A Refinement Logic for the Fork Calculus

Klaus Havelund* Kim Guldstrand Larsen†

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 $|$, whereas FC contains a unary ***fork*** operator. We provide FC with an operational semantics, together with a congruence relation between processes. Further, a refinement logic for program specification and design is presented. In this logic it is possible to freely mix programming constructs with specification constructs, thereby allowing us to define a compositional proof system. The proof rules of this system are applied to a non-trivial example.

1 Introduction

One goal for work within program specification is to provide a theory for the formal refinement of specifications into programs via sequences of verified-correct development steps. In this paper we shall pursue this goal by focusing on specification and stepwise refinement into programs in the Fork Calculus.

The Fork Calculus, FC, first presented in [HL93, Hav94], is a process algebra at the level of CCS [Mil89]. It provides a language for programming parallel systems, and it is kept minimal in size (as CCS) in order to allow for theoretical dissection. Both calculi include an operator for the parallel activation of processes, that may synchronise (communicate) on named channels. But the two operators are, however, very different. In CCS there is a binary operator, '$|$', for the parallel composition of two processes, and two processes p and q are composed to run in parallel by $p|q$. In FC there is a unary ***fork*** operator, and p is activated to run in parallel with q by $\textbf{\textit{fork}}(p); q$. Sequential composition of arbitrary processes is another essential construct in FC, in contrast to CCS which has action prefixing.

One can argue that the above differences is just a question of syntax, but it appears to be somewhat more profound. Consider for example the process $\textbf{\textit{fork}}(p)$. This process behaves like p, if regarded in isolation, but surely $\textbf{\textit{fork}}(p); q$ behaves in general differently from $p; q$, given that sequential composition has the usual meaning: "first p and then q".

*Email: havelund@lix.polytechnique.fr. Ecole Polytechnique Paris, LIX, 91128 Palaiseau Cedex, France.

†Email: kgl@iesd.auc.dk. Aalborg University, Institute for Electronic Systems, Frederik Bajersvej 7, 9220 Aalborg, Denmark. The work of this author was supported partly by the Danish Basic Research Foundation project BRICS and partly by the ESPRIT Basic Research Action 7166, CONCUR2.

The observation to make is that p in $\boldsymbol{fork}(p)$ has the ability to "be in parallel with future computation, whatever that might be". The problems arise of course because we require that $\boldsymbol{fork}(p)$ must have a semantics on its own, and not just when put into a final context. The definition of a semantics and equivalences for FC has been influenced by the work on Facile [PGM90] and CML [Rep91], languages that integrate functional and concurrent programming.

The paper is organised as follows. In section 2 we present the Fork Calculus, FC. In section 3 we present the refinement logic, and in section 4 we partly present a proof system. Section 5 reports on a non-trivial example based on a protocol which is developed by refinement. Finally in section 6 some conclusions are drawn. For the complete proof system and a detailed treatment of the example, we refer to the full version of our paper [HL94].

2 The Fork Calculus

In this section we present FC. We give its syntax, its operational semantics and we define an equivalence relation between terms of the process language. This equivalence is in addition a congruence. FC differs from CCS in that it has a unary $\boldsymbol{fork}$-operator instead of binary parallel composition, it has sequential composition instead of action prefixing, and finally it has guarded choice, instead of unguarded choice, in order to obtain desirable properties of the logic we are going to define later. The syntax of the calculus is as follows.

$$p \ ::= \ \sum_{i \in I} \alpha_i ; p_i \mid p_1 ; p_2 \mid \boldsymbol{fork}(p) \mid (a)p \mid \boldsymbol{fix}\, x \cdot p \mid x$$

The $\sum_{i \in I} \alpha_i ; p_i$ construct represents an action guarded choice between a finite number of processes p_i, each guarded by an action α_i. An action α can either be an input action $a?$, an output action $a!$, where a is a channel name, or the silent (internal) action τ. When writing choice expressions we use $+$ to combine the alternatives, leaving out the indices. As an example, $a!; p_1 + \tau; p_2$ represents the process that either can perform an $a!$-action and then continue as p_1, or it can perform a τ-action, and then continue as p_2. We shall use the constant **nil** to represent the empty choice where the index set $I = \emptyset$. This is the inactive process. $p_1; p_2$ denotes sequential composition. A process p is forked with $\boldsymbol{fork}(p)$. It means that a separate evaluation of p is begun which becomes in parallel with the rest of the program. The $\boldsymbol{fork}(p)$ term itself terminates immediately after starting the separate evaluation of p. 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 term $(a)p$ is similar to channel restriction $p \backslash a$ of CCS. Finally $\boldsymbol{fix}\, x \cdot p$ is the usual way to introduce recursion.

We adopt the convention that the operators have decreasing binding power in the following order: Sequential composition (tightest binding), Choice, Recursion, Restriction. We shall further use the convention to interpretate a process $\alpha; p$ as $\sum_{i \in \{1\}} \alpha; p$. Finally, the process α is short for $\alpha; \mathbf{nil}$. We denote by $\mathcal{L}_0$ the set of all well guarded process terms, and by $\mathcal{L}$ the set of all closed and well guarded process terms. [1]

[1] The notions of closedness and well guardedness can be defined in standard manner on the syntactic structure of process terms.

We define a structured operational semantics [Plo81] for the language of the calculus. The semantics of CCS is normally given in terms of a single *labelled transition system*. 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. In the next layer, we give semantics to multisets of processes running in parallel. When "running" a process, for example $\boldsymbol{fork}(p); q$ we start out with a multiset consisting of that process. After the forking, we have a multiset containing two processes, p and q, running in parallel.

Processes

In this section we give semantics to processes seen in isolation. We shall do this by defining the labelled transition system $(\mathcal{L}, Lab, \hookrightarrow)$. 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:

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

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 will finally be observable. 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 $\boldsymbol{fork}(p)$. Lab also includes labels of the form $\Lambda(k)$ ($k \in Chan$) which arise from evaluation of processes of the form $(a)p$. The latter two kinds of labels will not be observable at the second layer.

We now define the transition relation $\hookrightarrow \subseteq \mathcal{L} \times Lab \times \mathcal{L}$. Before defining this transition relation we define the set $Stop \subseteq \mathcal{L}$ of stopped processes by the grammar: $s ::= \mathbf{nil} \mid s_1; s_2 \mid (a)s \mid \boldsymbol{fix}\, x \cdot s$. The operational semantics of FC processes is then as follows. Let $\hookrightarrow$ be the smallest subset of $\mathcal{L} \times Lab \times \mathcal{L}$ closed under the following rules:

$$(\textbf{Choice}) \quad \frac{}{\sum_i \alpha_i; p_i \overset{\alpha_i}{\hookrightarrow} p_i} \qquad\qquad (\textbf{Fork}) \quad \frac{}{\boldsymbol{fork}(p) \overset{\Phi(p)}{\hookrightarrow} \mathbf{nil}}$$

$$(\textbf{Sequence}_1) \quad \frac{p_1 \overset{l}{\hookrightarrow} p_1'}{p_1; p_2 \overset{l}{\hookrightarrow} p_1'; p_2} \qquad\qquad (\textbf{Allocate}) \quad \frac{}{(a)p \overset{\Lambda(k)}{\hookrightarrow} p[k/a]}\, k \in Chan$$

$$(\textbf{Sequence}_2) \quad \frac{p_2 \overset{l}{\hookrightarrow} p_2'}{p_1; p_2 \overset{l}{\hookrightarrow} p_2'}\, Stop(p_1) \qquad\qquad (\textbf{Recursion}) \quad \frac{p[(\boldsymbol{fix}\, x \cdot p)/x] \overset{l}{\hookrightarrow} p'}{\boldsymbol{fix}\, x \cdot p \overset{l}{\hookrightarrow} p'}$$

Note the use of the appropriate higher order labels in the (**Fork**) and (**Allocate**) rules.

Configurations

A *program* is a multiset of processes. We let $\mathcal{M}$ denote the set of programs. In order to give semantics to programs that allocate (internal) channels, we introduce a component into the semantics, that explicitly keeps track of 'already allocated channels'. We refer to this component as the *record*, and we represent it as a set of (the already allocated) channels: $Record \stackrel{\text{def}}{=} \mathcal{P}(Chan)$. We let K range over $Record$. A channel allocation yields a new channel that is not already in the record, and the record is thereafter updated (extended) with the new channel.

A *K-configuration* $K \triangleright P$ consists of a record K and a program P. Note that we shall only consider well formed K-configurations $K \triangleright P$, where the record K includes all the channels occurring in the program P. The semantics of K-configurations is given in terms of the labelled transition system $(KCon, Act, \rightarrowtail)$, where $KCon$ denotes the set of well formed K-configurations. Thus a K-configuration can only perform the actions in the set Act; i.e. actions of the form $a?$, $a!$ or τ.

We shall now define the transition relation: $\rightarrowtail \subseteq KCon \times Act \times KCon$. We need the auxiliary function $rev : Com \to Com$, 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 K-configurations is then as follows. Let $\rightarrowtail$ be the smallest subset of $KCon \times Act \times KCon$ closed under the following rules:

$$(\textbf{Action}^g) \qquad \frac{p \stackrel{\alpha}{\hookrightarrow} p'}{K \triangleright \{\!|p|\!\} \stackrel{\alpha}{\rightarrowtail} K \triangleright \{\!|p'|\!\}}$$

$$(\textbf{Fork}^g) \qquad \frac{p \stackrel{\Phi(q)}{\hookrightarrow} p' \;,\; K \triangleright \{\!|p', q|\!\} \stackrel{\alpha}{\rightarrowtail} K' \triangleright R}{K \triangleright \{\!|p|\!\} \stackrel{\alpha}{\rightarrowtail} K' \triangleright R}$$

$$(\textbf{Allocate}^g) \qquad \frac{p \stackrel{\Lambda(k)}{\hookrightarrow} p' \;,\; K \cup \{k\} \triangleright \{\!|p'|\!\} \stackrel{\alpha}{\rightarrowtail} K' \triangleright R}{K \triangleright \{\!|p|\!\} \stackrel{\alpha}{\rightarrowtail} K' \triangleright R}\, k \notin K$$

$$(\textbf{Parallel}^g_1) \qquad \frac{K \triangleright P_1 \stackrel{\alpha}{\rightarrowtail} K' \triangleright P_1'}{K \triangleright P_1 \cup P_2 \stackrel{\alpha}{\rightarrowtail} K' \triangleright P_1' \cup P_2}$$

$$(\textbf{Parallel}^g_2) \qquad \frac{K \triangleright P_1 \stackrel{c}{\rightarrowtail} K' \triangleright P_1' \;,\; K \triangleright P_2 \stackrel{rev(c)}{\rightarrowtail} K'' \triangleright P_2'}{K \triangleright P_1 \cup P_2 \stackrel{\tau}{\rightarrowtail} K' \cup K'' \triangleright P_1' \cup P_2'}\, K = K' \cap K''$$

The ($\textbf{Fork}^g$) rule explains how a $\Phi(q)$ label is used: if a process p can fork a process q and thereby go into p', and if p' and q in parallel can go into R, then p can go into R (with a corresponding K-transformation). Likewise, the ($\textbf{Allocate}^g$) rule explains how a $\Lambda(k)$ is used: if a process p can allocate a channel k (where k is new; that is: not in K) and thereby go into p', and if p' with an updated K can go into R, then p can go into R (with a corresponding K-transformation). The ($\textbf{Parallel}^g_2$) rule shows how two distinct subsets of a program may communicate, resulting in a τ action. The condition on this rule states that if P_1 and P_2 allocate new channels "on the way", resulting in K' and K'', then none of these new channels must be in common (the only common channels are those in K).

We need to extend the semantics further. In order to internalise dynamic channels (the channels that are introduced by $(\_)\_$), we need a component in the semantics, that identifies the static channels (channels not introduced by a $(\_)\_$). Note that the record initially contains all the static channels, but updating the record makes it no longer possible to identify the initial record. We refer to this component as the *window*, and we represent it as the set of static channels: $\mathit{Window} \stackrel{\text{def}}{=} \mathcal{P}(\mathit{Chan})$. We let W range over Window. The window never changes throughout the execution of a program. A window together with a record is referred to as an *environment*: $\mathit{Env} \stackrel{\text{def}}{=} \mathit{Window} \times \mathit{Record}$.

A *configuration* $(W, K) \triangleright P$ consists of an environment (W, K) and a program P. We shall only consider well formed configurations, where the window (and the set of free channels in the program) is a subset of the record. We denote by Con the set of (well formed) configurations. The semantics of configurations is given in terms of the labelled transition system $(\mathit{Con}, \mathit{Act}, \longrightarrow)$. In order to define $\longrightarrow$ we define the predicate $\_\ \mathit{allows}\ \_ : \mathit{Window} \times \mathit{Act}$ as follows: $W\ \mathit{allows}\ \tau = \mathit{true}$, $W\ \mathit{allows}\ k? = (k \in W)$ and $W\ \mathit{allows}\ k! = (k \in W)$. Then we can define the dynamic behaviour of configurations. Let $\longrightarrow$ be the smallest subset of $\mathit{Con} \times \mathit{Act} \times \mathit{Con}$ satisfying the following rule:

$$\frac{K \triangleright P \stackrel{\alpha}{\rightarrowtail} K' \triangleright P'}{(W, K) \triangleright P \stackrel{\alpha}{\longrightarrow} (W, K') \triangleright P'} W\ \mathit{allows}\ \alpha$$

Process Congruence

We now define a bisimulation–like equivalence relation $\equiv \subseteq \mathcal{L} \times \mathcal{L}$ between processes, which has been proven to be preserved by all constructs of the calculus (i.e. $\equiv$ is a congruence). To formalise this, we shall, however, first define an equivalence relation $\smile \subseteq \mathit{Con} \times \mathit{Con}$ between configurations. We define $\smile$ in terms of the concept of bisimulation [Mil89].

A binary relation $S \subseteq \mathit{Con} \times \mathit{Con}$ is a *bisimulation* iff $(P, Q) \in S$ implies, for all $\alpha \in \mathit{Act}$,

1. Whenever $P \stackrel{\alpha}{\longrightarrow} P'$ for some P' then $Q \stackrel{\alpha}{\longrightarrow} Q'$ for some Q' and $(P', Q') \in S$
2. Whenever $Q \stackrel{\alpha}{\longrightarrow} Q'$ for some Q' then $P \stackrel{\alpha}{\longrightarrow} P'$ for some P' and $(P', Q') \in S$

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

Now, two processes are equivalent, if they are equivalent when "lifted" to configurations. Formally, we associate to each process p its *initial configuration* $\mathit{Config}[p]$. Let $\mathit{CV}[p]$ denote the set of free channel names occurring in the process p, that is: channels not under the scope of a channel restriction. Then:

$$\mathit{Config}[p] \stackrel{\text{def}}{=} (\mathit{CV}[p], \mathit{CV}[p]) \triangleright \{\!| p; \pi |\!\}$$

The π-action is a special reserved action that is not allowed to occur in p. Its purpose is to make it possible to observe the termination of the process p; termination in the sense that p might have forked processes which are still active, but p itself has terminated. As an example, consider the two processes $p \stackrel{\text{def}}{=} \boldsymbol{fork}(a!)$ and $q \stackrel{\text{def}}{=} a!$. Regarded in isolation, their behaviours are the same, they can both perform a $a!$-action. If however we put them into a context, for example $\_; \pi$, then in $p; \pi$, the action $a!$ will be in parallel with π, which is not the case in $q; \pi$. The difference lies essentially in the ability of the action $a!$ in p to be in parallel with future computation, which is here represented by the action π.

This possibility of termination of the main process in combination with non-termination of forked processes is one of the key-characteristics in FC in comparison with CCS, where once a process has terminated, everything it has created has also terminated. We are now able to give the following formal definition of the *process congruence* $\equiv \subseteq \mathcal{L} \times \mathcal{L}$:

$$p \equiv q \Leftrightarrow \mathit{Config}[p] \smile \mathit{Config}[q]$$

Surely $\equiv$ is an equivalence, and it is also a congruence (preserved by the operators of FC) as stated in the following theorem.

Theorem 2.1 (Congruence Property) *Assume for two processes p_1, p_2 that $p_1 \equiv p_2$, and that for the processes p_i, p_i' ($i \in I$ for some index set I) that $p_i \equiv p_i'$. Then for any process q: $\sum_i \alpha_i; p_i \equiv \sum_i \alpha_i; p_i'$, $q; p_1 \equiv q; p_2$, $p_1; q \equiv p_2; q$, $\boldsymbol{fork}(p_1) \equiv \boldsymbol{fork}(p_2)$ and $(a)p_1 \equiv (a)p_2$.*

3 A Refinement Logic

In this section we introduce a refinement logic for FC. That is, a logic which includes programming constructs as well as specification constructs. The programming constructs are those of FC, while the specification constructs are those of the modal μ-calculus [Koz82], or equivalently: Hennessy-Milner logic [HM85] with recursion [Lar90].

3.1 Syntax and Semantics

The syntax of the logic is as follows:

$$\psi \;\;::=\;\; \sum_i \alpha_i; \psi_i \mid \psi_1; \psi_2 \mid \boldsymbol{fork}(\psi) \mid (a)\psi \mid \mathbf{tt} \mid \psi_1 \,\wedge\, \psi_2 \mid \neg\psi \mid \langle\alpha\rangle\psi \mid \nu\, x \cdot \psi \mid x$$

For recursive formulae $\nu\, x \cdot \psi$ we shall assume that any free occurrence of x within ψ is under the scope of an even number of negations (this will ensure monotonicity of ψ). Given a formula ψ in this logic and given a FC program p, we shall formally define what it means for the program to satisfy the formula, which we write as $p \models \psi$. Let us first give an informal description.

The first four alternatives defining ψ are just the (non-recursive) operators of FC. Suppose Op is one of these operators (actions α are part of the choice operator) and suppose $\psi_1, \ldots, \psi_n$ are formulae, then $p \models Op(\psi_1, \ldots, \psi_n)$ if $p \equiv Op(q_1, \ldots, q_n)$ for some processes $q_1, \ldots, q_n$ such that $q_i \models \psi_i$ for $i \in \{1, \ldots, n\}$.

The remaining alternatives are the logic constructs of Hennessy-Milner-Logic with recursion. These include the truth $\mathbf{tt}$ which any process satisfies, and conjunction and negation with the obvious meanings. The formula $\langle\alpha\rangle\psi$ is satisfied by any process that can perform an α-action (as well as possibly other actions) and then become a process that satisfies ψ. Finally, the maximal fixpoint $\nu\, x \cdot \psi$ provides the basic mechanism for recursion. Note that there is no need for a special $\boldsymbol{fix}\, x \cdot \psi$ construct for writing recursive programs. The $\nu\, x \cdot \psi$ serves this purpose as well.

We let Ψ_o stand for all the formulae generated by the above syntax, including formulae with free variables. The set of closed formulae is denoted by Ψ.

Concerning the semantics of formulae, a formula denotes a set of processes. The semantics is defined as follows. An environment is a function $\rho : Env = X \rightarrow \mathcal{P}(\mathcal{L})$ from variables to sets of processes. For an environment ρ, variable x and a process set $S \subseteq \mathcal{L}$ we use $\rho[S/x]$ to mean ρ updated to have the value S at x. The denotation of a formula ψ is a function $[\![\psi]\!] : Env \rightarrow \mathcal{P}(\mathcal{L})$ defined structurally as follows.

$$\begin{array}{rcl}
[\![\mathbf{tt}]\!]\rho & = & \mathcal{L} \\
[\![\psi_1 \wedge \psi_2]\!]\rho & = & [\![\psi_1]\!]\rho \cap [\![\psi_2]\!]\rho \\
[\![\neg\psi]\!]\rho & = & \mathcal{L} - [\![\psi]\!]\rho \\
[\![\langle\alpha\rangle\psi]\!]\rho & = & \{p \in \mathcal{L} \mid \exists C \in Con \cdot Config[p] \xrightarrow{\alpha} C \wedge C \models_\rho \psi\} \\
[\![\nu\, x \cdot \psi]\!]\rho & = & \bigcup\{S \subseteq \mathcal{L} \mid S \subseteq [\![\psi]\!]\rho[S/x]\} \\
[\![x]\!]\rho & = & \rho(x) \\
[\![Op(\psi_1, \ldots, \psi_n)]\!]\rho & = & \{p \in \mathcal{L} \mid \exists q_1, \ldots, q_n \in \mathcal{L} \cdot q_i \in [\![\psi_i]\!]\rho \wedge p \equiv Op(q_1, \ldots, q_n)\}
\end{array}$$

where for any configuration C in Con and for any formula ψ in Ψ_o:

$$C \models_\rho \psi \stackrel{\text{def}}{=} \exists q \in \mathcal{L} \cdot q \in [\![\psi]\!]\rho \wedge C \smile Config[q]$$

A special case of $[\![Op(\psi_1, \ldots, \psi_n)]\!]\rho$ is when Op is the empty choice **nil** (so $n = 0$) in which case the denotation is the set of processes p where $p \equiv \mathbf{nil}$. As a convention, when ψ is closed, we shall let $p \models \psi$ mean that $p \in [\![\psi]\!]\rho$ for any ρ.

The denotation $[\![\nu\, x \cdot \psi]\!]\rho$ is the maximal fixpoint of the function: $F = \lambda S \cdot [\![\psi]\!]\rho[S/x]$. From Tarski's theorem (theorem [Tar55]) we know that this fixpoint exists whenever F is monotonic. That is, for any $S_1, S_2 \subseteq \mathcal{L}$ it must hold that $S_1 \subseteq S_2$ implies $F(S_1) \subseteq F(S_2)$. The operators of the logic are all monotonic, and the existence of the maximal fixpoint is then guaranteed.

We may now define the natural *refinement* (implementation) relation $\Rightarrow \subseteq \Psi \times \Psi$ between formulae of the logic as simply that of logical implication. That is $\psi_1 \Rightarrow \psi_2$ if and only if $\mathcal{L}(\psi_1) \subseteq \mathcal{L}(\psi_2)$, where $\mathcal{L}(\psi)$ stand for the programs satisfying ψ, i.e. $\mathcal{L}(\psi) \stackrel{\text{def}}{=} \{p \in \mathcal{L} \mid p \models \psi\}$. In section 4 we shall provide a proof system for proving statements of the form $\psi_1 \Rightarrow \psi_2$.

3.2 Properties of the Logic

In this section we state some properties of the satisfaction relation $\models$ which may increase our confidence in its definition. Note first, that our logic Ψ_o (regarded as a set of formulae) in a sense to be defined contains the programming language $\mathcal{L}_o$ as a subset. That is, we define a mapping $\varphi[\_] : \mathcal{L}_o \rightarrow \Psi_o$ as follows:

$$\begin{array}{rcl}
\varphi[\boldsymbol{fix}\, x \cdot p] & = & \nu\, x \cdot \varphi[p] \\
\varphi[x] & = & x \\
\varphi[Op(p_1, \ldots, p_n)] & = & Op(\varphi[p_1], \ldots, \varphi[p_n])
\end{array}$$

The first theorem we state says that equivalent processes satisfy each other.

Theorem 3.1 (Equivalence and satisfaction) *For any processes* $p, q \in \mathcal{L}$,

$$p \models \varphi[q] \quad \textit{iff} \quad p \equiv q$$

The proof of this theorem is surprisingly involved. Similar theorems have been given in [GS86] and [LT88]. The following theorem states that the refinement logic is adequate with respect to process congruence $\equiv$. That is, two processes are equivalent, if and only if they satisfy the same formulae. Let for any process $p \in \mathcal{L}$, $\Psi(p)$ be defined as the set of properties that p satisfies: $\Psi(p) = \{\psi \in \Psi \mid p \models \psi\}$. Then we can state adequateness as follows:

Theorem 3.2 (Adequateness wrt. $\equiv$) *For any processes* $p, q \in \mathcal{L}$,

$$\Psi(p) = \Psi(q) \quad \textit{iff} \quad p \equiv q$$

The last theorem states a compositionality result which is the basis for a practical proof system. It says that components of a system can be replaced by refinements, thereby obtaining a refinement of the system.

Theorem 3.3 (Compositionality) *For any formulae* $\psi_1, \psi_1', \ldots, \psi_n, \psi_n'$ *and for any operator* $Op \in \{\sum_i \alpha_i;\_\,,\ \_;\_\,,\ \textbf{\textit{fork}}(\_)\,,\ (a)\_\}$ *with arity* n,

$$\begin{array}{ll} \textit{Whenever} & \psi_1 \Rightarrow \psi_1' \textit{ and } \ldots \textit{ and } \psi_n \Rightarrow \psi_n' \\ \textit{then} & Op(\psi_1, \ldots, \psi_n) \Rightarrow Op(\psi_1', \ldots, \psi_n') \end{array}$$

3.3 Derived Forms

We shall define a set of derived forms of formulae, that are useful for writing examples. The following derived forms are the obvious ones to define first: $\mathbf{ff} \stackrel{\text{def}}{=} \neg\mathbf{tt}$, $\psi_1 \vee \psi_2 \stackrel{\text{def}}{=} \neg(\neg\psi_1 \wedge \neg\psi_2)$, $[\alpha]\psi \stackrel{\text{def}}{=} \neg\langle\alpha\rangle\neg\psi$ and $\mu\, x \cdot \psi[x] \stackrel{\text{def}}{=} \neg\nu\, x \cdot \neg\psi[\neg x]$. We shall often assume some fixed finite set $\mathcal{A}$ of actions. Typically it will be the external actions (in contrast to internal actions on internal channels) of a given process under examination. Then we shall use the following two shorthands, for any subset $A = \{\alpha_1, \ldots, \alpha_n\}$ of $\mathcal{A}$:

$$\begin{array}{lcl} \langle A\rangle\psi & \stackrel{\text{def}}{=} & \langle\alpha_1\rangle\psi \vee \ \ldots \ \vee \langle\alpha_n\rangle\psi \\ {[A]}\psi & \stackrel{\text{def}}{=} & [\alpha_1]\psi \wedge \ \ldots \ \wedge [\alpha_n]\psi \end{array}$$

We shall finally define the following derived forms, some of which correspond to classical temporal modalities. These forms are heavely used in the example in section 5. Let $A \subseteq \mathcal{A}$ be a finite set of actions:

$$\ll A \gg \ \stackrel{\text{def}}{=} \ \langle A\rangle\mathbf{tt} \wedge [\mathcal{A} - A]\mathbf{ff}$$

$$\begin{array}{lcllcl} \Box\psi & \stackrel{\text{def}}{=} & \nu\, X \cdot \psi \wedge [\mathcal{A}]X & \circ\{A\} & \stackrel{\text{def}}{=} & \mu\, X \cdot \ll A, \tau \gg \wedge [\tau]X \\ \Diamond\,\psi & \stackrel{\text{def}}{=} & \mu\, X \cdot \psi \vee (\ll\tau\gg \wedge [\tau]X) & \circ_w\{A\} & \stackrel{\text{def}}{=} & \nu\, X \cdot \ll A, \tau \gg \wedge [\tau]X \end{array}$$

$p \models \ll A \gg$ if p can execute at least one of the actions in the set A, and it cannot execute actions outside A. $p \models \Box\psi$ if p at all points in its execution satisfies ψ. $p \models \Diamond\,\psi$ if p after a finite number of τ-actions will reach a state satisfying ψ. $p \models \circ\{A\}$ if p will execute one of the actions in A after a finite number of τ-actions. $p \models \circ_w\{A\}$ if either p performs an infinite sequence of τ-actions, or if it performs one of the actions in A after a finite number of τ-actions.

4 Proof Rules

In this section we illustrate part of a sound proof system for deducing statements of the form $\psi_1 \Rightarrow \psi_2$. We shall use $\psi_1 \Leftrightarrow \psi_2$ as short for $\psi_1 \Rightarrow \psi_2$ and $\psi_2 \Rightarrow \psi_1$. The rules can be divided into three groups. The first group of rules states how pure logic constructs relate to each other and is essentially based on the axiomatization of the modal μ–calculus in [Koz82]. As an example the following rule makes it possible to reason about maximal fixpoints:

$$(\textbf{Maximal}) \quad \frac{\psi' \Rightarrow \psi[\psi'/x]}{\psi' \Rightarrow \nu\, x \cdot \psi}$$

This rule reflects closely the semantics of fixpoints: the denotation of a maximal fixpoint is defined by $[\![\nu\, x \cdot \psi]\!]\rho = \bigcup\{S \subseteq \mathcal{L} \mid S \subseteq [\![\psi]\!]\rho[S/x]\}$. So for any set $S \subseteq \mathcal{L}$, if it holds that $S \subseteq [\![\psi]\!]\rho[S/x]$, then $S \subseteq [\![\nu\, x \cdot \psi]\!]\rho$, since the maximal fixpoint is the union of such S. This is exactly what is reflected in the proof rule, where ψ' represents S.

The second group of rules states how programming constructs relate to each other and is essentially based on a complete axiomatisation of FC, that has been produced in addition to the work presented here. An earlier version of this FC axiomatisation can be found in [HL93]. As an example the following rule expresses the fact that sequential composition is associative:

$$(\textbf{Associative}) \quad (\psi_1; \psi_2); \psi_3 \Leftrightarrow \psi_1; (\psi_2; \psi_3)$$

The third group of rules, presented in figure 1, states how programming constructs relate to logic modalities. Typically, to the left of $\Rightarrow$ we find some programming construct at the outermost level, while on the right hand side of $\Rightarrow$, we find a modal construct at the outermost level.

Of course all the rules of the proof system are sound, which is stated in the following theorem.

Theorem 4.1 *The refinement rules are sound. That is, whenever the refinement rules allow us to deduce $\psi_1 \Rightarrow \psi_2$, for some formulae ψ_1 and ψ_2 in Ψ, then $\mathcal{L}(\psi_1) \subseteq \mathcal{L}(\psi_2)$.*

5 Example

In this section we shall apply the presented refinement calculus to an example. That is, we shall provide a sequence of specifications, each (except the first) postulated to be a refinement of its predecessor, and with the final one being a program in FC. We shall also sketch proofs of the postulated refinements. The example taken is that of a protocol (see figure 2), and for this we will provide an initial specification, a design and a program. The initial specification will be a pure logic formula containing no programming constructs. The design will be a mixture of modal logic and programming constructs, while finally the program will consist solely of programming constructs. In the first section we present the specification, the design and the program. In the next section we prove the correctness.

$$(\mathbf{Fork}\langle\alpha\rangle) \qquad \frac{\psi \Rightarrow \langle\alpha\rangle\psi'}{\boldsymbol{fork}(\psi) \Rightarrow \langle\alpha\rangle\boldsymbol{fork}(\psi')}$$

$$(\mathbf{Fork}[\alpha]) \qquad \frac{\psi \Rightarrow [\alpha]\psi'}{\boldsymbol{fork}(\psi) \Rightarrow [\alpha]\boldsymbol{fork}(\psi')}$$

$$(\mathbf{Sum}\langle\alpha\rangle) \qquad \textstyle\sum_i \alpha_i;\psi_i \Rightarrow \langle\alpha_i\rangle\psi_i$$

$$(\mathbf{Sum}[\alpha]) \qquad \textstyle\sum_i \alpha_i;\psi_i \Rightarrow [\alpha]\bigvee_{i:\alpha=\alpha_i}\psi_i \qquad \textit{([}\alpha\textit{]}\boldsymbol{ff}\textit{ in case } \forall i\cdot\alpha\neq\alpha_i\textit{)}$$

$$(\mathbf{Seq}\langle\alpha\rangle) \qquad \frac{\psi_1 \Rightarrow \langle\alpha\rangle\psi_1'}{\psi_1;\psi_2 \Rightarrow \langle\alpha\rangle(\psi_1';\psi_2)}$$

$$(\mathbf{Seq}_\Phi\langle\alpha\rangle) \qquad \frac{\psi_2 \Rightarrow \langle\alpha\rangle\psi_2'}{\boldsymbol{fork}(\psi_1);\psi_2 \Rightarrow \langle\alpha\rangle(\boldsymbol{fork}(\psi_1);\psi_2')}$$

$$(\mathbf{Seq}_\Phi\langle\tau\rangle) \qquad \frac{\psi_1 \Rightarrow \langle c\rangle\psi_1'\ ,\ \psi_2 \Rightarrow \langle rev(c)\rangle\psi_2'}{\boldsymbol{fork}(\psi_1);\psi_2 \Rightarrow \langle\tau\rangle(\boldsymbol{fork}(\psi_1');\psi_2')}$$

$$(\mathbf{Seq}_\Phi[c]) \qquad \frac{\psi_1 \Rightarrow [c]\psi_1'\ ,\ \psi_2 \Rightarrow [c]\psi_2'}{\boldsymbol{fork}(\psi_1);\psi_2 \Rightarrow [c]((\boldsymbol{fork}(\psi_1');\psi_2)\ \vee\ (\boldsymbol{fork}(\psi_1);\psi_2'))}$$

$$(\mathbf{Seq}_\Phi[\tau]) \qquad \frac{\forall\alpha\cdot(\psi_1 \Rightarrow [\alpha]\psi_1^\alpha\ ,\ \psi_2 \Rightarrow [\alpha]\psi_2^\alpha)}{\boldsymbol{fork}(\psi_1);\psi_2 \Rightarrow [\tau]((\boldsymbol{fork}(\psi_1^\tau);\psi_2)\ \vee\ (\boldsymbol{fork}(\psi_1);\psi_2^\tau)\ \vee\ \bigvee_c(\boldsymbol{fork}(\psi_1^c);\psi_2^{rev(c)}))}$$

$$(\mathbf{Allocate}\langle\alpha\rangle) \qquad \frac{\psi \Rightarrow \langle\alpha\rangle\psi'}{(a)\psi \Rightarrow \langle\alpha\rangle(a)\psi'}\alpha\neq a$$

$$(\mathbf{Allocate}[\alpha]_1) \qquad \frac{\psi \Rightarrow [\alpha]\psi'}{(a)\psi \Rightarrow [\alpha](a)\psi'}$$

$$(\mathbf{Allocate}[\alpha]_2) \qquad (a)\psi \Rightarrow [\alpha]\mathbf{ff} \qquad (\alpha = a)$$

Figure 1: Proof rules relating programming constructs and modalities

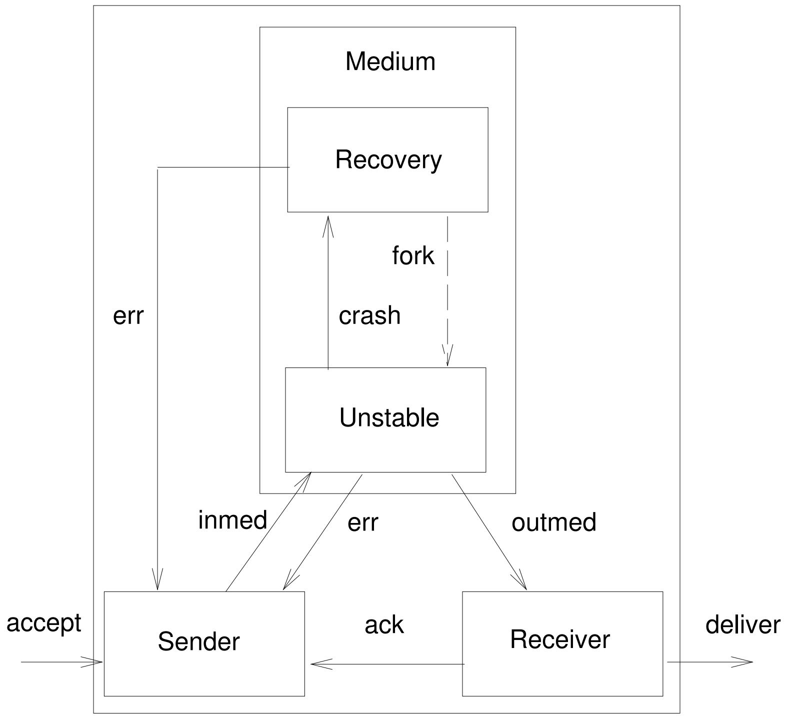

Figure 2: The Protocol

5.1 Specification, Design and Program

The protocol is very simple in that it just transmits messages. There are two actions: *accept*? and *deliver*!, and the behaviour of the protocol is supposed to be an infinite sequence of *accept*? − *deliver*! communications (disregarding the τ-action). The protocol *specification* is as follows:

$$Protocol \stackrel{\text{def}}{=} (\circ\{accept?\}) \wedge (\Box([accept?] \circ_w \{deliver!\})) \wedge (\Box([deliver!] \circ \{accept?\}))$$

The first conjunct says that the next action (after a finite number of τ's) must be *accept*?. The second conjunct says that whenever an *accept*? is performed, then the next action will be *deliver*!, alternatively the protocol may diverge with an infinite number of τ's. This divergence could correspond to the repeated loss of the message by an unreliable medium. Since we later introduce an unreliable medium, we allow divergence at this stage. The final conjunct says that whenever a *deliver*! is performed, then the next action will be *accept*?.

In the next *design*, we implement the protocol as three processes composed in parallel: a sender, a medium and a receiver. The sender accepts a message from the external world by an *accept*?-action and passes it to the medium by an *inmed*!-action, after which it waits for either an acknowledgement, *ack*?, or an error message, *err*?, indicating that the message is lost. If the sender receives an acknowledgement it returns to its initial state, otherwise it tries to resend the message. After the medium has received a message, $inmed?$, it either looses its information which is signaled by the $err!$-action, or the message gets to the receiver by an $outmed!$-action. In both cases the medium returns to its initial state. The receiver receives a message by $outmed?$, delivers it to the external environment by $deliver!$, then it sends an acknowledgement, $ack!$, and returns to its initial state. The protocol design is as follows:

$$
\begin{array}{rcl}
Protocol_D & \stackrel{\text{def}}{=} & (inmed)(outmed)(ack)(err) \\
 & & \boldsymbol{fork}(Receiver); \boldsymbol{fork}(Medium); \boldsymbol{fork}(Sender) \\
Sender & \stackrel{\text{def}}{=} & \nu\, S \cdot accept?; \nu\, S_1 \cdot inmed!; (ack?; S + err?; S_1) \\
Receiver & \stackrel{\text{def}}{=} & \nu\, R \cdot outmed?; deliver!; ack!; R \\
Medium & \stackrel{\text{def}}{=} & (\circ\{inmed?\}) \ \wedge\ (\Box([inmed?] \circ \{outmed!, err!\})) \ \wedge \\
 & & (\Box([outmed!, err!] \circ \{inmed?\}))
\end{array}
$$

The channels $inmed$, $outmed$, ack and err are all local. The sender and the receiver are given as programs in FC, while the medium is underspecified in terms of a formula in pure logic. This is then an example of how programming constructs can be mixed with specification constructs.

In the last step, we implement the medium as two processes composed in parallel: an unreliable medium and a recovery system. The final protocol *program* is illustrated in figure 2. After the unreliable medium has received a message, $inmed?$, it can loose the message which is signaled by the $err!$-action, or the message gets to the receiver by an $outmed!$-action. In both cases the unreliable medium returns to its initial state. A third possibility is that the unreliable medium crashes, which is signaled to the environment by an $crash!$-action. After a crash, the unreliable medium is dead. The recovery system receives the $crash?$ signal, sends an error message, $err!$, to the sender (telling that the message is lost), and finally starts a new unreliable medium. The implementation of the medium is as follows:

$$
\begin{array}{rcl}
Medium_P & \stackrel{\text{def}}{=} & (crash)\boldsymbol{fork}(Unstable); Recovery \\
Unstable & \stackrel{\text{def}}{=} & \nu\, U \cdot inmed?; outmed!; U + err!; U + crash!; \mathbf{nil} \\
Recovery & \stackrel{\text{def}}{=} & \nu\, R \cdot crash?; err!; \boldsymbol{fork}(Unstable); R
\end{array}
$$

We can finally obtain the protocol program by replacing the implementation of the medium for the medium specification in the design (we will not repeat the definitions of the sender and the receiver):

$$
Protocol_P \stackrel{\text{def}}{=} Protocol_D[Medium_P/Medium]
$$

5.2 Proving Correctness

We first prove that $Protocol_D \Rightarrow Protocol$. For this purpose, we shall first examine and describe the phases, or "states", that $Protocol_D$ goes through during execution (there are finitely many such). That is, we identify a set of formulae $\{S_0, \ldots, S_n\}$ such that $Protocol_D \Rightarrow S_0$ and such that for any $i \in \{0, \ldots, n\}$ it holds that $S_i \Rightarrow [\mathcal{A}](S_0 \vee \ldots \vee S_n)$. That is, "no matter what move is taken, we stay within the states $S_0, \ldots, S_n$". For each of these states S_i we shall in addition carefully select a transition property P_i such that $S_i \Rightarrow P_i$. With this aparatus we shall be well prepared when we prove that $Protocol_D \Rightarrow Protocol$.

Before describing the states of $Protocol_D$ we shall first describe and name the states of respectively *Sender*, *Receiver* and *Medium*. Since each of these are sequential (not a parallel composition of several processes), this is just a matter of naming what remains after each action. Concerning the sender, we introduce the auxiliary name $Sender_1$ for the part of *Sender* that follows the action $accept?$:

$$Sender_1 \stackrel{\text{def}}{=} \nu\, S_1 \cdot inmed!; (ack?; Sender + err?; S_1)$$

Then the states of *Sender* are as follows (unfolding maximal fixpoints):

$$\begin{aligned} Accept? &\stackrel{\text{def}}{=} accept?; Inmed! \\ Inmed! &\stackrel{\text{def}}{=} inmed!; Ack?\_Err? \\ Ack?\_Err? &\stackrel{\text{def}}{=} ack?; Sender + err?; Sender_1 \end{aligned}$$

The states of *Receiver* are:

$$\begin{aligned} Outmed? &\stackrel{\text{def}}{=} outmed?; Deliver! \\ Deliver! &\stackrel{\text{def}}{=} deliver!; Ack! \\ Ack! &\stackrel{\text{def}}{=} ack!; Receiver \end{aligned}$$

Finally, the states of *Medium* are:

$$\begin{aligned} Inmed? &\stackrel{\text{def}}{=} (\circ\{inmed?\}) \ \wedge\ (\Box([inmed?] \circ \{outmed!, err!\})) \ \wedge \\ &\quad\ (\Box([outmed!, err!] \circ \{inmed?\})) \\ Outmed!\_Err! &\stackrel{\text{def}}{=} (\circ\{outmed!, err!\}) \ \wedge\ (\Box([inmed?] \circ \{outmed!, err!\})) \ \wedge \\ &\quad\ (\Box([outmed!, err!] \circ \{inmed?\})) \end{aligned}$$

Note that the 'invariance' formulae ($\Box$) are the same in the two states of the medium, while the 'next' formula ($\circ\{\_\}$) is different.

We now compose these states of the individual processes into the states of $Protocol_D$. It has 5 states $S_0 - S_4$, each of the form $(I)\boldsymbol{fork}(R); \boldsymbol{fork}(M); \boldsymbol{fork}(S)$ where (I) stands for $(inmed)(outmed)(ack)(err)$ and where $Protocol_D \Rightarrow S_0$. Below we define R, M and S for each state.

state	R	M	S
S_0	*Outmed?*	*Inmed?*	*Accept?*
S_1	*Outmed?*	*Inmed?*	*Inmed!*
S_2	*Outmed?*	*Outmed!_Err!*	*Ack?_Err?*
S_3	*Deliver!*	*Inmed?*	*Ack?_Err?*
S_4	*Ack!*	*Inmed?*	*Ack?_Err?*

These states are related as presented in the following lemma, which is proved using the proof rules in figure 1.

Lemma 5.1 (Transition Properties for *Protocol$_D$*)
The states $S_0 - S_4$ satisfy the following transition properties:

$$\begin{array}{lcl}
S_0 & \Rightarrow & (\ll\tau, accept?\gg) \wedge ([\tau]S_0) \wedge ([accept?]S_1) \wedge (\circ\{accept?\}) \\
S_1 & \Rightarrow & (\ll\tau\gg) \wedge ([\tau](S_1 \vee S_2)) \\
S_2 & \Rightarrow & (\ll\tau\gg) \wedge ([\tau](S_1 \vee S_2 \vee S_3)) \\
S_3 & \Rightarrow & (\ll\tau, deliver!\gg) \wedge ([\tau]S_3) \wedge ([deliver!]S_4) \\
S_4 & \Rightarrow & (\ll\tau\gg) \wedge ([\tau](S_0 \vee S_4)) \wedge (\Diamond\, S_0)
\end{array}$$

We only explain the first implication: in state S_0 the protocol can perform either a τ-action or an *accept*?-action. If it performs a τ-action it stays in state S_0. If it performs an *accept*?-action, it enters state S_1. Finally, at some moment (after a finite number of τ-actions) an *accept*?-action *will* be performed, so we are guaranteed progress.

To prove that the design refines the specification means to prove that $Protocol_D \Rightarrow$ *Protocol*. That is, we must show that:

$$Protocol_D \Rightarrow (\circ\{accept?\}) \wedge (\Box([accept?] \circ_w \{deliver!\})) \wedge (\Box([deliver!] \circ \{accept?\}))$$

We only show the proof of the second conjunct: $Protocol_D \Rightarrow \Box([accept?] \circ_w \{deliver!\})$. The property to be proved stands for the following maximal fixpoint:

$$\Box([accept?] \circ_w \{deliver!\}) = \nu\, X \cdot \underbrace{[accept?] \circ_w \{deliver!\} \wedge [\mathcal{A}]X}_{F(X)}$$

To show that $Protocol_D \Rightarrow \nu\, X \cdot F(X)$, we show first $S \Rightarrow \nu\, X \cdot F(X)$, where $S = S_0 \vee S_1 \vee S_2 \vee S_3 \vee S_4$, and then since $Protocol_D \Rightarrow S$ we have the result. To show that $S \Rightarrow \nu\, X \cdot F(X)$, we show that $S \Rightarrow F(S)$, and then apply the (**Maximal**) proof rule. So we show that:

$$\begin{array}{l}
(S_0 \vee S_1 \vee S_2 \vee S_3 \vee S_4) \Rightarrow \\
\qquad ([accept?] \circ_w \{deliver!\} \wedge [\mathcal{A}](S_0 \vee S_1 \vee S_2 \vee S_3 \vee S_4))
\end{array}$$

This reduces to showing the following two properties for any $i \in \{0, \ldots, 4\}$:

$$S_i \Rightarrow [accept?] \circ_w \{deliver!\} \quad (1)$$

$$S_i \Rightarrow [\mathcal{A}](S_0 \vee S_1 \vee S_2 \vee S_3 \vee S_4) \quad (2)$$

The properties in group (2) follow immediately from the transition properties in lemma 5.1. Concerning the properties in group (1), we have, again due to lemma 5.1, that for all $i \neq 0$: $S_i \Rightarrow [accept?]\mathbf{ff}$ and thereby that $S_i \Rightarrow [accept?] \circ_w \{deliver!\}$. The last deduction follows from the fact that **ff** refines any formula, and because our operators are monotonic wrt. $\Rightarrow$.

Concerning the proof of $S_0 \Rightarrow [accept?] \circ_w \{deliver!\}$ we proceed as follows. From lemma 5.1 we have that $S_0 \Rightarrow [accept?]S_1$. So obviously $S_0 \Rightarrow [accept?] \circ_w \{deliver!\}$ if $S_1 \Rightarrow \circ_w\{deliver!\}$. The formula $\circ_w\{deliver!\}$ stands for a maximal fixpoint:

$$\circ_w\{deliver!\} = \nu\, X\cdot \ll deliver!, \tau \gg \wedge [\tau]X$$

Since $S_1 \Rightarrow S_1 \vee S_2 \vee S_3$ we have succeeded if we can prove that: $(S_1 \vee S_2 \vee S_3) \Rightarrow \nu\, X\cdot \ll deliver!, \tau\gg \wedge [\tau]X$. Due to the (**Maximal**) rule, we just need to prove that:

$$(S_1 \vee S_2 \vee S_3) \Rightarrow \ll deliver!, \tau\gg \wedge [\tau](S_1 \vee S_2 \vee S_3)$$

We prove each of the cases:

$$\begin{array}{lcl} S_1 & \Rightarrow & \ll deliver!, \tau\gg \wedge [\tau](S_1 \vee S_2 \vee S_3) \\ S_2 & \Rightarrow & \ll deliver!, \tau\gg \wedge [\tau](S_1 \vee S_2 \vee S_3) \\ S_3 & \Rightarrow & \ll deliver!, \tau\gg \wedge [\tau](S_1 \vee S_2 \vee S_3) \end{array}$$

This will hold since by lemma 5.1 we have that:

$$\begin{array}{lcl} S_1 & \Rightarrow & \ll\tau\gg \wedge [\tau](S_1 \vee S_2) \\ S_2 & \Rightarrow & \ll\tau\gg \wedge [\tau](S_1 \vee S_2 \vee S_3) \\ S_3 & \Rightarrow & \ll\tau, deliver!\gg \wedge [\tau]S_3 \end{array}$$

Note that $\ll\tau\gg \Rightarrow \ll deliver!, \tau\gg$.

Finally, to prove that the program refines the design means to prove that $Protocol_P \Rightarrow Protocol_D$. Recall that $Protocol_P$ only differs from $Protocol_D$ in that $Medium_P$ has been substituted for *Medium*. Due to the compositionality of the proof system, it then suffices to show that $Medium_P \Rightarrow Medium$.

6 Concluding Remarks and Related Work

This paper presents two main results: the Fork Calculus FC, equipped with an operational semantics together with an induced congruence; and an associated refinement logic. The Fork Calculus is of interest on its own: it provides a basis for developing theories of programming languages that have ***fork***-like primitives for process creation. One of these languages, CML, was in fact the original inspiration for the design of FC, ([HL93]). It seems that process creation is more frequent than parallel composition (ala CCS) in modern programming languages.

The refinement logic is a step towards a practical framework for specification and refinement of concurrent programs based on message passing and process creation. Other attempts have been made to define refinement logics for process calculi. In [Hol89] as well as in [GS86] such logics have been defined for variants of CCS. Both these attempts have influenced the work presented here. In [Win86] such a logic is defined for SCCS.

References

[GS86] S. Graf and J. Sifakis. A Logic for the Description of Nondeterministic Programs and their Properties. *Information and Control*, 68(1-3):254–270, 1986.

[Hav94] K. Havelund. *The Fork Calculus – Towards a Logic for Concurrent ML*. PhD thesis, Institute for Computer Science – University of Copenhagen (DIKU), March 1994. DIKU technical report 94/4.

[HL93] K. Havelund and K. Larsen. The Fork Calculus. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *20th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 700*, pages 544–557, 1993.

[HL94] K. Havelund and K. Larsen. A Refinement Logic for the Fork Calculus. Technical Report LIX/RR/94/03, Ecole Polytechnique Paris, LIX, 1994.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of ACM*, 32(1):137–161, January 85.

[Hol89] S. Holmström. A Refinement Calculus for Specifications in Hennessy-Milner Logic with Recursion. *Formal Aspects of Computing*, 1:242–272, 1989.

[Koz82] D. Kozen. Results on the Propositional mu-calculus. In *9th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 140*, 1982.

[Lar90] K. G. Larsen. Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *Theoretical Computer Science*, 72:265–288, 1990.

[LT88] K. G. Larsen and B. Thomsen. A Modal Process Logic. Technical Report R 88-5, Aalborg University Center, January 1988.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[PGM90] S. Prasad, A. Giacalone, and P. Mishra. Operational and Algebraic Semantics for Facile. In *17th International Colloquium on Automata, Languages and Programming (ICALP), LNCS 443*, 1990.

[Plo81] G. Plotkin. A Structural Approach to Operational Semantics. FN 19, DAIMI, Aarhus University, Denmark, 1981.

[Rep91] J. H. Reppy. CML: A Higher-order Concurrent Language. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (SIGPLAN Notices 26(6))*, pages 294–305, 1991.

[Tar55] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. Math.*, 5, 1955.

[Win86] G. Winskel. A Complete Proof System for SCCS with Modal Assertions. *Fundamenta Informaticae, North-Holland*, IX:401–419, 1986.