A Synchronous Process Calculus for Service Costs

Siva Anantharaman, Jing Chen, Gaétan Hains

A Synchronous Process Calculus for Service Costs

Siva Anantharaman, Jing Chen, Gaétan Hains
LIFO - Université d'Orléans (France),
e-mail: {siva, chen, ghains}@lifo.univ-orleans.fr

Abstract

We present a process calculus where synchronous composition is the central algebraic notion; equivalences between processes via bisimilarity or trace can be studied quite simply in this calculus, which in addition allows us to model naturally other notions such as service, and quality of service. They can be studied in an algebraic semi-ring setup using notions of cost on the transitions.

1. Introduction

In an earlier work [1], we proposed a calculus for building finite recursion free processes, with synchronization as the central parallel composition operation; such a view allowed us in particular to show that strong bisimulation between finite processes can be identified with the equational congruence between such processes seen as terms, with respect to an equational signature referred to as PACUID: '$+$' denoting non-deterministic choice is Associative-Commutative, Idempotent, and admits the null action as Unit; a binary '$*$' denoting synchronous composition is 2-sided Distributive over '$+$', and Prefixes for actions. To underline the difference with the classical process calculi, we denoted the internal action of processes as θ in [1], instead of using the usual notation τ; it was also shown that the interleaving semantics of CCS, although not part of the formal algebraic setup presented [1], is recoverable in terms of '$+, *$'.

The purpose of the current paper is two-fold. The first is on the formal side where the concern will be to extend such a synchronous calculus to the case of *recursive* finite state processes, so as to provide a sufficiently expressive algebraic basis: we mean thereby it should be able to serve for the formal analysis of processes, based on strong or weak bisimulation equivalence, and strong or weak trace equivalence. For achieving such a formal goal, we shall be shifting the view from processes-as-terms to processes-as-state-machines, as is classically done: Recursive finite state processes will be defined as state machines with guarded transitions, e.g., as in [7], or the ACP calculus of [3] or [2]. But our view, while remaining simple, will also show a natural way based on our synchronization operator '$*$' for reducing the bisimilarity problem between finite state recursive processes to the case of finite *recursion free* processes; the interest is that such a reduction can be used for deciding (non-)bisimulation via equational techniques. This constitutes the essence of the first part of the paper.

The second part of the paper is a little less formal and has a software engineering bias; we show there how our synchronous composition operator '$*$' is handy for building a method for modeling a notion of quality of service for a given client with respect to servers serving one or more clients.

This paper is structured as follows. We first recall (in Section 2) our synchronous calculus for finite recursion-free processes, essentially as was done in [1]. In Section 3 we define recursive finite state processes as a set of nodes (or 'process constants') and a given set of guarded equalities between them. The principal result is that two finite state processes P, Q are bisimilar (notation: $P \sim Q$) if and only if, for every *finite* process J without 'choice' (i.e., without '$+$'), we have $P * J \sim Q * J$. Similar results hold also for equivalence under weak bisimulation or trace (although not mentioned in this paper). The less formal part of the paper starts from Section 4, where we present an approach for modeling a notion of service in a clients-server configuration; subsequently is also defined a notion of cost for a given client for getting the service done; the synchronization operator '$*$' gives a natural and functional setup for developing these notions. A notion of quality of service is defined, for a given client in a given clients-server configuration, where the various server agents are assumed to operate on a shared memory basis. These aspects are all presented in an algebraic setup where every atomic event in the service is assigned a symbolic cost, and the set of all such costs forms a semi-ring. Such a view in particular allows us also to compute the service costs as the powers of a suitable matrix over the semi-ring. We then outline the application of classical matrix algorithms to the efficient computation of service costs, thanks to the semi-ring setup, and estimate the practical gain in these algorithms that is due to our synchronous composition operator. In a concluding section we give some indications on how to extend this cost calculus to the case where the server agents operate on a multiprocessor.

2. A Calculus for Finite State Processes

Our processes are constructed over a given set `EAct` of (extended) action symbols. Let `EAct` $= \{\theta\} \biguplus \{a, \bar{a}, b, \bar{b}, \ldots\}$; here θ is a special 'internal' action similar to the τ of CCS, but in our calculus it will also be used to symbolize an 'idling' action of any process. `Act` will denote the subset of non-θ actions of `EAct`, referred to occasionally as *standard* actions; bars serve in pairing out synchronizing (or communicating) actions, in particular it is assumed that $\bar{\bar{x}} = x$; we refer to $\bar{x}$ as the conjugate of x. The special action θ is assumed self-conjugate. In our calculus, the synchronization between actions is assumed defined via a (partial) binary operation denoted as '$*$', and defined as follows:

$$(\texttt{STAR}) \quad x{*}y = \begin{cases} \theta & \text{if } x = \bar{y} \\ x & \text{if } y = \theta;\ \text{or } y \text{ if } x = \theta \\ \texttt{undef} & \text{or } \bot,\ \text{otherwise} \end{cases}$$

2.1. Finite Recursion Free Processes

We first consider finite non-recursive processes; the grammar for generating them is as follows:

$$\texttt{Proc} ::= \mathbf{0} \mid \texttt{EAct}.\texttt{Proc} \mid \texttt{Proc} + \texttt{Proc} \mid \texttt{Proc} * \texttt{Proc} \mid \texttt{Proc} \setminus \texttt{Act}.$$

Its operations are respectively called null process, prefix, (non-deterministic) choice, (parallel) synchronous composition, and restriction. Process $x.\mathbf{0}$ will be abbreviated to x and $x.(y.\mathbf{0})$ to $x.y$ when no confusion is likely between processes and their traces. In general $x, y, \ldots$ will stand for actions, and $P, Q, R, \ldots$ for processes. Any *finite* process P can (and will) be seen as a finite term over the signature formed by '$+, *$', and the set `EAct` of action prefixes. The operational semantics of our processes is a labeled transition system, defined by the following inference rules:

$$(\texttt{Prefix}) \quad \frac{}{x.P \xrightarrow{x} P}$$

$$(\texttt{Sum}) \quad \frac{P \xrightarrow{x} P'}{P + Q \xrightarrow{x} P'} \qquad \frac{Q \xrightarrow{x} Q'}{P + Q \xrightarrow{x} Q'}$$

$$(\texttt{Sync}) \quad \frac{P \xrightarrow{x} P' \quad Q \xrightarrow{y} Q' \quad x * y = z \neq 0}{P * Q \xrightarrow{z} P' * Q'}$$

$$(\texttt{Restr}) \quad \frac{P \xrightarrow{x} P', \quad H \subseteq \texttt{Act}, \quad x \notin H}{P \setminus H \xrightarrow{x} P' \setminus H}$$

Remark 1. i) Rule (`Sync`) is symmetric in $x, \bar{x}$ because we are assuming $\bar{\bar{x}} = x$.

ii) If P, Q are processes and $\alpha = a_1 a_2 \ldots a_p$ is any string of actions, then the notation $P \xrightarrow{\alpha} Q$ will mean that there exist processes $P_i, 0 \leq i \leq p$ such that $P_0 = P, P_p = Q$, and for any $i \in \{0, p-1\}$ we have $P_i \xrightarrow{a_{i+1}} P_{i+1}$; the $P_i, 0 \leq i \leq p$ will be said to be sub-processes of $P_0 = P$; and Q is said to be an α-successor of P.

Bisimulation: Simulation, simulation equivalence and bisimulation between processes are defined as usual: Let P, Q be two processes. A binary relation $\mathcal{S}$ from the set of sub-processes of P into the P set of sub-processes of Q is said to be a *simulation* of P by Q, iff the following holds: $(P, Q) \in \mathcal{S}$ and for every $a \in$ `EAct` such that $P \xrightarrow{a} P'$, there exists a sub-process Q' of Q such that $Q \xrightarrow{a} Q'$ and $(P', Q') \in \mathcal{S}$. Notation: $P \trianglelefteq_{\mathcal{S}} Q$, or more simply $P \trianglelefteq Q$ if the relation $\mathcal{S}$ can be left implicit. If we have $P \trianglelefteq Q$ and $Q \trianglelefteq P$, then P, Q are said to be simulation equivalent.

A relation $\mathcal{R}$ which is a simulation of P by Q is said to be a *bisimulation* between P and Q iff its opposite relation $\mathcal{R}^{-1}$ defines a simulation of Q by P. If P is bisimilar to Q under $\mathcal{R}$ we shall write $P \sim_{\mathcal{R}} Q$, or more simply $P \sim Q$ if the relation $\mathcal{R}$ can be left implicit. Bisimulation satisfies the usual 'congruence' relations on processes; in particular, it is classical that the set of all processes (without any finiteness constraint) satisfies the following equational axioms (referred to as ACUI up to bisimulation:

$$\text{(A)} \quad X + (Y + Z) \doteq (X + Y) + Z$$
$$\text{(C)} \quad X + Y \doteq Y + X, \qquad \text{(U)} \quad X + \mathbf{0} \doteq X.$$
$$\text{(I)} \quad X + X \doteq X.$$

2.2. Recursive Finite State Processes

The usual definitions of finite state recursive processes, e.g. as given in [7], extend a grammar for finite processes (similar to the one of our previous section, except that '$*$' is traditionally replaced by the '$|$' of asynchronous parallel composition); this extension resorts to operators such as "*def*" or "*Rec*", which make use of the so-called 'process constants' and defining equalities between them; transitions are defined via operational semantics as above, plus some additional conditions of 'non-empty guards'. For instance: if X, Y, Z are given process constants, then the following equations: $X = a.Y,\ Y = Z,\ Z = b.X$ – where $a, b, c, ..$ are action symbols – define a recursive process where X 'can go back' to X, after first performing a, then b; the 'guard' for this composite loop from X to X is by definition the word $a.b$ over the set of action symbols. But a set of equalities like $X = a.Y + Z,\ Z = X,\ Y = c.\mathbf{0}$ is not allowed since the guard for going back from X to X would then be the empty word ϵ (the empty action symbol).

It is clear that these definitions are based on viewing processes as state machines; so we shall henceforth use the word *node* or *state*, instead of 'process constant'; in the following, nodes will be denoted by capital letters such as $U, V, X, Y, Z, \ldots$, with or without suffixes. As previously, `EAct` will denote a given finite set of action symbols (including θ), set run over by the symbols $a, b, c, ..., x, y, z., ,$

Definition 1 *i) A finite state process P is a pair $(\mathcal{E}, X)$ where $\mathcal{E}$ is a finite set of finite equalities of the form: $Z_i = \Sigma_j a_{ij} X_j$, where the Z_i's , X_j's are nodes, and:*
(i) distinct equalities have distinct left-hand-sides (lhs)*;*
(ii) the a_{ij}'s, are elements of `EAct` $\cup\, \epsilon$*;*
(iii) X is a node appearing as the lhs of an equality in $\mathcal{E}$, referred to as the starting state *of P.*

The '$+$' of these equalities is assumed to satisfy the ACUI-axioms of Section 2.1; as usual **0** will denote the additive unit. (It is obvious that any finite processes as defined earlier, in Section 2.1, can be translated into the format of defining equalities.) On a process P defined as above, the a_{ij}'s are called the *transitions of* P; we shall say more precisely there is an a_{ij}-transition from Z_i to X_j. (As usual, ϵ stands for an empty label on the transition between the nodes concerned.) Given any two states U and V on a process P and a word $\alpha \in (\texttt{EAct} \cup \epsilon)^*$, the notion of an α-path from state U to state V on P is defined as in Section 2.1.

A process P is said to be *recursive* iff there is a path on P from some state of P to itself. P is said to be *guarded* iff there is no ϵ-path on P from any state to itself; we shall assume that every recursive process is guarded.

Given two processes $P_i, i = 1, 2$ with $X_i, i = 1, 2$ as their respective starting states, their sum $P_1 + P_2$ is defined as the process obtained by adding to the union of their defining equality-sets, an additional equality $X = X_1 + X_2$, where X is assumed not present in P_1, P_2. Their '$*$'-product $P_1 * P_2$ is defined as the process whose states are all of the form $U * V$ where U (resp. V) is a state on P_1 (resp. P_2), and with $X_1 * X_2$ as starting state; the transitions of $P_1 * P_2$ are defined by using the distributivity of $*$ over $+$, and the table (`STAR`) of Section 2 defining '$*$' on actions, extended in an obvious manner to cover the case of ϵ. (When such a product gets constructed in practice, its nodes may be renamed for notational convenience.)

Asynchronous Parallel Composition (the 'Interleaving Semantics'): Although not formally part of our setup, the notion of asynchronous parallel composition between processes – for which we shall use the classical notation '$|$' - can be defined uniquely up to bisimulation. Consider first recursion-free processes. Let $P = \sum_{a_i} a_i.P_i$ and $Q = \sum_{b_j} b_j.Q_j$ be additive normal forms, respectively for P, Q. Then $P \mid Q$ is defined uniquely up to bisimulation, as:

$$P \mid Q \;=\; \sum_{a_i} a_i.(P_i \mid Q) \;+\; \sum_{b_j} b_j.(P \mid Q_j) \\ \qquad + \sum_{a_i \neq \theta \neq b_j} (a_i * b_j).(P_i \mid Q_j)$$

where the binary '$*$' is defined as above. The last branch concerns only synchronization between standard non-θ actions, so it is straightforward that we get the asynchronous composition of P and Q of CCS, once every occurrence of θ in the resulting term is replaced by CCS's τ. As for finite state recursive processes, observe that they are defined via equalities; now the above definition of $P \mid Q$ is also via equalities, so is extended naturally to the case of finite state recursive processes. To be consistent with the point of view developed in this paper, we shall adopt a slightly different definition for '$|$', however: the last summand of the above definition will be taken over *all* pairs of actions of P and Q, including θ.

Parallel asynchronous composition of any finite family of processes $P_1, \ldots, P_r$ is defined similarly, by treating the processes successively in pairs; it is uniquely determined up to bisimulation, and is associative-commutative on $P_1, \ldots, P_r$; we shall employ the standard CCS notation $P_1 \mid \ldots \mid P_r$ for this parallel asynchronous composition.

Note: Synchronous composition is non-associative in general; i.e., in general $P * (Q * R) \not\sim (P * Q) * R$; for instance, $((a.\mathbf{0}) * (\bar{a}.\mathbf{0})) * (c.\mathbf{0})$ can do action c, but $(a.\mathbf{0}) * ((\bar{a}.\mathbf{0}) * (c.\mathbf{0}))$ can do no action.

Remarks 2. On the set of finite state (possibly recursive) processes, the notions of simulation, simulation equivalence and bisimulation are defined exactly as in Section 2.1, keeping in mind now that the sub-processes of a process P correspond to its nodes. Besides satisfying ACUI, bisimulation has also the following additional properties:

(i) For any P, we have an *additive normal form*:

$$P \sim \sum_{P \xrightarrow{a} P'} a.P'.$$

The proof is classical, needs only that processes are defined via equalities.

(ii) Prefix and synchronous composition 'commute':

$$a.P * b.Q \sim (a * b).(P * Q).$$

The proof is by showing that the binary relation $\mathcal{R}$ defined as the set of pairs $(a.P * b.Q, (a * b).(P * Q))$ where a, b run over actions and P, Q over processes, is a strong bisimulation. (The next assertion is proved similarly too.)

(iii) Synchronous composition '$*$' is commutative, and distributes over '$+$', up to bisimulation: We have:

$$P * Q \sim Q * P, \qquad P * \mathbf{0} \sim \mathbf{0}.$$
$$P * (Q + R) \sim P * Q + P * R.$$

In other words, bisimulation satisfies the equational theory PACUID, that one obtains by augmenting ACUI with the following additional axioms:

$$\text{(D)} \quad X * (Y + Z) \doteq X * Y + X * Z$$
$$\text{(Z)} \quad X * \mathbf{0} \doteq \mathbf{0}, \quad \text{(P)} \quad a.X * b.Y \doteq (a * b).(X * Y)$$

Bisimulation is PACUID-Congruence, for *finite* processes: Let $\equiv$ be the smallest congruence on the set `Proc` of all *finite* non-recursive process terms defined by $P \equiv Q$ iff $P = Q$ or P can be obtained from Q by applying one or more of the PACUID-equational axioms. Then, for any two *finite* processes P, Q we have: $P \sim Q$ *if and only if* $P \equiv Q$. The proof is by putting together the following two propositions.

Proposition 1 *$P \equiv Q$ implies $P \sim Q$*

Proof: Suppose $R \equiv R'$ be an instance of any one of the axioms. Then from the properties mentioned in Remarks 2, we have $R \sim R'$, and since bisimulation is a congruence for process terms, we also have $P[R] \sim P[R']$. Now any equational proof of $P \equiv Q$ is a finite sequence of transformations via such instances. All of them preserve bisimulation, so by the transitivity of bisimulation, $P \sim Q$. $\square$

For the reverse implication, we need a lemma.

Lemma 1 *For finite P, $P \sim \mathbf{0}$ implies $P \equiv \mathbf{0}$.*

Proof: Since P is bisimilar to $\mathbf{0}$, we get $\texttt{EAct}(P) = \emptyset$. We then reason by induction, on the size of the finite process P, since any P is bisimilar to its additive normal form.

If $P = \mathbf{0}$ there is nothing to prove. Note that P may not be of the form $a.P'$ because $\texttt{EAct}(P) = \emptyset$.

If $P = P' + P''$ then we must have $\texttt{EAct}(P') = \texttt{EAct}(P'') = \emptyset$; but then, since P', P'' are smaller terms than P, we have by induction hypothesis $P' \equiv \mathbf{0}$, $P'' \equiv \mathbf{0}$, so $P = P' + P'' \equiv \mathbf{0}$.

If $P = P' * P''$, let the additive normal form for P' and P'' be respectively $\sum_i a_i.P'_i$, $\sum_j b_j.P''_j$. By induction hypothesis, P' and P'' are also equationally equivalent to those sums and distributivity implies $P = P' * P'' \equiv \sum_{i,j}(a_i.P'_i) * (b_j.P''_j)$. Now every product $a_i * b_j$ must be undefined, otherwise the rule for transitions of a product would have implied a transition for $P = P' * P'$. Hence the axioms imply that $\forall i, j,\ (a_i.P'_i) * (b_j.P''_j) \equiv \mathbf{0}$, and thus $P = P' * P'' \equiv \sum_{i,j} \mathbf{0} \equiv \mathbf{0}$.

If $P = P' \setminus a$ then either $P' \sim \mathbf{0}$, or not. In the first case, by induction we have $P' \equiv \mathbf{0}$, so $P \equiv (\mathbf{0} \setminus a) \equiv \mathbf{0}$. In the other case, suppose there exists a $b \in \texttt{EAct}(P')$; since $P' \setminus a$ has no possible action, b must be a and hence $P' \sim \sum_i a.P'_i$. Since P' is smaller than P, by induction hypothesis we get $P' \equiv \sum_i a.P'_i$. So, $P = P' \setminus a \equiv (\sum_i a.P'_i) \setminus a \equiv \sum_i (a.P'_i \setminus a) \equiv \sum_i \mathbf{0} \equiv \mathbf{0}$. $\square$

Proposition 2 *For finite P, Q, $P \sim Q$ implies $P \equiv Q$.*

Proof: If $P \sim \mathbf{0}$ then this follows by the previous lemma. So we assume that $\texttt{EAct}(P)$ and $\texttt{EAct}(Q)$ are non-empty. Now from the assumption we get $\texttt{EAct}(P) = \texttt{EAct}(Q)$; let A denote this common set of actions. Then we know that $P \sim \sum_{a \in A} P_a$ and $Q \sim \sum_{a \in A} Q_a$. Bisimulation implies that for every $a \in A$ and a P_a we have a Q_a such that $P_a \sim Q_a$; by induction hypothesis, the terms P_a, Q_a being smaller than the terms P, Q respectively, we get then $P_a \equiv Q_a$. One deduces then, by commutativity and associativity of $+$, that $P \equiv Q$. $\square$

3. Deciding Strong Bisimulation

For any process P and X' node on P, we define $Act_P(X')$ as the set of all $a \in \texttt{EAct}$ such that X' has an a-successor on P.

Suppose given two finite state processes P, Q, with respective initial states X_0, Y_0. We give here an algorithm for deciding that $P \sim Q$, based on classical reasonings e.g. as given in [7]. We shall be denoting the 'generic' nodes on P (resp. on Q) by $X', X'', ...$ (resp. by $Y', Y'', ...$) with or without suffixes. And **InEqu** will denote a set of 'inequivalences' (or inequalities), of the form $X' \neq Y'$; it is assumed that **InEqu** $= \emptyset$ at the start.

```
Step i)  For every pair $(X', Y')$,
    add $X' \neq Y'$ to **InEqu** if  $Act_P(X') \neq Act_Q(Y')$.
Step ii)  If **InEqu** $= \emptyset$,  then return "$P$ bisimilar to $Q$",
    else  set $Pairs = \{(X', Y') \mid X' \neq Y' \notin$ **InEqu**$\}$.
Step iii)  Choose an $(X', Y') \in Pairs$;
  - let $a \in \texttt{EAct}$ such that $X' \xrightarrow{a} X''$; if for every $Y''$
    with $Y' \xrightarrow{a} Y''$ we have $X'' \neq Y'' \in$ **InEqu**, then
    add $X' \neq Y'$ to **InEqu**;
  - let $b \in \texttt{EAct}$ such that $Y' \xrightarrow{b} Y''$; if for every $X''$
    with $X' \xrightarrow{b} X''$ we have $X'' \neq Y'' \in$ **InEqu**, then
    add $X' \neq Y'$ to **InEqu**;
  - set $Pairs := Pairs \setminus \{(X', Y')\}$.
Step iv)  If $Pairs \neq \emptyset$, then GOTO Step iii).
Step v)  If $X_0 \neq Y_0 \in$ **InEqu**,
    then return "$P$ not bisimilar to $Q$",
    else return "$P$ is bisimilar to $Q$".
```

Note: This simple backward reasoning algorithm suffices for our purposes here: namely its use in the proof of Proposition 3 below. (Several optimizations are possible for lowering appreciably its complexity; see e.g., [6].)

3.1. Process Equivalences via '$*$'

We give here a characterization for strong bisimulation based on '$*$'; the idea is straightforward, and shows the usefulness of our synchronization operator $*$ and the role played by the action symbol θ.

Proposition 3 *Given processes P, Q, $P \sim Q$ holds if and only if for all finite* linear processes J (meaning: J has no 'choice', i.e. J is any finite sequence of actions), *we have: $P * J \sim Q * J$.*

Proof: To prove the "only if" assertion, we check that given any J the set $\mathcal{B}$ of pairs $\{(X' * U', Y' * U') \mid X' \sim Y'\}$, where U' is any state on J, and X', Y' are respectively states on P and Q, is a strong bisimulation. Now, by definition, the possible transitions from $X' * U'$ must be of the form: $X' * U' \xrightarrow{a*b} X'' * U''$ where $X' \xrightarrow{a} X''$ on

P and $U' \xrightarrow{b} U''$ on J; but then there must be a transition $Y' \xrightarrow{a} Y''$ on Q with $X'' \sim Y''$; so there is a transition $Y' * U' \xrightarrow{a*b} Y'' * U''$, and the pair $(X'' * U'', Y'' * U'')$ is in the set $\mathcal{B}$.

As for the "if" assertion, assume that P and Q are not bisimilar. Then, the set **InEqu** as defined in the algorithm of Section 2.3 is non-empty when the algorithm halts, and contains the inequality $X_0, \neq Y_0$ formed of the respective starting nodes of P and Q. Then there exists, by definition a *shortest* sequence $\alpha \in \texttt{EAct}^*$ of actions satisfying the following condition:

$$X_0 \xrightarrow{\alpha} X' \text{ on } P, \text{ and for } \textbf{any } Y_0 \xrightarrow{\alpha} Y' \text{ on } Q,$$
$$\text{we have} \quad Act_P(X') \neq Act_Q(Y'),$$

(and/or a similar condition with the roles of P, Q reversed). If $\alpha = a_1.a_2 \ldots a_m$, we define a finite linear process J as $J = \bar{a}_1.\bar{a}_2 \ldots \bar{a}_m.\theta$. We then have $P * J \not\sim Q * J$. □

Remark 3. The above Proposition together with Proposition 2, suggests another way for deciding bisimulation, by looking at the problem in its negated form; i.e., deciding non-bisimulation between P and Q. For doing this we look for a linear $J = x_1.x_2 \ldots x_N.\mathbf{0}$ (where N is the maximum number of transitions on P or Q, and) the $x_i, i = 1..N$ are 'action variables' to be solved for, such that $P * J \not\sim Q * J$; since J is assumed finite, solving for such a J amounts to solving a special and weak case of disunification problem over the PACUID-equational theory.

For any action symbol a, classically $\overset{a}{\Longrightarrow}$ stands for the *weak transition* relation. defined as $(\xrightarrow{\theta})^* \xrightarrow{a} (\xrightarrow{\theta})^*$; and $\approx$ denotes the notion of weak bisimulation between processes, defined w.r.t. these weak transition relations. The operator '$|$' of asynchronous parallel composition can be expressed in a very concise manner in terms of '$*$', up to weak bisimulation, as follows. To every process P (defined by a set of guarded equalities) associate a process denoted as $\hat{P}$, obtained by adding a θ-loop at every node. E.g., if P is defined by $X = a.X + b.Y$, $Y = \mathbf{0}$, then $\hat{P}$ is defined by $X = a.X + b.Y + \theta.X$, $Y = \theta.Y + \mathbf{0}$. It is clear that $\hat{P}$ is weak-bisimilar to P for any P (intuitively, $\hat{P}$ is just P except that it may 'idle' before doing any action). The process $\hat{P}$ thus constructed will be referred to as the '*hatted extension*' of P. We then have the following result.

Proposition 4 $P \mid Q \ \approx \hat{P} * \hat{Q}$.

Proof: This is done by showing that the following relation $\mathcal{R}$ – where P, Q run over the set of all processes – is a weak bisimulation:

$$\mathcal{R} \ = \ \{(P \mid Q \ , \ \hat{P} * \hat{Q})\}.$$

We first show that whenever $P \mid Q \xrightarrow{a} A$, there exists $\hat{P} * \hat{Q} \xrightarrow{a} B$, therefore $\hat{P} * \hat{Q} \overset{a}{\Longrightarrow} B$ too, such that $(A, B) \in \mathcal{R}$; thus actually $\hat{P} * \hat{Q}$ simulates $P \mid Q$. The proof is by case analysis on $P \mid Q \xrightarrow{a} A$.

1. $A = (P' \mid Q)$ and $P \xrightarrow{a} P'$: obviously we have here $\hat{P} \xrightarrow{a} \hat{P}'$ and $\hat{Q} \xrightarrow{\theta} \hat{Q}$, and then $\hat{P} * \hat{Q} \xrightarrow{a} \hat{P}' * \hat{Q} = B$, therefore $(A, B) \in \mathcal{R}$.
2. $A = (P \mid Q')$ and $Q \xrightarrow{a} Q'$: similar to case 1.
3. $A = (P' \mid Q')$ and $P \xrightarrow{b} P'$, $Q \xrightarrow{\bar{b}} Q'$ with $a = \theta$: obviously then $\hat{P} \xrightarrow{b} \hat{P}'$, and $\hat{Q} \xrightarrow{\bar{b}} \hat{Q}'$, so $\hat{P} * \hat{Q} \xrightarrow{\theta} \hat{P}' * \hat{Q}' = B$. Therefore $(A, B) \in \mathcal{R}$.

We show next that the opposite relation of $\mathcal{R}$ is a weak simulation; that is to say, $P \mid Q$ weakly simulates $\hat{P} * \hat{Q}$: for this consider any non-θ transition c from $\hat{P} * \hat{Q}$; by definition c must be such that $c = a * b$ with $\hat{P} \xrightarrow{a} P'$ and $\hat{Q} \xrightarrow{b} Q'$, with one of a, b being a θ, but not the other; say $a \neq \theta, b = \theta$, so $c = a$; but then we will have, $P \xrightarrow{a} P'$, therefore $P \mid Q \xrightarrow{a} P' \mid Q \xrightarrow{\theta} P' \mid Q'$, and we deduce: $(P' \mid Q', \hat{P}' * \hat{Q}') \in \mathcal{R}$. □

Corollary 1 *It follows that '$*$' is associative-commutative on the class of 'hatted' processes (those which can 'idle' as and when needed).*

A Notion of Timeout: The 'hatted' processes as they are defined above can idle indefinitely, in particular can indefinitely delay the choice between two branches. In practical situations however, it is often useful or even necessary to bound such a delay. This leads to the following *timeout* notion. Given two processes P, Q, for any positive integer n, define inductively a process as follows:

$Timout(P, 0, Q) = P + Q$,
$Timout(P, n, Q) = P + \theta.Timout(P, n-1, Q)$.

Intuitively: $Timout(P, n, Q)$ is the process which can do the (idling) θ-action at most n times before choosing between P and Q.

Such a notion is easily defined also for finite state processes defined via equalities: Given such a process P and an integer $n \geq 0$, we shall denote by $\hat{P}_{(n)}$ the finite state process, which at any of its nodes can do the idling θ-action at most n times before being forced to branch off to a successor node. For instance, if $X = E$ is any one of the defining equalities of P and $n = 1$, then the corresponding defining equality for $\hat{P}_{(1)}$ will be $X = E + \theta.E$. The timeout notion can actually be seen as a 'timed' notion, part of a timed process calculus extending ours. We shall be using the timeouts only in a limited manner in the following section, for modeling notions of services and their costs.

4. Modeling Services

Our concern in this section will not be on the formal side, but rather will have an engineering flavor. We propose to show how our calculus - with its obvious bias for the synchronous branches - is handy for modeling notions such as service, and quality or denial of service, in a clients-server configuration. Services rendered by a server to a client can certainly be modeled as processes in various ways; but it is most natural to specify them informally as protocols of the following form:

- The client sends out a nonce (identity, date,...) to the server, which on reception sends back a session key; a sequence of messages then get exchanged, and the session ends when the client 'exits'.

The issues of security concerning the messages will not be our concern here (they will be studied elsewhere). We are only concerned with the issue of modeling the notion of service in an appropriate way, that will allow us to propose a model for defining a notion of cost for the client to get the service done. The modeling we present is best understood in the setup of value-passing processes, but it is not difficult to bring out the ideas in the pure (non value-passing) case: Let S denote the server and C the client; then, schematically we can depict them as the following finite state processes:

$$C ::= \bar{n}_C\,.k_C\,.\bar{m}_C\,.e_C.\mathbf{0}$$
$$S ::= n_S\,.\bar{k}_S\,.m_S\,.\theta_S\,.S$$

The phase (or action) $\bar{n}_C$ is where the client C sends out a nonce, k_C is where he gets the session key, $\bar{m}_C$ is for his sending the message requesting the service, and e_C is where he exits. The phases of action of the server S are the respective conjugates of these actions of C, except the final θ_S which symbolizes an idling step of S when registering the exit of C (this, to be in accordance with our synchronous view). The recursive definition of S means S can get back to serve again, possibly some other client.

A service having been *specified* in such a manner, what we want actually is to model a situation where some given number N of server agents will be serving more than one clients. For this, we shall make the following assumptions:

- each server agent is a copy of S as defined above;
- the N copies are in parallel asynchronous composition and use a given operating system on a *shared memory* basis, for the various steps of synchronization with the actions of the various clients;
- each such synchronization step is a 'machine process' of the operating system.

(It is tacitly assumed that the various machine processes get executed by the operating system on a fair basis.)

Service as a Trace : The notions of service, as well as its quality or its denial, will all be defined with respect to some (arbitrarily) given client C. Consider first case where the server S is unique; the service rendered by S to C is defined as a trace, namely the finite prefix of the synchronous product $C * S$ ending up with the exit action e_C of C, and such that there is no other e_C along the trace. In the case of N server agents operating under the above assumptions and several (unspecified number of) clients, we define the server as the process $\mathcal{S}_N = S \mid S \mid \ldots \mid S$, ($N$ times); it is assumed that any action a_S of any of the server agents is conjugate to any action $\bar{a}_{C'}$ of any client C'. We propose then to define the service rendered by $\mathcal{S}_N$ to the given client C, as *any* appropriate trace of a product process $\mathcal{S}_N * \hat{C}_{(B)}$ ending up with the exit action e_C, *and* satisfying some additional requirements, specified in the next subsection. B denotes here and in the pages to come, a given positive integer referred to as the timeout bound (on the client).

The reason for using a timeout extension $\hat{C}_{(B)}$ instead of C itself (cf. end of previous section) is easily explained: after any machine process corresponding to a synchronization step between a server agent and client C, the operating system may take over a machine process concerning some other client; consequently, the sequence of synchronization steps between the given client C and the server $\mathcal{S}_N$, constituting what has been *specified* as service rendered to C, will be interleaved in general with steps of synchronization *not* part of this specified service; operationally such steps can (and will) be viewed as those where the given client C waits or 'idles', before the operating system continues again with steps concerning C. The timeout bound B of $\hat{C}_{(B)}$ is actually a fairness assumption on the client: no client will be allowed to idle indefinitely, since otherwise he could occupy a server agent indefinitely. The clients-server configuration presented this way also shows that it is unnecessary to specify any given number of clients.

Such a vision of service as a trace is in conformity with the usual operational notion. Moreover, in any clients-server configuration a client may get a service done in more than one way, and not all of them will be costing him the same amount. This will be discussed in the next section.

4.1. Representing a Service and its Cost

In the context of process algebras it is useful to associate costs to processes so as to calculate for instance: maximal duration, minimal time before deadlock, maximal space requirement etc. Notions of cost usually relate paths on labeled graphs with path weights. A useful approach is to apply the classical algebra of paths [5] to the labeled graphs of processes. In such a vision, the space of costs is a semi-ring (i.e. an additive monoïd, enriched with a 'product' operation distributing over the 'sum'), and one associates an a priori cost to every process *event*. The desired 'global cost' is then computed for each problem considered, as the 'sum' in the semi-ring of costs of the various traces (this 'sum' being union, or maximum, or minimum, or disjunction.. depending on the problem); the cost of a trace is calculated as the semi-ring product (resp. intersection, or sum, or maximum, or conjunction, ...) of the costs of its individual events.

Such a semi-ring vision will be followed here only in a limited manner, to propose a notion of cost for a service rendered by the server $\mathcal{S}_N$ to a given client C. To every 'event' in a service, which is by definition a machine process, i.e. a synchronization step, we first associate a *symbolic cost*. We shall be doing this by annotating our θ's suitably, as per the following rules (where the symbol "_" will stand for some unknown client, *other than* C):

(i) To any non-idling action $a \neq \theta$ of the server $\mathcal{S}_N$, is attached a fixed positive real number t_a, referred to as its 'weight' (interpretable if desired as the time spent by operating system for a synchronizing machine process step in a service). Any two conjugate actions are assumed to have the same weight.

(ii) For a (non-idling) action $a \neq \theta$ of the server $\mathcal{S}_N$, if the conjugate action $\bar{a}$ is performed by C, then their synchronous product θ is annotated with the pair (t_a, C); the symbolic cost of this event in the service is defined as the annotated symbol $\theta_{(t_a,C)}$.

(iii) For a (non-idling) action $a \neq \theta$ of the server $\mathcal{S}_N$, synchronizing with a θ-action of C (i.e., when the conjugate action $\bar{a}$ is *supposedly* from some client other than C – so the event visible in the trace will be a), the symbolic cost of the event is defined as $\theta_{(t_a,\_)}$.

(iv) For the event where the idling action θ_S of the server $\mathcal{S}_N$ synchronizes with a non-exit action a of C, the symbolic cost of the event in the trace is defined as $\theta_{(t_a,\_)}$.

(v) For the event where the idling action θ_S of the server $\mathcal{S}_N$ synchronizes with the exit action of some client, the symbolic cost of the event is defined as the special symbol $\mathbf{1}$, with an empty annotation.

In other words, the symbolic cost of the atomic events on $\mathcal{S}_N * \hat{C}_{(B)}$ is defined via a *cost observation operator* $\mathcal{H}$ such that, along the traces of $\mathcal{S}_N * \hat{C}_{(B)}$, each atomic event is seen as its symbolic cost. The process $\mathcal{S}_N * \hat{C}_{(B)}$ observed under $\mathcal{H}$ will be denoted as $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$; the set of all atomic events of this process is then obviously a semi-ring under the following operations: the 'product' is the concatenation '.' of the symbols, and the 'sum' is the choice '$+$' between the branches. (Since we are considering traces now, the following equation is also assumed: $x.(X+Y) = x.X + x.Y$.)

Symbolic Representation of a Service: We assume that the service has been specified schematically as above via the protocol processes. The traces of the process $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$ which correspond to a service rendered by $\mathcal{S}$ to C, in some manner, can be defined as follows. Let `SCost` be the alphabet formed of the symbolic costs associated to the various atomic events of $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$; for notational convenience, we shall drop the suffixes $S, C, \_$, of the action symbols in the annotations of the θ's.

Definition 2 *Let* $\Sigma = \texttt{SCost} \setminus \{\mathbf{1}\}$, *be the set of elements* $\neq \mathbf{1}$ *in* `SCost`. *Then a trace of* $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$ *represents a service rendered by* $\mathcal{S}$ *to* C, if and only if *it is an element of the following language:* $\Sigma^*.\,\theta_{(t_n,C)}.\,\Sigma^*.\,\theta_{(t_k,C)}.\,\Sigma^*.\,\theta_{(t_m,C)}.\,\Sigma^*.\,\mathbf{1}$.

Any such trace will be referred to as a service trace. *A* wasted trace *is a trace of* $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$ *which is not a service trace. The process* $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$ *itself will be referred to as the* symbolic model *of the clients-server configuration (for cost analysis).*

In intuitive terms: a trace of $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$ is a symbolic representation of a service rendered by $\mathcal{S}$ to C iff it ends up with a $\mathbf{1}$ with no earlier occurrence of $\mathbf{1}$, and all the actions constituting the service have been accomplished, possibly interspersed with some idling actions. (Note: there may be traces ending up with a $\mathbf{1}$ and with no earlier occurrence of $\mathbf{1}$, but which may not be a service trace.)

Costs of a Service : The definition of cost for a service rendered to a *given* client C, is based on a given non-negative real denoted as x_C, referred to as the *billing coefficient* for C, and a given random function $f(\_)$ generating non-negative reals, meant as the billing coefficients for the unknown clients other than C.

Definition 3 *To every atomic event of* $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$, *we associate a non-negative real number, calculated as:*

- $t_a \times x_C$ if the event is of the form $\theta_{(t_a,C)}$;
- $t_a \times f(\_)$ if the event is of the form $\theta_{(t_a,\_)}$.

i) The total cost *of any trace of* $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$ *is defined as the sum of all the real numbers associated to the atomic events composing the trace.*

ii) The total functioning cost *of the clients-server configuration is defined as the sum of all the total costs of all the* maximal *traces of* $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$.

iii) The total service cost *of the configuration is defined as the sum of the total costs of all the service traces.*

These notions are all well-defined: indeed, the process $(\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$ is necessarily finite since $\hat{C}_{(B)}$ is finite.We define the *Quality of Service (QoS)* of the clients-server configuration with respect to the given client C, as the non-negative real number which is the ratio (*total service cost*)/(*total functioning cost*). Note that it depends on the number N of server agents rendering the service, as well as the timeout bound B for the client C.

Example 1. For better readability, we shall shorten the service protocol as follows:

$S \ ::= \ n_S.\bar{k}_S.\theta_S.S$

$C \ ::= \ \bar{n}_C.k_C.e_C.\mathbf{0}$

and define $\mathcal{S}_2 = S \mid S$, i.e., there are two server agents functioning in parallel. We also assume that the client C has a timeout bound of 2. Then, with the above notation, the process $(\mathcal{S}_2 * \hat{C}_{(2)})/\mathcal{H}$ has many traces, among which the following two, ending with $\mathbf{1}$, represent branches where the service for client C gets done:

$\theta_{(t_n,C)}.\theta_{(t_n,\_)}.\theta_{(t_k,C)}.\mathbf{1}$

$\theta_{(t_n,C)}.\theta_{(t_k,C)}.\theta_{(t_n,\_)}.\theta_{(t_k,\_)}.\mathbf{1}$

On the other hand the following trace represents a branch where C cannot get the service done (due to other clients' competition): $\theta_{(t_n,\_)}.\theta_{(t_n,\_)}$. The reason why this trace has only length 2 is due to the timeout bound for C: $\hat{C}_{(2)}$ can idle at most twice before executing $\bar{n}_C$; during two such idling steps, both server agents get busy with serving other clients, and the only possible actions of $\mathcal{S}_2$ ready to synchronize are their respective actions $\bar{k}_S$. But $\bar{k}_S$ cannot synchronize with $\bar{n}_C$, so after these two idling steps of C the system cannot proceed on; we therefore get a branch with the wasted trace $\theta_{(t_n,\_)}.\theta_{(t_n,\_)}$.

If the client has a timeout bound of 4 and $N = 4$, we get the configuration $(\mathcal{S}_4 * \hat{C}_{(4)})/\mathcal{H}$; here we have an example of a branch with trace: $\theta_{(t_n,\_)}.\theta_{(t_n,\_)}.\theta_{(t_k,\_)}.\mathbf{1}$, ending up with $\mathbf{1}$, containing no earlier occurrence of $\mathbf{1}$, but this is a wasted trace. $\Box$

4.2. Service Costs as Matrix Powers

The application of semi-ring techniques is actually generic and is readily adapted to any other cost model by simply changing the semi-ring algebra: redefine the 'sum' and 'product' operators suitably. This allows us to show how the classical algorithms can be applied to the computation of service costs for large-scale processes in our model.

To compute the QoS value (definition 3 above) one needs to compute the set of all traces of $(\mathcal{S}_{(N)} * \hat{C}_{(B)})/\mathcal{H}$ and the set of service traces for the same process, and then the ratio of the sum of costs over those sets. Now we already observed that the process $(\mathcal{S}_{(N)} * \hat{C}_{(B)})$ is finite. Moreover, it admits an additive normal form defined via guarded equalities; actually we have an easy, more general, result:

Lemma 2 *For any finite state (possibly recursive) process P and any linear process $I = \beta_0.\beta_1.\ldots.\beta_N.\mathbf{0}$, $P * I$ is a finite process, defined by equalities explicitly constructible from those of P.*

Proof: Write $I_i = \beta_i.I_{i+1}, i = 0..N$, with $I_0 = I, I_{N+1} = \mathbf{0}$. Now the defining equalities of the process P are of the form $X_i = \sum_j x_{ij}.X_j$, where one of the lhs 'variables' is the starting state of P. So, $P * I = I * P$ can be represented by a finite term using induction on the size of I:

- For any X, $\mathbf{0} * X = \mathbf{0}$ is a finite term;
- If $X_i = \sum_j x_{ij}.X_j$, and if $I_{i+1} * X_j$ is a finite term for any j, then

$I_i * X_i = \sum_j (\beta_i * x_{ij}).(I_{i+1} * X_j)$ is of course also a finite term. $\Box$

The state machine of the finite process $P = (\mathcal{S} * \hat{C}_{(B)})/\mathcal{H}$ modeling our clients-server configuration can therefore be seen as a directed acyclic graph (dag), its edges labeled with cost symbols from `SCost`. On the other hand, let SC be a regular automaton (or state machine) over the alphabet `SCost`, which recognizes the service language (i.e., the set of service traces) defined in Definition 2. Such an automaton may have loops, and that is not suitable for the matrix calculation that we are going to present; for that we need a directed acyclic graph, which as an automaton recognizes the service language.

To achieve this we first define the 'synchronous product $*$' on the alphabet `SCost` as follows: for any $c', c'' \in$ `SCost`, $c' * c''$ is defined if and only $c' = c''$; and $c' * c' = c'$. Then the '$*$'-product automaton (or state machine) $P' = ((\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}) * SC$ that one can define in an obvious manner (by induction from $a.P * b.Q = (a * b).(P * Q)$), is then a dag, recognizing exactly the language of service traces for the client C.

We can now detail the efficient computation of the sum of trace costs for either P (for the total function cost) or P' (for the total service cost), which is sufficient to obtain the QoS value. Without loss of generality we shall consider P. Since P is a directed acyclic graph, it follows that the set of traces between any two nodes can be represented as a regular expression without the Kleene-star, i.e., a regular expression using only actions (from `SCost`), choice and concatenation. Let RE be the set of such regular expressions. Then $(RE, +, ., 0, \epsilon)$ is a semi-ring where '$+$' is union, '$.$' denotes concatenation, 0 the empty language and ϵ the empty trace. Moreover the set of square matrices over this semi-ring, with the usual matrix operations, is also a semi-ring. The rows and columns of such matrices represent the nodes X on the graph P.

Let M be the (square) transition matrix of P defined as follows: $M(X,Y) = a$ if $X \xrightarrow{a} Y$, and 0 otherwise. Define also the identity matrix as $I(X,X) = \epsilon$ and $I(X,Y) = 0$ if $X \neq Y$. Then it is well known that matrix $M^k(X,Y)$ contains a regular expression for all traces (paths) of length k from state X to state Y. Moreover if we define $M^{(k)} = I + M + M^2 + \ldots + M^k$ then $M^{(k)}(X,Y)$ contains a regular expression for all traces of length *at most* k from state X to state Y. Finally, in an idempotent semi-ring as is the case here (In RE, the 'sum' represents set union and is therefore idempotent, and so is a posteriori the pointwise addition of matrices over RE) we have $(I + M)^k = M^{(k)}$ for the simple reason that the Pascal triangle coefficients in the expansion of $(I + M)^k$ are all equal to one, since M+M=M. This last identity allows us to apply successive matrix powers to compute $M^{(k)}$ in $O(\log k)$ matrix products.

Finally we observe that if n is the number of states of P, then $M^{(n)}$ contains regular expressions for all traces between given pairs of states. In particular, if X_P is the initial state then the sum of row $M^{(n)}(X_P, \_)$ is a regular expression for the trace language of P. This expression is thus computed in $O(\log n)$ matrix products, without enumerating the potentially exponential set of traces in P. From this expression, the total functioning cost can be computed in linear time by interpreting RE operations in bottom-up fashion on the syntax.

The same algorithm, polynomial in the number of states, can be applied to compute the total service cost from process P' and hence obtain the QoS value. The advantage of our synchronous process algebra is then made clear: to obtain finite service descriptions, use explicit dag descriptions and then reuse classical matrix algorithms without concern for convergence conditions.

4.3. Examples, Comments on Complexity

Example 2. We study here the client-server configuration $P = (\mathcal{S}_1 * \hat{C}_{(1)})/\mathcal{H}$, where:

$S = n.\bar{k}.\theta.S$,
$\mathcal{S}_1 = S$, i.e., just one server agent,
$C = \bar{n}.k.e.\mathbf{0}$;

The process $\hat{C}_{(1)}$ is client C with a timeout bound of 1, in the sense defined above. To present the symbolic model of the configuration, we first develop the server and the client as state machines, as follows:

$S = n.S',\ \ S' = \bar{k}.S'',\ \ S'' = \theta.S,$
$C_1 = \bar{n}.D_1 + \theta.C_1',\ \ C_1' = \bar{n}.k.e.\mathbf{0},$
$D_1 = k.D_2 + \theta.D_1',\ D_1' = k.e.\mathbf{0},$
$D_2 = e.\mathbf{0} + \theta.e.\mathbf{0}$

The symbolic model of the configuration is given in Figure 1. The QoS evaluates to 0.33, under the assumption that all the real numbers of Definition 3 are set to 1. □

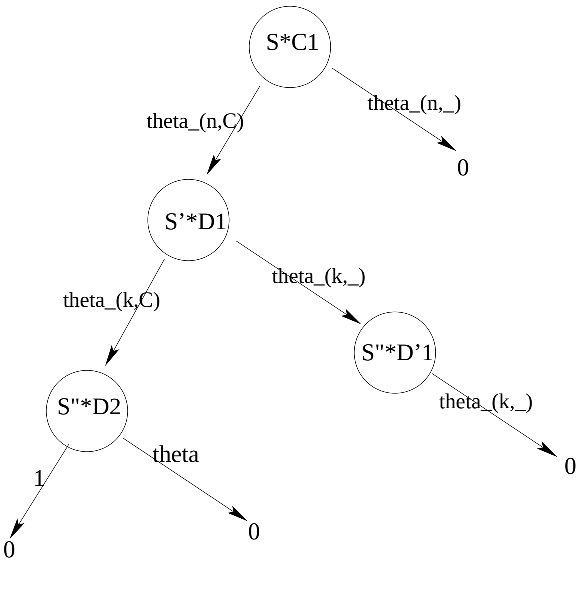

Figure 1. Symbolic Cost Model for Example 2

Example 3. This example illustrates the interest of our cost model above from a complexity viewpoint. We consider now the system $P = (\mathcal{S}_2 * \hat{C}_{(1)})/\mathcal{H}$ where

$S = n.\bar{k}.\theta.S,$
$\mathcal{S}_2 = S \mid S,$
$C = \bar{n}.k.e.\mathbf{0}$

and $\hat{C}_{(1)}$ is process C with a timeout bound of 1, in the sense defined above. Expansion of the asynchronous parallel composition yields 7 states for $\mathcal{S}_2$ which is equivalent to the following equations, with initial state SS:

$$\begin{aligned}
&SS = n.ST\\
&ST = \bar{k}.SU + n.TT\\
&SU = \theta.SS + n.TU + n.ST\\
&TT = \bar{k}.TU\\
&TU = \bar{k}.UU + \theta.ST + \bar{k}.SU\\
&UU = \theta.SU + \theta.SS
\end{aligned}$$

The client process $\hat{C}_{(2)}$ is equivalent to the following acyclic equations, with initial state $C2$:

$$\begin{aligned}
&C2 = \bar{n}.D + \theta.C2'\\
&C2' = \bar{n}.D\\
&D = k.E + \theta.D'\\
&D' = k.E\\
&E = e.\mathbf{0} + \theta.E'\\
&E' = e.\mathbf{0}
\end{aligned}$$

Now the synchronous product $P = (\mathcal{S}_2 * \hat{C}_{(1)})$ is equivalent to the following acyclic equations, with initial state $SS * C2$ and 14 states (12 plus two for $e.\mathbf{0}$ and $n.\mathbf{0}$):

$$\begin{aligned}
&SS * C2 = \theta.(ST * D) + n.(ST * C2')\\
&ST * D = \theta.(SU * E) + \bar{k}.(SU * D') + n.(TT * D')\\
&ST * C2' = \theta.(TT * D)\\
&SU * E = e.\mathbf{0} + \theta.\mathbf{0} + n.e.\mathbf{0} + n.\mathbf{0}\\
&SU * D' = k.n.\mathbf{0}\\
&TT * D' = \theta.(TU * E)\\
&TT * D = \theta.(TU * E) + \bar{k}.(TU * D')\\
&TU * E = \bar{k}.e.\mathbf{0} + \theta.\mathbf{0} + \bar{k}.e.\mathbf{0} + e.\mathbf{0}\\
&TU * D' = \theta.(UU * E) + k.(ST * E) + \theta.(SU * E)\\
&UU * E = e.\mathbf{0} + \theta.e.\mathbf{0} + \theta.\mathbf{0}\\
&ST * E = \bar{k}.e.\mathbf{0} + n.\mathbf{0}\\
&SU * E = e.\mathbf{0} + \theta.\mathbf{0} + n.e.\mathbf{0} + n.\mathbf{0}
\end{aligned}$$

This synchronous product has generated only 14 of the 36 possible states for the product of $\mathcal{S}_2$ and $\hat{C}_{(1)}$. As a result the matrix encoding $(\mathcal{S}_2 * \hat{C}_{(1)})/\mathcal{H}$ will be 14*14, compared to the almost 36*36 matrix that an asynchronous product model would have generated. Since the overall algorithm for calculating costs is $O(n^3) * \log n$ ($\log n$ products of $n * n$ matrices, as explained earlier), the practical gain due to synchronous composition is here estimated to a factor of $(36/14)^3 \sim 17$, even on this small example. □

We have also carried out some further calculations (essentially by hand) in order to study the evolution of the QoS measure when N (the number of servers) and B (the time-out bound for the client) vary. The following table gives these numerical results, obtained again under the assumption that all the real numbers of Definition 3 are set to 1.

N \ B	0	1	2
1	1.0	0.33	0.08
2	1.0	0.71	0.38

These values confirm the following intended features of our model and support its relevance:

i) $B = 0$ forces a server to follow its protocol trace and thus complete the service. In other words, the client accepts no delay and is served in 100% of the time.

ii) QoS increases with the number of servers N, even if some of the service branches get slowed down.

Our current state of work thus confirms the qualitative value of the QoS measure and its correlation with the parameters of the protocol being analyzed. Future work on our model will necessitate an implementation of the polynomial time algorithm for QoS evaluation mentioned at the end of Section 4.2, and some experimental work to confirm the actual numerical values of QoS as a statistics of response time.

5. Conclusion

We have proposed in this work a process calculus focusing principally on the branches of synchronization between the various agents. It is our belief that such a calculus is well suited for analyzing the flow of information between the agents, in particular for the formal analysis of communication protocols. After having shown in a first part of the paper that our calculus is powerful enough on the formal side, we have shown in the second part how it can be used to model formally the cost analysis of communication protocols. We saw in particular that the cost analysis of a clients-server configuration, where the servers are assumed to operate in parallel on a shared memory basis, can be modeled as a process of the form $P = (\mathcal{S}_N * \hat{C}_{(B)})/\mathcal{H}$ where $\mathcal{S}_N = S \mid S \mid \ldots S$ (N times) stands for N servers operating in parallel.

The interleaving semantics of the asynchronous $S \mid S$ parallel composition has for (realistic) consequence that more server actions may slow down some branches of service. But there are systems where multiprocessors are used precisely to improve quality of service in the sense of response time; it is possible to adapt our QoS model to a multiprocessor situation where the multiple server $\mathcal{S}_N$ occupies multiple asynchronous machines, by proceeding as follows. Let us assume to simplify that the number N of servers is equal to the number of available processors. Client C is then only slowed down by interleaved actions which occupy more than N servers: up to N clients may be served simultaneously. An extension of the process algebra with such a vision of cost setup has been defined in [9]; it uses a new syntax $\langle S, S, \ldots, S\rangle$ to denote ”data-parallel” composition i.e. a process whose components S are placed on asynchronous processors. The operational semantics of this operator is similar to $S \mid S \mid \ldots S$ except for the possibility of barrier synchronizations (notion not relevant to this paper) and the labeling of interleaved actions by their processor of origin. The general case with N different from the number of processors can be treated by a mixture of asynchronous and data-parallel composition.

One direction of possible future work - on the practical side - is to adapt our “synchronous algebraic vision” suitably, so as to express the cost calculus of a distributed memory setup, in a manner similar to the one presented above for the shared memory case. A possible application is the parallel implementations of the matrix power algorithm of Section 4.2 for large-scale verification, independently of the actual shared- or distributed- memory represented by the model. A second possible direction for further developments is on the formal side: it consists in formulating the notion of non-interference (cf. e.g., [4]) in our synchronous calculus, and to apply such a vision to the formal analysis of information flow on communicating systems.

References

[1] S. Anantharaman, G. Hains. *A Synchronous Bisimulation Based Approach for Information Flow Analysis*, In Proc. AVOCS03, Southampton (UK), April 2003.

[2] J.A. Bergstra, A. Ponse, S.A. Smolka (Editors) *Handbook of Process Algebra,* Elsevier, Amsterdam, 2001.

[3] J.C. Baeten, C.A. Middelberg. *Process Algebra with Timing,* Springer, Berlin-Heidelberg, 1998.

[4] G. Boudol and I. Castellani, *Non-Interference for Concurrent Programs and Thread Systems.* Theoretical Computer Science, 281(1-2):109–130, June 2002.

[5] M. Gondran, M. Minoux. *Graphes et Algorithmes,* Chapitre 3, “Les Algèbres de Chemins”, Eyrolles, 1985.

[6] P.C. Kanellakis, S.A. Smolka. *CCS expressions, finite state processes and three problems of equivalence.* Information and Computation, 86 (1):43–68, May 1990.

[7] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[8] F. Muller, S.A. Smolka. *On the Computational Complexity of Bisimulation, Redux* In Proc. PCK50, pages 55-59, San Diego, June 2003, ACM.

[9] A. Merlin, G. Hains. *A generic cost model for concurrent and data-parallel meta-computing* In Proc. AVOCS’04, London, September 2004.