\,v.Q$. We are guaranteed that all requests for communications will be delivered to *Comm* but not guaranteed that *Comm* responds to every request it receives. For example, there are fair runs of $\mathcal{M}(R)$ in which *Comm* is repeatedly asked to choose between replying to the "get" message that continues as P_1 and the "get" message that continues as P_2—and always chooses the first of these.

6 Appendix: *Comm*

Here is a definition of a deterministic version of *Comm* that makes choices by communicating with *Choose*. A state is now a tuple $\langle \mathit{reqArray}, q, \mathit{waiting}\rangle$ where

$$\begin{array}{rcl} \mathit{reqArray} & : & \mathit{Loc} \xrightarrow{\mathit{fp}} \mathit{PrefixList} \\ q & : & \mathit{Queue}(\mathit{Reqs}) \\ \mathit{waiting} & : & \mathbb{B} \end{array}$$

The variable *reqArray* is what used to be called state_C, and plays the same role; q keeps track of the communication requests that have been received by *Comm* but not yet examined; and *waiting* is true when *Comm* is waiting for an answer from *Choose*.

The construction will guarantee invariant 4:

Invariant 4 *In any reachable state* $\langle \mathit{reqArray}, q, \mathit{waiting}\rangle$*:*

- *If* $l_1 : p_1$ *and* $l_2 : p_2$ *are distinct entries in* q*,* $l_1 \neq l_2$*.*
- *If* $l : p$ *is an entry in* q*,* $\mathit{reqArray}[l] = \mathit{nil}$*.*

Let nil_Q be the empty queue and define

$$\begin{array}{rcl} \mathit{add}(\mathit{nil}_Q, \mathit{reqArray}) & = & \mathit{reqArray} \\ \mathit{add}(\mathit{enqueue}(l : \vec{\pi}, q), \mathit{reqArray}) & = & \mathit{add}(q, \mathit{reqArray}[l \ := \ \vec{\pi}]) \end{array}$$

Note that if $q' \subseteq q$ and q satisfies invariant 4, then so does q'. If q' and *reqArray* satisfy invariant 4 the result of $\mathit{add}(q, \mathit{reqArray})$ is to add all the requests in q' to *reqArray*.

Comm responds to three kinds of input messages: *Reqs* from target processes, triples from *Choose*, and continue messages that it sends to itself. So if state $\sigma = \langle \mathit{reqArray}, q, \mathit{waiting}\rangle$, the definition of $\mathit{next}(m, \sigma)$ splits into three cases:

Case $m = \mathit{reqs} \in \mathit{Reqs}$

$$\mathit{next}(m, \sigma) = (\langle \mathit{reqArray}, \mathit{enqueue}(\mathit{reqs}, q), \mathit{waiting}\rangle, [l_{\mathit{comm}} : \mathsf{continue}])$$

When it receives *reqs* from a target task, *Comm* enqueues *reqs* on q and sends itself a continue message.

Case $m = \langle i', j', l'\rangle$

$$\mathit{next}(m, \sigma) = (\langle \mathit{reqArray}[l' := \mathit{nil}], q, \mathsf{false}\rangle, [l_{\mathit{comm}} : \mathsf{continue}]@\mathit{msgs})$$

where *msgs* is as defined as in section 2.1.

When it receives a triple from *Choose*, *Comm* sends *msgs*, the value and acknowledgment determined by that triple, and deletes from *reqArray* the request that has been matched; it is no longer *waiting*, and sends itself a continue message (so that it will process unexamined messages in q).

Case $m = \mathsf{continue}$

There are two subcases.

If $\mathit{waiting} = \mathsf{true}$,

$$\mathit{next}(m, \sigma) = (\sigma, \mathit{nil})$$

A continue message that arrives while *Comm* is awaiting a reply from *Choice* is ignored.

If $\mathit{waiting} = \mathsf{false}$, let $q = q_1 \oplus q_2$, where

- no entry in q_2 matches any other entry in q_2 or any entry in *reqArray*
- either $q_1 = \mathit{nil}_Q$ or $\mathit{head}(q_1)$ has a match in $\mathit{add}(q_2, \mathit{reqArray})$.

It's easy to see that there is a unique pair (q_1, q_2) satisfying these conditions. Split once again, on the value of q_1.

If $q_1 = \mathit{nil}_Q$

$$\mathit{next}(m, \sigma) = (\langle \mathit{add}(q, \mathit{reqArray}), \mathit{nil}_Q, \mathsf{false}\rangle, \mathit{nil})$$

In this case, there are no matches among the unexamined requests, all are added to the pending requests, emptying the queue. *Comm* is not waiting and sends no messages.

If $q_1 \neq \mathit{nil}_Q$

$$\mathit{next}(m, \sigma) = (\langle \mathit{add}(q_2, \mathit{reqArray}), \mathit{tail}(q_1), \mathsf{true}\rangle, [l_{comm} : \mathsf{continue}, l_C : \mathit{matches})$$

where *matches* is the list of triples representing all possible matches of elements of $\mathit{head}(q_2)$ with elements of $\mathit{add}(q_2, \mathit{reqArray})$.

In this case, no matches can be found in q_2, so all of its elements are added to the pending requests in *reqArray*; $\mathit{head}(q_1)$ can be matched to something in $\mathit{add}(q_2, \mathit{reqArray})$, so *Choice* is asked to pick one and *Comm* is now waiting; the unexamined requests are those remaining in q_2; finally, since q_2 may not be empty, *Comm* tells itself to continue.

References

[1] Mark Bickford and Robert Constable. Generating Event Logics with Higher-Order Processes as Realizers. *To appear.*

[2] Robin Milner. The Polyadic π-Calculus: A Tutorial. LFCS report ECS-LFCS-91-180. http://www.lfcs.inf.ed.ac.uk/reports/91/ECS-LFCS-91-180/