Confluence of Processes and Systems of Objects

Xinxin Liu* and David Walker

Department of Computer Science
University of Warwick
Coventry CV4 7AL, U.K.

Abstract

An extension to the theory of confluence in the process calculus CCS is presented. The theory is generalized to an extension of the π-calculus. This calculus is used to provide semantics by translation for a parallel object-oriented programming language. The confluence theory is applied to prove the indistinguishability in an arbitrary program context of two class definitions which generate binary tree data structures one of which allows concurrent operations.

1 Introduction

The aims of this paper are to present an extension to the theory of confluence in process calculus and to illustrate the new theory by applying it to a problem concerning concurrent operations on binary tree data structures expressed in a parallel object-oriented programming language. In fact, the development of the theory was stimulated by the problem. We begin by describing it.

Characteristic of the object-oriented style of programming is the description of a computational system as a collection of *objects* each of which is a self-contained entity possessing data (references to objects and simple values) and procedures (*methods*) for acting on those data. A program of a parallel object-oriented language typically consists of a collection of class definitions, each of which provides a template for its object *instances*, together with an indication of how a computation should be initiated. Such a program often describes a highly mobile concurrent system in which new objects are created as computation proceeds and references to objects are passed in communications. Parallel object-oriented languages differ from one another in the ways in which they integrate parallelism with object-oriented features; see e.g. [1, 8].

In [4] a development method for concurrent programs is proposed. Central to it are the application of program transformations to control the introduction of concurrency into designs and the use of ideas from object-oriented programming to control

*Supported by a grant from the U.K. Engineering and Physical Sciences Research Council

interference. The problem we consider concerns two classes taken, with minor modifications, from [4] whose instances may be used to construct binary tree-structured symbol tables. The first of these is as follows.

```
class T
var K:NAT, V:ref(A), L:ref(T), R:ref(T)
method Insert(X:NAT, W:ref(A))
    if K=nil then (K:=X ; V:=W ; L:=new(T) ; R:=new(T))
    else if X=K then V:=W
         else if X<K then L!Insert(X,W)
              else R!Insert(X,W) ;
    return
method Search(X:NAT):ref(A)
    if K=nil then return nil
    else if X=K then return V
         else if X<K then return L!Search(X)
              else return R!Search(X)
```

An object of this class represents a node which stores in its variables K, V, L, R an integer key, a value (a reference to an object of some class A) and references to two instances of the class (its left and right children in the tree structure of which it is a component). It has two actions: the method Insert which allows a key-value pair to be inserted, and the method Search which returns the value associated with its key parameter (or nil if there is none). When the expression new(T) is evaluated a new instance of the class is created; the value of the expression is a reference to that object. When an object is created all its variables have nil values and it assumes a quiescent state in which any of its methods may be invoked. On completing a method invocation an object returns to its quiescent state; another method may then be invoked. Execution of the statement L!Insert(X,W) involves left-to-right evaluation of L, X, W and then the invocation in the object to which the value of L is a reference of the Insert method with the values of X, W as parameters. The activity of the invoking object is suspended until it is released from the rendezvous by execution of a return statement by the object in which the method was activated. Note that new (empty) leaf nodes are created when an insertion with a fresh key is made; thus the data structures are full binary trees. The value of the expression L!Search(X) is the value returned to the object by the execution of a return statement in the child node to which the value of L is a reference.

A tree-structured symbol table is accessible to other objects in a system only through its root. Moreover when a method is invoked in the root the entire tree becomes inaccessible until the invocation has been passed down through the structure to the node which should handle it, the appropriate activity has taken place, a sequence of return statements has rippled back along the relevant path, and the root has released from the rendezvous the object which made the initial invocation. The second class definition is as follows.

```
class T
var K:NAT, V:ref(A), L:ref(T), R:ref(T)
method Insert(X:NAT, W:ref(A))
    return ;
    if K=nil then (K:=X ; V:=W ; L:=new(T) ; R:=new(T))
    else if X=K then V:=W
         else if X<K then L!Insert(X,W)
              else R!Insert(X,W)
method Search(X:NAT):ref(A)
    if K=nil then return nil
    else if X=K then return V
         else if X<K then commit L!Search(X)
              else commit R!Search(X)
```

Referring now to the first class as $\mathtt{T}_0$, the new class `T` can be obtained from it by applying two transformations: firstly, moving the `return` statements in the `Insert` method to the beginning of the body; and second, in the `Search` method, replacing the `return` statements invoking `Search` methods in the children by `commit` statements. The effect of moving the `return` statement in the `Insert` method is to free the invoking object from the rendezvous thus allowing it to proceed in parallel with the node which then proceeds to carry out the insertion. When an object α executes a `commit` statement by invoking a method in an object β, it is implicit (i) that β should return its result not to α but to the object γ to which α should return a result, and (ii) that α is freed from the task of returning a result to γ. In particular, execution of α may proceed in parallel with that of β. Thus if the `Search` method is invoked in a node with a key smaller (resp. larger) than that stored there, the node will commit that search to its left (resp. right) child, and we may think of the node as passing to the child the return address to which the result of the search should be sent. This address will have been received by the node either directly from the initiator of the search (if the node is the root) or from its parent in the tree.

The problem which stimulated the work of this paper was to determine whether the two classes above are interchangeable in an arbitrary program context, that is whether or not the observable behaviour of a program could be altered by replacing one of the classes by the other. A more difficult problem is to determine general conditions under which transformations such as the movement of `return` statements and the replacement of `return` statements by `commit` statements illustrated in the `Insert` and `Search` methods respectively are sound. Such general transformation rules are proposed in [4]. One point of the present work is to bring to the surface some difficulties concerning tractable, general transformation rules. This issue is discussed in the concluding section.

In previous work [22] it has been argued that process calculus provides a good framework for studying the behaviour of systems of concurrent objects. It has been shown [21, 23, 24, 22, 5, 7] that one may give quite natural semantics for parallel object-oriented languages by translation to calculi based on the π-calculus [13] in which agents may pass names of communication links to one another. In particular one may give precise, abstract descriptions of the behaviours of systems and employ the apparatus of process theory to reason about them. The present work gives further evidence of the utility of this kind of framework. To solve the problem stated above we isolate a class of agent contexts which contains all encodings of program contexts of the language in question and show that the agents representing the symbol table classes can not be distinguished by any context in this class.

An intuition which suggests that the symbol tables classes are interchangeable is that the behaviours which they generate are determinate. That is, if requested to execute any sequence of method invocations a table would reach a uniquely determined abstract state having returned a uniquely determined value for each call. That is not to say, however, that tables of the two classes would not be distinguishable in some environments. For example, a table of class T might accept several Search requests before returning the results of the searches in an order different from that in which they were initiated. By contrast, a T0-table imposes a strict invoke-release discipline on its environment.

A precise definition of *determinacy* is introduced and studied in the setting of the process calculus CCS in [11]. It is shown, among other things, that determinacy is not in general preserved by the CCS composition and restriction operators. With the aim of providing a theoretical framework within which one may build from determinate components systems which are guaranteed, by construction, to be determinate, a refined notion of determinacy, *confluence*, is then introduced. This notion arises in a variety of forms in the theory of computation. Its essence, to quote Milner [11], is that "of any two possible actions, the occurrence of one will never preclude the other". Its pertinence in the present setting is clear. To tackle the problem of the symbol table classes it is necessary to make a significant extension to the theory of confluence as an agent representing a program context may be highly non-confluent. (Incidentally, the generalization of the theory of confluence in CCS to the π-calculus raises interesting questions; a study of this topic will appear in [17].) We introduce a new notion of *partial confluence* which requires of an agent that it be "well-behaved" with respect to a distinguished class of actions. The precise definition is somewhat delicate and uses the branching bisimilarity introduced in [20]. It is chosen to be generous enough to encompass many systems but restrictive enough that it enjoys a strong theory. The main result states, roughly, that in certain contexts a partially-confluent agent may be replaced by a simpler "pruned" version of itself without altering the observable behaviour of the system. Confluence in value-passing CCS has been studied in [18, 19]. In the latter work the definition, from an unpublished note by Milner, of a notion called "partial confluence" for pure CCS agents is stated. Although the same name is used, the version of confluence studied here differs from it. The relationship between the two is explained in the text.

In the following section we collect background material. In section 3 we introduce the theory of partial confluence in the CCS setting and in section 4 generalize it to the process calculus, essentially an amalgamation of value-passing CCS and the π-calculus, which we use as semantic basis. In section 5 we use the theory developed to prove the interchangeability of the two symbol table classes in an arbitrary program context. The paper ends with some concluding remarks. To meet the space requirement most proofs are omitted.

Acknowledgment We thank Cliff Jones whose related research provided an important stimulus for the present work and with whom the second author has enjoyed helpful conversations.

2 Preliminaries

In this section we collect necessary background material on the parallel object-oriented programming language which is the setting for the problem described in the Introduction and on its semantics by translation to the π_v-calculus described in [22].

2.1 The programming language

The programming language is a variant of the $\pi o\beta\lambda$-language [4] which in turn derives from the POOL family [1]. The language has types NAT (natural numbers), BOOL (booleans), UNIT and ref(A) for A a class name. The UNIT type plays a rôle similar to that of the type of that name in the language Standard ML and a mode in ALGOL 68; it has a single value. A value of type ref(A) is a reference to an object of class A; class definitions are explained below. The principal syntactic entities are *statements* each of which is assigned a type in a standard way; we omit the details. The language has constant symbols 0, 1,... and nil, the last of which is overloaded and is used to represent a reference to no object, the "undefined" value of type NAT and the value of type UNIT. In the abstract syntax definitions below we use K to range over constants, M over method names, A over class names, T over types, X, Y, Z over variables, and S over statements, and we write $\tilde{Z}$ for a tuple $Z_1, \ldots, Z_n$ of syntactic entities. Statements are the well-typed phrases given as follows:

$$\begin{array}{rcl} S & ::= & K \mid X \mid \mathsf{new}(A) \mid S!M(\tilde{S}) \mid \mathsf{op}(\tilde{S}) \mid \\ & & X := S \mid S_1; S_2 \mid \mathsf{if}\ S\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid \\ & & \mathsf{output}\ S \mid \mathsf{return}\ S \mid \mathsf{commit}\ S!M(\tilde{S}). \end{array}$$

The value of new(A) is a reference to a newly-created object of class A. The evaluation of $S!M(\tilde{S})$ involves the left-to-right evaluation of S and the statements in the tuple $\tilde{S}$ followed by the invocation in the object to which the value of S is a reference of the method M of the class of that object with parameters the values of $\tilde{S}$. The value of the statement is the simple value or reference returned to the object as the result of the method invocation. We assume that the basic operation symbols op are $=$, $<$ on NAT and not, and on BOOL. The assignment, sequence and conditional statements are standard. In output S, the statement S of type NAT is evaluated and its value output. The return and commit statements are explained in the Introduction.

Declarations are given as follows. Firstly, variable declarations are given by

$$Vdec \ ::= \ \mathsf{var}\, X_1 : T_1 \,, \ldots, \ X_n : T_n$$

Then method declarations are given by

$$Mdec \ ::= \ \mathsf{method}\, M(\tilde{Y} : \tilde{T}) : T, \ Vdec, \ S$$

where $\tilde{T}$ are the types of the formal parameters $\tilde{Y}$, T is the result type and S is the body of the method with *Vdec* declaring variables local to it. In the examples in the Introduction, we omit UNIT as result type and abbreviate return nil to return. Sequences of method declarations are given by

$$Mdecs \ ::= \ Mdec_1 \,, \ldots, \ Mdec_q$$

class declarations by

$$Cdec \ ::= \ \mathsf{class}\, A, \ Vdec, \ Mdecs$$

and finally program declarations by

$$Pdec \ ::= \ Cdec_1 \,, \ldots, \ Cdec_r, \ \mathsf{trigger}\, S$$

where S is of the form $\mathsf{new}(A)!M(\widetilde{K})$. The statement S acts as a trigger to initiate the computation by creating and activating a *root object* of one of the classes A.

2.2 The translational semantics

This paper is based on the translational semantics given in [22] which uses the π_v-calculus. To illustrate the sort discipline employed, consider the symbol table classes. As we will see below the class T is encoded as an abstraction of sort (LINK[T]). The sorting decrees that a name of sort LINK[T] carries a pair

$$m_1, m_2 : \mathsf{METH}^I_\mathsf{T}[\mathsf{NAT}, \mathsf{ref(A)}; \mathsf{UNIT}], \mathsf{METH}^S_\mathsf{T}[\mathsf{NAT}; \mathsf{ref(A)}]$$

of names, one for each method of the class (I for Insert, S for Search). A name $m_1 : \mathsf{METH}^I_\mathsf{T}[\mathsf{NAT}, \mathsf{ref(A)}; \mathsf{UNIT}]$ in turn carries an integer, an object identifier of class ref(A), and a return link of sort $\mathsf{RES}^I_\mathsf{T}[\mathsf{UNIT}]$, and similarly $m_2 : \mathsf{METH}^S_\mathsf{T}[\mathsf{NAT}; \mathsf{ref(A)}]$ carries an integer and a return link of sort $\mathsf{RES}^S_\mathsf{T}[\mathsf{ref}(A)]$. We write $\mathsf{OBJECT[T]} \equiv \mathsf{METH}^I_\mathsf{T}[\mathsf{NAT}, \mathsf{ref(A)}; \mathsf{UNIT}], \mathsf{METH}^S_\mathsf{T}[\mathsf{NAT}; \mathsf{ref(A)}]$.

A program $Pdec \equiv Cdec_1, \ldots, Cdec_r$, $\mathsf{trigger}\, S$ is encoded as a restricted composition of the translations of the class definitions and the trigger S:

$$[\![Pdec]\!] \stackrel{\text{def}}{=} (\nu \ldots)\,([\![Cdec_1]\!]\langle\ldots\rangle \mid \ldots \mid [\![Cdec_r]\!]\langle\ldots\rangle \mid [\![S]\!]\langle\ldots\rangle).$$

It has at most one free name, namely out of sort LINK[NAT] at which integer values may be emitted; it is used in the translation of statements of the form output S. A class definition is translated as a replication with a *link* name to pass out an *object* *identifier* each time a new object is required. For example the encoding of the table class T has the following form:

$$[\![\mathtt{T}]\!] \overset{\text{def}}{=} (new)\,!\,(\nu ins, srch)\,\overline{new}(ins, srch).\,\mathrm{Obj}_{\mathtt{T}}\langle new, ins, srch\rangle$$

where new is the link name, $(ins, srch)$ is the object identifier, and $\mathrm{Obj}_{\mathtt{T}}\langle new, ins, srch\rangle$ represents the object. Note that $ins, srch$ are private names since each new instance of the class has a fresh identifier. Also, (!) is the replication operator from [12] which may be eliminated in favour of an agent constant. We can think of $!P$ as $P|P|P|\ldots$.

We do not repeat here the definition of $[\![\cdot]\!]$; see the papers cited above. Instead, we present two π_v-calculus agents T_0 and T which are simplifications of $[\![\mathtt{class\ T_0}]\!]$ and $[\![\mathtt{class\ T}]\!]$ respectively and explain some ideas of the translation using them. Thus:

$$\begin{array}{lcl} T_0 & \equiv & (new)\,!\,(\nu ins, srch)\,\overline{new}(ins,\ srch).\,E_0\langle new, ins, srch\rangle \\ T & \equiv & (new)\,!\,(\nu ins, srch)\,\overline{new}(ins,\ srch).\,E\langle new, ins, srch\rangle \end{array}$$

where $E_0, E : (\mathsf{LINK[T]}, \mathsf{OBJECT[T]})$, $C_0, C : (\mathsf{LINK[T]}, \mathsf{OBJECT[T]}, \mathsf{N}, \mathsf{OBJECT[A]}, \mathsf{OBJECT[T]}^2)$ are the abstractions defined as follows (where cond is a nested conditional):

$$\begin{array}{l} E_0 \overset{\text{def}}{=} (new,\ ins,\ srch) \\ \quad ins(x, w, r).\,new(insl,\ srchl).\,new(insr,\ srchr).\,\overline{r}.\,C_0\langle\ldots,\ x,\ w,\ \ldots\rangle \\ \quad +\,srch(x, r).\,\overline{r}(\mathsf{nil}).\,E_0\langle new,\ ins,\ srch\rangle \end{array}$$

$$\begin{array}{l} C_0 \overset{\text{def}}{=} (new,\ ins,\ srch,\ k,\ v,\ insl,\ srchl,\ insr,\ srchr) \\ \quad ins(x, w, r).\,\underline{\mathsf{cond}}\,(\ x = k\ : \overline{r}.\,C_0\langle\ldots, k, w, \ldots\rangle, \\ \qquad\qquad\qquad\qquad x < k\ : \overline{insl}(x, w, \nu r').\,r'.\,\overline{r}.\,C_0\langle\ldots, k, v, \ldots\rangle, \\ \qquad\qquad\qquad\qquad \mathsf{else}\ \ : \overline{insr}(x, w, \nu r').\,r'.\,\overline{r}.\,C_0\langle\ldots, k, v, \ldots\rangle) \\ \quad +\,srch(x, r).\,\underline{\mathsf{cond}}\,(\ x = k : \overline{r}v.\,C_0\langle\ldots, k, v, \ldots\rangle, \\ \qquad\qquad\qquad\qquad x < k : \overline{srchl}(x, \nu r').\,r'(y).\,\overline{r}y.\,C_0\langle\ldots, k, v, \ldots\rangle, \\ \qquad\qquad\qquad\qquad \mathsf{else}\ \ : \overline{srchr}(x, \nu r').\,r'(y).\,\overline{r}y.\,C_0\langle\ldots, k, v, \ldots\rangle) \end{array}$$

$$\begin{array}{l} E \overset{\text{def}}{=} (new,\ ins,\ srch) \\ \quad ins(x, w, r).\,\overline{r}.\,new(insl,\ srchl).\,new(insr,\ srchr).\,C\langle\ldots, x, w, \ldots\rangle \\ \quad +\,srch(x, r).\,\overline{r}(\mathsf{nil}).\,E\langle new,\ ins,\ srch\rangle \end{array}$$

$$\begin{array}{l} C \overset{\text{def}}{=} (new,\ ins,\ srch,\ x,\ w,\ insl,\ srchl,\ insr,\ srchr) \\ \quad ins(x, w, r).\,\overline{r}.\,\underline{\mathsf{cond}}\,(\ x = k : C\langle\ldots, k, w, \ldots\rangle, \\ \qquad\qquad\qquad\qquad x < k : \overline{insl}(x, w, \nu r').\,r'.\,C\langle\ldots, k, v, \ldots\rangle, \\ \qquad\qquad\qquad\qquad \mathsf{else}\ \ : \overline{insr}(x, w, \nu r').\,r'.\,C\langle\ldots, k, v, \ldots\rangle) \\ \quad +\,srch(x, r).\,\underline{\mathsf{cond}}\,(\ x = k : \overline{r}v.\,C\langle\ldots, k, v, \ldots\rangle, \\ \qquad\qquad\qquad\qquad x < k : \overline{srchl}(x, r).\,C\langle\ldots, k, v, \ldots\rangle, \\ \qquad\qquad\qquad\qquad \mathsf{else}\ \ : \overline{srchr}(x, r).\,C\langle\ldots, k, v, \ldots\rangle). \end{array}$$

Note that we write e.g. $\overline{insl}(x, w, \nu r')$ as an abbreviation for $(\nu r')\overline{insl}(x, w, r')$. From the expressions defining C_0 and C it is not hard to convince oneself (and it can be proved) that T_0 and T are branching bisimilar to $[\![\mathtt{class\ T_0}]\!]$ and $[\![\mathtt{class\ T}]\!]$ respectively.

3 Partial confluence in CCS

In [19], a notion called "R-partial confluence" due to Milner is defined in the setting of pure CCS, where R is a set of labels. It is a generalization of the notion of confluence in that a process is confluent just in case it is R-partially confluent when R is the set of all labels. However, a partially confluent process, like a confluent one, must be determinate with respect to internal moves. So this notion cannot be applied in a situation where internal moves may change an agent's state. In this section we introduce and study a more general version of confluence. Before defining it we introduce a convenient abbreviation: we write $P \Longrightarrow\xrightarrow{\alpha} P'$ to mean that $P \Longrightarrow P'' \xrightarrow{\alpha} P'$ for some P'' with $P'' \approx_b P$ and moreover if $\alpha = \tau$ then $P' \not\approx_b P$.

Definition 3.1 Let $R \subseteq \mathcal{L}$ be a set of labels. An agent P is *R-confluent* if for every derivative Q_0 of P, whenever $Q_0 \xrightarrow{r} Q_1$ and $Q_0 \Longrightarrow\xrightarrow{\alpha} Q_2$ with $r \in R$ and $\alpha \in Act$, then either $\alpha = r$ and $Q_1 \approx_b Q_2$, or $\alpha \neq r$ and agents Q'_1, Q'_2 can be found so that $Q_1 \Longrightarrow\xrightarrow{\alpha} Q'_1$, $Q_2 \Longrightarrow\xrightarrow{r} Q'_2$ and $Q'_1 \approx_b Q'_2$.

It is easy to see that confluence implies R-confluence (for any R). However the converse fails as e.g. $(\tau.a + \tau.b) \mid r$ is $\{r\}$-confluent but not confluent. Although for convienience we have dropped the word "partial", the definition of "R-confluence" generalizes that of "R-partial confluence" stated in [19]. In fact, a process P is R-partially confluent if and only if P is R-confluent and for every derivative Q of P, whenever $Q \xrightarrow{\tau} Q'$ then $Q \approx_b Q'$. This follows from the fact that if P is determinate with respect to internal moves, i.e. for every derivative P' of P, whenever $P' \xrightarrow{\tau} P''$ then $P' \approx P''$, then $P \approx Q$ just in case $P \approx_b Q$. It is necessary to use $\approx_b$ instead of $\approx$ in the generalized theory of partial confluence. Roughly speaking, the reason is that since we no longer require processes to be determinate with respect to internal moves, $\approx$ cannot guarantee that R-confluence is a property of equivalence classes. It is straightforward to show that R-confluence is preserved by $\approx_b$.

It is convenient to introduce a further abbreviation. For $s = \alpha_1 \ldots \alpha_n \in Act^*$ we write $\xrightarrow{s}_\approx$ for the composite relation $\approx_b\xrightarrow{\alpha_1}\approx_b \ldots \approx_b\xrightarrow{\alpha_n}\approx_b$. The following result enunciates a simple but useful property of R-confluent agents. Recall from the Preliminaries that s/α is the excess of s over α.

Lemma 3.2 If P is R-confluent then for any $s \in R^*$ and $\alpha \in Act$, whenever $P \xrightarrow{s}_\approx P_1$ and $P \Longrightarrow\xrightarrow{\alpha} P_2$, either α occurs in s and $P_2 \xrightarrow{s/\alpha}_\approx P_1$, or α does not occur in s and there is P_0 such that $P_1 \Longrightarrow\xrightarrow{\alpha} P_0$ and $P_2 \xrightarrow{s}_\approx P_0$. □

Using this lemma we can establish the first significant result. It implies that the state of a restricted composition of R-confluent agents is not changed up to $\approx_b$ by an interaction between components via a name and a co-name in R, provided all names in R are restricted.

Lemma 3.3 If P, T are R-confluent, $P \xrightarrow{s}_\approx P'$ and $T \xrightarrow{\overline{s}}_\approx T'$ with $s, \overline{s} \in R^*$, $s = r_1 \ldots r_n$, $\overline{s} = \overline{r_1} \ldots \overline{r_n}$, and $R \subseteq L$ then $(P|T)\backslash L \approx_b (P'|T')\backslash L$. □

We now have a rather long definition.

Definition 3.4 Suppose M, R are disjoint sets of names and $\psi : M \to R$ a function. A derivation-closed set $\mathcal{S}$ of R-confluent agents is (M, R, ψ)-*tidy* if there is a partition $\{\mathcal{S}^{\tilde{r}} \mid \tilde{r}$ a finite submultiset of R$\}$ of $\mathcal{S}$ (an (M, R, ψ)-*tidy partition*) such that:

1. if $P \in \mathcal{S}^{\tilde{r}}$ and $P \xrightarrow{\alpha} P'$ where $\alpha \notin M \cup R$ then $P' \in \mathcal{S}^{\tilde{r}}$;
2. if $P \in \mathcal{S}^{\tilde{r}}$ and $P \xrightarrow{m} P'$ where $m \in M$ then $P' \in \mathcal{S}^{\tilde{r},\psi(m)}$;
3. if $P \in \mathcal{S}^{\tilde{r}}$ and $P \xrightarrow{r} P'$ where $r \in R$ then $r \in \tilde{r}$ and $P' \in \mathcal{S}^{\tilde{r}-r}$.

Further, $\mathcal{S}$ is (M, R, ψ)-*disciplined* if it is (M, R, ψ)-tidy and

4a. if $P \in \mathcal{S}^{r}$ (where r is a singleton multiset) and $P \xrightarrow{m}$ where $m \in M$ then $P \Longrightarrow\xrightarrow{r}$;

and $\mathcal{S}$ is (M, R, ψ)-*ready* if it is (M, R, ψ)-tidy and

4b. if $P \in \mathcal{S}^{\tilde{r}}$ and $r \in \tilde{r}$ then $P \xrightarrow{r}$.

To grasp the motivation for this definition consider the agent T from the Preliminaries. An (M, R, ψ)-tidy partition is to capture the relationship between method invocations and the corresponding returns. One can view each derivative T' of T as a tree-structured collection of agents each of which represents a tree node. Each time a method is invoked, via $ins(x, w, r)$ or $srch(x, r)$, a return link (r) which does not occur free in T' is received. This name will occur free in each derivative of T' until the result of the call is returned. Now the notion of (M, R, ψ)-tidy partition express the relationship between $ins(x, w, r), r$ and $srch(x, r), r$ by setting $\psi(ins(x, w, r)) = \overline{r}$ and $\psi(srch(x, r)) = \overline{r}$. In more general terms, we can view ψ as an association between names of the distinguished sets M and R such that $\psi(m) \in R$ is the companion action, in some sense, of $m \in M$. An (M, R, ψ)-tidy partition of $\mathcal{S}$ divides its agents, all of which are required to be R-confluent, into classes whose indices record the outstanding companion actions of their elements; conditions 1–3 ensure that this interpretation is accurate. Condition 4b stipulates that an agent must be able to engage immediately in any of its outstanding companion actions. (In the case of the object example this corresponds to the property that when an object invokes a method, its activity is suspended as it awaits the return of the result of that invocation.) Finally, condition 4a requires that if an agent has one outstanding companion action r and it may initiate another activity via an action in M, then it may also perform r, possibly after some τ-actions which, however, do not change its $\approx_b$-state. The purpose of this condition is to ensure, in conjunction with the others, that in certain contexts the behaviour of the agent is indistinguishable from that of an agent obtained by pruning parts of its state space. This is explained in detail in the theorem which follows the definition of the pruning operation.

Definition 3.5 Given a labelled transition system $\mathcal{T}$ and a subset W of its set of points, $\mathcal{T}\lceil W$ is the system obtained by removing all points not in W and all arrows incident on such points. If $P \in W$ we write $\hat{P}$ for the corresponding point of $\mathcal{T}\lceil W$.

We now have the main result in the CCS case.

Theorem 3.6 Suppose M, R are disjoint sets of names and $\psi : M \to R$, and define $\overline{\psi} : \overline{M} \to \overline{R}$ by setting $\overline{\psi}(\overline{m}) = \overline{\psi(m)}$. Suppose $\mathcal{P}$ is an (M, R, ψ)-ready system with (M, R, ψ)-tidy partition $\{\mathcal{P}^{\overline{r}}\}_{\overline{r}}$, and $\mathcal{T}$ an $(\overline{M}, \overline{R}, \overline{\psi})$-disciplined system with $(\overline{M}, \overline{R}, \overline{\psi})$-tidy partition $\{\mathcal{T}^{\overline{r}}\}_{\overline{r}}$. Suppose $P \in \mathcal{P}^{\emptyset}$, $T \in \mathcal{T}^{\emptyset}$ and $M \cup R \subseteq L$. Let $\widehat{T}$ be the agent corresponding to T in $\mathcal{T}\lceil(\mathcal{T}^{\emptyset} \cup \bigcup\{\mathcal{T}^{r} \mid r \text{ a singleton multiset}\})$. Then $(P|T)\backslash L \approx_b (P|\widehat{T})\backslash L$.

Proof: Let $(S_1, S_2) \in \mathcal{B}^0$ if $S_1 = (P_1|T_1)\backslash L$ and $S_2 = (P_2|\widehat{T_2})\backslash L$ where $P_2 \in \mathcal{P}^{\emptyset}$, $T_2 \in \mathcal{T}^{\emptyset}$, $P_1 \xrightarrow{s}_{\approx} P_2$, $T_1 \xrightarrow{\overline{s}}_{\approx} T_2$ with $s, \overline{s} \in R^*$, $s = r_1 \ldots r_n$, $\overline{s} = \overline{r_1} \ldots \overline{r_n}$, and $M \cup R \subseteq L$. Let also $(S_1, S_2) \in \mathcal{B}^1$ if $S_1 = (P_1|T_1)\backslash L$ and $S_2 = (P_2|\widehat{T_2})\backslash L$ where $P_2 \in \mathcal{P}^{r}$, $T_2 \in \mathcal{T}^{\overline{r}}$, $P_1 \xrightarrow{s}_{\approx} P_2$, $T_1 \xrightarrow{\overline{s}}_{\approx} T_2$ with $s, \overline{s} \in R^*$, $s = r_1 \ldots r_n$, $\overline{s} = \overline{r_1} \ldots \overline{r_n}$, and $M \cup R \subseteq L$. Then $\mathcal{B}^0 \cup \mathcal{B}^1 \cup \approx_b$ is a branching bisimulation. The most interesting part of the proof is as follows.

Suppose $(S_1, S_2) \in \mathcal{B}^1$ where $S_1 = (P_1|T_1)\backslash L$ and $S_2 = (P_2|\widehat{T_2})\backslash L$ are as in the definition with $P_2 \in \mathcal{P}^{r}$ and $T_2 \in \mathcal{T}^{\overline{r}}$. Suppose $S_1 \xrightarrow{\tau} Q_1$, and $Q_1 \equiv (P_1' \mid T_1')\backslash L$ where $P_1 \xrightarrow{m} P_1'$ and $T_1 \xrightarrow{\overline{m}} T_1'$ where $m \in M$. Let $\mathcal{T}^1$ be $\mathcal{T}^{\emptyset} \cup \bigcup\{\mathcal{T}^{\overline{r}}\}_{r}$.

Since $T_1 \xrightarrow{\overline{s}}_{\approx} T_2$, by Lemma 3.2 there are T_0, T_2' such that $T_2 \Longrightarrow T_0 \xrightarrow{\overline{m}} T_2'$ and $T_1' \xrightarrow{\overline{s}}_{\approx} T_2'$ with $T_0 \approx_b T_2$. Since $\mathcal{T}$ is $(\overline{M}, \overline{R}, \overline{\psi})$-disciplined, $T_0 \in \mathcal{T}^{\overline{r}}$ and $m \in M$, $T_0 \Longrightarrow\xrightarrow{\overline{r}}$. Hence $T_2 \Longrightarrow\xrightarrow{\overline{r}} T_3$ for some T_3. Hence as T_2 is R-confluent, $T_3 \Longrightarrow T_4 \xrightarrow{\overline{m}} T_5$ and $T_2' \xrightarrow{\overline{r}}_{\approx} T_5$ for some T_4 and T_5 with $T_4 \approx_b T_3$. Now $T_3, T_4 \in \mathcal{T}^{\emptyset}$ and $T_5 \in \mathcal{T}^{\overline{r'}}$ where $r' = \psi(m)$, so the transitions $\widehat{T_2} \Longrightarrow\xrightarrow{\overline{r}} \widehat{T_3} \Longrightarrow \widehat{T_4} \xrightarrow{\overline{m}} \widehat{T_5}$ exist in $\mathcal{T}\lceil\mathcal{T}^1$.

Now since $P_1 \xrightarrow{s}_{\approx} P_2$, by Lemma 3.2 there are P_0 and P_2' such that $P_2 \Longrightarrow P_0 \xrightarrow{m} P_2'$, $P_0 \approx_b P_2$ and $P_1' \xrightarrow{s}_{\approx} P_2'$. Because $\mathcal{P}$ is (M, R, ψ)-ready and $P_2 \in \mathcal{P}^{r}$, $P_2 \xrightarrow{r} P_3$ for some P_3. Since P_2 is R-confluent, $P_3 \Longrightarrow P_4 \xrightarrow{m} P_5$ and $P_2' \xrightarrow{r}_{\approx} P_5$ for some P_4 and P_5 with $P_4 \approx_b P_3$. Thus $(P_2|\widehat{T_2})\backslash L \Longrightarrow\xrightarrow{\tau} (P_3|\widehat{T_3})\backslash L \Longrightarrow (P_4|\widehat{T_4})\backslash L \xrightarrow{\tau} (P_5|\widehat{T_5})\backslash L$.

It remains to note that by the construction $(S_1, (P_4 \mid \widehat{T_4})\backslash L) \in \mathcal{B}^0$ and $(Q_1, (P_5 \mid \widehat{T_5})\backslash L) \in \mathcal{B}^1$. These claims follow from Lemma 3.2. $\square$

4 Partial confluence in the π_v-calculus

We now generalize the theory presented in the previous section.

Notation 4.1 (a) We write $\mathsf{subj}(\alpha)$ for the subject of the action α. If $s = \alpha_1 \ldots \alpha_n$ is a sequence of actions then $\mathsf{subj}(s) = \mathsf{subj}(\alpha_1) \ldots \mathsf{subj}(\alpha_n)$.

(b) Let R be a sort. We write R^+ (resp. R^-) for the set of actions with a positive (resp. negative) subject whose name is in R, and $R^{\pm}$ for $R^+ \cup R^-$.

Definition 4.2 Let R be a sort. A process P is *R-confluent* if for every derivative Q of P:

1. if $\rho \in R^{\pm}$, $\mathsf{subj}(\alpha) \neq \mathsf{subj}(\rho)$, $Q \xrightarrow{\rho} Q_1$ and $Q \Longrightarrow\xrightarrow{\alpha} Q_2$ then for some Q', $Q_1 \Longrightarrow\xrightarrow{\alpha} Q'$ and $Q_2 \Longrightarrow\xrightarrow{\rho}\dot{\approx}_b Q'$;
2. if $\rho_1, \rho_2 \in R^-$, $\mathsf{subj}(\rho_1) = \mathsf{subj}(\rho_2)$, $Q \xrightarrow{\rho_1} Q_1$ and $Q \Longrightarrow\xrightarrow{\rho_2} Q_2$ then $\rho_1 = \rho_2$ and $Q_1 \dot{\approx}_b Q_2$;
3. if $\rho \in R^+$, $Q \xrightarrow{\rho} Q_1$ and $Q \Longrightarrow\xrightarrow{\rho} Q_2$ then $Q_1 \dot{\approx}_b Q_2$.

The theory presented in the previous section can be generalized, with some changes, to the new setting. In the space available, however, we can only state the main definition and theorem.

We first formulate analogues of the notions "(M, R, ψ)-tidy" etc. In the CCS case the purpose of the function ψ was to record an association between names in M and names in R. In the present setting this is achieved in a different way, namely via a sorting which associates with the sort M a tuple of sorts of the form $(\widetilde{S}, R, \widetilde{S'})$ where R does not occur in the tuples $\widetilde{S}$ and $\widetilde{S'}$. Then if $\alpha \in M^{\pm}$, the name of the subject of α is m and the component, $\mathsf{obj}_R(\alpha)$, of the object of α of sort R is r, then m and r are associated. It is appropriate also to reflect in the following definition the asymmetric nature of communication in the π_v-calculus. Thus we define "(M^-, R^+)-tidy" (with requirements on output actions whose subjects have names in M and input actions with subjects in R) rather than "(M, R)-tidy" etc.

Definition 4.3 Suppose M and R are distinct sorts and the sorting Σ is such that $\Sigma(M) = (\widetilde{S}, R, \widetilde{S'})$ for some tuples of sorts $\widetilde{S}$ and $\widetilde{S'}$ not containing R and R occurs in no other object sort. A derivation-closed set $\mathcal{S}$ of R-confluent processes is *(M^-, R^+)-tidy* if there is a partition $\{\mathcal{S}^{\tilde{r}} \mid \tilde{r}$ a finite subset of R$\}$ of $\mathcal{S}$ (an *(M^-, R^+)-tidy partition*) such that:

1. if $P \in \mathcal{S}^{\tilde{r}}$ and $P \xrightarrow{\alpha} P'$ where $\alpha \notin M^- \cup R^+$ then $P' \in \mathcal{S}^{\tilde{r}}$;
2. if $P \in \mathcal{S}^{\tilde{r}}$ and $P \xrightarrow{\alpha} P'$ where $\alpha \in M^-$ and $r \notin \tilde{r}$ where $r = \mathsf{obj}_R(\alpha)$ then $P' \in \mathcal{S}^{\tilde{r},r}$;
3. if $P \in \mathcal{S}^{\tilde{r}}$ and $P \xrightarrow{\alpha} P'$ where $\alpha \in R^+$ then $\mathsf{subj}(\alpha) = r \in \tilde{r}$ and $P' \in \mathcal{S}^{\tilde{r}-r}$.

Further, $\mathcal{S}$ is *(M^-, R^+)-ready* if it is (M^-, R^+)-tidy and

4a. if $P \in \mathcal{S}^{\tilde{r}}$ and $r \in \tilde{r}$ then $P \xrightarrow{\alpha}$ for any $\alpha \in R^+$ with $\mathsf{subj}(\alpha) = r$.

Similarly, we define *(M^+, R^-)-tidy (partition)* and say $\mathcal{S}$ is *(M^+, R^-)-disciplined* if it is (M^+, R^-)-tidy with (M^+, R^-)-tidy partition $\{\mathcal{S}^{\tilde{r}}\}_{\tilde{r}}$ and

4b. if $P \in \mathcal{S}^r$ (where r is a singleton multiset) and $P \xrightarrow{\alpha}$ where $\alpha \in M^+$ then $P \Longrightarrow\xrightarrow{\beta}$ for some $\beta \in R^-$ with $\mathsf{subj}(\beta) = \overline{r}$.

Theorem 4.4 Suppose M and R are distinct sorts and the sorting Σ is such that $\Sigma(M) = (\widetilde{S}, R, \widetilde{S'})$ for some tuples of sorts $\widetilde{S}$ and $\widetilde{S'}$ not containing R and R occurs in no other object sort. Suppose $\mathcal{P}$ is an (M^-, R^+)-ready set with (M^-, R^+)-tidy partition $\{\mathcal{P}^{\widetilde{r}}\}_{\widetilde{r}}$, and $\mathcal{T}$ an (M^+, R^-)-disciplined set with (M^+, R^-)-tidy partition $\{\mathcal{T}^{\widetilde{r}}\}_{\widetilde{r}}$. Suppose $P \in \mathcal{P}^{\emptyset}$, $T \in \mathcal{T}^{\emptyset}$ and no derivative of $(\nu\widetilde{z})(P \mid T)$ contains a free occurrence in subject position of a name of sort R or of sort M. Let $\widehat{T}$ be the process corresponding to T in $\mathcal{T}\lceil(\mathcal{T}^{\emptyset} \cup \bigcup\{\mathcal{T}^r \mid r \text{ a singleton multiset}\})$. Then $(\nu\widetilde{z})(P|T) \stackrel{\cdot}{\approx}_b (\nu\widetilde{z})(P|\widehat{T})$.

5 An example

In Section 2.4 we defined agents T_0 and T which are simplified encodings of the symbol table classes T0 and T respectively. To illustrate the theory of partial confluence we now use it to establish the equivalence of T_0 and T in the encoding of an arbitrary program context, thus proving the interchangeability of the symbol table classes T0 and T as discussed in the Introduction.

Suppose $\mathcal{C}[\cdot]$ is the encoding of a program context into which an abstraction of sort (LINK[T]), the sort of the agents encoding the classes, may be placed. Then $\mathcal{C}[\cdot]$ is of the form $(\nu new)(P \mid \cdot\langle new\rangle)$ where P is the encoding of the other classes and the trigger of the program. Since the classes have more than one method, it is necessary to generalize the definitions of "(M^-, R^+)-tidy" etc. to the case when instead of single sorts M and R we have tuples $M = M_1 \ldots M_n$ and $R = R_1 \ldots R_n$ of distinct sorts where the sorting Σ is such that for each i, $\Sigma(M_i) = (\widetilde{S_i}, R_i, \widetilde{S_i'})$ for some tuples $\widetilde{S_i}$ and $\widetilde{S_i'}$. This is straightforward. The following theorem expresses the equivalence of the class definitions.

Theorem 5.1 Let $\mathcal{C}[\cdot] = (\nu new)(P \mid \cdot\langle new\rangle)$ be the translation of an arbitrary program context and T_0, T as in the Preliminaries. Then $\mathcal{C}[T] \approx_b \mathcal{C}[T_0]$.

PROOF: We want to show that $(\nu new)(P \mid T_0\langle new\rangle) \approx_b (\nu new)(P \mid T\langle new\rangle)$. Let $\mathcal{T}$ be the process system generated by $T_0\langle new\rangle$ and $T\langle new\rangle$, and $\mathcal{P}$ the process system generated by P. Let M be the pair of sorts $\mathsf{METH}^I_\mathsf{T}[\mathsf{NAT}, \mathtt{ref(A)}; \mathsf{UNIT}]$, $\mathsf{METH}^S_\mathsf{T}[\mathsf{NAT}; \mathtt{ref(A)}]$ and R the pair $\mathsf{RES}^I_\mathsf{T}[\mathsf{UNIT}]$, $\mathsf{RES}^S_\mathsf{T}[\mathtt{ref(A)}]$ and consider the generalized definitions of "(M^+, R^-)-tidy" etc. The proof involves showing:

1. $\mathcal{P}$ is (M^-, R^+)-ready.
2. $\mathcal{T}$ is (M^+, R^-)-disciplined.
3. $T_0\widehat{\langle new\rangle} \approx_b T\widehat{\langle new\rangle}$, where $T_0\widehat{\langle new\rangle}$ and $T\widehat{\langle new\rangle}$ are the states corresponding to $T_0\langle new\rangle$ and $T\langle new\rangle$ respectively in $\mathcal{T}\lceil(\mathcal{T}^{\emptyset} \cup \bigcup\{\mathcal{T}^r \mid r \text{ a singleton multiset}\})$.

By 1 and 2 above, applying the main theorem in the previous section we have

$$\begin{array}{lcll} (\nu new)(P \mid T_0\langle new\rangle) & \stackrel{\cdot}{\approx}_b & (\nu new)(P \mid T_0\widehat{\langle new\rangle}) & \text{and} \\ (\nu new)(P \mid T\langle new\rangle) & \stackrel{\cdot}{\approx}_b & (\nu new)(P \mid T\widehat{\langle new\rangle}). & \end{array}$$

Together with 3 and the observation that by the nature of the translation the theorem's condition on names of sort M or R is met, these facts complete the proof.

In the space available we can only sketch the proof. We consider first 1. The names of sort R are return links for invocations via names of sort M. In the translation the use of these names has a very strict pattern: $\overline{m}(\tilde{p}, \nu r)$ where $m : M$ is always followed by an action with subject $r : R$, and an action with subject $r : R$ is always preceded by $\overline{m}(\tilde{p}, \nu r)$ where $m : M$. As illustrated in the Preliminaries, P is a restricted composition and it follows that any derivative of it must (up to structural congruence) have the form

$$Q \equiv (\nu\tilde{p})\,(r_1(x_1).P_1 \mid \ldots \mid r_n(x_n).P_n \mid r'_1.P'_1 \mid \ldots \mid r'_m.P'_m \mid Q') \qquad (1)$$

where $r_1, \ldots, r_n$: $\mathsf{RES}^S_\mathsf{T}[\mathtt{ref(A)}]$ and $r'_1, \ldots, r'_m$: $\mathsf{RES}^I_\mathsf{T}[\mathsf{UNIT}]$ are pairwise distinct (as private names are used for returning results), and for any action $\alpha \in R^+$, $Q' \not\xrightarrow{\alpha}$, $P_i \not\xrightarrow{\alpha}$, $P'_i \not\xrightarrow{\alpha}$. As each of its elements has this form, $\mathcal{P}$ is an R-confluent process system. Now let $\{\mathcal{P}^{\tilde{r}} \mid \tilde{r}$ a finite subset of $R\}$ be the partition of $\mathcal{P}$ defined by setting $Q \in \mathcal{P}^{\tilde{r}}$ for Q of form (1) if $\tilde{r} = \{r_1, \ldots, r_n, r'_1, \ldots, r'_m\}$. It may be checked that this partition is (M^-, R^+)-tidy. It is clear that $\mathcal{P}$ is (M^-, R^+)-ready.

For $T_0\langle new\rangle$ and $T\langle new\rangle$ we can obtain the general form of the derivatives by analyzing their syntax. We may then apply the technique of unique solution of process equations to prove that $T_0\langle new\rangle \approx_b T\langle new\rangle$. Now by the translation no derivative of $\mathcal{C}[T] \equiv (\nu new)(P \mid T\langle new\rangle)$ contains a free occurrence in subject position of a name of sort M or R. Hence from 1, 2 and 3 above it follows by the generalized version of the main theorem of the preceding section that $\mathcal{C}[T_0] \mathrel{\dot{\approx}_b} \mathcal{C}[\widehat{T_0}]$ and $\mathcal{C}[T] \mathrel{\dot{\approx}_b} \mathcal{C}[\widehat{T}]$ and hence $\mathcal{C}[T] \mathrel{\dot{\approx}_b} \mathcal{C}[T_0]$. Moreover since out is the only free name it follows that $\mathcal{C}[T] \approx_b \mathcal{C}[T_0]$. □

6 Conclusion

The notion of partial confluence introduced here is worthy of further investigation. In addition to the intrinsic interest of the theory it may be useful in e.g. the study of concurrency control in databases [15]. As a further example we intend to study concurrent operations on binary search trees and B-trees as presented in e.g. [9, 10]. Also of interest are connections with non-interleaving semantics of concurrent systems and action/process refinement as in e.g. [3].

The proof presented in Section 5 can be viewed, as mentioned in the Introduction, as establishing the soundness of particular instances of general transformation rules such as those proposed in [6]. Those rules stipulate conditions under which (in the notation of the present paper) a statement of the form S; $\mathsf{return}\,S'$ may be replaced by $\mathsf{return}\,S'; S$, and a statement of the form $\mathsf{return}\,X!M(Y)$ by $\mathsf{commit}\,X!M(Y)$. These conditions refer to termination of (executions of) statements and to properties of a distinguished class of program variables called "private references". No semantic account of "private references" if given in [6], nor is it stipulated how they may be used in programs. It appears that to satisfy the side conditions of the rules in question, quite severe syntactic restrictions may be necessary. Moreover the requirement to establish termination of (executions of) statements which may invoke methods in systems of objects with dynamically-evolving structure may be very demanding. The theory developed in this paper provides a framework, at an appropriate level of abstraction, for reasoning rigorously about the behaviours of systems of objects whose reference structures are (forests of) trees. It remains a tough challenge to provide a comparable framework on which to base a proof of the soundness of general transformation rules whose side conditions are not unduly restrictive or intractable.

References

[1] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1:366–411, 1989.

[2] U. Engberg and M. Nielsen. A calculus of communicating systems with label passing. Technical report, University of Aarhus, 1986.

[3] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In *CONCUR'91*, pages 298–316. Springer, 1991.

[4] C. Jones. Constraining interference in an object-based design method. In *Proceedings of TAPSOFT'93*, pages 136–150, 1993.

[5] C. Jones. A pi-calculus semantics for an object-based design notation. In *Proceedings of CONCUR'93*, pages 158–172, 1993.

[6] C. Jones. Process-algebraic foundations for an object-based design notation. Technical report, University of Manchester, 1993.

[7] C. Jones. Process algebra arguments about an object-based design method. In *Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[8] D. Kafura and R. G. Lavender. Concurrent object-oriented languages and the inheritance anomoly. In T. Casavant, editor, *Parallel Computers: Theory and Practice*. Computer Society Press, to appear.

[9] H. Kung and P. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5:354–382, 1980.

[10] P. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6:650–670, 1981.

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

[12] R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification.* Springer, 1992.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts 1 and 2. *Information and Computation*, 100:1–77, 1992.

[14] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993.

[15] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Society Press, 1986.

[16] M. Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming.* PhD thesis, University of Geneva, 1992.

[17] A. Philippou. PhD thesis, University of Warwick, forthcoming.

[18] M. Sanderson. *Proof techniques for CCS.* PhD thesis, University of Edinburgh, 1982.

[19] C. Tofts. *Proof methods and pragmatics for parallel programming.* PhD thesis, University of Edinburgh, 1990.

[20] R. van Glabbeek and P. Weijland. Branching time and abstraction in bisimulation semantics. In *Information Processing '89*, pages 613–618, 1989.

[21] D. Walker. π-calculus semantics for object-oriented programming languages. In *Proceedings of TACS'91*, pages 532–547. Springer, 1991.

[22] D. Walker. Algebraic proofs of properties of objects. In *Proceedings of ESOP'94*, pages 501–516. Springer, 1994.

[23] D. Walker. Objects in the π-calculus. *Information and Computation*, to appear.

[24] D. Walker. Process calculus and parallel object-oriented programming lanaguages. In T. Casavant, editor, *Parallel Computers: Theory and Practice.* Computer Society Press, to appear.