URL: http://www.elsevier.nl/locate/entcs/volume7.html 19 pages

Process Creation and Full Sequential Composition in a Name-Passing Calculus [1]

Thomas Gehrke and Arend Rensink

Institut für Informatik, Universität Hildesheim
Postfach 101363, D–31113 Hildesheim, Germany
email: {gehrke,rensink}@informatik.uni-hildesheim.de
Phone: (+49) 5121 883 734, Fax: (+49) 5121 883 768

Abstract

This paper presents the underlying theory for a process calculus featuring *process creation* and *sequential composition*, instead of the more usual *parallel composition* and *action prefixing*, in a setting where mobility is achieved by communicating channel names. We discuss the questions of scope and name binding, raised by the interaction of mobility and sequential composition. Substitution of names is integrated as a syntactic operator in the calculus. We present an axiomatic theory for the calculus and show its soundness and completeness w.r.t. bisimulation equivalence.

Keywords: Process Algebra, Mobility, Sequential Composition, Process Creation.

To appear in: EXPRESS '97: Expressiveness in Languages for Concurrency, C. Palamidessi and J. Parrow (Eds.)

1 Introduction

Reactive and distributed systems are of increasing importance in theory and practice of computer science. These systems can be described by three characteristics: structure, behaviour and data. For the specification of the first two aspects the formalism of *process algebras* [2,16,18,20] is widely used. Process algebras provide a powerful theory on behavioural preorders and equivalences and allow for formal reasoning on correctness issues, but usually they are weaker on the treatment of data. In order to include the data aspect into system specifications, in the recent years languages like *Concurrent ML* [22], *Facile* [11] and *ProFun* [9] have been developed, which combine the paradigms of process algebras and functional programming languages.

The semantic treatment of such concurrent functional languages is not obvious; some approaches are described in [7,8,24]. In this paper, we investigate a direct process algebraic formalisation: we present a calculus that can be

[1] Research partially supported by the HCM network *EXPRESS* (Expressiveness of Languages for Concurrency); first author supported by the DFG grant *EREAS* (Entwurf reaktiver Systeme).

©1997 Published by Elsevier Science B. V.

used as a semantic foundation for the language *ProFun*. This calculus has to deal with higher-order features, because *ProFun* (like *CML* and *Facile*) allows dynamic change of the linkage structure of systems. For this purpose, adopting the ideas of the π-calculus (see [20]), we define a communication mechanism where in particular *channel names* are passed in communication actions. Sangiorgi has shown that this provides all the necessary expressive power for higher-order programming [23].

In process algebras like the π-calculus, concurrency is usually realised by a binary operator $t|u$, which represents the parallel composition of the processes t and u. On the other hand, the concurrent functional languages mentioned above rather rely on a (unary) operator to create or *spawn* a new process, which then runs concurrently to the remainder of the program.

Process creation is used in combination with an operator for the *sequential composition* of subprograms, which again is in contrast to (in fact, a generalisation of) the *action prefix* operator seen in most process algebras. It turns out that especially the combination of communication and sequential composition introduces nontrivial questions of *variable scope*, which we solve in this paper by distinguishing between the binding and scoping of variables.

Independent interest in sequential composition exists from the area of *action refinement*; see, e.g., [12,13]. Action refinement allows for the stepwise construction of reactive systems. Single communication actions are replaced by process terms, which describe the behaviour of these actions in more detail. The notion of action refinement, in its syntactic interpretation as substitution within terms, calls for sequential composition rather than action prefixing. For example, if in a term $a.b.\mathbf{0}$, the action a should be refined by a term t, there is no obvious way to denote the resulting behaviour $t.b.\mathbf{0}$ without resorting to sequential composition.

The interaction of process creation and sequential composition in the setting of process algebra has been studied before by Baeten and Vaandrager in [3] and by Havelund and Larsen in [14,15]. Only the latter address higher order features as well, also through name passing. Their solution to the scoping problem, however, is quite restrictive, since they essentially return to action prefixing for input actions, which implies all terms that raise scoping questions are *a priori* ruled out. A more detailed comparison of the various approaches is given in the conclusions.

The remainder of this paper is structured as follows: first we introduce in Section 2 the *basic calculus*, containing communication but no mobility; we give an operational semantics and a complete axiomatisation for finite terms. The *full calculus*, extending the basic calculus with the possibility to communicate channel names, is presented in Section 3, again with operational and axiomatic treatment. As was to be expected, the axiomatics of the full calculus are more involved than for the basic case; in fact, we encounter some well-known problems from the π-calculus. Finally, Section 4 compares the approaches mentioned above and contains some concluding remarks. The proofs of the theorems and propositions can be found in the full version of this paper [10].

2 The Basic Calculus

In this chapter we introduce the basic calculus. In contrast to the full calculus, it does not allow parameter passing during communication or process invocation.

Syntax and operational semantics.

Similar to *CCS* [18] we consider communication to be a synchronous action between two processes which can perform corresponding communication actions. We assume a countable set $\mathcal{C}$ of channel names, ranged over by a,b,c. A channel a can be used either for *input*, denoted $a?$, or for *output*, denoted $a!$. We sometimes use '†' to as a "metavariable" denoting either '!' or '?'. The set of communication actions is denoted $\mathcal{A} = \{a\dagger \mid a \in \mathcal{C}\}$, ranged over by α, β. We represent internal behaviour by $\tau = \iota!$, where $\iota \in \mathcal{C}$ is a special channel which may not otherwise occur. $\mathcal{N}$, ranged over by n, n', is a set of process names. The basic calculus of this section, $\mathcal{B}$, ranged over by t, u, v, is defined through the following grammar:

$$t ::= \mathbf{1} \mid g \mid spawn(t) \mid (a:t) \mid t;t \ .$$
$$g ::= \mathbf{0} \mid g+g \mid (a:g) \mid g;t \mid t;g \mid n \mid \alpha \ .$$

We distinguish between *guarded terms* (g) and *non-guarded terms* (t), where the former start with an action before they may terminate. $\mathbf{1}$ denotes a successfully terminated term, $\mathbf{0}$ the inactive process. $spawn(t)$ creates a new process which performs t concurrently to the spawning term. $(a:t)$ restricts the execution of t to the actions in $\mathcal{A}\backslash\{a?, a!\}$. $+$ is the guarded choice operator, which is resolved by the execution of one of its alternatives. $t;u$ denotes the sequential composition of t and u, i.e., u can perform actions when t has terminated. A process name $n \in \mathcal{N}$ is interpreted by a function $\Theta : \mathcal{N} \to \mathcal{B}$: n denotes a process call of $\Theta(n)$. The unfolding of the definition will accompanied by an internal action, hence such a call is guarded. For syntactical convenience we assume that ; has a higher priority than $+$; for instance, $a! + b!; c!$ is $a! + (b!; c!)$. Furthermore, we assume sequential composition to be right associative, i.e. $t; u; v$ is $t; (u; v)$.

Now we define the operational semantics of the basic calculus. For this purpose, we use the general notion of a *labelled transition system* [18], extended with a predicate to denote the *successful termination* of a state.

Definition 2.1 *A labelled ✓-transition system is a tuple $\langle \mathcal{L}, S, \to, \checkmark \rangle$ where*

- $\mathcal{L}$ *is a* label set*;*
- S *is a set of* states*;*
- $\to \subseteq S \times \mathcal{L} \times S$ *is a* transition relation*, whose elements are denoted $s \xrightarrow{\ell} s'$;*
- $\checkmark \subseteq S$ *is a* termination predicate*, such that $s \in \checkmark$ and $s \xrightarrow{\ell} s'$ implies $s' \in \checkmark$.*

Transition systems are ranged over by T, U. For the calculus presented above, we have $\mathcal{L} = \mathcal{A}$ and $S = \mathcal{B}$. The termination and transition predicates are defined through operational rules, in Figure 1.

$$\frac{}{\mathbf{1}\checkmark}\,\mathrm{T}_1 \qquad \frac{}{spawn(t)\checkmark}\,\mathrm{T}_2 \qquad \frac{t\checkmark \quad u\checkmark}{(t;u)\checkmark}\,\mathrm{T}_3 \qquad \frac{t\checkmark}{(a:t)\checkmark}\,\mathrm{T}_4$$

$$\frac{}{\alpha \xrightarrow{\alpha} \mathbf{1}}\,\mathrm{R}_1 \qquad \frac{}{n \xrightarrow{\tau} \Theta(n)}\,\mathrm{R}_2 \qquad \frac{t \xrightarrow{\alpha} t'}{t+u \xrightarrow{\alpha} t'}\,\mathrm{R}_3 \qquad \frac{u \xrightarrow{\alpha} u'}{t+u \xrightarrow{\alpha} u'}\,\mathrm{R}_4$$

$$\frac{t \xrightarrow{a\dagger} t' \quad a \neq b}{(b:t) \xrightarrow{a\dagger} (b:t')}\,\mathrm{R}_5 \qquad \frac{t \xrightarrow{\alpha} t'}{spawn(t) \xrightarrow{\alpha} spawn(t')}\,\mathrm{R}_6$$

$$\frac{t \xrightarrow{\alpha} t'}{t;u \xrightarrow{\alpha} t';u}\,\mathrm{R}_7 \qquad \frac{t\checkmark \quad u \xrightarrow{\alpha} u'}{t;u \xrightarrow{\alpha} t;u'}\,\mathrm{R}_8 \qquad \frac{t\checkmark \quad t \xrightarrow{\alpha} t' \quad u \xrightarrow{\beta} u' \quad \{\alpha,\beta\} = \{a!,a?\}}{t;u \xrightarrow{\tau} t';u'}\,\mathrm{R}_9$$

Fig. 1. Transition rules for the basic calculus.

The termination predicate extends the usual notion, in that terminated terms may at the same time still perform actions, namely if they are *spawn*ed off as parallel processes. (A similar approach is seen in [3].) Note that we need no rule for the termination of choice, since the restriction to *guarded* choice guarantees that in $t + u$, neither t nor u can be terminated. This simplifies matters greatly and is, in fact, precisely the reason for the restriction to guarded choice. Since there appears to be growing consensus that guarded choice suffices in practical applications of process calculi, our restriction seems quite reasonable.

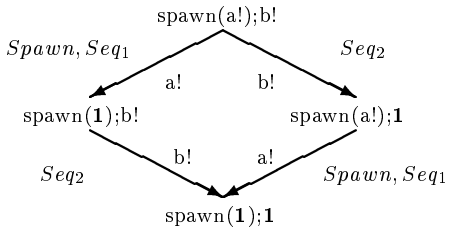
With respect to sequential composition, the standard operational rules are as follows (cf. [2]):

$$\frac{t \xrightarrow{\alpha} t'}{t;u \xrightarrow{\alpha} t';u} \qquad \frac{t\checkmark \quad u \xrightarrow{\alpha} u'}{t;u \xrightarrow{\alpha} u'}$$

In our setup, the first rule is fine but the second one is not, since it discards the first operand. In the case where the first operand equals $spawn(t)$ for some t, this is not the desired behaviour; rather, $spawn(t)$ should still be there in the target term. In general, if the first operand is terminated, the sequential composition behaves very much like standard *parallel* composition. This is indeed our intuition; in fact we also allow *communication* between $spawn(t)$ and u in $spawn(t);u$.

Apart from sequential composition, there is only one unusual rule in our semantics, namely the one for recursion, which specifies an internal step to perform the unfolding of a process call into its body. Our new approach to sequential composition is realised in the rule R_6 for *spawn* and the three rules R_7, R_8 and R_9 for sequential composition. In particular, R_9 expresses communication. If a process term t is terminated, but may also perform an action α, it is clear that t must contain a term of the form $spawn(t'')$. If u is able to perform the corresponding action β such that $\{\alpha,\beta\} = \{a!,a?\}$, implying that α and β specify input and output over the same channel, communication

is possible. Consider the following example:



The operational semantics generates a ✓-transition system. In particular, the condition regarding the persistency of termination is satisfied.

Proposition 2.2 *$\langle \mathcal{A}, \mathcal{B}, \rightarrow, \checkmark \rangle$ is a ✓-transition system.*

Bisimulation and Axiomatisation.

We define a notion of process equivalence which is based on the concept of *bisimulation* [18]. Two processes are called *bisimilar* if it is not possible for an external observer to distinguish between their behaviours. We will treat the internal action τ just like any other action. This leads to a rather strict equivalence which is called *strong bisimulation.*

Definition 2.3 *Let T be a ✓-transition system. A symmetrical relation $R \subseteq S \times S$ is called a* bisimulation relation *if for all $(s_1, s_2) \in R$*

- *if $s_1 \xrightarrow{\ell} s_1'$ then $\exists s_2 \xrightarrow{\ell} s_2'$ such that $(s_1', s_2') \in R$;*
- *if $s_1\checkmark$ then $s_2\checkmark$.*

$s_1, s_2 \in S$ are called bisimilar*, denoted $s_1 \sim_{\mathcal{B}} s_2$, if $(s_1, s_2) \in R$ for some bisimulation relation R.*

The only non-standard part is the condition on the termination predicates, which is necessary to ensure congruence. Without this condition, we would have $spawn(a!) \sim_{\mathcal{B}} a!$, but these terms generate different behaviour in the context of sequential composition: for instance, $spawn(a!); b! \xrightarrow{b!} spawn(a!); \mathbf{1}$ whereas $a!; b! \not\xrightarrow{b!}$.

In particular, $t \sim_{\mathcal{B}} u$ iff the corresponding ✓-transition systems (see Proposition 2.2) are bisimilar. We establish that $\sim_{\mathcal{B}}$ is a congruence w.r.t. the operators of our calculus. For this purpose, rather than giving a direct proof, we derive the result from existing meta-theory. It has been shown that, if the rules of the operational semantics are compatible with certain *formats*, we are able to establish properties of this semantics. A typical property which can be proved in this way is the congruence of equivalence relations [5,17]. Because of the occurrence of the predicate ✓ in our rules we have to use the *path* format [1] which allows the use of predicates. It is easy to verify that all the rules in Figure 1 satisfy the conditions for the *path* format, so (strong) bisimulation is a congruence for our calculus.

Theorem 2.4 *$\sim_{\mathcal{B}}$ is a congruence over the basic calculus.*

In Figure 2, we give a set $\mathcal{A}X_{\mathcal{B}}$ of axioms for the axiomatisation of $\sim_{\mathcal{B}}$. Examples for derived equations are given in Figure 3.

$$\mathbf{1};t = t \quad (1) \qquad t;\mathbf{1} = t \quad (2) \qquad \mathbf{0};t = \mathbf{0} \quad (3)$$

$$t;(u;v) = (t;u);v \quad (4) \qquad t+\mathbf{0} = t \quad (5) \qquad t+u = u+t \quad (6)$$

$$t+t = t \quad (7) \qquad (t+u)+v = t+(u+v) \quad (8) \qquad (t+u);v = t;v+u;v \quad (9)$$

$$n = \tau;\Theta(n) \quad (10) \qquad (a:\mathbf{0}) = \mathbf{0} \quad (11) \qquad (a:\mathbf{1}) = \mathbf{1} \quad (12)$$

$$(a:(a:t)) = (a:t) \quad (13) \qquad (a:(b:t)) = (b:(a:t)) \quad (14)$$

$$(a:a\dagger;t) = \mathbf{0} \quad (15) \qquad (a:b\dagger;t) = b\dagger;(a:t) \quad \text{if } a \neq b \quad (16)$$

$$(a:t+u) = (a:t)+(a:u) \quad (17) \qquad (a:spawn(t)) = spawn((a:t)) \quad (18)$$

$$spawn(\mathbf{0}) = \mathbf{1} \quad (19)$$

$$spawn(t);spawn(u) = spawn(u);spawn(t) \quad (20)$$

$$spawn(t);spawn(u) = spawn(spawn(t);u) \quad (21)$$

$$spawn(t;spawn(u)) = spawn(t;u) \quad (22)$$

$$spawn((t;spawn(u))+v) = spawn((t;u)+v) \quad (23)$$

if $t \equiv \sum_{i\in\mathcal{I}} \alpha_i;t_i$ and $u \equiv \sum_{k\in\mathcal{K}} \beta_k;u_k$ then (24)

$$spawn(t);u = \sum_{i\in\mathcal{I}} \alpha_i;spawn(t_i);u + \sum_{k\in\mathcal{K}} \beta_k;spawn(t);u_k + \sum_{\substack{i\in\mathcal{I}\\ k\in\mathcal{K}\\ \{\alpha_i,\beta_k\}=\{a?,a!\}}} \tau;spawn(t_i);u_k$$

Fig. 2. Axioms of the basic calculus.

$$spawn(\mathbf{1}) = \mathbf{1} \quad (25)$$

$$spawn(spawn(t)) = spawn(t) \quad (26)$$

Fig. 3. Derived equations.

Theorem 2.5 *The theory $\mathcal{AX}_{\mathcal{B}}$ is sound with respect to $\sim_{\mathcal{B}}$.*

We denote the finite fragment of $\mathcal{B}$, i.e., without the recursion operator, by $\mathcal{B}_{fin}$. For the proof of completeness it is useful to define *normal forms* of terms. Therefore, we use the sum notation for a more concise representation of choice operators: if $I = \{i_1, \ldots, i_n\}$ then $\sum_{i\in I} t_i = t_{i_1} + \ldots + t_{i_n}$, where $\sum_{i\in\emptyset} t_i$ equals $\mathbf{0}$ and $\sum_{i\in\{x\}} t_i$ equals t_x. This is a valid notation because of Eqs. (5)–(8).

Definition 2.6 *A term $t \in \mathcal{B}_{fin}$ is in* basic normal form *if t is a term in N:*

$$N ::= \sum_{i\in\mathcal{I}} \alpha_i;N_i \mid spawn(B) \qquad B ::= \sum_{i\in\mathcal{I}} \alpha_i;B_i$$

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

Normal form terms do not contain nested *spawn* applications (hence, for instance, the term $spawn(a!; spawn(b!) + c!; spawn(d!)); e!$ is not in normal form), use only action prefix and no restriction operators. This means that basic normal form terms abstract from the individual *spawn*-applications and just describe the possible interleaving sequences of actions a term can perform. For instance, a term α has the basic normal form $\alpha; spawn(\mathbf{0})$. For example, by the expansion law (Equation (24)) and other axioms it can be deduced that

$$spawn(a!; spawn(b!) + c!; spawn(d!)); c? = a!; (b!; c? + c?; spawn(b!)) \\ + c!; (d!; c? + c?; spawn(d!)) + c?; spawn(a!; b! + c!; d!) + \tau; spawn(d!)$$

Theorem 2.7 *For all $t \in \mathcal{B}_{fin}$, there is an $u \in \mathcal{B}_{fin}$ in basic normal form such that $t \sim_{\mathcal{B}} u$.*

With the existence of a normal form of each term $t \in \mathcal{B}_{fin}$, we can deduce a completeness result of $\mathcal{A}X_{\mathcal{B}}$ for finite process terms, i.e., terms not containing process calls.

Theorem 2.8 *The theory $\mathcal{A}X_{\mathcal{B}}$ is complete for $\sim_{\mathcal{B}}$ on the finite fragment $\mathcal{B}_{fin}$ of $\mathcal{B}$.*

3 The Full Calculus

We now extend the basic calculus with mobility in the fashion of the π-calculus. It turns out that due to the presence of sequential composition in the language, some of the assumptions underlying the π-calculus have to be reconsidered.

The basic question is one of binding and scope. For instance, in the term $t_1 = (x?y; y!z); z!y$, the second occurrence of y is clearly bound by the first; but what about the third? Since we want to preserve associativity of sequential composition, the answer is immediate: in $x?y; (y!z; z!y)$, both of the latter y's are bound by the first, hence this must be the case in t_1 as well.

As a further step, consider $t_2 = (x?z + y?z); z!a$. Here, it is not uniquely determined what the binding occurrence of the latter z is; depending on how the choice is resolved, it could be either of the first two z's. One might argue that terms with this property should be disallowed; however, we feel that t_2 is a typical example why sequential composition is considered practically useful. Since sequential composition right-distributes over choice, t_2 is equivalent to $t_2' = x?z; z!a + y?z; z!a$; however, t_2' does not immediately convey the fact that the choice operands differ only in the first action. In fact, it turns out that terms like t_2 pose no essential complication in the theory.

In terms such as $t_3 = (x?y + x?z); z!a$, the question whether the second z is bound at all appears to depend on the resolution of the choice. However, the property of unique binding (a variable receives a value only once) is necessary for a smooth formalisation of the semantics; therefore every variable should be either bound or free in a given term. For this reason we say that in the left hand operand of the subterm $x?y + x?z$ of t_3, z is implicitly bound, viz. to itself.

A further complication, also due to sequential composition, lies in the notion of syntactic substitution, which is the basic mechanism for replacing variables by values in the π-calculus. In our calculus, the scope of a bound variable is in general unlimited, except when explicitly restricted. For instance, $x?y$ binds y in all subsequent subterms; as long as no explicit scope restriction is given, it will always be possible to sequentially append further y-containing terms. This is in contrast to the action prefix term $x?y;t$, where y is only valid within the given term t. As a consequence, in our calculus it is not immediately clear where to apply substitution.

Restriction.

We solve these problems by distinguishing between variable binding and scoping. A variable x can be bound explicitly through a receive action $(a?x)$, or implicitly, corresponding to the dynamic generation of a new channel. Scope restriction is denoted $(x : t)$ as before. As mentioned above, we keep to the declarative principle that a variable is bound, i.e., receives a value, only once in its lifetime. On the other hand, it is also restricted only once in its lifetime.

The operator $(x : t)$ has twofold effect. First, it restricts the scope of x to the term t. Second, it influences the syntax of the context of t, because in a term $t' = u;(x : t);v$ the name x must not occur in u or v; otherwise the t' would not be *well-formed* (see below).

With respect to the scoping aspects of restriction, a phenomenon occurs in the operational semantics that is known from the π-calculus: the scope of a channel name can change during the lifetime of a system. The situation that a channel name becomes known outside its original scope is called *scope extrusion*. It is reflected by a syntactic change: the restriction operator disappears, and is subsequently reapplied to a super-term of its original operand. For a proper treatment of this phenomenon, we rely on a notion of *restricted names* (corresponding to the π-calculus' *bound* names). For every channel name $a \in \mathcal{C}$, we assume a restricted name $\tilde{a}$. We define $\tilde{\mathcal{C}} = \{\tilde{a} \mid a \in \mathcal{C}\}$, and use χ, ξ to range over $\mathcal{C} \cup \tilde{\mathcal{C}}$. For every $\Xi \subseteq \mathcal{C} \cup \tilde{\mathcal{C}}$, combining restricted and unrestricted channels, $r_\Xi = \{a \mid \tilde{a} \in \Xi\}$ denotes the "restriction content" of Ξ, and $c_\Xi = (\Xi \cap \mathcal{C}) \cup r_\Xi$ denotes the original channel names in Ξ.

Summarising, we have three kinds of variable occurrence: variables can be free (visible and unassigned), bound (visible, but with an assigned, though as yet unknown, value) and restricted (invisible). Binding occurs *implicitly* when restricting a non-bound variable, and when specifying a choice between operands with distinct sets of bound variables. Implicit binding always generates a *fresh* value, which is syntactically indicated by the variable name itself.

Syntactic substitution.

We need a way to connect concrete values to variables. Again following the ideas of the π-calculus, we use a notion of substitution for this purpose. However, as discussed above, the scoping aspects of sequential composition require a more sophisticated approach than the case of action prefixing; in

fact, we need a form of "delayed" substitution, which is stored for future use. For this purpose, we introduce substitutions as part of the syntax of our calculus.

In general, substitution will be finite sets $\sigma = \{x_1 \leftarrow \xi_1, \ldots, x_m \leftarrow \xi_m\}$, where $x_i \in \mathcal{C}$ and $\xi_i \in \mathcal{C} \cup \tilde{\mathcal{C}}$ such that $x_i \neq c_{\xi_i}$, $x_i \neq x_j$ and $c_{\xi_i} \neq x_j$ for all distinct i, j (hence σ is one-to-one with disjoint domain and range), and $c_{\xi_i} = c_{\xi_j}$ implies $\xi_i = \xi_j$ (hence images with the same channel name have the same restriction content). We write $dom\,\sigma = \{x_1, \ldots, x_m\}$ for the domain of σ, $rng\,\sigma = \{\xi_1, \ldots, \xi_m\}$ for its range, $\sigma(x_i) = \xi_i$ for all $1 \leq i \leq m$ (which is well-defined due to the above requirements on σ) and $\sigma(x) = x$ for all $x \notin dom\,\sigma$ (hence σ may be considered as a function $\mathcal{C} \to (\mathcal{C} \cup \tilde{\mathcal{C}})$). The class of substitutions is denoted $\mathbf{S}$.

A substitution σ indicates that all $x \in dom\,\sigma$ are to be bound to the corresponding channel $c_{\sigma(x)}$, while at the same time $r_{\sigma(x)}$ is to be restricted, to deal with scope extrusion. We define the following constructions on substitutions:

$$\begin{aligned} \sigma \uparrow a &= \{(x, a) \mid (x, \tilde{a}) \in \sigma\} \cup \{(x, \xi) \in \sigma \mid a \neq c_\xi\} \\ \sigma \downarrow a &= \{(x, \tilde{a}) \mid (x, a) \in \sigma\} \cup \{(x, \xi) \in \sigma \mid x \neq a \neq c_\xi\} \\ \sigma_1 \circ \sigma_2 &= \{(x, \sigma_1(\xi)) \mid (x, \xi) \in \sigma_2\} \cup \{(x, \xi) \in \sigma_1 \mid x \notin dom\,\sigma_2\} \end{aligned}$$

$\sigma \uparrow a$ "frees" the image a in σ, i.e., changes it from restricted to unrestricted; the dual construction $\sigma \downarrow a$ changes it into restricted, and also removes a from $dom\,\sigma$ (if it was there). Finally, $\sigma_1 \circ \sigma_2$ is the composition of the substitutions, considered as (partial) functions.

σ is called *free* (that is, non-restricted) if $rng\,\sigma \subseteq \mathcal{C}$. Free substitutions are used as explicit language constants; non-free substitutions only occur as part of the semantics. We sometimes write a free $\sigma = \{x_1 \leftarrow a_1, \ldots, x_m \leftarrow a_m\}$ as $\vec{x} \leftarrow \vec{a}$, where the $\vec{x}$ and $\vec{a}$ represent *vectors* of channels corresponding to $x_1 \cdots x_m$ and $a_1 \cdots a_m$, respectively. $\{\vec{x}\} = \{x_1, \ldots, x_m\}$ denotes the set of elements of the vector $\vec{x}$; the empty vector is denoted ε. We also write $\sigma(\vec{y})$, with the obvious meaning.

The problems of scope and binding are aggravated by the introduction of syntactic substitution, because in combination with restriction and sequential composition it gives rise to a new form of scope extrusion, which could be called *forward extrusion* in contrast to the known *parallel extrusion* through communication. For instance, the term $(a : \{x \leftarrow a\}; t); x!b$ expresses that all x are to be replaced by a, including the last x, which is outside the a-restriction; hence a becomes known outside its scope. Rather than giving $(a : t); \{x \leftarrow a\}; x!b$ as the result of this substitution, we extend the restriction to cover $x!b$, i.e., the result of the substitution is equivalent to $(a : \{x \leftarrow a\}; t; x!b)$.

The full calculus,

denoted $\mathcal{F}$, is generated by the following grammar:

$$\begin{aligned} t &::= \mathbf{1} \mid \sigma \mid g \mid spawn(t) \mid (x : t) \mid t; t \ . \\ g &::= \mathbf{0} \mid g + g \mid (x : g) \mid g; t \mid t; g \mid n(\vec{x}) \mid x!x \mid x?x \mid [x{=}x] \ . \end{aligned}$$

As before, t stands for an arbitrary term and g for a guarded term; x stands for a channel variable, and σ for a free substitution. Note that restricted channel names cannot occur anywhere in the syntax. $[a{=}b]$ corresponds to the *matching* operator of the π-calculus: if $a = b$ then it is equivalent to τ, otherwise to $\mathbf{0}$. We sometimes use $(\vec{x} : t)$ to abbreviate $(x_1 : (x_2 : \cdots (x_m : t) \cdots))$. The process environment Θ is assumed to consist of rules of the form $n(\vec{x}) \mapsto t$, where $\vec{x}$ is a vector of formal parameters. Such rules are *interpreted up to α-conversion*, meaning that the names in $\vec{x}$, as well as other variable names local to t, can be replaced by arbitrary different names. This is necessary because semantically, a process call $n(\vec{a})$ is treated by "inlining" a substitution instance of its body t; this could give rise to a non-well-formed term (see below) if variable names in t cannot be chosen at will. [2]

Well-formed terms.

Not all terms of $\mathcal{F}$ are acceptable; for instance, as discussed above, we want unique binding of variables. More precisely, we want the following informal properties to be satisfied:

- No variable is bound sequentially *after* it occurs free.
- Variables bound within a *spawn* or process definition should be restricted to that scope. This condition ensures the locality of binding.
- Restricted variables may not occur outside their scope.

This is formalised using the concepts of *free*, *bound* and *restricted* variables of a term t, defined in Figure 4 as $fv(t)$, $bv(t)$ and $rv(t)$, respectively. We also use $var(t) = fv(t) \cup bv(t) \cup rv(t)$.

Example 3.1 *Consider the term $t = b?a + a!c$. In the left hand operand, a is bound by communication. With the rules for choice we can deduce that $fv(t) = \{b, c\}$ and $bv(t) = \{a\}$, therefore a is implicitly bound in the right hand operand; its value is assumed to be a itself. a is even bound implicitly in terms like $b?a + c!d$, where it does not occur in the other alternative. Implicit binding also takes place in restriction operators: in $t = (a : x!a)$, the name a is bound implicitly, because it does not occur free in t.*

The purpose of this definition is to restrict the set of allowable terms; in the remainder we will assume that the following conditions are satisfied:

- for all terms $(x : t)$, $x \notin rv(t)$;
- for all terms $t; u$, $rv(t) \cap var(u) = var(t) \cap (bv(u) \cup rv(u)) = \emptyset$;
- for all terms $spawn(t)$, $bv(t) = \emptyset$
- for all definitions $\Theta{:}\, n(\vec{x}) \mapsto t$, $bv(t) = \emptyset$ and $fv(t) \subseteq \{x_1, \ldots, x_m\}$.

The first two conditions ensure that each name is restricted at most once. The third condition demands that each name bound in a spawned process must be restricted. The fourth condition ensures that in a process definition, all

[2] Note that α-conversion is *different* from the notion of substitution regarded here, since the former also replaces restricting and binding occurrences of variables.

t	$fv(t)$	$bv(t)$	$rv(t)$
$\mathbf{0}$	$\emptyset$	$\emptyset$	$\emptyset$
$\mathbf{1}$	$\emptyset$	$\emptyset$	$\emptyset$
σ	$dom\,\sigma \cup rng\,\sigma$	$\emptyset$	$\emptyset$
$n(\vec{a})$	$\{\vec{a}\}$	$\emptyset$	$\emptyset$
$x!y$	$\{x, y\}$	$\emptyset$	$\emptyset$
$x?y$	$\{x\}$	$\{y\}$	$\emptyset$
$[x{=}y]$	$\{x, y\}$	$\emptyset$	$\emptyset$
$u + v$	$(fv(u) \setminus bv(v)) \cup (fv(v) \setminus bv(u))$	$bv(u) \cup bv(v)$	$rv(u) \cup rv(v)$
$(x : u)$	$fv(u) \setminus \{x\}$	$bv(u) \setminus \{x\}$	$rv(u) \cup \{x\}$
$spawn(u)$	$fv(u)$	$\emptyset$	$rv(u)$
$u; v$	$fv(u) \cup (fv(v) \setminus bv(u))$	$bv(u) \cup bv(v)$	$rv(u) \cup rv(v)$

Fig. 4. Free, bound and restricted variables.

bound names have to be restricted as well, and the free names have to be a subset of the parameters. The latter two conditions realise the concept of *locality* known from programming languages, i.e. names should be restricted to the subterm in which they are bound. Terms satisfying the conditions are called *well-formed*. In the remainder of the paper, we implicitly restrict to well-formed terms, unless stated otherwise.

Example 3.2 *The following terms are not well-formed:* $(a : a!b); a!c$, $b?a; c?a$, $spawn(a?b)$, $(a : b!a) + a?c$, $\Theta : n(x) \mapsto a!x$.

Structural equivalence.

The effect of substitution is not expressed operationally. Instead, we adapt the idea of a *structural equivalence*, proposed for another purpose by Milner in [19], to capture the effect of substitution. $\equiv$ is defined as the smallest congruence satisfying the equations in Figure 5.

Note that the restriction to well-formed terms drastically limits the applicability of the structural equivalence axioms. For instance, by applying (30)–(32), we can derive $(b, x : a?c; x!b; x!c) \equiv a?c; (x : (b : x!b); x!c)$ but not $(x : (b : x!b); x!c) \equiv (x, b : x!b); x!c$, since the latter term is not well-formed. On the other hand, the axioms can always be applied from left to right, in which case they precisely describe the principle of scope extrusion. The precise (technical) requirement for structural equivalence lies in a special syntactic form that all terms can be rewritten to.

Definition 3.3 *A term is in* structural normal form *(*snf*) if it equals* $(\vec{x} : t_1; \cdots; t_n; \sigma)$ *for some* $n \geq 0$, *where* $\{\vec{x}\} \cap dom\,\sigma = \emptyset$ *and for all* $1 \leq i \leq n$, t_i

$\mathbf{1}; t = t$	(27)	$\sigma; \mathbf{0} = \mathbf{0}; \sigma$	(34)
$t; \emptyset = t$	(28)	$\sigma; \sigma' = \sigma \circ \sigma'$	(35)
$(t; u); v = t; (u; v)$	(29)	$\sigma; x\dagger y = \sigma(x)\dagger\sigma(y); \sigma$	(36)
$(x : (y : t)) = (y : (x : t))$	(30)	$\sigma; [x{=}y] = [\sigma(x){=}\sigma(y)]; \sigma$	(37)
$t; (x : u) = (x : t; u)$	(31)	$\sigma; n(\vec{x}) = n(\sigma(\vec{x})); \sigma$	(38)
$(x : t); u = (x : t; u)$	(32)	$\sigma; spawn(t) = spawn(\sigma; t); \sigma$	(39)
$(x : t; \{x{\leftarrow}a\}) = t$	(33)	$\sigma; (t + u) = \sigma; t + \sigma; u$	(40)

Fig. 5. Structural equivalence

equals one of the following:

- $a!b$, $a?b$, $[x{=}y]$, $n(\vec{a})$, $\mathbf{1}$ *or* $\mathbf{0}$;
- $spawn(t'_i)$ *where* t'_i *is in* snf;
- $\sum_{k \in K_i} u_k$ *where* $|K_i| > 1$ *and for all* $k \in K_i$, u_k *is in* snf.

We then have the following property:

Proposition 3.4 *For every* $t \in \mathcal{F}$, *there is a unique* $u \equiv t$ *with* u *in* snf.

Termination.

We have seen above that the effect of substitution is not constricted to the term currently in question, but may also extend to its context, in particular to sequentially appended terms. Furthermore, the effect of substitution may be modified by scope extrusion. For instance, the term $(a : \{x{\leftarrow}a\})$ not only has the substitution $x{\leftarrow}a$ that may carry over to the right, as in $(a : \{x{\leftarrow}a\}); b!x$, but also the restricted name a that may escape its current scope by this means.

Operationally, we deal with this effect by adapting the termination predicate, so that it records additional information about the "residual" of a terminated term, consisting of the remaining substitution and the resultant scope extrusion. Residuals are modelled as (non-free) substitution functions. For the full calculus, therefore, termination will be modelled by an indexed predicate $\checkmark_\sigma$, where σ is a substitution, and $r_{rng\,\sigma}$ expresses which of the σ-images are scope-extruded by substitution. For instance, $(a : \{x{\leftarrow}a\})\checkmark_{\{x{\leftarrow}\tilde{a}\}}$. $\checkmark_\emptyset$ is abbreviated to $\checkmark$. The rules of termination are listed in Figure 6.

Operational semantics.

The transition rules for the full calculus extend those of the basic calculus with channel parameters. Transition labels are the following:

- $a!b$: output of value b over channel a;
- $a!\tilde{b}$: output of a *fresh* value b over a (called *restricted output*);
- $a?x$: input of a value over channel a, to be assigned to the variable x;
- $a?\tilde{x}$: *restricted* input over a, to be assigned to the *local* variable x.

Again, internal action labels are treated as a special case of output: $\tau = \iota!\iota$, where $\iota \notin bv(t) \cup rv(t)$ for all terms t. The set of transition labels is denoted

$$\frac{}{\mathbf{1}\checkmark}\,\mathrm{T}_5 \qquad \frac{}{spawn(t)\checkmark}\,\mathrm{T}_6 \qquad \frac{}{\sigma\checkmark_\sigma}\,\mathrm{T}_7 \qquad \frac{t\checkmark_{\sigma_t} \quad u\checkmark_{\sigma_u}}{(t;u)\checkmark_{\sigma_t\circ\sigma_u}}\,\mathrm{T}_8 \qquad \frac{t\checkmark_\sigma}{(a:t)\checkmark_{\sigma\downarrow a}}\,\mathrm{T}_9$$

$$\frac{}{a!b \xrightarrow{a!b} \mathbf{1}}\,\mathrm{R}_{10} \qquad \frac{}{a?x \xrightarrow{a?x} \mathbf{1}}\,\mathrm{R}_{11} \qquad \frac{}{[a{=}a] \xrightarrow{\tau} \mathbf{1}}\,\mathrm{R}_{12} \qquad \frac{\Theta{:}\, n(\vec{x}) \mapsto t}{n(\vec{a}) \xrightarrow{\tau} (\vec{x} : \vec{x}{\leftarrow}\vec{a}; t)}\,\mathrm{R}_{13}$$

$$\frac{t \xrightarrow{a\dagger\xi} t' \quad a \neq b \neq c_\xi}{(b:t) \xrightarrow{a\dagger\xi} (b:t')}\,\mathrm{R}_{14} \qquad \frac{t \xrightarrow{a\dagger b} t' \quad a \neq b}{(b:t) \xrightarrow{a\dagger\tilde{b}} t'}\,\mathrm{R}_{15}$$

$$\frac{t\checkmark \quad t \xrightarrow{\alpha} t' \quad u \xrightarrow{\beta} u' \quad \{\alpha,\beta\} = \{a!\xi, a?\chi\}}{t;u \xrightarrow{\tau} (r_{\{\xi,\chi\}} : \{c_\chi{\leftarrow}c_\xi\}; t'; u')}\,\mathrm{R}_{16} \qquad \frac{u \equiv t \quad t \xrightarrow{\alpha} t' \quad t' \equiv u'}{u \xrightarrow{\alpha} u'}\,\mathrm{R}_{17}$$

Fig. 6. Transition rules for the full calculus

$\mathcal{L}$, ranged over by α, β. Restricted input and output are generated when input or output actions occur within a scope restriction.

The definition of a transition system has to be adapted to the extended termination predicate and transition labels. Namely, if $s \xrightarrow{a\dagger\tilde{b}} s'$ and s is terminated with residual substitution σ, then s' is terminated, too, such that in the corresponding residual, the b-images of σ are "freed" by scope extrusion (see also rule R_{15}).

Definition 3.5 *An extended $\checkmark$-transition system is a tuple $\langle \mathcal{L}, S, \rightarrow, \checkmark \rangle$ where $\checkmark \subseteq S \times \mathbf{S}$ is a termination relation, such that if $s\checkmark_\sigma$ and $s \xrightarrow{a\dagger\xi} s'$ then $s'\checkmark_{\sigma\uparrow c_\xi}$.*

The operational rules for the choice and *spawn* operators and the non-communication rules of sequential composition are unchanged, and omitted here. For the other operators, the rules are given in Figure 6. Some comments on the operational rules are in order.

- The rule R_{13} for recursion inserts a substitution in front of the term $\Theta(n)$, which replaces the formal parameters by the current names in the process call. Additionally, the formal parameters are restricted to the term t to ensure their locality.
- The rule R_{16} for communication combines a number of features. Two labels α and β can communicate if and only if $\{\alpha, \beta\} = \{a!\xi, a?\chi\}$ for some a, ξ, χ; this results in a syntactic substitution $c_\chi{\leftarrow}c_\xi$ which implements the transfer of a data value (i.e., a channel name), and a potential restriction of $r_{\{\xi,\chi\}}$ $(= r_\xi \cup r_\chi)$, which implements scope extrusion. (Note that r_ξ is non-empty iff $a!\xi$ is a restricted output, and r_χ iff $a?\chi$ a restricted input.)

  The communication rule corresponds to *late binding*. For instance, we can derive

$$(x : a?x; x!b) \xrightarrow{a?\tilde{x}} \mathbf{1}; x!b$$

  after which a value for x, presumably generated by communication, can be

instantiated later through substitution:

$$\begin{aligned} spawn(a!c);(x : a?x; x!b) &\xrightarrow{\tau} (x : \{x{\leftarrow}c\}; spawn(\mathbf{1}); \mathbf{1}; x!b) \\ &\equiv (x : c!b; \{x{\leftarrow}c\}) \equiv c!b \end{aligned}$$

(the equation $spawn(\mathbf{1}) = \mathbf{1}$ is known from the basic calculus).

- A crucial rule is R_{17}, which lifts the transition relation *modulo* structural equivalence. This allows us to "shift" all substitutions out of the way before computing the transitions.

Note that the rules R_8 and R_{16} for sequential composition and communication demand the first operand to fulfil the predicate $\checkmark_\emptyset$. Therefore, in terms $t; u$ with $t\checkmark_\sigma, \sigma \neq \emptyset$, the actions of u can only occur after σ has been applied to u by the rules of structural equivalence. This mechanism prevents non-determinism caused by the application sequence of structural equivalence and transition rules.

Example 3.6 *Consider the process definition* $\Theta{:}\, n(x, y) \mapsto (a : x?a; y!a)$. *Note that the variable* a, *which is bound in the body, must be restricted (otherwise the term would not be well-formed). A process call gives rise to the following behaviour.*

$$\begin{aligned} & n(b, c); b!c \\ \xrightarrow{\tau}\ & (x, y : \{x{\leftarrow}b, y{\leftarrow}c\}; (a : x?a; y!a)); b!c \equiv (x, y, a : b?a; \{x{\leftarrow}b, y{\leftarrow}c\}; y!a); b!c \\ \xrightarrow{b?\tilde{a}}\ & (x, y : \mathbf{1}; \{x{\leftarrow}b, y{\leftarrow}c\}; y!a); b!c \equiv (x, y : c!a; \{x{\leftarrow}b, y{\leftarrow}c\}); b!c \\ \xrightarrow{c!a}\ & (x, y : \mathbf{1}; \{x{\leftarrow}b, y{\leftarrow}c\}); b!c \equiv b!c \\ \xrightarrow{b!c}\ & \mathbf{1} \end{aligned}$$

Proposition 3.7 $\langle \mathcal{L}, \mathcal{F}, \rightarrow, \checkmark \rangle$ *is an extended* $\checkmark$*-transition system.*

Bisimulation and axiomatisation.

Bisimulation is adapted to the extended termination predicate as follows:

Definition 3.8 *Let* T *be an extended* $\checkmark$*-transition system. A symmetrical relation* $R \subseteq S \times S$ *is called a* bisimulation relation *if for all* $(s_1, s_2) \in R$,

- *if* $s_1 \xrightarrow{\ell} s_1'$ *then* $\exists s_2 \xrightarrow{\ell} s_2'$ *such that* $(s_1', s_2') \in R$;
- $s_1\checkmark_\sigma$ *then* $s_2\checkmark_\sigma$.

$s_1, s_2 \in S$ *are called* bisimilar, *denoted* $s_1 \mathrel{\dot{\sim}_{\mathcal{F}}} s_2$, *if* $(s_1, s_2) \in R$ *for some bisimulation relation* R.

Example 3.9 *The terms* $\emptyset$ *and* $\{a{\leftarrow}b\}$ *are not bisimilar; on the other hand, for instance* $\{b{\leftarrow}c\}; spawn(a!b); c!d \mathrel{\dot{\sim}_{\mathcal{F}}} (a!c; c!d + c!d; spawn(a!c)); \{b{\leftarrow}c\}$.

Since we use structural equivalence, we can no longer rely on SOS theory to prove congruence. Still, we have the following result:

$$(x:t) = t \quad \text{if } x \notin var(t) \tag{41}$$
$$(x:t+u) = (x:t) + (x:u) \tag{42}$$
$$(x:x\dagger y;t) = \mathbf{0} \tag{43}$$
$$(x:[x{=}y];t) = \mathbf{0} \tag{44}$$
$$(x:spawn(t)) = spawn((x:t)) \tag{45}$$
$$spawn(\sigma) = \mathbf{1} \tag{46}$$
$$n(\vec{a}) = \tau;(\vec{x}:\vec{x}{\leftarrow}\vec{a};t) \quad (\Theta{:}\, n(\vec{x}) \mapsto t) \tag{47}$$
$$[x{=}y] = [y{=}x] \tag{48}$$
$$[x{=}x] = \tau \tag{49}$$
$$[x{=}y] = \mathbf{0} \quad \text{if } x \neq y \tag{50}$$
$$\emptyset = \mathbf{1} \tag{51}$$

$$\text{if } t = \sum_{i\in\mathcal{I}} \alpha_i;t_i \text{ and } u = \sum_{k\in\mathcal{K}} \beta_k;u_k \tag{52}$$
$$\text{then } spawn(t);u = \sum_{i\in\mathcal{I}} \alpha_i;spawn(t_i);u + \sum_{k\in\mathcal{K}} \beta_k;spawn(t);u_k + \sum_{\{\alpha_i,\beta_k\}=\{x!a,y?z\}} [x{=}y];\{z{\leftarrow}a\};spawn(t_i);u_k$$

Fig. 7. The theory $\mathcal{A}X_{\mathcal{F}}$ for the full calculus $\mathcal{F}$.

Theorem 3.10

- $\dot{\sim}_{\mathcal{F}}$ *is congruent for spawn, restriction and choice, and in the first operand of sequential composition.*
- *if* $(\vec{x}:\sigma);t \dot{\sim}_{\mathcal{F}} (\vec{x}:\sigma);u$ *for all* $\sigma \in \mathbf{S}$*, then* $v;t \dot{\sim}_{\mathcal{F}} v;u$ *for all* $v \in \mathcal{F}$*.*
- $t \dot{\sim}_{\mathcal{F}} u$ *implies* $\alpha;t \dot{\sim}_{\mathcal{F}} \alpha;u$ *for all* $\alpha \in \mathcal{L}$*.*

The lack of congruence in the second operand of sequential composition is well-known from the π-calculus. For instance, if $t = spawn(x!a);y?b$ and $u = x!a;y?b + y?b;spawn(x!a)$ then $t \dot{\sim}_{\mathcal{F}} u$ but $\{x{\leftarrow}y\};t \not\dot{\sim}_{\mathcal{F}} \{x{\leftarrow}y\};u$. If we adapt the equational proof rule for congruence correspondingly, then we can give a limited completeness result for $\dot{\sim}_{\mathcal{F}}$. For this purpose, we define the theory $\mathcal{A}X_{\mathcal{F}}$ to consist of the $\sim_{\mathcal{B}}$-axioms of Figure 2 except for the restriction axioms, the recursion axiom and the expansion law, combined with the equations in Figure 7.

The expansion law for $\mathcal{F}$, (52), is very similar to the one for the basic calculus, (24). In the case of communication, the τ is replaced by a matching operator, which expresses that the channels are equal, so communication may take place. An explicit τ to model the communication is not necessary, because successful matching equals τ. In contrast to [20], we do not need to consider restricted in- or output in the expansion law, because with the help of (31) and (42) it is always possible to expand a restriction to both communication partners before the communication takes place.

Theorem 3.11 *The theory* $\mathcal{AX}_{\mathcal{F}}$ *is sound with respect to* $\dot{\sim}_{\mathcal{F}}$*.*

To obtain congruence we have to require bisimilarity over all substitutions.

Definition 3.12 *Two terms* t *and* u *are* congruent*, written* $t \sim_{\mathcal{F}} u$*, iff*

$\forall \sigma \in \mathbf{S} : (\vec{x} : \sigma); t \dot{\sim}_{\mathcal{F}} (\vec{x} : \sigma); u.$

Let $\mathcal{A}X'_{\mathcal{F}}$ denote the theory $\mathcal{A}X_{\mathcal{F}}$ without axiom (50).

Theorem 3.13 *The theory $\mathcal{A}X'_{\mathcal{F}}$ is sound with respect to $\sim_{\mathcal{F}}$.*

We denote the finite fragment of $\mathcal{F}$ by $\mathcal{F}_{fin}$. Furthermore, for syntactical convenience we extend the notation α, β, ... for actions to matching operators. The completeness result once more relies on a normal form:

Definition 3.14 *A term $t \in \mathcal{F}_{fin}$ is in* full normal form *(*fnf*), if t is a term in N in the following grammar:*

$$N ::= \textstyle\sum_{i \in \mathcal{I}} \alpha_i; N_i + \sum_{j \in \mathcal{J}} (x_j : y_j \dagger x_j; N_j) \;\mid\; (\vec{x} : spawn(B); \sigma)$$

$$B ::= \textstyle\sum_{i \in \mathcal{I}} \alpha_i; B_i + \sum_{j \in \mathcal{J}} (x_j : y_j \dagger x_j; B_j)$$

where $x_j \neq y_j$ and $\{\vec{x}\} \subseteq rng\,\sigma$.

In contrast to the basic normal forms in Definition 2.6, the full normal forms may contain restrictions. This is necessary to capture restricted in- and output, e.g., in $(a : b?a)$. Note that the restriction operator is placed directly before the action with the first occurrence of the restricted channel and ranges until the end of the term. For instance, the normal form of $(a : b!c; d?a); e!f$ is $b!c; (a : d?a; e!f)$. Substitutions occur only at the end of terms, because we are able to shift them through a term by structural equivalence. This trailing substitution may be still restricted, because in terms like $(a : spawn(y!a); \{x \leftarrow a\})$ the restriction cannot be removed. Note that either the trailing substitution and/or the corresponding restriction may be empty. Terms like $\alpha; \sigma$ are in normal form, because they are equal to $\alpha; (\varepsilon : spawn(\mathbf{0}); \sigma)$.

Theorem 3.15 *For all $t \in \mathcal{F}_{fin}$, there is an $u \in \mathcal{F}_{fin}$ in full normal form such that $t \sim_{\mathcal{F}} u$.*

With Theorem 3.15 we can derive the following completeness result for the theory $\mathcal{A}X_{\mathcal{F}}$.

Theorem 3.16 *In an equational derivation system* not *including a congruence rule for the second operand of sequential composition, the theory $\mathcal{A}\mathcal{X}_{\mathcal{F}}$ is complete for $\dot{\sim}_{\mathcal{F}}$ over the finite fragment $\mathcal{F}_{fin}$ of $\mathcal{F}$.*

Parrow and Sangiorgi in [21] give a corresponding complete axiomatisation for congruence, which relies on a more general form of conditional than that provided by the matching operator; we conjecture that an appropriate adaptation of their solution will yield a complete axiomatisation for $\sim_{\mathcal{F}}$.

4 Conclusions and Future Work

We have presented a direct process algebraic formalisation of operators for process creation and sequential composition, integrated them in a name-passing calculus, and provided this calculus with operational and axiomatic semantics. We now discuss some similar investigations in the literature.

An early formalisation of an operator for process creation is given by Baeten and Vaandrager [3] in the setting of *ACP* [2,4], of which sequential composition has always been an integral part. Our basic calculus $\mathcal{B}$ is quite similar to their solution, except that they rely on an auxiliary "asymmetric parallel composition" $\lceil$ such that $t \lceil u$ (in their calculus) precisely corresponds to our $spawn(t); u$. Furthermore, they have a slightly different treatment of termination, due to which they do not need to restrict to guarded choice. However, they do not consider mobility.

Another existing approach along the same lines as ours is the *fork calculus* of Havelund and Larsen [15], extended to the *ψ-calculus* in [14]: they, too, develop a calculus with process creation, sequential composition and name passing. They give a two-level semantics: the first level models the local behaviour of a single process, the second the global system's behaviour as a *multiset* of processes. The latter effectively corresponds to parallel composition restricted to the outermost level; this can again be regarded as an auxiliary operator. The ψ-calculus extension allows name passing in π-calculus style, just as our full calculus $\mathcal{F}$; however, sequential composition is once more restricted to action prefixing, at least for input actions (i.e., the binding constructors). In this way, at the cost of severely restricting the use of sequential composition, the ψ-calculus avoids the problems we have solved by distinguishing between binding and scoping and introducing substitution as a syntactic construct.

The work reported here is part of an ongoing project investigating methods for the design of reactive systems. We are planning to develop a design methodology which allows for a top-down design of systems, based on the language *ProFun* [9]. The calculus presented in this paper has been developed as a basis for reasoning about the behaviour aspects of *ProFun* programs.

As a next step, in order to reflect all aspects of *ProFun*, we aim to integrate data into the calculus. We are planning to consider names as a representation for functional expressions and identifiers; for instance the function application *f 3 4* will be a valid name and the declaration $x = f\ 3\ 4$ can be translated directly into the substitution $\{x \leftarrow f\ 3\ 4\}$. We claim that this treatment of expressions easily allows for realising *eager* and *lazy evaluation* semantics. Channel values will be represented by a specific data type. The introduction of substitution as a syntactic operator simplifies the operational semantics of the calculus with data, because there is no need for semantic *environments* to reflect bindings (cf. [6]).

Furthermore, the approach of top-down design has to provide mechanisms for the *stepwise* refinement of reactive systems. Therefore, we are planning to adapt techniques for action refinement for our calculus; as mentioned before, this has been another major reason for investigating sequential composition.

Acknowledgements.
We are grateful to Heike Wehrheim for comments on this paper. The first author would like to thank Michaela Huhn and Cosimo Laneve for discussions on the integration of concurrency and functional programming.

References

[1] J.C.M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In E. Best, editor, *Proceedings of CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 1993.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[3] Jos C. M. Baeten and Frits W. Vaandrager. An Algebra for Process Creation. *Acta Informatica*, 29(4):303–334, 1992.

[4] J.A. Bergstra and J.W. Klop. Algebra for Communicating Processes with Abstraction. *Journal of Theoretical Computer Science*, 37:77–121, 1985.

[5] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation Can't Be Traced. *Journal of the ACM*, 42(1):232–268, January 1995.

[6] Gian-Luigi Ferrari, Ugo Montanari, and Paola Quaglia. A π-calculus with explicit substitutions. *Theoretical Computer Science*, 168:53–103, 1996.

[7] W. Ferreira and M. Hennessy. Towards a Semantic Theory of CML. Technical Report 2/95, University of Sussex, February 1995.

[8] William Ferreira, Matthew Hennessy, and Alan Jeffrey. A Theory of Weak Bisimulation for Core CML. Technical Report 05/95, University of Sussex, September 1995.

[9] Thomas Gehrke and Michaela Huhn. ProFun – a Language for Executable Specifications. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of PLILP '96*, volume 1140 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 1996.

[10] Thomas Gehrke and Arend Rensink. Process Creation and Full Sequential Composition in a Name-Passing Calculus. Hildesheimer Informatik-Bericht HIB 7/97, Institut für Informatik, Universität Hildesheim, May 1997.

[11] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2), 1989.

[12] Ursula Goltz, Roberto Gorrieri, and Arend Rensink. Comparing Syntactic and Semantic Action Refinement. *Information and Computation*, 125:118–143, 1996.

[13] Ursula Goltz and Rob J. van Glabbeek. Equivalence Notions for Concurrent Systems and Refinement of Actions. In *Proceedings of MFCS '89*, Lecture Notes in Computer Science, pages 237–248. Springer, 1989.

[14] Klaus Havelund. *The Fork Calculus.* PhD thesis, DIKU, University of Kopenhagen, 1994.

[15] Klaus Havelund and Kim G. Larsen. The Fork-Calculus. *Nordic Journal of Computing*, 1:346–363, 1994.

[16] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[17] J.F.Groote and F.W.Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, 1992.

[18] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[19] Robin Milner. Functions as Processes. Technical Report 1154, INRIA, February 1990.

[20] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I+II. *Information and Computation*, 100, 1992.

[21] Joachim Parrow and Davide Sangiorgi. Algebraic theories for name-passing calculi. *Information and Computation*, 120:174–197, 1995.

[22] J.H. Reppy. Concurrent ML: Design, application and semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer, 1992.

[23] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, Department of Computer Science, University of Edinburgh, 1992. No. CST–99–93; also available as ECS–LFCS–93–266.

[24] Bent Thomsen, Lone Leth, and Alessandro Giacalone. Some Issues in the Semantics of Facile Distributed Programming. In *Proceedings of REX Workshop "Semantics: Foundations and Applications"*, volume 666 of *Lecture Notes in Computer Science.* Springer, 1992.