
PERMISSION-BASED SEPARATION LOGIC FOR MESSAGE-PASSING CONCURRENCY

ADRIAN FRANCALANZA^a, JULIAN RATHKE^b, AND VLADIMIRO SASSONE^c

^a ICT, University of Malta
e-mail address: adrian.francalanza@um.edu.mt

^{b,c} ECS, University of Southampton, UK
e-mail address: jr2@ecs.soton.ac.uk, vs@ecs.soton.ac.uk

ABSTRACT. We develop local reasoning techniques for message passing concurrent programs based on ideas from separation logics and resource usage analysis. We extend processes with permission-resources and define a reduction semantics for this extended language. This provides a foundation for interpreting separation formulas for message-passing concurrency. We also define a sound proof system permitting us to infer satisfaction compositionally using local, separation-based reasoning.

1. INTRODUCTION

Reasoning about concurrent programs is widely acknowledged to be a difficult business due to the intricate interferences between threads scheduled non-deterministically and to the intrinsic difficulty of scaling reasoning techniques to account for these. The use of *local* reasoning techniques in the guise of separation logic [33, 28] represents a promising advance for this area. Here, the state of resources acted upon by threads are reasoned about independently, where possible. This approach has spawned numerous papers [7, 5, 37, 11, 15, 29, 10] targetting the shared-variable concurrency model.

An alternative, albeit slightly higher-level, model of concurrency is that of message-passing whereby the only shared resources allowed are the message-passing channels themselves. Access to these shared resources is controlled by the message-passing programming interface and so interfering behaviour is more explicit and therefore can be tracked more readily. This paradigm has been extensively studied using process calculi [21, 26, 27, 34] but has also been efficiently implemented and deployed in more programming oriented settings [16, 32, 3].

In this paper we develop a local reasoning proof system for message-passing concurrent programs, based on ideas from both concurrent separation logics [28] and permission-based resource analyses [6, 5]. Our initial step towards the broader and ambitious goal of local reasoning for message-passing systems focusses on the study of confluent value-passing programs, a class large enough to present a significant theoretical challenge while still being of considerable practical interest.

1998 ACM Subject Classification: F.3.1, F.3.2, F.3.3.

Key words and phrases: process calculi, separation logic, deterministic concurrency.

Our approach to using processes as a model for separation-based Hoare-style reasoning centers around the conceptual partitioning of message-passing programs into ‘program state’, *i.e.*, the values emitted on asynchronous outputs, and ‘program code’, *i.e.*, the remaining parallel processes acting on this state. For instance, one way to view the program

$$c_1!4 \parallel c_2!2 \parallel c_1?x.c_2?y.\text{if } x = y \text{ then } (c_1!(x, y, x+x)\|d!) \text{ else } (c_2!(x, y, x+y)\|d!) \quad (1.1)$$

would be to consider the asynchronous outputs

$$c_1!4 \parallel c_2!2 \quad (1.2)$$

as the ‘state’, holding values 4 and 2 at ‘addresses’ c_1 and c_2 and the process

$$c_1?x.c_2?y.\text{if } x = y \text{ then } (c_1!(x, x+x)\|d!) \text{ else } (c_2!(x, y, x+y)\|d!) \quad (1.3)$$

as the ‘code’, or state transformer, consuming the values on channels c_1 and c_2 and producing a new state holding the previous values consumed from c_1 and c_2 together with their summation on either of the *previously used* channels c_1 and c_2 , depending on whether these values were equal or not, and signals on channel d . Using such an analogy, we can decompose our analysis and reason about sub-programs independently. We can interpret assertions over processes such as

$$c_1\langle 4 \rangle * c_2\langle 2 \rangle \quad (1.4)$$

This assertion, a conjunction, describes a process reducing to a ‘soup’ of two asynchronous outputs on channels c_1 and c_2 , holding values 4 and 2, respectively; the process in (1.2) would satisfy this assertion. This state-based process view also permits an intuitive formulation of Hoare-style sequents of the form

$$\{c_1\langle 4 \rangle * c_2\langle 2 \rangle\} \text{ --- } \{c_2\langle 4, 2, 6 \rangle * d\langle \rangle\} \quad (1.5)$$

Such a sequent describes a process that, once composed with the state described by the precondition $c_1\langle 4 \rangle * c_2\langle 2 \rangle$, reduces to some other stable state described by the postcondition $c_2\langle 4, 2, 6 \rangle * d\langle \rangle$, with values 4, 2, 6 on channel c_2 and an empty tuple on channel d acting as a signal, indicating that the data on channel c_2 can now be accessed; the process in (1.3) would satisfy this sequent. In compositional fashion, we can then determine that the entire program of (1.1) reduces to a stable state satisfying $c_2\langle 4, 2, 6 \rangle * d\langle \rangle$ from separate analyses relating to the two sub-programs.

This state-based logical view of processes lends itself well to the specification of deterministic computation whose operation can be decomposed into asynchronous parallel subcomponents. Application examples range from parallel processing of data, [23], to distributed agreement problems, [25]. State-based specifications would allow a more natural expression of the expected behaviour of these algorithms because they are agnostic *wrt.* the specific temporal order of the generation and consumption of this state. For instance, as opposed to temporal logics such as [35], the formula (1.4) does not specify whether the sub-state $c_1\langle 4 \rangle$ is to be produced before $c_2\langle 2 \rangle$ or vice-versa. Dually, sequents such as (1.5) do not necessarily specify if and how this data on channels is to be consumed. The temporal agnosticism in ‘spatial’ specifications is also more amenable to intuitive decompositions and composition of analysis; we can verify that a process P satisfies the formula (1.4) from sub-processes making up P that satisfy $c_1\langle 4 \rangle$ and $c_2\langle 2 \rangle$.

The state-based logical process view is also appealing because the specifications of the algorithms we are considering are also, in some sense, more data-centric rather than control-centric and focus more on the relationships between data at the beginning and the end of computation. One can in fact view the sequent in (1.5) as a description on how the data on channels c_1 and c_2 in the precondition changes to the data on c_2 in the postcondition; the dependencies between such data

will be made more explicit later on once we introduce value variables. Finally, data-centric applications such as in-place sorting also tend to reuse data-placeholders during computation, possibly at different types and formats *e.g.*, the code in (1.3), in order to minimise resource usage. Correctness specifications such as the sequent in (1.5) handle this aspect rather naturally as opposed to traditional correctness analysis for message passing programs, such as type systems in [4, 36], which often limit channel usage to one form of data.

A central assumption underlying our process interpretations is the absence of program interference and the deterministic reduction of processes. In a message-passing paradigm, program interference is caused by *races* for values, through multiple outputs or inputs competing for *shared* channels. In cases such as (1.1) above, where channels are reused, rudimentary analysis based on the free names of processes *e.g.*, [1] are too coarse for adequate race detection. Moreover, these type based safety analyses *e.g.*, [4, 36] tend to avoid reasoning about data, approximating control over branching as a result.

To reason about such interferences in the presence of channel reuse, we define a resource-semantics for processes, based on *linear* input and output permissions. Every process is embellished with a set of permissions, $\lceil P \rceil_\rho$, denoting that process P ‘owns’ the permissions in set ρ (*cf. ownership hypothesis*, [28]). The resource-semantics limits communication to the permissions owned by a process. Thus, for example, for the following reduction to occur

$$\lceil c_1!4 \rceil_\rho \parallel \lceil c_1?x.P \rceil_\mu \longrightarrow \lceil P\{4/x\} \rceil_{\rho \cup \mu} \quad (1.6)$$

the output process, $c_1!4$, (*resp.* the input process, $c_1?x.P$), must have the permission to output (*resp.* to input) on channel c_1 in its permission-set ρ (*resp.* μ). Since permissions are not part of the original process semantics (they are only added in the resource-semantics to aid reasoning) the above enriched reduction also describes the *implicit* transfer of permissions ρ from the output process, $c_1!4$, to the input process, $c_1?x.P$, *i.e.*, adding ρ to the already owned permissions μ , as a result of their synchronisation (*cf. ownership transfer* [28]).

$$\{c_1\langle 4 \rangle * c_2\langle 2 \rangle\} \left[\begin{array}{l} c_1?x.c_2?y. \text{if } x = y \text{ then } c_1!(x, x+x)\|d! \\ \text{else } c_2!(x, y, x+y)\|d! \end{array} \right]_{\{\downarrow c_1, \downarrow c_2, \uparrow d\}} \{c_2\langle 4, 2, 6 \rangle * d\langle \rangle\} \quad (1.7)$$

The earlier sequent (1.5) can now be stated in terms of the process of (1.3) confined by the permissions $\downarrow c_1, \downarrow c_2$ and $\uparrow d$, as shown in (1.7). Note how channel reuse manifests itself through the fact that our permission-confined process in (1.7) does not own the output permissions $\uparrow c_1$ and $\uparrow c_2$, even though they are clearly used in this code. These however will be obtained from the precondition; from a permission perspective, the inputs on channels c_1 and c_2 act as guards, masking the use of the permissions $\uparrow c_1$ and $\uparrow c_2$.

Making ownership explicit also simplifies the detection of races in the model and provides an immediate notion of process separation in terms of owned permissions. For instance, in (1.6) we determine that there are no races across the two processes $\lceil c_1!4 \rceil_\rho$ and $\lceil c_1?x.P \rceil_\mu$ without having to analyse the actual structure of the respective confined processes $c_1!4$ and $c_1?x.P$; instead we simply check that their permission sets are disjoint *i.e.*, $\rho \cap \mu = \emptyset$ (*cf. separation property* [28]). This assumed disjunction of permissions will also play a major role in the semantic definition of (1.7), as it allows us to give a separation interpretation to our sequents.

$$\{c_1\langle 4 \rangle * c_2\langle 2 \rangle\} c_1?x.c_2?y. \left(\begin{array}{l} \text{if } x = y \text{ then } c_1!(x, x+x)\|d! \\ \text{else } c_2!(x, y, x+y)\|d! \end{array} \right) \{c_2\langle 4, 2, 6 \rangle * d\langle \rangle\} \quad (1.8)$$

Another pleasing property of this embellishment is that, in the absence of races, this resource-semantics corresponds to the standard (permission-less) reduction semantics. Thus the permission semantics can be used as a *narrative* to support reasoning about confluent reductions of processes. This therefore means that we can abstract over the existence of such a narrative in our sequents and express (1.7) simply as the permission-less sequent in (1.8), thereby returning to our original aim and obtaining Hoare-triple specifications in terms of processes.

We define a sound proof system for the aforementioned logic and resource-confined processes with judgements of the form:

$$\Gamma; b \vdash \{\varphi\} S \{\psi\}$$

The environment, Γ , associates channels with ownership transfer invariants of permissions, and S denotes a system of processes confined by permissions. These sequents depart slightly from previous work on concurrent separation logic [28], as value-domain assertions - assertions interpreted exclusively in terms of the domain of values communicated and thus independent of the process structure, S - are extracted from the pre and post-conditions, φ and ψ , and consolidated as a boolean expression, b . Correctness proofs in this proof system weave together two inter-dependent mechanisms. On the one hand, they verify, *in sequential fashion*, the satisfaction of the post-condition ψ for system S , assuming the precondition φ ; the soundness of this sequential analysis stems from the non-interference properties guaranteed by the resource semantics of S . On the other hand, sequents construct race-free systems S , using assumptions from the environment, Γ , and the pre-condition, φ .

We have already argued for the naturality of our specifications *wrt.* deterministic message-passing programs and how our analysis can handle more refined branching control analysis, even when this is *data dependent* as in (1.1). Another, perhaps even more crucial advantage of our approach over existing analysis techniques for message-passing concurrency (e.g., [20, 2, 13]) is *locality of reasoning*. By concentrating on deterministic code, our reasoning need not take into account the different interleaving of concurrent code executing in context; this facilitates substantially proof *compositionality* and induces a *lightweight* sequential form of analysis. Explicit permission ownership simplifies interference delineation, even in the presence of channel reuse; such delineation is a major obstacle when defining manageable compositional proof rules (e.g., [13]).

The paper is structured as follows. We introduce our language in Section 2. In Section 3 we define a resource-semantics for permission-confined processes and state its key properties. We define our assertion logic and interpret it using a separation model over confined processes in Section 4. In Section 5 we present our proof system and declare its soundness whereas in Section 6 we apply this system to prove properties about message-passing programs. Finally, in Section 7 we make concluding remarks regarding related and future work.

2. LANGUAGE

Our language, an asynchronous value-passing CCS, is described in Figure 1 and consists of three syntactic categories. Values, $v, u \in \text{VALUES}$, are numerals denoting integers. *Side-effect free* expressions, e , denote integer operations that may contain variables $x, y \in \text{VARS}$. We assume an evaluation function from closed expressions to values, $e \Downarrow v$. We also assume a denumerable set of channel names $c, d \in \text{NAMES}$ and process names $K \in \text{PNAMES}$ and denote lists of values, variables and channels as \vec{v}, \vec{x} and \vec{c} respectively.

Values, Expressions, Boolean Expressions and Processes

$$\begin{aligned}
 v, u &::= 0 \mid 1 \mid \dots & e &::= v \mid x \mid e + e \mid e - e & b &::= e \leq e \mid \neg b \mid b \wedge b \\
 P, Q &::= c! \vec{e} \mid c? \vec{x}. P \mid \text{if } b \text{ then } P \text{ else } Q \mid K(\vec{e})[\vec{c}/\vec{d}] \mid \text{nil} \mid P \parallel Q \mid (\text{new } c)P
 \end{aligned}$$

Structural Equivalence Rules

$$\begin{array}{ll}
 \text{sCOM} & P \parallel Q \equiv Q \parallel P & \text{sASS} & P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R \\
 \text{sNEW} & (\text{new } c)\text{nil} \equiv \text{nil} & \text{sSWP} & (\text{new } c)(\text{new } d)P \equiv (\text{new } d)(\text{new } c)P \\
 \text{sNIL} & P \parallel \text{nil} \equiv P & \text{sEXT} & P \parallel (\text{new } c)Q \equiv (\text{new } c)(P \parallel Q) \quad \text{if } c \notin \text{fn}(P)
 \end{array}$$

Reduction Rules

$$\begin{array}{ll}
 \text{rTHN} & \frac{b \Downarrow tt}{\text{if } b \text{ then } P \text{ else } Q \longrightarrow P} & \text{rELS} & \frac{b \Downarrow ff}{\text{if } b \text{ then } P \text{ else } Q \longrightarrow Q} \\
 \text{rCOM} & \frac{\vec{e} \Downarrow \vec{v}}{c! \vec{e} \parallel c? \vec{x}. P \longrightarrow P[\vec{v}/\vec{x}]} & \text{rPRC} & \frac{K(\vec{x}) \triangleq P \quad \vec{e} \Downarrow \vec{v}}{K(\vec{e})[\vec{c}/\vec{d}] \longrightarrow P[\vec{v}/\vec{x}][\vec{c}/\vec{d}]} \\
 \text{rRES} & \frac{P \longrightarrow P'}{(\text{new } c)P \longrightarrow (\text{new } c)P'} & \text{rPAR} & \frac{P \longrightarrow P'}{P \parallel Q \longrightarrow P' \parallel Q} & \text{rSTR} & \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}
 \end{array}$$

Figure 1: Processes, Structural Equivalence and Reduction

2.1. Syntax. The main syntactic category is that of processes which can asynchronously send the evaluation of expressions on a channel¹, $c! \vec{e}$, receive values on a channel, $c? \vec{x}. P$, and branch on the evaluation of boolean expressions, $\text{if } b \text{ then } P \text{ else } Q$. Processes may assume a number of parameterised (possibly recursive) process definitions, $K(\vec{x}) \triangleq P$; these can be invoked by the call $K(\vec{e})[\vec{c}/\vec{d}]$, instantiating the process variables \vec{x} with the evaluation of \vec{e} and renaming the names \vec{d} to \vec{c} . Finally, processes may also be inactive, nil , execute in parallel, $P \parallel Q$, and can restrict the scope of channels to subsets of processes, $(\text{new } c)P$.

2.2. Reduction Semantics. The rules for the judgement $P \longrightarrow Q$ in Figure 1 describe the dynamics of *closed* processes *i.e.*, processes whose message variables \vec{x} are all bound by input constructs $c? \vec{x}.$, and process names are all defined. Closed boolean expressions, *i.e.*, boolean formulas without free variables, have a *classical* interpretation over the boolean domain $\{tt, ff\}$, characterised by the two judgements $b \Downarrow tt$ and $b \Downarrow ff$. Although this is entirely standard, we explicitly stated here in Definition 2.1 due to its central role in subsequent development (*cf.* Section 5).

Definition 2.1 (Boolean Condition Interpretation).

$$\begin{aligned}
 e_1 \leq e_2 \Downarrow & \begin{cases} tt & \text{if } e_1 \Downarrow v_1, e_2 \Downarrow v_2 \text{ and } v_1 \leq v_2 \\ ff & \text{if } e_1 \Downarrow v_1, e_2 \Downarrow v_2 \text{ and } v_2 < v_1 \end{cases} & \neg b \Downarrow & \begin{cases} tt & \text{if } b \Downarrow ff \\ ff & \text{if } b \Downarrow tt \end{cases} \\
 b_1 \wedge b_2 \Downarrow & \begin{cases} tt & \text{if } b_1 \Downarrow tt \text{ and } b_2 \Downarrow tt \\ ff & \text{if } (b_1 \Downarrow ff \text{ and } b_2 \Downarrow tt) \text{ or } (b_1 \Downarrow tt \text{ and } b_2 \Downarrow ff) \text{ or } (b_1 \Downarrow ff \text{ and } b_2 \Downarrow ff) \end{cases}
 \end{aligned}$$

¹Our language does not allow channel names to be communicated, as in the piCalculus [27, 34].

A number of shorthand conventions are used. We write $c!$ for $c!\vec{e}$ and $c?.P$ for $c?\vec{x}.P$ when $|\vec{e}| = 0$ (resp. $|\vec{x}| = 0$). We elide arguments and renaming from process calls, resp. $K[\vec{e}/\vec{d}]$ and $K(\vec{e})$, whenever these are empty lists. We also write $e_1 = e_2$ for $(e_1 \leq e_2) \wedge (e_2 \leq e_1)$, $e_1 < e_2$ for $\neg(e_2 \leq e_1)$, **true** for $0 \leq 1$, **false** for $1 \leq 0$, $b_1 \vee b_2$ for $\neg(\neg b_1 \wedge \neg b_2)$ and $b_1 \Rightarrow b_2$ for $\neg b_1 \vee b_2$. Finally, we use the shorthand $\vec{e} \Downarrow \vec{v}$ for the evaluation of lists of expressions $e_1 \Downarrow v_1 \dots e_n \Downarrow v_n$ whenever $\vec{e} = e_1 \dots e_n$ and $\vec{v} = v_1 \dots v_n$.

Substitutions, $\sigma \in \text{SUB}$, are total maps from variables to values, $\text{VARS} \rightarrow \text{VALUES}$, and are used to define the semantics of rules RCOM and RPRC . They are finitely denoted as $\{\vec{v}/\vec{x}\}$, meaning that every $x_i \in \vec{x}$ is mapped to its respective $v_i \in \vec{v}$, while abstracting over all the other variable mappings in the substitution. In the case of RPRC only, we abuse this notation to express the *renaming* of \vec{d} to \vec{c} . In Section 5 we abuse again this notation to describe substitutions from variables to expressions, $\{\vec{e}/\vec{x}\}$. Our semantics assumes the following property of expression evaluations, which will be useful later in Section 5.

Assumption 2.2. $e_1 \{\vec{v}/\vec{x}\} \Downarrow v_1$ and $\vec{e} \Downarrow \vec{v}$ implies $e_1 \{\vec{e}/\vec{x}\} \Downarrow v_1$

A brief note on some conventions used. To improve readability we have attempted to minimise the use of universal and existential quantifiers in our statements. Thus, unless explicitly stated, free variables introduced to the left of an implication are to be understood as universally quantified, whereas free variables introduced to the right of an implications are understood as existentially quantified.

As is standard in process calculi presentations [27, 34], the definition of the reduction semantics is kept compact through the rule RSTR and the use of process structural equivalence rules, $P \equiv Q$, defined also in Figure 1. Later on, this structural equivalence will play a role in abstracting away from the precise structure of processes when describing the satisfaction of our logic (cf. Section 4).

2.3. Process Determinism. The reduction semantics of Figure 1 induces the following definitions relating to stability, evaluation and determinism, where \longrightarrow^* denotes the reflexive transitive closure of \longrightarrow .

Definition 2.3 (Stability). $P \not\rightarrow \stackrel{\text{def}}{=} \nexists Q. P \longrightarrow Q$

Definition 2.4 (Evaluation). $P \Downarrow Q \stackrel{\text{def}}{=} \exists Q'. P \longrightarrow^* Q'$ and $Q' \not\rightarrow$ and $Q' \equiv Q$

Our definition of process determinism, Definition 2.6, differs from that given in [26] in that it requires convergence, $P \Downarrow$ cf. Definition 2.5. We also define divergence, as the dual of convergence in standard fashion, in order to describe the existence of an infinite reduction path. Defining determinism in terms of convergence carries other advantages apart from the obvious relevance of termination in resource-aware settings of computation; it arguably allows for a more intuitive definition of determinism in terms of the comparison of the stable processes evaluated to (the second clause in Definition 2.6). Moreover, it fits well with our running theme of a state-based view of processes.

Definition 2.5 (Convergence and Divergence). \Downarrow is the least predicate over processes satisfying the conditions:

$$P \Downarrow = P \not\rightarrow \text{ or } (\forall Q. P \longrightarrow Q \text{ implies } Q \Downarrow)$$

Divergence, $P \Uparrow$, denotes the inverse, $P \Downarrow$.

Definition 2.6 (Determinism). P is deterministic iff:

- (1) $P \Downarrow$
- (2) $P \Downarrow Q_1$ and $P \Downarrow Q_2$ implies $Q_1 \equiv Q_2$

Concurrent code is notoriously hard to analyse. One major source of complication is the potential non-deterministic behaviour of this code, which impacts the ability to tractably define manageable compositional proof rules (e.g., [13]). More precisely, generic non-deterministic code requires one to take into account the various interleaving of concurrent code executing in its context potentially affecting its execution.

Although message passing concurrency minimises this interference to well defined interfaces, problems persist due to races on shared channels. Channel reuse together with the lack of an explicit account of resource usage makes interference hard to delineate.

Example 2.7. The (composite) process Prg takes two inputs x_1, x_2 on channels c_1, c_2 respectively. It discards x_2 and, if x_1 is less than 10, outputs the value x_1 itself together with its double on c_1 while using c_4 as a signal. Otherwise, it uses c_4 to output x_1 by itself.

$$\begin{aligned} Prg &\triangleq (\text{new } c_3) (Fltr \parallel Dbl) \\ Dbl &\triangleq c_2 ? x_2 . c_3 ? x_4 . c_1 ! (x_4 + x_4) \\ Fltr &\triangleq c_1 ? x_1 . \text{if } x_1 \leq 9 \text{ then } c_3 ! x_1 \parallel c_1 ? x_3 . (c_1 ! (x_1, x_3) \parallel c_4 !) \text{ else } c_4 ! x_1 \end{aligned}$$

Internally, Prg is composed of two sub-processes, $Fltr$ and Dbl , sharing a scoped channel, c_3 . Process $Fltr$ filters whether x_1 is less than 10 and forwards the value to process Dbl on channel c_3 which, in turn, *reuses* channel c_1 to return the doubled value.

The process Prg trivially converges as it is stable. When placed in the context of race-free outputs such as $c_1 ! v_1 \parallel c_2 ! v_2$, Prg still converges and evaluates *deterministically* to

$$\begin{aligned} Prg \parallel c_1 ! v_1 \parallel c_2 ! v_2 &\Downarrow c_4 ! \parallel c_1 ! (v_1, 2 \times v_1) && \text{when } v_1 \leq 9 \text{ and;} \\ Prg \parallel c_1 ! v_1 \parallel c_2 ! v_2 &\Downarrow c_4 ! v_1 \parallel (\text{new } c_3) (c_3 ? x_4 . c_1 ! (x_4 + x_4)) && \text{when } v_1 > 9 \end{aligned}$$

On the other hand, races on, for example, channel c_1 make Prg behave *non-deterministically*. For instance, when placed in the context of two outputs on c_1 , such as $c_1 ! 1 \parallel c_2 ! v_2 \parallel c_1 ! 3$, we have a race for the processing of Prg yielding two possible outcomes;

$$\begin{aligned} Prg \parallel c_1 ! 1 \parallel c_2 ! v_2 \parallel c_1 ! 3 &\Downarrow c_4 ! \parallel c_1 ! (1, 2) \parallel c_1 ! 3 && \text{or;} \\ Prg \parallel c_1 ! 1 \parallel c_2 ! v_2 \parallel c_1 ! 3 &\Downarrow c_4 ! \parallel c_1 ! (3, 6) \parallel c_1 ! 1 \end{aligned}$$

More subtly, $Prg \parallel c_1 ! 1 \parallel c_2 ! v_2 \parallel c_1 ! 3$ may also behave in unexpected ways, since we have a second race condition when channel c_1 is reused internally in Prg , i.e., when Dbl sends back its answer to $Fltr$ on c_1 , thereby obtaining

$$\begin{aligned} Prg \parallel c_1 ! 1 \parallel c_2 ! v_2 \parallel c_1 ! 3 &\Downarrow c_4 ! \parallel c_1 ! (1, 3) \parallel c_1 ! 2 && \text{or;} \\ Prg \parallel c_1 ! 1 \parallel c_2 ! v_2 \parallel c_1 ! 3 &\Downarrow c_4 ! \parallel c_1 ! (3, 2) \parallel c_1 ! 6 \end{aligned}$$

When placed in the context of two outputs on c_1 with values that are less than 10 and also values that are bigger or equal to 10, such as $c_1 ! 1 \parallel c_2 ! v_2 \parallel c_1 ! 10$, non-deterministic behaviour varies even more widely in structure. In fact we can have:

$$\begin{aligned} Prg \parallel c_1 ! 1 \parallel c_2 ! v_2 \parallel c_1 ! 10 &\Downarrow c_4 ! \parallel c_1 ! (1, 2) \parallel c_1 ! 10 && \text{or;} \\ Prg \parallel c_1 ! 1 \parallel c_2 ! v_2 \parallel c_1 ! 10 &\Downarrow c_4 ! 10 \parallel (\text{new } c_3) (c_3 ? x_4 . c_1 ! (x_4 + x_4)) \parallel c_1 ! 1 \end{aligned}$$

Dually, when Prg is placed in the context of $c_1!1 \parallel c_2!v_2 \parallel c_1?x.nil$, which introduces another input competing for the output on c_1 , we have even more non-deterministic behaviour. We can have:

$$\begin{aligned} Prg \parallel c_1!1 \parallel c_2!v_2 \parallel c_1?x.nil &\Downarrow c_4! \parallel c_1!(1, 2) \parallel c_1?x.nil && \text{or;} \\ Prg \parallel c_1!1 \parallel c_2!v_2 \parallel c_1?x.nil &\Downarrow (\text{new } c_3)(Filtr \parallel c_3?x_4.c_1!(x_4 + x_4)) && \text{or even;} \\ Prg \parallel c_1!1 \parallel c_2!v_2 \parallel c_1?x.nil &\Downarrow c_1?x_3.(c_1!(1, x_3) \parallel c_4!) \end{aligned}$$

In practice, a substantial body of concurrent code is expected to behave deterministically under some form of non-interference assumptions. One example is the in-place quicksort algorithm, which can be encoded in our language as shown in Example 2.8. In this example, determinism is even harder to ascertain because, apart from channel reuse, the code is also recursively defined. This gives us scope for developing refined analysis techniques for deterministic code which lend themselves better to compositionality.

Example 2.8 (In-Place Quicksort). The process definition $Qck(i, j)$ defines a quicksort algorithm, sorting arrays of values *in-place* and signalling on channel r once sorting completes. Arrays of integers $a = [v_1, \dots, v_n]$ are represented as a set of messages $a_1!v_1 \parallel \dots \parallel a_n!v_n$ on an indexed set of channels $a_1 \dots a_n$.² When arrays are of length 1, $Qck(i, i)$ signals immediately on channel r . Otherwise, it chooses the value at the lowest index, $a_i!v_i$, as the pivot, partitions the array, and then calls quicksort recursively on the two partitions, renaming the returning signal to a fresh channel name in each case. Once the two sub-sortings signal back, the process can signal back on r .

$$Qck(i, j) \triangleq \begin{cases} \text{if } i = j \text{ then } r! \\ \text{else } (\text{new } r_3) \left(\begin{array}{l} Prt(i, j)[r_3/r] \\ \parallel r_3?x.(\text{new } r_1, r_2) \left(\begin{array}{l} Qck(i, x-1)[r_1/r] \\ \parallel Qck(x+1, j)[r_2/r] \\ \parallel r_1?.r_2?.r! \end{array} \right) \end{array} \right) \end{cases}$$

At the heart of quicksort is $Prt(i, j)$, which partitions an array into two sub-arrays separated by a pivot cell, $a_p!v_p$, and signals completion by outputting the partition index as a value, $r!p$. After partitioning completes, the values in the first sub-array (*i.e.*, indexes less than p) are less than v_p and the values of the second sub-array (*i.e.*, indexes greater than p) are bigger or equal to v_p . Partitioning calls the array traversal process $Trv(l, h, x, p, c)$, initialising the pivot value x to v_i , the pivot index p to i , the counter index c to $i+1$ and low and high array boundaries l, h to i and j respectively.

$$Prt(i, j) \triangleq a_i?x. Trv(i, j, x, i, i+1)$$

Traversal loops through the indexes $i+1$ up to h , (6) then (1), comparing their values with the pivot value, (2). If the current value is greater or equal to x , in-place partitioning restores the cell and increments the counter, (3). Otherwise, it increments the pivot index to $p+1$, swaps the current value with the value at the new pivot index, and proceeds to the next index, (4). Since reads are destructive in value passing concurrency, swapping occurs only when the two indexes are distinct, (5). Once traversal exceeds the highest index of the array, (6), the pivot value at the lowest index l is swapped with the value at the current pivot index p and the pivot index is returned as the return

²Since our language can branch on integer values, channel indexing can be encoded.

value $r!p$, (7); again swapping is avoided if these two indexes are the same.

$$\text{Trv}(l, h, x, p, c) \triangleq \left(\begin{array}{l} \text{if } \overbrace{c > h}^{(6)} \text{ then } \overbrace{\text{if } l=p \text{ then } (a_l!x \parallel r!p) \text{ else } a_p?y. (a_l!y \parallel a_p!x \parallel r!p)}^{(7)} \\ \text{else } \overbrace{a_c?y.}^{(1)} \left(\begin{array}{l} \text{if } \overbrace{x \leq y}^{(2)} \text{ then } \overbrace{a_c!y \parallel \text{Trv}(l, h, x, p, c+1)}^{(3)} \\ \text{else } \overbrace{\left(\begin{array}{l} \text{if } \overbrace{c=p+1}^{(5)} \text{ then } (a_c!y \parallel \text{Trv}(l, h, x, p+1, c+1)) \\ \text{else } a_{p+1}?z. \left(\begin{array}{l} a_c!z \parallel a_{p+1}!y \parallel \\ \text{Trv}(l, h, x, p+1, c+1) \end{array} \right) \end{array} \right)}^{(4)} \end{array} \right) \end{array} \right)$$

We note that the splitting of the array during recursive calls in $Qck(i, j)$ in Example 2.8 is *data dependent*, based on the pivot value returned after a call to $Prt(i, j)$. This fact complicates confluence analysis through static techniques such as type systems for resource usage (e.g., [4, 36]). To be able to deal with the refined analysis required for this example, we define a resource-semantics for our processes in Section 3, which does not approximate over data dependent branching. This extended semantics then serves as a model for a resource-aware separation logic for processes, given in Section 4. In Section 5 we then define a compositional proof system for verifying properties in this logic.

3. RESOURCING FOR PROCESSES

We define a reduction semantics for our programs by *confining* their behaviour through linear permissions for channel input and output. This confined-process semantics helps us to reason about deterministic behaviour of processes and lays the foundation for the semantics of the logic to be presented in Section 4. In particular, it (1) gives us a basis for process separation, in terms of the permissions owned by processes, (2) assists race detection, and (3) acts as a narrative as to why a process is deterministic.

3.1. Systems. We start by defining *permission sets*. These are used as logical embellishments to readily track channel usage and detect race conditions through conflicting permission usage.

Definition 3.1 (Permissions). The set of *permissions* is $\text{PERM} \stackrel{\text{def}}{=} \{\downarrow, \uparrow\} \times \text{NAMES}$, where $\downarrow c$ (resp. $\uparrow c$) represents the permission to input (resp. output) on channel c . A *permission-set*, ranged over by the variables ρ, μ , is a subset of permissions, $\rho \subseteq \text{PERM}$.

Permissions are linear in the sense that there is at most one output permission and one input permission per channel. This is not to be confused with linear (resp. affine) assumptions [17] or types [24], which restrict channel usage to exactly (resp. at most) once. In our case, permissions are not consumed once used, but are instead transferred around and reused. Thus, instead of restricting the number of uses of a particular channel, they ensure that, at any stage during computation, there is at most one processes that can output (resp. input) on a particular channel.

Figure 2 defines the syntax and semantics of *systems* of confined processes, $S, T, R \in \text{SYS}$, whereby processes, P , are confined by permission sets, ρ , and denoted as $[P]_\rho$. Systems can also be composed in parallel and their channels can also be scoped.

Confined Processes (Systems)

$$S, T, R ::= \lceil P \rceil_\rho \mid S \parallel T \mid (\text{new } c)S$$

Permission Violation Detection Rules

$$\begin{array}{c} \text{eOUT} \frac{\uparrow c \notin \rho}{\lceil c! \vec{e} \rceil_\rho \longrightarrow_{\text{err}}} \quad \text{eIN} \frac{\downarrow c \notin \rho}{\lceil c? \vec{x}. P \rceil_\rho \longrightarrow_{\text{err}}} \\ \\ \text{ePAR} \frac{S \longrightarrow_{\text{err}}}{S \parallel T \longrightarrow_{\text{err}}} \quad \text{eRES} \frac{S \longrightarrow_{\text{err}}}{(\text{new } c)S \longrightarrow_{\text{err}}} \quad \text{eSTR} \frac{T \equiv S \longrightarrow_{\text{err}}}{T \longrightarrow_{\text{err}}} \end{array}$$

Structural Equivalence Rules

$$\begin{array}{ll} \text{scCOM} & S \parallel T \equiv T \parallel S \\ \text{scNEW} & (\text{new } c) \lceil \text{nil} \rceil_\emptyset \equiv \lceil \text{nil} \rceil_\emptyset \\ \text{scNIL} & S \parallel \lceil \text{nil} \rceil_\emptyset \equiv S \\ \text{scASS} & S \parallel (T \parallel R) \equiv (S \parallel T) \parallel R \\ \text{scSWP} & (\text{new } c)(\text{new } d)S \equiv (\text{new } d)(\text{new } c)S \\ \text{scEXT} & S \parallel (\text{new } c)T \equiv (\text{new } c)(S \parallel T) \quad \text{if } c \notin \text{fn}(S) \end{array}$$

Reduction Rules

$$\begin{array}{c} \text{cTHN} \frac{b \Downarrow tt}{\lceil \text{if } b \text{ then } P \text{ else } Q \rceil_\rho \longrightarrow \lceil P \rceil_\rho} \quad \text{cELS} \frac{b \Downarrow ff}{\lceil \text{if } b \text{ then } P \text{ else } Q \rceil_\rho \longrightarrow \lceil Q \rceil_\rho} \\ \\ \text{cCOM} \frac{\vec{e} \Downarrow \vec{v} \quad \uparrow c \in \rho \quad \downarrow c \in \mu}{\lceil c! \vec{e} \rceil_\rho \parallel \lceil c? \vec{x}. P \rceil_\mu \longrightarrow \lceil P[\vec{v}/\vec{x}] \rceil_{\mu \cup \rho}} \quad \text{cPRC} \frac{K(\vec{x}) \triangleq P \quad \vec{e} \Downarrow \vec{v}}{\lceil K(\vec{e})[\vec{c}/\vec{d}] \rceil_\rho \longrightarrow \lceil P[\vec{v}/\vec{x}][\vec{c}/\vec{d}] \rceil_\rho} \\ \\ \text{cRES} \frac{S \longrightarrow S'}{(\text{new } a)S \longrightarrow (\text{new } a)S'} \quad \text{cPAR} \frac{S \longrightarrow S'}{S \parallel T \longrightarrow S' \parallel T} \quad \text{cSTR} \frac{S \equiv S' \quad S' \longrightarrow T' \quad T' \equiv T}{S \longrightarrow T} \\ \\ \text{cSPL} \frac{}{\lceil P \parallel Q \rceil_{\rho \uplus \mu} \longrightarrow \lceil P \rceil_\rho \parallel \lceil Q \rceil_\mu} \quad \text{cLCL} \frac{}{\lceil (\text{new } c)P \rceil_\rho \longrightarrow (\text{new } c) \lceil P \rceil_{\rho \uplus \{\downarrow c, \uparrow c\}}} \\ \\ \text{cTGH} \frac{\{\downarrow c, \uparrow c\} \cap \rho \neq \emptyset \quad c \notin \text{fn}(P)}{(\text{new } c) \lceil P \rceil_\rho \parallel S \longrightarrow \lceil P \rceil_{\rho \setminus \{\downarrow c, \uparrow c\}} \parallel (\text{new } c)S} \quad \text{cDsc} \frac{\rho \neq \emptyset}{\lceil \text{nil} \rceil_\rho \longrightarrow \lceil \text{nil} \rceil_\emptyset} \end{array}$$

Figure 2: A Permission-Confined CCS

Confinement allows us to define *separation* across systems, $S \perp T$ on the basis of the (visible) permissions owned by a system, Definition 3.2. In what follows, we assume systems of confined processes to always be *well-resourced*, meaning that all confined parallel processes are *separate*, i.e., there is no overlap across owned permission sets, and that permissions are *linear*. System well-resourcing, denoted $\vdash S$, is formalised in Definition 3.4. It can be easily checked statically by induction on the structure of systems.

Definition 3.2 (*(Visibly) Owned Permissions*).

$$\mathbf{prm}(S) \stackrel{\text{def}}{=} \begin{cases} \rho & \text{if } S = \lceil P \rceil_\rho \\ \mathbf{prm}(T) \cup \mathbf{prm}(R) & \text{if } S = T \parallel R \\ \mathbf{prm}(T) \setminus \{\downarrow c, \uparrow c\} & \text{if } S = (\mathbf{new } c)T \end{cases}$$

Definition 3.3 (*Separation*). $S \perp T \stackrel{\text{def}}{=} \mathbf{prm}(S) \cap \mathbf{prm}(T) = \emptyset$

Definition 3.4 (*Well-Resourced System*). A system S is well-resourced, denote as $\vdash S$, if it is included in the least set defined by the following three rules.

$$\begin{array}{c} \text{wPrC} \frac{}{\vdash \lceil P \rceil_\rho} \quad \text{wPAR} \frac{\vdash S \quad \vdash T \quad S \perp T}{\vdash S \parallel T} \quad \text{wRES} \frac{\vdash S}{\vdash (\mathbf{new } c)S} \end{array}$$

Process confinement also facilitates the *detection of races*, which leads to non-deterministic behaviour in the process semantics of Section 2. The judgement $S \longrightarrow_{\text{err}}$, defined by the rules in Figure 2, describes the *detection* of permission violations. As we shall see later on in Section 3.3 and Section 3.4, the absence of permission violations also implies the absence of channel communication races.

The reduction rules in Figure 2 enforce proper permission usage. Rule cCom imposes additional restrictions to rCom of Figure 1: the output process (*resp.* the input process) is required to own the permission to output, $\uparrow c$ (*resp.* input, $\downarrow c$) on channel c . Confined processes cannot arbitrarily create permissions but need to *transfer* them to other processes at specific interaction points (*i.e.*, communication through cCom). The new rules cSpl and cLcl enforce this *resourcing* of permissions: cSpl requires that newly spawned processes *partition* the parent permissions amongst them whereas cLcl ensures that scoped names generate a single pair of input-output permissions for every channel. Note that cSpl is inherently non-deterministic as it does not specify how the permissions are partitioned amongst the parallel processes: *cf.* Section 3.6 for a discussion of this. Rules cThn, cEls, cPrC, cRes, cPar and cStr in Figure 2 are analogous to those in Figure 1. Structural equivalence extends to systems directly with $\lceil \text{nil} \rceil_\emptyset$ as the parallel composition identity.

The rule cDsc allows systems to discard permissions whenever it is clear that they will not be used anymore, whereas cTgh is a convenient rule that allows us to tighten name scoping irrespective of permissions; together with cStr and scNil and scNew it allows us to discard redundant scoping of channels as computation progresses (*cf.* Example 3.33 for an example on how this rule is used.) These last two rules are not essential for determining whether a process is deterministic but help de-clutter extraneous permissions. This enables us to express eventual stable systems more succinctly which, in turn, permits simpler definitions for assertion satisfaction later on in Section 4.

3.2. Dynamic Properties of Systems. Reductions preserve locality. This means that the permissions owned by a process provide a footprint for its reductions and that any process it reduces to will be confined to these permissions. This property is key for compositional reasoning when ensuring that global properties, such as that of being well-resourced, are preserved. For instance, if the system $S \parallel T$ is well-resourced, then by Definition 3.4 it must be the case that the two sub systems are separate *i.e.*, $S \perp T$. If $S \longrightarrow S'$, locality *i.e.*, $\mathbf{prm}(S') \subseteq \mathbf{prm}(S)$ immediately implies that $S' \perp T$ and therefore, that the global system $S' \parallel T$ is still well-resourced. Thus reduction also preserves well-resourcing.

Lemma 3.5 (Locality). $S \longrightarrow T$ implies $\mathbf{prm}(T) \subseteq \mathbf{prm}(S)$

Lemma 3.6 (Resourcing). $\vdash S$ and $S \longrightarrow T$ implies $\vdash T$

(Proof for Lemma 3.6 & Lemma 3.5). The proof is by rule induction on $S \longrightarrow T$. The main cases are:

cCOM: $S = [c!\vec{e}]_\rho \parallel [c?\vec{x}.P]_\mu$, $T = [P\{\vec{v}/\vec{x}\}]_{\rho\cup\mu}$ where $\vec{e} = \vec{v}$. It is immediate that $\mathbf{prm}(S) = \mathbf{prm}(T)$.

Moreover, $\vdash T$ by **wPRC**.

cPAR: We have $S = R_1 \parallel R_2$, $T = R'_1 \parallel R_2$ and $R_1 \longrightarrow R'_1$. Moreover, $\vdash S$ implies $\mathbf{prm}(R_1) \cap \mathbf{prm}(R_2) = \emptyset$, $\vdash R_1$ and $\vdash R_2$. Also recall that $\mathbf{prm}(S) = \mathbf{prm}(R_1) \cup \mathbf{prm}(R_2)$ and that $\mathbf{prm}(T) = \mathbf{prm}(R'_1) \cup \mathbf{prm}(R_2)$.

By $\vdash R_1$, $R_1 \longrightarrow R'_1$ and I.H. we obtain $\vdash R'_1$ and $\mathbf{prm}(R'_1) \subseteq \mathbf{prm}(R_1)$. By, $\mathbf{prm}(R'_1) \subseteq \mathbf{prm}(R_1)$ and $\mathbf{prm}(R_1) \cap \mathbf{prm}(R_2) = \emptyset$ we deduce $\mathbf{prm}(R'_1) \cap \mathbf{prm}(R_2) = \emptyset$ and by $\vdash R'_1$ and $\vdash R_2$ we deduce $\vdash T$. Moreover, by $\mathbf{prm}(R'_1) \subseteq \mathbf{prm}(R_1)$ we obtain $\mathbf{prm}(T) \subseteq \mathbf{prm}(S)$.

cSPL: $S = [P]_\rho \parallel [Q]_{\rho\uplus\mu}$ and $T = [P]_\rho \parallel [Q]_\mu$. $\rho \uplus \mu$ implies $\mathbf{prm}([P]_\rho) \cap \mathbf{prm}([Q]_\mu) = \emptyset$ and since $\vdash [P]_\rho$ and $\vdash [Q]_\mu$ (by **wPRC**), we get $\vdash T$. Moreover $\mathbf{prm}(T) = \mathbf{prm}(S)$.

cDsc: $S = [\text{nil}]_\rho$ and $T = [\text{nil}]_\emptyset$. Trivially, $\vdash T$ (by **wPRC**) and $\mathbf{prm}(T) = \emptyset \subseteq \mathbf{prm}(S)$. \square

Another important property of our resource semantics is that reductions do not hide prior permission violations *i.e.*, permission violations are preserved by reductions. This allows us to ignore intermediary steps during the evaluation of a confined process (*cf.* Definition 3.8) and simply inspect the resulting stable system to determine whether that evaluation resulted in any permission violations. In what follows, we shall refer to evaluations without permission violations as *safe*.

Lemma 3.7 (Violation Preservation). $S \longrightarrow^* T$ and $S \longrightarrow_{\text{err}} T$ implies $T \longrightarrow_{\text{err}}$

Proof. First we show $S \longrightarrow T$ and $S \longrightarrow_{\text{err}} T$ implies $T \longrightarrow_{\text{err}}$ by rule induction on $S \longrightarrow T$. The main cases are:

cCOM: $S = [c!\vec{e}]_\rho \parallel [c?\vec{x}.P]_\mu$ where $\uparrow c \in \rho$ and $\downarrow c \in \mu$. By case analysis, if $S \longrightarrow_{\text{err}}$ then either $[c!\vec{e}]_\rho \longrightarrow_{\text{err}}$ because $\uparrow c \notin \rho$ (by **eOUT**) or $[c?\vec{x}.P]_\mu \longrightarrow_{\text{err}}$ because $\downarrow c \notin \mu$ (by **eIN**); both cases lead to a contradiction.

cPAR: $S = R_1 \parallel R_2$, $T = R'_1 \parallel R_2$ and $R_1 \longrightarrow R'_1$. By $S = R_1 \parallel R_2$, **ePAR**, **eSTR** and **scCOM** we know $S \longrightarrow_{\text{err}}$ because either:

$R_1 \longrightarrow_{\text{err}}$: By $R_1 \longrightarrow R'_1$ and I.H. $R'_1 \longrightarrow_{\text{err}}$ and by $T = R'_1 \parallel R_2$ and **ePAR** we get $T \longrightarrow_{\text{err}}$.

$R_2 \longrightarrow_{\text{err}}$: By $T = R'_1 \parallel R_2$, **ePAR**, **eSTR** and **scCOM** we obtain $T \longrightarrow_{\text{err}}$.

The second part of the proof is by induction on the number n of reductions used *i.e.*, $S \longrightarrow^n T$. \square

3.3. System Determinism. The first two main results of our resource semantics establish that system evaluation is deterministic up-to the terminal permissions owned (*cf.* Theorem 3.11 and Theorem 3.12).

We first lay the ground for these results by giving the following definitions. Systems evaluation in Definition 3.8, $S \Downarrow T$, is limited to safe-stability, $T\checkmark$, and excludes reductions to racy systems. The operation $| - |$ denotes a permission-erasure function whereby $|S|$ returns the process in S stripped of all its confining permissions; it allows us to express equivalence up-to owned permissions in Theorem 3.11. System Convergence, Definition 3.10, is the least set of systems that converge to a stable state (but not necessarily a safe one) and is used for Theorem 3.12.

Definition 3.8 (Safe-Stability and Evaluation).

$$\begin{aligned} S \checkmark &\stackrel{\text{def}}{=} S \not\rightarrow \text{ and } S \not\rightarrow_{\text{err}} \\ S \Downarrow T &\stackrel{\text{def}}{=} \exists T'. S \longrightarrow^* T' \text{ and } T' \checkmark \text{ and } T \equiv T' \end{aligned}$$

Definition 3.9 (Permission Confinement Erasure).

$$|S| \stackrel{\text{def}}{=} \begin{cases} P & \text{if } S = \lceil P \rceil_{\rho} \\ |T| \parallel |R| & \text{if } S = T \parallel R \\ (\text{new } c) |T| & \text{if } S = (\text{new } c)T \end{cases}$$

Definition 3.10 (System Convergence). \Downarrow is the least predicate over systems satisfying the equation

$$S \Downarrow = S \not\rightarrow \text{ or } (\forall T. S \longrightarrow T \text{ implies } T \Downarrow)$$

In conformance with Definition 2.6, by system determinism we understand that (1) no system can evaluate to two distinct safely-stable systems, up-to owned permissions *i.e.*, Theorem 3.11 and that (2) no system can evaluate to a safely-stable system and, at the same time, diverge along a different execution path *i.e.*, Theorem 3.12.

Theorem 3.11 (Evaluation Determinism). $S \Downarrow T_1 \text{ and } S \Downarrow T_2 \text{ implies } |T_1| \equiv |T_2|$

Theorem 3.12 (System Evaluation implies System Convergence). $S \Downarrow \text{ implies } S \Downarrow$

These properties follow, at an intuitive level, from the partial-confluence property, as stated in Lemma 3.13.

Lemma 3.13 (Partial Confluence). $S \longrightarrow T_1 \text{ and } S \longrightarrow T_2 \text{ implies either of the following:}$

- (1) $|T_1| \equiv |T_2|$ or;
- (2) $\exists T_3. T_1 \longrightarrow T_3 \text{ and } T_2 \longrightarrow T_3$

However, the full technical details of the proofs for Theorem 3.11 and Theorem 3.12 are more delicate; on first reading, the reader may skip them and progress to Section 3.4. Before though, we highlight Proposition 3.14, which establishes sufficient and necessary conditions on the structure of safely-stable systems; these conditions will then act as a guiding principle when formulating our logic formulas. In essence, safely stable systems consist of mismatching asynchronous outputs and input-blocked processes composed in parallel, each owning the respective output and input permissions so as not to generate an error.

Proposition 3.14 (Safe-Stability and System Structure).

$$S \checkmark \text{ iff } S \equiv (\text{new } \vec{d}) \left(\parallel_{i=0}^n \lceil c_i ! \vec{e}_i \rceil_{\rho_i} \parallel_{j=0}^m \lceil c'_j ? \vec{x}_j . P_j \rceil_{\mu_j} \right)$$

where

- $\{c_1, \dots, c_n\} \cap \{c'_1, \dots, c'_m\} = \emptyset$
- $\bigwedge_{i=0}^n \uparrow c_i \in \rho_i$
- $\bigwedge_{j=0}^m \downarrow c_j \in \mu_j$

and where $\parallel_{i=0}^n \lceil c_i ! \vec{e}_i \rceil_{\rho_i}$ and $\parallel_{j=0}^m \lceil c'_j ? \vec{x}_j . P_j \rceil_{\mu_j}$ denote $\lceil \text{nil} \rceil_{\emptyset}$.

The proofs for Theorem 3.11 and Theorem 3.12 require us to work at a tighter relation than process structural equivalence for the intermediary steps of an evaluation, namely \approx defined in

Definition 3.15, because process structural equivalence, \equiv , loses information wrt. the currently owned permissions of a system. The relation \cong lies between system structural equivalence and the respective process structural equivalence after confinement erasure (cf. Proposition 3.16.)

Definition 3.15 (Equivalence up-to owned permissions). $S \cong T$ is defined as the least relation satisfying the following rules:

$$\frac{}{\llbracket P \rrbracket_\rho \cong \llbracket P \rrbracket_\mu} \quad \frac{S_1 \cong S_2 \quad T_1 \cong T_2}{S_1 \parallel T_1 \cong S_2 \parallel T_2} \quad \frac{S_1 \cong S_2}{(\text{new } c)S_1 \cong (\text{new } c)S_2} \quad \frac{S_1 \equiv S_2 \cong T_2 \equiv T_1}{S_1 \cong T_1}$$

Proposition 3.16. $S \equiv T$ implies $S \cong T$ implies $|S| \equiv |T|$

Note that $|S| \equiv |T|$ does not imply $S \cong T$. For instance, $|\llbracket P \parallel Q \rrbracket_{\rho \uplus \mu}| \equiv |\llbracket P \rrbracket_\rho \parallel \llbracket Q \rrbracket_\mu|$ but $\llbracket P \parallel Q \rrbracket_{\rho \uplus \mu} \not\cong \llbracket P \rrbracket_\rho \parallel \llbracket Q \rrbracket_\mu$.

Lemma 3.17 (Properties of \cong with respect to reductions).

- (1) $S \cong T$ and $T \longrightarrow T'$ and $S \not\rightarrow_{\text{err}}$ implies $S \longrightarrow S'$ and $S' \cong T'$
- (2) $S \cong T$ and $S \checkmark$ implies $T \not\rightarrow$

Proof. See Appendix A.2 □

The system relation \cong allows us to specify a tighter relationship which characterises more precisely Partial Confluence, i.e., Lemma 3.18. This is then used to prove Lemma 3.21, upon which Theorem 3.11 rests. We here relegate the proofs of Lemmas used by Lemma 3.21 to Appendix A.2. Note also that Lemma 3.13, stated earlier to give an intuition for how linear permissions ensure confluence, follows immediately from Lemma 3.18 and Proposition 3.16.

Lemma 3.18 (Partial Confluence). $S \longrightarrow T_1$ and $S \longrightarrow T_2$ implies either of the following:

- (1) $T_1 \cong T_2$ or;
- (2) $\exists T_3. T_1 \longrightarrow T_3$ and $T_2 \longrightarrow T_3$

Proof. See Appendix A.2 □

Definition 3.19 (System Evaluation Predicates). $S \Downarrow \stackrel{\text{def}}{=} \exists T. S \Downarrow T$

Lemma 3.20 (Evaluation Preservation for \cong).

$$S \cong T \text{ and } S \Downarrow \text{ and } T \longrightarrow T' \text{ implies } S \longrightarrow S' \text{ where } S' \cong T' \text{ and } S' \Downarrow$$

Proof. See Appendix A.2 □

Lemma 3.21 (Evaluation and \cong).

$$S \cong T \text{ and } S \longrightarrow^n S' \checkmark \text{ and } T \longrightarrow^m T' \checkmark \text{ implies } S' \cong T' \text{ and } n = m$$

Proof. By (strong) induction on the number of reductions leading to a safely-stable system from any system $S \longrightarrow^n S'$.

$n = 0$: By $S \not\rightarrow$ and Lemma 3.17(2) we know $T \not\rightarrow$ which implies $m = 0$ and $T' = T \cong S$.

$n = k + 1$: We have

$$\exists S'' \text{ such that } S \longrightarrow S'' \longrightarrow^k S' \tag{3.1}$$

Lemma 3.7 and $S' \checkmark, T' \checkmark$ implies

$$S \not\rightarrow_{\text{err}} \text{ and } T \not\rightarrow_{\text{err}}$$

and $S \rightarrow S''$ and Lemma 3.17(1) implies that $m > 0$ i.e.,

$$\exists T'' \text{ such that } T \rightarrow T'' \xrightarrow{m-1} T' \quad (3.2)$$

Moreover, $S \xrightarrow{n} S' \checkmark$ and $T \xrightarrow{m} T' \checkmark$ imply $S \Downarrow S'$, $T \Downarrow T'$ respectively, and by $S \cong T$, $S \rightarrow S''$ and Lemma 3.20 we obtain

$$\exists T_1, T'_1, l \text{ such that } T \rightarrow T_1 \quad (3.3)$$

$$T_1 \cong S'' \quad (3.4)$$

$$T_1 \xrightarrow{l} T'_1 \checkmark \quad (3.5)$$

By $S'' \xrightarrow{k} S'$ from (3.1), (3.4), (3.5) and I.H. we obtain

$$S' \cong T'_1 \text{ and } l = k \quad (3.6)$$

i.e., $T_1 \xrightarrow{k} T'_1 \checkmark$. Now by Lemma 3.18, (3.3) and $T \rightarrow T''$ from (3.2) we have two sub-cases:

$T_1 \cong T''$: By (3.5) and (3.6) we know $T_1 \xrightarrow{k} T'_1 \checkmark$ and by, $T'' \xrightarrow{m-1} T'$ from (3.2) I.H. we deduce

$$T' \cong T'_1 \text{ and } (m-1) = k$$

and by transitivity and (3.6) we conclude $T' \cong S'$ and $m = (k+1) = n$ as required.

$\exists T_3. T_1 \rightarrow T_3$ **and** $T'' \rightarrow T_3$: We here have two further sub-cases:

$\exists T'_3, h$ such that $T_3 \xrightarrow{h} T'_3 \checkmark$: This implies $T_1 \xrightarrow{h+1} T'_3 \checkmark$ and by (3.1), (3.4) and I.H. we obtain

$$T'_3 \cong S' \text{ and } (h+1) = k \quad (3.7)$$

We also know that $T'' \xrightarrow{h+1} T'_3 \checkmark$ and by (3.7) we obtain $T'' \xrightarrow{k} T'_3 \checkmark$ and, since $T'' \cong T''$ (reflexivity of \cong), using (3.2) and I.H. we obtain

$$T'_3 \cong T' \text{ and } (m-1) = k$$

which, first implies $m = (k+1) = n$ and then, by (3.7), implies $T' \cong S'$ as required.

$T_3 \Downarrow$: By $T_1 \rightarrow T_3$, $T_1 \cong T_1$ (reflexivity of \cong), (3.5) and Lemma 3.20 we know

$$\exists T_4, T'_4, i \text{ such that } T_1 \rightarrow T_4 \quad (3.8)$$

$$T_4 \cong T_3 \quad (3.9)$$

$$T_4 \xrightarrow{i} T'_4 \checkmark \quad (3.10)$$

Similarly, by $T'' \rightarrow T_3$, $T'' \cong T''$, (3.2), $T' \checkmark$ and Lemma 3.20

$$\exists T_5, T'_5, j \text{ such that } T'' \rightarrow T_5 \quad (3.11)$$

$$T_5 \cong T_3 \quad (3.12)$$

$$T_5 \xrightarrow{j} T'_5 \checkmark \quad (3.13)$$

Now (3.8) and (3.10) imply $T_1 \xrightarrow{i+1} T'_4 \checkmark$ and by (3.6), $T_1 \cong T_1$ and I.H. we obtain

$$T'_4 \cong T'_1 \cong S' \text{ and } (i+1) = k \text{ i.e., } T_4 \xrightarrow{k-1} T'_4 \checkmark \quad (3.14)$$

Moreover, (3.9), (3.12) and transitivity imply $T_4 \cong T_5$, and by (3.14), (3.13) and I.H. we obtain

$$T'_5 \cong T'_4 \cong S' \text{ and } j = (k-1) \quad (3.15)$$

By (3.11) and (3.15) we obtain $T'' \xrightarrow{k} T'_5$ and by $T'' \cong T''$, (3.2) and I.H. we obtain

$$T' \cong T'_5 \cong S' \text{ and } (m - 1) = k$$

which also implies $m = (k + 1) = n$ as required. \square

Theorem 3.11 (Evaluation Determinism). $S \Downarrow T_1$ and $S \Downarrow T_2$ implies $|T_1| \equiv |T_2|$

Proof. By reflexivity we know $S \cong S$ and by Lemma 3.21 we know $T_1 \cong T_2$ which, by Proposition 3.16, implies $|T_1| \equiv |T_2|$. \square

Convergence for systems, Theorem 3.12, largely follows from Lemma 3.20 and Lemma 3.21. We prove Theorem 3.12 by generalising the hypothesis to systems related by \cong in Lemma 3.22, so as to make the induction go through.

Lemma 3.22. $S \Downarrow$ and $S \cong T$ implies $T \Downarrow$.

Proof. By induction on n where $S \xrightarrow{n} R\checkmark$ for some witness safely-stable R justifying $S \Downarrow$.

$n = 0$: This means that $S\checkmark$ and thus by Lemma 3.17(2) we have $T \dashrightarrow$ which implies $T \Downarrow$.

$n = k + 1$: We have

$$S \longrightarrow S' \xrightarrow{k} R\checkmark \tag{3.16}$$

We have two sub-cases. If $T \dashrightarrow$ then this trivially implies convergence. Otherwise, if $T \longrightarrow T'$, by Lemma 3.20 we obtain

$$S \longrightarrow S'' \text{ such that } S'' \cong T' \text{ and } S'' \Downarrow \tag{3.17}$$

$S'' \Downarrow$ implies that for some m and R' , $S \xrightarrow{m} R'\checkmark$, and since $S \cong S$, by (3.16) and Lemma 3.21 this implies that $m = k + 1$ which means that $S'' \xrightarrow{k} R'$. Thus by $S'' \cong T'$ from (3.17) and I.H. we obtain $T' \Downarrow$ which implies $T \Downarrow$. \square

Theorem 3.12 (System Evaluation implies System Convergence). $S \Downarrow$ implies $S \Downarrow$

Proof. Immediate by Lemma 3.22 and $S \cong S$. \square

3.4. Process Determinism. The second main batch of results relate system evaluations in our resource semantics with process determinism in the unconstrained semantics of Section 2 (cf. Corollary 3.25). In particular, Theorem 3.23 states that any well-resourced permission allocation S that allows a process $|S|$ to evaluate down to a safely-stable system, T , implies that any evaluation for process $|S|$ - in the unconstrained semantics - corresponds, up to structural equivalence, to this system T stripped of its constraining permissions *i.e.*, $|T| \equiv Q$ whenever $|S| \Downarrow Q$. On the other hand, Theorem 3.24 states that if S evaluates successfully to a safely-stable process, then the corresponding process $|S|$ must be convergent. Together, these two theorems effectively state that finding a single allocation (narrative) S of linear permissions for a process $|S|$ that allows it to evaluate to some T suffices to show that $|S|$ is deterministic in the unconstrained semantics (Corollary 3.25.)

Theorem 3.23 (Process Evaluation Determinism). $S \Downarrow T$, $|S| \Downarrow Q_1$, $|S| \Downarrow Q_2$ implies $Q_1 \equiv Q_2 \equiv |T|$

Theorem 3.24 (Process Convergence). $S \Downarrow$ implies $|S| \Downarrow$

Corollary 3.25. $S \Downarrow$ implies $|S|$ is deterministic.

Proof. By Definition 2.6, Theorem 3.23 and Theorem 3.24. \square

We next discuss in detail the proofs for Theorem 3.23 and Theorem 3.24; the reader may safely skip them on first reading and proceed to Section 3.5.

Theorem 3.23 follows directly from Lemma 3.31, which in turn relies heavily on Lemma 3.28. In essence, this lemma states that a system that evaluates to a safely stable system can match any sequence of reductions (in the unconstrained semantics) of the system stripped of its constraining permission. This lemma is based on Lemma 3.27, which proves the property for the case of a single unconstrained reduction, and also depends on the the property of corrective reductions, Lemma 3.26. This lemma states that any system that can evaluate safely, $S \Downarrow$, is guaranteed to be able to “correct” wrong partitioning of permissions (*cf.* `cSPL` in Figure 2) along a particular reduction path that result in systems that can not evaluate safely. Stated otherwise, this means that there must exist a permission partition that leads to a full evaluation along that particular execution path.

Lemma 3.26 (Corrective Reductions).

$$S \Downarrow \text{ and } S \longrightarrow^n T \text{ and } T \Downarrow \text{ implies } \exists R \text{ such that } S \longrightarrow^n R \text{ and } R \cong T \text{ and } R \Downarrow$$

Proof. Immediate from Lemma A.9 from Appendix A.2 and the fact that $S \cong S$. □

Lemma 3.27 (Reduction Correspondence).

$$S \Downarrow \text{ and } |S| \longrightarrow Q \text{ implies } \exists R \text{ such that } S \longrightarrow^+ R \text{ and } |R| \equiv Q$$

Proof. By rule induction on $|S| \longrightarrow Q$; see Appendix A.2 □

Lemma 3.28 (Multi-step Reduction Correspondence).

$$|S| \longrightarrow^n Q \text{ and } S \Downarrow \text{ implies } S \longrightarrow^{n+m} R \text{ such that } R \Downarrow \text{ and } |R| \equiv Q.$$

Proof. Proof by induction on the number of reduction steps that lead to a stable process $|S| \longrightarrow^n Q$:

$n = 0$: Immediate since $Q = |S|$ and $S \longrightarrow^0 S$ where $S \Downarrow$.

$n = k + 1$: This means that $\exists P$ such that $|S| \longrightarrow P \longrightarrow^k Q$. By $S \Downarrow$ and Lemma 3.27 we know:

$$\exists T \text{ such that } S \longrightarrow^l T, l > 0 \text{ and } |T| \equiv P \tag{3.18}$$

Thus by $P \longrightarrow^k Q$, $|T| \equiv P$ from (3.18) and `rSTR` we have

$$|T| \longrightarrow^k Q \tag{3.19}$$

At this point we have two cases:

$T \Downarrow$: By I.H. implies we deduce that $T \longrightarrow^{k+m} R$ such that $R \Downarrow$ and $|R| \equiv Q$, and by $S \longrightarrow^l T$ from (3.18) we obtain

$$S \longrightarrow^{k+m+l} R \text{ such that } R \Downarrow \text{ and } |R| \equiv Q.$$

$T \Downarrow$: By $S \longrightarrow^l T$ from (3.18) and Lemma 3.26, we know

$$\exists T' \text{ such that } S \longrightarrow^l T' \text{ and } T' \cong T \text{ and } T' \Downarrow \tag{3.20}$$

Now, by Proposition 3.16, $T' \cong T$ and implies $|T'| \equiv |T|$. Thus by (3.19) and `rSTR` we deduce $|T'| \longrightarrow^k Q$. Thus by $T' \Downarrow$ and I.H. we obtain $T' \longrightarrow^{k+m} R$ such that $R \Downarrow$ and $|R| \equiv Q$, and by $S \longrightarrow^l T'$ from (3.20) we obtain $S \longrightarrow^{k+m+l} R$ such that $R \Downarrow$ and $|R| \equiv Q$. □

Lemma 3.31 uses also Lemma 3.30, which maps stable processes to safely stable systems.

Lemma 3.29 (Correspondence). $S \longrightarrow T \text{ implies } |S| \longrightarrow |T| \text{ or } |S| \equiv |T|$

Proof. The proof is by rule induction on $S \longrightarrow T$ and we relegate this to Appendix A.2. □

Lemma 3.30 (Correspondence and Termination). $|S| \not\rightarrow$ and $S \Downarrow T$ implies $|T| \equiv |S|$

Proof. By induction on n where $S \rightarrow^n T$. The inductive case uses the contrapositive of Lemma 3.29. See Appendix A.2 \square

Lemma 3.31 (Evaluation Determinism). $|S| \Downarrow Q$ and $S \Downarrow T$ implies $Q \equiv |T|$.

Proof. $|S| \Downarrow Q$ implies that

$$|S| \rightarrow^n Q \not\rightarrow \text{ for some } n \quad (3.21)$$

By $|S| \rightarrow^n Q$, $S \Downarrow T$ and Lemma 3.28 we know that $S \rightarrow^{n+m} R$ such that $R \Downarrow$ and $|R| \equiv Q$. Since $Q \not\rightarrow$, (3.21), then by Corollary A.2 we obtain $|R| \not\rightarrow$ and thus, by $R \Downarrow$ and Lemma 3.30 we know that

$$R \Downarrow T' \text{ for some } T' \text{ where } |T'| \equiv |R| \quad (3.22)$$

By $S \rightarrow^{n+m} R$ and $R \Downarrow T'$ of (3.22) we deduce that $S \Downarrow T'$ and by $S \Downarrow T$ and Theorem 3.11 from Section 3.3 we know $|T| \equiv |T'|$. Thus by transitivity we obtain $|T| \equiv |T'| \equiv |R| \equiv Q$ as required. \square

Theorem 3.23 (Process Evaluation Determinism). $S \Downarrow T$, $|S| \Downarrow Q_1$, $|S| \Downarrow Q_2$ implies $Q_1 \equiv Q_2 \equiv |T|$

Proof. By Lemma 3.31 we know $Q_1 \equiv |T|$ and $Q_2 \equiv |T|$ and the required result follows by transitivity of \equiv . \square

The theorem relating system evaluation and process convergence uses the following corollary, obtained from Proposition 3.14 of Section 3.3.

Corollary 3.32. $S \not\rightarrow_{err}$ and $S \not\rightarrow$ implies $|S| \not\rightarrow$

Proof. Follows from Proposition 3.14. \square

Theorem 3.24 (Process Convergence). $S \Downarrow$ implies $|S| \Downarrow$

Proof. By contradiction. Assume that $|S| \uparrow$. Since, by $S \Downarrow$ and Theorem 3.12, any reduction sequence starting from S is finite, by $|S| \uparrow$ there must exist a long enough reduction sequence

$$|S| \rightarrow^n Q \rightarrow \dots$$

where, by Lemma 3.28, $S \Downarrow T$ and $|T| \equiv Q$. Now since $T \checkmark$, then by Corollary 3.32 we must have $Q \not\rightarrow$ which contradicts our assumption. Thus $|S| \Downarrow$. \square

3.5. Confined Semantics Application. The following examples expound on the use of linear permission allocations for reasoning about deterministic code.

Example 3.33. $\text{Prg} \parallel c_1!v_1 \parallel c_2!v_2$ can be shown to be deterministic by finding a permission assignment for every process below that permits a safe evaluation.

$$\llbracket \text{Prg} \rrbracket_{\rho_1} \parallel \llbracket c_1!2 \rrbracket_{\rho_2} \parallel \llbracket c_2!5 \rrbracket_{\rho_3} \Downarrow \llbracket c_1!(2,4) \rrbracket_{\mu_1} \parallel \llbracket c_4! \rrbracket_{\mu_2}$$

Two possible assignments for ρ_1 , ρ_2 and ρ_3 that permit the above evaluation are:

$$\rho_1 = \{\downarrow c_1, \downarrow c_2, \uparrow c_4\}, \quad \rho_2 = \{\uparrow c_1\}, \quad \rho_3 = \{\uparrow c_2\} \quad \text{or}; \quad (3.23)$$

$$\rho_1 = \{\downarrow c_1, \downarrow c_2\}, \quad \rho_2 = \{\uparrow c_1, \uparrow c_4\}, \quad \rho_3 = \{\uparrow c_2\} \quad (3.24)$$

Stated otherwise, we have at least two possible linear-permission based narratives explaining why $\text{Prg} \parallel c_1!v_1 \parallel c_2!v_2$ is deterministic. For both assignments $\uparrow c_1 \in \mu_1$ and $\uparrow c_4 \in \mu_2$ must hold for the resulting safely-stable system $\llbracket c_1!(2,4) \rrbracket_{\mu_1} \parallel \llbracket c_4! \rrbracket_{\mu_2}$, but the remaining permissions $\downarrow c_1$, $\downarrow c_2$ and

$\uparrow c_2$, which are redundant at that point, can arbitrarily be split amongst μ_1 and μ_2 . More specifically, recall from Example 2.7 that

$$\begin{aligned} \text{Prg} &\triangleq (\text{new } c_3) (\text{Fltr} \parallel \text{DbI}) & \text{DbI} &\triangleq c_2 ? x_2 . c_3 ? x_4 . c_1 ! (x_4 + x_4) \\ \text{Fltr} &\triangleq c_1 ? x_1 . \text{if } x_1 \leq 9 \text{ then } c_3 ! x_1 \parallel c_1 ? x_3 . (c_1 ! (x_1, x_3) \parallel c_4 !) \text{ else } c_4 ! x_1 \end{aligned}$$

Using the permission assignment in (3.23) we can have the reduction sequence below. Reduction (3.25) can be derived using the rules cLCL, cSTR and cPAR from (cf. Figure 2) whereas reduction (3.26) is derived using cSPL, cPAR and cRES; other reductions can be derived in similar fashion. For the most part, we have abstract away from structural manipulation of terms, with the exception of reduction (3.33) which employs cTGH and cSTR to discard the redundant scoped channel name c_3 and the permissions associated with it.

$$\lceil \text{Prg} \rceil_{\{\downarrow c_1, \downarrow c_2, \uparrow c_4\}} \parallel \lceil c_1 ! 2 \rceil_{\{\uparrow c_1\}} \parallel \lceil c_2 ! 5 \rceil_{\{\uparrow c_2\}} \longrightarrow \quad (3.25)$$

$$(\text{new } c_3) \left(\lceil \text{Fltr} \parallel \text{DbI} \rceil_{\{\downarrow c_1, \downarrow c_2, \uparrow c_4, \uparrow c_3, \downarrow c_3\}} \parallel \lceil c_1 ! 2 \rceil_{\{\uparrow c_1\}} \parallel \lceil c_2 ! 5 \rceil_{\{\uparrow c_2\}} \right) \longrightarrow \quad (3.26)$$

$$(\text{new } c_3) \left(\lceil \text{Fltr} \rceil_{\{\downarrow c_1, \uparrow c_4, \uparrow c_3\}} \parallel \lceil \text{DbI} \rceil_{\{\downarrow c_2, \downarrow c_3\}} \parallel \lceil c_1 ! 2 \rceil_{\{\uparrow c_1\}} \parallel \lceil c_2 ! 5 \rceil_{\{\uparrow c_2\}} \right) \longrightarrow \quad (3.27)$$

$$(\text{new } c_3) \left(\left[\begin{array}{l} \text{if } 2 \leq 9 \text{ then} \\ c_3 ! 2 \parallel c_1 ? x_3 . (c_1 ! (2, x_3) \parallel c_4 !) \\ \text{else } c_4 ! 2 \end{array} \right]_{\{\downarrow c_1, \uparrow c_4, \uparrow c_3, \uparrow c_1\}} \parallel \lceil \text{DbI} \rceil_{\{\downarrow c_2, \downarrow c_3\}} \parallel \lceil c_2 ! 5 \rceil_{\{\uparrow c_2\}} \right) \longrightarrow \quad (3.28)$$

$$(\text{new } c_3) \left(\lceil c_3 ! 2 \parallel c_1 ? x_3 . (c_1 ! (2, x_3) \parallel c_4 !) \rceil_{\{\downarrow c_1, \uparrow c_4, \uparrow c_3, \uparrow c_1\}} \parallel \lceil \text{DbI} \rceil_{\{\downarrow c_2, \downarrow c_3\}} \parallel \lceil c_2 ! 5 \rceil_{\{\uparrow c_2\}} \right) \longrightarrow \quad (3.29)$$

$$(\text{new } c_3) \left(\lceil c_3 ! 2 \rceil_{\{\uparrow c_3, \uparrow c_1\}} \parallel \lceil c_1 ? x_3 . (c_1 ! (2, x_3) \parallel c_4 !) \rceil_{\{\downarrow c_1, \uparrow c_4\}} \parallel \lceil \text{DbI} \rceil_{\{\downarrow c_2, \downarrow c_3\}} \parallel \lceil c_2 ! 5 \rceil_{\{\uparrow c_2\}} \right) \longrightarrow \quad (3.30)$$

$$(\text{new } c_3) \left(\left[\begin{array}{l} \lceil c_3 ! 2 \rceil_{\{\uparrow c_3, \uparrow c_1\}} \parallel \lceil c_1 ? x_3 . (c_1 ! (2, x_3) \parallel c_4 !) \rceil_{\{\downarrow c_1, \uparrow c_4\}} \\ \parallel \lceil c_3 ? x_4 . c_1 ! (x_4 + x_4) \rceil_{\{\downarrow c_2, \downarrow c_3, \uparrow c_2\}} \end{array} \right] \right) \longrightarrow \quad (3.31)$$

$$(\text{new } c_3) \left(\lceil c_1 ? x_3 . (c_1 ! (2, x_3) \parallel c_4 !) \rceil_{\{\downarrow c_1, \uparrow c_4\}} \parallel \lceil c_1 ! (2+2) \rceil_{\{\downarrow c_2, \downarrow c_3, \uparrow c_2, \uparrow c_3, \uparrow c_1\}} \right) \longrightarrow \quad (3.32)$$

$$\begin{aligned} (\text{new } c_3) \left(\lceil c_1 ! (2, 4) \parallel c_4 ! \rceil_{\{\downarrow c_1, \uparrow c_4, \downarrow c_2, \downarrow c_3, \uparrow c_2, \uparrow c_3, \uparrow c_1\}} \right) &\equiv \\ (\text{new } c_3) \left(\lceil c_1 ! (2, 4) \parallel c_4 ! \rceil_{\{\downarrow c_1, \uparrow c_4, \downarrow c_2, \downarrow c_3, \uparrow c_2, \uparrow c_3, \uparrow c_1\}} \parallel \lceil \text{nil} \rceil_{\emptyset} \right) &\longrightarrow \quad (3.33) \end{aligned}$$

$$\lceil c_1 ! (2, 4) \parallel c_4 ! \rceil_{\{\downarrow c_1, \uparrow c_4, \downarrow c_2, \uparrow c_2, \uparrow c_1\}} \parallel (\text{new } c_3) (\lceil \text{nil} \rceil_{\emptyset}) \equiv$$

$$\lceil c_1 ! (2, 4) \parallel c_4 ! \rceil_{\{\downarrow c_1, \uparrow c_4, \downarrow c_2, \uparrow c_2, \uparrow c_1\}} \longrightarrow \quad (3.34)$$

$$\lceil c_1 ! (2, 4) \rceil_{\{\uparrow c_1, \downarrow c_2, \uparrow c_2\}} \parallel \lceil c_4 ! \rceil_{\{\downarrow c_1, \uparrow c_4\}} \not\rightarrow \not\rightarrow_{\text{err}} \quad (3.35)$$

We highlight two important aspects of this reduction sequence. First, reduction (3.30) could have been interleaved with any of the reductions (3.27), (3.28) and (3.29) while still yielding the same safely-stable system; this holds because these reductions are confluent, as the separate permissions held by each subsystem attest. Second, we could have opted for a different permission partitioning in the reductions (3.26), (3.29) and (3.34), and still attained a safely-stable system. For instance, in (3.26) we could have allocated permission $\uparrow c_4$ to the process *DbI* and, similarly, in the case of (3.29) permission $\uparrow c_4$ could have been allocated to the process $c_3 ! 2$, without altering the eventual safely-stable system reached.

From the fact that (3.35) is safely-stable and the contrapositive of Lemma 3.7 we know that permissions were never violated throughout the reduction sequence. Theorem 3.11 guarantees that the process part of any system evaluation will be structurally equivalent to $c_1 ! (2, 4) \parallel c_4 !$ and, by

Theorem 3.23 and Theorem 3.24, this implies that $\text{Prg} \parallel c_1!v_1 \parallel c_2!v_2$ *deterministically* evaluates to $c_1!(2, 4) \parallel c_4!$ *i.e.*, it always converges.

From a compositional perspective, permission-sets also delineate the footprint of every process and, indirectly, the requirement for well-resourcing of Definition 3.4 defines an interface for detecting race conditions. Consider for example the system:

$$\lceil \text{Prg} \rceil_{\{\downarrow c_1, \downarrow c_2, \uparrow c_4\}}$$

In order for this system to be safe, it needs the permission $\downarrow c_1$ (otherwise it would yield a permission violation through rule eIN). Recall the context $c_1!1 \parallel c_2!v_2 \parallel c_1?x.\text{nil}$ from Example 2.7 which had introduced a race condition on inputs on channel c_1 . In order for this system not to violate permissions itself, it must own a permission set μ *i.e.*, $\lceil c_1!1 \parallel c_2!v_2 \parallel c_1?x.\text{nil} \rceil_\mu$, where $\downarrow c_1 \in \mu$ as well. However, the separation condition for well-resourcing prohibits us from composing these two systems together because their respective permissions are not disjoint *i.e.*, $\{\downarrow c_1, \downarrow c_2, \uparrow c_4\} \not\perp \mu$.

Example 3.34. If, in the array $a_1!v_1 \parallel \dots \parallel a_n!v_n$ to be sorted, we assign the permission set $\mu_i = \{\uparrow a_i\}$ to every element $a_i!v_i$ and assign the permission set $\rho = \{\downarrow a_1, \dots, \downarrow a_n, \uparrow r\}$ to $\text{Qck}(1, n)$ then it turns out that we can show that

$$\lceil \text{Qck}(1, n) \rceil_\rho \parallel \lceil a_1!v_1 \rceil_{\mu_1} \parallel \dots \parallel \lceil a_n!v_n \rceil_{\mu_n} \Downarrow T$$

for some safely stable system T where

$$T \equiv \lceil a_1!u_1 \rceil_{\mu_1} \parallel \dots \parallel \lceil a_n!u_n \rceil_{\mu_n} \parallel \lceil r! \rceil_\rho$$

Note how, as in Example 3.33, ρ in $\lceil \text{Qck}(1, n) \rceil_\rho$ defines an interface that parallel processes to be composed with it to respect, in order for it to evaluate deterministically.

3.6. Discussion. Process spawning, cSPL , is intentionally non-deterministic: apart from alleviating permission annotation,³ its non-deterministic nature is in line with the unspecified way that permissions can be allocated in a confined system. Correspondingly, through Theorem 3.11 and Corollary 3.25, we have seen how there may be more than one way how to validly distribute permissions across processes so as to prove determinacy.

Since we eventually plan to use confined processes as part of the model for our logic (*cf.* Section 4), we here opt for the most flexible solution *i.e.*, non-deterministic splits for parallel composition, which permits more narratives explaining process determinism while still restricting the permission allocations that can be used. This setup gives better separation of concerns between confined process reduction and the model used for our logic. In particular, this model incorporates environments describing permission-transfer invariants, apart from confined processes. These environments are however orthogonal to the properties of confined processes derived in this section. In fact, their purpose is that of allowing for better compositional analysis when determining assertion satisfactions, as we shall see in Section 4 and Section 5.

³ The current formulation leads to a more lightweight form of annotation for confined processes. The other alternative would have been to extend the definition of parallel composition at the process level and have systems of the form $\lceil P \parallel_{(\mu_1, \mu_2)} Q \rceil_\rho$, whereby μ_1 and μ_2 specify *deterministically* how ρ is to be apportioned amongst P and Q .

4. Logic

We define a separation-based logic that enables us to reason about programs that deterministically evaluate to stable systems satisfying assertions describing their state. Our logic concentrates more on describing data held at asynchronous outputs in stable systems, and abstracts away from issues dealing with control for deterministic evaluation. For this reason, the logic semantics is not defined directly on bare processes. Instead, the confined processes of Section 3 together with the definitions for safe-stability and evaluations, Definition 3.8, provide the basis for a model to our separation logic whereby the permissions owned constitute our units of separation (*cf.* Definition 3.3). Together with the associated proof system of Section 5, this amounts to our proposal for a logical framework for reasoning over non-interfering concurrent programs.

4.1. Permission Environments. In our logic, channels have a dual role. Apart from acting as a mechanism for communicating data, they also act as delimiters of *mutual-exclusion groups of resources*, modeling condition-critical regions[28]. Each input process $c?\vec{x}.P$ abides to use certain permissions in P only *after* it synchronises on channel c whereas each output-process $c!\vec{e}$ obliges to own the permissions *guarded* by c ; these guarded permissions are transferred *dynamically* upon communication on c using rule cCom of Figure 2 and enable us to reason about channel reuse in deterministic systems.

The invariants relating to permission mutual-exclusion are characterised as *permission environments*, $\Gamma \in \text{CHANS} \rightarrow \mathcal{P}(\text{PERM})$, partial maps associating channels c to permission-sets ρ . They require abiding processes to own all the permissions in ρ when outputting on c and, dually, allow processes to assume the acquisition of all permissions in ρ when inputting on c . The constraints in Definition 4.1 ensure that (1) permission transfer always includes the permission $\uparrow c$ to output over the communicating channel, but never the capability $\downarrow c$ to input over it, as this must already belong to the receiving process; (2) environments are suitably closed.

Definition 4.1 (*Permission Environment*). Γ is a finite map from names to permission sets such that:

- (1) forall $c \in \text{dom}(\Gamma)$ $\downarrow c \notin \Gamma(c)$ and $\uparrow c \in \Gamma(c)$,
- (2) $\rho \in \text{cod}(\Gamma)$ implies $\text{nm}(\rho) \subseteq \text{dom}(\Gamma)$,

where $\text{nm}(\rho) \stackrel{\text{def}}{=} \{c \mid \downarrow c \in \rho \text{ or } \uparrow c \in \rho\}$.

4.2. Logical Formulas. Our logic formulas, ranged over by the meta-variables φ, ψ , characterise a ‘spatial’ notion of *state* for deterministic processes in terms of the data held on asynchronous channels at stable processes. In order to simplify our conceptual process interpretations, we limit ourselves to describing only the states of stable processes, abstracting away from the intermediary reductions that lead to stability. For this we require asynchronous output data assertions, $c\langle\vec{e}\rangle$, the ‘separated conjunction’, $\varphi * \psi$, and its unit, **emp**; formulas constructed using just these constructs are denoted by the metavariable χ and are called *state* formulas. Guided by Proposition 3.14, stability requires our formulas to describe (input) blocked processes, **blk**(c). Finally, we also describe unrestricted terminating process by **any** whenever we want to abstract away completely from the structure of a terminating process.

Definition 4.2 (Formulas).

$$\begin{aligned} \chi, \eta \in \text{SFRM} &::= \mathbf{emp} \mid c\langle\vec{e}\rangle \mid \chi * \chi \\ \varphi, \psi \in \text{FRM} &::= \mathbf{emp} \mid \mathbf{any} \mid c\langle\vec{e}\rangle \mid \mathbf{blk}(c) \mid \varphi * \varphi \end{aligned}$$

$\Gamma, S \models \mathbf{emp}$	iff $S \Downarrow [\mathbf{nil}]_\emptyset$;
$\Gamma, S \models \mathbf{any}$	iff $S \Downarrow T$;
$\Gamma, S \models c\langle \vec{e} \rangle$	iff $S \Downarrow [c!e^\vec{e}]_\rho$ with $\vec{e} \Downarrow \vec{v}$, $e^\vec{e} \Downarrow \vec{v}$ and $\Gamma(c) \subseteq \rho$;
$\Gamma, S \models \mathbf{blk}(c)$	iff $S \Downarrow (\mathbf{new} \vec{d}) [c?\vec{x}.P]_\rho$ with $c \notin \vec{d}$ and $c \in \mathbf{dom}(\Gamma)$;
$\Gamma, S \models \varphi_1 * \varphi_2$	iff $S \Downarrow (\mathbf{new} \vec{d}) (S_1 \parallel S_2)$ with $\vec{d} \notin \mathbf{dom}(\Gamma)$ and $\Gamma, S_1 \models \varphi_1$ and $\Gamma, S_2 \models \varphi_2$;

Figure 3: Formula Satisfaction

Our formulas are interpreted over permission environments and well-formed systems, *i.e.*, $\Gamma, S \models \varphi$. They are defined in Figure 3, inductively on the structure of *closed* formulas *i.e.*, formulas with no free variables in the expressions \vec{e} of $c\langle \vec{e} \rangle$. Our definition of formula satisfaction relies heavily on the evaluation judgement, $S \Downarrow T$, which is only defined for closed systems (Definition 3.8); recall that system evaluation *existentialises* over a reduction path leading to a stable system .

The satisfaction relation in Figure 3 describes the state of a system once it stabilises. The main assertion satisfaction is that for data assertions, $c\langle \vec{e} \rangle$, as it relates the data held on asynchronous outputs of a stable system with the data stated in the assertion. To do this, the definition relies on the assumption that S is closed to establish the equality between the two expressions \vec{e} and $e^\vec{e}$. Moreover, it uses the environment, Γ , to ensure that the (stable) asynchronous output owns the permissions imposed by the permission guarding invariants. Its use has already been discussed in Section 4.1 and will be elaborated further when we consider compositional analysis of satisfaction in Section 5. Data assertions are typically composed together using the separating conjunction assertion, $\varphi_1 * \varphi_2$, and the empty assertion, **emp**. For the satisfaction for **emp**, the system $[\mathbf{nil}]_\emptyset$ is chosen to be the identity interpretation for our model *wrt.* separation, thereby making the interpretation for just these constructs a commutative monoid (*cf.* Lemma 4.9).

The satisfaction definition of the separating conjunction, $\varphi_1 * \varphi_2$, is however more complicated than one would have expected, as it needs to handle conjunctions with **blk**(c) and **any** formulas as well; the interpretation for the latter two formulas is rather straightforward. Thus, apart from relying on the system well-resourcing assumption to guarantee that the partitioned sub-systems are separate, $S_1 \perp S_2$ (*cf.* Definition 3.3), satisfaction for the separating conjunction also enforces that a system is stable before it is split, *i.e.*, $S \Downarrow S_1 \parallel S_2$. This condition rules out systems whose subcomponents satisfy the sub-formulas of a conjunction $\varphi_1 * \varphi_2$, but then violate stability once composed together; we return to this later in Example 4.4. The fact that separating conjunction ranges over input-blocked processes also requires a satisfaction definition that ignores scoping of channel names across separation *i.e.*, $S \Downarrow (\mathbf{new} \vec{d}) (S_1 \parallel S_2)$; these scoped names \vec{d} refer to channels used in the continuations of blocked processes, as explained later in Example 4.3, and cannot be abstracted away using structural equivalence rules such as **scExt** and **scNew** from Figure 2.

Example 4.3 (Satisfiability). Recall the process definitions

$$Prg \triangleq (\text{new } c_3) (Fltr \parallel Dbl)$$

$$Dbl \triangleq c_2?x_2.c_3?x_4.c_1!(x_4+x_4)$$

$$Fltr \triangleq c_1?x_1.\text{if } x_1 \leq 9 \text{ then } c_3!x_1 \parallel c_1?x_3.(c_1!(x_1, x_3) \parallel c_4!) \text{ else } c_4!x_1$$

from Example 2.7. Assuming the environment

$$\Gamma = c_1:\{\uparrow c_1\}, c_2:\{\uparrow c_2\}, c_4:\{\uparrow c_4, \downarrow c_1\}$$

we have the following satisfactions:

$$\Gamma, [Prg]_{\{\downarrow c_1, \downarrow c_2, \uparrow c_4\}} \parallel [c_1!2]_{\{\uparrow c_1\}} \parallel [c_2!5]_{\{\uparrow c_2\}} \models c_1\langle 2, 4 \rangle * c_4\langle \rangle \quad (4.1)$$

$$\Gamma, [Prg \parallel c_1!2 \parallel c_2!5]_{\{\downarrow c_1, \downarrow c_2, \uparrow c_4, \uparrow c_1, \uparrow c_2\}} \models c_1\langle 2, 4 \rangle * c_4\langle \rangle \quad (4.2)$$

$$\Gamma, [c_1!(5-3, 3+1)]_{\{\uparrow c_1\}} \parallel [c_4!]_{\{\uparrow c_4, \downarrow c_1\}} \models c_1\langle 2, 4 \rangle * c_4\langle \rangle \quad (4.3)$$

whereby, according to the definition in Figure 3, satisfaction is only concerned with the existence of a reduction path to a stable system, where the outputs corresponding to data assertions are required to own the permissions expected by permission environment Γ ; the reduction path (4.1) and (4.2) has already been discussed in Example 3.33. Satisfaction for (4.3) is more straightforward to determine as the system is stable. On the other hand, for Γ defined above, the following do not satisfy their respective assertions:

$$\Gamma, [Prg]_{\{\downarrow c_1, \downarrow c_2, \uparrow c_4\}} \parallel [c_1!2]_{\emptyset} \parallel [c_2!5]_{\{\uparrow c_2\}} \not\models c_1\langle 2, 4 \rangle * c_4\langle \rangle \quad (4.4)$$

$$\Gamma, [Prg]_{\{\downarrow c_2, \uparrow c_4\}} \parallel [c_1!2]_{\{\uparrow c_1\}} \parallel [c_2!5]_{\{\uparrow c_2\}} \not\models c_1\langle 2, 4 \rangle * c_4\langle \rangle \quad (4.5)$$

$$\Gamma, [\{c_1!\}(5-3, 3+1)]_{\{\uparrow c_1\}} \parallel [c_4!]_{\{\uparrow c_4\}} \not\models c_1\langle 2, 4 \rangle * c_4\langle \rangle \quad (4.6)$$

$$\Gamma, [c_1!(2, 3)]_{\{\uparrow c_1\}} \parallel [c_4!]_{\{\uparrow c_4, \downarrow c_1\}} \not\models c_1\langle 2, 4 \rangle * c_4\langle \rangle \quad (4.7)$$

The first two systems fail to satisfy the assertion because they cannot evaluate to safely-stable systems due to lack of permission. In particular, in (4.4) process $c_1!2$ does not own permission $\uparrow c_1$ required for communication (cf. $cCom$ in Figure 2) whereas in (4.5) Prg is missing permission $\downarrow c_1$. The third system, (4.6), fails to satisfy the assertion although it is already a safely-stable system, as it violates the permission obligations for outputs imposed by Γ i.e., output $c_4!$ does not own permission $\downarrow c_1$. Finally, the fourth system (4.7) fails to satisfy the assertion due to a mismatch between the data expected by the assertions and the data communicated by the outputs. We also have the following satisfactions involving the other assertion forms of the logic:

$$(\Gamma, c_3:\{\uparrow c_3\}), [Fltr \parallel Dbl]_{\{\downarrow c_1, \downarrow c_2, \uparrow c_4, \downarrow c_3\}} \parallel [c_1!10]_{\{\uparrow c_1\}} \parallel [c_2!5]_{\{\uparrow c_2\}} \models c_4\langle 10 \rangle * \mathbf{blk}(c_3) \quad (4.8)$$

$$\Gamma, [Prg]_{\{\downarrow c_1, \downarrow c_2, \uparrow c_4\}} \parallel [c_1!10]_{\{\uparrow c_1\}} \parallel [c_2!5]_{\{\uparrow c_2\}} \models c_4\langle 10 \rangle * \mathbf{any} \quad (4.9)$$

$$\Gamma, [Prg]_{\{\downarrow c_1, \downarrow c_2, \uparrow c_4\}} \parallel [c_1!10]_{\{\uparrow c_1\}} \parallel [c_2!5]_{\{\uparrow c_2\}} \models \mathbf{any} \quad (4.10)$$

$$\Gamma, (\text{new } c_3) ([c_1?.c_3!]_{\{\downarrow c_1\}} \parallel [c_2?.c_3?.nil]_{\{\downarrow c_2\}}) \models \mathbf{blk}(c_1) * \mathbf{blk}(c_2) \quad (4.11)$$

Satisfaction (4.8) requires us to extend Γ to account for the permission invariants of channel c_3 , which is not scoped. We also need the input permission $\downarrow c_3$ as dictated by the satisfaction of the subassertion $\mathbf{blk}(c_3)$ in Figure 3. In the subsequent satisfaction, (4.9), \mathbf{any} is used to describe the input-blocked process on a scoped channel c_3 that is scoped in Prg (recall that $Prg \triangleq (\text{new } c_3) (Fltr \parallel Dbl)$). Note also how, in (4.11), since $c_3 \notin \mathbf{dom}(\Gamma)$ (cf. satisfaction for $\varphi_1 * \varphi_2$ in Figure 3), the scoping of c_3 does not prohibit us from splitting the system to determine the satisfaction of the subcomponents of the formula i.e., $\mathbf{blk}(c_1)$ and $\mathbf{blk}(c_2)$.

The requirement that satisfaction is limited to *safe* evaluations in Figure 3 intentionally makes certain formulas unsatisfiable. Alternative definitions could have been possible whereby we allow systems to temporarily satisfy a formula but then fail to satisfy it as computation progresses, meaning that the eventual stable system would not necessarily satisfy the formula. However, as discussed briefly in the Introduction, in our eventual framework of Section 5, systems will have the dual role of acting both as state as well as state-transformers. We therefore opted for the simpler interpretation that is conceptually easier to work with and chose a satisfaction interpretation that can be easily reasoned about in terms of the eventual stable systems reached.

Example 4.4 (Unsatisfiability). Formulas such as the ones below are unsatisfiable under the interpretation given in Figure 3.

$$c\langle 5 \rangle * c\langle 6 \rangle \qquad c\langle 1 \rangle * \mathbf{blk}(c)$$

In the first case, *i.e.*, $c\langle 5 \rangle * c\langle 6 \rangle$, sub-systems respectively satisfying $c\langle 5 \rangle$ and $c\langle 6 \rangle$ can never be merged into a well-resourced system as they must conflict on the permission $\uparrow c$ irrespective of the narrative chosen, due to the environment constraints set out in Definition 4.1. This is desirable because any system satisfying the first formula will create a race condition for any inputs on the channel c .

In later case, *i.e.*, $c\langle 1 \rangle * \mathbf{blk}(c)$, sub-systems satisfying the sub-formulas of the separating conjunction become *unstable* once they are composed in parallel violating their respective sub-formula satisfaction. Hence any such satisfying system would violate the evaluation condition imposed on the satisfaction of the conjunct formula $\varphi_1 * \varphi_2$ in Figure 3. In fact, any sub-system S_1 satisfying $c\langle 1 \rangle$ must evaluate to a stable system of the form $\lceil c!e \rceil_\rho$ where $e \Downarrow 1$. Similarly any sub-system S_2 satisfying $\mathbf{blk}(c)$ must evaluate to a stable system that is structurally equivalent to $(\mathbf{new} \vec{d}) \lceil c?x.P \rceil_\mu$ (where $c \notin \vec{d}$). This means that, by the semantics of Section 3, $\lceil c!e \rceil_\rho \parallel (\mathbf{new} \vec{d}) \lceil c?x.P \rceil_\mu$ is not stable, even if it is well-resourced (*i.e.*, $\rho \cap \mu = \emptyset$). Our satisfaction definition $\varphi_1 * \varphi_2$ rules out this possibility by first requiring the composite system evaluates to a stable system before splitting. There are two reasons for this stricter interpretation. First, once the reduction happens leading to an evaluation to some other stable state S_3

$$\lceil c!e \rceil_\rho \parallel (\mathbf{new} \vec{d}) \lceil c?x.P \rceil_\mu \longrightarrow (\mathbf{new} \vec{d}) \left[P \parallel 1/x \right]_{\mu \cup \rho} \Downarrow S_3$$

it may be the case that S_3 does not satisfy $c\langle 1 \rangle * \mathbf{blk}(c)$ anymore. Second, and perhaps more importantly, the above reduction can potentially trigger permission-violating or non-terminating behaviour in $(\mathbf{new} \vec{d}) \left[P \parallel 1/x \right]_{\mu \cup \rho}$. For instance, process P may be of the form $d!1 \parallel d!2 \parallel d?y.c!(x+y)$ *i.e.*, it has two competing outputs on channel d . This implies that, whereas $(\mathbf{new} \vec{d}) \lceil c?x.P \rceil_\mu$ is safely-stable, its continuation is permission-violating, irrespective of the permissions held at that point, because it can hold at most one permission to output on channel d .

Since structural equivalence is central to Definition 3 (\Downarrow in Definition 3.8 incorporates it), satisfaction abstracts over structurally equivalent systems, which allows us to work up-to structural equivalence when reasoning about systems. Moreover, we can also reason about formula satisfaction from existing system-formula satisfaction and systems that reduce (converge) to them in zero or more steps.

Proposition 4.5 (Satisfaction and Evaluation). $\Gamma, S \models \varphi$ implies $\exists T. S \Downarrow T$ and $\Gamma, T \models \varphi$

Proposition 4.6 (Structural Eq. and Satisfaction). $\Gamma, S \models \varphi$ and $S \equiv T$ implies $\Gamma, T \models \varphi$

Proposition 4.7 (Satisfaction and Convergence). $\Gamma, S \models \varphi$ and $T \longrightarrow^* S$ implies $\Gamma, T \models \varphi$

We overload \models to denote semantic implication amongst formulas in standard fashion. We then are able to prove certain properties about our logic, stated in Lemma 4.9.

Definition 4.8 (Semantic Implication). $\varphi \models \psi \stackrel{\text{def}}{=} \Gamma, S \models \varphi \text{ implies } \Gamma, S \models \psi$

Lemma 4.9 (Formula equivalence). The following bidirectional implications hold:

- (1) $\mathbf{emp} * \varphi \models \varphi$
- (2) $\varphi_1 * (\varphi_2 * \varphi_3) \models (\varphi_1 * \varphi_2) * \varphi_3$
- (3) $\varphi * \psi \models \psi * \varphi$

4.3. Composing satisfactions. Recall, from Example 4.4, that the satisfaction of the sub-assertions φ_1 and φ_2 does not necessarily imply the satisfaction of the composite assertion, $\varphi_1 * \varphi_2$. Nevertheless it is possible to determine when it is safe to infer this by analysing the structure of the sub-formulas. This analysis is formalised as the formula separation judgement, denoted as $\varphi_1 \perp \varphi_2$ and defined in Definition 4.10. This judgement relies on the functions $\mathbf{edg}()$ and $\mathbf{trg}()$ to conservatively approximate matching outputs and inputs across sub-systems satisfying the formulas φ_1 , φ_2 and, by prohibiting such matching channel operations, it ensures that no new reductions are introduced when sub-systems are composed in parallel. As a result, sub-systems that satisfy sub-formulas in a separating conjunction formulas must still satisfy the conjunction formula once composed, as stated in Lemma 4.11. This formula separation judgement is used later on by the proof system in Section 5 to circumvent the construction of problematic formulas such as those discussed in Example 4.4.

Definition 4.10 (Formula Edges, Triggers and Separation).

$$\mathbf{edg}(\varphi) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } \varphi = \mathbf{emp} \text{ or } \varphi = \mathbf{blk}(c) \\ \{\uparrow c\} & \text{if } \varphi = c(\vec{e}) \\ \mathbf{edg}(\varphi_1) \cup \mathbf{edg}(\varphi_2) & \text{if } \varphi = \varphi_1 * \varphi_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathbf{trg}(\varphi) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } \varphi = \mathbf{emp} \text{ or } \varphi = c(\vec{e}) \\ \{\uparrow c\} & \text{if } \varphi = \mathbf{blk}(c) \\ \mathbf{trg}(\varphi_1) \cup \mathbf{trg}(\varphi_2) & \text{if } \varphi = \varphi_1 * \varphi_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\varphi \perp \psi \stackrel{\text{def}}{=} \mathbf{edg}(\varphi) \cap \mathbf{trg}(\psi) = \emptyset \quad \wedge \quad \mathbf{edg}(\psi) \cap \mathbf{trg}(\varphi) = \emptyset$$

Lemma 4.11 (Merging Assertions).

$$\Gamma, S \models \varphi \text{ and } \Gamma, T \models \psi \text{ and } S \perp T \text{ and } \varphi \perp \psi \text{ implies } \Gamma, S \parallel T \models \varphi * \psi$$

Proof. See Appendix A.3. □

Note that, for a number of conjunctions, the sub-formulas are trivially separate making formula separation checks superfluous. For instance, \mathbf{emp} is separate from any formula; also state formulas $\chi_1 * \chi_2$ are trivially separate, $\chi_1 \perp \chi_2$ as stated in Proposition 4.12.

Proposition 4.12. *For any environment, Γ , state formulas, χ, η and formula φ we have:*

- (1) $\chi \perp \eta$
- (2) $\varphi \perp \mathbf{emp}$

Proof. Immediate from 4.10 □

5. PROOF SYSTEM

We complete our framework by developing a compositional proof-system for the logic of §4, interpreted according to the satisfaction of Figure 3. Our sequents, inspired by Hoare triples, have the format

$$\Gamma; b \vdash \{\varphi\} S \{\psi\},$$

where S is a well-resourced system, φ and ψ are respectively the pre-condition and post-condition, Γ is a permission environment, and b is a boolean expression defined in Figure 1, now serving as a boolean formula over our value domain. The system, formulas and boolean condition in a sequent are potentially open *i.e.*, that may have free variables. Thus, the meaning of our sequents quantifies over all substitutions, $\sigma \in \text{SUB}$ that make the boolean condition evaluate to true, and also over all systems $T \in \text{Sys}$ which are separate from S and which satisfy the precondition in the following way.

Definition 5.1 (*Sequent satisfaction*).

$$\Gamma, b \models \{\varphi\} S \{\psi\} \stackrel{\text{def}}{=} \forall \sigma, T. b\sigma \Downarrow tt, \Gamma, T\sigma \models \varphi\sigma, T\sigma \perp S\sigma \text{ implies } \Gamma, (T \parallel S)\sigma \models \psi\sigma$$

As in [19], our sequents tease apart auxiliary reasoning about our value domain, since determining the truth (or otherwise) of these boolean formulas is *process-independent*. Such disentangling also allows us to make refined claims about derivations in our system. For instance, if we limit value expressions to Presburger arithmetic, we know that our boolean formula derivations exist and are decidable [30].

We note that our sequents deal with *total-correctness*. Formula satisfaction, defined in Figure 3, centers around system evaluation, $S \Downarrow T$, which *existentially* quantifies over one sequence of system reductions. The strength of what may, at first, seem a rather weak behaviour assertion comes from the determinism properties afforded by our model of confined processes. In fact, Theorem 3.11 (Evaluation Determinism) allows us to extend such behaviour assertions to universal system behaviour, up-to redundant permissions. What we are ultimately interested in however is universal *processes* behaviour. This can then be retrieved in immediate fashion through Definition 5.6 (Process Satisfaction), defined later in Section 5.3, Theorem 3.24 (Process Convergence), and ultimately, Theorem 3.23 (Process Evaluation Determinism).

The proof system, defined by the rules in Figure 4, assumes the derivation judgement $b_1 \models b_2$ between two (possibly open) boolean formulas, with the expected property that

$$\forall \sigma : \text{SUB}. b_1 \models b_2 \text{ and } b_1\sigma \Downarrow tt \text{ implies } b_2\sigma \Downarrow tt$$

Most of the logical rules are rather intuitive and their ‘naturalness’ is, in part, due to the strong substratum provided by process confinement, in terms of absence of races. We have four logical axioms where LNIL , LBlk and LOut deal with stable systems. More precisely, LNIL acts as a wire between the precondition and the postcondition, LFls trivialises proofs with an unsatisfiable boolean condition, LBlk generates input-blocked process assertions, and LOut generates data assertions.

The rule LIn is central to the proof system as it is the only rule that consumes part of the precondition. Together with LOut and LPar they capture process communication in our proof system. In particular, they observe the permission mutual-exclusion invariants dictated by the environment, whereby the side-condition in LOut , *i.e.*, $\Gamma(c) \subseteq \rho$, forces outputs to own the permissions guarded by the mutual exclusion through the side-condition $\Gamma(c) \in \rho$, whereas the premise in LIn permit inputs to assume ownership of these guarded permissions after communication, through the masking of these permissions in the conclusion, *i.e.*, $\rho \setminus \Gamma(c)$. The permission checking side-conditions in

Logical Rules		
$\text{LNIL} \frac{}{\Gamma; b \vdash \{\varphi\} \llbracket \text{nil} \rrbracket_\rho \{\varphi\}}$	$\text{LFLS} \frac{}{\Gamma; \text{false} \vdash \{\varphi\} S \{\psi\}}$	$\text{LBLK} \frac{\downarrow c \in \rho}{\Gamma; b \vdash \{\mathbf{emp}\} \llbracket c? \vec{x}. P \rrbracket_\rho \{\mathbf{blk}(c)\}}$
$\text{LOUT} \frac{\Gamma(c) \subseteq \rho}{\Gamma; b \vdash \{\mathbf{emp}\} \llbracket c! \vec{e} \rrbracket_\rho \{c(\vec{e})\}}$	$\text{LIN} \frac{\downarrow c \in \rho \quad \Gamma; b \vdash \{\varphi\} \llbracket P \llbracket \vec{e}/\vec{x} \rrbracket_\rho \rrbracket_\rho \llbracket S \rrbracket \{\psi\}}{\Gamma; b \vdash \{\varphi * c(\vec{e})\} \llbracket c? \vec{x}. P \rrbracket_{\rho \uparrow \Gamma(c)} \llbracket S \rrbracket \{\psi\}}$	
$\text{LIF} \frac{\Gamma; b_1 \wedge b_2 \vdash \{\varphi\} \llbracket P \rrbracket_\rho \llbracket S \rrbracket \{\psi\} \quad \Gamma; b_1 \wedge \neg b_2 \vdash \{\varphi\} \llbracket Q \rrbracket_\rho \llbracket S \rrbracket \{\psi\}}{\Gamma; b_1 \vdash \{\varphi\} \llbracket \text{if } b_2 \text{ then } P \text{ else } Q \rrbracket_\rho \llbracket S \rrbracket \{\psi\}}$	$\text{LDEF} \frac{K(\vec{x}) \triangleq P \quad \Gamma; b \vdash \{\varphi\} \llbracket P \llbracket \vec{e}/\vec{x} \rrbracket_\rho \llbracket \vec{c}/\vec{d} \rrbracket_\rho \rrbracket_\rho \llbracket S \rrbracket \{\psi\}}{\Gamma; b \vdash \{\varphi\} \llbracket K(\vec{e}) \llbracket \vec{c}/\vec{d} \rrbracket_\rho \rrbracket_\rho \llbracket S \rrbracket \{\psi\}}$	
$\text{LPAR} \frac{\Gamma; b \vdash \{\varphi_1\} S \{\psi_1 * \varphi_3\} \quad \varphi_2 \perp \varphi_3 \quad \Gamma; b \vdash \{\varphi_2 * \varphi_3\} T \{\psi_2\} \quad \psi_1 \perp \psi_2}{\Gamma; b \vdash \{\varphi_1 * \varphi_2\} S \llbracket T \rrbracket \{\psi_1 * \psi_2\}}$	$\text{LSPL} \frac{\Gamma; b \vdash \{\varphi\} \llbracket P \rrbracket_\rho \llbracket \llbracket Q \rrbracket_\mu \rrbracket_\rho \llbracket S \rrbracket \{\psi\}}{\Gamma; b \vdash \{\varphi\} \llbracket P \llbracket Q \rrbracket_{\rho \uparrow \mu} \rrbracket_\rho \llbracket S \rrbracket \{\psi\}}$	
$\text{LRES} \frac{\Gamma; b \vdash \{\varphi\} S \{\psi\}}{\Gamma \setminus \vec{c}; b \vdash \{\varphi\} (\text{new } \vec{c}) S \{\psi \setminus \vec{c}\}}$	$\text{LLCL} \frac{\Gamma; b \vdash \{\varphi\} (\text{new } c) \llbracket P \rrbracket_{\rho \uparrow \{c, \uparrow c\}} \llbracket S \rrbracket \{\psi\}}{\Gamma; b \vdash \{\varphi\} \llbracket (\text{new } c) P \rrbracket_\rho \llbracket S \rrbracket \{\psi\}}$	
Structural Rules		
$\text{LINST} \frac{\Gamma; b \vdash \{\varphi\} S \{\psi\}}{\Gamma; b \llbracket e/x \rrbracket \vdash \{\varphi \llbracket e/x \rrbracket\} S \llbracket e/x \rrbracket \{\psi \llbracket e/x \rrbracket\}}$	$\text{LSUB} \frac{b \vDash x = e \quad \Gamma; b \vdash \{\varphi \llbracket e/x \rrbracket\} S \llbracket e/x \rrbracket \{\psi \llbracket e/x \rrbracket\}}{\Gamma; b \vdash \{\varphi\} S \{\psi\}}$	
$\text{LIMP} \frac{\Gamma; b' \vdash \{\varphi_1\} T \{\psi_1\} \quad b \vDash b' \quad \varphi \vDash \varphi_1 \quad S \equiv T \quad \psi_1 \vDash \psi}{\Gamma; b \vdash \{\varphi\} S \{\psi\}}$	$\text{LREN} \frac{d \notin \mathbf{fn}(\Gamma, \varphi, \psi, S) \quad \Gamma; b \vdash \{\varphi\} S \{\psi\}}{\Gamma \llbracket d/c \rrbracket; b \vdash \{\varphi \llbracket d/c \rrbracket\} S \llbracket d/c \rrbracket \{\psi \llbracket d/c \rrbracket\}}$	

Figure 4: Sequent Rules

the axioms LOUT and LBLK ensure that stable systems are safe; similarly, the permission checking side-condition in LIN ensures that evaluations are also safe - recall that any permission violation is propagated down to the eventual stable system by Lemma 3.7.

The system parallel composition rule (LPAR) is central to our proof system. It is the only rule that allows us to introduce a cut-middle formula in the hypotheses, φ_3 . The asymmetry in the hypotheses of this rule guarantees the existence of a reduction sequence across two independently verified sub-systems since the unidirectional cut disallows mutual dependencies across the premise sequents; this prevents deadlocks and ensures *total* correctness. LPAR also carries two side-conditions, $\psi_1 \perp \psi_2$ and $\varphi_2 \perp \varphi_3$, denoting formula separation, defined in Definition 4.10.

The proof system also has a rule for process parallel composition, (LSPL), which forces a partitioning of permission-resources, analogously to cSPL from Figure 2; similarly, the process scoping

rule (LCL) follows rule cLCL from Figure 2. The system scoping rule (LRES) restricts the permission-guarding invariants relating to the scoped channels and filters assertions blocked by the scoping using the function $\varphi \setminus \vec{c}$, as defined in Definition 5.2 ; in particular this function over-approximates to **any** any message state assertions and input-blocked assertions affected by the name scoping of the restriction. LRES also uses an environment restriction operation $\Gamma \setminus c$ defined in Definition 5.3.

Definition 5.2 (Formula Restriction).

$$\varphi \setminus \vec{c} \stackrel{\text{def}}{=} \begin{cases} d\langle \vec{c} \rangle & \text{if } \varphi = d\langle \vec{c} \rangle \text{ and } d \notin \vec{c} \\ \mathbf{blk}(d) & \text{if } \varphi = \mathbf{blk}(d) \text{ and } d \notin \vec{c} \\ \mathbf{emp} & \text{if } \varphi = \mathbf{emp} \\ (\varphi_1 \setminus \vec{c}) * (\varphi_2 \setminus \vec{c}) & \text{if } \varphi = \varphi_1 * \varphi_2 \\ \mathbf{any} & \text{otherwise} \end{cases}$$

Definition 5.3 (Environment Restriction).

$$\Gamma \setminus c \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } \Gamma = \emptyset \\ \Gamma' \setminus c & \text{if } \Gamma = \Gamma', c : \rho \\ (\Gamma' \setminus c), d : (\rho \setminus \{\downarrow c, \uparrow c\}) & \text{if } \Gamma = \Gamma', d : \rho \text{ and } c \neq d \end{cases}$$

Proposition 5.4. *If Γ is a permission environment then $\Gamma \setminus c$ is as well.*

Proof. It is immediate to check that Definition 4.1 is still observed by $\Gamma \setminus c$, in particular that it is suitably closed (Definition 4.1.2).

The remaining logical rules are fairly straightforward. In the conditional proof rule LIF, the hypotheses on each branch are augmented with the corresponding assertion, as usual in Hoare logics; this mechanism works in pairs with the structural rule LFLs which trivialises the proof obligations on unreachable branches. LDEF completes the treatment of the logical rules in the obvious way. Note that rules LIN and LDEF abuse the substitution notation, extending it from values to (possibly open) expressions.

The proof system also has a number of structural rules. The rule (LINST) permits instantiations of generic sequents whereas (LSUB) permits substitutions of expressions to variables that can be inferred to be equivalent from the sequent boolean expression. The rule LREN renames channel names in sequents; the rule side-condition guarantees that the name d is fresh which make renaming injective. Finally, (LIMP) endows proofs with a basic understanding of structural equivalence, \equiv , and of logical implication, \models .

5.1. Derived Rules. Although LPAR is used extensively when proving properties of parallel communicating processes, it turns out that we often do not require its full power which makes it somewhat cumbersome to use. We therefore derive lightweight versions of LPAR, enabling parallel code to be either logically sequenced thereby focussing on cutting intermediary formulas (LCUT), or else considered totally separate, where composite pre-conditions are assumed to produce composite post-conditions (LSEP). These derived rules require fewer side-conditions relating to formula separation. For instance, LCUT disposes of the side-conditions entirely, and LSEP limits them to one check.

$$\text{LCUT} \frac{\Gamma; b \vdash \{\varphi_1\} S \{\psi\} \quad \Gamma; b \vdash \{\psi\} T \{\varphi_2\}}{\Gamma; b \vdash \{\varphi_1\} S \parallel T \{\varphi_2\}} \quad \text{LSEP} \frac{\Gamma; b \vdash \{\varphi_1\} S \{\psi_1\} \quad \Gamma; b \vdash \{\varphi_2\} T \{\psi_2\} \quad \psi_1 \perp \psi_2}{\Gamma; b \vdash \{\varphi_1 * \varphi_2\} S \parallel T \{\psi_1 * \psi_2\}}$$

For state formula pre and postconditions, an even simpler version LSEP is obtained by Corollary 4.12, *i.e.*, LSEPST , which requires no side-conditions at all.

$$\text{LSEPST} \frac{\Gamma; b \vdash \{\eta_1\} S \{\chi_1\} \quad \Gamma; b \vdash \{\eta_2\} T \{\chi_2\}}{\Gamma; b \vdash \{\eta_1 * \eta_2\} S \parallel T \{\chi_1 * \chi_2\}}$$

The derivations of these lightweight parallel rules are straightforward and use formula semantic implications from Lemma 4.9 together with properties for formula separation from Proposition 4.12; See Appendix A.4.

The output axiom rule LOUT appears frequently in most derivations using our proof system. We find it convenient to formulate another derived rule that facilitates comparisons between the expression outputted by the process and that specified by the state formula, even when these expressions do not syntactically match.

$$\text{LOUTD} \frac{b \models \vec{e}_1 = \vec{e}_2 \quad \Gamma(c) \subseteq \rho}{\Gamma; b \vdash \{\mathbf{emp}\} \lceil c!e_1 \rceil_\rho \{c\langle \vec{e}_2 \rangle\}}$$

Dually, the rule LIN is used frequently to dispose of cut-formulas. However the direct use of this rule can become unwieldy due to necessary system structural manipulations required to get the system in form required by the rule. A more convenient version can be derived that abstracts away from structural equivalence manipulations.

$$\text{LIND} \frac{T \equiv \lceil c?x.P \rceil_{\rho \setminus \Gamma(c)} \parallel S \quad \downarrow c \in \rho \quad \Gamma; b \vdash \{\varphi\} \lceil P \lceil \vec{x} \rceil \rceil_\rho \parallel S \{\psi\}}{\Gamma; b \vdash \{\varphi * c\langle \vec{x} \rangle\} T \{\psi\}}$$

The proofs for these derived rules are straightforward and relegated to Appendix A.4.

Derived rules similar to LIN can be obtained for LDEF , LIFSPL and LCL using an analogous derivation. In Section 6 we shall often abuse this fact and use the derived rule named as the respective proof rule while at the same time abstracting away from structural manipulations.

5.2. Frame Rule. The frame rule embodies local reasoning in separation-based logics [33]. For satisfiable post-conditions, a variant of the frame rule can be derived in our proof system.

$$\text{LFRM} \frac{\Gamma; b \vdash \{\varphi_1\} S \{\varphi_2\} \quad \varphi_2 \perp \psi}{\Gamma; b \vdash \{\varphi_1 * \psi\} S \{\varphi_2 * \psi\}}$$

Moreover, for the special case when the pre and post conditions are state formulas, the frame rule eliminates the need for the side condition as stated below.

$$\text{LFRMST} \frac{\Gamma; b \vdash \{\chi_1\} S \{\chi_2\}}{\Gamma; b \vdash \{\chi_1 * \eta\} S \{\chi_2 * \eta\}}$$

We here show the derivation for the more general version of frame rule, *i.e.*, LFRM , using the proof rules (LNIL), (LPAR) and (LIMPL) and the structural rule $S \parallel \lceil \text{nil} \rceil_\emptyset \equiv S$.

$$\frac{\Gamma; b \vdash \{\varphi_1\} S \{\varphi_2\} \quad \frac{\Gamma; b \vdash \{\psi\} \lceil \text{nil} \rceil_\emptyset \{\psi\} \quad \varphi_2 \perp \psi}{S \equiv S \parallel \lceil \text{nil} \rceil_\emptyset} \text{LNIL}}{\frac{\Gamma; b \vdash \{\varphi_1 * \psi\} S \parallel \lceil \text{nil} \rceil_\emptyset \{\varphi_2 * \psi\} \quad S \equiv S \parallel \lceil \text{nil} \rceil_\emptyset}{\Gamma; b \vdash \{\varphi_1 * \psi\} S \{\varphi_2 * \psi\}} \text{LIMP}} \text{LSEP}$$

Our proof-system is sound with respect to Definition 5.1.

Theorem 5.5 (Soundness). $\Gamma; b \vdash \{\varphi\} S \{\psi\}$ implies $\Gamma, b \models \{\varphi\} S \{\psi\}$.

Proof. By rule induction on $\Gamma; b \vdash \{\varphi\} S \{\psi\}$. We here show the main rules:

LOut: For arbitrary σ, T we have:

$$b\sigma \Downarrow tt \quad (5.1)$$

$$\Gamma, T\sigma \models \mathbf{emp}\sigma \quad (5.2)$$

$$T\sigma \perp \lceil c!\vec{e} \rceil_\rho \sigma \quad (5.3)$$

and the side-condition

$$\Gamma(c) \subseteq \rho \quad (5.4)$$

By Figure 3 and (5.2) we know

$$T\sigma \Downarrow \lceil \text{nil} \rceil_\emptyset \quad (5.5)$$

By (5.3) we know that $T\sigma \parallel \lceil c!\vec{e} \rceil_\rho \sigma$ is well-resourced. Moreover, by (5.5) and cPAR and scNIL of Figure 2 we deduce

$$T\sigma \parallel \lceil c!\vec{e} \rceil_\rho \sigma \longrightarrow^* \lceil \text{nil} \rceil_\emptyset \parallel \lceil c!\vec{e} \rceil_\rho \sigma \equiv \lceil c!\vec{e} \rceil_\rho \sigma \quad (5.6)$$

Clearly, $\lceil c!\vec{e} \rceil_\rho \sigma \not\rightarrow$. Moreover by the conditions imposed on environment mappings in Definition 4.1, we know $\uparrow c \in \Gamma(c)$ and thus by (5.4) we deduce that $\uparrow c \in \rho$ and hence that $\lceil c!\vec{e} \rceil_\rho \sigma \not\rightarrow_{\text{err}}$. As a result, from (5.6) we obtain $T\sigma \parallel \lceil c!\vec{e} \rceil_\rho \sigma \Downarrow \lceil c!\vec{e} \rceil_\rho \sigma$ and for some \vec{v} where $\vec{e}\sigma \Downarrow \vec{v}$ and by (5.4) and Figure 3 we obtain $\Gamma, (T \parallel \lceil c!\vec{e} \rceil_\rho)\sigma \models (c\langle \vec{e} \rangle)\sigma$.

LIn: For arbitrary σ, T we have:

$$b\sigma \Downarrow tt \quad (5.7)$$

$$\Gamma, T\sigma \models (\varphi * c\langle \vec{e} \rangle)\sigma \quad (5.8)$$

$$T\sigma \perp (\lceil c?\vec{x}.P \rceil_{\rho \setminus \Gamma(c)} \parallel S)\sigma \quad (5.9)$$

and the side-condition

$$\downarrow c \in \rho \quad (5.10)$$

By (5.8) and Figure 3 we know

$$T \Downarrow (\mathbf{new} \vec{d})(T_1 \parallel T_2) \quad (5.11)$$

$$\text{where } \vec{d} \notin \mathbf{dom}(\Gamma) \quad (5.12)$$

$$\Gamma, T_1 \models \varphi\sigma \quad (5.13)$$

$$\text{and } \Gamma, T_2 \models c\langle \vec{e} \rangle\sigma \quad (5.14)$$

By $\Gamma, T_2 \models c\langle \vec{e} \rangle\sigma$ and Figure 3 we know

$$T_2 \Downarrow \lceil c!\vec{e} \rceil_\mu \quad \text{where } \vec{e}\sigma \Downarrow \vec{v}, \vec{e}'\sigma \Downarrow \vec{v} \text{ and } \Gamma(c) \subseteq \mu \quad (5.15)$$

By (5.9) we know $T\sigma \perp (\lceil c?\vec{x}.P \rceil_{\rho \setminus \Gamma(c)} \parallel S)\sigma$ is well-resourced and by (5.12) and $\Gamma(c) \subseteq \mu$ of (5.15) we know that $c \notin \vec{d}$ and that $\vec{d} \notin \mathbf{nm}(\mu)$. Thus by (5.11), (5.15) and cPAR, cCOM, (5.12) and scEXT we obtain

$$T\sigma \parallel (\lceil c?\vec{x}.P \rceil_{\rho \setminus \Gamma(c)} \parallel S)\sigma \longrightarrow^* \equiv (\mathbf{new} \vec{d})(T_1) \parallel \lceil c!\vec{e} \rceil_\mu \parallel (\lceil c?\vec{x}.P \rceil_{\rho \setminus \Gamma(c)} \parallel S)\sigma \quad (5.16)$$

$$(\mathbf{new} \vec{d})(T_1) \parallel \lceil c!\vec{e} \rceil_\mu \parallel (\lceil c?\vec{x}.P \rceil_{\rho \setminus \Gamma(c)} \parallel S)\sigma \longrightarrow (\mathbf{new} \vec{d})T_1 \parallel (\lceil c?\vec{x}.P \rceil_{\rho \setminus \Gamma(c)} \parallel S)\sigma \quad (5.17)$$

By (5.16) and Lemma 3.6 we know that $(\text{new } \vec{d})(T_1) \parallel (\llbracket c?x.P \rrbracket_{\rho \setminus \Gamma(c)} \parallel S)\sigma$ is well-resourced, and by $\Gamma(c) \subseteq \mu$ of (5.15) we deduce that

$$(\text{new } \vec{d})T_1 \perp (\llbracket P \rrbracket_{\rho} \parallel S)\sigma \quad (5.18)$$

By (5.13), (5.12) and Lemma A.17 we obtain

$$\Gamma, (\text{new } \vec{d})T_1 \models \varphi\sigma$$

and thus by (5.7), (5.18), the premise $\Gamma; b \vdash \{\varphi\} \llbracket P \rrbracket_{\rho} \parallel S \models \{\psi\}$ and I.H. we obtain

$$\Gamma, (\text{new } \vec{d})T_1 \parallel (\llbracket P \rrbracket_{\rho} \parallel S)\sigma \models \psi\sigma \quad (5.19)$$

By $\vec{e}\sigma \Downarrow \vec{v}$ of (5.15) and Lemma A.16 we get $\Gamma, (\text{new } \vec{d})T_1 \parallel (\llbracket P \rrbracket_{\rho} \parallel S)\sigma \models \psi\sigma$. Moreover by Lemma A.23 we also obtain

$$\Gamma, (\text{new } \vec{d})T_1 \parallel (\llbracket P \rrbracket_{\rho \cup \mu} \parallel S)\sigma \models \psi\sigma$$

Thus by (5.16), (5.17) and Proposition 4.7 we obtain $\Gamma, T\sigma \parallel (\llbracket c?x.P \rrbracket_{\rho \setminus \Gamma(c)} \parallel S)\sigma \models \psi\sigma$ as required.

LPAR: For arbitrary σ, R we have:

$$b\sigma \Downarrow tt \quad (5.20)$$

$$\Gamma, R\sigma \models (\varphi_1 * \varphi_2)\sigma \quad (5.21)$$

$$R\sigma \perp S\sigma \parallel T\sigma \quad (5.22)$$

and side-conditions

$$\varphi_2 \perp \varphi_3 \quad (5.23)$$

$$\psi_1 \perp \psi_2 \quad (5.24)$$

By (5.21) we know

$$R\sigma \Downarrow (\text{new } \vec{c})(R_1 \parallel R_2) \quad (5.25)$$

$$\text{where } \vec{c} \notin \mathbf{dom}(\Gamma) \quad (5.26)$$

$$\Gamma, R_1 \models \varphi_1\sigma \quad (5.27)$$

$$\text{and } \Gamma, R_2 \models \varphi_2\sigma \quad (5.28)$$

By (5.25), (5.22) and Lemma 3.6 we know

$$R_1 \perp R_2 \quad (5.29)$$

$$\text{and } R_1 \perp S\sigma \parallel T\sigma \quad (5.30)$$

$$\text{and } R_2 \perp S\sigma \parallel T\sigma \quad (5.31)$$

By (5.20), (5.27), $R_1 \perp S\sigma$ from (5.30) and I.H. we have $\Gamma, R_1 \parallel S\sigma \models (\psi_1 * \varphi_3)\sigma$ and from the satisfaction definition of Figure 3 we obtain

$$R_1 \parallel S\sigma \Downarrow (\text{new } \vec{d})(S_1 \parallel S_2) \quad (5.32)$$

$$\text{where } \vec{d} \notin \mathbf{dom}(\Gamma) \quad (5.33)$$

$$\Gamma, S_1 \models \psi_1\sigma \quad (5.34)$$

$$\text{and } \Gamma, S_2 \models \varphi_3\sigma \quad (5.35)$$

By (5.29) and (5.31) we know $R_1 \parallel S\sigma \perp R_2$. Thus, by (5.32) and Lemma 3.5 we derive $S_1 \perp R_2$, and by (5.28), (5.35), the rule side-condition (5.23) and Lemma 4.11 we obtain

$$\Gamma, R_2 \parallel S_2 \models (\varphi_2 * \varphi_3)\sigma \quad (5.36)$$

Using (5.29) and (5.32) we can also derive $R_2 \parallel S_2 \perp T\sigma$ and by (5.20), (5.36) and I.H. we derive

$$\Gamma, R_2 \parallel S_2 \parallel T\sigma \models \psi_2\sigma \quad (5.37)$$

By (5.29) and (5.32) we also derive $S_1 \perp (R_2 \parallel S_2 \parallel T)$ and by the rule side-condition (5.24) and Lemma 4.11 we obtain

$$\Gamma, S_1 \parallel R_2 \parallel S_2 \parallel T\sigma \models (\psi_1 * \psi_2)\sigma$$

Thus by (5.26), (5.33) and Lemma A.17 we deduce

$$\Gamma, (\text{new } \vec{c}, \vec{d})(S_1 \parallel R_2 \parallel S_2 \parallel T)\sigma \models (\psi_1 * \psi_2)\sigma \quad (5.38)$$

From (5.25), (5.32), cPAR, cRES, cSTR and scEXT we derive

$$R\sigma \parallel S\sigma \parallel T\sigma \longrightarrow^* \equiv (\text{new } \vec{c})(R_1 \parallel R_2 \parallel S\sigma \parallel T\sigma) \longrightarrow^* \equiv (\text{new } \vec{c}, \vec{d})(S_1 \parallel R_2 \parallel S_2 \parallel T\sigma)$$

and by (5.38), Proposition 4.7 and Proposition 4.6 we obtain $\Gamma, R\sigma \parallel S\sigma \parallel T\sigma \models (\psi_1 * \psi_2)\sigma$ as required. \square

5.3. Process Sequent Satisfaction. We conclude this section with Definition 5.6, which extends sequent satisfaction to processes by assuming the *existence* of a permission environment and the respective permission-set, required by the satisfaction definition of Figure 3. This allows for the possibility of having multiple narratives explaining determinism, and is in line with the “*ownership is in the eye of the asserter*” principle [28].

Definition 5.6 (Process Sequent Satisfaction).

$$b \models \{\varphi\} P \{\psi\} \stackrel{\text{def}}{=} \text{exists } \Gamma, \rho \text{ such that } \Gamma, b \models \{\varphi\} \lceil P \rceil_\rho \{\psi\}$$

Example 5.7. According to 5.6, we can now state that *Prg*, from Example 2.7 satisfies the property

$$x \leq 9 \models \{c_1 \langle x \rangle * c_2 \langle y \rangle\} \text{Prg} \{c_1 \langle x, 2x \rangle * c_4 \langle \rangle\}, \quad (5.39)$$

while abstracting over the narrative as to why *Prg* is deterministic. It can be read as saying that, given two values x and y on channels c_1 and c_2 respectively, *Prg* returns the value of x together with its double on c_1 and a signal on c_4 , provided that the value of x is less than 10. Mirroring the previous discussion in Example 2.7, *Prg* also satisfies the property

$$x > 9 \models \{c_1 \langle x \rangle * c_2 \langle y \rangle\} \text{Prg} \{c_4 \langle x \rangle * \mathbf{any}\}, \quad (5.40)$$

where **any** abstracts over the blocked code $(\text{new } c_3)(c_3 ? x_4 . c_1 !(x_4 + x_4))$, as described earlier in Example 4.3.

We are also in a position to specify the correctness of our quicksort algorithm through some macro definitions for compactness.

Example 5.8 (Specifying Correctness for Parallel Quicksort). The expected behaviour of $Qck(i, j)$ from Example 2.8 can be expressed through the sequent satisfaction

$$\left(\mathbf{ord}(\vec{y}_i^j) \wedge \vec{x}_i^j \doteq \vec{y}_i^j \right) \models \{A_i^j\langle \vec{x}_i^j \rangle\} \quad Qck(i, j) \quad \{A_i^j\langle \vec{y}_i^j \rangle * r\langle \rangle\} \quad (5.41)$$

using the following macro definitions, whereby \vec{x}_i^j denotes lists of variables $x_i \dots x_j$ when $i \leq j$ and the empty-list ϵ otherwise

$$\begin{aligned} A_i^j\langle \vec{x}_i^j \rangle &\stackrel{\text{def}}{=} \begin{cases} \mathbf{emp} & \text{if } i > j \\ a_i\langle x_i \rangle * A_{i+1}^j\langle \vec{x}_{i+1}^j \rangle & \text{if } i \leq j \end{cases} \\ \mathbf{ord}(\vec{x}_i^j) &\stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{if } i = j \\ x_i \leq x_{i+1} \wedge \mathbf{ord}(\vec{x}_{i+1}^j) & \text{if } i < j \end{cases} \\ \vec{x}_i^j \doteq \vec{y}_i^j &\stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{if } i = j \\ \bigvee_{i \leq k \leq j} (\vec{y}_i^j = \vec{y}_i^{k-1} x_i \vec{y}_{k+1}^j) \wedge (\vec{x}_{i+1}^j \doteq \vec{y}_i^{k-1} \vec{y}_{k+1}^j) & \text{if } i < j \end{cases} \end{aligned}$$

The specification of (5.41) above states that when $Qck(i, j)$ is composed with an array of arbitrary values on channels $a_i \dots a_j$, denoted by the assertion macro $A_i^j\langle \vec{x}_i^j \rangle$, it returns another array of values on the same channel list, $A_i^j\langle \vec{y}_i^j \rangle$, together with a signal on channel r denoting completion. Moreover, the values returned are

- (1) ordered, expressed as the predicate $\mathbf{ord}(\vec{y}_i^j)$
- (2) equal, up to reordering, to the original values, expressed as the predicate $\vec{x}_i^j \doteq \vec{y}_i^j$.

6. APPLICATION

We conclude by revisiting the properties stated in Section 5.3 and show how our proof-system can be used to prove properties about them. In Example 6.1 we see how proofs about concurrent code are performed by running through only one possible reduction trace, even when other interleavings are possible. The main appeal of these proofs is however their amenability to *compositionality* as shown in Example 6.2. In this example proof, the behaviour of sub-programs is verified in terms of their pre and post conditions only, without any concern towards external interference from other concurrent code. Independently verified sub-programs are then merged together using LPar (and its variants LCut , LSEP and LSEPST), as long as the sub-programs are separate *wrt.* the permissions that they own.

Example 6.1 (Proving Satisfiability). We prove the specifications (5.39) and (5.40) stated earlier in Example 5.7 by first augmenting the satisfaction specification with an appropriate narrative for determinism as stated in Definition 5.6. One possible narrative is the permission-set $\{\downarrow c_1, \downarrow c_2, \uparrow c_4\}$ together with the permission-transfer invariants

$$\Gamma = c_1 : \{\uparrow c_1\}, c_2 : \{\uparrow c_2\}, c_4 : \{\uparrow c_4, \downarrow c_1\}$$

yielding the system specification

$$\Gamma, x \leq 9 \models \{c_1\langle x \rangle * c_2\langle y \rangle\} \quad [\text{Prg}]_{\{\downarrow c_1, \downarrow c_2, \uparrow c_4\}} \quad \{c_1\langle x, 2x \rangle * c_4\langle \rangle\} \quad (6.1)$$

Another possible narrative is the permission-set $\{\downarrow c_1, \downarrow c_2\}$ and the environment

$$\Gamma' = c_1 : \{\uparrow c_1, \uparrow c_4\}, c_2 : \{\uparrow c_2\}, c_4 : \{\uparrow c_4, \downarrow c_1\}$$

The proof is similar to that of (6.2), where we first apply LDEF , LCL and LRES , which leaves us with the following sequent:

$$\Gamma''; x > 9 \vdash \{c_1\langle x \rangle * c_2\langle y \rangle\} \quad [\text{Ftr} \parallel \text{Dbf}]_{\{\downarrow c_1, \downarrow c_2, \downarrow c_3, \uparrow c_3, \uparrow c_4\}} \quad \{c_4\langle x \rangle * \mathbf{blk}(c_3)\} \quad (6.5)$$

where, this time, we have the premise postcondition obtained as $c_4\langle x \rangle * \mathbf{blk}(c_3) \setminus c_3 = c_4\langle x \rangle * \mathbf{any}$ according to Definition 5.2. Again, similar to the proof for (6.2), we apply LSPL to (6.5) followed by two applications of LDEF for Ftr and Dbf . Then we apply LIN twice for c_1 and c_2 to consume the state formula in the precondition, and then by applying LIF . This time, the rule for conditional gives us a different unreachable branch since $x > 9 \wedge x \leq 9 \Rightarrow \text{false}$. The reachable premise can be proved as follows:

$$\frac{\frac{\Gamma''(c_4) \subseteq \{\downarrow c_1, \uparrow c_1, \uparrow c_3, \uparrow c_4\}}{\text{(emp)} \quad \overline{\{c_4!x\}}_{\{\downarrow c_1, \uparrow c_1, \uparrow c_3, \uparrow c_4\}} \quad \{c_4\langle x \rangle\}} \text{LOUT}}{\vdots \quad \frac{\downarrow c_3 \in \{\downarrow c_2, \downarrow c_3, \uparrow c_2\}}{\text{(emp)} \quad \overline{\{c_3?x_4.c_1!(x_4+x_4)\}}_{\{\downarrow c_2, \downarrow c_3, \uparrow c_2\}} \quad \{\mathbf{blk}(c_3)\}} \text{LBLK} \quad c_4\langle x \rangle \perp \mathbf{blk}(c_3)}{\text{(emp)} \quad \overline{\{c_4!x\}}_{\{\downarrow c_1, \uparrow c_1, \uparrow c_3, \uparrow c_4\}} \parallel \overline{\{c_3?x_4.c_1!(x_4+x_4)\}}_{\{\downarrow c_2, \downarrow c_3, \uparrow c_2\}} \quad \{c_4\langle x \rangle * \mathbf{blk}(c_3)\}} \text{LSEP}}$$

Example 6.2 (Proving Correctness for Parallel Quicksort). To prove the correctness property (5.41) for $Qck(i, j)$, as stated in Example 5.8, we choose a narrative where the environment is

$$\Gamma = a_i: \{\uparrow a_i\}, \dots, a_j: \{\uparrow a_j\}, r: \rho(r, i, j)$$

and $Qck(i, j)$ owns the permission set $\rho(r, i, j)$ defined as

$$\rho(x, i, j) \stackrel{\text{def}}{=} \{\uparrow x, \downarrow a_i, \dots, \downarrow a_j\}.$$

The permissions associated with r express the fact that the array can only be read *after* the signal denoting completion is consumed.

We argue, by induction on $n = j - i$ (where $i \leq j$), that if we show that the following sequent holds for arbitrary i and j ,

$$\Gamma; \left(\mathbf{ord}(\vec{y}_i^j) \wedge \vec{x}_i^j \doteq \vec{y}_i^j\right) \vdash \{A_i^j\langle \vec{x}_i^j \rangle\} \quad [\text{Qck}(i, j)]_{\rho(r, i, j)} \quad \{A_i^j\langle \vec{y}_i^j \rangle * r\langle \rangle\} \quad (6.6)$$

this would imply correctness for $Qck(i, j)$ with the above narrative *i.e.*,

$$\Gamma, \left(\mathbf{ord}(\vec{y}_i^j) \wedge \vec{x}_i^j \doteq \vec{y}_i^j\right) \models \{A_i^j\langle \vec{x}_i^j \rangle\} \quad [\text{Qck}(i, j)]_{\rho(r, i, j)} \quad \{A_i^j\langle \vec{y}_i^j \rangle * r\langle \rangle\}$$

which, by Definition 5.6, would prove the satisfaction (5.41).

For the *base case* of (6.6), *i.e.*, $n = 0$ assuming $i = j$ as part of the sequent boolean expression, we trivially prove the sequent using LIF , the state frame rule, LFRMST , and LOUT as shown below. In what follows, we often elide the sequent environment and boolean condition from our proofs.

$$\frac{\frac{\overline{\text{(emp)} \quad \overline{\{r!\}_{\rho(r, i, j)} \{r\langle \rangle\}}} \text{LOUT}}{\{A_i^j\langle \vec{x}_i^j \rangle\} \quad \overline{\{r!\}_{\rho(r, i, j)} \{A_i^j\langle \vec{x}_i^j \rangle * r\langle \rangle\}}} \text{LFRMST}}{\{A_i^j\langle \vec{x}_i^j \rangle\} \quad \overline{\{r!\}_{\rho(r, i, j)} \{A_i^j\langle \vec{y}_i^j \rangle * r\langle \rangle\}}} \text{LSUB} \quad \frac{\overline{\{A_i^j\langle \vec{x}_i^j \rangle\} \quad \overline{\{\dots\}_{\rho(r, i, j)} \{A_i^j\langle \vec{y}_i^j \rangle * r\langle \rangle\}}} \text{LFLS}}{\{A_i^j\langle \vec{x}_i^j \rangle\} \quad \overline{\{\text{if } i = j \text{ then } r! \text{ else } \dots\}_{\rho(r, i, j)} \{A_i^j\langle \vec{y}_i^j \rangle * r\langle \rangle\}}} \text{LIF}} \text{LDEF} \quad \{A_i^j\langle \vec{x}_i^j \rangle\} \quad [\text{Qck}(i, j)]_{\rho(r, i, j)} \quad \{A_i^j\langle \vec{y}_i^j \rangle * r\langle \rangle\}$$

The *inductive case*, $n+1 = j-i$, i.e., adding $i < j$ to the sequent boolean expression, assumes that the property holds for all $m \leq n$, i.e., all $m < j-i$ (the inductive hypothesis), and follows from proving the following two sequents

$$\Gamma_1; b \vdash \left\{ A_i^j \langle \vec{x}_i^j \rangle \right\} \quad [\text{Pr}t(i, j)]_{\rho(r_3, i, j)} \quad \left\{ \begin{array}{l} A_i^{p-1} \langle \vec{z}_i^{p-1} \rangle * a_p \langle y_p \rangle \\ * A_{p+1}^j \langle \vec{z}_{p+1}^j \rangle * r_3 \langle p \rangle \end{array} \right\} \quad (6.7)$$

$$\Gamma_1; b \vdash \left\{ \begin{array}{l} A_i^{p-1} \langle \vec{z}_i^{p-1} \rangle * a_p \langle y_p \rangle \\ * A_{p+1}^j \langle \vec{z}_{p+1}^j \rangle * r_3 \langle p \rangle \end{array} \right\} \quad \left[\begin{array}{l} r_3 ?x.(\text{new } r_1, r_2) \\ \text{Qck}(i, x-1)[r_1/r] \\ \parallel \text{Qck}(x+1, j)[r_2/r] \\ \parallel r_1?.r_2?.r! \end{array} \right]_{\{\downarrow r_3 \uparrow r\}} \quad \left\{ A_i^j \langle \vec{y}_i^j \rangle * r \langle \rangle \right\} \quad (6.8)$$

where Γ_1 extends Γ with the mapping for r_3 i.e., $\Gamma_1 = \Gamma, r_3 : \rho(r_3, i, j)$ and b is a stronger boolean condition defined as:

$$b = \mathbf{ord}(\vec{y}_i^j) \wedge \vec{x}_i^j \doteq \vec{y}_i^j \wedge \underbrace{(\vec{y}_i^{p-1} \doteq \vec{z}_i^{p-1} \wedge \vec{y}_{p+1}^j \doteq \vec{z}_{p+1}^j)}_{(i)} \wedge \underbrace{(\bigwedge_{k=i}^{p-1} z_k < y_p)}_{(ii)} \wedge \underbrace{(\bigwedge_{k=p+1}^j y_p \leq z_k)}_{(iii)}$$

It requires intermediary lists of values \vec{z}_i^{p-1} and \vec{z}_{p+1}^j , returned by partitioning $\text{Pr}t(i, j)$, to be reorderings of the final values \vec{y}_i^{p-1} and \vec{y}_{p+1}^j , (i), that the values in \vec{z}_i^{p-1} are less than the pivot, (ii), and also that the values \vec{z}_{p+1}^j are greater than or equal to the pivot, (iii).

The proof for sequent (6.6) is derived from (6.7) and (6.8) by applying the derived rule LCut which logically sequentialises the two systems; then we apply LInst to substitute $\vec{y}_i^{p-1} \vec{y}_{p+1}^j$ for $\vec{z}_i^{p-1} \vec{z}_{p+1}^j$ in b (notice that the substitution leaves the pre/post-conditions and the system unchanged as $\vec{z}_i^{p-1} \vec{z}_{p+1}^j$ are not free in them), then LImpl to recover the boolean condition $(\mathbf{ord}(\vec{y}_i^j) \wedge \vec{x}_i^j \doteq \vec{y}_i^j)$, then LRes to recover Γ from Γ_1 , and finally LLcl and LDef to recover $[\text{Qck}(i, j)]_{\rho(r, i, j)}$.

The proof of sequent (6.8) follows from the following three sequents (6.9), (6.10) and (6.11) below, where LRes is used to extend Γ_1 as

$$\Gamma_2 = \Gamma_1, r_1 : \rho(r_1, i, p-1), r_2 : \rho(r_2, p+1, j)$$

to account for the mappings associated with the channels r_1 and r_2 . Notice how this rule allows us to choose the permission association relating to r_1 and r_2 *dynamically*, depending on the index p returned by the partitioning phase of sequent (6.7). Such data dependencies normally complicate similar dependency analyses based on type systems such as [36, 4].

$$\Gamma_2; b \vdash \left\{ A_i^{p-1} \langle \vec{z}_i^{p-1} \rangle \right\} \quad [\text{Qck}(i, p-1)]_{\rho(r_1, i, p-1)} \quad \left\{ A_i^{p-1} \langle \vec{y}_i^{p-1} \rangle * r_1 \langle \rangle \right\} \quad (6.9)$$

$$\Gamma_2; b \vdash \left\{ A_{p+1}^j \langle \vec{z}_{p+1}^j \rangle \right\} \quad [\text{Qck}(p+1, j)]_{\rho(r_2, p+1, j)} \quad \left\{ A_{p+1}^j \langle \vec{y}_{p+1}^j \rangle * r_2 \langle \rangle \right\} \quad (6.10)$$

$$\Gamma_2; b \vdash \left\{ A_i^j \langle \vec{y}_i^j \rangle * r_1 \langle \rangle * r_2 \langle \rangle \right\} \quad [r_1?.r_2?.r!]_{\{\downarrow a_i, \downarrow r_1, \downarrow r_2, \uparrow r\}} \quad \left\{ A_i^j \langle \vec{y}_i^j \rangle * r \langle \rangle \right\} \quad (6.11)$$

Sequents (6.9) and (6.10) follow from the inductive hypotheses. Sequent (6.11) can be easily derived using LFRmSt , which eliminates $A_i^j \langle \vec{y}_i^j \rangle$ from the pre and post conditions, and then applying LIn twice for r_1 and r_2 respectively, followed by applying LOut once for r ; the two inputs on r_1 and r_2 would hand over the permissions $\downarrow a_i, \dots, \downarrow a_{p-1}$ and $\downarrow a_{p+1}, \dots, \downarrow a_j$ respectively; these are necessary for the output on r to be derived.

We recover the proof of sequent (6.8) as follows. Sequents (6.9) and (6.10) can be composed together as separate parallel code using LSEP , and then extended to include $a_p \langle y_p \rangle$ in the pre and

post-conditions using LFRMST . This allows us to logically sequence these two systems *before* the system $[r_1?.r_2?.r!]_{\{\downarrow a_p, \downarrow r_1, \downarrow r_2, \uparrow r\}}$ of sequent (6.11), thereby cutting the pre-condition of this sequent, using LCUT . Then we scope the two channels r_1 and r_2 using a combination of LRES , LLCL and LSPL (which leaves the pre and post conditions intact since they do not contain any mention of the channels r_1 and r_2), and finally precede this whole system by an input on r_3 using LIN , which adds $r_3\langle p \rangle$ to the precondition.

This leaves us with only sequent (6.7) to prove to complete the main proof. This sequent proof follows immediately from a proof for the following sequent

$$\Gamma_1; b \vdash \left\{ A_{i+1}^j \langle \vec{x}_{i+1}^j \rangle \right\} \left[\text{Trv}(i, j, x_i, i+1) \right]_{\rho(r_3, i, j) \cup \{\uparrow a_i\}} \left\{ \begin{array}{l} A_i^{p-1} \langle \vec{z}_i^{p-1} \rangle * a_p \langle y_p \rangle \\ * A_{p+1}^j \langle \vec{z}_{p+1}^j \rangle * r_3 \langle p \rangle \end{array} \right\} \quad (6.12)$$

through one application of LIN , which reinstates $a_i \langle x_i \rangle$ in the pre-condition, then an application of LDEF to recover $\left[\text{Prt}(i, j) \right]_{\rho(r_3, i, j)}$.

We prove (6.12) by proving the more general sequent

$$\Gamma_1; b' \vdash \left\{ \begin{array}{l} \overbrace{A_{i+1}^q \langle \vec{w}_{i+1}^q \rangle}^{(i)} \\ * \underbrace{A_{q+1}^{c-1} \langle \vec{w}_{q+1}^{c-1} \rangle}_{(ii)} * \underbrace{A_c^j \langle \vec{x}_c^j \rangle}_{(iii)} \end{array} \right\} \left[\text{Trv}(i, j, x_i, q, c) \right]_{\rho(r_3, i, j) \cup \{\uparrow a_i\}} \left\{ \begin{array}{l} A_i^{p-1} \langle \vec{z}_i^{p-1} \rangle * a_p \langle y_p \rangle \\ * A_{p+1}^j \langle \vec{z}_{p+1}^j \rangle * r_3 \langle p \rangle \end{array} \right\} \quad (6.13)$$

$$\text{where } b' = b \wedge (i \leq q < c \leq j+1) \wedge \underbrace{(\vec{x}_{i+1}^{c-1} \doteq \vec{w}_{i+1}^{c-1})}_{(iii)} \wedge \underbrace{\left(\bigwedge_{k=i+1}^q w_k < x_i \right)}_{(i)} \wedge \underbrace{\left(\bigwedge_{k=q+1}^{c-1} x_i \leq w_k \right)}_{(ii)}$$

Sequent (6.13) allows us to stratify every iteration of the traversal, thereby proving the sequent by induction on $n = (j+1) - c$. At each iteration, c , with pivot index q and pivot value x_i , (6.13) expects a precondition split into 3 parts: $A_{i+1}^q \langle \vec{w}_{i+1}^q \rangle$ holds processed values that are less than the pivot x_i , (i), $A_{q+1}^{c-1} \langle \vec{w}_{q+1}^{c-1} \rangle$ holds processed values that are greater than or equal to the pivot x_i , (ii), and $A_c^j \langle \vec{x}_c^j \rangle$ is the part of the array that still needs to be traversed. Note also that the values preceding the current counter, \vec{w}_{i+1}^{c-1} , must be equal, up to reordering, of the values already processed \vec{x}_{i+1}^{c-1} , (iii). The base case, *i.e.*, when $c = j+1$ (and thus $A_c^j \langle \vec{x}_c^j \rangle = A_{j+1}^j \langle \vec{x}_{j+1}^j \rangle = \mathbf{emp}$), establishes the post-condition in (6.13) whereas the inductive case works up towards the base case, whereby the value comparison at every iteration adds to the ordering information expressed by b' . Both proof cases use a mixture of rules LIN , LOUT , LIF and, LSEPST and LCUT in a manner similar to that discussed already above; the details are left for the interested reader.

To obtain (6.12) from (6.13), we take q and c to be i and $i+1$ respectively. This case makes the array assertions $A_{i+1}^q \langle \vec{w}_{i+1}^q \rangle$ and $A_{q+1}^{c-1} \langle \vec{w}_{q+1}^{c-1} \rangle$ in the precondition of (6.13) empty, *i.e.*, $A_{i+1}^q \langle \vec{w}_{i+1}^q \rangle = A_{q+1}^{c-1} \langle \vec{w}_{q+1}^{c-1} \rangle = A_{i+1}^i \langle \vec{w}_{i+1}^i \rangle = \mathbf{emp}$, which by Lemma 4.9 and LMP , leaves us with $A_{i+1}^j \langle \vec{x}_{i+1}^j \rangle$ *i.e.*, the precondition of (6.12). Moreover, for this case the boolean expression b' is of the form $b \wedge (i \leq i < i+1 \leq j+1)$ which is implied by b , *i.e.*, $b \models b'$. This means that we can recover b for our sequent simply by applying LMP as well.

7. CONCLUSION

We have developed a logic for deterministic processes, interpreted over systems whose behaviour is confined by sets of linear permissions. We also developed a sound proof system through which we

can determine, in compositional fashion, the satisfaction of formulas in this logic. We applied this logic and proof system to specify and prove the correctness of an in-place parallel quicksort.

7.1. Related Work. Modal logics have traditionally been used in process calculi for the specification of behavioural properties. Proof systems for these logics have been developed in a variety of settings (e.g., [20, 2, 1, 12, 13, 4]) and some of these have focused on compositional reasoning as a means of dealing with the scalability problem (e.g., [2, 13, 4]). However, there has been little focus on locality of reasoning in these efforts. Approaching compositionality without necessarily modelling locality does seem to have been at the expense of general, but long-winded proof rules for parallel composition (e.g. [13]). In addition, termination is often not a major focus in these logics; in fact, the bisimulation proof technique, often associated with these logics, is insensitive to divergence. Termination is central to the logical characterisations that we give in this work.

Despite the apparent resemblance, spatial logics for process calculi such as [8, 9] differ from our interpretation of the separating conjunction: we separate on *permissions*, logical embellishments on processes, whereas their logical separation is more intensional and operates on the structure of processes, describing parallel composition directly. Moreover, their aims appear to differ from ours since they model mobility and channel privacy; we focus on data, non-interference and locality, and deal with implicit transfer of permissions.

Following [28], the use of separation logic to support local reasoning for concurrent programs has been studied intensively over the past few years for the shared-variable model of concurrency. The initial main idea of ownership transfer of resources between threads impacting upon local reasoning already appears in [28]. This was then extended to co-exist with Rely/Guarantee reasoning [37, 15] and recently refined through fractional permissions as Deny/Guarantee reasoning [14]. The latter is interesting to us as a means of widening our class of programs under analysis. For instance, [18] uses this approach for dealing with dynamically allocated resource locks.

Separation Logic has been applied to process calculi on at least two occasions. In [22], they give a separation semantics for a variant of the piCalculus, based on traces. Their work differs from ours in a number of respects in that they only deal with *explicit* ownership transfer of resources and are not concerned with developing a proof system. In [31], they also use a process calculus as a model for a separation logic. They are quite general *wrt.* the form of resources and how these are transferred across processes and, as a result, our model of confined processes seems related to theirs. However, aspects such as the use of SCCS on their part, where processes evolve in synchrony, and the focus on value passing and stability on ours, lead to a substantially different satisfaction relation of the logics. The aim of their work is also different from ours; they establish a correspondence between strong bisimulation and logic satisfaction whereas we focus on developing a compositional proof system. Separation logic has also been applied to an imperative concurrent language with message passing in [38] where the main focus is the implementability of message-passing communication as a copy-less communication over a shared memory model. Although their technical development is considerably different from ours, this work can be seen as complementary to ours if implementation aspects of our language are considered.

7.2. Future Work. There is much further work to be done in the area of local reasoning for message-passing concurrency.

With respect to the work presented here, there are a number of design decisions that are worth exploring. For instance, at the level of the proof system, a partial correctness interpretation of our

sequents (as opposed to total correctness) would probably allow us to design a version of the parallel proof rule, LPar , that is more symmetric. Another avenue worth exploring is that of relaxing the interpretation of our logical assertions so as to not limit them to safely-stabilising systems. This would simplify the verification of certain formulas, such as **any**, and would also allow us to have models where formulas such as $c\langle v \rangle * \mathbf{blk}(c)$ are satisfiable. At the same time, this satisfaction weakening would also entail that our existing assertion interpretation changes to one where systems satisfy a formula at some point during their evaluation but may then fail to satisfy it as computation progresses. Although it is not yet clear whether this is a desirable property to have from the point of view of the application of the logic, it has appealing benefits in terms of the assertion satisfaction definition, as it streamlines the satisfaction of core formulas like the separating conjunction with existing interpretations. Moreover, we also conjecture that this altered interpretation would eliminate the need for the side conditions present in the existing parallel rule, LPar .

At a more general level, we also seek to widen the class of programs we can treat by introducing non-confluent behaviour in a controlled way. We intend to extend our setting to allow for more interesting forms of data to be communicated, including say channel names. We also need to develop algorithms for inferring the permission-set maps, develop tools to support the proof-system reasoning. Finally, and perhaps most importantly, we need to expand our suite of case studies and consider larger example proofs.

ACKNOWLEDGMENT

The authors wish to acknowledge numerous referees for their incisive comments on a preliminary version of this paper.

REFERENCES

- [1] R. Amadio and M. Dam. Toward a modal theory of types for the π -calculus. In *FTRTFT*, volume 1135 of *LNCS*, 1996.
- [2] H. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal μ -calculus. In *LICS*, volume 4-7, pages 144–153, 1994.
- [3] J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.
- [4] M. Berger, K. Honda, and N. Yoshida. Completeness and full abstraction in modal logics for typed mobile processes. In *ICALP*, volume 5126 of *LNCS*, pages 99–111, 2008.
- [5] R. Bornat, C. Calcagno, P. O’Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
- [6] J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72, 2003.
- [7] S. Brookes. A semantics for concurrent separation logic. *TCS*, 375(1-3):227–270, 2007.
- [8] L. Caires and L. Cardelli. A spatial logic for concurrency (i). *I&C*, pages 1–37, 2001.
- [9] L. Caires and L. Cardelli. A spatial logic for concurrency (ii). *TCS*, 322(3):517–565, 2004.
- [10] C. Calcagno, P. W. O’Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378, 2007.
- [11] C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, volume 4634 of *LNCS*, pages 233–238, 2007.
- [12] M. Dam. Model checking mobile processes. In *CONCUR*, volume 715 of *LNCS*, pages 22–36, 1993.
- [13] M. Dam. Proof systems for pi-calculus logics. In *Logic for Concurrency and Synchronisation*, Trends in Logic, Studia Logica Library, pages 145–212, 2003.
- [14] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP 2009*, volume 5502 of *LNCS*, pages 363–377, 2009.
- [15] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, pages 173–188, 2007.
- [16] J. Galletly. *Occam-2 (2nd ed.): including occam.2.1*. UCL Press, 1997.

- [17] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.
- [18] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, volume 4807 of *LNCS*, pages 19–37, 2007.
- [19] M. Hennessy and H. Lin. Proof systems for message-passing process algebras. In *Formal Aspects of Computing*, pages 379–407, 1996.
- [20] M. Hennessy and H. Liu. A modal logic for message passing processes. *Act. Inf.*, 32:375–393, 1995.
- [21] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, 1978.
- [22] T. Hoare and P. O’Hearn. Separation logic semantics for communicating processes. *ENTCS*, 212:3–25, 2008.
- [23] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
- [24] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. *ACM ToPLAS*, 21(5):914–947, 1999.
- [25] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [26] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.
- [27] R. Milner. *Communicating and mobile systems: the π -calculus*. Cambridge Univ., 1999.
- [28] P. W. O’Hearn. Resources, concurrency and local reasoning. *TCS*, pages 49–67, 2004.
- [29] M. Parkinson, R. Bornat, and P. W. O’Hearn. Modular verification of a non-blocking stack. *SIGPLAN Not.*, 42(1):297–302, 2007.
- [30] Mojżesz Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.
- [31] D. Pym and C. Tofts. A calculus & logic of resources & processes. *FAC*, 18:495–517, 2006.
- [32] J. H. Reppy. *Concurrent programming in ML*. Cambridge University Press, 1999.
- [33] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
- [34] D. Sangiorgi and D. Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
- [35] C. Stirling. *Modal and temporal properties of processes*. Springer-Verlag, 2001.
- [36] T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *TOPLAS*, 30(5):1–30, 2008.
- [37] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.
- [38] J. Villard, É. Lozes, and C. Calcagno. Proving copyless message passing. In *APLAS’09*, volume 5904 of *LNCS*, pages 194–209, Seoul, Korea, 2009.

APPENDIX A. PROOFS

A.1. Processes.

Lemma A.1 (Structural Equivalence and Reductions).

$$P \equiv Q \text{ and } P \longrightarrow P' \text{ implies } \exists Q'. Q \longrightarrow Q' \text{ and } P' \equiv Q'$$

Proof. By rule induction on $P \equiv Q$. □

Corollary A.2 (Structural Equivalence and Reductions). $P \equiv Q$ and $P \not\rightarrow$ implies $Q \not\rightarrow$

A.2. Confined Processes.

Lemma A.3. $S \equiv T$ implies $|S| \equiv |T|$

Proof. By Rule induction on $S \equiv T$

scNIL: $|S| \parallel |\llbracket \text{nil} \rrbracket_\emptyset| = |S| \parallel |\llbracket \text{nil} \rrbracket_\emptyset| = |S| \parallel \text{nil}$ and $|S| \parallel \text{nil} \equiv |S|$ by sNIL.

scCOM, scASS, scNEW, scSWP: By the corresponding structural rules sCOM, sASS, sNEW, sSWP.

scEXT: By sEXT and the fact that $c \notin \text{fn}(S)$ implies $c \notin \text{fn}(|S|)$. □

Lemma 3.29 (Correspondence). $S \longrightarrow T$ implies $|S| \longrightarrow |T|$ or $|S| \equiv |T|$

Proof. The proof is by rule induction on $S \longrightarrow T$.

cTHN, cELS, cCOM, cPRC: There is a corresponding reduction rule in the semantics of Figure 1.

cSPL, cRST, cDSC: Satisfies $|S| \equiv |T|$.

cPAR, cRES: Follows by I.H.

cSTR: By rSTR and Lemma A.3 □

Corollary A.4. $|S| \not\rightarrow$ implies $S \not\rightarrow$ or $(\exists T. S \longrightarrow T \text{ and } |S| \equiv |T|)$

Lemma 3.17 (Properties of \approx with respect to reductions).

(1) $S \approx T$ and $T \longrightarrow T'$ and $S \not\rightarrow_{\text{err}}$ implies $\exists S'. S \longrightarrow S'$ and $S' \approx T'$

(2) $S \approx T$ and $S \checkmark$ implies $T \not\rightarrow$

Proof. The first clause is proved by case analysis of $T \longrightarrow T'$ using Lemma A.5 to infer the structure of T , then use the definition $S \approx T$ to determine the structure of S . The second clause is proved by assuming that $\exists T'$ such that $T \longrightarrow T'$ and then use the first clause to show that this leads to a contradiction. □

Lemma A.5 (Reduction and System Structure). $S \longrightarrow T$ implies

(1) $S \equiv (\text{new } \vec{c}) (\llbracket c! \vec{e} \rrbracket_\rho \parallel \llbracket c? \vec{x}. P \rrbracket_\mu \parallel R)$, $T \equiv (\text{new } \vec{c}) (\llbracket P \parallel \vec{x} \rrbracket_{\rho \cup \mu} \parallel R)$, $\uparrow c \in \rho$, $\downarrow c \in \mu$, $\vec{e} \Downarrow \vec{v}$ or;

(2) $S \equiv (\text{new } \vec{c}) (\llbracket \text{if } b \text{ then } P \text{ else } Q \rrbracket_\rho \parallel R)$, $T \equiv (\text{new } \vec{c}) (\llbracket P \rrbracket_\rho \parallel R)$ or $T \equiv (\text{new } \vec{c}) (\llbracket Q \rrbracket_\rho \parallel R)$ or;

(3) $S \equiv (\text{new } \vec{c}) (\llbracket K(\vec{e})[\vec{d}_1/\vec{d}_2] \rrbracket_\rho \parallel R)$, $T \equiv (\text{new } \vec{c}) (\llbracket P \parallel \vec{x} \rrbracket_\rho \parallel \llbracket \vec{d}_1/\vec{d}_2 \rrbracket_\rho \parallel R)$, $\vec{e} \Downarrow \vec{v}$ or;

(4) $S \equiv (\text{new } \vec{c}) (\llbracket P \parallel Q \rrbracket_{\rho \cup \mu} \parallel R)$, $T \equiv (\text{new } \vec{c}) (\llbracket P \rrbracket_\rho \parallel \llbracket Q \rrbracket_\mu \parallel R)$ or;

(5) $S \equiv (\text{new } \vec{c}) (\llbracket (\text{new } c) P \rrbracket_\rho \parallel R)$, $T \equiv (\text{new } \vec{c}) ((\text{new } c) (\llbracket P \rrbracket_{\rho \cup \{\downarrow c, \uparrow c\}} \parallel R))$ or;

(6) $S \equiv (\text{new } \vec{c}) (\llbracket \text{nil} \rrbracket_\rho \parallel R)$, $S \equiv (\text{new } \vec{c}) (\llbracket \text{nil} \rrbracket_\emptyset \parallel R)$, $\rho \neq \emptyset$

Proof. By rule induction on $S \longrightarrow T$. □

Proposition 3.14 (Safe-Stability and System Structure).

$$S \checkmark \text{ iff } S \equiv (\text{new } \vec{d}) \left(\prod_{i=0}^n [c_i!e_i]_{\rho_i} \prod_{j=0}^m [c'_j?x_j.P_j]_{\mu_j} \right)$$

where

- $\{c_1, \dots, c_n\} \cap \{c'_1, \dots, c'_m\} = \emptyset$.
- $\bigwedge_{i=0}^n \uparrow c_i \in \rho_i$ and $\bigwedge_{j=0}^m \downarrow c_j \in \mu_j$

and where $\prod_{i=0}^n [c_i!e_i]_{\rho_i}$ and $\prod_{j=0}^m [c'_j?x_j.P_j]_{\mu_j}$ denote $\ulcorner \text{nil} \urcorner_{\emptyset}$.

Proof. Immediate by case analysis of Lemma A.5 and then the conditions for $S \longrightarrow_{\text{err}}$ from Figure 2. \square

Lemma 3.18 (Partial Confluence). $S \longrightarrow T_1$ and $S \longrightarrow T_2$ implies either of the following:

- (1) $T_1 \cong T_2$ or;
- (2) $\exists T_3. T_1 \longrightarrow T_3$ and $T_2 \longrightarrow T_3$

Proof. By case analysis of the possible forms of S using Lemma A.5, then restricting the possibilities using properties of well-formed systems. We here overview the two main cases.

- For $S \longrightarrow T_1$ we have

$$S \equiv (\text{new } \vec{c}) \left(\ulcorner c_1!e_1 \urcorner_{\rho_1} \parallel \ulcorner c_1?x.P_1 \urcorner_{\mu_1} \parallel R_1 \right), \quad T_1 \equiv (\text{new } \vec{c}) \left(\ulcorner P \parallel \vec{x} \urcorner_{\rho_1 \cup \mu_1} \parallel R_1 \right), \quad \uparrow c_1 \in \rho_1, \quad \downarrow c_1 \in \mu_1.$$

Also for $S \longrightarrow T_2$ we have

$$S \equiv (\text{new } \vec{c}) \left(\ulcorner c_2!e_2 \urcorner_{\rho_2} \parallel \ulcorner c_2?x.P_2 \urcorner_{\mu_2} \parallel R_2 \right), \quad T_2 \equiv (\text{new } \vec{c}) \left(\ulcorner P \parallel \vec{x} \urcorner_{\rho_2 \cup \mu_2} \parallel R_2 \right), \quad \uparrow c_2 \in \rho_2, \quad \downarrow c_2 \in \mu_2.$$

We have two sub-cases

$c_1 \neq c_2$: The two redexes in S are distinct and, for some system R , we have

$$R_1 \equiv (\text{new } \vec{d}_2) \left(\ulcorner c_2!e_2 \urcorner_{\rho_2} \parallel \ulcorner c_2?x.P_2 \urcorner_{\mu_2} \parallel R \right) \quad \text{and} \quad R_2 \equiv (\text{new } \vec{d}_1) \left(\ulcorner c_1!e_1 \urcorner_{\rho_1} \parallel \ulcorner c_1?x.P_1 \urcorner_{\mu_1} \parallel R \right)$$

from which we can then find a common T_3 that both T_1 and T_2 reduce to.

$c_1 = c_2$: The conditions that $\uparrow c_1 \in \rho_1, \downarrow c_1 \in \mu_1, \uparrow c_2 \in \rho_2$ and $\downarrow c_2 \in \mu_2$ and the fact that S is well-formed ensure that $S \longrightarrow T_1$ and $S \longrightarrow T_2$ refer to the same reduction (modulo structural equivalence) i.e., $\rho_1 = \rho_2, \mu_1 = \mu_2, e_1 = e_2, P_1 = P_2$ and $R_1 = R_2$ which implies $T_1 \equiv T_2$, thus $T_1 \cong T_2$ by Proposition 3.16.

- For $S \longrightarrow T_1$ we have $S \equiv (\text{new } \vec{c}) \left(\ulcorner P_1 \parallel Q_1 \urcorner_{\rho_1 \cup \mu_1} \parallel R_1 \right), T_1 \equiv (\text{new } \vec{c}) \left(\ulcorner P_1 \urcorner_{\rho_1} \parallel \ulcorner Q_1 \urcorner_{\mu_1} \parallel R_1 \right)$ and for $S \longrightarrow T_2$ we have $S \equiv (\text{new } \vec{c}) \left(\ulcorner P_2 \parallel Q_2 \urcorner_{\rho_2 \cup \mu_2} \parallel R_2 \right), T_2 \equiv (\text{new } \vec{c}) \left(\ulcorner P_2 \urcorner_{\rho_2} \parallel \ulcorner Q_2 \urcorner_{\mu_2} \parallel R_2 \right)$. By the assumption that S is well-formed, we have the following sub-cases:

$(\rho_1 \uplus \mu_1) \neq (\rho_2 \uplus \mu_2)$: Then we have different redexes meaning that for some R we have

$$R_1 \equiv (\text{new } \vec{d}_2) \left(\ulcorner P_2 \parallel Q_2 \urcorner_{\rho_2 \cup \mu_2} \parallel R \right) \quad \text{and} \quad R_2 \equiv (\text{new } \vec{d}_1) \left(\ulcorner P_1 \parallel Q_1 \urcorner_{\rho_1 \cup \mu_1} \parallel R \right),$$

which guarantees the existence of a common system T_3 that T_1 and T_2 can reduce to.

$(\rho_1 \uplus \mu_1) = (\rho_2 \uplus \mu_2)$: Then we must have the same redexes, i.e., $P_1 = P_2, Q_1 = Q_2$ and $R_1 = R_2$. This implies $T_1 \cong T_2$. \square

The following technical Lemmas deal with the restricted non-determinism of confined processes and how it can be characterised using the relation \cong . In particular, Lemma A.8 is useful because it allows us to correct reductions that lead to systems that do not evaluate by instead reducing to systems that are related to them by \cong , which in turn means, by Proposition 3.16, that they contain the same process structure.

Lemma A.6. $S \Downarrow$ and $S \longrightarrow T$ and $T \Downarrow$ implies $\exists P, Q, R. S \equiv (\text{new } \vec{c})(\Gamma P \parallel Q \upharpoonright_{\rho \uplus \mu} \parallel R)$ and $T \equiv (\text{new } \vec{c})(\Gamma P \upharpoonright_{\rho} \parallel \Gamma Q \upharpoonright_{\mu} \parallel R)$

Proof. By induction on the number of reductions in $S \Downarrow$ leading to a safely-stable system i.e., $S \longrightarrow^n S' \checkmark$ for some S' .

$n = 1$: By Lemma 3.18 and $S' \checkmark$ (i.e., $S' \not\rightarrow$) it must be the case that $T \cong S'$. By Lemma 3.17.2 this also implies $T \not\rightarrow$ and since $T \Downarrow$ it must be the case that $T \longrightarrow_{\text{err}}$. Now by case analysis of Lemma A.5, the only system structure that allows this is when $S \equiv (\text{new } \vec{c})(\Gamma P \parallel Q \upharpoonright_{\rho \uplus \mu} \parallel S'')$ and $T \equiv (\text{new } \vec{c})(\Gamma P \upharpoonright_{\rho} \parallel \Gamma Q \upharpoonright_{\mu} \parallel S'')$.

$n = k + 1$: We have $S \Downarrow S \longrightarrow S'' \xrightarrow{k} S' \checkmark$. By Lemma 3.18 we have two sub-cases. The first case subsumes the second in some cases, so we here consider the mutually exclusive variants:

$\exists T'. T \longrightarrow T'$ and $S'' \longrightarrow T'$: $T \Downarrow$ implies $T' \Downarrow$, and by $S'' \longrightarrow T'$, $S'' \xrightarrow{k} S' \checkmark$ and I.H. we obtain $S'' \equiv (\text{new } \vec{c})(\Gamma P \parallel Q \upharpoonright_{\rho \uplus \mu} \parallel S''')$ and $T' \equiv (\text{new } \vec{c})(\Gamma P \upharpoonright_{\rho} \parallel \Gamma Q \upharpoonright_{\mu} \parallel S''')$. Now $T \Downarrow$ and $S'' \xrightarrow{k} S' \checkmark$ implies $T \neq S''$. Thus by that fact that $S \longrightarrow T \longrightarrow T'$ and the uniqueness of linear permissions, it must be the case that $S \equiv (\text{new } \vec{c})(\Gamma P \parallel Q \upharpoonright_{\rho \uplus \mu} \parallel S''')$ and $T \equiv (\text{new } \vec{c})(\Gamma P \upharpoonright_{\rho} \parallel \Gamma Q \upharpoonright_{\mu} \parallel S''')$ for some S'''' such that $S'''' \longrightarrow S''$.

$T \cong S''$ **where** $\nexists T'. T \longrightarrow T'$ and $S'' \longrightarrow T'$: Clearly, since $T \Downarrow$ we have $T \neq S''$. Also the fact that there is no common system T and S'' can reduce to means that the reductions from S were not from separate redexes. By case analysis of Lemma A.5, the only possible option for having non-deterministic reductions from the same redex is the case where $S \equiv (\text{new } \vec{c})(\Gamma P \parallel Q \upharpoonright_{\rho \uplus \mu} \parallel S''')$ and $T \equiv (\text{new } \vec{c})(\Gamma P \upharpoonright_{\rho} \parallel \Gamma Q \upharpoonright_{\mu} \parallel S''')$. \square

Lemma A.7. $S \Downarrow$ and $S \equiv (\text{new } \vec{c})(\Gamma P \parallel Q \upharpoonright_{\rho} \parallel R)$ implies $\exists \mu_1, \mu_2$ such that $\mu_1 \uplus \mu_2 = \rho$ and $(\text{new } \vec{c})(\Gamma P \upharpoonright_{\mu_1} \parallel \Gamma Q \upharpoonright_{\mu_2} \parallel R) \Downarrow$

Proof. By induction on the number of reductions in $S \Downarrow$ leading to a safely-stable system i.e., $S \longrightarrow^n S' \checkmark$ for some S' .

$n = 1$: By cSPL, $S \equiv (\text{new } \vec{c})(\Gamma P \parallel Q \upharpoonright_{\rho} \parallel R)$ can reduce to $(\text{new } \vec{c})(\Gamma P \upharpoonright_{\rho_1} \parallel \Gamma Q \upharpoonright_{\rho_2} \parallel R)$, for some ρ_1, ρ_2 , and by Lemma 3.18 and $S' \not\rightarrow$ we must have $S' \cong (\text{new } \vec{c})(\Gamma P \upharpoonright_{\rho_1} \parallel \Gamma Q \upharpoonright_{\rho_2} \parallel R)$, and since $S' \not\rightarrow_{\text{err}}$, this implies

$\exists \mu_1, \mu_2$ such that $\mu_1 \uplus \mu_2 = \rho$ and $(\text{new } \vec{c})(\Gamma P \upharpoonright_{\mu_1} \parallel \Gamma Q \upharpoonright_{\mu_2} \parallel R) \Downarrow$.

$n = k + 1$: We have $S \longrightarrow S' \xrightarrow{k} S'' \checkmark$ for some S', S'' . Lemma A.5 gives us two sub-cases:

$S' \equiv (\text{new } \vec{c})(\Gamma P \parallel Q \upharpoonright_{\rho} \parallel R')$ **where** $R \longrightarrow R'$: By $S' \xrightarrow{k} S'' \checkmark$ and I.H. we obtain $\exists \mu_1, \mu_2$ such that $\mu_1 \uplus \mu_2 = \rho$ and $(\text{new } \vec{c})(\Gamma P \upharpoonright_{\mu_1} \parallel \Gamma Q \upharpoonright_{\mu_2} \parallel R')$ \Downarrow which implies that $\exists \mu_1, \mu_2$ such that $\mu_1 \uplus \mu_2 = \rho$ and $(\text{new } \vec{c})(\Gamma P \upharpoonright_{\mu_1} \parallel \Gamma Q \upharpoonright_{\mu_2} \parallel R) \Downarrow$.

$(\text{new } \vec{c})(\Gamma P \upharpoonright_{\rho_1} \parallel \Gamma Q \upharpoonright_{\rho_2} \parallel R)$: Immediate. \square

Lemma A.8 (Corrective Reductions). $S \Downarrow$ and $S \longrightarrow T$ and $T \Downarrow$ implies $\exists R$ such that $S \longrightarrow R$ and $R \cong T$ and $R \Downarrow$

Proof. By Lemma A.6 we know $S \equiv (\text{new } \vec{c})(\Gamma P \parallel Q \upharpoonright_{\rho \uplus \mu} \parallel S')$ as well as $T \equiv (\text{new } \vec{c})(\Gamma P \upharpoonright_{\rho} \parallel \Gamma Q \upharpoonright_{\mu} \parallel S')$. By Lemma A.7 we know $\exists \mu_1, \mu_2$ such that $\mu_1 \uplus \mu_2 = \rho$ and $(\text{new } \vec{c})(\Gamma P \upharpoonright_{\mu_1} \parallel \Gamma Q \upharpoonright_{\mu_2} \parallel S') \Downarrow$. Since $T \cong (\text{new } \vec{c})(\Gamma P \upharpoonright_{\mu_1} \parallel \Gamma Q \upharpoonright_{\mu_2} \parallel S')$ this implies that we can correct the permission split and be able to reduce to a safely-stable state. \square

In order to apply corrective actions to multiple reduction steps, we need to extend Lemma A.8 to systems that are related by \cong , due to reductions of type (1) of Lemma 3.18. The next Lemmas deal with this. Lemma 3.20 states that there exist matching reductions for systems related by \cong preserving the evaluation property and Lemma A.9 extends this to multiple reductions. This allows us to prove the existence of corrective reductions over multiple reductions.

Lemma 3.20 (Evaluation Preservation for \cong). $S \cong T$ and $S \Downarrow$ and $T \longrightarrow T'$ implies $\exists S'$ such that $S \longrightarrow S'$ where $S' \cong T'$ and $S' \Downarrow$

Proof. By $S \Downarrow$ and Lemma 3.7 we have $S \not\rightarrow_{\text{err}}$ and by Lemma 3.17.1 we know $\exists S_1$ such that $S \longrightarrow S_1$ and $S_1 \cong T'$. At this point we have two sub-cases: if $S_1 \Downarrow$ then the result follows immediately. Otherwise, if $S_1 \not\Downarrow$, then Lemma A.8 states that $\exists S_2$ such that $S \longrightarrow S_2$ and $S_2 \cong S_1$ and $S_2 \Downarrow$. By transitivity we have $S_2 \cong S_1 \cong T'$. \square

Lemma A.9 (Evaluation Preservation for \cong). $S \cong T$ and $S \Downarrow$ and $T \longrightarrow^n T'$ implies $\exists S'$ such that $S \longrightarrow^n S'$ where $S' \cong T'$ and $S' \Downarrow$

Proof. By induction on n , the number of reductions in $T \longrightarrow^n T'$.

$n = 0$: Immediate.

$n = k + 1$: We have $T \longrightarrow T'' \longrightarrow^k T'$. From $T \longrightarrow T''$ and Lemma 3.20 we obtain $\exists S'$ such that $S \longrightarrow S'$ where $S' \cong T'$ and $S' \Downarrow$. By I.H. we know $S' \longrightarrow^k S''$ for some S'' such that $S'' \cong T'$ and $S'' \Downarrow$ and $S \longrightarrow S' \longrightarrow^k S''$ gives us the required reduction sequence. \square

Lemma A.10. $|S| \equiv Q$ and $S \Downarrow$ implies $\exists T$ such that $S \longrightarrow^* T$ and $T \Downarrow$ and $|T| = Q$.

Proof. By rule induction on $|S| \equiv Q$. \square

Lemma A.11. $|S| \equiv Q$ implies $\exists T. S \longrightarrow^* T$ or $S \equiv T S$ where $|T| = Q$

Proof. By rule induction on $|S| \equiv Q$ and then a tedious consideration of all the possible permutations of S that may lead to $|S|$.

sAss: If $|S| = P_1 \parallel (P_2 \parallel P_3)$ then $Q = (P_1 \parallel P_2) \parallel P_3$ and S can be either of the following:

$S = \lceil P_1 \parallel (P_2 \parallel P_3) \rceil_\rho$: By 2 applications of cSPL and then an application of cSTR using scAss we obtain $S \longrightarrow^+ (\lceil P_1 \rceil_{\rho_1} \parallel \lceil P_2 \rceil_{\rho_2}) \parallel \lceil P_3 \rceil_{\rho_3}$ where $\rho_1 \uplus \rho_2 \uplus \rho_3 = \rho$ and $|(\lceil P_1 \rceil_{\rho_1} \parallel \lceil P_2 \rceil_{\rho_2}) \parallel \lceil P_3 \rceil_{\rho_3}| = Q$.

$S = \lceil P_1 \rceil_{\rho_1} \parallel \lceil (P_2 \parallel P_3) \rceil_\rho$: By one application of cSPL and one application of cSTR using scAss we obtain $S \longrightarrow^+ (\lceil P_1 \rceil_{\rho_1} \parallel \lceil P_2 \rceil_{\rho_2}) \parallel \lceil P_3 \rceil_{\rho_3}$ where $\rho_2 \uplus \rho_3 = \rho$ and $|(\lceil P_1 \rceil_{\rho_1} \parallel \lceil P_2 \rceil_{\rho_2}) \parallel \lceil P_3 \rceil_{\rho_3}| = Q$.

$S = \lceil P_1 \rceil_{\rho_1} \parallel \lceil (P_2 \parallel P_3) \rceil_{\rho_3}$: By scAss we obtain $S \equiv (\lceil P_1 \rceil_{\rho_1} \parallel \lceil P_2 \rceil_{\rho_2}) \parallel \lceil P_3 \rceil_{\rho_3}$ where $|(\lceil P_1 \rceil_{\rho_1} \parallel \lceil P_2 \rceil_{\rho_2}) \parallel \lceil P_3 \rceil_{\rho_3}| = Q$.

The symmetric case where $|S| = (P_1 \parallel P_2) \parallel P_3$ and $Q = P_1 \parallel (P_2 \parallel P_3)$ is similar.

sCOM: Similar to sAss case.

sNIL: If $|S| = P \parallel \text{nil}$ and $Q = P$ then we have two cases:

$S = \lceil P \parallel \text{nil} \rceil_\rho$: By cSPL, cSTR and scNIL we obtain $S \longrightarrow^+ \lceil P \rceil_\rho$ and $|\lceil P \rceil_\rho| = P = Q$.

$S = \lceil P \rceil_{\rho_1} \parallel \lceil \text{nil} \rceil_{\rho_2}$: By cDsc, cSTR and scNIL we obtain $S \longrightarrow^+ \lceil P \rceil_\rho$ and $|\lceil P \rceil_\rho| = P = Q$.

If $|S| = P$ and $Q = P \parallel \text{nil}$, then by scNIL we have $S \equiv S \parallel \lceil \text{nil} \rceil_\emptyset$ and $|S \parallel \lceil \text{nil} \rceil_\emptyset| = Q$.

sNEW: The most difficult case is when $S = \lceil (\text{new } c) \text{nil} \rceil_\rho$ and $Q = \text{nil}$. By cLCL, cDsc, cSTR with scNEW we obtain $S \longrightarrow^+ \lceil \text{nil} \rceil_\emptyset$ and $|\lceil \text{nil} \rceil_\emptyset| = Q$. The other cases are similar.

sSWP: There are three cases; $S = \lceil (\text{new } c)(\text{new } d)P \rceil_\rho$, $S = (\text{new } c) \lceil (\text{new } d)P \rceil_\rho$ and $S = (\text{new } c)(\text{new } d) \lceil P \rceil_\rho$ and proved similar to the cases above using cLCL, cSTR and scSWP.

sEXP: When $|S| = P \parallel (\text{new } c)Q$ we have three cases: $S = \lceil P \parallel (\text{new } c)Q \rceil_\rho$, $S = \lceil P \rceil_{\rho_1} \parallel \lceil (\text{new } c)Q \rceil_{\rho_2}$ and $S = \lceil P \rceil_{\rho_1} \parallel (\text{new } c) \lceil Q \rceil_{\rho_2}$ and the proof follows using the rules cSPL, cLCL, cSTR and scEXT. The symmetric case when $|S| = (\text{new } c)P \parallel Q$ is similar. \square

Lemma 3.27 (Reduction Correspondence).

$$S \Downarrow \text{ and } |S| \longrightarrow Q \text{ implies } \exists R \text{ such that } S \longrightarrow^+ R \text{ and } |R| \equiv Q$$

Proof. By rule induction on $|S| \longrightarrow Q$. We here consider the main cases:

rCOM We have $|S| = c!\vec{e} \| c?\vec{x}.P$ and $Q = P \llbracket \vec{v}/\vec{x} \rrbracket$ where $\vec{e} \Downarrow \vec{v}$. We have two sub-cases for S :

$$S = \lceil c!\vec{e} \rceil_{\rho} \| c?\vec{x}.P \rceil_{\rho}: \text{ By } S \Downarrow, \exists \mu_1, \mu_2 \text{ such that } \mu_1 \uplus \mu_2 = \rho \text{ and } S \longrightarrow \lceil c!\vec{e} \rceil_{\mu_1} \| \lceil c?\vec{x}.P \rceil_{\mu_2} \longrightarrow \lceil P \llbracket \vec{v}/\vec{x} \rrbracket \rceil_{\rho}$$

$$\text{ and } \lceil \lceil P \llbracket \vec{v}/\vec{x} \rrbracket \rceil_{\rho} \rceil = Q.$$

$$S = \lceil c!\vec{e} \rceil_{\mu_1} \| \lceil c?\vec{x}.P \rceil_{\mu_2}: \text{ Similar}$$

rPAR We have $|S| = P_1 \| P_2$ and $Q = P'_1 \| P_2$ because $P_1 \longrightarrow P'_1$. We have two sub-cases for S :

$$S = \lceil P_1 \| P_2 \rceil_{\rho}: \text{ By } S \Downarrow \exists \mu_1, \mu_2 \text{ such that } \mu_1 \uplus \mu_2 = \rho \text{ and } \lceil P_1 \| P_2 \rceil_{\rho} \longrightarrow \lceil P_1 \rceil_{\mu_1} \| \lceil P_2 \rceil_{\mu_2} \Downarrow. \text{ Now } \lceil P_1 \rceil_{\mu_1} \| \lceil P_2 \rceil_{\mu_2} \Downarrow \text{ implies } \lceil P_1 \rceil_{\mu_1} \Downarrow \text{ and by } P_1 \longrightarrow P'_1 \text{ and I.H. we know } \exists R \text{ such that } \lceil P_1 \rceil_{\mu_1} \longrightarrow^+ R \text{ and } |R| \equiv P'_1. \text{ Thus, by cPAR, } \lceil P_1 \rceil_{\mu_1} \| \lceil P_2 \rceil_{\mu_2} \longrightarrow R \| \lceil P_2 \rceil_{\mu_2} \text{ and } |R \| \lceil P_2 \rceil_{\mu_2}| = Q.$$

$$S = S_1 \| S_2 \text{ where } |S_1| = P_1 \text{ and } |S_2| = P_2: \text{ Similar}$$

rSTR We have $|S| = P_1$ and $Q = P_2$ because $P_1 \equiv P'_1, P'_1 \longrightarrow P'_2, P'_2 \equiv P_2$. By $|S| = P_1$ and Lemma A.10 we know $\exists R_1$ such that $S \longrightarrow^* \equiv R_1$ and $R_1 \Downarrow$ and $|R_1| = P'_1$. By $P'_1 \longrightarrow P'_2$ and I.H. we know $\exists R_2$ such that $R_1 \longrightarrow^* \equiv R_2$ and $|R_2| = P'_2$, and by $P'_2 \equiv P_2$ and Lemma A.11 we know $\exists R_3$ such that $R_2 \longrightarrow^* \equiv R_3$ and $|R_3| = P_2$. This implies $S \longrightarrow^* \equiv R_1 \longrightarrow^* \equiv R_2 \longrightarrow^* \equiv R_3$, i.e., $S \longrightarrow^+ R_3$ where $|R_3| = Q$. \square

Lemma 3.30 (Correspondence and Termination). $|S| \not\rightarrow$ and $S \Downarrow T$ implies $|T| \equiv |S|$

Proof. By induction on the number of reductions that lead to a safely-stable system $S \longrightarrow^n T$

$n = 0$: We have $S = T$ which implies $|S| = |T|$

$n = k + 1$: We have $S \longrightarrow R$ and $R \Downarrow T$. By $S \longrightarrow R$ and Cor. A.4 we get $|S| \equiv |R|$ and thus $|R| \not\rightarrow$. Hence by I.H. and $R \Downarrow T$ we get $|T| \equiv |R|$ and by transitivity we obtain $|T| \equiv |S|$. \square

A.3. The Logic.

Lemma A.12. When $S \not\rightarrow$ and $\Gamma, S \models \varphi$

- $S \equiv \lceil c!\vec{e} \rceil_{\rho} \| R$ implies $\uparrow c \in \mathbf{edg}(\varphi)$ or $\mathbf{edg}(\varphi)$ is undefined;
- $S \equiv (\mathbf{new} \vec{d}) \lceil c?\vec{x}.P \rceil_{\rho} \| R$ and $c \in \vec{d}$ implies $\uparrow c \in \mathbf{trg}(\varphi)$ or $\mathbf{trg}(\varphi)$ is undefined.

Proof. By induction on the structure of φ . \square

Lemma A.13. $\Gamma, S \models \varphi, S \not\rightarrow$ and $\Gamma, T \models \psi, T \not\rightarrow$ and $\varphi \perp \psi$ implies $\Gamma, S \| T \not\rightarrow$

Proof. Since $S \not\rightarrow$ and $T \not\rightarrow$, by Lemma A.5 we know that $S \| T \longrightarrow R$ for some R can only happen if:

$$S \equiv (\mathbf{new} \vec{d}) \left(\lceil c?\vec{x}.P \rceil_{\mu} \| S' \right) \text{ where } c \notin \vec{d} \text{ and } \downarrow c \in \mu \quad (\text{A.1})$$

$$T \equiv \lceil c!\vec{e} \rceil_{\rho} \| T' \text{ where } \uparrow c \in \rho \quad (\text{A.2})$$

or vice-versa. We here focus on the case where (A.1) and (A.2) have to hold; the dual case is identical. By $\varphi \perp \psi$ we know that $\mathbf{trg}(\varphi)$, $\mathbf{edg}(\varphi)$, $\mathbf{trg}(\psi)$ and $\mathbf{edg}(\psi)$ must all be defined. Thus by (A.1), $\Gamma, S \models \varphi$ and Lemma A.12 we must have $\uparrow c \in \mathbf{trg}(\varphi)$. Similarly by (A.2), $\Gamma, T \models \psi$ and Lemma A.12 we must have $\uparrow c \in \mathbf{edg}(\psi)$. This however would contradict $\varphi \perp \psi$ which requires that $\mathbf{trg}(\varphi) \cap \mathbf{edg}(\psi) = \emptyset$. Thus $S \| T \not\rightarrow$. \square

Lemma 4.11 (Merging Assertions). $\Gamma, S \models \varphi$ and $\Gamma, T \models \psi$ and $S \perp T$ and $\varphi \perp \psi$ implies $\Gamma, S \| T \models \varphi * \psi$

Proof. $S \perp T$ implies $S \| T$ is well-resourced. From $\Gamma, S \models \varphi, \Gamma, T \models \psi$ and Proposition 4.5 we know that $S \Downarrow S'$ and $T \Downarrow T'$ where $\Gamma, S' \models \varphi$ and $\Gamma, T' \models \psi$. Lemma A.13 we know also that $S \| T \Downarrow S' \| T'$ and the result follows by satisfaction on Figure 3. \square

A.4. **The Proof System.** Proofs for the derived rules from Section 5.1.

The proof for LCUT:

$$\frac{\frac{\Gamma; b \vdash \{\varphi_1\} S \{\psi\}}{\Gamma; b \vdash \{\varphi_1\} S \{\mathbf{emp} * \psi\}} \text{LIMP} \quad \frac{\Gamma; b \vdash \{\psi\} T \{\varphi_1\}}{\Gamma; b \vdash \{\mathbf{emp} * \psi\} T \{\varphi_2\}} \text{LIMP} \quad \mathbf{emp} \perp \psi \quad \mathbf{emp} \perp \varphi_2}{\frac{\Gamma; b \vdash \{\varphi_1 * \mathbf{emp}\} S \parallel T \{\mathbf{emp} * \varphi_2\}}{\Gamma; b \vdash \{\varphi_1\} S \parallel T \{\varphi_2\}} \text{LIMP}} \text{LPAR}$$

The proof for LSEP:

$$\frac{\frac{\Gamma; b \vdash \{\varphi_1\} S \{\psi_1\}}{\Gamma; b \vdash \{\varphi_1\} S \{\psi_1 * \mathbf{emp}\}} \text{LIMP} \quad \frac{\Gamma; b \vdash \{\varphi_2\} T \{\psi_2\}}{\Gamma; b \vdash \{\varphi_2 * \mathbf{emp}\} T \{\psi_2\}} \text{LIMP} \quad \psi_1 \perp \psi_2 \quad \varphi_2 \perp \mathbf{emp}}{\Gamma; b \vdash \{\varphi_1 * \varphi_2\} S \parallel T \{\psi_1 * \psi_2\}} \text{LPAR}$$

The proof for LOUTD:

$$\frac{b \models b \wedge e_1^* = e_1^* \wedge e_1^* = e_2^*}{\Gamma; b \vdash \{\mathbf{emp}\} \lceil c!e_1^* \rceil_\rho \{c\langle e_2^* \rangle\}} \text{LIMP} \quad \frac{\frac{\Gamma(c) \subseteq \rho}{\Gamma; b \wedge \vec{x} = \vec{e}_1^* \wedge \vec{x} = \vec{e}_2^* \vdash \{\mathbf{emp}\} \lceil c!e_2^* \rceil_\rho \{c\langle e_2^* \rangle\}} \text{LOUT} \quad \frac{\vec{x} \notin \mathbf{fn}(b) \cup \mathbf{fn}(\vec{e}_2^*, \vec{e}_1^*)}{\Gamma; b \wedge \vec{x} = \vec{e}_1^* \wedge \vec{x} = \vec{e}_2^* \vdash \{\mathbf{emp}\} \lceil c!\vec{x} \rceil_\rho \{c\langle \vec{e}_2^* \rangle\}} \text{LSUB}}{\Gamma; b \wedge \vec{e}_1^* = \vec{e}_1^* \wedge \vec{e}_1^* = \vec{e}_2^* \vdash \{\mathbf{emp}\} \lceil c!e_1^* \rceil_\rho \{c\langle \vec{e}_2^* \rangle\}} \text{LINST}} \text{LINST}$$

The proof for LIND:

$$\frac{\frac{\downarrow c \in \rho \quad \Gamma; b \vdash \{\varphi\} \lceil P\langle \vec{e}/\vec{x} \rangle \rceil_\rho \parallel S \{\psi\}}{\Gamma; b \vdash \{\varphi * c\langle \vec{e} \rangle\} \lceil c?\vec{x}.P \rceil_{\rho \setminus \Gamma(c)} \parallel S \{\psi\}} \text{LIN}}{\Gamma; b \vdash \{\varphi * c\langle \vec{e} \rangle\} T \{\psi\}} \text{LIMP}}$$

Lemma A.14. Assume that $\llbracket e^v/x \rrbracket$ is a substitution that non-deterministically substitutes either e or v for x . Then we have

$$S \llbracket v/x \rrbracket \longrightarrow T \llbracket v/x \rrbracket \text{ and } e \Downarrow v \text{ implies } S \llbracket e/x \rrbracket \longrightarrow R \text{ where } R = T \llbracket e^v/x \rrbracket$$

for some non-deterministic substitution $T \llbracket e^v/x \rrbracket$

Proof. By rule induction on $S \llbracket v/x \rrbracket \longrightarrow T \llbracket v/x \rrbracket$ □

Lemma A.15. $\Gamma, T \llbracket \vec{e}/\vec{x} \rrbracket \models \varphi$ and $\vec{e} \Downarrow \vec{v}$ and $T \not\rightarrow$ implies $\Gamma, T \llbracket \vec{e}/\vec{x} \rrbracket \models \varphi$

Proof. By induction on the structure of φ □

Lemma A.16. $\Gamma, T \llbracket \vec{v}/\vec{x} \rrbracket \models \varphi$ and $\vec{e} \Downarrow \vec{v}$ implies $\Gamma, T \llbracket \vec{e}/\vec{x} \rrbracket \models \varphi$

Proof. Follows from Lemma A.14 and Lemma A.15. □

Lemma A.17. $\Gamma, S \models \varphi$ and $d \notin \mathbf{dom}(\Gamma)$ implies $\Gamma, (\mathbf{new} d)S \models \varphi$

Proof. By induction on the structure of φ . For instance:

$c\langle \vec{e} \rangle$: We know $S \Downarrow \lceil c!e_1^* \rceil_\rho$ where $\vec{e} \Downarrow \vec{v}$, $e_1^* \Downarrow \vec{v}$ and $\Gamma(c) \subseteq \rho$. By cRES and then by cTGH and $d \notin \mathbf{nm}(c\langle \vec{e} \rangle) \cup \mathbf{nm}(\Gamma)$ we deduce

$$\begin{aligned} (\mathbf{new} d)S &\longrightarrow^* \equiv (\mathbf{new} d) \left(\lceil c!e_1^* \rceil_\rho \right) \equiv (\mathbf{new} d) \left(\lceil c!e_1^* \rceil_\rho \parallel \lceil \mathbf{nil} \rceil_\emptyset \right) \\ &(\mathbf{new} d) \left(\lceil c!e_1^* \rceil_\rho \parallel \lceil \mathbf{nil} \rceil_\emptyset \right) \longrightarrow \lceil c!e_1^* \rceil_{\rho \setminus \{\downarrow d, \uparrow d\}} \parallel (\mathbf{new} d) \left(\lceil \mathbf{nil} \rceil_\emptyset \right) \equiv \lceil c!e_1^* \rceil_{\rho \setminus \{\downarrow d, \uparrow d\}} \end{aligned}$$

Since $d \notin \mathbf{dom}(\Gamma)$ then by Definition 4.1.2, i.e., the environment is suitably closed, it follows that $\Gamma(c) \subseteq (\rho \setminus \{\downarrow d, \uparrow d\})$ and hence $\Gamma, \lceil c!e_1^* \rceil_{\rho \setminus \{\downarrow d, \uparrow d\}} \models c\langle \vec{e} \rangle$ and by Proposition 4.7 that $\Gamma, (\mathbf{new} d)S \models c\langle \vec{e} \rangle$. □

Definition A.18 (Permission Restriction).

$$S \setminus \mu \stackrel{\text{def}}{=} \begin{cases} \lceil P \rceil_{\rho \setminus \mu} & \text{if } S = \lceil P \rceil_{\rho} \\ S_1 \setminus \mu \parallel S_2 \setminus \mu & \text{if } S = S_1 \parallel S_2 \\ (\text{new } c)(T \setminus (\mu \setminus \{\downarrow c, \uparrow c\})) & \text{if } S = (\text{new } c)T \end{cases}$$

Proposition A.19. $S \setminus \mu \not\rightarrow_{\text{err}}$ implies $S \not\rightarrow_{\text{err}}$

Lemma A.20. $S \setminus \mu \not\rightarrow$ implies $\exists T. S \longrightarrow^* T \not\rightarrow$ where $(T \setminus \mu) \equiv (S \setminus \mu)$

Proof. By Proposition 3.14 we know,

$$S \setminus \mu \equiv (\text{new } \vec{d}) \left(\parallel_{i=0}^n \lceil c_i! \vec{e}_i \rceil_{\rho_i} \parallel_{j=0}^m \lceil c'_j? \vec{x}_j.P_j \rceil_{\mu_j} \right) \quad \text{where } \{c_1, \dots, c_n\} \cap \{c'_1, \dots, c'_m\} = \emptyset \quad (\text{A.3})$$

By system structural equivalence, \equiv , the only sub-systems in S that are abstracted away from $S \setminus \mu$ in $(\text{new } \vec{d}) \left(\parallel_{i=0}^n \lceil c_i! \vec{e}_i \rceil_{\rho_i} \parallel_{j=0}^m \lceil c'_j? \vec{x}_j.P_j \rceil_{\mu_j} \right)$ are those of the form $\lceil \text{nil} \rceil_{\rho}$ where $\rho \subseteq \mu$; the operation made these systems equivalent to $\lceil \text{nil} \rceil_{\emptyset}$ which could then be eliminated through scNIL . In S , sub-systems of the form $\lceil \text{nil} \rceil_{\rho}$ can still be eliminated through cDsc and then scNIL (as before), leaving us with the same array of mismatching confined output and input processes found in $S \setminus \mu$, less the restricted permissions. \square

Lemma A.21. $S \setminus \mu \longrightarrow T \setminus \mu$ implies $S \longrightarrow T$

Proof. By rule induction on $S \setminus \mu \longrightarrow T \setminus \mu$. \square

Lemma A.22. $(S \setminus \mu) \Downarrow T$ implies $S \Downarrow R$ where $R \setminus \mu \equiv T$

Proof. Follows from Lemma A.21, Lemma A.20 and Proposition A.19. \square

Lemma A.23. $\Gamma, (S \setminus \mu) \models \varphi$ implies $\Gamma, S \models \varphi$

Proof. By induction on the structure of φ using Lemma A.22. \square