INFORMATION AND COMPUTATION **125**, 118–143 (1996)
ARTICLE NO. 0026

Comparing Syntactic and Semantic Action Refinement*

URSULA GOLTZ

Institut für Informatik, University of Hildesheim, Postfach 101363, D-31113 Hildesheim, Germany
E-mail: goltz@informatik.uni-hildesheim.de

ROBERTO GORRIERI

Dipartimento di Scienze dell'Informazione, University of Bologna, Piazza di Porta San Donato 5, I-40127 Bologna, Italy
E-mail: gorrieri@cs.unibo.it

AND

AREND RENSINK

Institut für Informatik, University of Hildesheim, Postfach 101363, D-31113 Hildesheim, Germany
E-mail: rensink@informatik.uni-hildesheim.de

The semantic definition of action refinement on labelled configuration structures is compared with the notion of *syntactic substitution*, which can be used as another notion of action refinement in a process algebraic setting. The comparison is done by studying a process algebra equipped with sequential composition, parallel composition with an explicit synchronisation set, and an operator for action refinement. On the one hand, the language (including the refinement operator) is given a configuration structure semantics. On the other hand, a reduction procedure transforms a process term P into a *flat* term (i.e., with the refinement operator not occurring in it) $red(P)$ by means of syntactic substitution, defined in a structural inductive way. The main aim of the paper is to investigate general conditions under which the terms P and $red(P)$ have the same semantics. The results we present are essentially dependent on the question whether the refined action can be synchronised or not. In the latter case, P and $red(P)$ give rise to isomorphic configuration structures under mild assumptions. The former case is considerably more difficult, since then refinement cannot be expected to distribute over parallel composition. We give necessary and sufficient *semantic* conditions under which distribution still holds up to semantic equivalence. Subsequently, we also give sufficient (but not necessary) *syntactic* conditions for reducible terms. Finally, we generalise these results to a language with recursion.

1. INTRODUCTION

The refinement of actions in concurrency theories has been proposed as a means for relating descriptions of concurrent systems at different levels of abstraction and for helping in their top-down design. The basic principle is to implement a given abstract action in terms of larger and more complex concrete behaviour. In this paper it is expressed by terms of the form $P[a \rightsquigarrow Q]$ where, intuitively, every time action a should be executed in P, the term Q is executed instead. This conceptually attractive principle has received widespread interest; however, to formalise it effectively is proving to be a complex issue, and consequently research on this subject has taken various different approaches.

Two main lines of research can be recognised. On the one hand there is *atomic refinement* [5, 14, 20, 21], where one takes the point of view that actions are atomic and their refinements should in some sense preserve this atomicity. On the other hand, there is a more liberal notion of refinement according to which atomicity is always relative to the current level of abstraction, and may in a sense be destroyed by refinement. This paper is concerned with the second approach.

Within this approach there are again essentially two notions of action refinement, which we call *semantic* and *syntactic*. In the *semantic* interpretation, a refinement operation is defined in the semantic domain that is used to interpret terms. Then the semantics of $P[a \rightsquigarrow Q]$ can be defined using this operator. For example, when using configuration structures as semantic domains, a configuration structure $\mathscr{D} = [\![Q]\!]$, representing the semantics of Q, would be substituted for every a-labelled event d in the configuration structure $\mathscr{C} = [\![P]\!]$. The refinement operation preserves the semantic embedding of events: e.g., if d is in conflict with an event e, then all the events of $\mathscr{D}$ will be in conflict with e. Investigations of such refinement operators can be found,

* This work has been partially supported by the Vigoni exchange program and the HCM network EXPRESS (Expressiveness of Languages for Concurrency). An extended abstract of this paper appeared *in* "Theoretical Aspects of Computer Software (TACS'94)," Lecture Notes in Computer Science, Vol. 789, pp. 385–404, Springer-Verlag, Berlin/New York, 1994.

0890-5401/96 $18.00

e.g., in [3, 9, 10, 13, 16, 22, 29, 17, 18, 30, 31] over the semantic domains of Prime, Free and Flow Event Structures, Configurations Structures, Families of Posets, Causal Trees, ST-Trees, and Petri Nets.

The *syntactic* approach takes a different starting point, namely a process algebra equipped with an ACP-like operation of sequential composition. Action refinement is understood as an operation of syntactic substitution of a process term for an action. Hence $P[a \rightsquigarrow Q]$ is interpreted as the term obtained from P if every action a is replaced by Q; i.e., Q is to be substituted for a in the *term* P rather than in the semantics of P. Therefore, the semantics of $P[a \rightsquigarrow Q]$ is, *by definition*, the semantics of the term $P\{Q/a\}$. This line of research has been pursued in [1, 25, 2].

These two approaches are inherently different; simple examples showing this are given below. Conceptually, this is a negative result which in general prevents the definition of an algebraic theory for action refinement. Indeed, syntactic refinement corresponds to a *homomorphism* between algebras whose operations are those of the language (since such a homomorphism is essentially generated by a mapping from actions to subterms, which is required to distribute over all the operators); on the other hand, semantic refinement defines an operation on the semantic model which is *compositional*. As the two approaches do not coincide, we cannot expect to be able, in general, to define compositional homomorphisms. In this paper, we compare the two approaches with the aim to identify under which restrictions they yield the same result. That is, we investigate conditions under which the following diagram commutes:

$$\begin{array}{lll} P[a \rightsquigarrow Q] & \xrightarrow{\text{syntactic ref.}} & P\{Q/a\} \quad \textit{syntax} \\ \downarrow & & \downarrow \\ [\![P]\!][a \rightsquigarrow [\![Q]\!]] & \xrightarrow{\text{semantic ref.}} & [\![P\{Q/a\}]\!] \quad \textit{semantics} \end{array} \tag{1}$$

The result not only gives a clearer understanding of the theory of action refinement, but also it is interesting for applications of action refinement to know when semantic refinement can be implemented by the simpler syntactic substitution.

In this paper, we consider a process algebra with sequential composition and synchronisation; the language is provided with a configuration structure semantics (following the definition of the operators reported in [33, 18]). It turns out that the difference between syntactic and semantics refinement can be traced to the problem whether refinement distributes over parallel composition with synchronisation. In this paper we use a TCSP-like synchronisation operator,[1] which takes the form $P_1 \|_A P_2$, where A denotes the set of *communication actions*, i.e., those actions on which both P_1 and P_2 are forced to synchronise. Distribution of refinement over parallel composition then means that the following semantic equation holds:

$$(P_1 \|_A P_2)[a \rightsquigarrow Q] \cong P_1[a \rightsquigarrow Q] \|_{A'} P_2[a \rightsquigarrow Q]. \tag{2}$$

Here A' may be some modification of the synchronisation set A (see below). This equation however does not hold in general. The terms $(a \|_{\{b\}} b; c)[a \rightsquigarrow b]$ and $a[a \rightsquigarrow b] \|_{\{b\}} (b; c)[a \rightsquigarrow b]$ for instance are not equivalent: intuitively, in the first term, c on the right hand side is prevented from occurring since the preceding b cannot synchronise with anything on the left hand side; hence this behaviour may only execute a, which is however refined to b. In the second term, b occurs as a result of synchronisation, after which c is executed.

In this example one could argue that the mismatch is due to the fact that on the right hand side, "new" actions (the b resulting from the refinement of a) are permitted to synchronise with "old" ones (the b already occurring before refinement). This is in contrast with the intuition that, in $P[a \rightsquigarrow Q]$, the actions of P and Q should be considered at different levels of abstraction (see also [5, 15] on this point). We will adopt this view and restrict our attention to those terms satisfying the following alphabet-disjointness condition; $P[a \rightsquigarrow Q]$ is *well-formed* if $(\{a\} \cup L(P)) \cap L(Q) = \varnothing$, where $L(P)$ denotes the alphabet of P. We first consider the case that synchronising actions are not refined, that is, $a \notin A$ for $(P_1 \|_A P_2)[a \rightsquigarrow Q]$. In this case we show that under well-formedness, (2) holds and we are therefore able to establish commutativity of (1).[2]

The situation becomes much more difficult if we consider refining synchronising actions, that is, $a \in A$ for $(P_1 \|_A P_2)[a \rightsquigarrow Q]$. Already intuitively, (2) can then no longer be expected to hold in general, since we would not distinguish a term $a \|_{\{a\}} a$ from a. And indeed, problems now occur in the distribution of refinement over synchronisation. For instance, we have that $(a \|_{\{a\}} a)[a \rightsquigarrow b; c + b; d]$ is not even completed trace equivalent to $a[a \rightsquigarrow b; c + b; d] \|_{\{b, c, d\}} a[a \rightsquigarrow b; c + b; d]$: the latter has a completed trace b which is not shared by the former. (Note that here we have also changed the synchronisation set while applying (2), so that rather than over the refined action, after distribution the terms synchronise over the alphabet of the substituted term.) The second result of this paper is the formulation of necessary and sufficient *semantic* conditions and sufficient *syntactic* conditions for (2) to hold. We subsequently extend the latter conditions for terms of the form $(P_1 \|_A P_2)[a \rightsquigarrow Q]$ to a characterisation of the sublanguage in which syntactic and semantics refinement coincide.

[1] This choice does not affect the central problem essentially. In Section 8 we briefly discuss the CCS setting of [1, 2].

[2] Non-well-formed terms may be dealt with at the price of adding an auxiliary operator of renaming, as illustrated in Section 8.

Next, we show how the results presented here can be generalised to a language with recursion. Finally, some concluding remarks (including a discussion of non-well-formed terms) and a comparison with related literature are given.

2. SYNTAX AND SEMANTICS OF THE LANGUAGE

We assume a global (infinite) set of actions Act. The following grammar defines the terms of the language (a finite process algebra with action refinement) that we will study in this paper:

$$P ::= a \mid P + P \mid P; P \mid P \parallel_A P \mid P[a \rightsquigarrow P].$$

Most of the operators are standard. We use a *family of synchronisation operators* $\{\parallel_A\}_{A \subseteq Act}$ corresponding to the TCSP approach: in $P \parallel_A Q$ processes P and Q are forced to synchronise on actions in A and forced to proceed asynchronously on actions not belonging to A. The *refinement operator* $P[a \rightsquigarrow Q]$ acts on single actions at a time. The behaviour of $P[a \rightsquigarrow Q]$ is derived from the behaviour of P by replacing every execution of the action a by the behaviour of Q. Σ will denote the set of all the terms generated by the syntax above; $\Sigma_{flat} \subseteq \Sigma$ denotes the set of terms that do not contain refinement operators. Brackets will be used as usual to show the structure of terms in Σ; to improve the readability, we will let sequential composition bind stronger than choice and synchronisation, and refinement stronger than any of the binary operators.

2.1. *Well-Formed Terms*

A useful notion in this investigation is the *alphabet* of a term P, denoted $L(P)$. Another, less standard notion is the *set of synchronising actions* of a term P, denoted $S(P)$, where basically an action a is called synchronising if $a \in A$ in a subterm $P_1 \parallel_A P_2$ of P, and moreover it is in the alphabet of either P_1 or P_2. These are defined inductively in Table 1. It follows that $S(P) \subseteq L(P)$ for all terms $P \in \Sigma$.

TABLE 1

Label Set and Synchronising Set

$$\begin{aligned}
L(a) &:= \{a\} \\
L(P+Q) &:= L(P) \cup L(Q) \\
L(P;Q) &:= L(P) \cup L(Q) \\
L(P \parallel_A Q) &:= L(P) \cup L(Q) \cup A \\
L(P[a \rightsquigarrow Q]) &:= \begin{cases} (L(P) \setminus \{a\}) \cup L(Q) & \text{if } a \in L(P) \\ L(P) & \text{otherwise} \end{cases} \\
\\
S(a) &:= \varnothing \\
S(P+Q) &:= S(P) \cup S(Q) \\
S(P;Q) &:= S(P) \cup S(Q) \\
S(P \parallel_A Q) &:= S(P) \cup S(Q) \cup ((L(P) \cup L(Q)) \cap A) \\
S(P[a \rightsquigarrow Q]) &:= \begin{cases} (S(P) \setminus \{a\}) \cup L(Q) & \text{if } a \in S(P) \\ S(P) \cup S(Q) & \text{if } a \in L(P) \setminus S(P) \\ S(P) & \text{otherwise.} \end{cases}
\end{aligned}$$

We now argue that it makes sense to restrict the refinement under consideration to a certain format. Consider a term of the form $P[a \rightsquigarrow Q]$. The intuition behind refinement tells us that Q represents an implementation of a and hence a is in some sense *more abstract* than the actions in $L(Q)$. It is only a small step from there to the assumption that *all* the actions of P are more abstract than those of Q; in other words, $L(Q)$ contains "new" actions that did not yet occur in the specification P. This makes it impossible for actions in P to synchronise with those in Q (after refinement) and hence rules out a kind of confusion of abstraction levels. In other words we assume

$$(\{a\} \cup L(P)) \cap L(Q) = \varnothing. \tag{3}$$

To put this assumption into effect we will restrict ourselves to a subset of the terms satisfying the *well-formedness predicate* $\vdash$ defined in Table 2, which effectively ensures (3). If this is felt to be an undue restriction, then—at the price of adding an (auxiliary) operator of *renaming* to the syntax—this assumption can be dropped and our results can be generalised to the entire Σ, as we will show in Section 8.

Note that technically, a less restrictive notion of well-formedness can be found if we require $S(P) \cap L(Q) = \varnothing$ rather than (3). Indeed, the results of this paper also hold for this more general case. However, the intuition behind well-formedness, i.e., absence of confusion between levels of abstraction, is expressed more appropriately by (3).

2.2. *Configuration Structure Semantics*

Throughout this paper, $\mathbf{E}$ will be a set of *events* (used to model the occurrences of actions) such that $* \notin \mathbf{E}$ and $(\mathbf{E} \cup \{*\}) \times (\mathbf{E} \cup \{*\}) \subseteq \mathbf{E}$. We interpret the terms of Σ in the model of (*stable*) *configuration structures* proposed by Winskel [33]. The interpretation is standard and can be found for instance in [18].

2.1. DEFINITION. A *configuration structure* is a tuple $\mathscr{C} = \langle C, \sqrt{}, \ell \rangle$ where

- $C \subseteq Fin(\mathbf{E})$ is a family of finite subsets of $\mathbf{E}$ (called *configurations*) such that
  - $\varnothing \in C$;
  - $\forall F, G.\ (\exists H \in C.(F \cup G) \subseteq H) \Rightarrow (F \cup G) \in C \wedge (F \cap G) \in C$;
  - $\forall F \in C.\ \forall d, e \in F.\ d \neq e \Rightarrow (\exists G \in C.d \in G \Leftrightarrow e \notin G)$.

TABLE 2

The Well-Formedness Predicate

$$\frac{}{\vdash a} \qquad \frac{\vdash P, Q \quad * \in \{;, +, \parallel_A\}}{\vdash P * Q} \qquad \frac{\vdash P, Q \quad (\{a\} \cup L(P)) \cap L(Q) = \varnothing}{\vdash P[a \rightsquigarrow Q]}$$

- $\surd \subseteq C$ is a set of *terminated configurations*, sometimes treated as a predicate in postfix notation, such that $F \subset G$ implies $\neg(F\surd)$ (i.e., terminated configurations must be maximal w.r.t. $\subseteq$);
- $\ell\colon (\bigcup C) \to Act$ is a *labelling function.*

The class of configuration structures is denoted **C**. Given a configuration structure $\mathscr{C}$, we will sometimes use $\surd_C$ and ℓ_C to denote the termination predicate and labelling function, respectively. $E_C := \bigcup C$ will denote the set of events of $\mathscr{C}$.

2.2. Definition. Two configuration structures, $\mathscr{C}, \mathscr{D}$ are *isomorphic*, denoted $\mathscr{C} \cong \mathscr{D}$, if there exists a bijection $f\colon E_C \to E_D$ which preserves event labelling, such that the pointwise extension of f to sets of events maps C to D and $\surd_C$ to $\surd_D$:

$$f(C) = D$$
$$f(\surd_C) = \surd_D$$
$$\ell_D \circ f = \ell_C.$$

Note that the $\subseteq$-structure is preserved automatically, because f is a bijection. Throughout this paper, configuration structures will be interpreted modulo isomorphism. A number of operations over **C** are defined as in [33, 18], corresponding to the operators of Σ. In the following two definitions we consider $\mathscr{C}, \mathscr{D} \in \mathbf{C}$ to be such that $E_C \cap E_D = \varnothing$.

2.3. Definition. The *choice between $\mathscr{C}$ and $\mathscr{D}$* is defined by

$$\mathscr{C} + \mathscr{D} := \langle C \cup D, \surd_C \cup \surd_D, \ell_C \cup \ell_D \rangle.$$

2.4. Definition. The *sequential composition of $\mathscr{C}$ and $\mathscr{D}$* is defined by

$$\mathscr{C}; \mathscr{D} := \langle C \cup D', \surd, \ell_C \cup \ell_D \rangle,$$

where $D' = \{F \cup G \mid F\surd_C, G \in D\}$ and $\surd = \{F \cup G \mid F\surd_C, G\surd_D\}$.

Before defining synchronisation and refinement on configuration structures (inspired, resp., by [12, 18]), we need some more notation first: if $F \subseteq (E \cup \{*\}) \times (E \cup \{*\})$ then

$$\pi_i(F) := \{e \mid \exists (e_1, e_2) \in F.\, e_i = e \neq *\}$$
$$\text{for} \quad i = 1, 2$$
$$F(d) := \{e \mid (d, e) \in F\}.$$

The latter regards F as a binary relation over $E \cup \{*\}$ and extends the notion of *image* of d from functions to this type of relations. It is used in *refinement* (see below), such that if d is a refined event then $F(d)$ is the configuration into which it is refined.

2.5. Definition. Let $\mathscr{C}_i = \langle C_i, \surd_i, \ell_i \rangle$ be configuration structures for $i = 1, 2$ and $A \subseteq Act$. The *synchronisation of $\mathscr{C}_1$ and $\mathscr{C}_2$ over A* is given by $\mathscr{C}_1 \|_A \mathscr{C}_2 := \langle C, \surd, \ell \rangle$ such that

- $E_C \subseteq \{(e, *) \mid \ell_1(e) \notin A\} \cup \{(*, e) \mid \ell_2(e) \notin A\} \cup \{(d, e) \mid \ell_1(d) = \ell_2(e) \in A\}$;
- $\varnothing \in C$; and for all $F \in C$ and $(e_1, e_2) \in E_C \backslash F$, $F \cup \{(e_1, e_2)\} \in C$ iff for both $i = 1, 2$, $e_i \notin \pi_i(F)$ and $\pi_i(F \cup \{(e_1, e_2)\}) \in C_i$;
- $F\surd$ iff (i) $F \in C$ and (ii) for both $i = 1, 2$, $\pi(F)\surd_i$;
- $\ell(e_1, e_2) = \begin{cases} \ell_1(e_1) & \text{if} \quad e_2 = *; \\ \ell_2(e_2) & \text{otherwise.} \end{cases}$

2.6. Definition. Let $\mathscr{C}, \mathscr{D} \in \mathbf{C}$ and $a \in Act$. The *refinement of a by $\mathscr{D}$ in $\mathscr{C}$* is given by $\mathscr{C}[a \rightsquigarrow \mathscr{D}] := \langle B, \surd, \ell \rangle$ such that

- $E_B = \{(d, *) \mid \ell_C(d) \neq a\} \cup \{(d, e) \in E_C \times E_D \mid \ell_C(d) = a\}$;
- $F \in B$ if and only if (i) $\pi_1(F) \in C$ and (ii) for all $(d, e) \in F$, either $e = *$ or $F(d) \in D$ and $\pi_1(F) \backslash \{d\} \notin C \Rightarrow F(d)\surd_D$;
- $F\surd$ if and only if (i) $\pi_1(F)\surd_C$ and (ii) for all $(d, e) \in F$, either $e = *$ or $F(d)\surd_D$.
- $\ell(d, e) = \begin{cases} \ell_C(d) & \text{of} \quad e = *; \\ \ell_D(e) & \text{otherwise.} \end{cases}$

All of the above operations are well-defined modulo isomorphism of the operands. We may therefore ignore the side condition of event disjointness in the definitions of choice and sequential composition: when applying these operations, one may always choose isomorphic representatives with disjoint event sets. The semantics of our language is given by a function $[\![\cdot]\!]\colon \Sigma \to \mathbf{C}$ (or, mathematically more precise, $[\![\cdot]\!]\colon \Sigma \to \mathbf{C}/\cong$), defined inductively as

$$[\![a]\!] := \langle \{\varnothing, \{e\}\}, \{e\}, (e, a) \rangle$$
$$[\![P + Q]\!] := [\![P]\!] + [\![Q]\!]$$
$$[\![P; Q]\!] := [\![P]\!]; [\![Q]\!]$$
$$[\![P \|_A Q]\!] := [\![P]\!] \|_A [\![Q]\!]$$
$$[\![P[a \rightsquigarrow Q]]\!] := [\![P]\!][a \rightsquigarrow [\![Q]\!]],$$

where the operators occurring on the left-hand side are the syntactic ones while those occurring on the right-hand side are the semantics ones defined above. This semantics then induces an equivalence relation $\cong$ over Σ, according to which two terms are equivalent when they give rise to isomorphic configuration structures:

$$P \cong Q :\Leftrightarrow [\![P]\!] \cong [\![Q]\!]$$

The following is trivially true.

2.7. PROPOSITION. *$\cong$ is a congruence for the operators of Σ.*

3. SYNTACTIC VERSUS SEMANTIC REFINEMENT

As mentioned in the introduction, the main goal of the paper is to investigate under which conditions syntactic action refinement coincides with its semantic version, presented in the previous section. Here we formally define what syntactic action refinement is. To this aim, we introduce the notation $P\{Q/a\}$ to denote the process term where all the occurrences of action a in P are replaced by Q. This intuitive concept can be rigorously defined by structural induction.

3.1. DEFINITION. Let $P, Q \in \Sigma_{flat}$ be two flat terms.

The operation of *syntactic substitution*, denoted $P\{Q/a\}$, is defined by induction on the syntactical structure of P as follows:

$$b\{Q/a\} := \begin{cases} Q & \text{if} \quad b = a \\ b & \text{otherwise} \end{cases}$$

$$(P_1 * P_2)\{Q/a\} := (P_1\{Q/a\}) * (P_2\{Q/a\}) \quad \text{where} \quad * \in \{+, ;\}$$

$$(P_1 \|_A P_2)\{Q/a\} := (P_1\{Q/a\}) \|_{A\{Q/a\}} (P_2\{Q/a\}).$$

The rule for parallel composition uses a special construct $A\{Q/a\}$, defined as follows:

$$A\{Q/a\} := \begin{cases} (A \setminus \{a\}) \cup L(Q) & \text{if} \quad a \in A \\ A & \text{otherwise.} \end{cases}$$

Note that we also substitute the actions in synchronisation sets. The following is immediate.

3.2. PROPOSITION. *If $P, Q \in \Sigma_{flat}$ and $a \in Act$ then $P\{Q/a\} \in \Sigma_{flat}$.*

We now define a *reduction* function over (flat and) non-flat terms which removes all occurrences of refinement operators from a given process expression, from the inside out so that syntactic substitution is only applied to terms which already have been reduced, i.e., to flat terms.

3.3. DEFINITION. The *reduction* of a term $P \in \Sigma$, denoted $red(P)$, is defined inductively on the structure of P as follows:

$$red(a) := a$$

$$red(P * Q) := red(P) * red(Q) \quad \text{where} \quad * \in \{+, ;, \|_A\}$$

$$red(P[a \rightsquigarrow Q]) := red(P)\{red(Q)/a\}.$$

Note that in the rule for refinement, we have $(\{a\} \cup L(P)) \cap L(Q) = \varnothing$ because we only consider well-formed terms. Due to Proposition 3.2, $red(P)\{red(Q)/a\}$ is always defined. The following proposition states that *red* is well-behaved in the sense that the alphabet and set of synchronising actions of a given term are insensitive to reduction of that term.

3.4. PROPOSITION. *If $P \in \Sigma$ then $L(red(P)) = L(P)$ and $S(red(P)) = S(P)$.*

Proof. Straightforward, by induction on the structure of P. The only interesting case is refinement, for which the property to be proved is the following: if P and Q are flat terms such that $(\{a\} \cup L(P)) \cap L(Q) = \varnothing$ then

$$L(P\{Q/a\}) = \begin{cases} (L(P) \setminus \{a\}) \cup L(Q) & \text{if} \quad a \in L(P) \\ L(P) & \text{otherwise} \end{cases}$$

$$S(P\{Q/a\}) = \begin{cases} (S(P) \setminus \{a\}) \cup L(Q) & \text{if} \quad a \in S(P) \\ S(P) \cup S(Q) & \text{if} \quad a \in L(P) \setminus S(P) \\ S(P) & \text{otherwise.} \end{cases}$$

The proof of these equalities is contained in the Appendix (Lemma A.1.). ∎

The aim of this paper can now be rephrased as follows: we are looking for general conditions under which

$$P \cong red(P), \tag{4}$$

where $\cong$ is the congruence induced by the isomorphism of the underlying configuration structures.

4. REFINEMENT OF NON-SYNCHRONISING ACTIONS

In this section, we focus our attention on a particular aspect of the problem which may be solved in a simple, neat way: the case when actions to be refined cannot be involved in a synchronisation. Recalling that $S(P)$ denotes the set of synchronising actions, this condition can be formally stated by requiring that $a \notin S(P)$ for any term of the form $P[a \rightsquigarrow Q]$. In order to prove (4) we firstly need the following lemma.

4.1. LEMMA. *Let $P, P_1, P_2, Q \in \Sigma$ be arbitrary terms, let $a, b \in Act$ and $A \subseteq Act$.*

1. $a[a \rightsquigarrow Q] \cong Q$
2. $b[a \rightsquigarrow Q] \cong b$ *provided that* $b \neq a$
3. $(P_1; P_2)[a \rightsquigarrow Q] \cong P_1[a \rightsquigarrow Q]; P_2[a \rightsquigarrow Q]$
4. $(P_1 + P_2)[a \rightsquigarrow Q] \cong P_1[a \rightsquigarrow Q] + P_2[a \rightsquigarrow Q]$
5. $(P_1 \|_A P_2)[a \rightsquigarrow Q] \cong P_1[a \rightsquigarrow Q] \|_A P_2[a \rightsquigarrow Q]$ *provided that* $a \notin A$.

Proof. Let us assume, with abuse of notation, that $[\![P_i]\!] = \mathscr{P}_i = \langle P_i, \surd_i, \ell_i \rangle$ and $[\![P_i[a \rightsquigarrow Q]]\!] = \mathscr{P}'_i = \langle P'_i, \surd'_i, \ell'_i \rangle$ for $i = 1, 2$. Finally let $\mathscr{Q} = \langle Q, \surd_Q, \ell_Q \rangle$ denote $[\![Q]\!]$.

1. Let $[\![a]\!] = \langle \{\varnothing, \{e\}\}, \{e\}, (e, a) \rangle$ and $[\![a[a \rightsquigarrow Q]]\!] = \mathscr{C} = \langle C, \surd, \ell \rangle$, hence $E_C = \{e\} \times E_Q$ by Definition 2.6. Therefore the function $f: E_C \to E_Q$ defined by $f: (e, d) \mapsto d$ is a bijection. Preservation of the configurations, of termination and of labelling are immediate; hence f is an isomorphism.

2. Trivial: in both structures there is only one event, which is labelled b.

3. Let $[\![(P_1; P_2)[a \rightsquigarrow Q]]\!] = \mathscr{C}$ and $[\![P_1[a \rightsquigarrow Q]; P_2[a \rightsquigarrow Q]]\!] = \mathscr{D}$. According to Definitions 2.4 and 2.6 we have

$$E_C = \{(e, *) \mid e \in (E_{P_1} \setminus \ell_1^{-1}(a)) \cup (E_{P_2} \setminus \ell_2^{-1}(a))\}$$
$$\cup \{(e, d) \mid e \in E_{P_1} \cup E_{P_2} \wedge \ell(e) = a \wedge d \in E_Q\}$$
$$E_{P'_i} = \{(e, *) \mid e \in E_{P_i} \setminus \ell_i^{-1}(a)\}$$
$$\cup \{(e, d) \mid e \in E_{P_i} \wedge \ell_i(e) = a \wedge d \in E_Q\}.$$

Since $E_D = E_{P'_1} \cup E_{P'_2} = E_C$, we can take as isomorphism simply the identity function. Preservation of the configurations, of the termination predicate and of the labelling function are obvious.

4. Similar to the previous one and thus omitted.

5. Let $[\![(P_1 \|_A P_2)[a \rightsquigarrow Q]]\!] = \mathscr{C}$ and $[\![P_1[a \rightsquigarrow Q] \|_A P_2[a \rightsquigarrow Q]]\!] = \mathscr{D}$. According to Definitions 2.5 and 2.6, we have then $E_C \subseteq X := \bigcup_{1 \leqslant i \leqslant 5} X_i$ and $E_D \subseteq Y := \bigcup_{1 \leqslant i \leqslant 5} Y_i$ where

$$X_1 := \{((e, *), *) \mid e \in E_{P_1} \setminus \ell_1^{-1}(a)\}$$
$$X_2 := \{((e, *), d) \mid e \in \ell_1^{-1}(a) \wedge d \in E_Q\}$$
$$X_3 := \{((*, e), *) \mid e \in E_{P_2} \setminus \ell_2^{-1}(a)\}$$
$$X_4 := \{((*, e), d) \mid e \in \ell_2^{-1}(a) \wedge d \in E_Q\}$$
$$X_5 := \{((e_1, e_2), *) \mid \ell_1(e) = \ell_2(e) \in A\}$$

$$Y_1 := \{((e, *), *) \mid e \in E_{P_1} \setminus \ell_1^{-1}(a)\}$$
$$Y_2 := \{((e, d), *) \mid e \in \ell_1^{-1}(a) \wedge d \in E_Q\}$$
$$Y_3 := \{(*, (e, *)) \mid e \in E_{P_2} \setminus \ell_2^{-1}(a)\}$$
$$Y_4 := \{(*, (e, d)) \mid e \in \ell_2^{-1}(a) \wedge d \in E_Q\}$$
$$Y_5 := \{((e_1, *), (e_2, *)) \mid \ell_1(e) = \ell_2(e) \in A\}.$$

Our candidate isomorphism $f: E_C \to E_D$ is defined as the restriction of the union $\bigcup_{1 \leqslant i \leqslant 5} f_i$ to E_C where $f_i: X_i \to Y_i$ are defined as follows:

$$f_1: \ ((e, *), *) \mapsto ((e, *), *)$$
$$f_2: \ ((e, *), d) \mapsto ((e, d), *)$$
$$f_3: \ ((*, e), *) \mapsto (*, (e, *))$$
$$f_4: \ ((*, e), d) \mapsto (*, (e, d))$$
$$f_5: ((e_1, e_2), *) \mapsto ((e_1, *), (e_2, *)).$$

Functions f_{1-5} are obviously bijective, as well as label-preserving. The proof of the preservation of the family of configurations is omitted; however, a similar proof for the more complex case of refinement of synchronised actions is reported in the next section. ∎

We would like to mention that in [19] we prove a similar lemma where $\cong$ is replaced by the stronger notion of *flow event structure isomorphism* (see [6]). Hence, (4) holds even for this finer equivalence. The following example shows that rule 5 of the lemma above does *not* hold in general for *non-well-formed terms*:

4.2. EXAMPLE. Let $P_1 = Q = a$, $A = \{a\}$ and $P_2 = a; b$.

$$(P_1 \|_A P_2)[b \rightsquigarrow Q] \cong (a \|_{\{a\}} a); a$$
$$P_1[b \rightsquigarrow Q] \|_A P_2[b \rightsquigarrow Q] \cong a \|_{\{a\}} (a; a).$$

These terms describe different behaviours. The upper one will execute action a twice and terminate successfully, whereas the lower one can execute only one a, whereafter it deadlocks: the right-hand synchronisation component wants to execute one more a in synchrony with the other, but the other component is already finished.

We now come to the first theorem, which states that for well-formed flat terms, refining a single non-synchronising action (semantically) is the same as substituting it (syntactically).

4.3. THEOREM. *Let $P, Q \in \Sigma_{flat}$ and $a \in Act$. If $a \notin S(P)$ then $P[a \rightsquigarrow Q] \cong P\{Q/a\}$.*

TABLE 3

Interference Freedom

$$\frac{}{\vdash_i a} \qquad \frac{\vdash_i P, Q \quad * \in \{+, ;, \|_A\}}{\vdash_i P * Q} \qquad \frac{\vdash_i P, Q \quad a \notin S(P)}{\vdash_i P[a \rightsquigarrow Q]}$$

Proof. By induction on the syntactic structure of P. The base cases are when P is an action. If $P = a$, then $a[a \rightsquigarrow Q] \cong Q$ by Lemma 4.1.1, and $Q \cong a\{Q/a\}$ because of Definition 3.1. Analogously if $P = b$. For the inductive case, let $* \in \{+, ;, \|_A\}$; then

$$\begin{aligned}
&(P_1 * P_2)[a \rightsquigarrow Q] \\
&\quad \cong P_1[a \rightsquigarrow Q] * P_2[a \rightsquigarrow Q] \\
&\qquad \text{(Lemma 4.1)} \\
&\quad \cong (P_1\{Q/a\}) * (P_2\{Q/a\}) \\
&\qquad \text{(induction hypothesis and congruence of } \cong) \\
&\quad = (P_1 * P_2)\{Q/a\} \\
&\qquad \text{(Definition 3.1).}
\end{aligned}$$

If $* = \|_A$ then Lemma 4.1 is applicable because $A \subseteq S(P)$, hence $a \notin A$. ∎

The following corollary, which extends the above result to the full language and multiple refinements, relies on a further predicate, called *interference freedom* and denoted $\vdash_i$. This predicate defined in Table 3, expresses that *no* actions being refined are synchronised on in the scope of the refinement.

4.4. COROLLARY. *Let $P \in \Sigma$. If $\vdash_i P$ then $P \cong red(P)$.*

Proof. Straightforward by induction on the syntactic structure of P. We show the case for refinement:

$$\begin{aligned}
red(P[a \rightsquigarrow Q]) &= red(P)\{red(Q)/a\} \\
&\qquad \text{(Definition 3.3)} \\
&\cong red(P)[a \rightsquigarrow red(Q)] \\
&\qquad \text{(Theorem 4.3)} \\
&\cong P[a \rightsquigarrow Q] \\
&\qquad \text{(induction and congruence of } \cong).
\end{aligned}$$

Theorem 4.3 is applicable because $\vdash_i P[a \rightsquigarrow Q]$ guarantees $a \notin S(P)$ $(= S(red(P)))$. ∎

5. REFINEMENT OF SYNCHRONISING ACTIONS

In this section we compare semantic and syntactic refinement for non-interference-free terms, i.e., terms in which it is allowed to refine synchronisation actions. The following example shows that rule 5 of Lemma 4.1 does not hold any more.

5.1. EXAMPLE. Let $P_1 = P_2 = a$, $A = \{a\}$ and $Q = b$.

$$\begin{aligned}
(P_1 \|_A P_2)[a \rightsquigarrow Q] &\cong b \|_{\{b\}} b \\
P_1[a \rightsquigarrow Q] \|_A P_2[a \rightsquigarrow Q] &\cong b \|_{\{a\}} a.
\end{aligned}$$

These terms are not equivalent: in the upper one, b is executed only once, whereas in the lower it is executed twice independently.

We can try to repair this situation by formulating a more accurate rule for distributing refinement over parallel composition. Since we are studying the correspondence of semantic and syntactic refinement, it is a natural choice to reuse the definition of syntactic substitution as a distribution rule for refinement, yielding

$$(P_1 \|_A P_2)[a \rightsquigarrow Q] \cong P_1[a \rightsquigarrow Q] \|_{(A \setminus \{a\}) \cup L(Q)} P_2[a \rightsquigarrow Q]. \tag{5}$$

(There are alternative ways of distributing refinement over synchronisation. In Section 8, we briefly discuss one particular other choice based on a CCS-like synchronisation operator.)

Example 5.1 above is indeed repaired by this change, because now the second term (in which refinement is distributed over the subterms) becomes

$$P_1[a \rightsquigarrow Q] \|_{(A \setminus \{a\}) \cup L(Q)} P_2[a \rightsquigarrow Q] \cong b \|_{\{b\}} b,$$

which is equivalent to the first (non-distributed) term. It is however important to note that there are instances of P_1, P_2 and Q for which we do not expect (5) to hold under *any* deadlock-sensitive equivalence relation. For instance, the following terms are not even completed trace equivalent (b is a completed trace in the second term but not in the first term):

$$(a \|_{\{a\}} a)[a \rightsquigarrow b; c_1 + b; c]$$

and

$$(a[a \rightsquigarrow b; c_1 + b; c_2]) \|_{\{b, c_1, c_2\}} (a[a \rightsquigarrow b; c_1 + b; c_2]).$$

Hence at this point, one possibility could be that of looking for a semantic notion other than $\cong$—if any such exists—under which (5) holds always. Instead, we investigate *necessary and sufficient conditions* for the validity of (5) under configuration structure isomorphism $\cong$. This extends rule 5 of Lemma 4.1 to non-interference-free terms. Note that this result necessarily depends on the choice of the

semantics: in a stronger semantics our conditions will in general to longer be sufficient, whereas in a weaker semantics they will no longer be necessary. We will come back to this point in Section 8.

Let us analyse the terms on both sides of the proposed new distribution rule (5). Define

$$\mathscr{C} = [\![(P_A \|_A P_2)[a \rightsquigarrow Q]]\!]$$
$$\mathscr{D} = [\![P_1[a \rightsquigarrow Q] \|_{(A\setminus\{a\}) \cup L(Q)} P_2[a \rightsquigarrow Q]]\!].$$

We can partially construct the event sets of $\mathscr{C}$ and $\mathscr{D}$: $E_C \subseteq \bigcup_{1 \leqslant i \leqslant 4} X_i$ and $E_D \subseteq \bigcup_{1 \leqslant i \leqslant 4} Y_i$ where

$$X_1 := \{((d_1, d_2), e) \in ((E_1 \times E_2) \times E_Q) \mid \ell_1(d_1) = \ell_2(d_2) = a)\}$$
$$X_2 := \{((d_1, d_2), *) \mid d_i \in E_i \wedge \ell_1(d_1) = \ell_2(d_2) \in (A\setminus\{a\})\}$$
$$X_3 := \{((d, *), *) \mid d \in E_1 \wedge \ell_1(d) \notin A\}$$
$$X_4 := \{((*, d), *) \mid d \in E_2 \wedge \ell_2(d) \notin A\}$$

$$Y_1 := \{((d_1, e_1), (d_2, e_2)) \mid \ell_1(d_1) = \ell_2(d_2) = a \wedge \ell_Q(e_1) = \ell_Q(e_2)\}$$
$$Y_2 := \{((d_1, *), (d_2, *)) \mid d_i \in E_i \wedge \ell_1(d_1) = \ell_2(d_2) \in (A\setminus\{a\})\}$$
$$Y_3 := \{((d, *), *) \mid d \in E_1 \wedge \ell_1(d) \notin A\}$$
$$Y_4 := \{(*, (d, *)) \mid d \in E_2 \wedge \ell_2(d) \notin A\}.$$

Now there is a natural candidate function $f: E_C \to E_D$ to prove $\mathscr{C} \cong \mathscr{D}$, viz. the restriction of the union $\bigcup_{1 \leqslant i \leqslant 4} f_i$ to E_C where $f_i: X_i \to Y_i$ are defined as follows:

$$f_1: ((d_1, d_2), e) \mapsto ((d_1, e), (d_2, e))$$
$$f_2: ((d_1, d_2), *) \mapsto ((d_1, *), (d_2, *))$$
$$f_3: ((d, *), *) \mapsto ((d, *), *)$$
$$f_4: ((*, d), *) \mapsto (*, (d, *)).$$

f_2–f_4 are obviously bijective. For f_1 this is not immediately clear; surjectivity requires that $e_1 = e_2$ for every $((d_1, e_1), (d_2, e_2)) \in (E_D \cap Y_1)$. We will show later that this is indeed the case. However, all f_j, and thereby also f, are clearly injective.

To formulate our conditions for the validity of (5) under $\cong$, we define a number of properties over configuration structures. For this purpose the following notation is useful: if $\mathscr{C}$ is a configuration structure and $F, G \in C$ then

$$F \xrightarrow{a}_C G :\Leftrightarrow \exists e \in G.F = G\setminus\{e\} \wedge \ell_C(e) = a.$$

We will drop the subscript C when it is clear from the context.

5.2. Definition. Let $\mathscr{C} = \langle C, \surd, \ell \rangle$ be a configuration structure; let a be an action.

- a is *executed* in $\mathscr{C}$ (at F) if $F \xrightarrow{a} G$ for some $F, G \in C$;
- a is *initial* in $\mathscr{C}$ if $\varnothing \xrightarrow{a} F$ for some $F \in C$.
- a is *noninitial* in $\mathscr{C}$ if $\varnothing \neq F \xrightarrow{a} G$ for some $F, G \in C$; otherwise a is *initial-only* in $\mathscr{C}$ (note that initial-only does not imply that a is in fact executed).
- a is *nondeterministic* in $\mathscr{C}$ if $F \xrightarrow{a} G$ and $F \xrightarrow{a} H \neq G$ for some $F, G, H \in C$; otherwise a is *deterministic* in $\mathscr{C}$;
- a is *auto-concurrent* in $\mathscr{C}$ (at F) if $F \xrightarrow{a} G$ and $F \xrightarrow{a} H \neq G$ and $G \cup H \in C$ for some $F, G, H \in C$; otherwise a is *auto-sequential* in $\mathscr{C}$ (at F). (Note that the auto-sequentiality of a also does not imply that a is actually executed.)

The following is a derived property that is defined only over structures of the form $\mathscr{C} = \mathscr{C}_1 \|_A \mathscr{C}_2$, where $\mathscr{C}_i \in \mathbf{C}$ for $i = 1, 2$:

- $a \in A$ is *two-way sequential* in $\mathscr{C}$ if a is auto-sequential in $\mathscr{C}_i$ at $\pi_i(F)$ for both $i = 1, 2$ whenever a is executed in $\mathscr{C}$ at F.

The following properties concern $\mathscr{C}$ as a whole, without reference to any particular action:

- $\mathscr{C}$ is *deterministic* if every action is deterministic in $\mathscr{C}$;
- $\mathscr{C}$ is *distinct* if $\mathscr{C}$ is deterministic and every initial action in $\mathscr{C}$ is initial-only;
- $\mathscr{C}$ is *atomic* if $\mathscr{C}$ is deterministic and *every* action is initial-only in $\mathscr{C}$ (hence all nonempty configurations in $\mathscr{C}$ are singleton sets).

In the following, we will say that a is executed, deterministic etc. in a *process term* P if it is executed, deterministic, etc. in $[\![P]\!]$.

The property of *two-way sequentiality* is the least familiar: it implies that every *execution of* a in a synchronisation is auto-sequential in both synchronising partners. It is slightly weaker than requiring that a is auto-sequential in both synchronising partners, since all a-autoconcurrent states in the partners may be unreachable (e.g., because of synchronisation deadlocks), in which case a is still two-way sequential.

5.3. Example. If $P_1 = a; (b \|_\varnothing b) + b$ and $P_2 = a + b; a$ then b is auto-concurrent in P_1 but two-way sequential in $P_1 \|_{\{a, b\}} P_2$.

We now present the main theorem of this paper.

5.4. THEOREM. *Let $P_1, P_2, Q \in \Sigma$ and $a \in A \subseteq Act$. The following distributivity property*

$$(P_1 \parallel_A P_2)[a \rightsquigarrow Q] \cong P_1[a \rightsquigarrow Q] \parallel_{(A \setminus \{a\}) \cup L(Q)} P_2[a \rightsquigarrow Q]$$

holds if and only if one of the following is satisfied:

C1. *a is not executed in $P_1 \parallel_A P_2$;*

C2. *a is two-way sequential in $P_1 \parallel_A P_2$, and Q is deterministic*;

C3. *a is auto-sequential in $P_1 \parallel_A P_2$, and Q is distinct*;

C4. *Q is atomic.*

Proof Strategy. We only give an outline of the proof here; the various steps are proved in Appendix A.

1. Prove $F_C \in C \Rightarrow f(F_C) \in D$, independent of conditions C1–C4 (Lemma A.5).

2. Prove $F_D \in D \Rightarrow \exists F_C \in C.\ f(F_C) = F_D$ (Lemma A.6) under each of the conditions C1–C4. This proves that f is onto E_D; because we already knew f to be injective, it follows that $f: E_C \to E_D$ is bijective and $f(C) = D$.

3. Prove $F_C \surd_C \Leftrightarrow f(F_C) \surd_D$ (Lemma A.7). Because we know that (the pointwise extension of) f is a bijection from C to D this proves $f(\surd_C) = \surd_D$.

4. $\ell_D = \ell_C \circ f^{-1}$ follows immediately from the analysis of the event sets and the definition of f, together with the fact that f is bijective.

This concludes the proof of the "if" part of the theorem.

5. If $\mathscr{C} \cong \mathscr{D}$ then $f(C) = D$. Now prove that each of conditions D1–D3 below is sufficient to construct a configuration in D which is not in $f(C)$ (Lemma A.8).

D1. a is executed in $P_1 \parallel_A P_2$ and Q is nondeterministic;

D2. a is not two-way sequential in $P_1 \parallel_A P_2$ and Q is not distinct;

D3. a is autoconcurrent in $P_1 \parallel_A P_2$ and Q is not atomic.

This concludes the proof of the "only if" part. ∎

5.5. EXAMPLE. To see the necessity of the conditions in the theorem above, consider $P_1 := a; c \parallel_\varnothing a; c$, $P_2 := a$, $A := \{a\}$ and $Q := b; b$. It follows that a is not two-way sequential in $P_1 \parallel_A P_2$ and Q is not distinct, and in fact we have

$$\begin{aligned}(P_1 \parallel_A P_2)[a \rightsquigarrow Q] &\cong ((a; c \parallel_\varnothing a; c) \parallel_{\{a\}} a)[a \rightsquigarrow b; b] \\ &\ncong (b; b; c \parallel_\varnothing b; b; c) \parallel_{\{b\}} b; b \\ &\cong (P_1[a \rightsquigarrow Q]) \parallel_{(A \setminus \{a\}) \cup L(Q)} (P_2[a \rightsquigarrow Q]).\end{aligned}$$

6. REDUCIBILITY

We now know precisely the conditions under which refinement is distributive over the various operators of Σ; it only remains to combine these separate facts and establish the class of (non-flat) terms in which all instances of refinement can be treated as syntactic substitution. We first give the semantic characterisation of the full class, and next a syntactic characterisation of a subclass.

6.1. *Semantic Reducibility*

As we defined it, syntactic substitution already deals with synchronising actions in the "proper" way, i.e., conforming to Theorem 5.4:

$$\begin{aligned}&\text{if } a \in A \text{ then } (P_1 \parallel_A P_2)\{Q/a\} \\ &\quad = (P_1\{Q/a\}) \parallel_{(A \setminus \{a\}) \cup L(Q)} (P_2\{Q/a\}).\end{aligned}$$

Hence for every term P containing refinement (possibly of synchronising actions), the reduction function $red: \Sigma \to \Sigma_{flat}$ yields a flat term which under the right circumstances is equivalent with P. "Under the right circumstances" here refers to the requirement that in synchronisation subterms, one of the four conditions of Theorem 5.4 should be fulfilled. To formalise this statement, we define the auxiliary property of *substitutivity* of a term P relative to a pair (a, Q), which basically reflects the fact that the refinement $P[a \rightsquigarrow Q]$ can be reduced away, i.e., replaced by syntactic substitutions $P\{Q/a\}$.

6.1. DEFINITION. (a, Q)-substitutivity is a property of flat terms, defined inductively by the following rules:

- b is (a, Q)-substitutive for all $b \in Act$.
- $P_1 + P_2$ and $P_1; P_2$ are (a, Q)-substitutive if P_1 and P_2 are (a, Q)-substitutive.
- $P_1 \parallel_A P_2$ is (a, Q)-substitutive if P_1 and P_2 are (a, Q)-substitutive and one of the following conditions holds:
  - $a \notin A$;
  - a is not executed in $P_1 \parallel_A P_2$;
  - a is two-way sequential in $P_1 \parallel_A P_2$ and Q is deterministic;
  - a is auto-sequential in $P_1 \parallel_A P_2$ and Q is distinct;
  - Q is atomic.

It follows (among other things) that P is automatically (a, Q)-substitutive if $a \notin L(P)$. Formally, (a, Q)-substitutivity corresponds to the property that refining a to Q syntactically and semantically has the same effect.

6.2. PROPOSITION. *$P \in \Sigma$ is (a, Q)-substitutive iff $P[a \rightsquigarrow Q] \cong P\{Q/a\}$.*

The proof is immediate by induction on the structure of P; the only interesting case is that of synchronisation, which however is covered by Theorem 5.4.

Subsequently, *reducibility* is a property of arbitrary terms, intended to capture the fact that a given term P possibly containing action refinement can be completely rewritten to a flat term $red(P)$ by interpreting all refinements syntactically.

6.3. DEFINITION. Reducibility is a property of terms, inductively defined by the following rules:

- a is reducible;
- $P+Q$, $P;Q$ and $P\|_A Q$ are reducible iff P and Q are reducible;
- $P[a \rightsquigarrow Q]$ is reducible iff P and Q are reducible and $red(P)$ is $(a, red(Q))$-substitutive.

The following theorem states that reducibility indeed captures the intended property. The proof is immediate by induction on the structure of P.

6.4. THEOREM. *$P \in \Sigma$ is reducible iff $P \cong red(P)$.*

6.2. *Syntactic Reducibility*

The conditions of Theorem 5.4 are based on the *semantic* properties in Definition 5.2. We are however also interested in a (decidable) *syntactic* characterisation of the (sub) language in which syntactic and semantic refinement coincide, i.e., which are reducible in the sense that $P \cong red(P)$.

We will only give *sufficient* syntactic conditions; we argue that it is useless to try giving necessary and sufficient conditions, e.g., for the occurrence of an action: such a result could never be extended to a language with recursion, since the halting problem can be reduced to it.[3] Also, necessary conditions are only necessary with respect to a given semantics: when moving to a weaker equivalence relation they are in general no longer necessary. Sufficient conditions, however, remain sufficient even with respect to weaker equivalences than $\cong$—which is important since most non-interleaving equivalences in the literature are indeed weaker than $\cong$.

We have chosen a fairly direct (not necessarily effective) encoding of the semantic properties, because we intend only to show that, in principle, it is possible to provide a syntactic characterisation of these conditions. In particular, of the four conditions in Theorem 5.4 under which $P_1\|_A P_2$ is (a, Q)-substitutive, the third one (a is auto-sequential in $P_1\|_A P_2$ and Q is distinct) is neglected, whereas the syntactic characterisation of the first condition (a does not occur in $P_1\|_A P_2$) will be subsumed by the case that a is not a synchronising action. A more detailed characterisation can be found in [19].

Table 4 defines various functions from Σ to $\mathbf{2}^{Act}$ inductively on the structure of the terms. I returns the *initial actions*, and D the set of *distributed actions* which may occur *auto-concurrently*. It follows that $I(P) \subseteq L(P)$ and $D(P) \subseteq L(P)$ for all $P \in \Sigma$. SD serves a more complicated purpose: it investigates in subterms of the form $P_1\|_A P_2$ which of the synchronising actions in A are already distributed in one of the operands. Hence, $SD(P) \subseteq S(P)$. This information is used to approximate the awkward semantical property of *two-way sequentiality*. Note that all of these functions, including L and S already defined in Section 2, are linear in the size of their arguments. The following proposition states that they indeed provide characterisations for the corresponding semantics properties.

6.5. PROPOSITION. 1. *If a is executed in P then $a \in L(P)$;*

2. *a is initial in P if and only if $a \in I(P)$;*

3. *If a is auto-concurrent in P then $a \in D(P)$.*

Proof. Straightforward; deferred to Appendix A.3. ∎

The following proposition states that all the syntactic functions above are insensitive to the reduction function *red*.

TABLE 4

Initial and Distributed Actions

$$
\begin{aligned}
I(a) &:= \{a\}\\
I(P+Q) &:= I(P) \cup I(Q)\\
I(P;Q) &:= I(P)\\
I(P\|_A Q) &:= ((I(P) \cup I(Q)) \setminus A) \cup (I(P) \cap I(Q) \cap A)\\
I(P[a \rightsquigarrow Q]) &:= \begin{cases} (I(P)\setminus\{a\}) \cup I(Q) & \text{if } a \in I(P)\\ I(P) & \text{otherwise.}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
D(a) &:= \varnothing\\
D(P * Q) &:= D(P) \cup D(Q), \text{ where } * \in \{+, ;\}\\
D(P;Q) &:= (D(P) \cap D(Q) \cap A)\\
&\qquad \cup (D(P) \cup D(Q) \cup (L(P) \cap L(Q))) \setminus A\\
D(P[a \rightsquigarrow Q]) &:= \begin{cases} (D(P)\setminus\{a\}) \cup L(Q) & \text{if } a \in D(P)\\ D(P) \cup D(Q) & \text{if } a \in L(P)\setminus D(P)\\ D(P) & \text{otherwise.}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
SD(a) &:= \varnothing\\
SD(P * Q) &:= SD(P) \cup SD(Q), \text{ where } * \in \{+, ;\}\\
SD(P\|_A Q) &:= SD(P) \cup SD(Q) \cup ((D(P) \cup D(Q)) \cap A)\\
SD(P[a \rightsquigarrow Q]) &:= \begin{cases} (SD(P)\setminus\{a\}) \cup L(Q) & \text{if } a \in SD(P)\\ SD(P) \cup D(Q) & \text{if } a \in S(P)\setminus SD(P)\\ SD(P) \cup SD(Q) & \text{if } a \in L(P)\setminus S(P)\\ SD(P) & \text{otherwise.}\end{cases}
\end{aligned}
$$

[3] More precisely, the halting problem can be reduced to the question whether P and $P\|_{\{a\}} \mathbf{0}$ (where $\mathbf{0}$ denotes a special *deadlock* constant which does nothing at all, in particular no a-action) are language equivalent, which is precisely the question whether a occurs in P. For the basic idea of the reduction we refer to [24], which discusses the comparable case of CCS.

This is necessary to make sure that in nested refinements, when a term is syntactically classified as reducible, this decision is not revoked after part of the reduction is done and some of the inner refinements are removed.

6.6. PROPOSITION. *If* $P \in \Sigma$ *then* $f(red(P)) = f(P)$ *for* $f = I, D, SD$.

Proof. Straightforward, by induction on the structure of P. Because *red* does not affect the outermost operator of P except if it is refinement, this is the only interesting case; the relevant property is stated and proved in the Appendix (Lemma A.9). ∎

Table 5 defines two predicates over Σ, intended to capture the rest of the semantics properties of Definition 5.2 in terms of syntax. $\vdash_{det} \subseteq \Sigma$ captures the notion of *determinism*, and $\vdash_{red} \subseteq \Sigma$ defines the notion of *syntactic reducibility*: in syntactically reducible terms, semantic refinement can be interpreted as syntactic substitution. The following proposition expresses that the sublanguage induced by $\vdash_{det}$ indeed contains only deterministic terms. Moreover, we introduce the notation $\sum A$ where A is a finite set of actions, for the choice between the actions in A; such terms are atomic in the sense of Definition 5.2.

6.7. PROPOSITION. 1. *If* $\vdash_{det} P$ *then* P *is deterministic*;
2. *If* $P = \sum A$ *then* P *is atomic.*

Proof. Straightforward; proofs of the more interesting cases are contained in Appendix A.3. ∎

Similar to the calculation of the action sets (cf. Proposition 6.6), we also need to know that the predicates defined above are insensitive to the process of reduction, with the same motivation: when some inner refinement operator of a number of nested refinements is reduced away by *red*, properties of the term as a whole should not be affected. This is formulated in the following proposition.

TABLE 5

Syntactic Determinism and Reducibility

$$\frac{}{\vdash_{det} a} \qquad \frac{\vdash_{det} P, Q \quad I(P) \cap I(Q) = \varnothing}{\vdash_{det} P + Q}$$

$$\frac{\vdash_{det} P, Q}{\vdash_{det} P; Q} \qquad \frac{\vdash_{det} P, Q \quad L(P) \cap L(Q) \subseteq A}{\vdash_{det} P \parallel_A Q}$$

$$\frac{\vdash_{det} P \quad a \notin L(P)}{\vdash_{det} P[a \rightsquigarrow Q]} \qquad \frac{\vdash_{det} P, Q}{\vdash_{det} P[a \rightsquigarrow Q]}$$

$$\frac{}{\vdash_{red} a} \qquad \frac{\vdash_{red} P, Q \quad * \in \{+, ;, \parallel_A\}}{\vdash_{red} P * Q}$$

$$\frac{\vdash_{red} P, Q \quad a \notin S(P)}{\vdash_{red} P[a \rightsquigarrow Q]} \qquad \frac{\vdash_{red} P, Q \quad a \notin SD(P) \ \vdash_{det} Q}{\vdash_{red} P[a \rightsquigarrow Q]}$$

$$\frac{\vdash_{red} P}{\vdash_{red} P[a \rightsquigarrow \sum A]}$$

6.8. PROPOSITION. *Let* $P \in \Sigma$.

1. *If* $\vdash_{det} P$ *then* $\vdash_{det} red(P)$.
2. *If* $\vdash_{red} P$ *then* $\vdash_{red} red(P)$.

Proof. By induction on the structure of P. Again, just as for Proposition 6.6, the proof is trivial except for refinement, because *red* does not actually affect the top level operator. For the case of refinement the necessary property is stated and proved in the appendix (Lemma A.10.). ∎

It is our intention that syntactic reducibility implies, but is not necessarily implied by, semantic reducibility (Theorem 6.4). This leads to the following theorem, which states that for reducible flat terms, semantic and syntactic refinement coincide.

6.9. THEOREM. *Let* $P, Q \in \Sigma_{flat}$ *and* $a \in Act$. *If* $\vdash_{red} P[a \rightsquigarrow Q]$ *then* $P[a \rightsquigarrow Q] \cong P\{Q/a\}$.

Proof. By induction on the structure of P. The cases are analogous to the proof of Theorem 4.3, except if $P = P_1 \parallel_A P_2$ such that $a \in A$. By induction $P_i\{Q/a\} \cong P_i[a \rightsquigarrow Q]$ for both $i = 1, 2$. There are two subcases.

- $a \notin SD(P)$ and $\vdash_{det} Q$. It follows, by construction of D, that $a \notin D(P_1) \cup D(P_2)$; hence by Proposition 6.5, a is not auto-concurrent in P_1 or P_2; hence a is two-way sequential in $P_1 \parallel_A P_2$. In addition, Q is deterministic by Proposition 6.7.1.
- $Q = \sum A$. It follows by Proposition 6.7.2 that Q is atomic.

In both cases, due to Theorem 5.4 we have

$$\begin{aligned} P[a \rightsquigarrow Q] &\cong P_1[a \rightsquigarrow Q] \parallel_{(A \setminus \{a\}) \cup L(Q)} P_2[a \rightsquigarrow Q] \\ &\cong P_1\{Q/a\} \parallel_{A\{Q/a\}} P_2\{Q/a\} \\ &= P\{Q/a\}. \end{aligned}$$

This concludes the proof. ∎

The following corollary extends the above result to Σ, using Propositions 6.6 and 6.8 which state that our syntactic machinery is insensitive to the application of the function *red*: by removing refinement operators from the inside out using *red*, it is ensured that syntactic substitution is applied only to flat terms. It follows that every reducible term can be rewritten to a flat term.

6.10. COROLLARY. *Let* $P \in \Sigma$. *If* $\vdash_{red} P$ *then* $P \cong red(P)$.

Proof. By induction on the structure of P. The only interesting case is refinement. Assume $P = P_1[a \rightsquigarrow Q]$; then $\vdash_{red} P_1$ and $\vdash_{red} Q$, implying $\vdash_{red} red(P_1)$ and $\vdash_{red} red(Q)$ by Proposition 6.8.2, and moreover one of the following holds:

- $a \notin S(P)$, hence $a \notin S(red(P_1))$ by Proposition 3.4;
- $a \notin SD(P)$ and $\vdash_{red} Q$, hence $a \notin SD(red(P))$ by Proposition 6.6 and $\vdash_{det} red(Q)$ by Proposition 6.8.1.
- $Q = \sum A = red(Q)$ is atomic by Proposition 6.7.2.

Each of these cases implies $\vdash_{red} red(P_1)[a \rightsquigarrow red(Q)]$.

$$\begin{aligned} red(P) &= red(P_1)\{red(Q)/a\} \\ &\cong red(P_1)[a \rightsquigarrow red(Q)] \\ &\cong P_1[a \rightsquigarrow Q] \end{aligned}$$

The second step is by Theorem 6.9, and the third one by the induction hypothesis and the fact that $\cong$ is a congruence with respect to refinement. ∎

7. RECURSION

In this section, we consider the effect of *recursion* on the issue of syntactic versus semantics refinement. In particular, we extend the characterisation of *reducibility* in Theorem 6.4 to a language including recursion. For the purpose of specifying recursive behaviour we introduce a set of *process names* Var, elements of which are denoted X, Y; Σ_{rec} denotes the language obtained by extending Σ to allow such names to occur in terms; i.e., Σ_{rec} is generated by the extended grammar

$$P ::= a \mid P + P \mid P; P \mid P \parallel_A P \mid P[a \rightsquigarrow P] \mid X,$$

where $X \in Var$ and the rest is as before (Section 2). The meaning of the process names is determined by a *process environment*, which is a function $\theta: Var \to \Sigma_{rec}$ assigning a *definition* to each process name. A term is *guarded* if it contains process names X only in subterms of the form $P; X$ (where P is an arbitrary term). An environment θ is guarded if its images $\theta(X)$ are all guarded terms.

The semantics of recursion is obtained through a standard fixpoint construction. We will go quickly over this construction; see, e.g., Winskel [33] for a more careful treatment. The set of configuration structures forms a complete partial order $\langle \mathbf{C}, \subseteq \rangle$ in which the supremum of a given configuration structure chain $\mathscr{C}_0 \subseteq \mathscr{C}_1 \subseteq \cdots$ is given by its union $\bigcup_{i \in N} \mathscr{C}_i$. The operators of our language are all based on pointwise constructions on single configurations, and are therefore continuous. One then defines θ-approximations X^i_θ for all $X \in Var$ and $i \in N$:

$$X^i_\theta = \begin{cases} \mathbf{0} & \text{if } \ i = 0 \\ \theta(X)\{\vec{Y}^{i-1}_\theta / \vec{Y}\} & \text{otherwise,} \end{cases}$$

where $\mathbf{0}$ is a special constant denoting *deadlock* interpreted by the smallest element of $\langle \mathbf{C}, \subseteq \rangle$, viz., the configuration structure $\{\varnothing\}$, and where $\vec{Y}$ is the vector of process names occurring in $\theta(X)$. It follows that for all $X \in Var$, $[\![X^0_\theta]\!] \subseteq [\![X^1_\theta]\!] \subseteq \cdots$ is a chain of configuration structures, the supremum of which is the smallest solution in $\mathbf{C}$ (for X) of the equation $\vec{Y} = \theta(\vec{Y})$, i.e., the smallest fixpoint. The semantics of θ then is a function $[\![\theta]\!]: Var \to \mathbf{C}$ defined by $[\![\theta]\!] = \lambda X. \bigcup_{i \in N} [\![X^i_\theta]\!]$. By inductive definition over the structure of terms, this gives rise to a semantics function $[\![\_]\!]_\theta: \Sigma_{rec} \to \mathbf{C}$ such that $[\![X]\!]_\theta := [\![\theta]\!](X)$. Now the following property of this semantics is vital to our purposes. It can be proved in an analogous way to the uniqueness of the interleaving semantics of guarded recursive terms; see [24, 8].

7.1. PROPOSITION. *If $\theta: Var \to \Sigma_{rec}$ is guarded, then the system of equations determined by $\vec{X} = \theta(\vec{X})$ has exactly one solution in $\mathbf{C}$ up to $\cong$, which is given by $[\![\theta]\!]$.*

Just as for the finite language, the question we try to answer is under which circumstances the refinement operators in a term may be interpreted by syntactic substitution. The answer turns out to depend whether refinement only occurs *outside* recursion, or also *within* it: the notion of reducibility extends easily to the first case, but not to the second one.

Notation. We will sometimes write $P \cong_\theta Q$ for $P, Q \in \Sigma_{rec}$ to express that θ is a process environment under which P and Q yield isomorphic semantics. We also use λ-calculus notation for functions θ, such that $\lambda X. P_X$ denotes the function which returns P_Y when applied to any variable $Y \in Var$.

7.1. *Refinement Outside Recursion*

We first restrict the terms $\theta(X)$ to $\Sigma_{flat, rec}$, i.e., to flat recursive terms, meaning that we do not allow refinement to appear within recursion. Let us call two process environments θ, η *equivalent* denoted $\theta \cong \eta$, if they give rise to equivalent semantics, i.e., if $[\![\theta]\!](X) \cong [\![\eta]\!](X)$ for all X (and hence $[\![P]\!]_\theta \cong [\![P]\!]_\eta$ for all $P \in \Sigma_{rec}$). Below we show one way to construct equivalent environments.

7.2. COROLLARY. *Let $\theta: Var \to \Sigma_{rec}$ be a process environment and for all $X \in Var$ let P_X be a guarded term such that $P_X \cong_\theta X$. Then $\theta \cong \lambda X. P_X$.*

Proof. Let $\eta := \lambda X. P_X$. By assumption, $[\![X]\!]_\theta \cong [\![\eta(X)]\!]_\theta$ and hence $[\![\theta]\!]$ solves the system of equations $\vec{Y} = \eta(\vec{Y})$ generated by η. The result therefore follows from Proposition 7.1. ∎

The properties of auto-concurrency, determinism etc. in Definition 5.2, on which the main result Theorem 5.4 relies, carry over smoothly to infinite configuration structures; in particular the proof of the main theorem does not depend on its operands being finite and hence the theorem remains

valid in Σ_{rec}. The only remaining problem is therefore to reduce terms of the form $X[a \rightsquigarrow Q]$. The essential idea is as follows: one introduces a *new* process name X_a^Q and extends the process environment by (essentially) mapping X_a^Q to $\theta(X)[a \rightsquigarrow Q]$. It should be obvious that $\theta(X)[a \rightsquigarrow Q]$ is guarded if both $\theta(X)$ and Q are guarded. $X[a \rightsquigarrow Q]$ is then reduced to X_a^Q, which is easily seen to be equivalent to $X[a \rightsquigarrow Q]$: since $X \cong_\theta \theta(X)$ by definition and $\cong$ is a congruence, it follows that $X_a^Q \cong_\theta \theta(X)[a \rightsquigarrow Q] \cong_\theta X[a \rightsquigarrow Q]$.

At first sight, this introduction of new process names on encountering subterms of the form $X[a \rightsquigarrow Q]$ might seem to lead to an infinite regression wherein ever more process names are needed. This, however, is actually not the case, since if in the process of reducing $\theta(X)[a \rightsquigarrow Q]$ we encounter another copy of X then the relevant term to be reduced would again be $X[a \rightsquigarrow Q]$, and we could reuse the same "new" name X_a^Q.

To formalise this idea, we introduce a *dependency ordering* over the process names Var relative to the process environment θ; $\prec_\theta \subseteq Var \times \Sigma_{rec}$ is the smallest "transitive" relation such that $X \prec_\theta P$ (pronounced "X is used in P") if X is a subterm of P, or, when P is itself a variable, X is a subterm of $\theta(P)$. Transitivity of this relation refers to the case where $X \prec_\theta Y \prec_\theta P$, meaning that X is used in $\theta(Y)$ and Y in P; then we also say that X is used in P, hence $X \prec_\theta P$. Note that $\prec_\theta$ might be a proper preorder if θ specifies mutual recursion. Furthermore we add the rule

$$X\{Q/a\} := X_a^Q$$

to Definition 3.1, where $X_a^Q \notin Var$ is a new process name; θ is extended with the definition

$$\theta: X_a^Q \mapsto \theta(X)\{Q/a\}.$$

Finally, we add the following clause to Definition 6.1:

- X is (a, Q)-substitutive for all $X \in Var$.

The following is immediate (by induction on the structure of P):

7.3. Lemma. *If P is (a, Q)-substitutive and $X[a \rightsquigarrow Q] \cong_\theta X_a^Q$ for all $X \prec_\theta P$, then $P[a \rightsquigarrow Q] \cong_\theta P\{Q/a\}$.*

Proposition 6.2, which states under which conditions a *single* refinement can be interpreted syntactically, can now be generalised to Σ_{rec}. Let us call $P \in \Sigma_{flat,\,rec}$ *recursively (a, Q)-substitutive* if P is (a, Q)-substitutive and for all $X \prec_\theta P$, $\theta(X)$ is (a, Q)-substitutive.

7.4. Proposition. *$P \in \Sigma_{flat,\,rec}$ is recursively (a, Q)-substitutive iff $P[a \rightsquigarrow Q] \cong P\{Q/a\}$.*

Proof. The interesting case is of course that of recursion, which is proved according to the outline given above. Assume that Var contains "new" process names X_a^Q for all $X \prec_\theta P$, where $\theta(X_a^Q) := \theta(X)\{Q/a\}$ as above. Define a new process environment η: $Var \to \Sigma_{rec}$ which differs from θ on the new variables:

$$\eta = \lambda X. \quad \text{if } X = Y_a^Q \text{ then } \theta(Y)[a \rightsquigarrow Q] \text{ else } \theta(X).$$

We will prove that θ is equivalent to η. First note that $X \cong_\eta \eta(X) = \theta(X)$ for all "old" names $X \in Var$. On the other hand, for all "new" names X_a^Q (i.e., for all $X \prec_\theta P$) we have

$$X[a \rightsquigarrow Q] \cong_\eta \eta(X)[a \rightsquigarrow Q] = \theta(X)[a \rightsquigarrow Q] = \eta(X_a^Q) \cong_\eta X_a^Q.$$

It follows (Lemma 7.3) that $P[a \rightsquigarrow Q] \cong_\eta P\{Q/a\}$ and $\theta(X)[a \rightsquigarrow Q] \cong_\eta \theta(X)\{Q/a\}$ for all $X \prec_\theta P$; the latter implies that

$$X_a^Q \cong_\eta \theta(X)[a \rightsquigarrow Q] \cong_\eta \theta(X)\{Q/a\} = \theta(X_a^Q).$$

According to Corollary 7.2 it follows that $\eta \cong \theta$. We may conclude $[\![P[a \rightsquigarrow Q]]\!]_\theta \cong [\![P[a \rightsquigarrow Q]]\!]_\eta \cong [\![P\{Q/a\}]\!]_\eta \cong [\![P\{Q/a\}]\!]_\theta$. ∎

Reducibility, being the property that *nested* refinements can be interpreted syntactically, can now also be generalised to Σ_{rec}, by changing the condition under which $P[a \rightsquigarrow Q]$ is called reducible: P should be *recursively* (a, Q)-substitutive rather than just (a, Q)-substitutive. The complete definition becomes as follows (compare Definition 6.3):

- a is reducible for all $a \in Act$;
- X is reducible for all $X \in Var$;
- $P + Q$, $P; Q$ and $P \parallel_A Q$ are reducible iff P and Q are reducible;
- $P[a \rightsquigarrow Q]$ is reducible iff P and Q are reducible and $red(P)$ is recursively $(a, red(Q))$-substitutive.

This gives us the desired result.

7.5. Theorem. *$P \in \Sigma_{rec}$ is reducible iff $P \cong red(P)$.*

7.2. *Syntactic Reducibility*

The syntactic characterisation of reducibility that we have presented in Section 6.2 can also be generalised to the language with recursion. Whereas for the semantic case, Theorem 5.4 extended directly to infinite operands, in the syntactic case we are forced to give explicit constructions for the functions L, S etc. as well as the predicates $\vdash_{det}$ and $\vdash_{red}$ over recursive terms. This is again done through smallest fixpoint constructions.

7.6. Definition.

- f: $\Sigma_{rec} \to \mathbf{2}^{Act}$ (where $f = L, S, I, D, SD$) returns the smallest set such that the equations in Tables 1 and 4 are satisfied and moreover $f(\vec{X}) = f(\theta(\vec{X}))$;

- $\vdash_x \subseteq \Sigma_{rec}$ (where $x = det, red$) is the largest predicate such that the rules in Table 5 are satisfied and moreover $\vdash_x \vec{X} \Leftrightarrow \vdash_x \theta(\vec{X})$.

The reason why this yields the appropriate notions is that the functions L, S, I, D, and SD as well as the predicates $\vdash_{det}$ and $\vdash_{red}$ are compositional and continuous with respect to the operators of Σ (on their respective domains $\mathbf{2}^{Act}$ ordered by subset inclusion for the functions and $\{\mathrm{tt}, \mathrm{ff}\}$ ordered by $\mathrm{ff} \leqslant \mathrm{tt}$ for the predicates). Therefore the fixpoint can in each case be constructed as the limit of the results obtained for the approximations X^i_θ. (Note that we need to take the *largest* predicates to interpret the $\vdash_x$: otherwise for instance the term X with definition $\theta(X) = a;\ X$ would not be deterministic.) The following property is therefore immediate.

7.7. Proposition. *Let $X \in Var$ be arbitrary.*

- *$f(X) = \bigcup_{i \in N} f(X^i_\theta)$ for all $f = L, S, I, D, SD$;*
- *$\vdash_x X \Leftrightarrow \forall i \in N.\ \vdash_x X^i_\theta$ for $x = det, red$.*

As a matter of fact, in the absence of refinement from the process environment θ, we only have to approximate to a depth bounded by the number of process names, $|Var|$: we state without proof that $f(X) = f(X^{|Var|+1}_\theta)$ and $\vdash_x X \Leftrightarrow \vdash_x X^{|Var|+1}_\theta$.

It remains to be proved that these functions and predicates still have the same meaning as in the finite case, i.e., that they provide sufficient conditions for the corresponding semantical properties.

7.8. Proposition. *Let $P \in \Sigma_{rec}$ be arbitrary.*

- *If a is executed in P then $a \in L(P)$;*
- *a is initial in P if and only if $a \in I(P)$;*
- *If a is auto-concurrent in P then $a \in D(P)$;*
- *If $\vdash_{det} P$ then P is deterministic.*

This follows immediately from the finite case (see Propositions 6.5 and 6.7) plus the following property, which in turns is an immediate consequence of the semantics of recursion (see above) and the definition of the various semantical properties (see Definition 5.2):

7.9. Lemma. *Let $a \in Act$ and $X \in Var$ be arbitrary.*

- *a is executed in X iff a is executed in X^i_θ for some $i \in N$;*
- *a is initial in X iff a is initial in X^i_θ for some $i \in N$;*
- *a is auto-concurrent in X iff a is auto-concurrent in X^i_θ form some $i \in N$;*
- *X is deterministic iff X^i_θ is deterministic for all $i \in N$.*

The following then extends Corollary 6.10 and is proved in much the same way, with the help of Theorem 7.5 above:

7.10. Theorem. *Let $P \in \Sigma_{rec}$. If $\vdash_{red} P$ then $P \cong red(P)$.*

7.11. Example. Consider $Var = \{X\}$ with the following definition:

$$X =_\theta a; b \parallel_{\{a,b\}} c;\ X + (a \parallel_\varnothing a; b)$$

Intuitively X can do any number of c actions, followed nondeterministically by either a and deadlock, or ab and successful termination. It follows that $L(X) = \{a, b, c\}$, $S(X) = \{a, b\}$, $I(X) = \{a, c\}$, $D(X) = \varnothing$ and $SD(X) = \{a\}$.

Now consider the process $P := a; b + X$. It follows that $P[a \rightsquigarrow d; e]$ does not satisfy $\vdash_{red}$ because $a \in SD(P)$ and $d; e$ is not of the form $\sum A$; on the other hand, $\vdash_{red} P[b \rightsquigarrow d; e]$, and $red(P[b \rightsquigarrow d; e]) = a; d; e + X^{d;e}_b$ where

$$X^{d;e}_b =_\theta a; d; e \parallel_{\{a,d,e\}} c;\ X^{d;e}_b + (a \parallel_\varnothing a; d; e).$$

According to the same intuition as above, $X^{d;e}_b$ can do any number of c's, then either a whereafter it deadlocks or ade whereafter it terminates.

7.3. *Refinement within Recursion*

A natural further question is whether it is possible to replace every *process environment* with refinement by an equivalent flat one. The answer is positive, mainly because the property of *well-formedness* that we have implicitly assumed to hold imposes a strong restriction on the kinds of refinement that can be allowed in process environments. For instance, if $\theta(X) = (a; X)[a \rightsquigarrow Q]$ then intuitively, X is not well-formed, because in fact $L(X) \cap L(Q) = L(Q)$. This is stated more generally in the following proposition.

7.12. Proposition. *A process environment θ: $Var \to \Sigma_{rec}$ is well-formed only if for any $X \in Var$ and any subterm $P[a \rightsquigarrow Q]$ of $\theta(X)$, $X \not\prec_\theta P$ and $X \not\prec_\theta Q$.*

This basically implies that the reduction process cannot get into an infinite cycle of introducing ever more process names; hence the techniques discussed in Section 7.1 still apply. Therefore Theorems 7.5 and 7.10 continue to hold.

The situation becomes radically different if we lift the condition of well-formedness. In that case, there is no general way to reduce terms. The reason is that action refinement within process environments in a sense "dynamically" changes the process environment every time it is unfolded; it thereby becomes possible to "encode" fairly complex behaviour in action refinements. For instance, consider

$$\theta\colon X \mapsto a; b; (X[a \rightsquigarrow a; b]),$$

which specifies a process having as partial runs all prefixes of the infinite string $ab^1ab^2ab^3\cdots$. This set of partial runs is not context-free; it follows that the behaviour of X cannot

be specified in $\Sigma_{flat,\,rec}$ using only a finite number of process names, not even modulo trace equivalence, let alone modulo any stronger semantics. In other words, no finite flat process environment can be equivalent to θ. Nor has this failure anything to do with parallel composition or synchronisation: it already occurs in the sequential, choice-less fragment of the language.

8. CONCLUSION

We have compared notions of syntactic substitution and semantic refinement, the latter of which is interpreted as a form of substitution as well, albeit on a semantic domain. In particular we have investigated conditions under which the two notions give rise to the same semantics, or in other words, refinement operators can be *removed* from terms by repeated syntactic substitution. It turns out that as long as we do not refine synchronising actions, the correspondence can be established under only mild assumptions on the alphabets, called well-formedness. This condition can furthermore be done away with at the cost following a *renaming* operator in the languages as illustrated in Section 8.1.

If we do allow synchronisation actions to be refined, then the correspondence is less straightforward. For this case we establish necessary and sufficient semantic properties for the distribution of refinement over synchronisation, and sufficient syntactic conditions under which refinement can be removed completely. The sensitivity of our results to the chosen semantic equivalence is briefly discussed in Section 8.2.

Finally, a short discussion on related work is contained in Section 8.3.

8.1. *Non-Well-Formed Terms*

We want to show that there is rather simple a way to deal with terms $P[a \rightsquigarrow Q]$ not satisfying the well-formedness condition (3). The possible confusion of abstraction levels, generated by the substitution of Q for a in P when $(\{a\} \cup L(P)) \cap L(Q) \neq \varnothing$, can be removed by suitably renaming the actions of Q. Hence one has to consider a slightly more general language, where also a renaming operator is allowed. We will show that any non-well-formed term P is equivalent (i.e., gives rise to isomorphic configuration structures) to a term R of the extended language that does satisfy (3).

Let $\varphi: Act \to Act$ be a total function over the set Act of actions, *not necessarily injective*. The set of terms, ranged over by R, is generated by extending Σ with the further operator $R\varphi$. This set is denoted Π, whilst $\Pi_{flat} \subseteq \Pi$ denotes the set of terms that do not contain refinement operators. Furthermore we define

$$L(R\varphi) := \varphi(L(R))$$

$$S(R\varphi) := \varphi(S(R))$$

$$red(R\varphi) := red(R)\,\varphi$$

The well-formedness relation $\vdash$ can be easily defined also over terms having the renaming operator as top operator in their abstract syntax tree:

$$\frac{\vdash R}{\vdash R\varphi}.$$

The denotational semantics for the renaming operator can be given easily, as well,

$$[\![R\varphi]\!] := [\![R]\!]\,\varphi,$$

where the *renaming of* $\mathscr{C} = \langle C, \surd, \ell \rangle$ *according to* φ is defined by

$$\mathscr{C}\varphi := \langle C, \surd, \varphi \circ \ell \rangle.$$

Given a term $P[a \rightsquigarrow Q] \in \Sigma$ which is not well-formed, we can always find an injective renaming function $\psi: Act \to Act$ such that $L(Q\psi) \cap (\{a\} \cup L(P)) = \varnothing$. Hence, $P[a \rightsquigarrow Q]$ can be replaced by the well-formed term $(P[a \rightsquigarrow Q\psi])\,\psi^{-1} \in \Pi$.

8.1. THEOREM. *Let* $P[a \rightsquigarrow Q] \in \Sigma$ *such that* $\nvdash P[a \rightsquigarrow Q]$. *There always exists an injective renaming* ψ *such that*:

(i) $L(Q\psi) \cap (\{a\} \cup L(P)) = \varnothing$;

(ii) $\vdash (P[a \rightsquigarrow Q\psi])\,\psi^{-1}$;

(iii) $P[a \rightsquigarrow Q] \cong (P[a \rightsquigarrow Q\psi])\,\psi^{-1}$.

We will call a function ψ *suitable* if it satisfies the conditions of the theorem above. The theorem then justifies the introduction of an auxiliary function $wf: \Sigma \to \Pi$ which transforms any term $P \in \Sigma$ into an equivalent, *well-formed* term $R \in \Pi$. Function wf is defined by structural induction as follows:

$$wf(a) := a$$

$$wf(P * Q) := wf(P) * wf(Q) \quad \text{where} \quad * \in \{+, ;, \|_A\}$$

$$wf(P[a \rightsquigarrow Q]) := (wf(P)[a \rightsquigarrow wf(Q\psi)])\,\psi^{-1},$$

where in the last rule, ψ should be a suitable renaming.

The remainder of the section is devoted to extend the definitions and results of the paper to non well-formed terms. So, we first define syntactic substitution and then we prove that the main theorems can be trivially lifted to the present case. In order to define how syntactic substitution

applies to renaming terms $(P\varphi)\{Q/a\}$, we need to introduce a shorthand notation, based on an obvious property. Given a set $A = \{a_1, ..., a_n\}$, $A \cap L(Q) = \varnothing$, let

$$P\{Q/A\} := (\cdots(P\{Q/a_1\})\cdots\{Q/a_n\}).$$

This notation makes sense because, given $P, Q \in \Sigma_{flat}$ such that $a, b \notin L(Q)$, we have that

$$(P\{Q/a\})\{Q/b\} = (P\{Q/b\})\{Q/a\}.$$

Hence, we can extend syntactic substitution to terms with renaming as follows: given $P, Q \in \Pi_{flat}$ and a suitable renaming ψ, we have

$$(P\varphi)\{Q/a\} = (P\{Q\psi/A\})(\psi^{-1} \circ \varphi),$$

where $A = \varphi^{-1}(a) \cap L(P)$.

A similar shorthand can also be introduced for action refinement, although a bit more care is needed. In fact, even if $P, Q \in \Pi_{flat}$ are such that $(\{a\} \cup L(P)) \cap L(Q) = \varnothing$ and $a, b \notin L(Q)$, the term $(P[a \rightsquigarrow Q])[b \rightsquigarrow Q]$ is not well-formed. However, with a suitable renaming ψ, the following holds:

$$(P[a \rightsquigarrow Q])[b \rightsquigarrow Q\psi]\,\psi^{-1} \cong (P[b \rightsquigarrow Q\psi])[a \rightsquigarrow Q]\,\psi^{-1}.$$

Given a set $A = \{a_1, ..., a_n\}$, $A \cap L(Q) = \varnothing$, and a set of suitable renamings ψ_i for $i = 1, ..., n$, let us define the following useful abbreviation

$$P[A \rightsquigarrow Q] := (\cdots(P[a_1 \rightsquigarrow Q\psi_1])\cdots[a_n \rightsquigarrow Q\psi_n])\,\psi_1^{-1} \circ \cdots \circ \psi_n^{-1}.$$

Finally we can extend Lemma 4.1 to renamings.

8.2. LEMMA. *Let $P, Q \in \Pi_{flat}$, $a \in Act$, and let φ be a renaming. Then*

$$(P\varphi)[a \rightsquigarrow Q] \cong (P[A \rightsquigarrow Q\psi])(\psi^{-1} \circ \varphi),$$

where $A = \varphi^{-1}(a) \cap L(P)$ and ψ is a suitable renaming, i.e., $(A \cup L(P)) \cap L(Q\psi) = \varnothing$.

Note that if P and Q are well-formed, so is $(P[A \rightsquigarrow Q\psi])(\psi^{-1} \circ \varphi)$. The final result is that any term $P \in \Sigma$, be it well-formed or not, is equivalent to a well-formed term $R \in \Pi_{flat}$. This extends Corollaries 4.4 and 6.10.

8.3. THEOREM. *Let $P \in \Sigma$. The following hold*:

1. $\vdash_i P \Rightarrow P \cong red(wf(P))$;
2. $\vdash_{red} P \Rightarrow P \cong red(wf(P))$.

In the presence of recursion, the situation becomes more complicated. Where previously, reducing a refinement $P[a \rightsquigarrow Q]$ required the introduction of new names X_a^Q only for the process names $X \prec_\theta P$, in combination with renaming, especially using non-injective functions, it is possible that many more new names are required.

8.4. EXAMPLE. Consider $X =_\theta (a; X)\,\varphi$ where φ maps a to b, b to c and all other actions to themselves. Reducing $X[c \rightsquigarrow c_1; c_2]$ yields $X_c^{c_1; c_2}$ where

$$X_c^{c_1; c_2} =_\theta (a; X_{b,c}^{c_1; c_2})\,\varphi$$
$$X_{b,c}^{c_1; c_2} =_\theta (a; X_{a,b,c}^{c_1; c_2})\,\varphi$$
$$X_{a,b,c}^{c_1; c_2} =_\theta (c_1; c_2; X_{a,b,c}^{c_1; c_2})\,\varphi.$$

It is seen that rather than a single new name, this simple reduction requires the introduction of three new names.

In general, the number of new names required depends on the size of the inverse image of the renaming functions inside the recursion. To take a simple case, if $Var = \{X\}$ and a single renaming function φ appears in the definition of X, one may define

$$\hat{\varphi}^{-i}(a) = \begin{cases} \{a\} & \text{if} \quad i = 0 \\ \{a\} \cup \varphi^{-1}(\hat{\varphi}^{-(i-1)}(a)) & \text{otherwise.} \end{cases}$$

Intuitively, $\hat{\varphi}^{-n}(a)$ is the set of all actions that are renamed into a by at most n applications of φ, as in the nth unrolling of $\theta(X)$. Now the number of new names required for the reduction of $X[a \rightsquigarrow Q]$ is one larger than the number of steps in which the $\hat{\varphi}^{-i}$-chain stabilises; i.e., the number of new names equals $n+1$ where n is the smallest number such that $\hat{\varphi}^{-n}(a) = \hat{\varphi}^{-(n+1)}(a)$. For instance, in the above example we have $\hat{\varphi}^{-1}(a) = \{a, b\}$ and $\hat{\varphi}^{-n}(a) = \{a, b, c\}$ for $n \geqslant 2$; hence the number of new names is 3.

Because of these additional technical difficulties, we will not treat the combination of renaming and recursion in detail.

8.2. *Sensitivity to Different Semantics*

One of the parameters in our comparison is the equivalence relation being considered. We work with isomorphism of configuration structures, which is quite strong; so strong, in fact, that all but a very few of the semantics that have been proposed in the literature are (strictly) less distinguishing. (The few exceptions are: *event structure isomorphism*, e.g., isomorphism of flow or stable event structures, which also distinguishes on the basis of "accidental" differences, e.g., the presence of self-conflicting events in flow event structures, and are therefore in general stronger than configuration structure isomorphism; and equivalences based on *localities* such as in Boudol *et al.* [7],

which are in general incomparable with event-based semantics.) The necessity of our semantic conditions for reducibility is obviously relative to the choice of the semantics: it may be expected to disappear in weaker semantics. For instance, in configuration structure isomorphism we have the inequivalence

$$a + a + a + a \not\cong a + a$$

with the consequence

$$(b \parallel_{\{b\}} b)[a \rightsquigarrow a + a] \not\cong (b[b \rightsquigarrow a + a]) \parallel_{\{a\}} (b[b \rightsquigarrow a + a]).$$

In fact this instance of distribution is ruled out by our conditions (Theorem 5.4) because the refinement term $a + a$ is not deterministic, and hence not atomic. However, there are many partial order bisimulation relations weaker than $\cong$, for instance *history preserving bisimulation* [27], which equates $P + P$ and P and hence also $a + a + a + a$ and $a + a$; hence under such a relation our conditions are no longer necessary. For instance, as the above example shows, the side condition of determinism may be removed from the property of atomicity. We conjecture that a generalisation of Theorem 5.4 to any other semantic equivalence $\sim$ can be obtained by parametrically relaxing the definitions of determinism, distinctness and atomicity as follows:

Q is deterministic if

$$Q \sim Q \parallel_{L(Q)} Q$$

Q is distinct if

$$Q; Q \sim Q; Q \parallel_{L(Q)} (Q \parallel_{\varnothing} Q)$$

Q is atomic if

$$(Q \parallel_{\varnothing} Q) + (Q \parallel_{\varnothing} Q) \sim (Q \parallel_{\varnothing} Q) \parallel_{L(Q)} (Q \parallel_{\varnothing} Q).$$

As a matter of fact, if we choose $\cong$ in place of $\sim$, we would get an alternative formulation of these notions, yet equivalent to those in Definition 5.2. Note that to check this conjecture for any other equivalence would be a task analogous to the one undertaken in this paper for $\cong$. We consider this to be an issue for further research.

On the other hand, the syntactic conditions we develop, which are sufficient to guarantee the correspondence of refinement to syntactic substitution, will obviously remain sufficient when the equivalence relation is weakened.

8.3. *Related Work*

The work in [25] can be considered as a forerunner of the present research; there a process algebra with refinement (but without communication and recursion) is given a linear-time, causality based semantics, and syntactic substitution is proved to agree with the semantic operator.

The problem of relating the two approaches is taken in the opposite direction in [23]: syntactic substitution, without any limitations, is taken as the starting point and the emphasis is on finding a sensible semantic operation which coincides with it. It turns out that a combination of semantic refinement and *self-synchronisation* is enough to achieve this.

Syntactic refinement has also been investigated in depth in [1, 2] for a finite process algebra: the second paper combines it with CCS synchronisation. There is however no notion of semantic refinement, and consequently the relation between the two approaches is not considered. Indeed, [2] allows refinements which would contradict the commutativity of diagram (1) in the Introduction under common interpretations of semantic refinement. Consider for instance the following CCS variation of Example 5.5: let $P := (a; c \mid a; c) \mid \bar{a}$ and let $\rho: a \mapsto b; b$ be a refinement function (mapping the complement of a to the complement of $\rho(a)$). In the execution of P, the action c is always performed; however this is not the case in the execution of

$$P\rho \simeq P\{\rho(a)/a\} = (b; b; c \mid b; b; c) \mid \bar{b}; \bar{b}.$$

On the other hand, in a CCS setting, such as that of [1, 2], our choice for the distribution rule may be questioned. As seen in those papers, one may choose to take advantage of the inherent asymmetry of the barred and unbarred versions of every action by refining those versions differently; i.e., such that the refinements of a and $\bar{a}$ are defined independently. The main requirement is then that the *synchronisation* of those refinements satisfies certain constraints. In our setting this idea could be implemented by a rule of the form

$$(P_1 \parallel_A P_2)[a \rightsquigarrow Q] \cong (P_1[a \rightsquigarrow Q_1]) \parallel_{(A \setminus \{a\}) \cup A'} (P_2[a \rightsquigarrow Q_2]),$$

where Q_1, Q_2 and A' are such that $Q_1 \parallel_{A'} Q_2 \cong Q$. There is however no obvious notion of syntactic substitution which coincides with this.

In [15], a language similar to ours and with a similar denotational semantics, is considered. There the emphasis in on finding an SOS operational semantics agreeing with the denotational one, up to causal bisimulation. Similar comparisons of operational and denotational models are documented in [9] (up to ST-bisimulation) and [29] (up to (essentially) configuration structure isomorphism). Also our paper can be examined in this perspective. Indeed, syntactic substitution provides a simple sound and complete—with some limitations—implementation technique for semantic action refinement up to isomorphism of configuration structures; the operational semantics of a term $P[a \rightsquigarrow Q]$ is the transition system with initial state $red(P)\{Q/a\}$ which, being flat, can be dealt with in a standard way.

We would also like to mention the approach documented in [3] in which the set of refinable symbols and synchronisable actions are explicitly kept disjoint. This means that if $P[a \rightsquigarrow Q]$ is a term then a can never be synchronised within P, and our criterion of interference-freedom is always fulfilled. Hence in this approach, (1) always commutes: syntactic and semantic refinement *always* coincide.

Recently, some new approaches to action refinement have been proposed that do not enforce the complete inheritance of causes: see, for instance, [28], which discusses a different framework for refinement, in which the question of semantic refinement does not occur in the same form, and [32], in which refinement is *parameterised* with a dependency relation. The question of semantic vs. syntactic refinement in the latter setting is under study.

APPENDIX A: PROOFS

A.1. *Proofs of Section* 3

A.1. LEMMA. *If* $P, Q \in \Sigma_{flat}$ *such that* $(\{a\} \cup L(P)) \cap L(Q) = \varnothing$ *and* $a \in Act$ *then*

$$L(P\{Q/a\}) = \begin{cases} (L(P)\setminus\{a\}) \cup L(Q) & \text{if} \quad a \in L(P) \\ L(P) & \text{otherwise} \end{cases}$$

$$S(P\{Q/a\}) = \begin{cases} (S(P)\setminus\{a\}) \cup L(Q) & \text{if} \quad a \in S(P) \\ S(P) \cup S(Q) & \text{if} \quad a \in L(P)\setminus S(P) \\ S(P) & \text{otherwise.} \end{cases}$$

Proof. By induction in the structure of P.

Actions. If $P = b$ then

$$L(P\{Q/a\}) = \begin{cases} L(Q) & \text{if } a = b \\ L(P) & \text{otherwise} \end{cases}$$

$$= \begin{cases} (L(P)\setminus\{a\}) \cup L(Q) & \text{if } a \in L(P) = \{b\} \\ L(P) & \text{otherwise.} \end{cases}$$

$$S(P\{Q/a\}) = \begin{cases} S(Q) & \text{if } a = b \\ S(P) & \text{otherwise} \end{cases}$$

$$= \begin{cases} (S(P)\setminus\{a\}) \cup L(Q) & \text{if } a \in S(P) = \varnothing \\ S(P) \cup S(Q) & \text{if } a \in L(P)\setminus S(P) \\ & \quad = \{b\} \\ S(P) & \text{otherwise.} \end{cases}$$

Choice and Sequential Composition. Straightforward.

Synchronisation. To minimise the number of brackets we will use L_P to denote $L(P)$ etc, and $A\setminus a$ to denote $A\setminus\{a\}$ etc. If $P = P_1 \parallel_A P_2$ then the following holds, where the first equality is by definition of L on parallel composition and the second one by induction.

$$L(P\{Q/a\}) = \begin{cases} L_{P_1\{Q/a\}} \cup L_{P_2\{Q/a\}} \cup (A\setminus a) \cup L_Q & \text{if} \quad a \in A \\ L_{P_1\{Q/a\}} \cup L_{P_2\{Q/a\}} \cup A & \text{otherwise} \end{cases}$$

$$= \begin{cases} ((L_{P_1}\setminus a) \cup L_Q) \cup ((L_{P_2}\setminus a) \cup L_Q) \cup (A\setminus a) \cup L_Q & \text{if} \quad a \in A \cap L_{P_1} \cap L_{P_2} \\ ((L_{P_1}\setminus a) \cup L_Q) \cup L_{P_2} \cup (A\setminus a) \cup L_Q & \text{if} \quad a \in (A \cap L_{P_1})\setminus L_{P_2} \\ L_{P_1} \cup ((L_{P_2}\setminus a) \cup L_Q) \cup (A\setminus a) \cup L_Q & \text{if} \quad a \in (A \cap L_{P_2})\setminus L_{P_1} \\ L_{P_1} \cup L_{P_2} \cup (A\setminus a) \cup L_Q & \text{if} \quad a \in A\setminus(L_{P_1} \cup L_{P_2}) \\ ((L_{P_1}\setminus a) \cup L_Q) \cup ((L_{P_2}\setminus a) \cup L_Q) \cup A & \text{if} \quad a \in (L_{P_1} \cap L_{P_2})\setminus A \\ ((L_{P_1}\setminus a) \cup L_Q) \cup L_{P_2} \cup A & \text{if} \quad a \in L_{P_1}\setminus(L_{P_2} \cup A) \\ L_{P_1} \cup ((L_{P_2}\setminus a) \cup L_Q) \cup A & \text{if} \quad a \in L_{P_2}\setminus(L_{P_1} \cup A) \\ L_{P_1} \cup L_{P_2} \cup A & \text{otherwise} \end{cases}$$

$$= \begin{cases} ((L_{P_1} \cup L_{P_2} \cup A)\setminus A) \cup L_Q & \text{if} \quad a \in A \cup L_{P_1} \cup L_{P_2} \\ L_{P_1} \cup L_{P_2} \cup A & \text{otherwise} \end{cases}$$

$$= \begin{cases} (L(P)\setminus a) \cup L(Q) & \text{if} \quad a \in L(P) \\ L(P) & \text{otherwise} \end{cases}$$

In the proof for S, we skip one step.

$$S(P\{Q/a\}) = \begin{cases} S_{P_1\{Q/a\}} \cup S_{P_2\{Q/a\}} \cup ((L_{P_1\{Q/a\}} \cup L_{P_2\{Q/a\}}) \cap ((A\setminus a) \cup L_Q)) & \text{if} \quad a \in A \\ S_{P_1\{Q/a\}} \cup S_{P_2\{Q/a\}} \cup ((L_{P_1\{Q/a\}} \cup L_{P_2\{Q/a\}}) \cap A) & \text{otherwise} \end{cases}$$

$$= \begin{cases} ((S_{P_1} \cup S_{P_2} \cup ((L_{P_1} \cup L_{P_2}) \cap A)) \setminus a) \cup L_Q & \text{if} \quad a \in A \cup S_{P_1} \cup S_{P_2} \\ S_{P_1} \cup S_{P_2} \cup ((L_{P_1} \cup L_{P_2}) \cap A) \cup S_Q & \text{if} \quad a \in (L_{P_1} \setminus S_{P_1}) \cup (L_{P_2} \setminus S_{P_2}) \\ S_{P_1} \cup S_{P_2} \cup ((L_{P_1} \cup L_{P_2}) \cap A) & \text{otherwise} \end{cases}$$

$$= \begin{cases} (S(P) \setminus a) \cup L(Q) & \text{if} \quad a \in S(P) \\ S(P) \cup S(Q) & \text{if} \quad a \in L(P) \setminus S(P) \\ S(P) & \text{otherwise.} \end{cases}$$

∎

A.2. *Proof of the Main Theorem*

Note that the first condition of Definition 2.5 does not characterise E_C completely: there may be synchronisation events that are prevented from ever occurring. In the following propositions we assume $\mathscr{C}_i = \langle C_i, \sqrt{}_i, \ell_i \rangle \in \mathbf{C}$ for $i = 1, 2$, and $\mathscr{C} := \mathscr{C}_1 \parallel_A \mathscr{C}_2 = \langle C, \sqrt{}, \ell \rangle$.

A.2. Proposition. *For all $F \in C$, π_i is injective on F and $\pi_i(F) \in C_i$ for $i = 1, 2$.*

Proof. Straightforward by induction on $|F|$. ∎

A.3. Proposition. *If $F \in C$ and $\pi_i(F) \setminus \{e_i\} \in C_i$ for $i = 1, 2$, then $F \setminus \{(e_1, e_2)\} \in C$.*

Proof. By induction on $|F|$.

Base Case. If $|F| = 0$ there are no such e_i; if $|F| = 1$ the property is trivial.

Induction Step. Assume the lemma holds whenever $|F| = n \geqslant 1$; now let $F \in C$ be such that $|F| = n + 1$, and define $G_i := \pi_i(F)$ for $i = 1, 2$. Let $G \in C$ be such that $G \xrightarrow{(d_1, d_2)} F$; define $F_i := \pi_i(G)$ for $i = 1, 2$. Because both π_i are injective on F (Proposition A.2) it follows that $d_i = e_i$ for either $i = 1$ or $i = 2$ implies $(d_1, d_2) = (e_1, e_2)$, in which case the result is trivial. Now assume $d_i \neq e_i$ for both $i = 1, 2$. Because $G_i \setminus \{e_i\} \cup F_i = G_i \in C_i$ it follows (by definition of stable configuration structures) that $F_i \setminus \{e_i\} = (G_i \setminus \{e_i\}) \cap F_i \in C_i$. Hence by induction $G' := G \setminus \{(e_1, e_2)\} \in C$, and hence $G \setminus \{(e_1, e_2)\} = G' \cup \{(d_1, d_2)\} \in C$ by Definition 2.5. ∎

We come to the actual proof of Theorem 5.4. For ease of reference we copy some of the definitions from Section 5. Let

$$\mathscr{C} = [\![(P_1 \parallel_A P_2)[a \rightsquigarrow Q]]\!]$$

$$\mathscr{D} = [\![P_1[a \rightsquigarrow Q] \parallel_{(A \setminus \{a\}) \cup L(Q)} P_2[a \rightsquigarrow Q]]\!];$$

then $E_C \subseteq X := \bigcup_{1 \leqslant i \leqslant 4} X_i$ and $E_D \subseteq Y := \bigcup_{1 \leqslant i \leqslant 4} Y_i$ where

$$X_1 := \{((d_1, d_2), e) \in ((E_1 \times E_2) \times E_Q) \mid \ell_1(d_1) = \ell_2(d_2) = a\}$$

$$X_2 := \{((d_1, d_2), *) \mid d_i \in E_i \wedge \ell_1(d_1) = \ell_2(d_2) \in (A \setminus \{a\})\}$$

$$X_3 := \{((d, *), *) \mid d \in E_1 \wedge \ell_1(d) \notin A\}$$

$$X_4 := \{((*, d), *) \mid d \in E_2 \wedge \ell_2(d) \notin A\}$$

$$Y_1 := \{((d_1, e_1), (d_2, e_2)) \mid \ell_1(d_1) = \ell_2(d_2) = a \wedge \ell_Q(e_1) = \ell_Q(e_2)\}$$

$$Y_2 := \{((d_1, *), (d_2, *)) \mid d_i \in E_i \wedge \ell_1(d_1) = \ell_2(d_2) \in (A \setminus \{a\})\}$$

$$Y_3 := \{((d, *), *) \mid d \in E_1 \wedge \ell_1(d) \notin A\}$$

$$Y_4 := \{(*, (d, *)) \mid d \in E_2 \wedge \ell_2(d) \notin A\}.$$

$f: E_C \to E_D$ is defined as the restriction of the union $\bigcup_{1 \leqslant i \leqslant 4} f_i$ to E_C where $f_i: X_i \to Y_i$ are defined as follows:

$$f_1: \ ((d_1, d_2), e) \mapsto ((d_1, e), (d_2, e))$$

$$f_2: ((d_1, d_2), *) \mapsto ((d_1, *), (d_2, *))$$

$$f_3: \ ((d, *), *) \mapsto ((d, *), *)$$

$$f_4: \ ((*, d), *) \mapsto (*, (d, *)).$$

We present one auxiliary lemma, which holds by definition of f.

A.4. Lemma. *If $F \subseteq X$ then*

$$\forall (d_1, d_2) \in \pi_1(F).\ F(d_1, d_2) = \pi_1(f(F))(d_1) = \pi_2(f(F))(d_2).$$

Some (abuse of) notations: $[\![P_i]\!] = \langle P_i, \sqrt{}_i, \ell_i \rangle$ and

$$\underbrace{\overbrace{(P_1 \parallel_A P_2)}^{\langle P, \sqrt{}_P, \ell_P \rangle}[a \rightsquigarrow \overbrace{Q}^{\langle Q, \sqrt{}_Q, \ell_Q \rangle}]}_{C} \qquad \underbrace{\overbrace{P_1[a \rightsquigarrow Q]}^{\langle P'_1, \sqrt{}'_1, \ell'_1 \rangle} \parallel_{(A \setminus \{a\}) \cup L(Q)} \overbrace{P_2[a \rightsquigarrow Q]}^{\langle P'_2, \sqrt{}'_2, \ell'_2 \rangle}}_{D}$$

Moreover, if we have $G_C \xrightarrow{a}_C F_C$ then we denote

$$F_P := \pi_1(F_C) \qquad (\in P)$$
$$F_i := \pi_i(F_P) \qquad (\in P_i)$$
$$F_D := f(F_C)$$
$$F'_i := \pi_i(F_D)$$
$$G_P := \pi_1(G_C) \qquad (\in P)$$
$$G_i := \pi_i(G_P) \qquad (\in P_i)$$
$$G_D := f(G_C) \qquad (\in D)$$
$$G'_i := \pi_i(G_D) \qquad (\in P'_i).$$

Note that by construction, $F_i = \pi_1(F'_i)$ and $G_i = \pi_1(G'_i)$ for $i = 1, 2$. The $\in$-relations between brackets partly follow from Proposition A.2, and partly from Definition 2.6. Figure 1 may be of use in keeping track of the various definitions. Now we prove the points of the proof strategy of Theorem 5.4.

A.5. LEMMA. $F \in C \Rightarrow f(F) \in D$.

Proof. By induction on the size of $F \in C$.

Zero step. If $|F| = 0$ then $f(F) = \varnothing \in D$.

Induction step. Assume that $F \in C \Rightarrow f(F) \in D$ for all F such that $|F| = n$. Now let $F_C \in C$ such that $|F_C| = n + 1$. From Definition 2.1 it follows that there is a configuration $G_C \in C$ such that $G_C \subset F_C$ and $|G_C| = n$. By the induction hypothesis it follows that $f(G_C) \in D$.

First we prove $F'_i \in P'_i$ for both $i = 1, 2$. By construction $\pi_1(F'_i) = F_i \in P_i$. On the other hand, if $(d, e) \in F'_i$ then there exists exactly one $(d_1, d_2) \in \pi_1(F_C)$ such that $d_i = d$, because π_i is injective on F_P for all $F_P \in P$ (see Proposition A.2). By construction of f, it follows that $((d_1, d_2), e) \in F_C \Leftrightarrow (d_i, e) \in F'_i$ and hence $F_C(d_1, d_2) = F'_i(d_i)$; hence either $e = *$ or $F'_i(d_i) \in F_Q$, and furthermore if $F_i \backslash \{d_i\} \notin P_i$ then also $F_P \backslash \{(d_1, d_2)\} \notin P$, which proves $\neg F_C(d_1, d_2) \surd_Q$, hence $\neg F'_i(d) \surd_Q$. It follows by Definition 2.6 that $F'_i \in P'_i$.

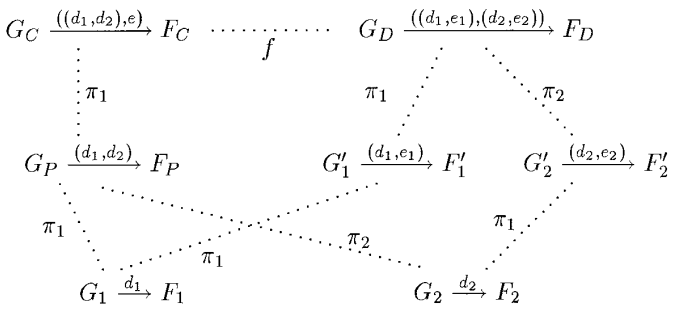

FIG. 1. Names and connections used in the proofs.

Next we prove $f(F_C) = F_D \in D$. Let $((d_1, d_2), e)$ be the event uniquely determined by $((d_1, d_2), e) \in F_C \backslash G_C$. It follows that $d_i = *$ or $d_i \in F_i \backslash G_i$ for $i = 1, 2$. We use case distinction based on the index k in $((d_1, d_2), e) \in X_k$:

$k = 1, 2$ It follows that $f((d_1, d_2), e) = ((d_1, e), (d_2, e))$; hence $(d_i, e) \notin G'_i$ for $i = 1, 2$, implying $\pi_i(G_D) \neq \pi_i(G_D) \cup \{(d_i, e)\} = G'_i \cup \{(d_i, e)\} = F'_i \in P'_i$.

$k = 3$ It follows that $d_2 = e = *$ and $f((d_1, d_2), e) = ((d_1, *), *)$; hence $(d_1, e) \notin G'_1$, implying $\pi_1(G_D) \neq \pi_1(G_D) \cup \{(d_1, e)\} = G'_1 \cup \{(d_1, e)\} = F'_1 \in P'_1$.

$k = 4$ Symmetrical to $k = 3$.

In each of these cases it follows by Definition 2.5 that $F_D \in D$. ∎

A.6. LEMMA. *If $P_1 \|_A P_2$ and Q satisfy one of the following conditions*

C1. *a is not executed in $P_1 \|_A P_2$;*

C2. *a is two-way sequential in $P_1 \|_A P_2$ and Q is deterministic;*

C3. *a is auto-sequential in $P_1 \|_A P_2$, and Q is distinct;*

C4. *Q is atomic.*

then $F_D \in D \Rightarrow \exists F_C \in C.\, f(F_C) = F_D$.

Proof. By induction on the size of configurations in D.

Zero step. If $|F_D| = 0$ then $F_C = \varnothing$ fulfills the condition.

Induction step. Assume that $F \in D \Rightarrow \exists F_C \in C.\, f(F_C) = F$ when $|F| = n$. Now let $F_D \in D$ such that $|F_D| = n + 1$. From Definition 2.1 it follows that there is a configuration $G_D \subset F_D$ such that $|G_D| = n$. By the induction hypothesis it follows that $G_D = f(G_C)$ for some $G_C \in C$. Hence the only objects of Figure 1 that we do not have are $F_P \in P$ and F_C such that $\pi_1(F_C) = F_P$ and $f(F_C) = F_D$; the proof obligation is to construct those.

If it has been established that F_C exists such that $f(F_C) = F_D$ and $\pi_1(F_C) \in P$, we can prove $F_C \in C$ as follows. Let $((d_1, d_2), e) \in F_C$ be arbitrary such that $e \neq *$; then for both $i = 1, 2$, $F_C(d_1, d_2) = F'_i(d_i)$ according to Lemma A.4; hence $F_C(d_1, d_2) \in Q$ because $F'_1 \in P'_1$. If $\pi_1(F_C) \backslash \{(d_1, d_2)\} \notin P$ then either $F_1 \backslash \{d_1\} \notin P_1$ or $F_2 \backslash \{d_2\} \notin P_2$ (Proposition A.3.), hence $\neg F'_i(d_i) \surd_Q$ for either $i = 1$ or $i = 2$, hence $\neg F_C(d_1, d_2) \surd_Q$. It follows that $F_C \in C$ by Definition 2.6.

We now proceed to prove the existence of F_C (i.e., such that $f(F_C) = F_D$ and $\pi_1(F_C) \in P$) by a case analysis on the shape of the event in $F_D \backslash G_D \subseteq Y_k$.

Y_1. $F_D \backslash G_D = \{((d_1, e_1), (d_2, e_2))\}$ where $d_i \in E_i$, $e_i \in E_Q$ such that $\ell_1(d_1) = \ell_2(d_2) = a$ and $\ell_Q(e_1) = \ell_Q(e_2)$, which is to say that $((d_1, e_1), (d_2, e_2))$ synchronises two actions from refinements of a, i.e. two actions of Q. Now we define

$$F_P := G_P \cup \{(d_1, d_2)\}$$
$$F_C := G_C \cup \{((d_1, d_2), e_1)\}.$$

First we prove $F_P \in P$. For this purpose it suffices to prove that either $F_P = G_P$, i.e. $(d_1, d_2) \in G_P$, or $d_i \notin G_i$ for both $i = 1, 2$; the other conditions of Definition 2.5 are already taken care of. Hence we assume $d_1 \in G_1$ and $d_2 \notin G_1$ and show that this leads to a contradiction under all conditions Cn. First note that $d_1 \in G_1$ implies $(d_1, d_2') \in G_P$ for some d_2', where $\ell_P(d_1, d_2') = \ell_1(d_1) = a$.

C1. $(d_1, d_2') \in G$ with $\ell_P(d_1, d_2') = a$ contradicts the assumption that a does not occur in P.

C2. Because $G_1'(d_1) \xrightarrow{\ell_Q(e_1)} F_1'(d_1)$ it follows that $G_1'(d_1)$ is not maximal in Q, and hence $\neg G_1'(d_1) \surd_Q$; hence also $\neg G_2'(d_2') \surd_Q$ by Lemma A.4; hence $\neg F_2'(d_2') \surd_Q$. It follows by Definition 2.6 that $H_2 := F_2 \backslash \{d_2'\} \in P_2$. It follows that $H_2 \cup G_2 = F_2 \in P_2$, and hence also $H_2 \cap G_2 \in P_2$ by the definition of stable configuration structures. But this implies

$$H_2 \cap G_2 \xrightarrow{\ell_2(d_2')}_P H_2 \xrightarrow{\ell_2(d_2)}_P F_2$$
$$H_2 \cap G_2 \xrightarrow{\ell_2(d_2)}_P G_2 \xrightarrow{\ell_2(d_2')}_P F_2.$$

This implies that $a = \ell_2(d_2) = \ell_2(d_2')$ is auto-concurrent in P_2 at H_2; hence a is not two-way sequential in P.

C3. It follows that $\varnothing \subset G_1'(d_1) \xrightarrow{\ell_Q(e_1)}_Q F_1'(d_1)$, whereas $\varnothing = G_2'(d_2) \xrightarrow{\ell_Q(e_2)}_Q F_2'(d_2)$. Since $\ell_Q(e_1) = \ell_Q(e_2)$ this contradicts the distinctness of Q.

C4. Since $\varnothing \subset G_1'(d_1) \subset F_1'(d_1)$ it follows that Q is not atomic.

By construction it follows immediately that $\pi_1(F_C) = F_P$. Now we prove $f(F_C) = F_D$. For this we merely need $e_1 = e_2$. $(d_1, d_2) \in F_P \in P$ proves that $\ell_P(d_1, d_2) = a$ occurs in P and hence C1 does not hold. Lemma A.4 implies $G_1'(d_1) = G_2'(d_2)$; from C2–C4, we know that Q is deterministic, hence from $G_i'(d_i) \xrightarrow{\ell_Q(e_i)}_Q F_i'(d_i)$ for $i = 1, 2$ and $\ell_Q(e_1) = \ell_Q(e_2)$ it follows that $F_1'(d_1) = F_2'(d_2)$, hence $e_1 = e_2$.

Y_2. $F_D \backslash G_D = \{((d_1, *), (d_2, *))\}$ where $d_i \in E_i$ for both $i = 1, 2$, corresponding to the synchronisation of two "ordinary," i.e. non-refined events. Let

$$F_P := G_P \cup \{(d_1, d_2)\}$$
$$F_C := G_C \cup \{((d_1, d_2), *)\}.$$

From the injectivity of π_i on F_D it follows that $(d_i, *) \notin G_i'$, and hence $\pi(G_i') \neq \pi(G_i') \cup \{d_i\} = G_i \cup \{d_i\} = F_i \in P_i$; it follows that $F_P \in P$. $\pi_1(F_C) = F_P$ and $f(F_C) = F_D$ are immediate.

Y_3. $F_D \backslash G_D = \{((d_1, *), *)\}$ for some $d_1 \in E_1$, corresponding to an unsynchronised event from P_1'. Let

$$F_P := G_P \cup \{(d_1, *)\}$$
$$F_C := G_C \cup \{((d_1, *), *)\}.$$

From the injectivity of π_1 on F_D it follows that $(d_1, *) \notin G_1'$, and hence $\pi(G_1') \neq \pi(G_1') \cup \{d_1\} = G_1 \cup \{d_1\} = F_1 \in P_1$; it follows that $F_P \in P$. $\pi_1(F_C) = F_P$ and $f(F_C) = F_D$ are immediate.

Y_4. Symmetrical to Y_3. ∎

A.7. Lemma. $F_C \surd_C \Leftrightarrow f(F_C) \surd_D$.

Proof. Let $F_D := f(F_C)$. The following statements are equivalent:

- $F_C \surd_C$;
- $\pi_1(F_C) \surd_P$ and for all $((d_1, d_2), e) \in F_C$, if $e \neq *$ then $F_C(d_1, d_2) \surd_Q$ (by Definition 2.6);
- for $i = 1, 2$, $\pi_1(\pi_i(F_D)) \surd_i$ (by Definition 2.5 and because $\pi_1(\pi_i(F_D)) = \pi_i(\pi_1(F_C))$) and for all $(d_i, e) \in \pi_i(F_D)$, if $e \neq *$ then $\pi_i(F_D)(d_i) \surd_Q$ (because $F_C(d_1, d_2) = \pi_i(F_D)(d_i)$);
- for $i = 1, 2$, $\pi_i(F_D) \surd_i'$ (by Definition 2.6);
- $F_D \surd_D$ (by Definition 2.5). ∎

A.8. Lemma. *Each of the following conditions D1–D3 below is sufficient to construct a configuration in D which is not in $f(C)$.*

D1. *a is executed in $P_1 \|_A P_2$ and Q is nondeterministic*;

D2. *a is not two-way sequential in $P_1 \|_A P_2$ and Q is not distinct*;

D3. *a is autoconcurrent in $P_1 \|_A P_2$ and Q is not atomic.*

Proof. Some general definitions first. Assume configurations $G_P, F_P \in P$ such that G_P contains no a-labelled event and $G_P \xrightarrow{a} F_P$; let $F_P \backslash G_P := \{(d_1, d_2)\}$. Furthermore let $G_Q \in Q$ be nonempty and let

$$H_C := (G_P \times \{*\}) \cup (\{(d_1, d_2)\} \times G_Q)$$
$$H_D := \{(e_1', e_2') \mid e_i' \in (\pi_i(G_P) \times \{*\}) \cup (\{d_i\} \times G_Q)\} \quad (= f(H_C))$$
$$H_i' := (\pi_i(G_P) \times \{*\}) \cup (\{d_i\} \times G_Q) \quad (= \pi_i(H_D))$$
$$H_i := \pi_i(G_P) \cup \{d_i\} \quad (= \pi_i(H_i')).$$

The following argument then shows $H_C \in C$. First we show that the events of H_C are elements of X. On the one hand, $\ell_P(e) \neq a$ for all $e \in G_P$, which implies $(G_P \times \{*\}) \subseteq \bigcup_{2 \leqslant j \leqslant 4} X_j$. On the other hand, $(\{(d_1, d_2)\} \times G_Q) \subseteq X_1$ because $\ell_P(d_1, d_2) = a$. Now we show the other conditions of Definition 2.6. $\pi_1(H_C) = G_P \in P$ if $G_Q = \varnothing$; otherwise $\pi_1(H_C) = G_P \cup \{d\} = F_P \in C$. If $((d_1', d_2'), e) \in H_C$ where $e \neq *$ then $(d_1', d_2') = (d_1, d_2)$, hence $H_C((d_1', d_2')) = G_Q \in Q$ and $F_P \subseteq \{(d_1', d_2')\} = G_P \in P$. It follows that $H_C \in C$, and hence $H_D \in D$, $H_i' \in P_i'$ and $H - i \in P_i$.

D1. Assume that G_P contains no a-labelled events, and that G_Q is such that there exist $e_1 \neq e_2$ with $\ell_Q(e_1) = \ell_Q(e_2) = b$ such that $G_Q \xrightarrow{b}_Q G_Q \cup \{e_1\}$ and $G_Q \xrightarrow{b}_Q G_Q \cup \{e_2\}$. Furthermore let

$$K_D := H_D \cup \{((d_1, e_1), (d_2, e_2))\}$$
$$K_i' := \pi_i(K_D) \quad (= H_i' \cup \{(d_i, e_i)\})$$
$$K_i := \pi_1(K_i') \quad (= \pi_i(F_P)).$$

We show $K_D \in D$. $(d_i, e_i) \notin H_i'$ for $i = 1, 2$ because $e_i \notin G_Q$. Furthermore, $K_i' \in P_i'$ because $\pi_1(K_i') = K_i = \pi_i(F_P) \in P_i$, and if $(d_i', e_i') \in K_i'$ such that $e_i' \neq *$ then $d_i' = d_i$ and $K_i'(d_i') = G_Q \cup \{e_i\} \in Q$; moreover $K_i \setminus \{d_i'\} = \pi_i(G_P) \in P_i$. It follows that $((d_1, e_1), (d_2, e_2)) \in E_D \setminus f(E_C)$.

D2. Assume that G_P is minimal such that (i) a is executed in P at G_P and (ii) a is not auto-sequential in C_i at $G_i := \pi_i(G_P)$, where either $i = 1$ or symmetrically $i = 2$. Assume $i = 1$. It follows that there exists a $d_1' \in E_1$ such that $d_1' \neq d_1$ and $G_1 \xrightarrow{a}_1 G_1 \cup \{d_1'\}$ and $G_1 \cup \{d_1, d_1'\} \in P_1$. Furthermore assume that G_Q is such that $\varnothing \xrightarrow{a}_Q \{e\}$ and $\varnothing \neq G_Q \xrightarrow{a}_Q G_Q \cup \{e'\}$. Finally, let

$$K_D := H_D \cup \{((d_1', e), (d_2, e'))\}$$
$$K_i' := \pi_i(K_D)$$
$$K_i := \pi_1(K_i')$$

We show $K_D \in D$. On the one hand, $d_1' \notin G_1$ and $d_1' \neq d_1$, hence $d_1' \notin H_1$, which implies $(d_1', e) \notin H_1'$; moreover $\pi_1(K_1') = G_1 \cup \{d_1, d_1'\} \in P_1$ and if $(d_1'', e) \in K_1'$ such that $e \neq *$ then either $d_1'' = d_1$, in which case $K_1'(d_1'') = G_Q \in Q$ and $K_1 \setminus \{d_1''\} = G_1 \cup \{d_1'\} \in P_1$, or $d_1'' = d_1'$, in which case $K_1'(d_1'') = \{e\} \in Q$ and $K_1 \setminus \{d_1''\} = G_1 \cup \{d_1\} \in P_1$. On the other hand, $(d_2, e') \notin H_2'$ because $e' \notin G_Q$; moreover $\pi_1(K_2') = K_2 \in P_2$, and if $(d_2', e'') \in K_2'$ such that $e'' \neq *$ then $d_2' = d_2$ and $K_2'(d_2) = G_Q \cup \{e'\} \in Q$ and $K_2 \setminus \{d_2'\} = \pi_2(G_P) \in P_2$.

At the same time, if $K_D \in f(C)$ then $e = e'$ and $K_C := f^{-1}(K_D) \in C$; then $K_P := \pi_1(K_C) \in P$ contains both (d_1, d_2) and (d_1', d_2), hence π_1 is not injective on K_P, which contradicts $K_P \in P$. It follows that $K_D \in D \setminus f(C)$.

D3. Let $G_P, G_P', G_P'' \in P$ be minimal such that $G_P \xrightarrow{a}_1 G_P'$, $F \xrightarrow{a}_1 G_P'' \neq G_P'$ and $G_P' \cup G_P'' \in P$; then $G_P' = G_P \cup \{(d_1, d_2)\}$ and $G_P'' = G_P \cup \{(d_1', d_2')\}$ where $(d_1, d_2) \neq (d_1', d_2')$; because π_i is injective on $G_P' \cup G_P''$ this implies $d_i \neq d_i'$ for both $i = 1, 2$. Assume that G_Q is such that $\varnothing \neq G_Q \xrightarrow{a}_Q G_Q \cup \{e\}$. Furthermore let

$$K_D := H_D \cup \{((d_1', e), (d_2, e))\}$$
$$K_i' := \pi_i(K_D)$$
$$K_i := \pi_1(K_i')$$

From here one proceeds in exactly the same manner as for D2, proving $K_D \in D \setminus F(C)$. ∎

A.3. *Proofs of the Syntactic Characterisation*

6.5. PROPOSITION. *Let $P \in \Sigma$ and $a \in Act$.*

1. *If a is executed in P then $a \in L(P)$;*
2. *a is initial in P if and only if $a \in I(P)$;*
3. *If a is auto-concurrent in P then $a \in D(P)$.*

Proof. Let $\langle C_P, \sqrt{}_P, \ell_P \rangle := [\![P]\!]$.

1. Trivial.
2. The property can be reformulated as follows:

$$I(P) = \{\ell_P(e) \mid \{e\} \in C_P\}.$$

The proof is straightforward by induction on the structure of P. We show the case for refinement. Assume $P = P_1[a \rightsquigarrow Q]$.

$$\begin{aligned}\{e \mid \{e\} \in C_P\} &= \{(e, *) \mid \{e\} \in C_{P_1} \wedge \ell_P(e) \neq a\} \\ &\quad \cup \{(e, d) \mid \{e\} \in C_{P_1} \wedge \ell_{P_1}(e) = a \wedge \{d\} \in C_Q\}\end{aligned}$$

and hence

$$\begin{aligned}\{\ell(e) \mid \{e\} \in C_P\} &= \{\ell_{P_1}(e) \mid \{e\} \in C_{P_1} \wedge \ell_{P_1}(e) \neq a\} \\ &\quad \cup \{\ell_Q(d) \mid \{e\} \in C_{P_1} \wedge \ell_{P_1}(e) = a \wedge \{d\} \in C_Q\} \\ &= \begin{cases} (I(P_1) \setminus \{a\}) \cup I(Q) & \text{if } a \in I(P_1) \\ (I(P_1) \setminus \{a\}) & \text{otherwise} \end{cases} \\ &= I(P_1[a \rightsquigarrow Q]).\end{aligned}$$

3. By induction on the structure of P. We show the cases of synchronisation and refinement.

Synchronisation. Assume $P = P_1 \|_A P_2$ and let $F \in C_P$ and $e_1 \neq e_2$ be such that $F \xrightarrow{a}_P F \cup \{e_1\}$ and $F \xrightarrow{a}_P F \cup \{e_2\}$ and $F \cup \{e_1, e_2\} \in C_P$. e_1 and e_2 are of the form (d_1, d_2) where $d_i = *$ implies $d_{3-i} \neq *$. If $\pi_i(e_1) \neq * \neq \pi_i(e_2)$ for some $i = 1, 2$ then $\pi_i(e_1) \neq \pi_i(e_2)$, $\pi_i(F \cup \{e_1, e_2\}) \in C_{P_i}$ and $\pi_i(F) \xrightarrow{a}_{P_i} \pi_i(F \cup \{e_j\})$ for both $j = 1, 2$; hence a is auto-concurrent in P_i and hence $a \in D(P_i)$ by induction.

Now if $a \in A$ then $\pi_i(e_j) \neq *$ for all $i, j \in \{1, 2\}$; therefore

$$a \in D(P_1) \cap D(P_2) \cap A.$$

On the other hand, if $a \notin A$ then either for some i, $\pi_i(e_1) = \pi_i(e_2) = *$, and hence

$$a \in D(P_{3-i}) \backslash A,$$

or $\pi_1(e_j) = * = \pi_2(e_{3-j})$ for some $j \in \{1, 2\}$, hence

$$a \in (L(P_1) \cap L(P_2)) \backslash A.$$

In each case $a \in D(P)$.

Refinement. Assume $P = P_1[a \rightsquigarrow Q]$ and let $F \in C_P$ and $e_1 \neq e_2$ be such that $F \xrightarrow{b}_P F \cup \{e_i\}$ for both $i = 1, 2$ and $F \cup \{e_1, e_2\} \in C_P$.

If $b \in L(Q)$, then e_1 and e_2 are events obtained by refinement; hence $\ell_{P_1}(\pi_1(e_i)) = a$ for both $i = 1, 2$, implying $a \in L(P_1)$. Either $\pi_1(e_1) = \pi_1(e_2)$, in which case $\pi_2(e_1) \neq \pi_2(e_2)$, implying that b is auto-concurrent in Q and hence $b \in D(Q)$ by induction, or $\pi_1(e_1) \neq \pi_1(e_2)$, implying that a is auto-concurrent in P_1 and hence $a \in D(P_1)$ by induction. Otherwise $b \in L(P_1)$, implying $\pi_2(e_1) = \pi_2(e_2) = *$; hence b is auto-concurrent in P_1, implying $b \in D(P_1)$ by induction. In each case $b \in D(P_1[a \rightsquigarrow Q])$. ∎

A.9. LEMMA. *Let $P, Q \in \Sigma_{flat}$ and $a \in Act$.*

$$I(P\{Q/a\}) = \begin{cases} (I(P) \backslash \{a\}) \cup I(Q) & \text{if } a \in I(P) \\ I(P) & \text{otherwise} \end{cases}$$

$$D(P\{Q/a\}) = \begin{cases} (D(P) \backslash \{a\}) \cup L(Q) & \text{if } a \in D(P) \\ D(P) \cup D(Q) & \text{if } a \in L(P) \backslash D(P) \\ D(P) & \text{otherwise} \end{cases}$$

$$SD(P\{Q/a\}) = \begin{cases} (SD(P) \backslash \{a\}) \cup L(Q) & \text{if } a \in SD(P) \\ SD(P) \cup D(Q) & \text{if } a \in S(P) \backslash SD(P) \\ SD(P) \cup SD(Q) & \text{if } a \in L(P) \backslash S(P) \\ SD(P) & \text{otherwise.} \end{cases}$$

Proof. By induction on the structure of P. We only prove the case of synchronisation for I, D and SD; the other cases are analogous. Assume that the lemma holds for P_1 and P_2 and let $P = P_1 \|_A P_2$. First assume $a \in A$; then $P\{Q/a\} = P_1\{Q/a\} \|_{(A \backslash \{a\}) \cup L(Q)} P_2\{Q/a\}$.

We denote $A' := (A \backslash \{a\}) \cup L(Q)$; also, we write I_P for $I(P)$ etc. to improve the readability by keeping the number of brackets down. The first equality is by definition of I, the second one by induction, and the last two by set-theoretic rewriting.

$$\begin{aligned} I(P\{Q/a\}) &= ((I_{P_1\{Q/a\}} \cup I_{P_2\{Q/a\}}) \backslash A') \cup (I_{P_1\{Q/a\}} \cap I_{P_2\{Q/a\}} \cap A') \\ &= \begin{cases} (((((I_{P_2} \backslash \{a\}) \cup I_Q) \cup ((I_{P_2} \backslash \{a\}) \cup I_Q)) \backslash A') & \\ \quad \cup (((I_{P_1} \backslash \{a\}) \cup I_Q) \cap ((I_{P_2} \backslash \{a\}) \cup I_Q) \cap A') & \text{if } a \in I_{P_1} \cap I_{P_2} \\ ((I_{P_1} \cup ((I_{P_2} \backslash \{a\}) \cup I_Q)) \backslash A') \cup (I_{P_1} \cap ((I_{P_2} \backslash \{a\}) \cup I_Q) \cap A') & \text{if } a \in I_{P_2} \backslash I_{P_1} \\ ((((I_{P_1} \backslash \{a\}) \cup I_Q) \cup I_{P_2}) \backslash A') \cup (((I_{P_1} \backslash \{a\}) \cup I_Q) \cap I_{P_2} \cap A') & \text{if } a \in I_{P_1} \backslash I_{P_2} \\ ((I_{P_1} \cup I_{P_2}) \backslash A') \cup (I_{P_1} \cap I_{P_2} \cap A') & \text{otherwise} \end{cases} \\ &= \begin{cases} ((I_{P_1} \cup I_{P_2}) \backslash A) \cup (I_{P_1} \cap I_{P_2} \cap (A \backslash \{a\})) \cup I_Q & \text{if } a \in I_{P_1} \cap I_{P_2} \\ ((I_{P_1} \cup I_{P_2}) \backslash A) \cup (I_{P_1} \cap I_{P_2} \cap A) & \text{otherwise} \end{cases} \\ &= \begin{cases} (I(P) \backslash \{a\}) \cup I(Q) & \text{if } a \in I(P) \cap A \\ I(P) & \text{if } a \in A \backslash I(P). \end{cases} \end{aligned}$$

In the derivations below we join the second and third steps to save space.

$$\begin{aligned} D(P\{Q/a\}) &= (D_{P_1\{Q/a\}} \cup D_{P_2\{Q/a\}} \cup (L_{P_1\{Q/a\}} \cap L_{P_2\{Q/a\}})) \backslash A' \cup (D_{P_1\{Q/a\}} \cap D_{P_2\{Q_a\}} \cap A') \\ &= \begin{cases} ((D_{P_1} \cup D_{P_2} \cup (L_{P_1} \cap L_{P_2}) \backslash A) \cup (D_{P_1} \cap D_{P_2} \cap (A \backslash \{a\})) \cup L_Q & \text{if } a \in D_{P_1} \cap D_{P_2} \\ ((D_{P_1} \cup D_{P_2} \cup (L_{P_1} \cap L_{P_2}) \backslash A) \cup (D_{P_1} \cap D_{P_2} \cap (A \backslash \{a\})) \cup D_Q & \text{if } a \in (L_{P_1} \backslash D_{P_1}) \cap (L_{P_2} \backslash D_{P_2}) \\ ((D_{P_1} \cup D_{P_2} \cup (L_{P_1} \cap L_{P_2}) \backslash A) \cup (D_{P_1} \cap D_{P_2} \cap A) & \text{otherwise} \end{cases} \end{aligned}$$

$$= \begin{cases} (D(P)\backslash\{a\}) \cup L(Q) & \text{if} \quad a \in D(P) \cap A \\ D(P) \cup D(Q) & \text{if} \quad a \in (L(P) \cap A)\backslash D(P) \\ D(P) & \text{if} \quad a \in A\backslash L(P). \end{cases}$$

$$SD(P\{Q/a\}) = SD_{P_1\{Q/a\}} \cup SD_{P_2\{Q/a\}} \cup ((D_{P_1\{Q/a\}} \cup D_{P_2\{Q/a\}}) \cap A')$$

$$= \begin{cases} ((SD_{P_1} \cup SD_{P_2})\backslash\{a\}) \cup ((D_{P_1} \cup D_{P_2}) \cap (A\backslash\{a\})) \cup L_Q & \text{if} \quad a \in S_{P_1} \cup S_{P_2} \cup D_{P_1} \cup D_{P_2} \\ (SD_{P_1} \cup SD_{P_2}) \cup ((D_{P_1} \cup D_{P_2}) \cap A) \cup D_Q & \text{if} \quad a \in (S_{P_1}\backslash D_{P_1}) \cup (S_{P_2}\backslash D_{P_2}) \\ (SD_{P_1} \cup SD_{P_2}) \cup ((D_{P_1} \cup D_{P_2}) \cap A) \cup SD_Q & \text{if} \quad a \in (L_{P_1}\backslash S_{P_1}) \cup (L_{P_2}\backslash S_{P_2}) \\ (SD_{P_1} \cup SD_{P_2}) \cup ((D_{P_1} \cup D_{P_2}) \cap A) & \text{otherwise} \end{cases}$$

$$= \begin{cases} (SD(P)\backslash\{a\}) \cup L(Q) & \text{if} \quad a \in SD(P) \cap A \\ SD(P) \cup D(Q) & \text{if} \quad a \in (S(P) \cap A)\backslash SD(P) \\ SD(P) \cup SD(Q) & \text{if} \quad a \in (L(P) \cap A)\backslash S(P) \\ SD(P) & \text{if} \quad a \in A\backslash L(P). \end{cases}$$

Now assume $a \notin A$; then $P\{Q/a\} = P_1\{Q/a\} \parallel_A P_2\{Q/a\}$.

$$I(P\{Q/a\}) = ((I_{P_1\{Q/a\}} \cup I_{P_2\{Q/a\}})\backslash A) \cup (I_{P_1\{Q/a\}} \cap I_{P_2\{Q/a\}} \cap A)$$

$$= \begin{cases} ((I_{P_1} \cup I_{P_2})\backslash A) \cup I_Q \cup ((I_{P_1} \cap I_{P_2} \cap A) & \text{if} \quad a \in I_{P_1} \cup I_{P_2} \\ ((I_{P_1} \cup I_{P_2})\backslash A) \cup (I_{P_1} \cap I_{P_2} \cap A) & \text{otherwise} \end{cases}$$

$$= \begin{cases} (I(P)\backslash\{a\}) \cup I(Q) & \text{if} \quad a \in I(P)\backslash A \\ I(P) & \text{if} \quad a \notin I(P) \cup A. \end{cases}$$

$$D(P\{Q/a\}) = (D_{P_1\{Q/a\}} \cup D_{P_2\{Q/a\}} \cup (L_{P_1\{Q/a\}} \cap L_{P_2\{Q/a\}}))\backslash A \cup (D_{P_1\{Q/a\}} \cap D_{P_2\{Q/a\}} \cap A)$$

$$= \begin{cases} (((D_{P_1} \cup D_{P_2} \cup (L_{P_1} \cap L_{P_2}))\backslash A) \cup (D_{P_1} \cap D_{P_2} \cap A))\backslash\{a\} \cup L_Q \\ \qquad \text{if} \quad a \in D_{P_1} \cup D_{P_2} \cup (L_{P_1} \cap L_{P_2}) \\ ((D_{P_1} \cup D_{P_2} \cup (L_{P_2} \cap L_{P_2}))\backslash A) \cup D_Q \cup (D_{P_1} \cap D_{P_2} \cap A) \\ \qquad \text{if} \quad a \in (L_{P_1}\backslash(D_{P_1} \cup L_{P_2})) \cup (L_{P_2}\backslash(D_{P_2} \cup L_{P_1})) \\ ((D_{P_1} \cup D_{P_2} \cup (L_{P_1} \cap L_{P_2}))\backslash A) \cup (D_{P_1} \cap D_{P_2} \cap A) \\ \qquad \text{otherwise} \end{cases}$$

$$= \begin{cases} (D(P)\backslash\{a\}) \cup L(Q) & \text{if} \quad a \in D(P)\backslash A \\ D(P) \cup D(Q) & \text{if} \quad a \in L(P)\backslash(D(P) \cup A) \\ D(P) & \text{if} \quad a \notin L(P) \cup A. \end{cases}$$

$$SD(P\{Q/a\}) = SD_{P_1\{Q/a\}} \cup SD_{P_2\{Q/a\}} \cup ((D_{P_1\{Q/a\}} \cup D_{P_2\{Q/a\}}) \cap A)$$

$$= \begin{cases} ((SD_{P_1} \cup SD_{P_1})\backslash\{a\}) \cup L_Q \cup ((D_{P_1} \cup D_{P_2}) \cap (A\backslash\{a\})) & \text{if} \quad a \in S_{P_1} \cup S_{P_2} \\ (SD_{P_1} \cup SD_{P_2}) \cup D_Q \cup ((D_{P_1} \cup D_{P_2}) \cap A) & \text{if} \quad a \in (S_{P_1}\backslash SD_{P_1}) \cup (S_{P_2}\backslash SD_{P_2}) \\ (SD_{P_1} \cup SD_{P_2}) \cup SD_Q \cup ((D_{P_1} \cup D_{P_2}) \cap A) & \text{if} \quad a \in (L_{P_1}\backslash S_{P_1}) \cup (L_{P_2}\backslash S_{P_2}) \\ (SD_{P_1} \cup SD_{P_2}) \cup ((D_{P_1} \cup D_{P_2}) \cap A) & \text{otherwise} \end{cases}$$

$$= \begin{cases} (SD(P)\backslash\{a\}) \cup L(Q) & \text{if} \quad a \in SD(P)\backslash A \\ SD(P) \cup D(Q) & \text{if} \quad a \in S(P)\backslash(SD(P) \cup A) \\ SD(P) \cup SD(Q) & \text{if} \quad a \in L(P)\backslash(S(P) \cup A) \\ SD(P) & \text{if} \quad a \notin A \cup L(P). \end{cases}$$

These two cases together imply the thesis. ∎

6.7. PROPOSITION. 1. *If* $\vdash_{det} P$ *then* P *is deterministic*;

2. *If* $P = \sum A$ *then* P *is atomic.*

Proof. 1. By induction on the structure of P. Let $\langle C_P, \surd_P, \ell_P \rangle := [\![P]\!]$. Assume that $\vdash_{det} P$ and P is not deterministic; we will show that this leads to contradictions in each case. There is a configuration $F \in C_P$ and events $e_1 \neq e_2$ such that $F \xrightarrow{a} F \cup \{e_i\}$ for both $i = 1, 2$ for some action $a = \ell_P(e_1) = \ell_P(e_2)$.

Actions. Assume $P = a$. There can be no such $e_1 \neq e_2$; contradiction.

Choice. Assume $P = P_1 + P_2$. It follows that $I(P_1) \cap I(P_2) = \varnothing$ and by induction that P_1 and P_2 are deterministic. If $e_1, e_2 \in E_{P_i}$ for some $i \in \{1, 2\}$ then $F \in C_{P_i}$; hence P_i is nondeterministic; contradiction. If $e_1 \in C_{P_i}$ and $e_2 \in C_{P_{3-i}}$ then $F = \varnothing$ and $a \in I(P_1) \cap I(P_2)$ according to Proposition 6.5; contradiction.

Sequential composition. Assume $P = P_1 ; P_2$. It follows by induction that P_1 and P_2 are deterministic. If $F \in C_{P_1}$ and $\neg F \surd_{P_1}$ then F, $F \cup \{e_i\} \in C_{P_1}$ for $i = 1, 2$, contradicting the determinism of P_1; otherwise

$$F \backslash E_{P_1} \xrightarrow{a}_{P_2} (F \cup \{e_i\}) \backslash E_{P_1},$$

contradicting the determinism of P_2.

Synchronisation. Assume $P_1 \|_A P_2$. It follows that $L(P) \cap L(Q) \subseteq A$ and by induction that P_1 and P_2 are deterministic. If $a \in A$ then $\pi_i(F) \xrightarrow{a}_{P_i} \pi_i(F \cup \{e_j\})$ for all $i, j \in \{1, 2\}$; contradiction. Otherwise either $a \in L(P_1)$ or $a \in L(P_2)$; assume $a \in L(P_1)$. It follows that $\pi_2(e_1) = \pi_2(e_2) = *$ and $\pi_1(F) \xrightarrow{a}_{P_1} \pi_1(F \cup \{e_i\})$ for both $i = 1, 2$, contradicting the determinism of P_1.

Refinement. Assume $P = P_1[a \rightsquigarrow Q]$. There are two cases.

- $a \notin L(P_1)$ and P_1 is deterministic. It follows that $\pi_2(e_1) = \pi_2(e_2) = *$, hence $\pi_1(e_1) \neq \pi_2(e_2)$ and $\pi_1(F) \xrightarrow{a}_{P_1} \pi_1(F \cup \{e_i\})$ for both $i = 1, 2$. This contradicts the determinism of P_1.
- P_1 and Q are deterministic. If $a \in L(P)$ then proceed as above. Otherwise $a \in L(Q)$ and $\ell_{P_1}(\pi_1(e_i)) = a$ for both $i = 1, 2$. If $\pi_1(e_1) \neq \pi_1(e_2)$ then proceed as above. Otherwise $\pi_2(e_1) \neq \pi_2(e_2)$ and $\pi_2(F) \xrightarrow{a}_{Q} \pi_2(F \cup \{e_i\})$ for both $i = 1, 2$; this contradicts the determinism of Q.

2. Immediate. ∎

A.10. LEMMA. *Let* $P, q \in \Sigma_{flat}$ *and* $a \in Act$.

1. $\vdash_{det} P\{Q/a\}$ *if* $\vdash_{det} P$ *and either* $a \notin L(P)$ *or* $\vdash_{det} Q$.

2. $\vdash_{red} P\{Q/a\}$ *if* $\vdash_{red} P$, Q *and in addition one of the following holds*:

   (a) $a \notin S(P)$;

   (b) $a \notin SD(P)$ *and* $\vdash_{det} Q$;

   (c) $Q = \sum A$.

Proof. Each of the statements is proved by induction on the structure of P. Note that we do not need the case for refinement, since P is assumed to be flat.

1. Immediate if $\vdash_{det} Q$. If $a \notin L(P)$ then $a \notin L(P_i)$ for both $i = 1, 2$ whenever $P = P_1 * P_2$ where $* \in \{+, ;, \|_A\}$.

2. We show only the case where $P = P_1 * P_2$, where $* \in \{+, ;, \|_1\}$ and $a \notin A$. It follows that $\vdash_{red} P_i$ for both $i = 1, 2$ and $P\{Q/a\} = P_1\{Q/a\} * P_2\{Q/a\}$.

   (a) It follows that $a \notin S(P_i)$ and hence by induction, $\vdash_{red} P_i\{red(Q)/a\}$ for both $i = 1, 2$.

   (b) It follows that $a \notin SD(P_i)$ and hence by induction, $\vdash_{red} P_i\{red(Q)/a\}$ for both $i = 1, 2$.

   (c) By induction, $\vdash_{red} P_i\{red(Q)/a\}$ for both $i = 1, 2$.

In each of these cases we can conclude $\vdash_{red} P\{Q/a\}$ according to Table 5. ∎

Received September 1994; final manuscript received January 4, 1996

REFERENCES

1. Aceto, L., and Hennessy, M. (1993), Towards action-refinement in process algebras, *Inform. and Comput.* **103**, 204–269.
2. Aceto, L., and Hennessy, M. (1994), Adding action refinement to a finite process algebra, *Inform. and Comput.* **115**(2), 179–247.
3. Best, E., Devillers, R., and Esparza, J., (1993), General refinement and recursion operators for the Petri box calculus, *in* "STACS 93" (P. Enjalbert, A. Finkel, and K. W. Wagner, Eds.), Lecture Notes in Computer Science, Vol. 665, pp. 130–140, Springer-Verlag, Berlin/New York.
4. de Bakker, J. W., de Roever, W.-P., and Rozenberg, G. (Eds.) (1989), "Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency," Lecture Notes in Computer Science, Vol. 354, Springer-Verlag, Berlin/New York.
5. Boudol, G. (1989), Atomic actions, *Bull. European Assoc. Theoret. Comput. Sci.* **38**, 136–144.
6. Boudol, G., and Castellani, I. (1994), Flow models of distributed computations: Three equivalent semantics for CCS, *Inform. and Comput.* **114**(2), 247–314.
7. Boudol, G., Castellani, I., Hennessy, M., and Kiehn, A. (1993), Observing localities, *Theoret. Comput. Sci.* **114**, 31–61.
8. Brinksma, E. (1992), On the uniqueness of fixpoints modulo observation congruence, *in* "Concur '92" (W. R. Cleaveland, Ed.), Lecture Notes in Computer Science, Vol. 630, pp. 62–76, Springer-Verlag, Berlin/New York.
9. Busi, N., van Glabbeek, R., and Gorrieri, R. (1994), Axiomatising ST bisimulation equivalence, *in* "Programming Concepts Methods and Calculi," IFIP Transactions, Vol. A-56, pp. 169–188, Elsevier, Amsterdam.

10. Castellano, L., De Michelis, G., and Pomello, L. (1987), Concurrency vs. interleaving: An instructive example, *Bull. European Assoc. Theoret. Comput. Sci.* **31**, 12–15, Note.
11. Cleaveland, W. R. (Ed.) (1992), "Concur '92," Lecture Notes in Computer Science, Vol. 630, Springer-Verlag, Berlin/New York.
12. Costantini, R. (1994), "Abstraktion in ereignisbasierten Modellen verteilter Systeme," Ph.D. thesis, Institut für Informatik, Universität Hildesheim.
13. Darondeau, P., and Degano, P. (1993), Refinement of actions in event structures and causal trees, *Theoret. Comput. Sci.* **118**, 21–48.
14. Degano, P., and Gorrieri, R. (1991), Atomic refinement for process description languages, *in* "Mathematical Foundations of Computer Science 1991" (A. Tarlecki, Ed.), Lecture Notes in Computer Science, Vol. 520, pp. 121–130, Springer-Verlag.
15. Degano, P., and Gorrieri, R. (1992), "An Operational Definition of Action Refinement," Technical report TR-28/92, Università di Pisa.
16. Degano, P., and Gorrieri, R. (1995), A causal operational semantics of action refinement, *Inform. and Comput.* **122**(1), 97–119.
17. van Glabbeek, R., and Goltz, U. (1989), Equivalence notions for concurrent systems and refinement of actions, *in* "Mathematical Foundations of Computer Science 1989" (A. Kreczmar and G. Mirkowska, Eds.), Lecture Notes in Computer Science, Vol. 379, pp. 237–248, Springer-Verlag, Berlin/New York.
18. van Glabbeek, R., and Goltz, U. (1990), Refinement of actions in causality based models, *in* "Stepwise Refinement of Distributed Systems—Models, Formalisms, Correctness" (J. W. de Bakker, W.-P. de Roever and G. Rozenberg, Eds.), Lecture Notes in Computer Science, Vol. 430, pp. 267–300, Springer-Verlag, Berlin/New York.
19. Goltz, U., Gorrieri, R., and Rensink, A. (1992), "On Syntactic and Semantic Action Refinement," Hildesheimer Informatik-Berichte 17/92, Institut für Informatik, Universität Hildesheim.
20. Gorrieri, R. (1992), A hierarchy of system descriptions via atomic linear refinement, *Fundam. Inform.* **16**, 289–336.
21. Gorrieri, R., Marchetti, S., and Montanari, U. (1990), A^2CSS: Atomic actions for CCS, *Theoret. Comput. Sci.* **72**, 203–223.
22. Jategaonkar, L., and Meyer, A. (1992), Testing equivalences for Petri nets with action refinement, *in* "Concur '92" (W. R. Cleaveland, Ed.), Lecture Notes in Computer Science, Vol. 630, pp. 17–31, Springer-Verlag, Berlin/New York.
23. Jategaonkar, L., and Meyer, A. (1993), Self-synchronization of concurrent processes, *in* "Automata, Languages and Programming" (A. Lingas, R. Karlsson, and S. Carlsson, Eds.), Lecture Notes in Computer Science, Vol. 700, pp. 519–531, Springer-Verlag, Berlin/New York.
24. Milner, R. (1989), "Communication and Concurrency," Prentice–Hall, Englewood Cliffs, NJ.
25. Nielsen, M., Engberg, U., and Larsen, K. G. (1993), Fully abstract models for a process language with refinement, *in* "Linear Time, Branching Time and Partial Order in Logic and Models for Concurrency" (J. W. de Bakker, W.-P. de Roever, and R. Rozenberg, Eds.), Lecture Notes in Computer Science, Vol. 354, pp. 523–549, Springer-Verlag, Berlin/New York.
26. Olderog, E.-R. (Ed.), (1994), "Programming Concepts, Methods and Calculi," Vol. A-56, IFIP Transactions, IFIP, Elsevier, Amsterdam.
27. Rabinovich, A., and Trakhtenbrot, B. A. (1988), Behaviour structure and nets, *Fundam. Inform.* **XI**(4), 357–404.
28. Rensink, A. (1994), Methodological aspects of action refinement, *in* "Programming Concepts, Methods and Calculi" (E.-R. Olderog, Ed.), Vol. A-56, IFIP Transactions, IFIP.
29. Rensink, A. (1995), An event-based SOS for a language with refinement, *in* "Structures in Concurrency Theory" (J. Desel, Ed.), Workshops in Computing, pp. 294–309, Springer-Verlag, London.
30. Vogler, W. (1990), Failures semantics based on interval semiwords is a congruence for refinement, *in* "STACS 90" (C. Choffrut and T. Lengauer, Eds.), Lecture Notes in Computer Science, Vol. 415, pp. 285–297, Springer-Verlag, Berlin/New York.
31. Vogler, W. (1993), Bisimulation and action refinement, *Theoret. Comput. Sci.* **114**, 173–200.
32. Wehrheim, H. (1994), Parametric action refinement, *in* "Programming Concepts, Methods and Calculi" (E.-R. Olderog, Ed.), IFIP Transactions, Vol. A-56, pp. 247–266, IFIP, Elsevier, Amsterdam.
33. Winskel, G. (1987), Event structures, *in* "Petri Nets: Applications and Relationships to Other Models of Concurrency," Lecture Notes in Computer Science, Vol. 255, pp. 325–392, Springer-Verlag, Berlin/New York.