Nordic Journal of Computing 1(1994), 346–363.

THE FORK CALCULUS

KLAUS HAVELUND*
Ecole Normale Supérieure, Laboratoire d'Informatique
45 rue d'Ulm, 75005 Paris, France
havelund@dmi.ens.fr

KIM GULDSTRAND LARSEN†
Aalborg University, Institute for Electronic Systems
Fr. Bajersvej 7, 9220 Aalborg, Denmark
kgl@iesd.auc.dk

Abstract. The Fork Calculus FC presents a theory of communicating systems in family with CCS, but it differs in the way that processes are put in parallel. In CCS there is a binary parallel operator $|$, and two processes p and q are put in parallel by $p|q$. In FC there is a unary **fork** operator, and a process p is activated to "run in parallel with the rest of the program" by **fork**(p). An operational semantics is defined, and a congruence relation between processes is suggested. In addition, a sound and complete axiomatisation of the congruence is provided. FC has been developed during an investigation of the programming language CML, an extension of ML with concurrency primitives, amongst them a fork operator.

Key words: process calculi, process creation, bisimulation, axiomatisation

CR Classification: D.3.1, D.3.3, F.3.1, F.3.2

1. Introduction

The Fork Calculus FC is motivated by problems encountered while originally investigating semantic properties of the programming language CML (Concurrent ML) [8, 9, 1]. In particular, we address the definition and axiomatisation of suitable equivalences between CML expressions. CML is an extension of ML with CCS-like concurrency primitives. The model behind CML is that of concurrently executing expressions that communicate by synchronous message passing on typed channels. The concurrent aspects of CML, however, differ from CCS in an essential way: In CCS there is a binary parallel operator $|$, and two processes p and q are put in parallel by $p|q$. In CML the binary parallel operator has been replaced by a unary `fork` operator, and a process `p` is activated to "run in parallel with the rest of the

* On Leave from DIKU, University of Copenhagen, Denmark. The work of this author was in addition supported by the Danish 'Forskerakademiet'.
† The work of this author was supported by the Danish Natural Science Research Council project DART and partially by the ESPRIT Basic Research Action 7166, CONCUR2.

Received November 1993. Revised September 1994.

program" by `fork(p)`. The concurrency primitives of CML are provided as a set of "functions", from which the following may be derived:

```
channel  : unit -> 'a chan
fork     : (unit -> 'a) -> unit
transmit : 'a chan * 'a -> unit
accept   : 'a chan -> 'a
```

There are minor differences in the choice of CML primitives, their types and names, in [8], [9] and [1]. In terms of the primitive features `sync`, `send` and `receive` of [1], which we have chosen to follow, `transmit` and `accept` may be defined as: `transmit(x) = (sync(send(x));())` and `accept(x) = sync(receive(x))`. The function `channel` yields a fresh channel each time it is applied. The function `transmit` sends a value to a channel. The function `accept` reads a value from a channel. Finally, the function `fork` starts a separate evaluation of its argument function; that is: `fork(f)` starts the evaluation of `f()`. As an example, consider an implementation in CML of a function `calc:real->real` having the following specification: `calc(x)=cos(x)+sin(x)`. The implementation can be given as follows:

```
fun calc(x) =
      let val r1 = channel()
          val r2 = channel()
      in
        fork(fn () => transmit(r1,cos(x)));
        fork(fn () => transmit(r2,sin(x)));
        (accept(r1) + accept(r2))
      end;
```

The function evaluates the expressions `cos(x)` and `sin(x)` in parallel. First, two (local) channels `r1` and `r2` are allocated. Then, the two evaluations are forked; The two forked evaluations return their respective results on the channels `r1` and `r2`.

One goal is to define an appropriate equivalence between CML expressions. As an example, consider that we reverse the order of the two fork expressions, thus forking the sinus calculation first. Then we would like the resulting function to be equivalent to the one above. The result in both cases is two parallel evaluations of the expressions `cos(x)` and `sin(x)` and we really do not care about how this result is obtained. In general, we will not want to observe the particular forking strategy — only the collected behaviour of the result. As an even more radical example, the implementation given above should satisfy the original specification `calc(x)=cos(x)+sin(x)`. Another natural and important requirement of our equivalence is that it is a *congruence* with respect to all CML constructs as this will allow compositional verification. In particular, equivalences between composite expressions should be inferable from equivalences of subexpressions.

Our work has been influenced by the work on Facile [7], a language that integrates functional and concurrent programming in a way quite similar to CML — there is for example a fork operator. In [7], Facile is given an operational semantics and a notion of equivalence is developed. The Facile equivalence is, however, not shown to be a congruence in the general case; basically due to proof-technical difficulties. The Facile equivalence may though turn out to be a congruence. We have concluded, that the issues of equivalence and compositionality in the presense of forking can best be studied in isolation from the other issues in CML, so we design a calculus, FC (Fork Calculus), that is close to CCS, but which provides a one-argument fork-operator instead of the two-argument parallel-operator found in CCS. Also, the unary prefix–operator of CCS is replaced by a binary operator for sequential composition. Although some of our techniques are related to those in [7], this simplification gives us a congruence, and in addition we approach its complete axiomatisation.

The outline of the remainder of this paper is as follows: in section 2 we present the syntax and a structured operational semantics of our calculus FC. In section 3 we present the obvious strong bisimulation equivalence between processes based on the operational semantics of section 2. This equivalence is, however, *not* a congruence with respect to the constructs of FC; note that strong bisimulation equivalence is preserved by almost all constructs introduced so far in process algebra (see e.g. [2]). In section 4, we provide an explicit characterization of the congruence induced by the strong bisimulation equivalence. In section 5, we offer a complete axiomatisation of this congruence.

2. Syntax and Semantics

The syntax of the calculus is as follows, where N denotes the set of process names.

$$\begin{array}{lcl} \mathcal{L} & ::= & \mathbf{nil} \mid \mathcal{A} \mid \mathcal{L}_1 + \mathcal{L}_2 \mid \mathcal{L}_1; \mathcal{L}_2 \mid \mathbf{fork}(\mathcal{L}) \mid N \\ \mathcal{A} & ::= & \tau \mid a? \mid a! \end{array}$$

nil is the terminated process that can perform no actions. Amongst the actions $\mathcal{A}$ that a process can perform are τ, the internal action, input actions of the form $a?$, and output actions of the form $a!$, where a is a channel name. Two processes that run in parallel may synchronise on complementary actions, one being an input action and the other being an output action containing the same name. The choice between two processes is written as $\mathcal{L}_1 + \mathcal{L}_2$. Two processes can be sequentially composed by $\mathcal{L}_1; \mathcal{L}_2$. This has the traditional interpretation that $\mathcal{L}_1$ is evaluated first until it terminates (becomes **nil**) whereupon $\mathcal{L}_2$ continues. Note that FC here differs from CCS which instead of sequential composition has action prefixing. Sequential composition has greater precedence than choice.

The other main difference from CCS is the **fork** expression: a process is forked with **fork**($\mathcal{L}$). It means that a separate evaluation of $\mathcal{L}$ is begun such that $\mathcal{L}$ is made to run in parallel with the rest of the program. The **fork** expression itself terminates immediately after starting the separate evaluation of $\mathcal{L}$. N is the call of a process, that has been named in a definition of the form $N \stackrel{\text{def}}{=} \mathcal{L}$. In the present theory we disallow recursion in order to obtain a complete axiomatisation, and we only introduce process naming to be able to write some more appealing (nonrecursive) examples. Except for the complete axiomatisation, it is very easy to extend the results of this paper to allow recursion.

So what are the consequences of this seemingly minor change compared to CCS: having sequential composition and forking instead of action prefixing and parallel composition? There are pragmatic as well as semantic consequences. As we shall see, the semantic consequences are quite drastic. Concerning the pragmatic consequences, let us study an example. Consider the following informal requirement specification:

A 'session' at a computer terminal consists of an 'initialisation' followed by a 'run'. The initialisation consists of a 'setup' phase followed by a 'dialog' phase. After the 'setup' phase, a report is sent to a paper printer which will 'print' the report.

This requirement specification can be directly presented in FC as follows (note that for convenience, we shall often leave out the '?' when writing input actions; actions are written with small letters by convention):

$$\begin{aligned} \mathit{Session} &\stackrel{\text{def}}{=} \mathit{Initialise}; \mathit{run} \\ \mathit{Initialise} &\stackrel{\text{def}}{=} \mathit{setup}; \mathbf{fork}(\mathit{print}); \mathit{dialog} \end{aligned}$$

The point to note is that we can locally in *Initialise* express the parallel activation of the printer (*print*) after the *setup* phase. This local activation of a process is not directly expressible in CCS, where we would write something like:

$$\mathit{Session} \stackrel{\text{def}}{=} \mathit{setup}.(\mathit{print}.\mathbf{nil} \mid \mathit{dialog}.\mathit{run}.\mathbf{nil})$$

We see that it is not possible to give a name (*Initialise*) to the pair *setup* and *dialog*: one looses abstraction. So characteristic of the **fork** operator is that we can start a process locally where the need arises, and this gives a possibility of naming a sequential composition that performs parallel activation as a "side effect". Of course the above pragmatic difference between FC and CCS is just a matter of taste, and we shall in the following concentrate on the formal implications of having '**fork**' and ';'.

First, we define an operational semantics for the language of the calculus. That is, the approach taken is that of structured operational semantics as introduced in [6] and later applied to CCS as described in [4]. The difference

between the semantics here and the two existing semantics of CML [9, 1] is (except for the different languages) that we let transitions be labelled with actions. Both in [9] and in [1], transitions are not labelled with actions. Action labels are important from the point of view of defining an appropriate bisimulation equivalence, since bisimulation is defined as identical action behaviour.

A labelled transition system is a triple $(St, Lab, \rightarrow)$, where St is the set of states (for example processes), Lab is the set of labels (actions performed by the processes) and $\rightarrow \subseteq St \times Lab \times St$ is the transition relation: $(st_1, l, st_2) \in \rightarrow$ may be interpreted as "the state st_1 is able to perform the action l and by doing so becomes the state st_2". Typically we use the notation $st_1 \xrightarrow{l} st_2$ for $(st_1, l, st_2) \in \rightarrow$. The transition relation thus defines the dynamic change of states as they perform actions.

The semantics of CCS is normally given in terms of a single labelled transition system where St is the set of CCS processes. In contrast to the CCS semantics, the FC semantics is divided into two layers, corresponding to two labelled transition systems. In the first layer we give semantics to processes seen in isolation (St is the set of processes). In the next layer, we give semantics to collections of processes running in parallel (St is the set of such process collections). When "running" a process, for example $\mathbf{fork}(p); q$ we start out with a collection consisting of that process. After the forking, we have a collection containing two processes, p and q, running in parallel. We refer to a collection of processes as a program (thus St is the set of programs).

2.1 Processes

In this section we give semantics to processes seen in isolation. We shall do this by defining a labelled transition system $(\mathcal{L}, Lab, \hookrightarrow)$ where $\mathcal{L}$ is the set of processes introduced in section 2. Concerning the definition of the labels Lab, assume an infinite set of (channel) names $Chan$. Then Lab (the labels on process transitions) is gradually defined as follows:

$$\begin{array}{lcl} Com & = & \{a? \mid a \in Chan\} \cup \{a! \mid a \in Chan\} \\ Act & = & Com \cup \{\tau\} \\ Lab & = & Act \cup \{\phi(p) \mid p \in \mathcal{L}\} \end{array}$$

The set Com, ranged over by c, is the set of input-output communications that processes can perform. The set Act, ranged over by $\alpha, \beta, \gamma, \ldots$, includes in addition the τ action, and it is the set of actions that we will be able to observe in the end, when executing programs. The set Lab, ranged over by l, includes further labels of the form $\phi(p)$ ($p \in \mathcal{L}$) which arise from evaluation of processes of the form $\mathbf{fork}(p)$. These labels will not be observable at the program layer, since at that level they will be converted into τ actions.

We now define the transition relation $\hookrightarrow \subseteq \mathcal{L} \times Lab \times \mathcal{L}$. Before defining this transition relation we define the predicate $Stop \subseteq \mathcal{L}$. We shall use the

notation $Stop(p)$ instead of $p \in Stop$. Now $Stop$ is defined as the least subset of $\mathcal{L}$ satisfying the following:

$Stop(\mathbf{nil})$,
If $Stop(p_1)$ and $Stop(p_2)$ then $Stop(p_1 + p_2)$ and $Stop(p_1; p_2)$,
If $Stop(p)$ and $N \stackrel{\text{def}}{=} p$ then $Stop(N)$

The operational semantics of FC processes is then as follows.

DEFINITION 1. (THE TRANSITION RELATION $\hookrightarrow$) *Let $\hookrightarrow$ be the smallest subset of $\mathcal{L} \times Lab \times \mathcal{L}$ closed under the following rules:*

$$(\mathbf{Act}) \quad \frac{}{\alpha \overset{\alpha}{\hookrightarrow} \mathbf{nil}}$$

$$(\mathbf{Sum}_1) \quad \frac{p_1 \overset{l}{\hookrightarrow} p_1'}{p_1 + p_2 \overset{l}{\hookrightarrow} p_1'} \qquad (\mathbf{Sum}_2) \quad \frac{p_2 \overset{l}{\hookrightarrow} p_2'}{p_1 + p_2 \overset{l}{\hookrightarrow} p_2'}$$

$$(\mathbf{Seq}_1) \quad \frac{p_1 \overset{l}{\hookrightarrow} p_1'}{p_1; p_2 \overset{l}{\hookrightarrow} p_1'; p_2} \qquad (\mathbf{Seq}_2) \quad \frac{p_2 \overset{l}{\hookrightarrow} p_2'}{p_1; p_2 \overset{l}{\hookrightarrow} p_2'} Stop(p_1)$$

$$(\mathbf{Fork}) \quad \frac{}{\mathbf{fork}(p) \overset{\phi(p)}{\hookrightarrow} \mathbf{nil}} \qquad (\mathbf{Const}) \quad \frac{P \overset{l}{\hookrightarrow} P'}{A \overset{l}{\hookrightarrow} P'} A \stackrel{\text{def}}{=} P$$

The rules should be fairly simple to read. The **Act**-rule says that a process of the form α can perform the action α and become **nil**. The $\mathbf{Sum}_1$-rule says that if p_1 can perform l and become p_1', then the sum $p_1 + p_2$ can also, and symmetrically by the $\mathbf{Sum}_2$-rule. The sequencing rules explain how sequencing proceeds with the leftmost process until it has stopped ($\mathbf{Seq}_1$), whereupon the rightmost process continues ($\mathbf{Seq}_2$). The **Fork**-rule shows how the higher order labels $\phi(p)$ are created. When we come to the program semantics their use will be explained. Finally, the **Const**-rule explains how the name of a process behaves as the defining body. Let us look at an example. The process $\alpha; \mathbf{fork}(\beta); \gamma$ can evaluate as follows:

$$\alpha; \mathbf{fork}(\beta); \gamma \overset{\alpha}{\hookrightarrow} \mathbf{nil}; \mathbf{fork}(\beta); \gamma \overset{\phi(\beta)}{\hookrightarrow} \mathbf{nil}; \gamma \overset{\gamma}{\hookrightarrow} \mathbf{nil}$$

Note that the forked process β just becomes part of the label, and that it is not further used. When executing programs, we will make sure that the forked process is put in parallel with the "rest of the program".

2.2 Programs

A program is a multiset of processes. A multiset can contain several copies of the same element, (in contrast to normal sets), corresponding to the fact that at a certain moment there may be several processes active with exactly the same structure. Formally, the multisets of A elements can be viewed as the elements in $MS(A) = A \rightarrow \mathbf{N}$. That is, a multiset of A elements is a total function from A to the natural numbers. Each A element is mapped to its number of occurrences. The union operator $\cup : (MS(A) \times MS(A)) \rightarrow MS(A)$ is defined by the equation $(S_1 \cup S_2)(a) = S_1(a) + S_2(a)$. A "finite" multiset S can be written $\{\!|p_1, \ldots, p_n|\!\}$ where the number of occurrences of a process q indicates the value $S(q)$. As an example, $\{\!|p|\!\}(p) = 1$ and $\{\!|p|\!\}(q) = 0$ whenever $p \neq q$. We let $\mathcal{L}_p$ denote the set of programs ($\mathcal{L}_p = MS(\mathcal{L})$). The semantics of programs is given in terms of the labelled transition system $(\mathcal{L}_p, Act, \longrightarrow)$, where $\mathcal{L}_p$ is defined here and Act was defined in the previous section. Thus a program can perform the actions in the set Act; recall that these were of the form $a?$, $a!$ or τ. The fork actions $\phi(p)$ are thus not amongst program actions.

We shall now define the transition relation: $\longrightarrow \subseteq \mathcal{L}_p \times Act \times \mathcal{L}_p$. We need the auxiliary function $rev : Com \rightarrow Com$, which for a given communication returns the complementary communication with which it can synchronise; i.e: $rev(a?) = a!$ and $rev(a!) = a?$. The operational semantics of FC programs is then as follows.

Definition 2. (The transition relation $\longrightarrow$) *Let $\longrightarrow$ be the smallest subset of $\mathcal{L}_p \times Act \times \mathcal{L}_p$ closed under the following rules:*

$$(\mathbf{Act}^{\{\!|\,|\!\}}) \quad \frac{p \overset{\alpha}{\hookrightarrow} p'}{\{\!|p|\!\} \overset{\alpha}{\longrightarrow} \{\!|p'|\!\}} \qquad (\mathbf{Fork}^{\{\!|\,|\!\}}) \quad \frac{p \overset{\phi(q)}{\hookrightarrow} p'}{\{\!|p|\!\} \overset{\tau}{\longrightarrow} \{\!|p', q|\!\}}$$

$$(\mathbf{Par}_1^{\{\!|\,|\!\}}) \quad \frac{P_1 \overset{\alpha}{\longrightarrow} P_1'}{P_1 \cup P_2 \overset{\alpha}{\longrightarrow} P_1' \cup P_2} \qquad (\mathbf{Par}_2^{\{\!|\,|\!\}}) \quad \frac{P_1 \overset{c}{\longrightarrow} P_1' \;,\; P_2 \overset{rev(c)}{\longrightarrow} P_2'}{P_1 \cup P_2 \overset{\tau}{\longrightarrow} P_1' \cup P_2'}$$

The $\mathbf{Act}^{\{\!|\,|\!\}}$-rule just says that if a process can perform an action, then a program containing that process can perform the action. Note that α ranges over Act, which does not contain $\phi(p)$ actions. The $\mathbf{Fork}^{\{\!|\,|\!\}}$-rule explains how a $\phi(\ldots)$ action generates a τ action at program level: the forking process and the forked process will after the transition be in parallel. We have chosen to let forking generate a τ action rather than just to do a "silent" fork. The reason for this is our original goal to deal with the existing CML semantics, where forking generates a τ action. Finally, the rule $\mathbf{Par}_1^{\{\!|\,|\!\}}$ explains how a "subset" of a program may perform actions on its own. Note, that we need only one such rule as $\cup$ is clearly commutative. The $\mathbf{Par}_2^{\{\!|\,|\!\}}$–rule shows how two distinct subsets of a program may communicate, generating a τ action.

3. Program and Process Equivalences

The purpose of this section is to define an equivalence relation $\sim \subseteq \mathcal{L} \times \mathcal{L}$ between FC processes. For this purpose we shall, however, first define an equivalence relation $\backsim \subseteq \mathcal{L}_p \times \mathcal{L}_p$ between FC programs.

3.1 Program Equivalence

Our notion of equivalence is based on the concept of bisimulation [4], which again is based on the idea that we only want to distinguish between two programs, if the distinction can be observed by an "observer" examining the actions that the two programs can perform. Note that in the following we shall regard the τ action just as observable as the other actions ($a?$ and $a!$) in *Act*. We shall for example distinguish $\{\!|\alpha;\tau;\beta|\!\}$ and $\{\!|\alpha;\beta|\!\}$. This will yield a rather strict congruence (fewer programs are congruent). We have chosen to make τ observable in our first attempt, since it is the classical choice: first a strong notion of equivalence is defined (where τ is observable), and then one abstracts from τ. The formal definition of $\backsim$ proceeds in the standard way [4] as follows. First, we define the notion of a bisimulation.

Definition 3. (Bisimulation) *A binary relation $S \subseteq \mathcal{L}_p \times \mathcal{L}_p$ is a bisimulation iff $(P,Q) \in S$ implies, for all $\alpha \in Act$,*

(1) Whenever $P \xrightarrow{\alpha} P'$ for some P' then $Q \xrightarrow{\alpha} Q'$ for some Q' and $(P',Q') \in S$

(2) Whenever $Q \xrightarrow{\alpha} Q'$ for some Q' then $P \xrightarrow{\alpha} P'$ for some P' and $(P',Q') \in S$

We write $P \backsim Q$ where $(P,Q) \in S$ for some bisimulation S.

As usual it may be shown that $\backsim$ is itself the largest bisimulation. Moreover, $\backsim$ is easily shown to be a congruence with respect to $\cup$.

3.2 Process Equivalence

In the previous section we introduced an equivalence on programs, and we stated it to be a congruence with respect to the basic operator on programs: $\cup$. What we are really interested in is, however, not programs, but rather processes. Processes are the terms of our language, and programs are "just" semantic objects used for giving semantics to processes. So our definite goal is to define an equivalence on processes. This equivalence must additionally be a congruence with respect to the operators on processes ($+$; **fork**). In this section we shall come up with a process equivalence $\sim$, which seems rather natural and correct, but which, however, turns out not to be satisfactory due to lack of the congruence property. In the section to follow we will define an equivalence $\equiv$ that is also a congruence, in fact the congruence induced by $\sim$; but the present exercise can hopefully motivate the final solution.

We shall define an equivalence relation $\sim \subseteq \mathcal{L} \times \mathcal{L}$ between FC processes. Two processes p and q are said to be equivalent if $(p, q) \in \sim$, which is written more conveniently as $p \sim q$. The idea is to say that two processes are equivalent, if they are equivalent when regarded as programs. Put differently: to see whether two processes are equivalent, "run" them (as programs) and see whether they behave the same way. Formally, we thus define $\sim$ as follows:

DEFINITION 4. (PROCESS EQUIVALENCE) *The relation* $\sim \subseteq \mathcal{L} \times \mathcal{L}$ *is defined as:*

$$p \sim q \Leftrightarrow \{\!|p|\!\} \backsim \{\!|q|\!\}$$

As we shall see, the **fork** operator causes $\sim$ not to be a congruence. This is rather unusual since (strong) bisimilarity is normally preserved by operators. Consider two processes that are related by $\sim$:

$$\tau;\alpha \quad \sim \quad \mathbf{fork}(\alpha)$$

To see this, consider the behaviours of programs $\{\!|\tau;\alpha|\!\}$ and $\{\!|\mathbf{fork}(\alpha)|\!\}$:

$$\begin{array}{lcl} \{\!|\tau;\alpha|\!\} & \xrightarrow{\tau} \{\!|\mathbf{nil};\alpha|\!\} & \xrightarrow{\alpha} \{\!|\mathbf{nil}|\!\} \\ \{\!|\mathbf{fork}(\alpha)|\!\} & \xrightarrow{\tau} \{\!|\mathbf{nil},\alpha|\!\} & \xrightarrow{\alpha} \{\!|\mathbf{nil},\mathbf{nil}|\!\} \end{array}$$

Clearly these behaviours are bisimilar (as they are identical). Now, suppose that we put the two processes into the same context $\_;\beta$. Will it then hold that $\tau;\alpha;\beta \sim \mathbf{fork}(\alpha);\beta$? Unfortunately no!. This can be seen from the behaviours of the programs $\{\!|\tau;\alpha;\beta|\!\}$ and $\{\!|\mathbf{fork}(\alpha);\beta|\!\}$, as illustrated by the transition trees for the two programs in Fig. 1.

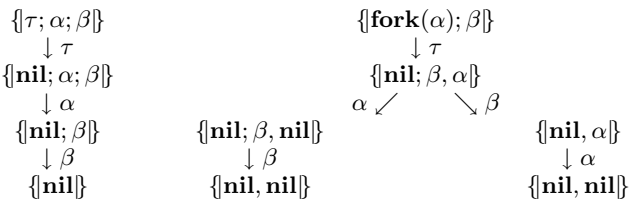

Fig. 1: Different Transition Trees

Thus, $\sim$ is not a congruence, and we have to repair this. In the following, we introduce a process relation that *is* a congruence. It is even the largest congruence induced by (contained in) $\sim$.

4. Process Congruence

In this section we introduce a new process equivalence $\equiv$ being strictly finer than the previously given equivalence $\sim$. As one of the main results of this paper we prove that $\equiv$ is precisely the congruence induced by $\sim$; i.e. $\equiv$ is preserved by all operators of FC, and is the largest such equivalence included in $\sim$. The fact that $\sim$ is not a congruence is illustrated by the previous example demonstrating the difference between the two processes $\tau;\alpha$ and $\mathbf{fork}(\alpha)$ when put into the context $\_;\beta$. The difference lies essentially in the ability of the action α to be in parallel with future computation, which is here represented by the action β.

It is this *ability to be in parallel with future computation* that we want to capture. Future computation is what computes after the *termination* of the observed process. When examining the two programs $\{\!|\tau;\alpha|\!\}$ and $\{\!|\mathbf{fork}(\alpha)|\!\}$, we cannot detect this termination: both programs perform a τ action and then an α action, and that's it. The process $\tau;\alpha$, however, terminates after the second action (α), while the process $\mathbf{fork}(\alpha)$ already terminates after the first action (τ), when the forking has been performed — the forked process α can then execute after this termination. We thus need to add some information that makes it possible to observe termination. This can be done by introducing a special event, π, that, when being performed, signals the successful termination of the observed process. For a given process p under observation, we shall examine $\{\!|p;\pi|\!\}$ rather than just $\{\!|p|\!\}$.

As an example, in order to observe the difference in terms of future computations between the processes $\tau;\alpha$ and $\mathbf{fork}(\alpha)$, we "run" $\tau;\alpha;\pi$ and $\mathbf{fork}(\alpha);\pi$ as programs. That is, we examine the multisets $\{\!|\tau;\alpha;\pi|\!\}$ and $\{\!|\mathbf{fork}(\alpha);\pi|\!\}$. Both programs can perform the trace $\tau\alpha\pi$, however only the program $\{\!|\mathbf{fork}(\alpha);\pi|\!\}$ can perform the trace $\tau\pi\alpha$, where the α action occurs after the π action. This may be illustrated by the transition trees for the two programs, which will be identical to the trees in Fig. 1 with π replacing β.

We are now able to give the following formal definition of the process congruence $\equiv$:

Definition 5. (Process congruence $\equiv$) *The relation $\equiv \subseteq \mathcal{L}\times\mathcal{L}$ is defined as:*

$$p \equiv q \Leftrightarrow \{\!|p;\pi|\!\} \backsim \{\!|q;\pi|\!\}$$

This equivalence is similar to one of the equivalences defined in [7]; however, no axiomatisation is provided in [7].

One of the main results of this paper is that $\equiv$ is the congruence (i.e. preserved by the operators of FC) induced by $\sim$. First of all, it is straightforward to show that $\equiv$ is an equivalence relation. That it is also a congruence is stated as the following theorem.

THEOREM 1. (CONGRUENCE PROPERTY) *Assume processes p_1, p_2 and q where $p_1 \equiv p_2$. Then:*

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathbf{fork}(p_1) & \equiv & \mathbf{fork}(p_2) & q;p_1 & \equiv & q;p_2 \\ p_1 + q & \equiv & p_2 + q & p_1;q & \equiv & p_2;q \end{array}$$

PROOF.

For each equation $p \equiv q$ we shall prove $\{\!|p;\pi|\!\} \smile \{\!|q;\pi|\!\}$. Then the equation follows from definition 5. To prove $\{\!|p;\pi|\!\} \smile \{\!|q;\pi|\!\}$ we define a relation S containing the pair $(\{\!|p;\pi|\!\}, \{\!|q;\pi|\!\})$, and then show that S is a bisimulation according to definition 3.

That $\equiv$ is the congruence induced by $\sim$ is formulated as follows.

THEOREM 2. (INDUCED CONGRUENCE PROPERTY) $\equiv$ *is the largest congruence contained in* $\sim$.

PROOF.

It is easy to show that $\equiv$ is contained in $\sim$. Essentially we show that $\{\!|p|\!\} \smile \{\!|q|\!\}$ whenever $\{\!|p;\pi|\!\} \smile \{\!|q;\pi|\!\}$ by using definition 3. To see that $\equiv$ is the largest such, assume that $\simeq$ is an arbitrary congruence included in $\sim$. Now assume $p \simeq q$. As $\simeq$ is assumed to be a congruence it follows that $p;\pi \simeq q;\pi$, and — as $\simeq$ is supposed to be included in $\sim$ — also $p;\pi \sim q;\pi$. However, then $p \equiv q$ according to definitions 4 and 5, and it follows that $\simeq$ is included in $\equiv$.

5. Strong Axiomatisation

In this section we present a sound and complete axiomatisation for the process congruence $\equiv$. The completeness of the axiomatisation is obtained in a classical manner through the use of normal forms — being process expressions with a very restricted use of the **fork**–operator. More precisely, the axiomatisation consists of a collection of basic equational axioms, and two expansion laws. The basic axioms achieve completeness for normal form processes, and the expansion laws (together with the basic axioms) enable arbitrary process expressions to be transformed into normal form, thus yielding completeness for the full calculus.

In more classical process calculi, expansion laws allow parallel composition to be replaced by non–determinism. In our calculus, the expansion laws will have an analogous purpose, namely that of replacing as much as possible forking with non–deterministic choice. However — as we shall see in the next section — our calculus seems to lack expressive power for suitable expansion laws to exist. To overcome this problem, we shall extend the calculus with an extra operator described below. This phenomenon is similar to the necessity of the leftmerge operator in PL in order to obtain a finite sound and complete equational axiomatisation as shown in [5].

5.1 Searching for an Expansion Law

We need expansion laws, that describe how forking may be expanded (as much as possible) into non–determinism — similar to the expansion law of CCS that expands parallel composition into non–determinism. For example, consider the following instance of the CCS expansion law:

$$a.\mathbf{nil} \mid b.\mathbf{nil} \quad = \quad a.b.\mathbf{nil} + b.a.\mathbf{nil}$$

Let us try to search for a similar expansion law for the FC **fork** operator. We try to expand the corresponding FC–process $\mathbf{fork}(a); b$. An initial guess can be the following:

$$\mathbf{fork}(a); b \quad = \quad a; b + b; a$$

However, this equation is not sound (it does not hold when replacing $=$ with $\equiv$), and can therefore immediately be ignored. Intuitively, the reason is that the right hand side contains no information about a's capability to be in parallel with "the rest of the program". In the next seemingly correct equation we try to take this into account:

$$\mathbf{fork}(a); b = a; b + b; \mathbf{fork}(a) \tag{1}$$

This equation is not sound either: the left hand side can perform a τ action as the first thing, while the right hand side can perform either an a action or a b action (note that since we want to axiomatise strong process congruence $\equiv$, τ actions are observable). Trying to repair this problem we obtain:

$$\mathbf{fork}(a); b \quad = \quad \tau; (a; b + b; \mathbf{fork}(a)) \tag{2}$$

This equation is not sound either: the left hand side can only perform one τ action, while the right hand side can perform two (the explicitly mentioned and the one caused by **fork**).

So it seems that we cannot in general expand forking! This leaves open the problem of how to establish equality between process expressions involving the **fork**-operator. The latter two attempts above suggest that we need a version of **fork** which is instantaneous without the initial internal transition caused by **fork**. Put alternatively, we lack the ability to express that something *has been* forked, with the initial τ action already having occurred previously. To gain this expressivity, we add a new instantaneous forking operator to our calculus. We call this operator **forked**, and we write $\mathbf{forked}(p)$ to mean 'fork p without a τ'. One can regard this operator alternatively by reading $\mathbf{forked}(p)$ as 'p has been forked', or shorter: 'forked

p'. The latter reading suggests the following relationship between **fork** and **forked**:

$$\mathbf{fork}(p) \quad = \quad \tau; \mathbf{forked}(p)$$

Let us now try to write new versions of Eq. (1) and Eq. (2), that contain the **forked** operator in appropriate positions. These equations can be shown sound on basis of the formal definition we give of the **forked** operator in the following section. The equations are:

$$\begin{aligned} \mathbf{forked}(a); b &= a; b + b; \mathbf{forked}(a) \\ \mathbf{fork}(a); b &= \tau; (a; b + b; \mathbf{forked}(a)) \end{aligned}$$

However, it remains to settle whether it is possible to find an operator **forked** satisfying the above equations. In the next section we give an affirmative answer by providing an operational semantics for this new operator.

5.2 Adding the 'Forked'-operator

The syntax is extended with the **forked** alternative:

$$\mathcal{L}^{\Phi} \quad ::= \quad \mathbf{nil} \mid \mathcal{A} \mid {\mathcal{L}^{\Phi}}_1 + {\mathcal{L}^{\Phi}}_2 \mid {\mathcal{L}^{\Phi}}_1; {\mathcal{L}^{\Phi}}_2 \mid \mathbf{fork}(\mathcal{L}^{\Phi}) \mid \mathbf{forked}(\mathcal{L}^{\Phi}) \mid N$$

The extended calculus is called FC^{Φ}. The set *Lab* of actions that a process can perform is extended accordingly:

$$Lab \quad = \quad Act \cup \{\phi(p) \mid p \in \mathcal{L}^{\Phi}\} \cup \{\Phi(p) \mid p \in \mathcal{L}^{\Phi}\}$$

The new semantics of processes and programs is obtained by adding the following rules:

$$(\mathbf{Forked}) \quad \frac{}{\mathbf{forked}(p) \overset{\Phi(p)}{\hookrightarrow} \mathbf{nil}} \qquad (\mathbf{Forked}^{\{\!|\!\}}) \quad \frac{p \overset{\Phi(q)}{\hookrightarrow} p', \{\!|p', q|\!\} \xrightarrow{\alpha} R}{\{\!|p|\!\} \xrightarrow{\alpha} R}$$

The definition of *Stop* is unchanged for the extended calculus. That is: $\neg Stop(\mathbf{forked}(p))$ for any p. The previous semantic definitions from section 2 — the sets *Com* and *Act* (however not *Lab*), and the operational semantic rules for the original operators — carry over unchanged to the extended calculus. Likewise do the definitions of the program equivalence $\smile$ and the process equivalences $\sim$ and $\equiv$ from sections 3 and 4. The proofs of the following properties also carry over: (1) $\smile$ is a congruence with respect to $\cup$, (2) $\sim$ is an equivalence (but not a congruence, as before), (3) $\equiv$ is an equivalence.

The congruence property of $\equiv$ does, however, not apply to the extended calculus. The reason for this can be illustrated by the following example. It is easily shown that $\mathbf{forked}(\mathbf{nil}) \equiv \mathbf{nil}$. Now, consider the two processes $\mathbf{forked}(\mathbf{nil}) + \alpha$ and $\mathbf{nil} + \alpha$. In order for $\equiv$ to be a congruence, these two processes should be equivalent (show the same behaviour). They are, however, not equivalent since $\mathbf{forked}(\mathbf{nil}) + \alpha$ can avoid to perform the α action while $\mathbf{nil} + \alpha$ must perform the α action.

We choose to loosen the requirement that $\equiv$ is a congruence. The congruence property holds for all the operators, except for the choice operator, hence it becomes conditioned (even a conditioned congruence can be useful from a practical point of view). Let $\mathit{Active}(q)$, for any process $q \in \mathcal{L}^{\Phi}$, mean that q can perform an action. That is: $\mathit{Active}(q) \stackrel{\text{def}}{=} \exists \alpha \in \mathit{Act}, P \in \mathcal{L}^{\Phi}_{p} \cdot \{\!|q|\!\} \xrightarrow{\alpha} P$. Then the loosened congruence property can be stated as follows: assume for two processes p_1, p_2 that $p_1 \equiv p_2$. Then for any process q:

$$p_1 + q \equiv p_2 + q \quad \textit{provided } \mathit{Active}(q) \Rightarrow (\mathit{Stop}(p_1) \Leftrightarrow \mathit{Stop}(p_2))$$

The condition is motivated by our previous example, where q is α, and p_1 is $\mathbf{forked}(\mathbf{nil})$ and p_2 is $\mathbf{nil}$, thus the condition is not satisfied in this case. Note that $\mathit{Stop}(\mathbf{forked}(\mathbf{nil}))$ does not hold according the definition of *Stop* which is unchanged for the extended calculus.

5.3 Basic Axioms and Inference Rules

The axiom system $\mathcal{AS}$ consists of a set $\mathcal{AX}$ of equational axiom schemes, two expansion laws $\mathcal{E}_1$ and $\mathcal{E}_2$, using normal forms, and a set $\mathcal{I}$ of inference rules, that represents the fact that $\equiv$ is an equivalence and (nearly) a congruence. In this section we shall present the basic equational axioms $\mathcal{AX}$.

DEFINITION 6. (BASIC AXIOM SYSTEM $\mathcal{AX}$) *The basic axiom system $\mathcal{AX}$ consists of the axioms:*

$$\begin{aligned} \mathbf{fork}(p) &= \tau; \mathbf{forked}(p) \\ \mathbf{forked}(\alpha; \mathbf{forked}(p) + q) &= \mathbf{forked}(\alpha; p + q) \\ \mathbf{forked}(\mathbf{nil}) &= \mathbf{nil} \end{aligned}$$

$$\begin{aligned} p; \mathbf{nil} &= p & p + (q + r) &= (p + q) + r \\ \mathbf{nil}; p &= p & p + q &= q + p \\ (p; q); r &= p; (q; r) & p + \mathbf{nil} &= p \\ (p + q); r &\stackrel{*}{=} p; r + q; r & p + p &= p \end{aligned}$$

The $\stackrel{*}{=}$ equality holds when $\neg \mathit{Stop}(p)$ and $\neg \mathit{Stop}(q)$.

5.4 Normal Forms

Our way to completeness is classical in that it is based on *normal forms*. For processes in normal form, the basic axioms $\mathcal{AX}$ will be shown to be complete, and we shall in the next section present two expansion laws that will enable any process term to be transformed into a provable equivalent normal form process term.

Processes in normal form contain no applications of the **fork** operator, and applications of the **forked** operator occur last. One intuition behind this is that when observing a process we cannot observe the individual forkings; and as long as the original process has not terminated, we cannot observe which actions come from it and which actions come from the forked processes. We can, however, in contexts observe when the process terminates and what at that time has been forked. In principle a process is brought into normal form by first replacing applications $\mathbf{fork}(p)$ by $\tau;\mathbf{forked}(p)$, and then by moving the applications of **forked** as much as possible "to the right". Note that we must deal with the **forked** operator in normal forms, since we must keep track of what is potentially in parallel with future computation.

As an example, consider the process $\mathbf{fork}(\alpha);\beta$. First, we can replace $\mathbf{fork}(\alpha)$ with $\tau;\mathbf{forked}(\alpha)$, obtaining $\tau;\mathbf{forked}(\alpha);\beta$. Then the process $\mathbf{forked}(\alpha);\beta$, is brought into the form $\alpha;\beta+\beta;\mathbf{forked}(\alpha)$. As result we obtain $\tau;(\alpha;\beta+\beta;\mathbf{forked}(\alpha))$. We see that there is no application of the **fork** operator, and that the application of the **forked** operator occurs last. Let us now formally introduce normal forms:

DEFINITION 7. (NORMAL FORM) *A process p is in normal form, if it is a term in N, where:*

$$N ::= \textstyle\sum_i \alpha_i; N_i + \sum_j \beta_j; \mathbf{forked}(B_j) \qquad \textit{and} \qquad B ::= \textstyle\sum_k \beta_k; B_k$$

A process p is in simple normal form, if it is a term in B.

Note that simple normal forms contain no applications of the **forked** operator. That is, once we have reached the termination of a process (second alternative in the definition of N), what remains has all been forked, and to express this we only need one application of the **forked** operator. That is, we do not care about any "further" forking.

5.5 Expansion Laws

The two expansion laws to be presented in this section will enable any process term to be transformed into normal form. To motivate the two expansion laws consider the case of a sequential composition of two process terms N and M already in normal form. Now assume N is of the following form:

$$N = \sum_i \alpha_i; N_i + \sum_j \beta_j; \mathbf{forked}(B_j)$$

Then, when using the laws for distributing ; over + and laws of associativity for ;, we obtain:

$$\vdash N;M = \sum_i \alpha_i;(N_i;M) + \sum_j \beta_j;(\mathbf{forked}(B_j);M)$$

To enable this term to be transformed into a normal form we introduce the following expansion law:

DEFINITION 8. (EXPANSION LAW $\mathcal{E}_1$)
Let: $A = \sum_i \alpha_i;A_i$, $N = \sum_j \beta_j;N_j$, $L = \sum_k \gamma_k;\mathbf{forked}(C_k)$ *and* $M = N + L$ *where* M *is not* **nil**. *Then:*

$$\mathbf{forked}(A);M = p+q+r+s+t$$

$$\textit{where} \begin{cases} p &= \sum_i \alpha_i;\mathbf{forked}(A_i);M \\ q &= \sum_j \beta_j;\mathbf{forked}(A);N_j \\ r &= \sum_k \gamma_k;\mathbf{forked}(A);\mathbf{forked}(C_k) \\ s &= \sum_{\alpha_i = rev(\beta_j)} \tau;\mathbf{forked}(A_i);N_j \\ t &= \sum_{\alpha_i = rev(\gamma_k)} \tau;\mathbf{forked}(A_i);\mathbf{forked}(C_k) \end{cases}$$

To be able to inductively transform $\mathbf{forked}(A);M$ into normal form, we see in definition 8 that we need to be able to handle terms of the form $\mathbf{forked}(A);\mathbf{forked}(B)$. This motivates the second expansion law below:

DEFINITION 9. (EXPANSION LAW $\mathcal{E}_2$)
Let: $A = \sum_i \alpha_i;A_i$ *and* $B = \sum_j \beta_j;B_j$. *Then:*

$$\mathbf{forked}(A);\mathbf{forked}(B) = \mathbf{forked}(p+q+r)$$

$$\textit{where} \begin{cases} p &= \sum_i \alpha_i;\mathbf{forked}(A_i);\mathbf{forked}(B) \\ q &= \sum_j \beta_j;\mathbf{forked}(A);\mathbf{forked}(B_j) \\ r &= \sum_{\alpha_i = rev(\beta_j)} \tau;\mathbf{forked}(A_i);\mathbf{forked}(B_j) \end{cases}$$

The basic axioms $\mathcal{AX}$, the two expansion laws $\mathcal{E}_1$ and $\mathcal{E}_2$ and the inference rules $\mathcal{I}$ constitute our proof system $\mathcal{AS}$, which we shall show to be (limited) complete. We write $\vdash p = q$ if the equivalence of p and q can be entailed within the proof system $\mathcal{AS}$.

5.6 Soundness and Limited Completeness

In this section we state, that the axiom system $\mathcal{AS}$ is sound and complete with respect to strong process congruence. That is, $\vdash p = q \Leftrightarrow p \equiv q$. The basic technique for showing completeness is classical; i.e. first to prove that each process term in the original **forked**-free calculus of section 2 is equivalent to a term in normal form, and then to prove completeness for normal forms. The fact that the completeness result only will apply to **forked**-free processes makes it "limited". First we state the soundness result.

THEOREM 3. ($\mathcal{AS}$ IS SOUND) *For all $p, q \in \mathcal{L}^{\Phi}$, whenever $\vdash p = q$ then $p \equiv q$.*

Note that the result holds for all p, q in FC^{Φ}. The following proposition states that there exists a normal form for every process term in FC. Note that there does not exist a normal form for every process term in FC^{Φ}; as an example, the process **forked**(α) has no normal form.

PROPOSITION 1. (NORMAL FORMS FOR FC) *For all $p \in \mathcal{L}$, there exists a normal form N such that: $\vdash p = N$.*

The next step is to show completeness for normal forms.

PROPOSITION 2. (NORMAL FORM COMPLETENESS) *For all normal forms $M, N \in \mathcal{L}^{\Phi}$, whenever $M \equiv N$ then $\vdash M = N$.*

These two propositions can be used to prove the (limited) completeness result. The completeness result is limited in the sense that it only holds for process terms of FC (and not FC^{Φ}).

THEOREM 4. ($\mathcal{AS}$ IS LIMITED COMPLETE) *For all $p, q \in \mathcal{L}$, whenever $p \equiv q$ then $\vdash p = q$.*

6. Conclusion and Future Work

In this paper we have identified and axiomatised a congruence $\equiv$ between FC–processes. Also, it has been demonstrated that $\equiv$ is the congruence induced by a natural strong bisimulation equivalence $\sim$.

In practice, however, both $\sim$ and $\equiv$ are too fine, as they are sensitive to internal computations. To obtain a more suitable equivalence, we may consider the *weak* bisimulation equivalence $\approx$ between processes, in which we abstract away from τ–transitions. Doing this, we will observe the same phenomena as we did for the strong equivalence: $\approx$ is not preserved by sequential composition. Applying the technique of this paper — i.e. defining $p \cong q$ iff $p;\pi \approx q;\pi$ — we obtain the coarsest equivalence contained in $\approx$ which is also preserved by ; (and **fork**). However, $\cong$ will not quite be a congruence as it will not be preserved by the choice–operator. This is a classical problem [4], and we conjecture that the classical techniques for overcoming this problem will apply to $\cong$. Also, we conjecture that the congruence obtained in this manner will be completely axiomatized by the axiomatization $\mathcal{AS}$ presented in this paper augmented with the classical τ–laws [4].

As immediate future work we intend to investigate the FC–calculus extended with primitives for dynamic channel creation and communication of channels. Further future work includes design of a (logical) specification language for specifying properties of FC programs. It should be shown that

the designed specification language is *adequate* with respect to a suitable process–equivalence (for example the one defined here); i.e. two processes enjoy the same properties of the specification language precisely when they are behaviourally equivalent. One may consider a modification of the specification language which is defined in [3] for CCS, usually referred to as Hennessy-Milner logic.

A more long–term ambition is to extend our work to the full language of CML.

References

[1] Berry, D., Milner, R., and Turner, D.N. 1992. A Semantics for ML Concurrency Primitives. In *Proceedings of the 19th ACM Symphosium on Principles of Programming Languages*.

[2] Groote, J. F. and Vaandrager, F. W. 1989. Structured Operational Semantics and Bisimulation as a Congruence. *LNCS 372*.

[3] Hennessy, M. and Milner, R. 1985. Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM 32*, 1 (January).

[4] Milner, R. 1989. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall.

[5] Moller, F. 1990. The Importance of the Left Merge Operator in Process Algebra. *LNCS 443*.

[6] Plotkin, G. 1981. A Structural Approach to Operational Semantics. FN 19, DAIMI, Aarhus University, Denmark.

[7] Prasad, S., Giacalone, A., and Mishra, P. 1990. Operational and Algebraic Semantics for Facile. In *Proceedings of ICALP90, LNCS 443*.

[8] Reppy, J. H. 1991. CML: A Higher-order Concurrent Language. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (SIGPLAN Notices 26(6))*.

[9] Reppy, J.H. 1991. An Operational Semantics of First–class Synchronous Operations. Tech. Report TR 91–1232, Cornell University, Department of Computer Science.