Reversing Algebraic Process Calculi

Iain Phillips[1] and Irek Ulidowski[2]

[1] Department of Computing, Imperial College London, England
iccp@doc.ic.ac.uk

[2] Department of Computer Science, University of Leicester, England
iu3@mcs.le.ac.uk

Abstract. Reversible computation has a growing number of promising application areas such as the modelling of biochemical systems, program debugging and testing, and even programming languages for quantum computing. We formulate a procedure for converting operators of standard algebraic process calculi such as CCS, ACP and CSP into reversible operators, while preserving their operational semantics.

1 Introduction

Reversible computation has a growing number of promising application areas such as the modelling of biochemical systems [8], program debugging and testing [20], and even programming languages for quantum computing [2]. Landauer [15] showed how irreversible computation generates heat; the efficient operation of future miniaturised computing devices could depend on exploiting reversibility [7]. We have been inspired to look at this area by the work of Danos and Krivine on reversible CCS [8, 9, 10] and Abramsky on mapping functional programs into reversible automata [1].

We wish to investigate reversibility for algebraic process calculi in the style of CCS [16], with Structural Operational Semantics (SOS) [19] rules. Given a forward *labelled transition relation* (ltr) $\rightarrow$ we are interested in obtaining a *reverse ltr* $\rightsquigarrow$ which is the inverse of $\rightarrow$. This can always be done, but if we just reverse a standard process language we end up with too many possibilities, since processes do not "remember" their past states. Danos and Krivine solve this problem by storing "memories" of past behaviour which are carried along with processes. Memories also keep track of which thread or threads performed an action. This has the effect that backtracking does not have to follow the exact order of forward computation in reverse. To take a simple example, suppose that the process $a.b \mid c$ performs a followed by b and then c (here "." and "|" are the prefixing and the parallel composition of CCS, respectively). The process can backtrack by reversing b, then a and finally c. However, a cannot be reversed before b has been reversed.

We wish to produce reversible process calculi without relying on external devices such as memories. Our starting point is that irreversibility in a language such as CCS comes from the consumption of guards and alternative choices. We therefore decide to leave these in place, so that process structure remains fixed throughout a computation. Returning to the example of $a.b \mid c$, we might let the state after a, b and c have been performed be denoted by $\underline{a}.\underline{b} \mid \underline{c}$, where the underlined actions are *past* actions. There is plainly just the right amount of information here to reverse the process, while allowing $\underline{c}$ to be reversed independently of $\underline{b}$ and $\underline{a}$. This approach allows us also to keep track of unused alternatives discarded during computation. Consider $a.b + c$, where "+" is the choice operator of CCS. After the initial a, the alternative c is discarded and we can only proceed with b. This state is represented as $\underline{a}.b + c$; it is clear which alternative was taken and what will happen next.

Reversibility can help to some extent to distinguish concurrency from causation. In the reversible world, Milner's expansion law does not hold: we have $a \mid b \neq a.b + b.a$ since $a \mid b \xrightarrow{a}\xrightarrow{b} \underline{a} \mid \underline{b} \overset{a}{\rightsquigarrow}$ but $a.b + b.a \xrightarrow{a}\xrightarrow{b} \underline{a}.\underline{b} + b.a \overset{a}{\not\rightsquigarrow}$.

When we come to consider autoconcurrency and communication we find that the simple method just outlined arguably discards too much information. For instance, the processes $a \mid a$ and $a.a$ cannot be distinguished by the above argument, as we are not able to tell apart the two occurrences of a. Moreover, the process $a \mid \overline{a}$ can evolve by a communication between a and the complementary action $\overline{a}$ to yield $\underline{a} \mid \underline{\overline{a}}$. This state could also have been reached by performing the two actions separately, and there is nothing in the notation to stop us from undoing the two actions separately. But if the communication represents a binding between two (biological) entities, then such separate backtracking of a and $\overline{a}$ is not reasonable.

Our solution is to use a more expressive form of past actions, where each occurrence of action a is "marked" by a fresh identifier m and written as $a[m]$. Also, we insist that the two parties to a communication between action a and $\overline{a}$ agree on this identifier or *communication key* which is unique to that communication. This means that a and $\overline{a}$ are now locked together and can only be undone together. Now, we can deal with the autoconcurrency example. Process $a \mid a$ can perform the a actions with keys m and n to produce $a[m] \mid a[n]$, and then reverse these actions in any order. However, $a.a$ cannot match this behaviour: after the a actions with keys m and n, the process $a[m].a[n]$ cannot reverse on $a[m]$.

We propose a method for reversing process operators that are definable by SOS rules in a general format. As far as we are aware, this is the first time this has been done for algebraic process calculi. As we have described informally above, we rely on reformulating operators of standard process calculi into new operators that can be easily reversed, while preserving their operational meaning. In this paper we attempt to balance the generality of the format on one hand and the technical simplicity of the proposed method on the other hand. The chosen format is general enough for the definitions of the majority of useful process operators, and the method presented is intuitive and easy to apply.

Our format is a subformat of the *path* format [3] and consists of *dynamic* rules, where the operator is destroyed by a transition, and *static* rules, where the operator remains present after the transition. Reversing static rules is easier because they preserve the context during execution. Dynamic rules, however, consume the context, removing the unused alternatives. The kernel of our method is to transform dynamic rules into static-like rules. *Auxiliary operators* and *predicates* are used to keep the structure of terms unchanged and to enforce correct use of subterms in the reformulated contexts. Once SOS rules for operators are reformulated as above, the reverse SOS rules are obtained simply as symmetric versions of the forward rules.

As an illustration of the method we consider the CCS choice operator +. We reformulate it as a static operator and use predicate std, meaning that the argument is a standard term that uses no past actions (and no keys), to control when arguments can fire in rules. The reverse rules (on the right) for the converted + are then obtained by symmetry:

$$\frac{X \xrightarrow{a} X' \quad \mathsf{std}(Y)}{X + Y \xrightarrow{a} X' + Y} \quad \frac{Y \xrightarrow{a} Y' \quad \mathsf{std}(X)}{X + Y \xrightarrow{a} X + Y'} \quad \frac{X \overset{a}{\rightsquigarrow} X' \quad \mathsf{std}(Y)}{X + Y \overset{a}{\rightsquigarrow} X' + Y} \quad \frac{Y \overset{a}{\rightsquigarrow} Y' \quad \mathsf{std}(X)}{X + Y \overset{a}{\rightsquigarrow} X + Y'}$$

We prove a number of results to show that our method yields well-behaved transition relations. We show that the new forward ltr is conservative over the standard ltr (Theorem 5.8). Also the new forward and reverse ltrs satisfy certain confluence properties (Propositions 5.4 and 5.5). The processes which are reachable from standard processes by forward-only transitions are closed under reverse transitions, meaning that a process can never reverse into an "inconsistent" past (Proposition 5.6). We also formulate a notion of *forward-reverse bisimulation*, which is a congruence (Theorem 6.7).

The rest of the paper is structured as follows. In Section 2 we define the simple process calculi which we shall be making reversible, and in Section 3 we describe our procedure for generating the new reversible calculi. In Section 4 we illustrate our method by applying it to CCS. We also discuss related work, and in particular RCCS [9]. In Section 5 we prove various results about the new reversible transition relations, and in Section 6 we define an appropriate notion of bisimulation. Section 7 indicates how to adapt the method to a more general format that contains constants and predicates. We end with some conclusions.

The proofs of the presented results, further results, examples and discussion are available in the full version of this paper [18].

2 Process Calculi

In this section we describe the process calculi to which we shall apply our procedure for generating reversible calculi.

A *signature* is a set Σ of operator symbols, each with a particular arity. The set of *terms* over Σ is denoted by $T(\Sigma)$. We shall tend to refer to terms as *processes*. We let $P, Q, \ldots$ range over processes.

A *process calculus* $L = (\Sigma, A, R)$, is given by a signature Σ, a set of actions A and a set R of SOS rules. We shall apply our procedure to a "standard" calculus $L_S = (\Sigma_S, \mathsf{Act}, R_S)$. Its terms are called *standard* terms and are denoted by Std. We shall assume that the only operator of arity zero (i.e. constant) is the deadlocked process $\mathbf{0}$. We let $f, \ldots$ range over Σ_S; $a, b, c, \ldots$ range over Act.

We next describe the rules R and their operational semantics.

The SOS theory gives us the flexibility and the benefits of working with whole classes of process calculi rather than with individual process calculi that are limited to a small number of operators. Typically, a class of operators is defined by a format of SOS rules that can be used to define them operationally. In this paper we shall consider simple *path* rules without copying [3]. More specifically, our rules will be mostly of the simpler *pxyft* and *pxyf* forms, where terms in the premises are variables and the source of the conclusion is a term constructed with a single operator.

Definition 2.1. Simple path *(forward) rules are expressions of the form*

$$\frac{\{\ X_i \xrightarrow{a_i} X'_i\ \}_{i\in I} \quad \{\ \mathsf{p_j}(X_j)\ \}_{j\in J}}{f(X_1,\ldots,X_n) \xrightarrow{a} t(X'_1,\ldots,X'_n)} \qquad \textit{and} \qquad \frac{\{\ \mathsf{p_j}(X_j)\ \}_{j\in J}}{\mathsf{p}(f(X_1,\ldots,X_n))}$$

where all variables X_i (X_j) and X'_i are distinct, and variables X'_i are such that $X'_i = X_i$ when $i \notin I$. Moreover, $I, J \subseteq \{1,\ldots,n\}$.

The sets of transitions and predicate expressions above the horizontal bars in the rules above are called premises*. Let r be the first rule above. Operator f is the* operator *of r. The transition below the bar in r is the* conclusion *of r. Action a in the conclusion is the* action *of r and $f(X_1,\ldots,X_n)$ and $t(X'_1,\ldots,X'_n)$ are the* source *and* target *of r, respectively. The i-th argument is* active *in r if r has a transition for X_i in the premises. The i-th argument of f is* active *if it is active in some rule for f. In the second rule, p is the* predicate *of the rule and the predicate expression below the bar is the* conclusion.

With any calculus $L = (\Sigma, A, R)$, all of whose rules are in simple *path* format, we associate an ltr $\rightarrow$ with labels A, together with a set of predicates, in the standard way; for details see [3]. Our standard calculus L_{S} will have all its rules R_{S} in simple *path* format. It will have no predicates in its rules. We shall write its ltr as $\rightarrow_{\mathrm{S}}$, and use this in writing down its rules for clarity.

We now define the precise form of SOS rules that operators of L_{S} can have. Consider an n-ary operator $f \in \Sigma_{\mathrm{S}}$ ($n \geq 1$). The set of arguments of f is $N_f = \{1,\ldots,n\}$. Operator f can have three kinds of rules: static rules, choice rules and choice axioms. We describe each in turn.

Definition 2.2. Static rules *of f are of the following form, where $I \neq \emptyset$:*

$$\textit{(I)} \quad \frac{\{X_i \xrightarrow{a_i}_{\mathrm{S}} X'_i\}_{i\in I}}{f(\overrightarrow{X}) \xrightarrow{a}_{\mathrm{S}} f(\overrightarrow{X'})}$$

We require that if two static rules for f have the same premises then they have the same conclusion (i.e. the action of the conclusion is unique). Let $S_f \subseteq N_f$ be the set of all arguments occurring in the premises of static rules of f, and let $E_f = N_f \setminus S_f$. Arguments in S_f are called static arguments.

The arguments of the CCS and CSP [14] parallel composition operators are static, as are those of the CCS restriction and relabelling operators and the CSP hiding operator.

Next we describe the choice rules.

Definition 2.3. *A* choice rule *of f is a rule of the following form:*

$$\textit{(II)} \quad \frac{X_d \xrightarrow{a}_{\mathrm{S}} X'_d}{f(\overrightarrow{X}) \xrightarrow{a}_{\mathrm{S}} X'_d}$$

We require that $d \in E_f$. Let D_f be the set of all arguments d occurring in the premises of choice rules of f. Arguments in D_f are called dynamic *arguments. Each dynamic argument d is required to be* permissive*, meaning that for each $a \in \mathsf{Act}$ there is a rule of type (II).*

Note that $D_f \subseteq E_f$, so that a dynamic argument cannot be static.

The choice operator of CCS has two dynamic arguments, both of which are permissive. The external choice operator of CSP also has two dynamic arguments, but they are not permissive: although they have choice rules for all $a \in \mathsf{Act} \setminus \{\tau\}$, they have no such rules for the τ—the rules for τ are static (see Section 7).

We also wish to encompass operators that have choice rules with empty premises such as, for example, CCS prefixing and CSP internal choice. This leads us to the third and final type of rule:

Definition 2.4. *A* choice axiom *of f is a rule r of the following form:*

$$\textit{(III)} \quad r \; \frac{}{f(\overrightarrow{X}) \xrightarrow{\mathsf{act}(r)}_{\mathrm{S}} X_{\mathsf{ta}(r)}}$$

Here $\mathsf{ta}(r)$ *is the* target argument*. We require* $\mathsf{ta}(r) \in E_f$*.*

Next, we define the class of simple process calculi that we shall reverse.

Definition 2.5. *A process operator f is* simple *if either f is the deadlocked process* $\mathbf{0}$*, or f has a nonzero arity and all its rules are as in Definitions 2.2, 2.3 and 2.4. A process calculus is* simple *if all its operators are simple.*

In what follows we omit the subscripts of the three sets of arguments where no confusion can arise.

We shall require that L_{S} is simple. Note that we leave out rules with predicates at this stage. This allows us to keep the presentation side of the work manageable. As a result, the main application of this work is to reformulate and reverse Milner's CCS, and many other operators from the process calculi ACP [4] and CSP [14] and their descendants.

3 The Procedure for Generating a Reversible Calculus

We shall transform L_{S} into an operationally equivalent calculus which is easily reversible. For this we shall need to augment the processes and reformulate the rules of L_{S}.

Let $\mathcal{K}$ be an infinite set of *communication keys* (or just *keys* for short), ranged over by $m, n, \ldots$. The set of *past actions*, or actions marked with keys, is denoted by $\mathsf{ActK} = \mathsf{Act} \times \mathcal{K}$. We write the ordered pair (a, m) as $a[m]$. We let $\mu, \ldots$ range over ActK, and $s, t, \ldots$ range over ActK^*.

We introduce the signature Σ_{A} of auxiliary operators $f_r[m]$, where r is a rule of type (III) for an operator f of R_{S}, and $m \in \mathcal{K}$. We let $\Sigma_{\mathrm{SA}} = \Sigma_{\mathrm{S}} \cup \Sigma_{\mathrm{A}}$, and let $\mathsf{Proc} = T(\Sigma_{\mathrm{SA}})$. Clearly, $\mathsf{Std} \subseteq \mathsf{Proc}$.

Our reformulation and reversing method relies on auxiliary unary predicates on Proc, namely $\mathsf{std}(P)$ and $\mathsf{fsh}[m](P)$ (all $m \in \mathcal{K}$). Informally, $\mathsf{std}(P)$ holds if $P \in \mathsf{Std}$ and $\mathsf{fsh}[m](P)$ holds if key m is fresh (i.e. not used) in P. The predicates are defined below, where the last four rules are rule schemas for all relevant operators and keys, and $m \neq n$ in the last rule schema.

$$\frac{}{\mathsf{std}(\mathbf{0})} \quad \frac{\{\mathsf{std}(X_i)\}_{i \in N}}{\mathsf{std}(f(\overrightarrow{X}))} \qquad \frac{}{\mathsf{fsh}[m](\mathbf{0})} \quad \frac{\{\mathsf{fsh}[m](X_i)\}_{i \in N}}{\mathsf{fsh}[m](f(\overrightarrow{X}))} \quad \frac{\{\mathsf{fsh}[m](X_i)\}_{i \in N}}{\mathsf{fsh}[m](f_r[n](\overrightarrow{X}))}$$

Note that if $\mathsf{std}(P)$ then $\mathsf{fsh}[m](P)$ for every $m \in \mathcal{K}$. Let R_{P} be the set of rules for the predicates std and $\mathsf{fsh}[m]$ for all $m \in \mathcal{K}$.

We define how to transform rules of type (I), (II) and (III) into rules in simple *path* format that can be easily reversed.

Definition 3.1. *For every operator f in Σ_{S}, every static rule of type (I) for f is converted into*

$$\textit{(1)} \quad \frac{\{X_i \overset{a_i[m]}{\rightarrow} X_i'\}_{i \in I} \quad \{\mathsf{std}(X_e)\}_{e \in E} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S \setminus I}}{f(\overrightarrow{X}) \overset{a[m]}{\rightarrow} f(\overrightarrow{X'})}$$

where $X_i' = X_i$ for all $i \notin I$. The reverse version is

$$\textit{(1R)} \quad \frac{\{X_i \overset{a_i[m]}{\rightsquigarrow} X_i'\}_{i \in I} \quad \{\mathsf{std}(X_e)\}_{e \in E} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S \setminus I}}{f(\overrightarrow{X}) \overset{a[m]}{\rightsquigarrow} f(\overrightarrow{X'})}$$

Note that (1) and (1R) are rule schemas for keys m. Also, $I \cap E = \emptyset$, and so predicates only apply to inactive arguments. This contributes to making our rules easily reversible. Finally note that we shall be able to prove that if $P \overset{a[m]}{\rightarrow} P'$ then $\mathsf{fsh}[m](P)$ (Lemma 5.2).

Definition 3.2. *For every operator f in Σ_{S}, every choice rule of type (II) for f is converted into*

$$\textit{(2)} \quad \frac{X_d \overset{a[m]}{\rightarrow} X_d' \quad \{\mathsf{std}(X_e)\}_{e \in E \setminus \{d\}} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S}}{f(\overrightarrow{X}) \overset{a[m]}{\rightarrow} f(\overrightarrow{X'})}$$

where $X_i' = X_i$ for all $i \neq d$. The reverse version of (2) is

$$\textit{(2R)} \quad \frac{X_d \overset{a[m]}{\rightsquigarrow} X_d' \quad \{\mathsf{std}(X_e)\}_{e \in E \setminus \{d\}} \quad \{\mathsf{fsh}[m](X_i)\}_{i \in S}}{f(\overrightarrow{X}) \overset{a[m]}{\rightsquigarrow} f(\overrightarrow{X'})}$$

Again (2) and (2R) are rule schemas for keys m, and again predicates are only applied to inactive arguments, since $d \notin S$.

In order to make operators f with rules of type (III) static we shall use auxiliary operators. These operators have their own rules (type (3′) below) which propagate the actions of a single argument leaving other arguments unchanged.

Definition 3.3. *For every operator f in Σ_{S}, every rule r of type (III) for f is converted into the rule schemas below for all $b \in \mathsf{Act}$ and keys m, n:*

$$\textit{(3)}\quad \frac{\{\mathsf{std}(X_e)\}_{e\in E} \quad \{\mathsf{fsh}[m](X_i)\}_{i\in S}}{f(\overrightarrow{X}) \xrightarrow{\mathsf{act}(r)[m]} f_r[m](\overrightarrow{X})}$$

$$\textit{(3′)}\quad \frac{X_{\mathsf{ta}(r)} \xrightarrow{b[m]} X'_{\mathsf{ta}(r)} \quad \{\mathsf{std}(X_e)\}_{e\in E\setminus\{\mathsf{ta}(r)\}} \quad \{\mathsf{fsh}[m](X_i)\}_{i\in S}}{f_r[n](\overrightarrow{X}) \xrightarrow{b[m]} f_r[n](\overrightarrow{X'})} \quad m \neq n$$

The reverse versions of rule schemas of type (3) and (3′) are

$$\textit{(3R)}\quad \frac{\{\mathsf{std}(X_e)\}_{e\in E} \quad \{\mathsf{fsh}[m](X_i)\}_{i\in S}}{f_r[m](\overrightarrow{X}) \overset{\mathsf{act}(r)[m]}{\rightsquigarrow} f(\overrightarrow{X})}$$

$$\textit{(3′R)}\quad \frac{X_{\mathsf{ta}(r)} \overset{b[m]}{\rightsquigarrow} X'_{\mathsf{ta}(r)} \quad \{\mathsf{std}(X_e)\}_{e\in E\setminus\{\mathsf{ta}(r)\}} \quad \{\mathsf{fsh}[m](X_i)\}_{i\in S}}{f_r[n](\overrightarrow{X}) \overset{b[m]}{\rightsquigarrow} f_r[n](\overrightarrow{X'})} \quad m \neq n$$

Again predicates are only applied to inactive arguments.

Now we are ready to define our procedure that reformulates standard operators and produces automatically their new forward and reverse rules. Note that all rules mentioned in Definitions 3.1, 3.2 and 3.3 are in the simple *path* format.

Definition 3.4 (Conversion Procedure). *A simple process calculus $L_{\mathrm{S}} = (\Sigma_{\mathrm{S}}, \mathsf{Act}, R_{\mathrm{S}})$ generates a* reversible process calculus with communication keys *$L = (\Sigma_{\mathrm{SA}}, \mathsf{ActK}, R_{\mathrm{F}}, R_{\mathrm{R}})$ as follows:*

1. *$\Sigma_{\mathrm{SA}} \overset{\mathrm{df}}{=} \Sigma_{\mathrm{S}} \cup \Sigma_{\mathrm{A}}$. The operators in Σ_{SA} are called* reversible *operators.*
2. *The forward rule set R_{F} is the least set such that*
   (a) $R_{\mathrm{P}} \subseteq R_{\mathrm{F}}$, where R_{P} is the set of rules for predicates defined above;
   (b) for every rule $r \in R_{\mathrm{S}}$ for f of type (I) or (II) the set R_{F} contains the converted rules r' of the corresponding type (1) or (2) as required by Definitions 3.1 and 3.2;
   (c) for every rule $r \in R_{\mathrm{S}}$ for f of type (III) the set R_{F} contains the converted rule r' of type (3), and all the rules of type (3′) for the auxiliary operators $f_r[m]$ as required by Definition 3.3.
3. *The reverse rule set R_{R} is defined like R_{F}, except that we use the reverse forms of the rules as in Definitions 3.1, 3.2 and 3.3.*

Once L is generated by the procedure in Definition 3.4, we associate with L, in the standard way [3], the forward and reverse ltrs $\rightarrow$ and $\rightsquigarrow$ over Proc with labels drawn from ActK, together with the set of predicates Pred that interpret std and $\mathsf{fsh}[m]$ (for $m \in \mathcal{K}$) over Proc.

We illustrate the application of the conversion procedure on two operators that use the three allowed types of rules. Firstly, we consider the internal choice "$\sqcap$" of CSP, which may be defined by two choice axioms ($\tau \in \mathsf{Act}$):

$$\frac{}{X \sqcap Y \xrightarrow{\tau}_{\mathrm{S}} X} \qquad \frac{}{X \sqcap Y \xrightarrow{\tau}_{\mathrm{S}} Y}$$

Arguments X and Y both belong to E. Definition 3.3 requires two families of auxiliary operators "$\sqcap_1[m]$" and "$\sqcap_2[m]$" for all $m \in \mathcal{K}$. To save space, we only give the converted rules and the reverse rules for the first argument X:

$$\frac{\mathsf{std}(X) \quad \mathsf{std}(Y)}{X \sqcap Y \xrightarrow{\tau[m]} X \sqcap_1 [m]Y} \qquad \frac{X \xrightarrow{a[n]} X' \quad \mathsf{std}(Y)}{X \sqcap_1 [m]Y \xrightarrow{a[n]} X' \sqcap_1 [m]Y}\ m \neq n$$

$$\frac{\mathsf{std}(X) \quad \mathsf{std}(Y)}{X \sqcap_1 [m]Y \overset{\tau[m]}{\rightsquigarrow} X \sqcap Y} \qquad \frac{X \overset{a[n]}{\rightsquigarrow} X' \quad \mathsf{std}(Y)}{X \sqcap_1 [m]Y \overset{a[n]}{\rightsquigarrow} X' \sqcap_1 [m]Y}\ m \neq n$$

Next, we convert Milner's interrupt operator "$\hat{}$" [16] defined by the first two rule schemas below (all $a, b \in \mathsf{Act}$). We have $S = I = \{X\}$, $D = \{Y\}$, $E = D$ and Y is permissive. Definitions 3.1 and 3.2 give us the last two forward rule schemas below, and the reverse rules are simply symmetric versions of the forward rules.

$$\frac{X \xrightarrow{a}_{\mathrm{S}} X'}{X \hat{} Y \xrightarrow{a}_{\mathrm{S}} X' \hat{} Y} \quad \frac{Y \xrightarrow{b}_{\mathrm{S}} Y'}{X \hat{} Y \xrightarrow{b}_{\mathrm{S}} Y'} \quad \frac{X \xrightarrow{a[m]} X' \quad \mathsf{std}(Y)}{X \hat{} Y \xrightarrow{a[m]} X' \hat{} Y} \quad \frac{Y \xrightarrow{b[n]} Y' \quad \mathsf{fsh}[n](X)}{X \hat{} Y \xrightarrow{b[n]} X \hat{} Y'}$$

4 CCS with Communication Keys

In this section we convert CCS to a reversible process calculus, which we call CCSK (CCS with communication Keys), following Definition 3.4. Let $\tau \in \mathsf{Act}$. We assume the following standard signature of finite CCS:

$$\Sigma_{\mathrm{S}} = \{\mathbf{0}\} \cup \{a. \mid a \in \mathsf{Act}\} \cup \{\setminus A, [f] \mid A \subseteq \mathsf{Act} \setminus \{\tau\}, f : \mathsf{Act} \rightarrow \mathsf{Act}\} \cup \{+, |\}$$

The single argument of prefixing is neither dynamic nor static, and prefixing has a choice axiom rule (type (III)). By Definition 3.3 CCSK contains a family of auxiliary operators $a[m].$ (past action prefixing) for all $a \in \mathsf{Act}$ and $m \in \mathcal{K}$. Both arguments of $+$ are dynamic and permissive, and obviously non-static. Parallel composition, restriction and relabelling are operators with static rules. The well-known SOS rules for CCS, which can be found in [16], are converted into the rules in Figure 1. The rules for the reverse ltr for CCSK are got by simply changing $\rightarrow$ into $\rightsquigarrow$ throughout. As is usual, we omit trailing $\mathbf{0}$s.

$$\frac{\mathsf{std}(X)}{a.X \xrightarrow{a[m]} a[m].X} \qquad \frac{X \xrightarrow{b[n]} X'}{a[m].X \xrightarrow{b[n]} a[m].X'}\ m \neq n$$

$$\frac{X \xrightarrow{a[m]} X' \quad \mathsf{std}(Y)}{X + Y \xrightarrow{a[m]} X' + Y} \qquad \frac{Y \xrightarrow{a[m]} Y' \quad \mathsf{std}(X)}{X + Y \xrightarrow{a[m]} X + Y'}$$

$$\frac{X \xrightarrow{a[m]} X' \quad \mathsf{fsh}[m](Y)}{X \mid Y \xrightarrow{a[m]} X' \mid Y} \qquad \frac{Y \xrightarrow{a[m]} Y' \quad \mathsf{fsh}[m](X)}{X \mid Y \xrightarrow{a[m]} X \mid Y'} \qquad \frac{X \xrightarrow{a[m]} X' \quad Y \xrightarrow{\overline{a}[m]} Y'}{X \mid Y \xrightarrow{\tau[m]} X' \mid Y'}$$

$$\frac{X \xrightarrow{a[m]} X'}{X \backslash A \xrightarrow{a[m]} X' \backslash A}\ a \notin A \qquad \frac{X \xrightarrow{a[m]} X'}{X[f] \xrightarrow{f(a)[m]} X'[f]}$$

Fig. 1. Forward SOS rules for CCSK

As an extension, we could add recursion rec$X.P$ to CCSK by introducing the structural congruence $\equiv$ generated by the law rec$X.P \equiv P\{\text{rec}X.P/X\}$. We would then add a *Structural Congruence Rule* schema as in [17] to the rules in Figure 1. The schema links structural congruence with deriving transitions of terms: $X \xrightarrow{a[m]} X'$ can be derived if $X \equiv Y$, $Y \xrightarrow{a[m]} Y'$ and $Y' \equiv X'$ for all labels $a[m]$. To incorporate this extension into our format from Sections 2 and 3, we would need to work with formats with structural congruence (cf. [17]).

Example 4.1. In CCSK we keep track of the identities of actions that communicate so that when we reverse we undo the correct past actions. Consider $P = (a.b \mid a.c \mid \overline{a}.d \mid \overline{a}.e)\backslash a$. Here the restriction of a prevents a and $\overline{a}$ being performed except as part of a communication. Suppose that $a.b$ communicates with $\overline{a}.d$ and then $a.c$ with $\overline{a}.e$. In CCSK we write this as follows:

$$P \xrightarrow{\tau[m]} (a[m].b \mid a.c \mid \overline{a}[m].d \mid \overline{a}.e)\backslash a \xrightarrow{\tau[n]} (a[m].b \mid a[n].c \mid \overline{a}[m].d \mid \overline{a}[n].e)\backslash a$$

Note that the process $a[m].b \mid a.c \mid \overline{a}[m].d \mid \overline{a}.e$ cannot regress by reversing $a[m]$ alone because key m is not fresh in $a.c \mid \overline{a}[m].d \mid \overline{a}.e$. The fact that m appears in $a.c \mid \overline{a}[m].d \mid \overline{a}.e$ which is in parallel with $a[m].b$ proves that the processes communicated a and $\overline{a}$.

Our notation does not allow us to backtrack by undoing a different pair of actions, but clearly we can change the order of reversing actions $\tau[m]$ and $\tau[n]$:

$$(a[m].b \mid a[n].c \mid \overline{a}[m].d \mid \overline{a}[n].e)\backslash a \overset{\tau[m]}{\rightsquigarrow} (a.b \mid a[n].c \mid \overline{a}.d \mid \overline{a}[n].e)\backslash a \overset{\tau[n]}{\rightsquigarrow} P$$

4.1 Related Calculi

The present work is mainly to be compared with Danos and Krivine's RCCS [9], but also in some sense to an earlier approach by Boudol and Castellani [6].

To aid comparison we give a simple example: the processes $(a \mid \overline{a}.b)\backslash a$ and $\tau.b$. We might reasonably expect them to be equivalent, and indeed they are

FR-bisimilar as stated in Section 6. We have $(a \mid \overline{a}.b)\backslash a \overset{\tau[m]}{\rightarrow} (a[m] \mid \overline{a}[m].b)\backslash a$ and $\tau.b \overset{\tau[m]}{\rightarrow} \tau[m].b$. In RCCS since $\langle\,\rangle \triangleright \nu a\,(a \mid \overline{a}.b) \equiv \nu a\,(\langle 1\rangle \triangleright a \mid \langle 2\rangle \triangleright \overline{a}.b)$ we write these transitions as

$$\nu a\,(\langle 1\rangle \triangleright a \mid \langle 2\rangle \triangleright \overline{a}.b) \overset{\langle 1\rangle,\langle 2\rangle:\tau}{\rightarrow} \nu a\,(\langle\langle 2\rangle, a, \mathbf{0}\rangle \cdot \langle 1\rangle \triangleright \mathbf{0} \mid \langle\langle 1\rangle, \overline{a}, \mathbf{0}\rangle \cdot \langle 2\rangle \triangleright b)$$

and $\langle\,\rangle \triangleright \tau.b \overset{\langle\,\rangle:\tau}{\rightarrow} \langle *, \tau, \mathbf{0}\rangle \triangleright b$, respectively. In RCCS transition labels contain extra information concerning which threads contribute. As a result it is harder to show that the processes are equivalent. Presumably one would have to abstract away from the thread information.

We might therefore say that, on the spectrum from intensionality to extensionality, the present work is more extensional than RCCS, though we see from the examples in Section 6 that CCSK definitely has a "true concurrency" flavour in terms of which processes it equates.

In [6] Boudol and Castellani developed *event systems*. Similarly to our approach, they keep track of the whole past of a transition by recording past actions and choices that have been made. These are recorded in the syntax of terms and, unlike in our approach, in the transition labels themselves. For example, where we write $(a \mid \overline{a}.b)\backslash a \overset{\tau[m]}{\rightarrow} (a[m] \mid \overline{a}[m].b)\backslash a$, in event systems this is $(a \mid \overline{a}.b)\backslash a \xrightarrow{\backslash a(a,\overline{a})} (\underline{a} \mid \underline{\overline{a}}.b)\backslash a$ and one needs to use additional rules to work out that the action label of the transition is a τ.

5 Properties of the Transition Relations

In this section we establish various properties of the forward and reverse transition relations defined earlier. In particular we show that the forward-reachable processes are closed under reverse transitions (Proposition 5.6); also that the new forward transition relation is in a sense conservative over the standard transition relation (Theorem 5.8).

We start by noting that the reverse transition relation inverts the forward transition relation:

Proposition 5.1. *Let $P, P' \in \mathsf{Proc}$ and $\mu \in \mathsf{ActK}$. Then $P \overset{\mu}{\rightarrow} P'$ iff $P' \overset{\mu}{\rightsquigarrow} P$.*

Each process has a set of keys. The set $\mathsf{keys}(P)$ of keys occurring in a process $P \in \mathsf{Proc}$ is defined as follows: $\mathsf{keys}(\mathbf{0}) \overset{\text{df}}{=} \emptyset$, $\mathsf{keys}(f(\overrightarrow{P})) \overset{\text{df}}{=} \bigcup_{i \in N} \mathsf{keys}(P_i)$ and $\mathsf{keys}(f_r[m](\overrightarrow{P})) \overset{\text{df}}{=} \{m\} \cup \bigcup_{i \in N} \mathsf{keys}(P_i)$. Clearly $P \in \mathsf{Std}$ iff $\mathsf{keys}(P) = \emptyset$. Also $\mathsf{fsh}[m](P)$ iff $m \notin \mathsf{keys}(P)$.

Any forward transition uses a fresh key:

Lemma 5.2. *Let $P, P' \in \mathsf{Proc}$. If $P \overset{a[m]}{\rightarrow} P'$ then $m \notin \mathsf{keys}(P)$ and $\mathsf{keys}(P') = \mathsf{keys}(P) \cup \{m\}$.*

Let $P \rightarrow Q$ iff $P \overset{\mu}{\rightarrow} Q$ for some μ. Let $\rightarrow^*$ denote the reflexive and transitive closure of $\rightarrow$.

Definition 5.3. *A process $P \in \mathsf{Proc}$ is* reachable *if it can be reached by a finite sequence of forward transitions from a process in Std, i.e. there is $Q \in \mathsf{Std}$ such that $Q \to^* P$. Let Rch denote the set of reachable processes.*

It is easy to check that if $P \in \mathsf{Rch}$ and P' is a subterm of P then also $P' \in \mathsf{Rch}$. It follows from Lemma 5.2 that if $P \in \mathsf{Rch}$ then every $\to$-computation from a process $Q \in \mathsf{Std}$ to P must have length $|\mathsf{keys}(P)|$.

Of course, not every process is reachable. In CCSK, $a.b[m]$ is not reachable. A more interesting example is $a[m].b[n] \mid \overline{b}[n].\overline{a}[m]$. Here the names and keys match up, but there is a causal inconsistency.

A "diamond" confluence property holds for reverse transitions:

Proposition 5.4 (Reverse Diamond Property). *Let $P, Q, R \in \mathsf{Proc}$.*

1. *If $P \overset{a[m]}{\rightsquigarrow} Q$ and $P \overset{b[m]}{\rightsquigarrow} R$ then $a = b$ and $Q = R$.*
2. *If $P \overset{a[m]}{\rightsquigarrow} Q$ and $P \overset{b[n]}{\rightsquigarrow} R$ with $m \neq n$, then there is S such that $Q \overset{b[n]}{\rightsquigarrow} S$ and $R \overset{a[m]}{\rightsquigarrow} S$.*

Proposition 5.4 implies that the reverse transition relation is finitely branching, since the number of reverse transitions of $P \in \mathsf{Proc}$ is bounded by $|\mathsf{keys}(P)|$.

The analogue of Proposition 5.4 does not hold for forward transitions, since two forward transitions $P \xrightarrow{a[m]} Q$ and $P \xrightarrow{b[n]} R$ may conflict. However we can complete the diamond if the forward transitions are compatible, in the sense that Q and R can reach a common process S by forward moves:

Proposition 5.5 (Forward Diamond Property). *Let $P, Q, R, T \in \mathsf{Proc}$.*

1. *If $P \xrightarrow{a[m]} Q \xrightarrow{s} T$ and $P \xrightarrow{b[m]} R \xrightarrow{t} T$ then $a = b$ and $Q = R$.*
2. *If $P \xrightarrow{a[m]} Q \xrightarrow{s} T$ and $P \xrightarrow{b[n]} R \xrightarrow{t} T$ with $m \neq n$, then there is S such that $Q \xrightarrow{b[n]} S$, $R \xrightarrow{a[m]} S$ and $S \xrightarrow{s \backslash b[n]} T$, $S \xrightarrow{t \backslash a[m]} T$.*

(Here for any $s \in \mathsf{ActK}^*$ and $\mu \in \mathsf{ActK}$, $s \backslash \mu$ is s with all instances of μ removed.)

The reachable terms are closed under reverse transitions, meaning that a process can never reverse into an "inconsistent" past:

Proposition 5.6. *If $P \in \mathsf{Rch}$, $\mu \in \mathsf{ActK}$ and $P \overset{\mu}{\rightsquigarrow} P'$ then $P' \in \mathsf{Rch}$.*

We now turn to showing that the new forward transition relation $\to$ is essentially conservative over the standard transition relation $\to_{\mathrm{S}}$. We have to take into account the fact that we have introduced auxiliary operators and keys. A nonstandard process can be converted to a corresponding standard process by "pruning" the auxiliary operators (cf. the forgetful map of [9]):

Definition 5.7. *The pruning map* $\pi : \mathsf{Proc} \to \mathsf{Std}$ *is defined as follows:*

$$\pi(\mathbf{0}) \overset{\mathrm{df}}{=} \mathbf{0}$$
$$\pi(f(\overrightarrow{P})) \overset{\mathrm{df}}{=} \begin{cases} \pi(P_d) & \textit{if } d \in D_f \wedge \neg\mathsf{std}(P_d) \wedge \forall e \in E_f \setminus \{d\}.\, \mathsf{std}(P_e) \\ f(\overrightarrow{\pi(P)}) & \textit{if } \forall e \in E_f.\, \mathsf{std}(P_e) \\ \mathbf{0} & \textit{otherwise} \end{cases}$$
$$\pi(f_r[m](\overrightarrow{P})) \overset{\mathrm{df}}{=} \begin{cases} \pi(P_{\mathsf{ta}(r)}) & \textit{if } \forall e \in E_f \setminus \{\mathsf{ta}(r)\}.\, \mathsf{std}(P_e) \\ \mathbf{0} & \textit{otherwise} \end{cases}$$

for any choice axiom r for f, and where $\overrightarrow{\pi(P)}$ is the vector $\pi(P_1), \ldots, \pi(P_n)$.

Clearly, if $P \in \mathsf{Std}$ then $\pi(P) = P$. It can easily be shown that the third case for $\pi(f(\overrightarrow{P}))$ and the second case for $\pi(f_r(\overrightarrow{P}))$ will not arise with reachable terms.

Theorem 5.8 (Conservation). *Suppose $P \in \mathsf{Proc}$.*

1. *If $P \xrightarrow{a[m]} P'$ then $\pi(P) \xrightarrow{a}_{\mathsf{S}} \pi(P')$.*
2. *If $\pi(P) \xrightarrow{a}_{\mathsf{S}} P'$ then for any $m \in \mathcal{K} \setminus \mathsf{keys}(P)$ there is P'' such that $P \xrightarrow{a[m]} P''$ and $\pi(P'') = P'$.*

6 Forward-Reverse Bisimulation

We can show that the reversible transition relation $\to$ induces essentially the same bisimulation equivalence on processes as the standard transition relation $\to_{\mathsf{S}}$. We first recall standard strong bisimulation on the standard terms:

Definition 6.1. *A symmetric relation $\mathcal{S}$ on* Std *is an* S-bisimulation *if whenever $\mathcal{S}(P, Q)$ then if $P \xrightarrow{a}_{\mathsf{S}} P'$ then there is Q' such that $Q \xrightarrow{a}_{\mathsf{S}} Q'$ and $\mathcal{S}(P', Q')$. We define $P \sim_{\mathsf{S}} Q$ iff there is an S-bisimulation $\mathcal{S}$ such that $\mathcal{S}(P, Q)$.*

The corresponding notion for forward transitions on Proc and predicates Pred is

Definition 6.2. *A symmetric relation $\mathcal{S}$ on* Proc *is an* F-bisimulation *if $\mathcal{S}(P, Q)$ implies*

- $\mathsf{p}(P) \Leftrightarrow \mathsf{p}(Q)$ *for all* $\mathsf{p} \in \mathsf{Pred}$*;*
- *if $P \xrightarrow{\mu} P'$ then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $\mathcal{S}(P', Q')$.*

We define $P \sim_{\mathsf{F}} Q$ iff there is an F-bisimulation $\mathcal{S}$ such that $\mathcal{S}(P, Q)$.

Note that the first item in Definition 6.2 could be written as $\mathsf{keys}(P) = \mathsf{keys}(Q)$, since $\mathsf{fsh}[m](P) \Leftrightarrow m \notin \mathsf{keys}(P)$ and $\mathsf{std}(P) \Leftrightarrow \mathsf{keys}(P) = \emptyset$.

F-bisimulation is conservative over S-bisimulation by the following result:

Proposition 6.3. *Let $P, Q \in \mathsf{Proc}$. Then $P \sim_{\mathsf{F}} Q$ iff $\pi(P) \sim_{\mathsf{S}} \pi(Q)$ and $\mathsf{p}(P) \Leftrightarrow \mathsf{p}(Q)$ for all $\mathsf{p} \in \mathsf{Pred}$.*

Proposition 6.4. *The relation $\sim_{\mathsf{F}}$ is a congruence with respect to all the operators of* Proc*.*

We now define bisimulation for both forward and reverse transitions:

Definition 6.5. *A symmetric relation $\mathcal{S}$ on* Proc *is a* forward-reverse (FR) bisimulation *if whenever $\mathcal{S}(P,Q)$ then*

- $\mathsf{p}(P) \Leftrightarrow \mathsf{p}(Q)$ *for all* $\mathsf{p} \in \mathsf{Pred}$*;*
- *if $P \xrightarrow{\mu} P'$ then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $\mathcal{S}(P',Q')$;*
- *if $P \overset{\mu}{\rightsquigarrow} P'$ then there is Q' such that $Q \overset{\mu}{\rightsquigarrow} Q'$ and $\mathcal{S}(P',Q')$.*

We define $P \sim_{\mathsf{FR}} Q$ iff there is an FR bisimulation $\mathcal{S}$ such that $\mathcal{S}(P,Q)$.

Proposition 6.6. *Let $P, Q \in$* Proc*. If $P \sim_{\mathsf{FR}} Q$ then $P \sim_{\mathsf{F}} Q$.*

The converse does not hold. For instance we have $a \mid a \sim_{\mathsf{F}} a.a$, but $a \mid a \not\sim_{\mathsf{FR}} a.a$. This is because $a \mid a \xrightarrow{a[m]}\xrightarrow{a[n]} a[m] \mid a[n] \overset{a[m]}{\rightsquigarrow} a \mid a[n]$ and $m \neq n$. This sequence of transitions cannot be matched by $a.a$: we have $a.a \xrightarrow{a[m]}\xrightarrow{a[n]} a[m].a[n] \overset{a[m]}{\not\rightsquigarrow}$. Similarly $a \mid b \sim_{\mathsf{F}} a.b + b.a$, but $a \mid b \not\sim_{\mathsf{FR}} a.b + b.a$.

On the positive side, we can show that for any $P \in \mathsf{Std}$, $P + P \sim_{\mathsf{FR}} P$. We can also show that for any $P \in \mathsf{Std}$, $(a \mid \overline{a}.P)\backslash a \sim_{\mathsf{FR}} \tau.(P\backslash a)$.

Theorem 6.7. *The relation $\sim_{\mathsf{FR}}$ is a congruence with respect to all the operators of* Proc*.*

Several notions of bisimulation taking into account backward as well as forward moves have been discussed in the literature. The *back and forth bisimulation* of [11] is constrained to only go back along the path that brought a process to its current state. Back and forth bisimulation where any reverse path can be followed is discussed in [5] both for transition systems and event structures. Essentially the same notion, but called *backward-forward bisimulation*, is defined in [13] for occurrence transition systems. The non-interleaving semantics community has proposed several bisimulation-like equivalences [12] and we intend to investigate how FR bisimulation compares with them.

7 Extensions

Our conversion procedure can be extended in several directions so that it applies to a wider class of operators. Naturally, this would result in extending the forms of SOS rules in Definitions 3.1–3.3. However, the extensions we now briefly describe mostly do not go beyond the simple *path* format as in Definition 2.1.

ACP action constants can be defined analogously to prefixing of CCS. We have the constant ε (successful termination) and constants a for each $a \in \mathsf{Act}$. The defining rules $a \xrightarrow{a}_{\mathrm{S}} \varepsilon$ are converted to $a \xrightarrow{a[m]} a[m]$, where $a[m]$ are auxiliary constants for all $m \in \mathcal{K}$. There are no forward SOS rules for the auxiliary constants and no transition rules for ε.

The next extension is to allow predicates in SOS rules. An example is the successful termination predicate trm in the rules for ACP's sequential composition "$\cdot$" below [4]. Care needs to be taken when adding predicates to premises in order to avoid *lookahead* in the reverse rules.

$$\frac{X \xrightarrow{a}_{\mathrm{S}} X'}{X \cdot Y \xrightarrow{a}_{\mathrm{S}} X' \cdot Y} \qquad \frac{Y \xrightarrow{b}_{\mathrm{S}} Y' \quad \mathsf{trm}(X)}{X \cdot Y \xrightarrow{b}_{\mathrm{S}} Y'}$$

With some simplifications, the converted and reverse rules are

$$\frac{X \xrightarrow{a} X' \quad \mathsf{std}(Y)}{X \cdot Y \xrightarrow{a} X' \cdot Y} \quad \frac{Y \xrightarrow{b} Y' \quad \mathsf{trm}(X)}{X \cdot Y \xrightarrow{b} X \cdot Y'} \quad \frac{X \overset{a}{\rightsquigarrow} X' \quad \mathsf{std}(Y)}{X \cdot Y \overset{a}{\rightsquigarrow} X' \cdot Y} \quad \frac{Y \overset{b}{\rightsquigarrow} Y' \quad \mathsf{trm}(X)}{X \cdot Y \overset{b}{\rightsquigarrow} X \cdot Y'}$$

(Here we extend trm to cover nonstandard processes.)

Finally, to allow the external choice operator of CSP we need to relax the condition that static arguments cannot be dynamic. The defining rules for "$\Box$" are given below, where the last two rules are rule schemas for all $a \in \mathsf{Act} \setminus \{\tau\}$.

$$\frac{X \xrightarrow{\tau}_{\mathrm{S}} X'}{X \Box Y \xrightarrow{\tau}_{\mathrm{S}} X' \Box Y} \quad \frac{Y \xrightarrow{\tau}_{\mathrm{S}} Y'}{X \Box Y \xrightarrow{\tau}_{\mathrm{S}} X \Box Y'} \qquad \frac{X \xrightarrow{a}_{\mathrm{S}} X'}{X \Box Y \xrightarrow{a}_{\mathrm{S}} X'} \quad \frac{Y \xrightarrow{a}_{\mathrm{S}} Y'}{X \Box Y \xrightarrow{a}_{\mathrm{S}} Y'}$$

By introducing an auxiliary predicate $\mathsf{before}(P)$, which holds if $P \in \mathsf{Std}$ or P is a derivative from a standard term via a sequence of silent actions, we obtain the following converted rules:

$$\frac{\mathsf{before}(Y) \qquad X \xrightarrow{\mu} X'}{X \Box Y \xrightarrow{\mu} X' \Box Y} \qquad \frac{\mathsf{before}(X) \qquad Y \xrightarrow{\mu} Y'}{X \Box Y \xrightarrow{\mu} X \Box Y'}$$

8 Conclusions

There has been much recent interest in reversible computing, including the pioneering work of Danos and Krivine on reversible CCS. We have introduced a method for converting standard irreversible operators of algebraic process calculi such as CCS into reversible operators. As far as we are aware, this is the first time that such a method has been proposed in the context of general process calculi. Our method works on operators with rules of a simple form. We arrive at new rules which preserve the structure of the terms. An important feature of our method is the introduction of keys to bind synchronised actions together. We have also obtained an appropriate notion of bisimulation on terms. Our work demonstrates that it is possible to make many standard operators reversible in a manner which is both algebraic and tractable.

Acknowledgements

We wish to thank Philippa Gardner, Daniele Varacca, Nobuko Yoshida, Shoji Yuen and the referees for helpful discussions and comments. The second author would like to thank the University of Leicester for granting study leave, and acknowledge gratefully support from Nagoya University during a research visit.

References

[1] S. Abramsky. A structural approach to reversible computation. *Theoretical Computer Science*, 347(3):441–464, 2005.

[2] T. Altenkirch and J. Grattage. A functional quantum programming language. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, LICS 2005*, pages 249–258. IEEE Computer Society Press, 2005.

[3] J.C.M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In *Proceedings of 4th International Conference on Concurrency Theory, CONCUR '93*, volume 715 of *LNCS*, pages 477–492. Springer-Verlag, 1993.

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

[5] M.A. Bednarczyk. Hereditary history preserving bisimulations or what is the power of the future perfect in program logics. Technical report, Institute of Computer Science, Polish Academy of Sciences, Gdańsk, 1991.

[6] G. Boudol and I. Castellani. Flow models of distributed computations: three equivalent semantics for CCS. *Information and Computation*, 114:247–314, 1994.

[7] H. Buhrman, J. Tromp, and P. Vitányi. Time and space bounds for reversible simulation. In *Proceedings of 28th International Colloquium on Automata, Languages and Programming, ICALP 2001*, volume 2076 of *LNCS*, pages 1017–1027. Springer-Verlag, 2001.

[8] V. Danos and J. Krivine. Formal molecular biology done in CCS-R. In *Proceedings of BioConcur, Marseille*, 2003.

[9] V. Danos and J. Krivine. Reversible communicating systems. In *Proceedings of the 15th International Conference on Concurrency Theory, CONCUR 2004*, volume 3170 of *LNCS*, pages 292–307. Springer-Verlag, 2004.

[10] V. Danos and J. Krivine. Transactions in RCCS. In *Proceedings of the 16th International Conference on Concurrency Theory, CONCUR 2005*, volume 3653 of *LNCS*, pages 398–412. Springer-Verlag, 2005.

[11] R. De Nicola, U. Montanari, and F. Vaandrager. Back and forth bisimulations. In *Proceedings of CONCUR '90, Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*, pages 152–165. Springer-Verlag, 1990.

[12] R.J. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37:229–327, 2001.

[13] U. Goltz, R. Kuiper, and W. Penczek. Propositional temporal logics and equivalences. In *Proceedings of 3rd International Conference on Concurrency Theory, CONCUR '92*, volume 630 of *LNCS*, pages 222–235. Springer-Verlag, 1992.

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

[15] R. Landauer. Irreversibility and heat generated in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

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

[17] M.R. Mousavi and M.A. Reniers. Congruence for structural congruences. In *Proceedings of the 8th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2005*, volume 3441 of *LNCS*, pages 47–62. Springer-Verlag, 2005.

[18] I.C.C. Phillips and I. Ulidowski. Reversing algebraic process calculi. Technical Report CS-06-01, Department of Computer Science, Leicester University, 2006.

[19] G.D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[20] Virtutech. *Simics Hindsight*. http://www.virtutech.com, 2005.