A Lean-Congruence Format for EP-Bisimilarity

Rob van Glabbeek* ©  
Peter Höfner ©  
Weiyou Wang
School of Informatics  
University of Edinburgh, UK  
School of Computer Science and Engineering  
University of New South Wales  
Sydney, Australia  
rvg@cs.stanford.edu

School of Computing  
Australian National University  
Canberra, Australia  
peter.hoefner@anu.edu.au  
weiyou.wang@anu.edu.au

Enabling preserving bisimilarity is a refinement of strong bisimilarity that preserves safety as well as liveness properties. To define it properly, labelled transition systems needed to be upgraded with a successor relation, capturing concurrency between transitions enabled in the same state. We enrich the well-known De Simone format to handle inductive definitions of this successor relation. We then establish that ep-bisimilarity is a congruence for the operators, as well as lean congruence for recursion, for all (enriched) De Simone languages.

1 Introduction

Recently, we introduced a finer alternative to strong bisimilarity, called enabling preserving bisimilarity. The motivation behind this concept was to preserve liveness properties, which are not always preserved by classical semantic equivalences, including strong bisimilarity.

Example 1.1 ([13]) Consider the following two programs, and assume that all variables are initialised to 0.

\[
\begin{align*}
\text{while (true) do} & \\
\text{choose} & \\
\text{if true then y := y+1; } & \\
\text{if x = 0 then x := 1; } & \\
\text{end} & \\
\text{od}
\end{align*}
\]

\[
\begin{align*}
\text{while (true) do} & \\
\text{od} & \\
\text{y := y+1} & \\
\text{y := y+1}
\end{align*}
\]

\[
\begin{align*}
\text{while (true) do} & \\
\text{x := 1; } & \\
\text{y := y+1; } & \\
\text{od}
\end{align*}
\]

The code on the left-hand side presents a non-terminating while-loop offering an internal nondeterministic choice. The conditional if x = 0 then x := 1 describes an atomic read-modify-write operation.\(^1\) Since the non-deterministic choice does not guarantee to ever pick the second conditional, this example should not satisfy the liveness property ‘eventually x=1’.

The example on the right-hand side is similar, but here two different components handle the variables x and y separately. The two programs should be considered independent – by default we assume they are executed on different cores. Hence the property ‘eventually x=1’ should hold.

The two programs behave differently with regards to (some) liveness properties. However, it is easy to verify that they are strongly bisimilar, when considering the traditional modelling of such code in terms of transition systems. In fact, their associated transition systems, also displayed above, are identical. Hence, strong bisimilarity does not preserve all liveness properties.

Enabling preserving bisimilarity (ep-bisimilarity) – see next section for a formal definition – distinguishes these examples and preserves liveness. In contrast to classical bisimulations, which are relations of type States \(\times\) States, this equivalence is based on triples. An ep-bisimulation additionally maintains

\(^*\)Supported by Royal Society Wolfson Fellowship RSWF\(\backslash R1\{221008\)

\(^1\)https://en.wikipedia.org/wiki/Read-modify-write

EPTCS 387, 2023, pp. 59–75, doi:10.4204/EPTCS.387.6
© R.J. van Glabbeek, P. Höfner & W. Wang  
This work is licensed under the  
Creative Commons Attribution License.
for each pair of related states \( p \) and \( q \) a relation \( R \) between the transitions enabled in \( p \) and \( q \), and this relation should be preserved when matching related transitions in the bisimulation game. When formalising this, we need transition systems upgraded with a successor relation that matches each transition \( t \) enabled in a state \( p \) to a transition \( t' \) enabled in \( p' \), when performing a transition from \( p \) to \( p' \) that does not affect \( t \). Intuitively, \( t' \) describes the same system behaviour as \( t \), but the two transitions could be formally different as they may have different sources. It is this successor relation that distinguishes the transition systems in the example above.

In [13], we showed that ep-bisimilarity is a congruence for all operators of Milner’s Calculus of Communication Systems (CCS), enriched with a successor relation. We extended this result to the Algebra of Broadcast Communication with discards and Emissions (ABCdE), an extension of CCS with broadcast communication, discard actions and signal emission. ABCdE subsumes many standard process algebras found in the literature.

In this paper, we introduce a new congruence format for structural operational semantics, which is based on the well-known De Simone Format and respects the successor relation. This format allows us to generalise the results of [13] in two ways: first, we prove that ep-bisimilarity is a congruence for all operators of any process algebras that can be formalised in the De Simone format with successors. Applicable languages include CCS and ABCdE. Second, we show that ep-bisimilarity is a lean congruence for recursion [9]. Here, a lean congruence preserves equivalence when replacing closed subexpressions of a process by equivalent alternatives.

2 Enabling Preserving Bisimilarity

To build our abstract theory of De Simone languages and De Simone formats, we briefly recapitulate the definitions of labelled transition systems with successors, and ep-bisimulation. A detailed description can be found in [13].

A labelled transition system (LTS) is a tuple \((S, Tr, source, target, \ell)\) with \( S \) and \( Tr \) sets of states and transitions, \( source, target : Tr \to S \) and \( \ell : Tr \to \mathcal{L} \), for some set \( \mathcal{L} \) of transition labels. A transition \( t \in Tr \) of an LTS is enabled in a state \( p \in S \) if \( source(t) = p \). The set of transitions enabled in \( p \) is \( en(p) \).

**Definition 2.1 (LTSS [13])** A labelled transition system with successors (LTSS) is a tuple \((S, Tr, source, target, \ell, \sim)\) with \((S, Tr, source, target, \ell)\) an LTS and \( \sim \subseteq Tr \times Tr \times Tr \) the successor relation such that if \((t, u, v) \in \sim\) (also denoted by \( t \sim_u v \)) then \( source(t) = source(u) \) and \( target(v) = target(u) \).

**Example 2.2** Remember that the ‘classical’ LTSs of Example 1.1 are identical. Let \( t_1 \) and \( t_2 \) be the two transitions corresponding to \( y := y + 1 \) in the first and second state, respectively, and let \( u \) be the transition for assignment \( x := 1 \). The assignments of \( x \) and \( y \) in the right-hand program are independent, hence \( t_1 \sim_u t_2 \) and \( u \sim t_0 u \). For the other program, the situation is different: as the instructions correspond to a single component (program), all transitions affect each other, i.e. \( \sim = \emptyset \).

**Definition 2.3 (Ep-bisimilarity [13])** Let \((S, Tr, source, target, \ell, \sim)\) be an LTSS. An enabling preserving bisimulation (ep-bisimulation) is a relation \( R \subseteq S \times S \times \mathcal{P}(Tr \times Tr) \) satisfying

1. if \((p, q, R) \in R\) then \( R \subseteq en(p) \times en(q) \) such that
   a. \( \forall t \in en(p) \). \( \exists u \in en(q) \). \( t R u \),
   b. \( \forall u \in en(q) \). \( \exists t \in en(p) \). \( t R u \), and
   c. if \( t R u \) then \( \ell(t) = \ell(u) \); and
2. if \((p, q, R) \in R\) and \( v R w \), then \((target(v), target(w), R') \in R\) for some \( R' \) such that
   a. if \( t R u \) and \( t \sim_v t' \) then \( \exists u', u \sim_w u' \land t' R' u' \), and
   b. if \( t R u \) and \( u \sim_w u' \) then \( \exists t'. t \sim_v t' \land t' R' u' \).
Two states $p$ and $q$ in an LTSS are enabling preserving bisimilar (ep-bisimilar), denoted as $p \leftrightarrow_{ep} q$, if there is an enabling preserving bisimulation $\mathcal{R}$ such that $(p, q, R) \in \mathcal{R}$ for some $R$.

Without Items 2.a and 2.b, the above is nothing else than a reformulation of the classical definition of strong bisimilarity. An ep-bisimulation additionally maintains for each pair of related states $p$ and $q$ a relation $R$ between the transitions enabled in $p$ and $q$. Items 2.a and 2.b strengthen the condition on related target states by requiring that the successors of related transitions are again related relative to these target states. It is this requirement which distinguishes the transition systems for Example 1.1. [13]

**Lemma 2.4** [Proposition 10 of [13]] $\leftrightarrow_{ep}$ is an equivalence relation.

### 3 An Introductory Example: CCS with Successors

Before starting to introduce the concepts formally, we want to present some motivation in the form of the well-known Calculus of Communicating Systems (CCS) [17]. In this paper we use a proper recursion construct instead of agent identifiers with defining equations. As in [3], we write $\langle X | S \rangle$ for the $X$-component of a solution of the set of recursive equations $S$.

CCS is parametrised with set $\mathcal{C}$ of handshake communication names. $\mathcal{C} := \{ \bar{c} \mid c \in \mathcal{C} \}$ is the set of handshake communication co-names. $\text{Act}_{CCS} := \mathcal{C} \cup \mathcal{C} \cup \{ \tau \}$ is the set of actions, where $\tau$ is a special internal action. Complementation extends to $\mathcal{C} \cup \mathcal{C}$ by $\bar{\bar{c}} := c$.

Below, $c$ ranges over $\mathcal{C} \cup \mathcal{C}$ and $\alpha, \ell, \eta$ over $\text{Act}_{CCS}$. A relabelling is a function $f : \mathcal{C} \to \mathcal{C}$; it extends to $\text{Act}_{CCS}$ by $f(\bar{c}) = \bar{f(c)}$, $f(\tau) := \tau$.

The process signature $\Sigma$ of CCS features binary infix-written operators $+$ and $|$, denoting choice and parallel composition, a constant $0$ denoting inaction, a unary action prefixing operator $\alpha.$ for each action $\alpha \in \text{Act}_{CCS}$, a unary restriction operator $\_ \setminus L$ for each set $L \subseteq \mathcal{C}$, and a unary relabelling operator $\_ [f]$ for each relabelling $f : \mathcal{C} \to \mathcal{C}$.

The semantics of CCS is given by the set $\mathcal{R}$ of transition rules, shown in Table 1. Here $\mathcal{L} := \{ \bar{c} \mid c \in L \}$. Each rule has a unique name, displayed in blue.\(^2\) The rules are displayed as templates, following the standard convention of labelling transitions with label variables $c$, $\alpha$, $\ell$, etc. and may be accompanied by side conditions in green, so that each of those templates corresponds to a set of (concrete) transition rules where label variables are “instantiated” to labels in certain ranges and all side conditions are met. The rule names are also schematic and may contain variables. For example, all instances of the transition rule template $\rightarrow_L$ are named $\rightarrow_L$, whereas there is one rule name $\rightarrow$ for each action $\alpha \in \text{Act}_{CCS}$.

\(^2\)Our colourings are for readability only.
The transition system specification \((\Sigma, R)\) is in De Simone format [22], a special rule format that guarantees properties of the process algebra (for free), such as strong bisimulation being a congruence for all operators. Following [13], we leave out the infinite sum \(\sum_{i \in I} x_i\) of CCS [17], as it is strictly speaking not in De Simone format.

In this paper, we will extend the De Simone format to also guarantee properties for ep-bisimulation. As seen, ep-bisimulation requires that the structural operational semantics is equipped with a successor relation \(\sim\). The meaning of \(\chi \sim_\zeta \chi'\) is that transition \(\chi\) is unaffected by \(\zeta\) – denoted \(\chi \sim_\zeta \zeta\) – and that when doing \(\zeta\) instead of \(\chi\), afterwards a variant \(\chi'\) of \(\chi\) is still enabled. Table 2 shows the successor rules for CCS, which allow the relation \(\sim\) to be derived inductively. It uses the following syntax for transitions \(\chi\), which will be formally introduced in Section 6. The expression \(t \xrightarrow{1} Q\) refers to the transition that is derived by rule \(+_t\) of Table 1, with \(t\) referring to the transition used in the unique premise of this rule, and \(Q\) referring to the process in the inactive argument of the \(+\)-operator. The syntax for the other transitions is analogous. A small deviation of this scheme occurs for recursion: \(rec_{Act}(X, S, t)\) refers to the transition derived by rule \(rec_{Act}\) out of the premise \(t\), when deriving a transition of a recursive call \(\langle X|S\rangle\).

In Table 2 each rule is named, in orange, after the number of the clause of Definition 20 in [13], were it was introduced.

The primary source of concurrency between transition \(\chi\) and \(\zeta\) is when they stem from opposite sides of a parallel composition. This is expressed by Rules 7a and 7b. We require all obtained successor statements \(\chi \sim_\zeta \chi'\) to satisfy the conditions of Definition 2.1 – this yields \(Q' = \text{target}(w)\) and \(P' = \text{target}(v)\); in [13] \(Q'\) and \(P'\) were written this way.

In all other cases, successors of \(\chi\) are inherited from successors of their building blocks.

When \(\zeta\) stems from the left side of a \(+\) via rule \(+_t\) of Table 1, then any transition \(\chi\) stemming from the right is discarded by \(\zeta\), so \(\chi \not\sim\zeta\). Thus, if \(\chi \sim \zeta\) then these transitions have the form \(\chi = t \xrightarrow{1}\bar{Q}\) and \(\zeta = v \xrightarrow{1} Q\), and we must have \(t \sim v\). So \(t \sim v\) for some transition \(t'\). As the execution of \(\zeta\) discards the summand \(\bar{Q}\), we also obtain \(\chi \sim_\zeta t'\). This motivates Rule 3a. Rule 4a follows by symmetry.

In a similar way, Rule 8a covers the case that \(\chi\) and \(\zeta\) both stem from the left component of a parallel composition. It can also happen that \(\chi\) stems form the left component, whereas \(\zeta\) is a synchronisation, involving both components. Thus \(\chi = t \xrightarrow{1} Q\) and \(\zeta = v \xrightarrow{1} w\). For \(\chi \sim \zeta\) to hold, it must be that \(t \sim v\), whereas the \(w\)-part of \(\zeta\) cannot interfere with \(t\). This yields the Rule 8b. Rule 8c is explained in a similar vain from the possibility that \(\xi\) stems from the left while \(\zeta\) is a synchronisation of both components. Rule 9 follows by symmetry. In case both \(\chi\) and \(\zeta\) are synchronisations involving both components, i.e., \(\chi = t \xrightarrow{1} u\) and \(\zeta = v \xrightarrow{1} w\), it must be that \(t \sim v\) and \(u \sim w\). Now the resulting variant \(\chi'\) of \(\chi\) after \(\zeta\) is simply \(t'\sim u'd\), where \(t \sim v\) and \(u \sim w\). This underpins Rule 10.

If the common source \(O\) of \(\chi\) and \(\zeta\) has the form \(P[f], \chi\) and \(\zeta\) must have the form \(t[f]\) and \(v[f]\). Whether \(t\) and \(v\) are concurrent is not influenced by the renaming. So \(t \sim v\). The variant of \(t\) that remains after doing \(v\) is also not affected by the renaming, so if \(t \sim_\psi t'\) then \(\chi \sim_\zeta t'[f]\). The case that \(O = P\setminus L\) is equally trivial. This yields Rules 11a and 11b.

In case \(O = \langle X|S\rangle\), \(\chi\) must have the form \(rec_{Act}(X, S, t)\), and \(\zeta\) has the form \(rec_{Act}(X, S, v)\), where \(t\) and \(v\) are enabled in \(\langle X|S\rangle\). Now \(\chi \sim_\zeta \zeta\) only if \(t \sim v\), so \(t \sim_\psi t'\) for some transition \(t'\). The recursive call disappears upon executing \(\zeta\), and we obtain \(\chi \sim_\zeta t'\). This yields Rule 11c.

Example 3.1 The programs from Example 1.1 could be represented in CCS as \(P := \langle X|S\rangle\) where \(S = \begin{cases} X = aX + bY \\ Y = aX \end{cases}\) and \(Q := \langle Z|\{Z = aZ\}\rangle|b.0\). Here \(a, b \in \text{Act}_{\text{CCS}}\) are the atomic actions incrementing \(y\) and \(x\). The relation matching \(P\) with \(Q\) and \(\langle Y|S\rangle\) with \(\langle Z|\{Z = aZ\}\rangle|0\) is a strong bisimulation. Yet, \(P\) and \(Q\) are not ep-bisimilar, as the rules of Table 2 derive \(u \sim_\zeta u\) (cf. Example 2.2)
Let $\xi$, and $\zeta'$ must have a leading operator $R$ and $S$ of the same type, and the same number of arguments. These leading operators must be rule names of the same type. Their arguments are either process variables $P, Q, \ldots$ or transition variables $t, u, \ldots$, as determined by the trigger sets $I_R$ and $I_S$ of $R$ and $S$. These are the sets of indices listing the arguments for which rules $R$ and $S$ have a premise. If the $i^{th}$ arguments of $R$ and $S$ are both process variables, they must be the same, but for the rest all these variables are different. For a subset $I$ of $I_R \cap I_S$, the rule has premises $t_i \sim_{u_i} t'_i$ for $i \in I$, where $t_i$ and $u_i$ are the $i^{th}$ arguments of $R$ and $S$, and $t'_i$ is a fresh variable. Finally, the right-hand side of the conclusion may be an arbitrary univariate transition expression, containing no other variables than:

- the $t'_i$ for $i \in I$,
- a $t_i$ occurring in $\zeta'$, with $i \notin I_S$,
- a fresh process variable $P'_i$ that must match the target of the transition $u_i$ for $i \in I_S \setminus I$,
- or a fresh transition variable whose source matches the target of $u_i$ for $i \in I_S \setminus I$, and
- any $P$ occurring in both $\zeta$ and $\zeta'$, or any fresh transition variable whose source must be $P$.

The rules of Table 2 only feature the first three possibilities; the others occur in the successor relation of ABCdE – see Section 8.

### 4 Structural Operational Semantics

Both the De Simone format and our forthcoming extension are based on the syntactic form of the operational rules. In this section, we recapitulate foundational definitions needed later on. Let $\mathcal{V}_P$ be an infinite set of process variables, ranged over by $X, Y, x, y, x_i, \ldots$
Definition 4.1 (Process Expressions [8]) An operator declaration is a pair \((Op, n)\) of an operator symbol \(Op \notin \mathcal{V}_P\) and an arity \(n \in \mathbb{N}\). An operator declaration \((c, 0)\) is also called a constant declaration. A process signature is a set of operator declarations. The set \(\mathbb{P}'(\Sigma)\) of process expressions over a process signature \(\Sigma\) is defined inductively by:

- \(\mathcal{V}_P \subseteq \mathbb{P}'(\Sigma)\).
- if \((Op, n) \in \Sigma\) and \(p_1, \ldots, p_n \in \mathbb{P}'(\Sigma)\) then \(Op(p_1, \ldots, p_n) \in \mathbb{P}'(\Sigma)\), and
- if \(V_S \subseteq \mathcal{V}_P\), \(S : V_S \rightarrow \mathbb{P}'(\Sigma)\) and \(X \in V_S\), then \(\langle \langle X \rangle | S \rangle \in \mathbb{P}'(\Sigma)\).

A process expression \(c()\) is abbreviated as \(c\) and is also called a constant. An expression \(\langle X | S \rangle\) as appears in the last clause is called a recursive call, and the function \(S\) therein is called a recursive specification. It is often displayed as \(\{X = S_X \mid X \in V_S\}\). Therefore, for a recursive specification \(S\), \(V_S\) denotes the domain of \(S\) and \(S_X\) represents \(S(X)\) when \(X \in V_S\). Each expression \(S_Y\) for \(Y \in V_S\) counts as a subexpression of \(\langle X | S \rangle\). An occurrence of a process variable \(y\) in an expression \(p\) is free if it does not occur in a subexpression of the form \(\langle X | S \rangle\) with \(y \in V_S\). For an expression \(p\), \(\text{var}(p)\) denotes the set of process variables having at least one free occurrence in \(p\). An expression is closed if it contains no free occurrences of variables. Let \(\mathbb{P}'(\Sigma)\) be the set of closed process expressions over \(\Sigma\).

Definition 4.2 (Substitution) A \(\Sigma\)-substitution \(\sigma\) is a partial function from \(\mathcal{V}_P\) to \(\mathbb{P}'(\Sigma)\). It is closed if it is a total function from \(\mathcal{V}_P\) to \(\mathbb{P}'(\Sigma)\).

If \(p \in \mathbb{P}'(\Sigma)\) and \(\sigma\) a \(\Sigma\)-substitution, then \(p[\sigma]\) denotes the expression obtained from \(p\) by replacing, for \(x\) in the domain of \(\sigma\), every free occurrence of \(x\) in \(p\) by \(\sigma(x)\), while renaming bound process variables if necessary to prevent name-clashes. In that case \(p[\sigma]\) is called a substitution instance of \(p\). A substitution instance \(p[\sigma]\) where \(\sigma\) is given by \(\sigma(x_i) = q_i\) for \(i \in I\) is denoted as \(p[q_i/x_i]_{i \in I}\), and for \(S\) a recursive specification \(\langle p | S \rangle\) abbreviates \(p[\langle Y | S \rangle / Y]_{Y \in V_S}\).

These notions, including “free” and “closed”, extend to syntactic objects containing expressions, with the understanding that such an object is a substitution instance of another one if the same substitution has been applied to each of its constituent expressions.

We assume fixed but arbitrary sets \(\mathcal{L}\) and \(\mathcal{N}\) of transition labels and rule names.

Definition 4.3 (Transition System Specification [16]) Let \(\Sigma\) be a process signature. A \(\Sigma\)-(transition) literal is an expression \(p \xrightarrow{H} q\) with \(p, q \in \mathbb{P}'(\Sigma)\) and \(a \in \mathcal{L}\). A transition rule over \(\Sigma\) is an expression of the form \(\frac{H}{\lambda}\) with \(H\) a finite list of \(\Sigma\)-literals (the premises of the transition rule) and \(\lambda\) a \(\Sigma\)-literal (the conclusion). A transition system specification (TSS) is a tuple \((\Sigma, \mathcal{R}, \mathcal{N})\) with \(\mathcal{R}\) a set of transition rules over \(\Sigma\), and \(\mathcal{N} : \mathcal{R} \rightarrow \mathcal{N}\) a (not necessarily injective) rule-naming function, that provides each rule \(r \in \mathcal{R}\) with a name \(\mathcal{N}(r)\).

Definition 4.4 (Proof) Assume literals, rules, substitution instances and rule-naming. A proof of a literal \(\lambda\) from a set \(\mathcal{R}\) of rules is a well-founded, upwardly branching, ordered tree where nodes are labelled by pairs \((\mu, R)\) of a literal \(\mu\) and a rule name \(R\), such that

- the root is labelled by a pair \((\lambda, S)\), and
- if \((\mu, R)\) is the label of a node and \((\mu_1, R_1), \ldots, (\mu_n, R_n)\) is the list of labels of this node’s children then \(\mu_1 \ldots \mu_n\) is a substitution instance of a rule in \(\mathcal{R}\) with name \(R\).

Definition 4.5 (Associated LTS [12]) The associated LTS of a TSS \((\Sigma, \mathcal{R}, \mathcal{N})\) is the LTS \((S, \mathcal{Tr}, \text{source}, \text{target}, \ell)\) with \(S := \mathbb{P}'(\Sigma)\) and \(\mathcal{Tr}\) the collection of proofs \(\pi\) of closed \(\Sigma\)-literals \(p \xrightarrow{a} q\) from \(\mathcal{R}\), where \(\text{source}(\pi) = p\), \(\ell(\pi) = a\) and \(\text{target}(\pi) = q\).

Above we deviate from the standard treatment of structural operational semantics [16, 8] on four counts. Here we employ CCS to motivate those design decisions.
In Definition 4.5, the transitions $Tr$ are taken to be proofs of closed literals $p \xrightarrow{a} q$ rather than such literals themselves. This is because there can be multiple $a$-transitions from $p$ to $q$ that need to be distinguished when taking the concurrency relation between transitions into account. For example, if $p := \langle X \{X = a.X + c.X\} \rangle$ and $q := \langle Y \{Y = a.Y\} \rangle$ then $p|q$ has three outgoing transitions:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.p \xrightarrow{a} p$</td>
<td>$</td>
</tr>
<tr>
<td>$a.p + c.p \xrightarrow{a} p$</td>
<td>$</td>
</tr>
<tr>
<td>$p \xrightarrow{a} p$</td>
<td>$</td>
</tr>
<tr>
<td>$p</td>
<td>q \xrightarrow{a} p</td>
</tr>
</tbody>
</table>

The rightmost transition is concurrent with the middle one, whereas the leftmost one is not. If we were to erase the rule names, the difference between these two transitions would disappear.

In Definition 4.3 we require the premises of rules to be lists rather than sets, and accordingly in Definition 4.4 we require proof trees to be ordered. This is to distinguish transitions/proofs in which a leading operator) with different proofs. This phenomenon does not occur in CCS, but we could have illustrated it with CSP [5] or ABCdE [13].

Finally, suppose that in Definition 4.3 we had chosen the rule-naming function $N$ to be the identity. This is equivalent to not having a rule-naming function at all, instead labelling nodes in proofs with rules rather than names of rules. Then in the transition

$$\frac{a.0 \xrightarrow{a} 0}{\langle X \{X = a.0\} \rangle \xrightarrow{a} 0 \text{recAct}}$$

we should replace the generic name $\text{recAct}$ of a recursion rule with the specific rule employed. This could be the rule $\frac{a.0 \xrightarrow{a} z}{\langle X \{X = a.0\} \rangle \xrightarrow{a} z}$, but just as well the rule $\frac{y \xrightarrow{a} z}{\langle X \{X = y\} \rangle \xrightarrow{a} z}$, when employing a substitution that sends $y$ to $a.0$. To avoid the resulting unnecessary duplication of transitions, we give both recursion rules the same name.

## 5 De Simone Languages

The syntax of a De Simone language is specified by a process signature, and its semantics is given as a TSS over that process signature of a particular form [22], nowadays known as the De Simone format. Here, we extend the De Simone format to support indicator transitions, as occur in [11, 10, 13]. These are transitions $p \xrightarrow{\ell} q$ for which it is essential that $p = q$. They are used to convey a property of the state $p$ rather than model an action of $p$. To accommodate them we need a variant of the recursion rule whose conclusion again is of the form $r \xrightarrow{\ell} r$. This variant will be illustrated in Section 8.

As for $\mathcal{L}$, we fix a set $\text{Act} \subseteq \mathcal{L}$ of actions.
Definition 5.1 (De Simone Format) A TSS $(\Sigma, R, N)$ is in De Simone format if for every recursive call $(X|S)$ and every $\alpha \in Act$ and $\ell \in \mathcal{L}\setminus Act$, it has transition rules

$$\frac{(S_X|S) \xrightarrow{\alpha} y}{(X|S) \xrightarrow{\alpha} y} \text{ recAct} \quad \text{and} \quad \frac{(S_X|S) \xrightarrow{\ell} y}{(X|S) \xrightarrow{\ell} (X'|S)} \text{ recIn} \quad \text{for some } y \notin \text{var}(S_X|S),$$

and each of its other transition rules (De Simone rules) has the form

$$\frac{x_i \xrightarrow{a_j} y_i \mid i \in I}{Op(x_1, \ldots, x_n) \xrightarrow{a_j} q}$$

where $(Op, n) \in \Sigma$, $I \subseteq \{1, \ldots, n\}$, $a, a_i \in \mathcal{L}$, $x_i$ (for $1 \leq i \leq n$) and $y_i$ (for $i \in I$) are pairwise distinct process variables, and $q$ is a univariate process expression containing no other free process variables than $x_i$ (for $1 \leq i \leq n \land i \notin I$) and $y_i$ (for $i \in I$), having the properties that

- each subexpression of the form $(X|S)$ is closed, and
- if $a \in \mathcal{L}\setminus Act$ then $a_i \in \mathcal{L}\setminus Act$ ($i \in I$) and $q = Op(z_1, \ldots, z_n)$, where $z_i := \begin{cases} y_i & \text{if } i \in I \\ x_i & \text{otherwise.} \end{cases}$

Here univariate means that each variable has at most one free occurrence in it. The last clause above guarantees that for any indicator transition $t$, one with $(t) \in \mathcal{L}\setminus Act$, we have $\text{target}(t) = \text{source}(t)$. For a De Simone rule of the above form, $n$ is the arity, $(Op, n)$ is the type, $a$ is the label, $q$ is the target, $I$ is the trigger set and the tuple $(\ell_1, \ldots, \ell_n)$ with $\ell_i = a_i$ if $i \in I$ and $\ell_i = *$ otherwise, is the trigger. Transition rules in the first two clauses are called recursion rules.

We also require that if $N(r) = N(r')$ for two different De Simone rules $r, r' \in R$, then $r, r'$ have the same type, target and trigger set, but different triggers. The names of the recursion rules are as indicated in blue above, and differ from the names of any De Simone rules.

Many process description languages encountered in the literature, including CCS [17] as presented in Section 3, SCCS [18], ACP [3] and MEIJE [2], are De Simone languages.

6 Transition System Specifications with Successors

In Section 4, a process is denoted by a closed process expression; an open process expression may contain variables, which stand for as-of-yet unspecified subprocesses. Here we will do the same for transition expressions with variables. However, in this paper a transition is defined as a proof of a literal $p \rightarrow a q$ from the operational rules of a language. Elsewhere, a transition is often defined as a provable literal $p \rightarrow a q$, but here we need to distinguish transitions based on these proofs, as this influences whether two transitions are concurrent.

It turns out to be convenient to introduce an open proof of a literal as the semantic interpretation of an open transition expression. It is simply a proof in which certain subproofs are replaced by proof variables.

Definition 6.1 (Open Proof) Given definitions of literals, rules and substitution instances, and a rule-naming function $N$, an open proof of a literal $\lambda$ from a set $R$ of rules using a set $V$ of (proof) variables is a well-founded, upwardly branching, ordered tree of which the nodes are labelled either by pairs $(\mu, R)$ of a literal $\mu$ and a rule name $R$, or by pairs $(\mu, px)$ of a literal $\mu$ and a variable $px \in V$ such that

- the root is labelled by a pair $(\lambda, \mathcal{X})$, 
- if $(\mu, px)$ is the label of a node then this node has no children,
if its root is labelled by \( t \) and 

- if \( (\mu, R) \) is the label of a node and \( (\mu_1, \chi_1), \ldots, (\mu_n, \chi_n) \) is the list of labels of this node’s children

then \( \mu_1, \ldots, \mu_n \) is a substitution instance of a rule named \( R \).

Let \( \mathcal{V}_T \) be an infinite set of transition variables, disjoint from \( \mathcal{V}_P \). We will use \( tx, ux, vx, ty, tx_i \), etc. to range over \( \mathcal{V}_T \).

**Definition 6.2 (Open Transition)** Fix a TSS \((\Sigma, \mathcal{R}, N)\). An open transition is an open proof of a \( \Sigma \)-literal from \( \mathcal{R} \) using \( \mathcal{V}_T \). For an open transition \( \hat{t} \), \( \mathit{var}(\hat{t}) \) denotes the set of transition variables occurring in \( \hat{t} \); if its root is labelled by \( (p \xrightarrow{a} q, \chi) \) then \( \mathit{src}_\mathcal{V}(\hat{t}) = p \), \( \mathit{tar}_\mathcal{V}(\hat{t}) = q \). The binding function \( \beta_i \) of \( \hat{t} \) from \( \mathit{var}(\hat{t}) \) to \( \Sigma \)-literals is defined by \( \beta_i(tx) = \mu \) if \( tx \in \mathit{var}(\hat{t}) \) and \( (\mu, tx) \) is the label of a node in \( \hat{t} \). Given an open transition, we refer to the subproofs obtained by deleting the root node as its direct subtransitions.

All occurrences of transition variables are considered free. Let \( \mathcal{T}'(\Sigma, \mathcal{R}, N) \) be the set of open transitions in the TSS \((\Sigma, \mathcal{R}, N)\) and \( \mathcal{T}'(\Sigma, \mathcal{R}, N) \) the set of closed open transitions. We have \( \mathcal{T}'(\Sigma, \mathcal{R}, N) = \mathcal{T} \).

Let \( \mathit{en}_\mathcal{V}(p) \) denote \( \{ \hat{t} \mid \mathit{src}_\mathcal{V}(\hat{t}) = p \} \).

**Definition 6.3 (Transition Expression)** A transition declaration is a tuple \((R, n, I)\) of a transition constructor \( R \), an arity \( n \in \mathbb{N} \) and a trigger set \( I \subseteq \{1, \ldots, n\} \). A transition signature is a set of transition declarations. The set \( \mathcal{T}\mathcal{E}'(\Sigma, \mathcal{R}, \Sigma_T) \) of transition expressions over a process signature \( \Sigma_P \) and a transition signature \( \Sigma_T \) is defined inductively as follows.

- if \( tx \in \mathcal{V}_T \) and \( \mu \) is a \( \Sigma \)-literal then \( (tx : \mu) \in \mathcal{T}\mathcal{E}'(\Sigma_P, \Sigma_T) \),
- if \( E \in \mathcal{T}\mathcal{E}'(\Sigma_P, \Sigma_T), S : V_P \rightarrow P'(\Sigma_P) \) and \( X \in \mathit{dom}(S) \) then \( \mathit{rec}\mathit{Act}(X, S, E), \mathit{rec}\mathit{In}(X, S, E) \in \mathcal{T}\mathcal{E}'(\Sigma_P, \Sigma_T) \), and
- if \((R, n, I) \in \mathcal{E}_T, E_i \in \mathcal{T}\mathcal{E}'(\Sigma_P, \Sigma_T) \) for each \( i \in I \), and \( E_i \in P'(\Sigma_P) \) for each \( i \in \{1, \ldots, n\} \setminus I \), then \( R(E_1, \ldots, E_n) \in \mathcal{T}\mathcal{E}'(\Sigma_P, \Sigma_T) \).

Given a TSS \((\Sigma, \mathcal{R}, N)\) in De Simone format, each open transition \( \hat{t} \in \mathcal{T}'(\Sigma, \mathcal{R}) \) is named by a unique transition expression in \( \mathcal{T}\mathcal{E}'(\Sigma, \mathcal{T}_T) \); here \( \mathcal{T}_T = \{(N(r), n, I) \mid r \in \mathcal{R} \) is a De Simone rule, \( n \) is its arity and \( I \) is its trigger set}\):

- if the root of \( \hat{t} \) is labelled by \( (\mu, tx) \) where \( tx \in \mathcal{V}_T \) then \( \hat{t} \) is named \((tx : \mu) \),
- if the root of \( \hat{t} \) is labelled by \((X|S) \xrightarrow{a} q, R) \) where \( a \in \mathit{Act} \) then \( \hat{t} \) is named \( \mathit{rec}\mathit{Act}(X, S, E) \) where \( E \) is the name of the direct subtransition of \( \hat{t} \),
- if the root of \( \hat{t} \) is labelled by \((X|S) \xrightarrow{\ell} (X|S), R) \) where \( \ell \in \langle \mathcal{L}\rangle \setminus \mathit{Act} \) then \( \hat{t} \) is named \( \mathit{rec}\mathit{In}(X, S, E) \) where \( E \) is the name of the direct subtransition of \( \hat{t} \), and
- if the root of \( \hat{t} \) is labelled by \((Op|p_1, \ldots, p_n) \xrightarrow{a} q, R) \) then \( \hat{t} \) is named \( R(E_1, \ldots, E_n) \) where, letting \( n \) and \( I \) be the arity and the trigger set of the rules named \( R, E_i \) for each \( i \in I \) is the name of the direct subtransitions of \( \hat{t} \) corresponding to the index \( i \), and \( E_i = p_i \) for each \( i \in \{1, \ldots, n\} \setminus I \).

We now see that the first requirement for the rule-naming function in Definition 5.1 ensures that every open transition is uniquely identified by its name.

**Definition 6.4 (Transition Substitution)** Let \((\Sigma, \mathcal{R}, N)\) be a TSS. A \((\Sigma, \mathcal{R})\)-substitution is a partial function \( \sigma_T : (\mathcal{V}_P \rightarrow P'(\Sigma)) \cup (\mathcal{V}_T \rightarrow \mathcal{T}'(\Sigma, \mathcal{R})) \). It is closed if it is a total function \( \sigma_T : (\mathcal{V}_P \rightarrow P'(\Sigma)) \cup (\mathcal{V}_T \rightarrow \mathcal{T}'(\Sigma, \mathcal{R})) \). A \((\Sigma, \mathcal{R})\)-substitution \( \sigma_T \) matches all process expressions. It matches an open transition \( \hat{t} \) whose binding function is \( \beta_i \) if for all \((tx, \mu) \in \beta_i \), \( \sigma_T(tx) \) being defined and \( \mu = (p \xrightarrow{a} q) \) implies \( \ell_i(\sigma_T(tx)) = a \) and \( \mathit{src}_\mathcal{V}(\sigma_T(tx)), \mathit{tar}_\mathcal{V}(\sigma_T(tx)) \) being the substitution instances of \( p, q \) respectively by applying \( \sigma_T \) to \( \mathcal{V}_P \).

If \( E \in P'(\Sigma) \cup \mathcal{T}'(\Sigma, \mathcal{R}) \) and \( \sigma_T \) is a \((\Sigma, \mathcal{R})\)-substitution matching \( E \), then \( E[\sigma_T] \) denotes the expression obtained from \( E \) by replacing, for \( tx \in \mathcal{V}_T \) in the domain of \( \sigma_T \), every subexpression of the form...
(tx :: µ) in E by σT(tx), and for x ∈ Vtx in the domain of σT, every free occurrence of x in E by σT(x), while renaming bound process variables if necessary to prevent name-clashes. In that case E[σT] is called a substitution instance of E.

Note that a substitution instance of an open transition can be a transition expression not representing an open transition. For example, applying a (Σ, R)-substitution σT given by σT(ty) := (tx :: y \xrightarrow{c_1} y') to the open transition (tx :: x \xrightarrow{c_2} x') | (ty :: y \xrightarrow{c_3} y') results in (tx :: x \xrightarrow{c_1} x') | (tx :: y \xrightarrow{c_2} y') which is not an open transition because the transition variable tx is used for two different Σ-literals. This will not happen if σT is closed.

**Observation 6.5** Given a TSS (Σ, R, N), if \( \hat{\ell} \in \text{en}_c(p) \) is a transition and σT is a closed (Σ, R)-substitution which matches \( \hat{\ell} \) then \( \hat{\ell}[σT] \in \text{Tr} \), \( \text{src}_c[\hat{\ell}[σT]] = \text{src}_c(\hat{\ell}) = \ell_c(\hat{\ell}) \) and \( \text{target}(\hat{\ell}[σT]) = \text{tar}_c(\hat{\ell})[σT] \).

**Definition 6.6** (Transition System Specification with Successors) Let (Σ, R, N) be a TSS. A (Σ, R)-transition literal is an expression \( \hat{\ell} \xrightarrow{\hat{u}} \hat{v} \) with \( \hat{u}, \hat{v} \in \prod'(Σ, R) \), \( \text{src}_c(\hat{\ell}) = \text{src}_c(\hat{u}) \) and \( \text{src}_c(\hat{v}) = \text{tar}_c(\hat{u}) \). A successor rule over (Σ, R) is an expression of the form \( H \xrightarrow{\hat{\ell}} G \) with \( H \) a finite list of (Σ, R)-literals (the premises of the successor rule) and \( \hat{\lambda} \) a (Σ, R)-literal (the conclusion). A transition system specification with successors (TSSS) is a tuple (Σ, R, N, U) with (Σ, R, N) a TSS and \( U \) a set of successor rules over (Σ, R).

**Definition 6.7** (Associated LTSS) For a TSSS (Σ, R, N, U), the associated LTSS is the LTSS (S, Tr, source, target, \( \ell, \sim \)) with \( S := \prod'(Σ) \), Tr the collection of proofs \( \pi \) of closed Σ-literals \( p \xrightarrow{a} q \) from R, where \( \text{source}(\pi) = p \), \( \ell(\pi) = a \) and \( \text{target}(\pi) = q \), and \( \sim := \{ (t, u, v) \mid \text{a proof of closed (Σ, R)}-\text{l literal } t \sim_u v \text{ from } U \text{ exists} \} \).

### 7 De Simone Languages with Successors

We have enriched standard definitions such as transition systems and specifications with successors. This allows us to add successors to the De Simone format to define a new congruence format.

**Definition 7.1** (De Simone Format) A TSSS (Σ, R, N, U) is in De Simone format if (Σ, R, N) is in De Simone format, for every recursive call \( \langle X | S \rangle \) and \( xa, ya, za \in \mathcal{L} \) it has a successor rule

\[
\frac{(tx :: Sx \xrightarrow{x} x')}{(ty :: Sx \xrightarrow{y} y')} \frac{(tz :: y' \xrightarrow{za} z')}{rec_X(X, S, tx :: Sx \xrightarrow{x} x')} \xrightarrow{rec_A(x, S, ty :: Sx \xrightarrow{y} y')} \hat{u}
\]

where \( \hat{u} = (tz :: y' \xrightarrow{za} z') \) if \( ya \in \text{Act} \) and \( \hat{u} = \text{rec}_X(X, S, tx :: Sx \xrightarrow{x} x') \) otherwise, \( \text{rec}_X = \text{rec}_A(x) \) if \( xa \in \text{Act} \) and \( \text{rec}_X = \text{rec}_I(x) \) otherwise, \( x', y', z' \) are pairwise distinct process variables not occurring in \( \langle X | S \rangle \), and \( tx, ty, tz \) are pairwise distinct transition variables. Moreover, each of its other successor rules has the form

\[
\frac{(tx_i :: x_{i} \xrightarrow{x_{i}} x_{i}')}{(ty_i :: y_{i} \xrightarrow{y_{i}} y_{i}')} \frac{(tz_i :: y_{i}' \xrightarrow{z_{i}} z_{i}')}{R(xe_1, \ldots, xe_n) \xrightarrow{s(xe_1, \ldots, xe_n)} \hat{v}}
\]

such that

- \( I \subseteq \{1, \ldots, n\} \),
- \( x_i, x_i', y_i, z_i' \) for all relevant \( i \) are pairwise distinct process variables,
- \( tx_i, ty_i, tz_i \) for all relevant \( i \) are pairwise distinct transition variables,
The other conditions of Definition 7.1 are illustrated by the Venn diagram of Figure 1. The outer circle depicts the indices 1,...,n numbering the arguments of the operator Op that is the common type of the De Simone rules named R and S; I_R and I_S are the trigger sets of R and S, respectively. In line with Definition 6.3, xe = x_i for i ∈ I_R, and xe = (tx_i :: x_i × y_i → x'_i) for i ∈ I_S. Likewise, ye = x_i for i ∈ I_S, and ye = (ty'_i :: x_i y_i → y'_i) for i ∈ I_S. So the premises of any rule named S are \{x_i \times y_i → y'_i | i ∈ I_S\}. By Definition 5.1 the target of such a rule is a univariate process expression q with no other variables than z_1, ... ,z_n, where z_i := x_i for i ∈ I_S and z_i := y'_i for i ∈ I_S. Since src_o(\dot{v}) = q, the transition expression \dot{v} must be univariate, and have no variables other than z_i e_i for i = 1,...,n, where z_i e_i is either the process variable z_i or a transition variable expression (t_i :: z_i → z'_i).

Figure 1: Inclusion between index sets I, I_R, I_S, I_T, I_G ⊆ \{1,...,n\}. One has (I_R ∩ I_G) \setminus I_S ⊆ I_T.

The annotations n_i show the location of index i (suppressed for unary operators) of rule n.
I is the set of indices \( i \) for which the above successor rule has a premise. Since this premise involves the transition variables \( t_x \) and \( t_y \), necessarily \( I \subseteq I_0 \cap I_S \). Let \( I_G \) be the set of indices for which \( z e_i \) occurs in \( \hat{v} \), and \( I_T \subseteq I_G \) be the subset where \( z e_i \) is a transition variable. The conditions on \( \hat{v} \) in Definition 7.1 say that \( I \cap I_G \subseteq I_T \) and \( (I_R \cap I_G) \setminus I_S \subseteq I_T \). For \( i \in I \cap I_G \), the transition variable \( t_x \) is inherited from the premises of the rule, and for \( i \in (I_R \cap I_G) \setminus I_S \) the transition variable \( t_z \) is inherited from its source.

In order to show that most classes of indices allowed by our format are indeed populated, we indicated the positions of the indices of the rules of CCS and (the forthcoming) ABCdE from Tables 2 and 5.

Any De Simone language, including CCS, SCCS, ACP and ME1E, can trivially be extended to a language with successors, e.g. by setting \( \mathcal{U} = \emptyset \). This would formalise the assumption that the parallel composition operator of these languages is governed by a scheduler, scheduling actions from different components in a nondeterministic way. The choice of \( \mathcal{U} \) from Table 2 instead formalises the assumption that parallel components act independently, up to synchronisations between them.

We now present the main theorem of this paper, namely that ep-bisimulation is a lean congruence for all languages that can be presented in De Simone format with successors. A lean congruence preserves equivalence when replacing closed subexpressions of a process by equivalent alternatives. Being a lean congruence implies being a congruence for all operators of the language, but also covers the recursion construct.

**Theorem 7.2 (Lean Congruence)** Ep-bisimulation is a lean congruence for all De Simone languages with successors. Formally, fix a TSSS \((\Sigma, \mathcal{R}, n, \mathcal{U})\) in De Simone format. If \( p \in \mathbb{P}^r(\Sigma) \) and \( \rho, \nu \) are two closed \( \Sigma \)-substitutions with \( \forall x \in \mathcal{V}_P, \rho(x) \leftrightarrow_{ep} \nu(x) \) then \( \rho[p] \leftrightarrow_{ep} \nu[p] \).

The proof can be found in Appendix A of the full version of this paper [15].

In contrast to a lean congruence, a full congruence would also allow replacement within a recursive specification of subexpressions that may contain recursion variables bound outside of these subexpressions. As our proof is already sophisticated, we consider the proof of full congruence to be beyond the scope of the paper. In fact we are only aware of two papers that provide a proof of full congruence via a rule format [21, 9].

We carefully designed our De Simone format with successors and can state the following conjecture.

**Conjecture 7.3** Ep-bisimulation is a full congruence for all De Simone languages with successors.

## 8 A Larger Case Study: The Process Algebra ABCdE

The *Algebra of Broadcast Communication with discards and Emissions* (ABCdE) stems from [13]. It combines CCS [17], its extension with broadcast communication [20, 11, 10], and its extension with signals [4, 6, 7, 10]. Here, we extend CCS as presented in Section 3.

ABCdE is parametrised with sets \( \mathcal{C} \) of *handshake communication names* as used in CCS, \( \mathcal{B} \) of broadcast communication names and \( \mathcal{I} \) of signals. \( \bar{\mathcal{I}} := \{ \bar{s} \mid s \in \mathcal{I} \} \) is the set of signal emissions. The collections \( \mathcal{B}, \mathcal{B}' \) and \( \mathcal{B} : \text{of broadcast, receive, and discard actions are given by } \mathcal{B}_s := \{ b \mid b \in \mathcal{B} \} \) for \( \bar{s} \in \{!, ?, :: \} \). \( \mathcal{A} := \mathcal{C} \cup \mathcal{B} \cup \{ \tau \} \cup \mathcal{B}' \cup \mathcal{L} \cup \mathcal{I} \) is the set of *actions*, with \( \tau \) the internal action, and \( \mathcal{L} := \mathcal{A} \cup \mathcal{B} : \bar{\mathcal{I}} \) is the set of *transition labels*. Complementation extends to \( \mathcal{C} \cup \mathcal{C} \cup \mathcal{I} \cup \bar{\mathcal{I}} \) by \( \bar{c} := c \).

Below, \( r \) ranges over \( \mathcal{C} \cup \mathcal{C} \cup \mathcal{L} \cup \mathcal{I} \), \( \eta \) over \( \mathcal{C} \cup \mathcal{C} \cup \{ \tau \} \cup \mathcal{I} \cup \bar{\mathcal{I}} \), \( \mathcal{A} \) over \( \mathcal{A} \), \( \ell \) over \( \mathcal{L} \), \( \gamma \) over \( \mathcal{I} \), \( \mathcal{B} \) over \( \mathcal{B} \), \( \bar{s}, \bar{s}_1, \bar{s}_2 \) over \( \{!, ?, :: \} \), \( s \) over \( \mathcal{I} \), \( S \) over recursive specifications and \( X \) over \( \mathcal{V}_S \). A *relabelling* is a function \( f : (\mathcal{C} \to \mathcal{C}) \cup (\mathcal{B} \to \mathcal{B}) \cup (\mathcal{I} \to \mathcal{I}) \); it extends to \( \mathcal{L} \) by \( f(\bar{c}) = \bar{f}(c) \), \( f(\tau) := \tau \) and \( f(b \bar{z}) = f(b) \bar{z} \).

Next to the constant and operators of CCS, the process signature \( \Sigma \) of ABCdE features a unary *signalling* operator \( \_s \) for each signal \( s \in \mathcal{I} \).
Table 3: Structural operational semantics of ABCdE

\[
\begin{align*}
\frac{b:*}{0 \xrightarrow{b} 0} & \quad \frac{\alpha \neq b?}{\alpha.x \xrightarrow{b} \alpha.x} & \quad \frac{x \xrightarrow{b} x', y \xrightarrow{b} y'}{x + y \xrightarrow{b} x' + y'} & +_c \\
\frac{x \xrightarrow{b_1} x', y \xrightarrow{b_2} y'}{(\#_1 \circ \#_2 = \# \neq \_)} & \frac{x|y \xrightarrow{b} x'|y'}{} & & |_c \\
\frac{x \xrightarrow{\alpha} x'}{x' \xrightarrow{\alpha} x'} & \frac{x \xrightarrow{\gamma} x'}{x' \xrightarrow{\gamma} x'' \xrightarrow{s}} & \frac{\langle Sx|S \rangle \xrightarrow{\gamma} y}{\langle X|S \rangle \xrightarrow{\gamma} \langle X|S \rangle} & \text{rec}_S
\end{align*}
\]

The semantics of ABCdE is given by the transition rule templates displayed in Tables 1 and 3. The latter augments CCS with mechanisms for broadcast communication and signalling. The rule \(|_c\) presents the core of broadcast communication [20], where any broadcast-action \(b!\) performed by a component in a parallel composition needs to synchronise with either a receive action \(b?\) or a discard action \(b:\_\) of any other component. In order to ensure associativity of the parallel composition, rule \(|_c\) also allows receipt actions of both components \((\#_1 = \#_2 = \_))\), or a receipt and a discard, to be combined into a receipt action.

A transition \(p \xrightarrow{b} q\) is derivable only if \(q = p\). It indicates that the process \(p\) is unable to receive a broadcast communication \(b!\) on channel \(b\). The Rule \(b:*\) allows the nil process (inaction) to discard any incoming message; in the same spirit \(b:\alpha\) allows a message to be discarded by a process that cannot receive it. A process offering a choice can only perform a discard-action if both choice-options can discard the corresponding broadcast (Rule \(+^S\). Finally, by rule \(\text{rec}_S\), a recursively defined process \(\langle X|S \rangle\) can discard a broadcast iff \(\langle Sx|S \rangle\) can discard it. The variant \(\text{rec}_P\) of \(\text{rec}_S\) is introduced to maintain the property that \(\text{target}(\theta) = \text{source}(\theta)\) for any indicator transition \(\theta\).

A signalling process \(p\)'s emits the signal \(s\) to be read by another process. A typically example is a traffic light being red. Signal emission is modelled as an indicator transition, which does not change the state of the emitting process. The first rule \((-^S)\) models the emission \(\tilde{s}\) of signal \(s\) to the environment. The environment (processes running in parallel) can read the signal by performing a read action \(s\). This action synchronises with the emission \(\tilde{s}\), via rule \(|_C\) of Table 1. Reading a signal does not change the state of the emitter.

Rules \(+^S\) and \(+^e\) describe the interaction between signal emission and choice. Relabelling and restriction are handled by rules \(\setminus_{L}\) and \(\{f\}\) of Table 1, respectively. These operators do not prevent the emission of a signal, and emitting signals never changes the state of the emitting process. Signal emission \(p\)'s does not block other transitions of \(p\).

It is trivial to check that the TSS of ABCdE is in De Simone format.

The transition signature of ABCdE (Table 4) is completely determined by the set of transition rule templates in Tables 1 and 3. We have united the rules for handshaking and broadcast communication by assigning the same name \(|_c\) to all their instances. When expressing transitions in ABCdE as expressions, we use infix notation for the binary transition constructors, and prefix or postfix notation for unary ones. For example, the transition \(b:*()\) is shortened to \(b:*\), \(\alpha(p)\) to \(\alpha p\), \(\setminus L(t)\) to \(\setminus Lp\), and \(\setminus_{L}(t, p)\) to \(\setminus_{L}p\).
Table 4: Transition signature of ABCdE

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Arity</th>
<th>Trigger Set</th>
<th>sAct</th>
<th>sIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b:0.</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b:α.</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+L</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+R</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+C</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+E</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1c</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1r</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1f</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1s</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>f</td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td></td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>[f]</td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5: Successor rules for ABCdE

<table>
<thead>
<tr>
<th>Meta</th>
<th>Variable Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>x₁</td>
</tr>
<tr>
<td>Q</td>
<td>x₂</td>
</tr>
<tr>
<td>P'</td>
<td>y₁</td>
</tr>
<tr>
<td>Q'</td>
<td>y₂</td>
</tr>
<tr>
<td>t</td>
<td>(tx₁ :: x₁</td>
</tr>
<tr>
<td>u</td>
<td>(tx₂ :: x₂</td>
</tr>
<tr>
<td>v</td>
<td>(ty₁ :: x₁</td>
</tr>
<tr>
<td>w</td>
<td>(ty₂ :: x₂</td>
</tr>
<tr>
<td>t'</td>
<td>(tz₁ :: y₁</td>
</tr>
<tr>
<td>u'</td>
<td>(tz₂ :: y₂</td>
</tr>
</tbody>
</table>

Table 5 extends the successor relation of CCS (Table 2) to ABCdE. P, Q are process variables, t, v transition variables enabled at P, u, w transition variables enabled at Q, P', Q' the targets of v, w, respectively and t', u' transitions enabled at P', Q', respectively. To express those rules in the same way as Definition 7.1, we replace the metavariables P, Q, t, u, etc. with variable expressions as indicated on the right. Here xaᵢ, yaᵢ, zaᵢ are label variables that should be instantiated to match the trigger of the rules and side conditions. As ABCdE does not feature operators of arity ≻2, the index i from Definition 7.1 can be 1 or 2 only.

To save duplication of rules 8b, 8c, 9b, 9c and 10 we have assigned the same name ₁c to the rules for handshaking and broadcast communication. The intuition of the rules of Table 5 is explained in detail in [13].

In the naming convention for transitions from [13] the sub- and superscripts of the transition constructors +, | and *s, and of the recursion construct, were suppressed. In most cases that yields no ambiguity, as the difference between ₁ and |₁, for instance, can be detected by checking which of its two arguments are of type transition versus process. Moreover, it avoids the duplication in rules 3a, 4a, 5, 6, 11c and 11d. The ambiguity between +₁ and +₁ (or +₁ and +₁) was in [13] resolved by adorning rules 3–6 with a side condition ₖ(v) /∈ ̂̂ or ₖ(w) /∈ ̂̂, and the ambiguity between recAct and recIn (or *sAct and *sIn) by adorning rules 11c and 11d with a side condition ₖ(v) /∈ Act; this is not needed here.

It is easy to check that all rules are in the newly introduced De Simone format, except Rule 1. However, this rule can be converted in to a collection of De Simone rules by substituting R(xₑ₁, .., xₑₙ) for X and S(yₑ₁, .., yₑₙ) for ζ, adding a premise in the form of xₑᵢ → xₑᵢ (tzᵢ :: yᵢ → zᵢ) if i ∈ ℓᵢ ∩ ℓₙ, for each i ∈ ℓᵢ ∩ ℓₙ.
for each pair of rules of the same type named $r$ and $s$. The various occurrences of $1$ in Figure 1 refer to these substitution instances. It follows that $\preceq_{ep}$ is a congruence for the operators of ABCdE, as well as a lean congruence for recursion.

9 Related Work & Conclusion

In this paper we have added a successor relation to the well-known De Simone format. This has allowed us to prove the general result that enabling preserving bisimilarity – a finer semantic equivalence relation than strong bisimulation – is a lean congruence for all languages with a structural operational semantics within this format. We do not cover full congruence yet, as proofs for general recursions are incredible hard and usually excluded from work justifying semantic equivalences.

There is ample work on congruence formats in the literature. Good overview papers are [1, 19]. For system description languages that do not capture time, probability or other useful extensions to standard process algebras, all congruence formats target strong bisimilarity, or some semantic equivalence or preorder that is strictly coarser than strong bisimilarity. As far as we know, the present paper is the first to define a congruence format for a semantic equivalence that is finer than strong bisimilarity.

Our congruence format also ensures a lean congruence for recursion. So far, the only papers that provide a rule format yielding a congruence property for recursion are [21] and [9], and both of them target strong bisimilarity.

In Sections 3 and 8, we have applied our format to show lean congruence of $ep$-bisimilarity for the process algebra CCS and ABCdE, respectively. This latter process algebra features broadcast communication and signalling. These two features are representative for issues that may arise elsewhere, and help to ensure that our results are as general as possible. Our congruence format can effortlessly be applied to other calculi like CSP [5] or ACP [3].

In order to evaluate $ep$-bisimilarity on process algebras like CCS, CSP, ACP or ABCdE, their semantics needs to be given in terms of labelled transition systems extended with a successor relation $\leadsto$. This relation models concurrency between transitions enabled in the same state, and also tells what happens to a transition if a concurrent transition is executed first. Without this extra component, labelled transition systems lack the necessary information to capture liveness properties in the sense explained in the introduction.

In a previous paper [13] we already gave such a semantics to ABCdE. The rules for the successor relation presented in [13], displayed in Tables 2 and 5, are now seen to fit our congruence format. We can now also conclude that $ep$-bisimulation is a lean congruence for ABCdE. In [14, Appendix B] we contemplate a very different approach for defining the relation $\leadsto$. Following [10], we understand each transition as the synchronisation of a number of elementary particles called synchrons. Then relations on synchrons are proposed in terms of which the $\leadsto$-relation is defined. It is shown that this leads to the same $\leadsto$-relation as the operational approach from [13] and Tables 2 and 5.

---

3This yields $1^2 + 2 \cdot 1 + 5 \cdot 3 + 3 \cdot 2 + 2 \cdot 1 = 26$ rules of types $(0,0)$, $(\alpha_{\leadsto},1)$, $(+,2)$, $(\cdot\!,1)$ and $(X|S)$ not included in Tables 2 and 5.
References


