Translation of CCS into CSP, Correct up to Strong Bisimulation

Gerard Ekembe Ngondi[1,2(✉)], Vasileios Koutavas[1,2(✉)], and Andrew Butterfield[1,2(✉)]

[1] Trinity College Dublin, Dublin, Ireland
[2] Lero - The Irish Software Research Centre, Limerick, Ireland
{Gerard.Ekembe,Vasileios.Koutavas,Andrew.Butterfield}@tcd.ie

Abstract. We present a translation of CCS into CSP which is correct with respect to strong bisimulation. To our knowledge this is the first such translation to enjoy a correctness property. This contributes to the unification of the CCS and CSP families of concurrent calculi, in the spirit of Hoare and He's unification programme through Unifying Theories of Programming. To facilitate this translation, we define CCSTau, the extension of CCS with visible synchronisation actions and the hiding operator. This separation of concerns between synchronisation and hiding turns out be sufficient to obtain our correct translation. Our translation, implemented in a Haskell prototype, makes it possible to use CSP-based verifiers such as FDR to reason about trace and failure (hence may- and must-testing) preorders for CCS processes.

Keywords: Concurrency theory · Calculus of Communicating Systems (CCS) · Communicating Sequential Processes (CSP) · Correct translation

1 Introduction

The CCS/Pi-calculus [1,16,17] and CSP/CSPmob [4,14,22] families of calculi are established formalisms for analysing concurrent systems. Not long after their inception there have been efforts to relate the two calculi and bridge their differences [15]. This would have clear benefits for theoreticians as it would allow them a deeper understanding of the nature of concurrency and the ability to transition from one mathematical formulation to the other in a rigorous manner. It would also benefit practitioners working in Process Algebra as it would allow them to use verification technology from both worlds to address challenges in assuring system correctness modelled in either family of calculi. To achieve this, semantics preserving transformations between CCS and CSP are needed.

In previous work, Van Glabbeek [10] builds a general framework for comparing the expressiveness of process calculi, with an application proposing a translation from CSP to CCS that is correct up to trace equivalence. Hatzel et al. [12] propose an encoding from CSP into asynchronous CCS with two notable

R. Calinescu and C. S. Păsăreanu (Eds.): SEFM 2021, LNCS 13085, pp. 243–261, 2021.
https://doi.org/10.1007/978-3-030-92124-8_14

encodings of CSP multiway synchronisation into CCS binary synchronisation. Brookes [3] encodes CSP models as synchronisation trees showing that CSP failure equivalence is implied by CCS observational equivalence under certain restrictions. He and Hoare [13] build a retract between CCS and CSP semantics.

To our knowledge however, no translation from CCS into CSP exists to date. The present paper aims to fill this gap. In particular, we present a translation from finite state CCS into CSP that is correct up to strong bisimulation, i.e., the source and target terms are strongly bisimilar. This correctness criterion allows us to use a prototype implementation of our translation to leverage FDR [5] for reasoning about trace and failure refinements of CCS terms. The translation is efficient as it only polynomially increases the size of the term. In the worst case, the target term has $O(nm)$ additional communication prefixes, where n and m are the maximum number of prefixes with the same name and corresponding co-name, respectively, in the source term. For practical systems with a relatively small number of synchronising prefixes this translation is thus tractable.

One major challenge in achieving a correct translation from CCS to CSP has been the reconciliation of the different communication primitives in the two languages, and how these interact with other primitives in the language. To bridge the gap between binary CCS and multiway CSP synchronisations, our translation assigns a unique name a_{ij} to every pair of $a/\overline{a}$-prefixes that might synchronise, and carefully annotates the interfaces between parallel processes to enable these synchronisations, effectively implementing binary synchronisation in multiway CSP semantics. Moreover, a unique name a_i is assigned to every CCS prefix that may be interleaved. This separation of interleaving and synchronisation is key to obtaining our translation (see Example 8). Finally, the CCS mixed-choice operator is translated to CSP external choice with a special *tau*-event to enable internal choice resolution (see Example 9). We use CSP hiding to turn internal synchronisation events a_{ij} and *tau* events into proper CSP τ-events.

Our translation from CCS to CSP relies on a novel intermediate language called CCSTau. This is a CCS-like calculus with *observable* binary synchronisation and the CSP hiding operator. Our translation is then obtained by the composition of an initial translation from CCS into CCSTau, a number of transformations within the CCSTau language itself, and a final translation from CCSTau into CSP including hiding of internal transitions. This sequence of smaller translation steps simplifies the task at hand and allows us to obtain a correct, up to strong bisimulation, overall translation. The contributions of this work are summarised as follows.

- We provide the first translation from finite state CCS to CSP which is correct up to strong bisimulation. The translation is efficient and only polynomially increases the size of the term.
- We propose CCSTau, which adapts CCS by making synchronisation actions visible and introducing CSP-like hiding, as a middle-ground between CCS and CSP. This calculus is instrumental in disentangling complex CCS behaviour

Table 1. CCS transition semantics (omitting symmetric rules).

$$\text{Prefix}: \alpha.P \xrightarrow{\alpha} P \qquad \text{SumL}: \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \text{Rec}: \frac{P[\mu\, X.P/X] \xrightarrow{\alpha} P'}{\mu\, X.P \xrightarrow{\alpha} P'}$$

$$\text{ParL}: \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \text{Com}: \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \text{Res}: \frac{P \xrightarrow{\alpha} P' \quad \alpha \notin B \cup \overline{B}}{P \setminus B \xrightarrow{\alpha} P' \setminus B}$$

such as mixed choice and combined interleaving/synchronisation and encoding it into CSP.
– We provide a prototype implementation of our translation in Haskell [23] which enables the use of the FDR refinement tool [5] to reason about may- and must-testing refinement of CCS processes.

In the rest of the paper we briefly overview definitions for CCS and CSP (Sect. 2), and provide the high-level intuitions of our translation (Sect. 3). We then define CCSTau, the extension of CCS with visible synchronisation actions and the hiding operator (Sect. 4), before defining the actual translation (Sect. 5 and Sect. 6) and prove its correctness (Sect. 7). Section 8 discusses an alternative translation, correct up to failure equivalence. Section 9 evaluates our translation against Gorla's criteria [11] for valid translations. Finally, we present conclusions and discuss future work (Sect. 10).

2 CCS, CSP, Correct Translations: a Brief Overview

CCS (Calculus of Communicating Systems) [1,17] and CSP (Communicating Sequential Processes) [14,22] are process algebras that allow reasoning about concurrent systems. Here we overview the main definitions of the two calculi.

2.1 CCS

In CCS [17], we assume a set of countable names $\mathcal{N}$, ranged over by a, b, c, with a total bijective function $\bar{\cdot}$ with the property that $\overline{\overline{a}} = a$. This function identifies *co-names*, the names that can synchronise. The symbol τ denotes an unobservable internal move. We let α range over names and τ. The syntax of CCS processes is given by the grammar

$$P, Q, R \ ::= \ 0 \mid \alpha.P \mid P + Q \mid P \,|\, Q \mid P \setminus B \mid \mu\, X.P \mid X$$

The set of names that a process can use, denoted by $\mathcal{A}(P)$ for a given CCS process P, is defined hereafter.

Definition 1 (Alphabet/Sort of CCS processes [17, Chap. 2, Def. 2]**).**

$$\mathcal{A}(0) \mathrel{\widehat{=}} \{\,\} \qquad \mathcal{A}(P+Q) \mathrel{\widehat{=}} \mathcal{A}(P) \cup \mathcal{A}(Q)$$
$$\mathcal{A}(\tau.P) \mathrel{\widehat{=}} \mathcal{A}(P) \qquad \mathcal{A}(P \,|\, Q) \mathrel{\widehat{=}} \mathcal{A}(P) \cup \mathcal{A}(Q)$$
$$\mathcal{A}(a.P) \mathrel{\widehat{=}} \{a\} \cup \mathcal{A}(P) \qquad \mathcal{A}(P \setminus B) \mathrel{\widehat{=}} \mathcal{A}(P) \backslash (B \cup \overline{B})$$

The semantics of CCS is traditionally given as a Labelled Transition System (LTS), shown in Table 1. Term 0 (or NIL) is the process that performs no action, whereas $\alpha.P$ performs an action α, where α is either a name a or τ, and then behaves like P (Prefix rule). The choice term $P + Q$ behaves either like P or Q (SumL rule). The parallel $P|Q$ runs P and Q in parallel; P and Q may interleave (Par rule) or synchronise on co-actions, resulting in a silent τ action (Com rule). Restriction $P \upharpoonright B$ cannot engage in actions consisting of names in $B \cup \overline{B}$, where $\overline{B} = \{\bar{a} | a \in B\}$ (Res rule); however, names in $B \cup \overline{B}$ can be used for internal synchronisation in P. Term $\mu\, X.P$ encodes recursion, where variable X, appearing as a process in P, denotes a recursive unfolding (Rec rule). We only consider closed processes where X is under a corresponding μX operator. Moreover, as we are interested in finite state processes, we apply the sufficient requirement that no parallel operator appears under recursion.

Equivalence based on bisimulations is the preferred choice for distinguishing CCS processes (cf. [17,21]).

Definition 2 (Strong Bisimulation [21]). *A strong bisimulation is a symmetric binary relation $\mathcal{R}$ on processes satisfying the following: $P\mathcal{R}Q$ and*

- $P \xrightarrow{\alpha} P'$ *imply that* $\exists\, Q' : Q \xrightarrow{\alpha} Q' \wedge P'\mathcal{R}Q'$
- $Q \xrightarrow{\alpha} Q'$ *imply that* $\exists\, P' : P \xrightarrow{\alpha} P' \wedge P'\mathcal{R}Q'$

P is strong bisimilar *to Q, written $P \sim Q$, if $P\mathcal{R}Q$ for some strong bisimulation $\mathcal{R}$.*

Example 3. In CCS, internal and external choices can be combined thus yielding a *mixed* choice, e.g., $a.P + b.Q + \tau.R$. According to Table 1, this process enables external choice between a and b, meaning that the context of the process, through synchronisation on a or b, can force this process to become P or Q. Additionally, the process itself can non-deterministically decide to evolve to R, resolving the choice independently from external stimuli. As we show in Example 9, this mixed choice in CCS needs to be encoded specifically into CSP, where internal reductions do not resolve an external choice. □

2.2 CSP

In CSP, we assume again a countable set of names $\mathcal{N}$, called the set of observable events and ranged over by a, b, c, the special ✓ event denotes termination, and the τ event denotes an internal move. We let α range all events. The syntax of CSP processes we consider is given by the grammar:[1]

$$P, Q, R ::= \text{SKIP} \mid \text{STOP} \mid a \rightarrow P \mid P \sqcap Q \mid P \,\square\, Q \mid P \underset{B}{\parallel} Q \mid$$
$$P \backslash B \mid f(P) \mid \mu\, X.P \mid X$$

The alphabet of CSP processes is defined hereafter.

[1] In what follows, whether P, Q, R refer to CCS or CSP will be clear by the context.

Table 2. CSP transition semantics (omitting symmetric rules).

$$\text{Term}: SKIP \xrightarrow{\checkmark} STOP \qquad \text{Prefix}: (a \to P) \xrightarrow{a} P \qquad \text{InChL}: P \sqcap Q \xrightarrow{\tau} P$$

$$\text{ExChL1}: \frac{P \xrightarrow{a} P'}{P \,\Box\, Q \xrightarrow{a} P'} \qquad \text{ExChL2}: \frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q} \qquad \text{Rec}: \frac{P[\mu\, X.P/X] \xrightarrow{\alpha} P'}{\mu\, X.P \xrightarrow{\alpha} P'}$$

$$\text{Hide1}: \frac{P \xrightarrow{a} P' \quad [a \notin B]}{P\backslash B \xrightarrow{a} P'\backslash B} \qquad \text{Hide2}: \frac{P \xrightarrow{a} P' \quad [a \in B]}{P\backslash B \xrightarrow{\tau} P'\backslash B} \qquad \text{Ren}: \frac{P \xrightarrow{\alpha} P'}{f(P) \xrightarrow{f(\alpha)} f(P')}$$

$$\text{ParL}: \frac{P \xrightarrow{a} P' \quad [a \notin B^{\checkmark}]}{P \underset{B}{\parallel} Q \xrightarrow{a} P' \underset{B}{\parallel} Q} \qquad \text{Sync}: \frac{P \xrightarrow{a} P' \;\; Q \xrightarrow{a} Q' \;\; [a \in B^{\checkmark}]}{P \underset{B}{\parallel} P \xrightarrow{a} P' \underset{B}{\parallel} Q'}$$

Definition 4 (Alphabet of CSP processes [14]).

$$\begin{aligned}
\mathcal{A}(STOP) &\mathrel{\widehat{=}} \{\} & \mathcal{A}(P \underset{B}{\parallel} Q) &\mathrel{\widehat{=}} \mathcal{A}(P) \cup \mathcal{A}(Q)\\
\mathcal{A}(a \to P) &\mathrel{\widehat{=}} \{a\} \cup \mathcal{A}(P) & \mathcal{A}(P\backslash B) &\mathrel{\widehat{=}} \mathcal{A}(P)\backslash B\\
\mathcal{A}(P \sqcap Q) &\mathrel{\widehat{=}} \mathcal{A}(P) \cup \mathcal{A}(Q) & \mathcal{A}(f(P)) &\mathrel{\widehat{=}} \{f(a) \mid a \in \mathcal{A}(P)\}\\
\mathcal{A}(P \,\Box\, Q) &\mathrel{\widehat{=}} \mathcal{A}(P) \cup \mathcal{A}(Q)
\end{aligned}$$

We present the semantics of CSP as an LTS in Table 2, following Schneider [22]: Term $SKIP$ refuses to engage in any event, terminates immediately (Term rule), and does not diverge. Term $STOP$ is unable to interact with its environment. The prefix process $\alpha \to P$ first engages in event α then behaves like P (Prefix rule). Term $P \sqcap Q$ behaves like P or Q, with the choice decided internally (InChL), whereas $P \,\Box\, Q$ behaves like P or Q, with the choice decided by the environment (ExtChL1,2 rules). Parallel $P \underset{B}{\parallel} Q$ runs processes P and Q in parallel, which must synchronise on the set of events in B and the $\checkmark$ event (ParL and Sync rules). The renaming term $f(P)$ engages in $f(a)$ whenever P engages in a (Ren rule). Hiding $P\backslash A$ engages in all events of P except those in A (Hide1, 2 rules), and $\mu\, X.P$ runs P recursively (Rec rule).

Equivalence based on (enriched versions of) traces is the preferred choice for distinguishing CSP processes (cf. [14,21,22]).

Definition 5 (Failure equivalence [21]). *A* failure *is a pair* (tr, A)*, where* tr *is a finite sequence of actions (or* trace*) and* A *a set of actions. The failure* (tr, A) *belongs to process* P *if, for some* P'*:* $P \xrightarrow{tr} P' \wedge \forall\, a \in A : \neg(P' \xrightarrow{a})$.

P *is* failure equivalent *to* Q*, written* $P =_{\mathcal{F}} Q$*, if they have the same sets of failures.*

2.3 Correct Translations

A correct translation of one language into another is a mapping from the valid expressions in the first language to those in the second, that preserves their meaning [10]. Below we recap the main two definitions of correctness.

Let $\mathcal{L} = (\mathbb{T}_{\mathcal{L}}, [\![\,]\!]_{\mathcal{L}})$ denote a language as a pair of a set $\mathbb{T}_{\mathcal{L}}$ of valid expressions in $\mathcal{L}$ and a surjective mapping $[\![\,]\!]_{\mathcal{L}} : \mathbb{T}_{\mathcal{L}} \to \mathcal{D}_{\mathcal{L}}$ from $\mathbb{T}_{\mathcal{L}}$ to some set of meanings $\mathcal{D}_{\mathcal{L}}$. Candidate instances of $[\![\,]\!]_{\mathcal{L}}$ are *traces* and *failures* (Definition 5).

Definition 6 (Correct Translation up to Semantic Equivalence [10]). *A translation* $\mathsf{T} : \mathbb{T}_{\mathcal{L}} \to \mathbb{T}_{\mathcal{L}'}$ *is* correct up to *a semantic equivalence* $\approx$ *on* $\mathcal{D}_{\mathcal{L}} \cup \mathcal{D}_{\mathcal{L}'}$ *when* $[\![\mathsf{T}(E)]\!]_{\mathcal{L}'} \approx [\![E]\!]_{\mathcal{L}}$ *for all* $E \in \mathbb{T}_{\mathcal{L}}$.

Operational correspondence allows matching the transitions of two processes, which can help determine the appropriate relation (semantic equivalence) between a term and its translation. Let the operational semantics of $\mathcal{L}$ be defined by the labelled transition system $(\mathbb{T}_{\mathcal{L}}, Act_{\mathcal{L}}, \rightarrow_{\mathcal{L}})$, where $Act_{\mathcal{L}}$ is the set of labels and $E \xrightarrow{\lambda}_{\mathcal{L}} E'$ defines transitions with $E, E' \in \mathbb{T}_{\mathcal{L}}$ and $\lambda \in Act_{\mathcal{L}}$.

Definition 7 (Labelled Operational Correspondence [8,20]). *Let* $\mathsf{T} : \mathbb{T}_{\mathcal{L}} \to \mathbb{T}_{\mathcal{L}'}$ *be a mapping from the expressions of a language* $\mathcal{L}$ *to those of a language* $\mathcal{L}'$*, and let* $\mathsf{f} : Act_{\mathcal{L}} \to Act_{\mathcal{L}'}$ *be a mapping from the labels of* $\mathcal{L}$ *to those of* $\mathcal{L}'$*. A translation* $\langle \mathsf{T}, \mathsf{f} \rangle$ *is operationally corresponding w.r.t. a semantic equivalence* $\approx$ *on* $\mathcal{D}_{\mathcal{L}} \cup \mathcal{D}_{\mathcal{L}'}$ *if it is:*

- *Sound:* $\forall\, E, E' : E \xrightarrow{\lambda}_{\mathcal{L}} E'$ *imply that* $\exists\, F : \mathsf{T}(E) \xrightarrow{\mathsf{f}(\lambda)}_{\mathcal{L}'} F$ *and* $F \approx \mathsf{T}(E')$
- *Complete:* $\forall\, E, F : \mathsf{T}(E) \xrightarrow{\lambda'}_{\mathcal{L}'} F$ *imply that* $\exists\, E' : E \xrightarrow{\lambda}_{\mathcal{L}} E'$ *and* $F \approx \mathsf{T}(E') \wedge \lambda' = \mathsf{f}(\lambda)$

3 Intuitions of the Translation

In this section, we illustrate some of the differences between CCS and CSP, and how we address them in the different stages of our translation shown in Fig. 1. We start with the challenges in translating CCS binary into CSP's multiway synchronisation in a term where both interleaving and synchronisation of prefixes is possible.

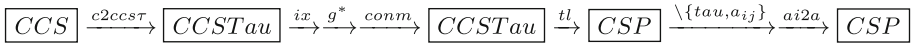

Fig. 1. Translation workflow

Example 8. Consider the CCS process $(a.P \,|\, \overline{a}.Q) \,|\, \overline{a}.R$ composed of three parallel sub-processes $a.P$, $\overline{a}.Q$ and $\overline{a}.R$. According to CCS semantics, binary synchronisation can occur between $a.P$ and either $\overline{a}.Q$ or $\overline{a}.R$. Both synchronisations result to τ-transitions in the LTS (Sync in Table 1).

We initially translate this process into CCSTau through the $c2ccs\tau$ function (Definition 11), which gives us $((a.P' \,|\, \overline{a}.Q') \setminus \{\tau[a|\overline{a}]\} \,|\, \overline{a}.R') \setminus \{\tau[a|\overline{a}]\}$. As we will see in the following section, in CCSTau, $a.P'$ can perform an observable $\tau[a|\overline{a}]$ synchronisation with one of the other two parallel processes. This transition is turned into an internal τ-transition via the hiding operator $(-\setminus\{\tau[a|\overline{a}]\})$ borrowed from CSP. After this first translation, the source and target terms have the same transition system, i.e., they are strongly bisimilar (Theorem 12).

We then apply a sequence of three transformations within CCSTau. The first one, ix (Property 13), assigns a unique index to the names of every prefix, thus obtaining the process $a_1.P'' \,|\, \overline{a}_2.Q'' \,|\, \overline{a}_3.R''$. The ix-indexed process cannot perform any synchronisation and therefore hiding of synchronisation actions is removed. However, the next transformation, g^* (Definition 15), adds new prefixes, denoted with double indices, which re-introduces these synchronisations (though without hiding them):

$$(a_1 + a_{12} + a_{13}).P''' \,|\, \big((\overline{a}_2 + \overline{a}_{12}).Q'''\big) \,|\, (\overline{a}_3 + \overline{a}_{13}).R'''$$

where $(a + b).S$ is syntactic sugar for $a.S + b.S$. For simplicity in this example, we assume that a does not appear in P, Q and R.

At this stage in our translation, every prefix that may lead to an interleaved action is represented by an a_i prefix, while every possible synchronisation has its own unique name and co-name, a_{ij}, $\overline{a}_{ij}$. In this way, we separate synchronisation from interleaving, which is crucial for translating into CSP.

Note here that the introduction of the additional a_{ij} prefixes also introduces interleaved a_{ij} transitions. These will be removed by hiding at a following stage of the translation.

Transformation *conm* (Definition 18) identifies co-names synchronisation events, and *tl* (Definition 20) maps CCS operators to corresponding CSP constructs while filling in the interface sets in every CSP parallel operator. We thus obtain:

$$\big((a_1 \,\Box\, a_{12} \,\Box\, a_{13}) \rightarrow P'''' \underset{\{a_{12}\}}{\parallel} (\overline{a}_2 \,\Box\, a_{12}) \rightarrow Q''''\big) \underset{\{a_{13}\}}{\parallel} (\overline{a}_3 \,\Box\, a_{13}) \rightarrow R''''$$

To obtain a CSP process with a transition system identical to the original CCS term, we need to apply the final two stages of the translation. These introduce a top-level hiding operator for *tau* events (not relevant in this example) and all a_{ij} synchronisation events, as well as a renaming operation $ai2a$ (Definition 25) which maps all a_i names to a. The final CSP term is thus:

$$\Big(\big((a \,\Box\, a_{12} \,\Box\, a_{13}) \rightarrow P'''' \underset{\{a_{12}\}}{\parallel} (\overline{a} \,\Box\, a_{12}) \rightarrow Q''''\big) \underset{\{a_{13}\}}{\parallel} (\overline{a} \,\Box\, a_{13}) \rightarrow R''''\Big) \backslash \{a_{12}, a_{13}\}$$

The original CCS and final CSP terms have indeed strongly bisimilar LTSs (Theorem 30). □

Example 9. Consider again the mixed choice $a.P + b.Q + \tau.R$ (Example 3). After applying the $c2ccs\tau$ translation and the ix, g^* and *conm* transformations, we obtain the CCSTau term: $a_1.P' + b_2.Q' + \tau.R'$. We assume here that P, Q, R do not contain $\overline{a}$ and $\overline{b}$ prefixes and thus no hiding or additional n_{ij} prefixes are introduced. Translation *tl* is the most important for this example. It results in the CSP process $a_1 \rightarrow P'' \,\Box\, b_2 \rightarrow Q'' \,\Box\, tau \rightarrow R''$. Crucially, the last prefix involves the special name *tau*, which is different than τ and can indeed resolve the choice. In order to turn *tau* into a CSP τ move, the translation then hides this name and, with the application of the final renaming function $ai2a$, the CSP term we obtain is $(a \rightarrow P''' \,\Box\, b \rightarrow Q''' \,\Box\, tau \rightarrow R''') \setminus \{tau\}$ which indeed has an LTS which is strongly bisimilar to that of the original CCS term. □

4 From CCS to CCSTau

We define CCSTau to serve as a middle-ground calculus between CCS and CSP for our translation. CCSTau is obtained from CCS, as described in Sect. 2.1, by two modifications: making binary synchronisation observable, and introducing CSP-style hiding.

To make binary synchronisation observable we introduce an additional action which can appear on the transitions of our LTS: $\tau[a|\overline{a}]$. We let β range over CCS actions α and the new synchronisation actions $\tau[a|\overline{a}]$, and define the CCSTau LTS with rules of the form $P \xrightarrow{\beta} Q$. To make synchronisation observable we use the following Com rule, instead of that in Table 1.

$$\text{Com} : \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P|_T Q \xrightarrow{\tau[\overline{a}|a]} P'|_T Q'}$$

Note that we annotate the parallel operator with a T-subscript to make clear that it is the CCSTau parallel. Its alphabet contains visible synchronisations.

Definition 10. $\mathcal{A}(P|_T Q) \mathrel{\widehat{=}} \mathcal{A}(P) \cup \mathcal{A}(Q) \cup \{\tau[a|\overline{a}] \mid a \in \mathcal{A}(P), \bar{a} \in \mathcal{A}(Q)\}$

To introduce hiding, we extend CCS syntax with the hiding construct: $P ::= \ldots \mid P \backslash_T B$. The set B contains actions which are names a or $\tau[a|\overline{a}]$, with the closure condition that "if $a \in B$ then $\overline{a} \in B$ and $\tau[a|\overline{a}] \in B$." We introduce the following hiding rules in the LTS which are similar to the CSP rules (Table 2)

$$\text{Hide1} : \frac{P \xrightarrow{\beta} P' \quad \beta \notin B}{P \backslash_T B \xrightarrow{\beta} P' \backslash_T B} \qquad \text{Hide2} : \frac{P \xrightarrow{\beta} P' \quad \beta \in B}{P \backslash_T B \xrightarrow{\tau} P' \backslash_T B}$$

The remaining LTS rules consist of Prefix, SumL (and its symmetric), ParL, Res and Rec from Table 1, with the only change that we now use β instead of α for transition annotations. Note that CCSTau restriction cannot restrict $\tau[a|\overline{a}]$ actions as these are single-name actions only.

Encoding CCS into CCSTau. We describe here a translation of CCS processes into CCSTau. This encoding is concerned with hiding the now-observable synchronisation actions.

Definition 11 ($c2ccs\tau$). *Translation function $c2ccs\tau$, when applied to a CCS process, returns a CCSTau process.*

$$\begin{aligned}
c2ccs\tau(0) &\mathrel{\widehat{=}} 0 & c2ccs\tau(P \upharpoonright B) &\mathrel{\widehat{=}} c2ccs\tau(P) \upharpoonright B \\
c2ccs\tau(\alpha.P) &\mathrel{\widehat{=}} \alpha.c2ccs\tau(P) & c2ccs\tau(\mu\, X.P) &\mathrel{\widehat{=}} \mu\, X.c2ccs\tau(P) \\
c2ccs\tau(P+Q) &\mathrel{\widehat{=}} c2ccs\tau(P) + c2ccs\tau(Q) & c2ccs\tau(X) &\mathrel{\widehat{=}} X \\
c2ccs\tau(P|Q) &\mathrel{\widehat{=}} (c2ccs\tau(P)|_T c2ccs\tau(Q)) \backslash_T \{\tau[a|\overline{a}] \mid a \in \mathcal{A}(P), \bar{a} \in \mathcal{A}(Q)\}
\end{aligned}$$

In the above definition the only interesting case is parallel which hides the CCStau synchronisation actions, leaving all other actions unaffected. The following theorem shows that the translated terms are strongly bisimilar to the original CCS terms, when there is no parallel under recursion.[2]

Theorem 12. *Let P be a CCS process. Then: $P \sim c2ccs\tau(P)$.*

Proof. By observing that $c2ccs\tau$ is the identity CCS-to-CCSTau translation on parallel-free processes, and then proving the conditions of bisimulation via rule induction on the LTS transitions. □

5 CCSTau Transformations

We provide a translation of CCSTau into CSP in two parts. Here we describe the first part involving the CCSTau transformations, ix, g^*, $conm$, mentioned in Sect. 3.

Indexing (ix)**.** The intention here is that an indexing function assigns unique indices to every prefix in a CCS process. There are many straightforward schemes to choose these indices from the set of natural numbers $\mathbb{N}$. Here, instead of defining a concrete scheme, we specify how it should distribute over CCSTau operators.

Property 13.

$$\begin{aligned}
ix(\tau.P) &= \tau.ix(P) & ix(P \upharpoonright \{a\}) &= ix(P) \upharpoonright \{a_i | a_i \in \mathcal{A}(ix(P))\} \\
ix(a.P) &= a_i.ix_{-i}(P) & ix(P\backslash_{\scriptscriptstyle T}\{a\}) &= ix(P)\backslash_{\scriptscriptstyle T}\{a_i | a_i \in \mathcal{A}(ix(P))\} \\
ix(P+Q) &= ix_1(P) + ix_2(Q) & ix(P\backslash_{\scriptscriptstyle T}\{\tau[a|\bar{a}]\}) &= ix(P) \\
ix(P|_{\scriptscriptstyle T}Q) &= ix_1(P)|_{\scriptscriptstyle T} ix_2(Q) & ix(\mu\, X.P) &= \mu\, X.i\, x(P)
\end{aligned}$$

where ix_{-i} is some indexing scheme which does not assign the i-index, and ix_1, ix_2 are some indexing schemes that assign disjoint indices.

Since ix generates unique indexed names, $ix(P)$ cannot synchronise, whence hidden $\tau[a|\bar{a}]$ synchronisations are dropped out. They will be recovered later on.

In the following, we assume an indexing function ix which satisfies the above properties. Our Haskell implementation [23] indeed implements such an indexing function.

Explicit Binary Synchronisation (g^*)**.** Given an indexed process $ix(P)$, function g^* generates, by over-approximation, a unique name a_{ij} for every possible synchronisation pair $(a_i, \bar{a}_j)$ from $ix(P)$.

Given a set S of names in the context, the next definition shows how g^* applies to a CCSTau action and set of actions. For technical convenience, the definition ensures that smaller indices always come first.

[2] Although more involved versions of $c2ccs\tau$ would lift the restriction on recursion here, the same restriction would be needed for the end-to-end translation into CSP.

Definition 14 $(g^*(S,a), g\pi_2(S,a))$**.**

$$g^*(S,\tau) \mathrel{\widehat{=}} \{\tau\}$$
$$g^*(S,a_i) \mathrel{\widehat{=}} \{a_i\} \cup g\pi_2(S,a_i) \qquad g\pi_2(S,a_i) \mathrel{\widehat{=}} \{a_{ij} \mid \bar{a}_j \in S, i<j\} \cup \{a_{ji} \mid \bar{a}_j \in S, j<i\}$$
$$g^*(S,B) \mathrel{\widehat{=}} \bigcup_{a_i \in B} g^*(S - \{a_i\}, a_i)$$

We can now define our transformation function g^* over CCSTau processes.

Definition 15 $(g^*(S,P))$**.** *Let P,Q be ix-indexed CCSTau processes and S a set of names such that $S \cap \mathcal{A}(P) = S \cap \mathcal{A}(Q) = \{\}$.*

$$g^*(S,0) \mathrel{\widehat{=}} 0 \qquad g^*(S, P \upharpoonright B) \mathrel{\widehat{=}} g^*(S,P) \upharpoonright g^*(S,B)$$
$$g^*(S,\alpha.P) \mathrel{\widehat{=}} \sum_{b \in g^*(S,\alpha)} b.g^*(S,P) \qquad g^*(S, P\backslash_\tau B) \mathrel{\widehat{=}} g^*(S,P)\backslash_\tau g^*(S \cup B, B)$$
$$g^*(S, P+Q) \mathrel{\widehat{=}} g^*(S,P) + g^*(S,Q) \qquad g^*(S, \mu\, X.P) \mathrel{\widehat{=}} \mu\, X.g^*(S,P)$$
$$g^*(S, P \mid_\tau Q) \mathrel{\widehat{=}} g^*(S \cup \mathcal{A}(Q), P) \mid_\tau g^*(S \cup \mathcal{A}(P), Q)$$

When P is the top context, we require $S = \{\}$. We define: $g^*(P) \mathrel{\widehat{=}} g^*(\{\}, P)$.

Condition $S \cap \mathcal{A}P = \{\}$ allows us to separate P from its context, while the condition that the processes are *ix*-indexed excludes processes where indexing has not been applied consistently such as $g^*((a_1 + a_2) \upharpoonright \{a_2\})$.

For restriction, no $a_i \in B$ should be able to interact with the environment. Hence, (dummy) synchronisations between B and S, $\{a_{ij} | a_i \in B, \bar{a}_j \in S | i < j\} \cup \{a_{ji} | a_i \in B, \bar{a}_j \in S | j < i\}$, should also be restricted.

Example 16.

1. $g^*\big((a_1.0 \mid_\tau \bar{a}_2.0) \upharpoonright \{a_1, a_2\}\big) = \big((a_1.0 + a_{12}.0) \mid_\tau (\bar{a}_2.0 + \bar{a}_{12}.0)\big) \upharpoonright \{a_1, a_2\}$
2. $g^*\big((a_1.0 \mid_\tau \bar{a}_2.0) \upharpoonright \{a_1, a_2\} \mid_\tau \bar{a}_3.0\big)$
   $= \big((a_1.0 + a_{12}.0 + a_{13}.0) \mid_\tau (\bar{a}_2.0 + \bar{a}_{12}.0)\big) \upharpoonright \{a_1, a_2, a_{13}\}\big) \mid_\tau \bar{a}_3.0$

Proper synchronisations remain unrestricted as illustrated above with a_{12}. Since CCS restriction '$- \upharpoonright \{a_{ij}\}$' will be translated to CSP '$- \underset{a_{ij}}{\parallel} STOP$' (cf. Definition 19), restricting proper synchronisation names would lead to deadlock in CSP. Instead, they will be added into the CSP interface-parallel operator later on (cf. Definition 20).

For hiding, no $a_i \in B$ should be visible. Unlike restriction, we must hide both dummy and proper synchronisations involving hidden a_is.

Example 17.

1. $g^*\big((a_1.0 \mid_\tau \bar{a}_2.0)\backslash_\tau \{a_1, a_2\}\big) = \big((a_1.0 + a_{12}.0) \mid_\tau (\bar{a}_2.0 + \bar{a}_{12}.0)\big)\backslash_\tau \{a_1, a_2, a_{12}\}$
2. $g^*\big((a_1.0 \mid_\tau \bar{a}_2.0)\backslash_\tau \{a_1, a_2\} \mid_\tau \bar{a}_3.0\big)$
   $= \big((a_1.0 + a_{12}.0 + a_{13}.0) \mid_\tau (\bar{a}_2.0 + \bar{a}_{12}.0)\big)\backslash_\tau \{a_1, a_2, a_{12}, a_{13}\}\big) \mid_\tau \bar{a}_3.0$

Identifying Co-names (*conm*)**.** Unlike CCSTau, synchronisation occurs in CSP between pairs of events that have the same name. That is, $(a \to P) \underset{\{a\}}{\parallel} \bar{a} \to Q$ would behave like $(a \to P) \underset{\{a\}}{\parallel} b \to Q$, *not* $(a \to P) \underset{\{a\}}{\parallel} a \to Q$. Before going into CSP, we need to ensure that a can synchronise with $\bar{a}$, more precisely, we only need a_{ij} to synchronise with $\bar{a}_{ij}$. This can be achieved through the following renaming function, *conm*, which transforms any $\bar{a}_{ij}$-name into an a_{ij}-name.

Definition 18 (*conm*)**.** *Let a_i, a_{ij} range over g^*-indexed names. Then:*

$$conm \mathrel{\widehat{=}} \{\tau \mapsto \tau, a_i \mapsto a_i, \bar{a}_i \mapsto \bar{a}_i, a_{ij} \mapsto a_{ij}, \bar{a}_{ij} \mapsto a_{ij} \mid i < j\}$$

6 From CCSTau to CSP

Translation into CSP (*tl*)**.** The translation of CCSTau processes into CSP requires us to translate CCSTau prefixes. To do this we use a fresh (not previously used) CSP event *tau*, which we will later hide, thus creating a true CSP internal transition. Moreover, we need to translate CCS restriction, which is part of the CCSTau language, into CSP. We do this by introducing a deadlock for the restricted names.

Definition 19. *Let P be a CSP process. Then: $P \upharpoonright_{csp} B \mathrel{\widehat{=}} P \underset{B \cup \overline{B}}{\parallel} STOP$.*

We can now present the translation *tl* from CCSTau to CSP.

Definition 20. *Let P and Q be CCSTau processes; let tau be a fresh, non-synchronising, CSP event.*

$$\begin{aligned}
tl(0) &\mathrel{\widehat{=}} STOP \\
tl(\tau.P) &\mathrel{\widehat{=}} tau \to tl(P) \\
tl(a.P) &\mathrel{\widehat{=}} a \to tl(P) \\
tl(P + Q) &\mathrel{\widehat{=}} tl(P) \,\Box\, tl(Q)
\end{aligned}
\qquad
\begin{aligned}
tl(P|_{\tau} Q) &\mathrel{\widehat{=}} tl(P) \underset{\{a \mid a \in \mathcal{A}(P) \cap \mathcal{A}(Q)\}}{\parallel} tl(Q) \\
tl(P \upharpoonright B) &\mathrel{\widehat{=}} tl(P) \upharpoonright_{csp} B \\
tl(P \backslash_{\tau} B) &\mathrel{\widehat{=}} tl(P) \setminus_{csp} B \\
tl(\mu\, X.P) &\mathrel{\widehat{=}} \mu\, X.tl(P)
\end{aligned}$$

The following example illustrates the rationale for our encoding restriction into CSP.

Example 21. In CCSTau (as in CCS) restriction obeys the law: $(a.P) \upharpoonright \{a\} \sim 0$. Definition 19 obeys the same law, viz., $(a \to t2csp(P)) \upharpoonright_{csp} \{a\} = STOP$.

$$tl((a.P) \upharpoonright \{a\}) = tl(a.P) \upharpoonright_{csp} \{a\} = (a \to tl(P)) \underset{\{a\}}{\parallel} STOP$$

The last process behaves like $STOP$ which is the *tl*-translation of 0. □

Hereafter, it will be convenient to refer to the composition of all CCSTau transformations and the *tl*-translation into CSP as a single function, which we call *t2csp*.

Definition 22 (*t2csp*)**.** *Let P be a CCSTau process. Then:*

$$t2csp(P) \mathrel{\widehat{=}} (tl \circ conm \circ g^* \circ ix(P))\backslash_{csp}\{tau\}$$

Example 23. We illustrate the translation of CCSTau parallel operator, to be contrasted with the translation of CCS parallel operator (cf. Example 26).

$$\begin{aligned}
&t2csp(a.0|_T \bar{a}.0)\\
&= tl\big(conm\big(g^*(\{\}, ix(a.0|_T \bar{a}.0))\big)\big)\backslash_{csp}\{tau\} && \text{(t2csp-Def. 22)}\\
&= tl\big(conm\big(g^*(\{\}, (a_1.0|_T \bar{a}_2.0))\big)\big)\backslash_{csp}\{tau\} && \text{(ix-Prop. 13)}\\
&= tl\big(conm\big((a_1.0 + a_{12}.0)|_T (\bar{a}_2.0 + \bar{a}_{12}.0)\big)\big)\backslash_{csp}\{tau\} && \text{(gstar-Def. 15)}\\
&= tl\big((a_1.0 + a_{12}.0)|_T (\bar{a}_2.0 + a_{12}.0)\big)\backslash_{csp}\{tau\} && \text{(conm-Def. 18)}\\
&= \big((a_1 \rightarrow STOP \,\Box\, a_{12} \rightarrow STOP) \underset{\{a_{12}\}}{\|} \\
&\qquad (\bar{a}_2 \rightarrow STOP \,\Box\, a_{12} \rightarrow STOP)\big)\backslash_{csp}\{tau\} && \text{(tl-Def. 20)}
\end{aligned}$$

Final CSP Transformations *($-\backslash_{csp}\{a_{ij}\}, ai2a$)***.** The final stages of our translation consists of hiding every a_{ij} synchronisation name (thus effectively turning them into τ) and renaming of all a_i names to a.

Definition 24. *Let a_i range over g^*-indexed names; $ai2a \mathrel{\widehat{=}} \{a_i \mapsto a, \bar{a}_i \mapsto \bar{a}\}$.*

The following definition gives the end-to-end translation from CCS to CSP, as described in Sect. 3.

Definition 25. *Let P be a CCS process. Then:*

$$ccs2csp(P) \;\mathrel{\widehat{=}}\; ai2a \circ (t2csp \circ c2ccs\tau(P))\backslash_{csp}\{a_{ij} | a_{ij} \in \mathcal{A}\big(t2csp(c2ccs\tau(P))\big)\}$$

Example 26. In Sect. 3, we discuss at length the translation of CCS binary synchronisation into CSP. This can be illustrated more succinctly as follows:

$$\begin{aligned}
&ccs2csp(a.0|\bar{a}.0)\\
&= ai2a \circ t2csp\big(c2ccs\tau(a.0|\bar{a}.0)\big)\backslash_{csp}\{a_{ij}|..\} && \text{(ccs2csp-Def. 25)}\\
&= ai2a \circ t2csp\big((a.0|_T \bar{a}.0)\backslash_T\{\tau[a|\bar{a}]\}\big)\backslash_{csp}\{a_{ij}|..\} && \text{(c2ccs}\tau\text{-par-Def. 11)}\\
&= ai2a \circ tl \circ conm \circ g^*(\{\}, ix\big((a.0|_T \bar{a}.0)\backslash_T\{\tau[a|\bar{a}]\}\big))\backslash_{csp}\{tau\}\backslash_{csp}\{a_{ij}|..\}\\
&\qquad && \text{(t2csp-Def. 22)}\\
&= ai2a \circ tl \circ conm \circ g^*\big((a_1.0|_T \bar{a}_2.0)\big)\backslash_{csp}\{tau\}\backslash_{csp}\{a_{ij}|..\} && \text{(ix-Prop. 13)}\\
&= ai2a \circ tl \circ conm\big((a_1.0 + a_{12}.0)|_T (\bar{a}_2.0 + \bar{a}_{12}.0)\big)\backslash_{csp}\{tau\}\backslash_{csp}\{a_{12}\} && \text{(gstar-Def. 15)}\\
&= ai2a \circ tl\big((a_1.0 + a_{12}.0)|_T (\bar{a}_2.0 + a_{12}.0)\big)\backslash_{csp}\{tau, a_{12}\} && \text{(conm-Def. 18)}\\
&= ai2a \circ \big((a_1 \,\Box\, a_{12} \rightarrow STOP) \underset{\{a_{12}\}}{\|} (\bar{a}_2 \,\Box\, a_{12} \rightarrow STOP)\big)\backslash_{csp}\{tau, a_{12}\} && \text{(tl-Def. 20)}\\
&= \big((a \,\Box\, a_{12} \rightarrow STOP) \underset{\{a_{12}\}}{\|} (\bar{a} \,\Box\, a_{12} \rightarrow STOP)\big)\backslash_{csp}\{tau, a_{12}\} && \text{(ai2a-Def. 24)}
\end{aligned}$$

7 Correctness of the Translation

Here we discuss the correctness up to a semantic equivalence (Definition 6) of functions g^*, *t2csp*, and *ccs2csp* defined above. In each case, we use labelled operational correspondence to relate a source term to its translation. In the end, the labelled operational correspondence between a CCS term and its CSP translation is a strong bisimulation, hence, translation *ccs2csp* is correct up to strong bisimulation.

First, we consider the correctness of g^*.

Theorem 27 (Operational Correspondence between P and $g^*(S, ix(P))$). *Let P be a CCSTau process. Let $c4star(S,P) \mathrel{\widehat{=}} g^*\big(S, ix(P)\big)$. Then:*

1. $P \xrightarrow{\tau} P'$ *imply that* $\forall S | S \cap \mathcal{A}ix(P) = \{\} : c4star(S,P) \xrightarrow{\tau} Q$ *and* $Q \equiv c4star(S,P')$
2. $\forall S | S \cap \mathcal{A}ix(P) = \{\} : c4star(S,P) \xrightarrow{\tau} Q$ *imply that* $\exists ! P' : P \xrightarrow{\tau} P'$ *and* $Q \equiv c4star(S,P')$
3. $P \xrightarrow{a} P'$ *imply that* $\forall S | S \cap \mathcal{A}ix(P) = \{\} : c4star(S,P) \xrightarrow{a_i} Q$ *and* $Q \equiv c4star(S,P')$
4. $\forall S | S \cap \mathcal{A}ix(P) = \{\} : c4star(S,P) \xrightarrow{a_i} Q$ *imply that* $\exists ! P' : P \xrightarrow{a} P'$ *and* $Q \equiv c4star(S,P')$
5. $P \xrightarrow{a} P'$ *imply that* $\forall S | S \cap \mathcal{A}ix(P) = \{\} : c4star(S,P) \xrightarrow{a_{ij}} Q$ *and* $Q \equiv c4star(S,P')$
6. $\forall S, \exists i,j | i \neq j \wedge a_i \in S \wedge \bar{a}_j \in \mathcal{A}ix(P) : c4star(S,P) \xrightarrow{a_{ij}} Q$ *imply that* $\exists ! P' : P \xrightarrow{a} P'$ *and* $Q \equiv c4star(S,P')$
7. $P \xrightarrow{\tau[a|\bar{a}]} P'$ *imply that* $\forall S | S \cap \mathcal{A}ix(P) = \{\} : c4star(S,P) \xrightarrow{\tau[a_{ij}|\bar{a}_{ij}]} Q$ *and* $Q \equiv c4star(S,P')$
8. $\forall S | S \cap \mathcal{A}ix(P) = \{\} : c4star(S,P) \xrightarrow{\tau[a_{ij}|\bar{a}_{ij}]} Q$ *imply that* $\exists ! P' : P \xrightarrow{\tau[a|\bar{a}]} P'$ *and* $Q \equiv c4star(S,P')$

Proof. By co-induction on the transitions of P and $c4star(S,P)$ respectively. E.g., consider every rule yielding a τ transition, e.g., Prefix, Sum. Induction over each rule yields structural induction over P. Apply the definition of c4star, then $c4star(S,P)$ has a τ transition by successive application of the Sum rule then the Prefix rule. Conversely, given $c4star(S,P)$, induction over each rule yields structural induction over $c4star(S,P)$.

Since a_{ij}-names denote synchronisation, $\tau[a_{ij}|\bar{a}_{ij}]$-actions only should be visible/allowed, viz., a_{ij}-names must be restricted. Hence, $g^*(S, ix(P))$ is not correct. We obtain a correct translation by restricting all a_{ij} names.

Corollary 28 (Correctness up to Strong Bisimulation of $g^*(S, ix(P))$). *Let P be a CCSTau process. Then, $g^*(S, ix(P)) \upharpoonright \{g\pi_2(S, a_i) | a_i \in \mathcal{A}ix(P)\}$ is correct up to strong bisimulation.*

Proof. Apply the restriction operator, $\restriction \{g\pi_2(S, a_i) | a_i \in \mathcal{A}ix(P)\}$*, to every clause in Theorem 27. This eliminates clauses 5 and 6 since* a_{ij} *can no longer occur.*

Theorem 29 (Operational Correspondence of $t2csp$**).** *Let* P *be a CCSTau process. Then:*

1. $P \xrightarrow{\tau} P'$ *imply that* $\forall S | S \cap \mathcal{A}ix(P) = \{\} : t2csp(S, P) \xrightarrow{\tau} t2csp(S, P')$
2. $\forall S | S \cap \mathcal{A}ix(P) = \{\} : t2csp(S, P) \xrightarrow{\tau} Q$ *imply that* $\exists ! P' : P \xrightarrow{\tau} P'$ *and* $Q = t2csp(S, P')$
3. $P \xrightarrow{a} P'$ *imply that* $\forall S | S \cap \mathcal{A}ix(P) = \{\} : t2csp(S, P) \xrightarrow{a_i} t2csp(S, P')$
4. $\forall S | S \cap \mathcal{A}ix(P) = \{\} : t2csp(S, P) \xrightarrow{a_i} Q$ *imply that* $\exists ! P' : P \xrightarrow{a} P'$ *and* $Q = t2csp(S, P')$
5. $P \xrightarrow{\tau[a|\bar{a}]} P'$ *imply that* $\forall S | S \cap \mathcal{A}ix(P) = \{\} : t2csp(S, P) \xrightarrow{a_{ij}} t2csp(S, P')$
6. $\forall S | S \cap \mathcal{A}ix(P) = \{\} : t2csp(S, P) \xrightarrow{a_{ij}} Q$ *imply that* $\exists ! P' : P \xrightarrow{\tau[a|\bar{a}]} P'$ *and* $Q = t2csp(S, P')$

Proof. By co-induction on the transitions of P *and* $t2csp(P)$ *respectively.*

Theorem 30 (Correctness of $ccs2csp$**).** *Let* P *be a CCS process. Then:*

1. $P \xrightarrow{\tau} P'$ *imply that* $\forall S | S \cap \mathcal{A}ix(P) = \{\} : ccs2csp(S, P) \xrightarrow{\tau} ccs2csp(S, P')$
2. $\forall S | S \cap \mathcal{A}ix(P) = \{\} : ccs2csp(S, P) \xrightarrow{\tau} Q$ *imply that* $\exists ! P' : P \xrightarrow{\tau} P'$ *and* $Q = ccs2csp(S, P')$
3. $P \xrightarrow{a} P'$ *imply that* $\forall S | S \cap \mathcal{A}ix(P) = \{\} : ccs2csp(S, P) \xrightarrow{a} ccs2csp(S, P')$
4. $\forall S | S \cap \mathcal{A}ix(P) = \{\} : ccs2csp(S, P) \xrightarrow{a} Q$ *imply that* $\exists ! P' : P \xrightarrow{a} P'$ *and* $Q = ccs2csp(S, P')$

We say that ccs2csp is correct up to strong bisimulation.

Proof. Apply c2ccsτ *(Definition 11), this turns CCS process* P *into CCSTau process* $ccs2\tau(P)$*. Apply* $-\backslash_{csp}\{a_{ij}\}$*, this hides every* a_{ij} *from* $t2csp(c2ccs\tau(P))$*. As a consequence, this eliminates clauses 5 and 6 from Theorem 29. Then apply ai2a, this renames every* a_i *into an* a*. Thus, every* a_i *from Theorem 29 becomes an* a*.*

The above theorems culminate to the following correctness result of our end-to-end translation.

Corollary 31 (Correctness of the Translation up to Strong Bisimulation). *Let* P *be a CCS process. Then:* $ccs2csp(P) \sim P$*.*

A trivial consequence of this corollary is that $P \sim Q \Leftrightarrow ccs2csp(P) \sim ccs2csp(Q)$. Since strong bisimulation is included in failure equivalence, we have $P \sim Q \Leftrightarrow ccs2csp(P) =_{\mathcal{F}} ccs2csp(Q)$. We illustrate this subsequently.

Example 32. In CCS, we have: $(a.0|\bar{a}.0) \upharpoonright \{a\} + b.0 \sim \tau.0 + b.0$. We check that: $ccs2csp\big((a.0|\bar{a}.0) \upharpoonright \{a\} + b.0\big) =_{\mathcal{F}} ccs2csp(\tau.0 + b.0)$.

We have:

$$
\begin{aligned}
& ccs2csp\big((a.0|\bar{a}.0) \upharpoonright \{a\} + b.0\big) \\
&= ai2a \circ t2csp \circ c2ccs\tau\big((a.0|\bar{a}.0) \upharpoonright \{a\} + b.0\big) && \text{(ccs2csp-Def. 25)} \\
&= ai2a \circ t2csp\big((a.0|_{\mathcal{T}} \bar{a}.0)\backslash_{\mathcal{T}}\{\tau[a|\bar{a}]\} \upharpoonright \{a\} + b.0\big) && \text{(c2ccs}\tau\text{-par-Def. 11)} \\
&= ai2a \circ [\big((a_1 \,\Box\, a_{12} \rightarrow STOP) \underset{\{a_{12}\}}{\parallel} (a_2 \,\Box\, a_{12} \rightarrow STOP)\big)\restriction_{csp}\{a_1, a_2\} \,\Box \\
&\qquad (b \rightarrow STOP)]\backslash_{csp}\{tau, a_{12}\} && \text{(Ex. 23, t2csp-Def. 22)} \\
&= [\big((a \,\Box\, a_{12} \rightarrow STOP) \underset{\{a_{12}\}}{\parallel} (a \,\Box\, a_{12} \rightarrow STOP)\big)\restriction_{csp}\{a\} \,\Box \\
&\qquad b \rightarrow STOP]\backslash_{csp}\{tau, a_{12}\} && \text{(ai2a-Def. 24)} \\
&= [\big((a \,\Box\, a_{12} \rightarrow STOP) \underset{\{a_{12}\}}{\parallel} (a \,\Box\, a_{12} \rightarrow STOP) \underset{\{a\}}{\parallel} STOP\big) \,\Box \\
&\qquad (b \rightarrow STOP)]\backslash_{csp}\{tau, a_{12}\} && \text{(res-Def. 19)} \\
&= (a_{12} \rightarrow STOP \,\Box\, b \rightarrow STOP)\backslash_{csp}\{a_{12}\} && \text{(CSP)} \\
&= (STOP \,\Box\, b \rightarrow STOP) \sqcap STOP && \text{(CSP[17, §3.5.1, L10])}
\end{aligned}
$$

We also have:

$$
\begin{aligned}
& ccs2csp(\tau.0 + b.0) \\
&= ai2a \circ t2csp \circ c2ccs\tau(\tau.0 + b.0) && \text{(ccs2csp-Def. 25)} \\
&= ai2a \circ t2csp \circ (\tau.0 + b.0) && \text{(ccs2csp-Def. 11)} \\
&= ai2a \circ (tau \rightarrow STOP \,\Box\, b \rightarrow STOP)\backslash_{csp}\{tau\} && \text{(t2csp-Def. 22)} \\
&= (tau \rightarrow STOP \,\Box\, b \rightarrow STOP)\backslash_{csp}\{tau\} && \text{(ai2a-Def. 24)} \\
&= (STOP \,\Box\, b \rightarrow STOP) \sqcap STOP && \text{(CSP[17, §3.5.1, L10])}
\end{aligned}
$$

8 Alternative Translation, Correct up to Failure Equivalence

De Nicola and Hennessy [18] define a version of CCS, called TCCS, which removes from CCS the summation operator and τ action, and adds external ($\Box$) and internal ($\sqcap$) choice operators. They further provide a translation from CCS to TCCS that is correct up to *must equivalence* ([18, Thm. 4.4]). Reusing

their translation ([18, Def. 4.1]), we arrive at the following CCS-to-CSP translation:

$$ccs2csp_2(P) \mathrel{\widehat{=}} ai2a \circ (tl_2 \circ comm \circ g^* \circ ix(P))\backslash_{csp}\{a_{ij}\} \qquad (ccs2csp_2\text{-def})$$

$$tl_2(\alpha.P) \mathrel{\widehat{=}} \begin{cases} tl_2(P) & \text{if } \alpha = \tau \\ \alpha \to tl_2(P) \end{cases} \qquad (tl_2\text{-prefix})$$

$$tl_2(P+Q) \mathrel{\widehat{=}} \begin{cases} tl_2(P) \,\Box\, tl_2(Q) & \text{if } \forall P' : \neg(P \xrightarrow{\tau} P' \vee Q \xrightarrow{\tau} P') \\ \Big(tl_2(P) \,\Box\, tl_2(Q)\Big) \sqcap \bigsqcap\{tl_2(P') | P \xrightarrow{\tau} P' \vee Q \xrightarrow{\tau} P'\} \end{cases}$$

$$(tl_2\text{-choice})$$

$$tl_2(P) \mathrel{\widehat{=}} tl(P) \qquad \text{if } P \text{ is not prefix or choice}$$

Example 33. In particular, from the definition of $ccs2csp_2$, we derive:

$$ccs2csp_2(a.P + b.Q) = a \to ccs2csp_2(P) \,\Box\, b \to ccs2csp_2(Q)$$
$$ccs2csp_2(\tau.P + b.Q) = (ccs2csp_2(P) \,\Box\, b \to ccs2csp_2(Q)) \sqcap ccs2csp_2(P)$$
$$ccs2csp_2(\tau.P + \tau.Q) = (ccs2csp_2(P) \,\Box\, ccs2csp_2(Q)) \sqcap ccs2csp_2(P) \sqcap ccs2csp_2(Q)$$

(tl_2-prefix) implies that τ prefixes are absent from the LTS of the CSP translation. As a consequence, P and $ccs2csp_2(P)$ *are not strong bisimilar*; however, they are failure equivalent: $P =_{\mathcal{F}} ccs2csp_2(P)$. Note that *ccs2csp* (Definition 25) is failure equivalent to $ccs2csp_2$: $ccs2csp_2(P) =_{\mathcal{F}} ccs2csp(P)$.[3]

9 Structural Properties of the Translation

In the literature of evaluating the relative expressiveness of different calculi, different evaluation criteria for encodings have been proposed (e.g., [6,8,10,11, 19,20]). Gorla [11] notably proposes five requirements for a translation to be *valid*: on the structural end, it must enjoy the compositionality and name invariance properties; on the behavioural end, operational correspondence, divergence reflection, and success sensitiveness.

The translation from CCS to CSP we provide here (*ccs2csp*, Definition 25) is correct up to strong bisimulation (Corollary 31). This is a stronger result than operational correspondence, and by definition, implies both divergence reflection (viz., if a CSP translation diverges then its source CCS term does) and success sensitiveness (viz., a CCS term converges if, and only if, its CSP translation converges, and both converge to the same success final term). Correctness up to strong bisimulation also implies name invariance. E.g., let P be a CCS process, f a given renaming function; then $ccs2csp(f(P)) \sim f(ccs2csp(P))$.

[3] E.g., let $P = Q = 0$ in Example 33 then, compare the mixed choice case with Example 32.

Our translation is not compositional in the sense of Gorla [11], whereby a compositional translation $\mathsf{T} : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ is such that $\mathsf{T}(op(S_1, ..., S_k)) = C^N_{op}(\mathsf{T}(S_1), ..., \mathsf{T}(S_k))$, where op is any operator of $\mathcal{L}_1$, C^N_{op} a context that coordinates translated subterms, and $N = fn(S_1, ..., S_k)$. However, Gorla acknowledges the existence of correct translations that are not compositional and further acknowledges that his proposal is not adequate to deal with encodings defined as a family of translations T_Σ, where Σ denotes auxiliary parameters of the translation (including sets of names) [11, Conclusion]. Our encoding from CCS to CSP, *ccs2csp* (Definition 25), falls into the latter category.

10 Conclusion and Future Work

In this paper we have studied the relationship between CCS and CSP as part of a greater work that aims to link also Pi-calculus [16] with CSPmob [4]. Many extensions were necessary in order to define the links from CCS to CSP. We have not explored here links in the opposite direction, from CSP to CCS. We leave this to future work. For reference, van Glabbeek [10] proposes a link from CSP to CCS that is correct up to trace equivalence.

We have defined CCSTau, which extends CCS with visible synchronisations and the hiding operator. This allowed us to separate synchronisation from hiding in a CCS context. We notably show that CCS is a subset of CCSTau. We then defined the translation from CCSTau to CSP. In order to achieve this, we extended CSP with the restriction operator.

The most difficult feature to translate was the CCS synchronisation mechanism. In CCS, a single name is capable of both interleaving and synchronisation; and synchronisation (automatically) implies hiding. This is unlike CSP where all these issues are handled in separate operators. The constraint then was to preserve CCS binary synchronisation from capture by CSP multiway synchronisation. To resolve this, we have proposed the g^* renaming approach: if two CCS processes can synchronise on an action b, then a name unique to these two processes, say b_{ij}, is generated to substitute b. Hence, only two processes will ever be able to synchronise on b_{ij} after application of g^*. This guarantees that in CSP, there will never be more than two processes capable of synchronising on b_{ij}, thus avoiding capture by multiway synchronisation. We show that the g^*-based translation is correct. Another solution is possible: extend CSP with binary synchronisation, then translate CCS binary synchronisation into CSP binary synchronisation. We leave the presentation of this alternative to a future publication.

We have proposed here the translation from CCS to CSP only. The main reason for this is our interest in using CSP tools such as FDR for reasoning about CCS processes. With regard to this concern, the g^*-renaming approach is more readily implementable than the second approach. The latter would require extending FDR with semantics (viz. rules) for m-among-n synchronisation.

A natural extension of this paper is to translate Pi-calculus [16] into CSPmob [4]. Assuming that CCS is a subset of Pi-calculus and given that CSP is a subset

of CSPmob, we will focus our attention on mobility constructs hence. Our final goal is to formalise our results in Unifying Theories of Programming (UTP) [13]. One advantage of the latter would be the extension of both CCS and CSP with a richer notion of state. For illustration, Garavel [9] deplores the limitations of the prefix operator in both CCS and CSP and shows that a richer form of sequential composition can be achieved based on a richer notion of state. Moving to UTP will also allow us to mechanise our results using ongoing mechanisation of UTP theories in Isabelle [7], and link up with Isabelle mechanisation of Psi-calculi [2] (a variant of Pi-calculus).

Acknowledgments. The authors are grateful to the anonymous reviewers for their suggestions on how to improve this paper. This work was funded in part by the Science Foundation Ireland grant 13/RC/2094 (LERO), and co-funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 754489. For the purpose of Open Access, the authors have applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission.

References

1. Aceto, L., Larsen, K.A., Ingolfsdottir, A.: An Introduction to Milner's CCS (2005). http://twiki.di.uniroma1.it/pub/MFS/WebHome/intro2ccs.pdf. Accessed 30 July 2021
2. Bengtson, J., Parrow, J., Weber, T.: Psi-calculi in Isabelle. J. Autom. Reason. **56**(1), 1–47 (2015). https://doi.org/10.1007/s10817-015-9336-2
3. Brookes, S.D.: On the relationship of CCS and CSP. In: Diaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 83–96. Springer, Heidelberg (1983). https://doi.org/10.1007/BFb0036899
4. Ngondi, G.E.: Denotational semantics of channel mobility in UTP-CSP. Formal Aspects Comput. **33**(1), 803–826 (2021). https://doi.org/10.1007/s00165-021-00546-3
5. FDR Documentation. https://cocotec.io/fdr/manual/. Accessed 30 July 2021
6. Felleisen, M.: On the expressive power of programming languages. Sci. Comput. Program. **17**, 35–75 (1991). https://doi.org/10.1016/0167-6423(91)90036-W
7. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: a mechanised theory engineering framework. In: Naumann, D. (ed.) UTP 2014. LNCS, vol. 8963, pp. 21–41. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14806-9_2
8. Fu, Y., Lu, H.: On the expressiveness of interaction. TCS **411**, 1387–1451 (2010). https://doi.org/10.1016/j.tcs.2009.11.011
9. Garavel, H.: Revisiting sequential composition in process calculi. J. Log. Algebraic Methods Program **84**, 742–762 (2015). https://doi.org/10.1016/j.jlamp.2015.08.001
10. van Glabbeek, R.: Musings on encodings and expressiveness. In: EPTCS, vol. 89, pp. 81–98 (2012). https://doi.org/10.4204/EPTCS.89.7
11. Gorla, D.: Towards a unified approach to encodability and separation results for process calculi. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 492–507. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_38

12. Hatzel, M., Wagner, C., Peters, K., Nestmann, U.: Encoding CSP into CCS. In: EXPRESS/SOS Workshop. EPTCS, vol. 190, pp. 61–75 (2015). https://doi.org/10.4204/EPTCS.190.5
13. He, J., Hoare, C.A.R.: CSP is a retract of CCS. TCS **411**, 1311–1337 (2010). https://doi.org/10.1016/j.tcs.2009.12.012
14. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Hoboken (1985)
15. Hoare, C.A.R.: Why ever CSP. ENTCS **162**, 209–215 (2006). https://doi.org/10.1016/j.entcs.2006.01.031
16. Milner, R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, Cambridge (1999)
17. Milner, R.: Communication and Concurrency. Prentice-Hall, Hoboken (1989)
18. De Nicola, R., Hennessy, M.: CCS without τ's. In: Ehrig, H., Kowalski, R., Levi, G., Montanari, U. (eds.) CAAP 1987. LNCS, vol. 249, pp. 138–152. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-17660-8_53
19. Parrow, J.: Expressiveness of process algebras. ENTCS **209**, 173–186 (2008). https://doi.org/10.1016/j.entcs.2008.04.011
20. Peters, K.: Comparing process calculi using encodings. In: EXPRESS/SOS Workshop. EPTCS, vol. 300, pp. 19–38 (2019). https://doi.org/10.4204/EPTCS.300.2
21. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press, Cambridge (2012)
22. Schneider, S.: Concurrent and Real-Time Systems - The CSP Approach. Wiley, Hoboken (2000)
23. Haskell Prototype Automation of CCS-to-CSP Translation: GitHub Repository. https://github.com/andrewbutterfield/ccs2csp. Accessed 30 July 2021