Correct-by-construction Process Composition Using Classical Linear Logic Inference

Link:
[Link to publication record in Edinburgh Research Explorer](#)

Document Version:
Peer reviewed version

Published In:
28th International Symposium on Logic-Based Program Synthesis and Transformation

Correct-by-construction Process Composition Using Classical Linear Logic Inference

Petros Papapanagiotou[1] and Jacques Fleuriot[1]

School of Informatics, University of Edinburgh
10 Crichton Street, Edinburgh EH8 9AB, United Kingdom
{ppapapan,jdf}@inf.ed.ac.uk

Abstract. The need for rigorous process composition is encountered in many situations pertaining to the development and analysis of complex systems. We discuss the use of Classical Linear Logic (CLL) for correct-by-construction resource-based process composition, with guaranteed deadlock freedom, systematic resource accounting, and concurrent execution. We introduce algorithms to automate the necessary inference steps for binary compositions of processes in parallel, conditionally, and in sequence. We combine decision procedures and heuristics to achieve intuitive and practically useful compositions in an applied setting.

Keywords: process modelling, composition, correct by construction, workflow, linear logic

1   Introduction

The ideas behind process modelling and composition are common across a variety of domains, including program synthesis, software architecture, multi-agent systems, web services, and business processes. Although the concept of a "process" takes a variety of names – such as agent, role, action, activity, and service – across these domains, in essence, it always captures the idea of an abstract, functional unit. Process composition then involves the combination and connection of these units to create systems that can perform more complex tasks. We typically call the resulting model a (*process*) *workflow*. Viewed from this standpoint, resource-based process composition then captures a structured model of the *resource flow* across the components, focusing on the resources that are created, consumed, or passed from one process to another within the system.

Workflows have proven useful tools for the design and implementation of complex systems by providing a balance between an intuitive **abstract model**, typically in diagrammatic form, and a **concrete implementation** through process automation. Evidence can be found, for example, in the modelling of clinical care pathways where workflows can be both understandable by healthcare stakeholders and yet remain amenable to formal analysis [10,15].

A scalable approach towards establishing trust in the correctness of the modelled system is that of *correct-by-construction* engineering [12,26]. In general, this

refers to the construction of systems in a way that guarantees correctness properties about them at design time. In this spirit, we have developed the `WorkflowFM` system for correct-by-construction process composition [21]. It relies on Classical Linear Logic (see Section 2.1) to rigorously compose abstract process specifications in a way that:

1. systematically accounts for resources and exceptions;
2. prevents deadlocks;
3. results in a concrete workflow where processes are executed concurrently.

The inference is performed within the proof assistant HOL Light, which offers systematic guarantees of correctness for every inference step [11]. The logical model can be translated through a process calculus to a concrete workflow implementation in a host programming language.

There are numerous aspects to and components in the `WorkflowFM` system, including, for instance, the diagrammatic interface (as shown in Fig. 1), the code translator, the execution engine, the process calculus correspondence, and the architecture that brings it all together [21]. In this particular paper we focus on the proof procedures that make such resource-based process compositions feasible and accessible. These are essential for creating meaningful workflow models with the correctness-by-construction properties highlighted above, but without the need for tedious manual CLL reasoning. Instead, the user can use high level composition actions triggered by simple, intuitive mouse gestures and without the need to understand the underlying proof, which is guaranteed to be correct thanks to the rigorous environment of HOL Light.

It is worth emphasizing that our work largely aims at tackling pragmatic challenges in real applications as opposed to merely establishing theoretical facts. We rely on existing formalisms, such as the proofs-as-processes theory described below, in our attempt to exploit its benefits in real world scenarios. As a result, the vast majority of our design decisions are driven by practical experience and the different cases we have encountered in our projects.

Table 1 is a list of some of our case studies in the healthcare and manufacturing domain that have driven the development of `WorkflowFM`. It includes an indication of the size of each case study based on (1) the number of (atomic) component processes, (2) the number of different types of resources involved in the inputs and outputs of the various processes (see Section 3), (3) the number of binary composition actions performed to construct the workflows (see Section 4), and (4) the total number of composed workflows.

2 Background

The systematic accounting of resources in our approach can be demonstrated through a hypothetical example from the healthcare domain [21]. Assume a process `DeliverDrug` that corresponds to the delivery of a drug to a patient by a nurse. Such a process requires information about the *Patient*, the *Dosage* of the drug, and some reserved *NurseTime* for the nurse to deliver the drug. The

Case study theme	Processes	Resource types	Actions	Workflows
Patient handovers	9	16	13	2
Tracheostomy care pathway	33	47	32	3
HIV care pathways	128	129	121	13
Pen manufacturing *(ongoing)*	42	45	60	20
Total	**212**	**237**	**226**	**38**

Table 1. Sample case studies and an indication of their size.

possible outcomes are that either the patient is *Treated* or that the drug *Failed*. In the latter case, we would like to apply the `Reassess` process, which, given some allocated clinician time (*ClinTime*) results in the patient being *Reassessed*. A graphical representation of these 2 processes, where dashed edges denote the optional outcomes of `DeliverDrug`, is shown at the top of Fig. 1.

If we were to compose the 2 processes in a workflow where the drug failure is always handled by `Reassess`, what would be the specification (or specifically the output) of the composite process?
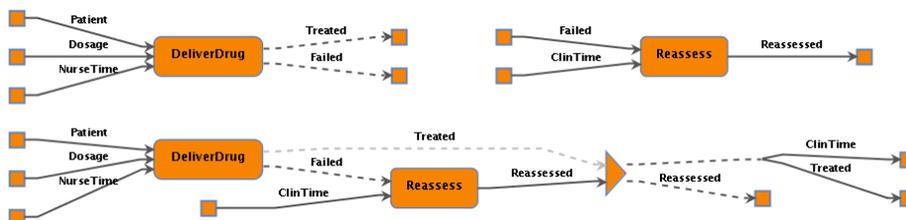

Fig. 1. The visualisation of the `DeliverDrug` and `Reassess` processes (top) and their sequential composition. The auxiliary triangle helps properly display the output.

Given the workflow representation in Fig. 1, one may be inclined to simply connect the *Failed* edge of `DeliverDrug` to the corresponding edge of `Reassess`, leading to an overall output of either *Treated* or *Reassessed*. However, this would be erroneous, as the input *ClinTime*, is consumed in the composite process even if `Reassess` is never used. Using our CLL-based approach, the workflow output is either *Reassessed* which occurs if the drug failed, or *Treated* coupled with the unused *ClinTime*, as shown at the bottom of Fig. 1 [21].

Systematically accounting for such unused resources is non-trivial, especially considering larger workflows with tens or hundreds of processes and many different outcomes. The CLL inference rules enforce this by default and the proof reflects the level of reasoning required to achieve this. In addition, the process code generated from this synthesis is fully asynchronous and deadlock-free, and relies on the existence of concrete implementations of `DeliverDrug` and `Reassess`.

2.1 Classical Linear Logic

Linear Logic, as proposed by Girard [9], is a refinement to classical logic where the rules of contraction and weakening are limited to the modalities ! and ?. Propositions thus resemble resources that cannot be ignored or copied arbitrarily.

In this work, we use a one-sided sequent calculus version of the multiplicative additive fragment of propositional CLL without units (MALL). Although there exist process translations of full CLL and even first-order CLL, the MALL fragment allows enough expressiveness while keeping the reasoning complexity at a manageable level (MALL is PSPACE-complete whereas full CLL is undecidable [14]). The inference rules for MALL are presented in Fig. 2.

$$\frac{}{\vdash A^\perp, A} \ Id \qquad\qquad \frac{\vdash \Gamma, C \quad \vdash \Delta, C^\perp}{\vdash \Gamma, \Delta} \ Cut$$

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B} \ \otimes \qquad\qquad \frac{\vdash \Gamma, A^\perp, B^\perp}{\vdash \Gamma, (A \otimes B)^\perp} \ \mathfrak{N}$$

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B} \ \oplus_L \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B} \ \oplus_R \qquad \frac{\vdash \Gamma, A^\perp \quad \vdash \Gamma, B^\perp}{\vdash \Gamma, (A \oplus B)^\perp} \ \&$$

Fig. 2. One-sided sequent calculus versions of the CLL inference rules.

In this version of MALL, linear negation $(\cdot^\perp)$ is defined as a syntactic operator with no inference rules, so that both A and $A^\perp$ are considered atomic formulas. The de Morgan style equations in Fig. 3 provide a *syntactic* equivalence of formulas involving negation [27]. This allows us to use syntactically equivalent formulas, such as $A^\perp \mathfrak{N} B^\perp$ and $(A \otimes B)^\perp$ interchangeably. In fact, in the proofs presented in this paper we choose to present formulas containing $\otimes$ and $\oplus$ over their counterparts $\mathfrak{N}$ and $\&$ due to the polarity restrictions we introduce in Section 3.

$$(A^\perp)^\perp \equiv A \qquad (A \otimes B)^\perp \equiv A^\perp \mathfrak{N} B^\perp \qquad (A \oplus B)^\perp \equiv A^\perp \ \& \ B^\perp$$
$$(A \mathfrak{N} B)^\perp \equiv A^\perp \otimes B^\perp \qquad (A \ \& \ B)^\perp \equiv A^\perp \oplus B^\perp$$

Fig. 3. The equations used to define linear negation for MALL.

In the 90s, Abramsky, Bellin and Scott developed the so-called proofs-as-processes paradigm [2,4]. It involved a correspondence between CLL inference and concurrent processes in the π-calculus [18]. They proved that cut-elimination in a CLL proof corresponds to reductions in the π-calculus translation, which in turn correspond to communication between concurrent processes. As a result, π-calculus terms constructed via CLL proofs are inherently free of deadlocks.

The implications of the proofs-as-processes correspondence have been the subject of recent research in concurrent programming by Wadler [28], Pfenning et al. [3,5,25], Dardha [7,8] and others. Essentially, each CLL inference step can be translated to an executable workflow, with automatically generated code to

appropriately connect the component processes. As a result, the CLL proofs have a direct correspondence to the "piping", so to speak, that realises the appropriate resource flow between the available processes, such that it does not introduce deadlocks, accounts for all resources explicitly, and maximizes runtime concurrency. The current paper examines CLL inference and we take the correspondence to deadlock-free processes for granted.

2.2 Related work

Diagrammatic languages such as BPMN [20] are commonly used for the description of workflows in different organisations. However, they typically lack rigour and have limited potential for formal verification [23]. Execution languages such as BPEL [19] and process calculi such as Petri Nets [1] are often used for workflow management in a formal way and our CLL approach could potentially be adapted to work with these. Linear logic has been used in the context of web service composition [22], but in a way that diverges significantly from the original theory and compromises the validity of the results. Finally, the way the resource flow is managed through our CLL-based processes is reminiscent of monad-like structures such as Haskell's arrows[1]. One of the key differences is the lack of support for optional resources, which is non-trivial as we show in this paper.

3 Process Specification

Since CLL propositions can naturally represent resources, CLL sequents can be used to represent processes, with each literal representing a type of resource that is involved in that process. These abstract types can have a concrete realisation in the host programming language, from primitive to complicated objects.

Our approach to resource-based composition is to construct CLL specifications of abstract processes based on their inputs (and preconditions) and outputs (and effects), also referred to as IOPEs. This is standard practice in various process formalisms, including WSDL for web services [6], OWL-S for Semantic Web services [16], PDDL for actions in automated planning [17], etc.

The symmetry of linear negation as shown in Fig. 3 can be used to assign a *polarity* to each CLL connective in order to distinctly specify input and output resources. We choose to treat negated literals, $\invamp$, and & as **inputs**, and positive literals, $\otimes$, and $\oplus$ as **outputs**, with the following intuitive interpretation:

- Multiplicative conjunction (*tensor* $\otimes$) indicates a pair of parallel outputs.
- Additive disjunction (*plus* $\oplus$) indicates exclusively optional outputs (alternative outputs or exceptions).
- Multiplicative disjunction (*par* $\invamp$) indicates a pair of simultaneous inputs.
- Additive conjunction (*with* &) indicates exclusively optional input.

---

[1] https://www.haskell.org/arrows

Based on this, a process can be specified as a CLL sequent consisting of a list of input formulas and a **single** output formula. In this, the order of the literals does not matter, so long as they obey the polarity restrictions (all but exactly one are negative). In practice, we treat sequents as multisets of literals and manage them using particular multiset reasoning techniques in HOL Light. The description of these techniques is beyond the scope of this paper.

The polarity restrictions imposed on our process specifications match the specification of Laurent's Polarized Linear Logic (LLP) [13], and has a proven logical equivalence to the full MALL. Moreover, these restrictions match the programming language paradigm of a *function* that can have multiple input arguments and returns a single (possibly composite) result.

4   Process Composition

Using CLL process specifications as assumptions, we can produce a composite process specification using forward inference. Each of the CLL inference rules represent a logically legal way to manipulate and compose such specifications.

The axiom $\vdash A, A^{\perp}$ represents the so-called *axiom buffer*, a process that receives a resource of type A and outputs the same resource unaffected.

Unary inference rules, such as the $\oplus_L$ rule, correspond to manipulations of a single process specification. For example, the $\oplus_L$ rule (see Fig. 2) takes a process P specified by $\vdash \Gamma, A$, i.e. a process with some inputs Γ and an output A, and produces a process $\vdash \Gamma, A \oplus B$, i.e. a process with the same inputs Γ and output either A or B. Note that the produced composite process relies on P and therefore will in practice always produce A and never B.

Binary inference rules, such as the $\otimes$ rule, correspond to binary process composition. The $\otimes$ rule in particular (see Fig. 2) takes a process P specified by $\vdash \Gamma, A$ and another process Q specified by $\vdash \Delta, B$ and composes them, so that the resulting process $\vdash \Gamma, \Delta, A \otimes B$ has all their inputs Γ and Δ and a simultaneous output $A \otimes B$. Notably, the *Cut* rule corresponds to the composition of 2 processes in sequence, where one consumes a resource A given by the other.

Naturally, these manipulations and compositions are primitive and restricted. Constructing meaningful compositions requires several rule applications and, therefore, doing this manually would be a very tedious and impractical task. Our work focuses on creating high level actions that use CLL inference to automatically produce binary process compositions that are correct-by-construction based on the guarantees described above. More specifically, we introduce actions for parallel (`TENSOR`), conditional (`WITH`), and sequential composition (`JOIN`).

Since we are using forward inference, there are infinitely many ways to apply the CLL rules and therefore infinite possible compositions. We are interested in producing compositions that are intuitive for the user. It is practically impossible to produce a formal definition of what these compositions should be. Instead, as explained earlier, we rely on practical experience and user feedback from the various case studies for workflow modelling (see Table 1).

Based on this, we have introduced a set of what can be viewed as *unit tests* for our composition actions, which describe the expected and logically valid results of example compositions. As we explore increasingly complex examples in practice, we augment our test set and ensure our algorithms satisfy them. *Selected* unit tests for the `WITH` and `JOIN` actions are shown in Tables 2 and 3 respectively. Moreover, as a general principle, our algorithms try to *maximize* resource usage, i.e. involve as many resources as possible, and *minimize* the number of rule applications to keep the corresponding process code more compact.

P	Q	Result
$\vdash X^{\perp}, Z$	$\vdash Y^{\perp}, Z$	$\vdash (X \oplus Y)^{\perp}, Z$
$\vdash X^{\perp}, Z$	$\vdash Y^{\perp}, W$	$\vdash (X \oplus Y)^{\perp}, A^{\perp}, B^{\perp}, Z \oplus W$
$\vdash X^{\perp}, A^{\perp}, B^{\perp}, Z$	$\vdash Y^{\perp}, Z$	$\vdash (X \oplus Y)^{\perp}, Z \oplus (Z \otimes A \otimes B)$
$\vdash X^{\perp}, A^{\perp}, Z$	$\vdash Y^{\perp}, B^{\perp}, W$	$\vdash (X \oplus Y)^{\perp}, A^{\perp}, B^{\perp}, (Z \otimes B) \oplus (W \otimes A)$
$\vdash X^{\perp}, A^{\perp}, C^{\perp}, Z$	$\vdash Y^{\perp}, B^{\perp}, W$	$\vdash (X \oplus Y)^{\perp}, A^{\perp}, B^{\perp}, C^{\perp}, (Z \otimes B) \oplus (W \otimes A \otimes C)$
$\vdash X^{\perp}, A^{\perp}, C^{\perp}, Z$	$\vdash Y^{\perp}, B^{\perp}, C^{\perp}, W$	$\vdash (X \oplus Y)^{\perp}, A^{\perp}, B^{\perp}, C^{\perp}, (Z \otimes B) \oplus (W \otimes A)$
$\vdash X^{\perp}, A^{\perp}, C^{\perp}, C^{\perp}, Z$	$\vdash Y^{\perp}, B^{\perp}, C^{\perp}, W$	$\vdash (X \oplus Y)^{\perp}, A^{\perp}, B^{\perp}, C^{\perp}, C^{\perp}, (Z \otimes B) \oplus (W \otimes A \otimes C)$
$\vdash X^{\perp}, A \otimes B$	$\vdash Y^{\perp}, B \otimes A$	$\vdash (X \oplus Y)^{\perp}, A \otimes B$
$\vdash X^{\perp}, A^{\perp}, Z \otimes A$	$\vdash Y^{\perp}, Z$	$\vdash (X \oplus Y)^{\perp}, A^{\perp}, Z \otimes A$
$\vdash X^{\perp}, A^{\perp}, A \otimes Z$	$\vdash Y^{\perp}, Z$	$\vdash (X \oplus Y)^{\perp}, A^{\perp}, A \otimes Z$
$\vdash X^{\perp}, A^{\perp}, Z \oplus (Z \otimes A)$	$\vdash Y^{\perp}, Z$	$\vdash (X \oplus Y)^{\perp}, A^{\perp}, Z \oplus (Z \otimes A)$

Table 2. Examples of the expected result of the `WITH` action between $X^{\perp}$ of a process `P` and $Y^{\perp}$ of a process `Q`.

All our algorithms are implemented within the Higher Order Logic proof tactic system of HOL Light. As a result, the names of some methods have the `_TAC` suffix, which is conventionally used when naming HOL Light tactics.

5 Auxiliary Processes

During composition, we often need to construct auxiliary processes that manipulate the structure of a CLL type in particular ways. We have identified 2 types of such processes: *buffers* and *filters*.

Buffers: Similarly to the axiom buffer introduced in the previous section, *composite buffers* (or simply *buffers*) can carry any composite resource without affecting it. This is useful when a process is unable to handle the entire type on its own, and some resources need to be simply *buffered* through. For example, if a process needs to handle a resource of type $A \otimes B$, but only has an input of type $A^{\perp}$, then B will be handled by a buffer.

More formally, buffers are processes specified by $\vdash A^{\perp}, A$, where A is arbitrarily complex. Such lemmas are always provable in CLL for any formula A. We have introduced an automatic procedure `BUFFER_TAC` that can accomplish this, but omit the implementation details in the interest of space and in favour of the more interesting composition procedures that follow.

We also introduce the concept of a *parallel buffer*, defined as a process $\vdash A_1^\perp, A_2^\perp, ..., A_n^\perp, A_1 \otimes A_2 \otimes ... \otimes A_n$. Such buffers are useful when composing processes with an optional output (see Section 8.3). Their construction can also be easily automated with a decision procedure we call `PARBUF_TAC`.

Filters: Often during process composition by proof, resources need to match exactly for the proof to proceed. In some cases, composite resources may not match exactly, but may be manipulated using the CLL inference rules so that they end up matching. For example, the term $A \otimes B$ does not directly match $B \otimes A$. However, both terms intuitively represent resources A and B in parallel. This intuition is reflected formally to the commutativity property of $\otimes$, which is easily provable in CLL: $\vdash (A \otimes B)^\perp, B \otimes A$. We can then use the *Cut* rule with this property to convert an output of type $A \otimes B$ to $B \otimes A$ (similarly for inputs).

We call such lemmas that are useful for converting CLL types to equivalent ones, *filters*. In essence, a filter is any provable CLL lemma that preserves our polarity restrictions. We prove such lemmas automatically using the proof strategies developed by Tammet [24].

We give some examples of how filters are used to match terms as we go through them below. However, as a general rule the reader may assume that, for the remainder of this paper, by "equal" or "matching" terms we refer to terms that are equal modulo the use of filters.

A main consequence of this is that our algorithms often attempt to match literals that do not match. For example, the attempt to compose $\vdash A^\perp, B$ in sequence with $\vdash C^\perp, D^\perp, E$ would generate and try to prove 2 false conjectures $\vdash B^\perp, C$ and $\vdash B^\perp, D$ in an effort to match the output B with any of the 2 inputs $C^\perp$ and $D^\perp$ before failing[2]. This highlights the need for an efficient proof procedure for filters, with an emphasis on early failure.

6 Parallel Composition - The `TENSOR` Action

The `TENSOR` action corresponds to the parallel composition of two processes so that their outputs are provided in parallel. It trivially relies on the tensor ($\otimes$) inference rule. Assuming 2 processes, $\vdash A^\perp, C^\perp, D$ and $\vdash B^\perp, E$, the `TENSOR` action will perform the following composition:

$$\frac{\vdash A^\perp, C^\perp, D \quad \vdash B^\perp, E}{\vdash A^\perp, B^\perp, C^\perp, D \otimes E} \otimes$$

7 Conditional Composition - The `WITH` Action

The `WITH` action corresponds to the *conditional* composition of two processes. This type of composition is useful in cases where each of the components of an optional output of a process needs to be handled by a different receiving process.

---

[2] In practice, the user will have to select a matching input to attempt such a composition (see Section 8).

P	Pr.	Q	Selected Input	Result
$\vdash X^\perp, A$		$\vdash A^\perp, Y$	$A^\perp$	$\vdash X^\perp, Y$
$\vdash X^\perp, A \otimes B$	L	$\vdash A^\perp, Y$	$A^\perp$	$\vdash X^\perp, Y \otimes B$
$\vdash X^\perp, A \otimes B \otimes C$	L	$\vdash A^\perp, Y$	$A^\perp$	$\vdash X^\perp, Y \otimes B \otimes C$
$\vdash X^\perp, A \oplus B$	L	$\vdash A^\perp, C^\perp, Y$	$A^\perp$	$\vdash X^\perp, C^\perp, Y \oplus (C \otimes B)$
$\vdash X^\perp, A \oplus B$	R	$\vdash B^\perp, C^\perp, Y$	$B^\perp$	$\vdash X^\perp, C^\perp, (C \otimes A) \oplus Y$
$\vdash X^\perp, A \oplus B$	L	$\vdash (B \oplus A)^\perp, Y$	$(B \oplus A)^\perp$	$\vdash X^\perp, Y$
$\vdash X^\perp, A \oplus (B \otimes C)$	L	$\vdash (B \oplus A)^\perp, Y$	$(B \oplus A)^\perp$	$\vdash X^\perp, Y \oplus (B \otimes C)$
$\vdash X^\perp, A \oplus (B \otimes C)$	RL	$\vdash (B \oplus A)^\perp, Y$	$(B \oplus A)^\perp$	$\vdash X^\perp, A \oplus (Y \otimes C)$
$\vdash X^\perp, A \oplus B$	L	$\vdash (C \oplus A \oplus D)^\perp, Y$	$(C \oplus A \oplus D)^\perp$	$\vdash X^\perp, Y \oplus B$
$\vdash X^\perp, C \oplus (A \otimes B)$	L	$\vdash C^\perp, A \otimes B$	$C^\perp$	$\vdash X^\perp, A \otimes B$
$\vdash X^\perp, C \oplus (A \otimes B)$	L	$\vdash C^\perp, B \otimes A$	$C^\perp$	$\vdash X^\perp, B \otimes A$
$\vdash X^\perp, C \oplus (A \otimes (B \oplus D))$	L	$\vdash C^\perp, (B \oplus D) \otimes A$	$C^\perp$	$\vdash X^\perp, (B \oplus D) \otimes A$
$\vdash X^\perp, C \oplus (A \otimes B)$	L	$\vdash C^\perp, Y \oplus (B \otimes A)$	$C^\perp$	$\vdash X^\perp, Y \oplus (B \otimes A)$
$\vdash X^\perp, C \oplus (A \otimes B)$	L	$\vdash C^\perp, (B \otimes A) \oplus Y$	$C^\perp$	$\vdash X^\perp, (B \otimes A) \oplus Y$
$\vdash X^\perp, (A \otimes B) \oplus C$	R	$\vdash C^\perp, Y \oplus (B \otimes A)$	$C^\perp$	$\vdash X^\perp, Y \oplus (B \otimes A)$
$\vdash X^\perp, (A \otimes B) \oplus C$	R	$\vdash C^\perp, (B \otimes A) \oplus Y$	$C^\perp$	$\vdash X^\perp, (B \otimes A) \oplus Y$
$\vdash X^\perp, C \oplus (A \otimes B)$	L	$\vdash C^\perp, Y \oplus (B \otimes A)$	$C^\perp$	$\vdash X^\perp, Y \oplus (B \otimes A)$
$\vdash X^\perp, A \oplus B$	L	$\vdash A^\perp, C^\perp, Y$	$A^\perp$	$\vdash X^\perp, Y \oplus (C \otimes B)$

Table 3. Examples of the expected result of the `JOIN` action between a process `P` and a process `Q`. Column *Pr.* gives the *priority* parameter (see Section 8.4).

For example, assume a process `S` has an optional output $A \oplus C$ where C is an exception. We want A to be handled by some process `P`, for example specified by $\vdash A^\perp, B^\perp, X$, while another process `Q` specified by $\vdash C^\perp, Y$ plays the role of the exception handler for exception C. For this to happen, we need to compose `P` and `Q` together using the `WITH` action so that $A \oplus C$ from `S` can be taken as input in one go. This composition can be viewed as the construction of an *if-then* statement where if A is provided then `P` will be executed (assuming B is also provided), and if C is provided then `Q` will be executed in a mutually exclusive choice. The generated proof tree for this particular example is the following:

$$
\cfrac{\cfrac{\overline{\vdash A^\perp, B^\perp, X} \; \texttt{P}}{\vdash A^\perp, B^\perp, X \oplus (Y \otimes B)} \oplus R \quad \cfrac{\cfrac{\overline{\vdash C^\perp, Y} \; \texttt{Q} \quad \overline{\vdash B^\perp, B} \; Id}{\vdash C^\perp, B^\perp, Y \otimes B} \otimes}{\vdash C^\perp, B^\perp, X \oplus (Y \otimes B)} \oplus L}{\vdash (A \oplus C)^\perp, B^\perp, X \oplus (Y \otimes B)} \; \& \tag{1}
$$

The `WITH` action fundamentally relies on the & rule of CLL. The following derivation allows us to compose 2 processes with different outputs X and Y:

$$
\cfrac{\cfrac{\vdash \Gamma, A^\perp, X}{\vdash \Gamma, A^\perp, X \oplus Y} \oplus L \quad \cfrac{\vdash \Gamma, C^\perp, Y}{\vdash \Gamma, C^\perp, X \oplus Y} \oplus R}{\vdash \Gamma, (A \oplus C)^\perp, X \oplus Y} \; \& \tag{2}
$$

The particularity of the & rule is that the context Γ, i.e. all the inputs except the ones involved in the `WITH` action, must be the same for both the involved processes. In practice, this means we need to account for unused inputs. In the example above, `P` apart from input $A^\perp$ has another input $B^\perp$ which is missing

from Q. In the conditional composition of P and Q, if exception C occurs, the provided B will not be consumed since P will not be invoked. In this case, we use a buffer to let B pass through together with the output Y of Q.

More generally, in order to apply the & rule to 2 processes P and Q, we need to minimally adjust their contexts Γ_P and Γ_Q (i.e. their respective multisets of inputs excluding the ones that will be used in the rule) so that they end up being the same $\Gamma = \Gamma_P \cup \Gamma_Q$.

In order to accomplish this for process Q, we calculate the multiset of "*missing*" inputs $\Delta_Q = \Gamma_P \backslash \Gamma_Q$. In the previous example in (1), we obtain $\Delta_Q = \{C^\perp\}$. We then construct a parallel buffer (see Section 5) of type $\otimes \Delta_Q^{\perp 3}$ (converting every input in Δ_Q to an output) using `PARBUF_TAC`. In the example, this is an atomic C buffer. The parallel composition between this buffer and Q results in the process $\vdash \Gamma_Q, \Delta_Q, C^\perp, Y \otimes (\otimes \Delta_Q^\perp)$. The same calculation for P yields $\Delta_P = \emptyset$ so no change is required for P.

Since $\Gamma_P \uplus \Delta_P = \Gamma_Q \uplus \Delta_Q = \Gamma$ (where $\uplus$ denotes multiset union), the & rule is now applicable and derivation (2) yields the following process:

$$\vdash \Gamma, (A \oplus C)^\perp, \left(X \otimes (\otimes \Delta_P^\perp)\right) \oplus \left(Y \otimes (\otimes \Delta_Q^\perp)\right) \tag{3}$$

The output Y of Q has now been paired with the buffered resources Δ_Q.

Finally, we consider the special case where the following holds:

$$\left(X \otimes (\otimes \Delta_P^\perp)\right) = \left(Y \otimes (\otimes \Delta_Q^\perp)\right) = G \tag{4}$$

In this case, the output of the composition in (3) will be $G \oplus G$. Instead we can apply the & directly without derivation (2), yielding the simpler output G.

Note that, as discussed in Section 5, (4) above does not strictly require equality. The special case can also be applied if we can prove and use the filter $\vdash \left(X \otimes (\otimes \Delta_P^\perp)\right)^\perp, \left(Y \otimes (\otimes \Delta_P^\perp)\right)$.

These results and the complexity underlying their construction demonstrate the non-trivial effort needed to adhere to CLL's systematic management of resources and, more specifically, its systematic accounting of unused resources.

8 Sequential Composition - The `JOIN` Action

The `JOIN` action reflects the connection of two processes in sequence, i.e. where (some of) the outputs of a process are connected to (some of) the corresponding inputs of another. More generally, we want to compose a process P with specification $\vdash \Gamma, X$, i.e. with some (multiset of) inputs Γ and output X in sequence with a process Q with specification $\vdash \Delta, C^\perp, Y$, i.e. with an input $C^\perp$, output Y, and (possibly) more inputs in context Δ. We also assume the user selects a subterm A of X in P and a matching subterm A of the input $C^\perp$ in Q.

The strategy of the algorithm behind the `JOIN` action is to construct a new input for Q based on the chosen $C^\perp$ such that it directly matches the output X

---

[3] $\otimes\{a_1, ..., a_n\}^\perp = a_1^\perp \otimes ... \otimes a_n^\perp$

of P (and prioritizing the output selection A). This will enable the application of the *Cut* rule, which requires the cut literal to match exactly. In what follows, we present how different cases for X are handled.

8.1 Atomic or Matching Output

If X is atomic, a straighforward use of the *Cut* rule is sufficient to connect the two processes. For example, the `JOIN` action between $\vdash A^\perp, B^\perp, X$ and $\vdash X^\perp, Z$ results in the following proof:

$$\cfrac{\cfrac{}{\vdash A^\perp, B^\perp,\ X}\ \text{P} \quad \cfrac{}{\vdash X^\perp, Z}\ \text{Q}}{\vdash A^\perp, B^\perp, Z}\ Cut$$

The same approach can be applied more generally for any non-atomic X as long as a matching input of type $X^\perp$ (including via filtering) is selected in Q.

8.2 Parallel Output

If X is a parallel output, such as $B \otimes C$, we need to manipulate process Q so that it can receive an input of type $(B \otimes C)^\perp$.

If Q has both inputs $B^\perp$ and $C^\perp$, then we can use the $\invamp$ rule to combine them. For example, the generated proof tree of the `JOIN` action between $\vdash A^\perp, D^\perp, B \otimes C$ and $\vdash B^\perp, C^\perp, E^\perp, Y$ is the following:

$$\cfrac{\cfrac{}{\vdash A^\perp, D^\perp, B \otimes C}\ \text{P} \quad \cfrac{\cfrac{}{\vdash B^\perp, C^\perp, E^\perp, Y}\ \text{Q}}{\vdash (B \otimes C)^\perp, E^\perp, Y}\ \invamp}{\vdash A^\perp, D^\perp, E^\perp, Y}\ Cut$$

As previously mentioned, the `JOIN` action attempts to connect the output of P to Q maximally, i.e. both B and C, regardless of the user choice. The user may, however, want to only connect one of the two resources. We have currently implemented this approach as it is the most commonly used in practice, but are investigating ways to enable better control by the user.

If Q has only one of the two inputs, for example $B^\perp$, i.e. Q is of the form $\vdash \Delta, B^\perp, Y$ and $C^\perp \notin \Delta$, then C must be buffered. In this case, we use the following derivation:

$$\cfrac{\cfrac{\cfrac{}{\vdash \Delta, B^\perp, Y}\ \text{Q} \quad \cfrac{\vdots}{\vdash C^\perp, C}\ ^{\text{BUFFER\_TAC}}}{\vdash \Delta, B^\perp, C^\perp, Y \otimes C}\ \otimes}{\vdash \Delta, (B \otimes C)^\perp, Y \otimes C}\ \invamp \tag{5}$$

We use `BUFFER_TAC` from Section 5 to prove the buffer of C.

Depending on the use of the $\otimes$ rule in (5), the resulting output could be either $Y \otimes C$ or $C \otimes Y$. We generally try to match the form of P's output, so in this case we would choose $Y \otimes C$ to match $B \otimes C$. Our algorithm keeps track of this orientation through the `orient` parameter (see Section 8.4).

8.3 Optional Output

If X is an optional output, such as $B \oplus C$, then we need to manipulate process Q to synthesize an input $(B \oplus C)^\perp$. Assume Q can handle B (symmetrically for C) and thus has specification $\vdash \Delta, B^\perp, Y$. We construct a parallel buffer (using PARBUF_TAC, see Section 5) of type $(\otimes \Delta^\perp) \otimes C$ (converting all inputs in Δ to outputs). We then apply derivation (2) as follows:

$$
\cfrac{
\cfrac{
\cfrac{}{\vdash \Delta, B^\perp, Y} \text{ Q}
}{\vdash \Delta, B^\perp, Y \oplus \left((\otimes \Delta^\perp) \otimes C\right)} \oplus L
\qquad
\cfrac{
\cfrac{\substack{\text{PARBUF\_TAC} \\ \vdots}}{\vdash \Delta, C^\perp, (\otimes \Delta^\perp) \otimes C}
}{\vdash \Delta, C^\perp, Y \oplus \left((\otimes \Delta^\perp) \otimes C\right)} \oplus R
}{\vdash \Delta, (B \oplus C)^\perp, Y \oplus \left((\otimes \Delta^\perp) \otimes C\right)} \&
\tag{6}
$$

Similarly to the WITH action, the particular structure of the & rule ensures the systematic management of unused resources. In the example above, if C is received then Q will never be executed. As a result, any resources in Δ will remain unused and need to be buffered together with C. This is the reason behind the type $(\otimes \Delta^\perp) \otimes C$ of the constructed buffer (as opposed to plainly using type C).

The proof tree of an example of the JOIN action between process P specified by $\vdash A^\perp, D^\perp, B \oplus C$ and process Q specified by $\vdash B^\perp, E^\perp, Y$ is shown below:

$$
\cfrac{
\cfrac{}{\vdash A^\perp, D^\perp, B \oplus C} \text{ P}
\quad
\cfrac{
\cfrac{
\cfrac{
\cfrac{}{\vdash B^\perp, E^\perp, Y} \text{ Q}
}{\vdash B^\perp, E^\perp, Y \oplus (C \otimes E)} \oplus L
\quad
\cfrac{
\cfrac{\cfrac{}{\vdash C^\perp, C} Id \quad \cfrac{}{\vdash E^\perp, E} Id}{\vdash C^\perp, E^\perp, C \otimes E} \otimes
}{\vdash C^\perp, E^\perp, Y \oplus (C \otimes E)} \oplus R
}{\vdash (B \oplus C)^\perp, E^\perp, Y \oplus (C \otimes E)} \&
}{\vdash A^\perp, D^\perp, E^\perp, Y \oplus (C \otimes E)} Cut
$$

It is interesting to consider a couple of special cases.

Case 1: If $\vdash \Delta, C^\perp, Y$ is a parallel buffer, (6) can be simplified as follows:

$$
\cfrac{
\cfrac{}{\vdash \Delta, B^\perp, Y} \text{ Q}
\qquad
\cfrac{\substack{\text{PARBUF\_TAC} \\ \vdots}}{\vdash \Delta, C^\perp, Y}
}{\vdash \Delta, (B \oplus C)^\perp, Y} \&
\tag{7}
$$

This may occur, for example, if $\Delta = \emptyset$ and $Y = C$. Such cases arise in processes used to recover from an exception. For instance, a recovery process $\vdash Exception^\perp, Resource$ can convert an output $Resource \oplus Exception$ to simply $Resource$ (which either was there in the first place, or was produced through the recovery process).

Target	Priority	Q	Result of INPUT_TAC
$X = \mathbf{A} \otimes (A \oplus B)$	$Left$	$\vdash A^{\perp}, \mathbf{Y}$	$\vdash X^{\perp}, \mathbf{Y} \otimes (A \oplus B)$
$X = A \otimes (\mathbf{A} \oplus B)$	$Right;\ Left$	$\vdash A^{\perp}, \mathbf{Y}$	$\vdash X^{\perp}, A \otimes (\mathbf{Y} \oplus B)$
$X = \mathbf{A} \oplus (B \otimes C)$	$Left$	$\vdash (B \oplus A)^{\perp}, \mathbf{Y}$	$\vdash X^{\perp}, \mathbf{Y} \oplus (B \otimes C)$
$X = A \oplus (\mathbf{B} \otimes C)$	$Right;\ Left$	$\vdash (B \oplus A)^{\perp}, \mathbf{Y}$	$\vdash X^{\perp}, A \oplus (\mathbf{Y} \otimes C)$

Table 4. Examples of how the `priority` parameter can affect the behaviour of INPUT_TAC. The selected subterms and the output of Q are highlighted in bold.

Case 2: If $Y = D \oplus E$ for some D and E such that $\vdash \Delta, C^{\perp}, D$ (or symmetrically $\vdash \Delta, C^{\perp}, E$) is a parallel buffer, then we can apply the following derivation:

$$
\cfrac{
\cfrac{\vdash \Delta, B^{\perp}, D \oplus E \quad \mathtt{Q} \quad
\cfrac{
\begin{array}{c} \mathtt{PARBUF\_TAC} \\ \vdots \\ \vdash \Delta, C^{\perp}, D \end{array}
}{\vdash \Delta, C^{\perp}, D \oplus E} \oplus L
}{\vdash \Delta, (B \oplus C)^{\perp}, D \oplus E} \&
}{} \tag{8}
$$

This may occur, for example, if $\Delta = \emptyset$ and $Y = C \oplus E$. The recovery process above may itself throw an exception: $\vdash Exception^{\perp}, Resource \oplus Failed$. This will convert output $Resource \oplus Exception$ to $Resource \oplus Failed$ (either we had the $Resource$ from the beginning, or we recovered and still got a $Resource$, or the recovery process failed) instead of $(Resource \oplus Failed) \oplus Resource$.

8.4 Putting It All Together

In the general case, the output X of P can be a complex combination of multiple parallel and optional outputs. For that reason, we apply the above proof strategies in a recursive, bottom-up way, prioritizing the user selections. We call the algorithm that produces the appropriate input $X^{\perp}$ (or equivalent) from Q "INPUT_TAC" and it has the following arguments (see Algorithm 1):

- **sel**: optional term corresponding to the user selected input $C^{\perp}$ of Q.
- **priority**: a list representing the path of the user selected subterm A in the syntax tree of the output X of P. For example, if the user selects B in the output $(A \otimes B) \oplus C$, the priority is $[Left;\ Right]$.
- **orient**: our latest path (left or right) in the syntax tree of X so that we add the corresponding buffers on the same side (see Section 8.2).
- **inputs**: a list of inputs of Q. We remove used inputs from this to avoid reuse.
- **target**: the input term we are trying to construct. This is initially set to X, but may take values that are subterms of X in recursive calls.
- **proc**: the CLL specification of Q as it evolves.

The `priority` parameter is useful when more than one subterms of the output either (a) are the same or (b) have the same matching input in Q. Table 4 shows examples of how different priorities change the result of INPUT_TAC.

Algorithm 1 Manipulates the specification "`proc`" to synthesize an input of type "`target`". Returns a new process specification that is logically derived from the original.

```
 1: function INPUT_TAC(sel, priority, orient, inputs, target, proc)
 2:    Try to match target with sel (if provided) or one of the inputs
 3:    if it matches then return proc

 4:    else if target is atomic then
 5:      if priority ≠ None then fail      ▷ we couldn't match the user selected output
 6:      else Create a target buffer using (5) depending on orient
 7:      end if

 8:    else if target is L ⊗ R then
 9:      if priority = Left then
10:        proc' = INPUT_TAC(sel, tail(priority), orient, inputs, L, proc)
11:        proc = INPUT_TAC(None, None, Right, inputs - {L}, R, proc')
12:      else
13:        proc' = INPUT_TAC(sel, tail(priority), orient, inputs, R, proc)
14:        proc = INPUT_TAC(None, None, Left, inputs - {R}, L, proc')
15:      end if
16:      Use the ⅋ rule to create the (L ⊗ R)⊥ input

17:    else if target is L ⊕ R then
18:      if priority = Left then
19:        proc = INPUT_TAC(sel, tail(priority), orient, inputs, L, proc)
20:        Try derivation (7) orElse Try derivation (8) orElse Use derivation (6)
21:      else if priority = Right then
22:        proc = INPUT_TAC(sel, tail(priority), orient, inputs, R, proc)
23:        Try derivation (7) orElse Try derivation (8) orElse Use derivation (6)
24:      else
25:        Try as if priority = Left orElse Try as if priority = Right
26:        else Create a target buffer using (5) depending on orient
27:      end if
28:    end if
29:    return proc
30: end function
```

9 Conclusion

CLL's inherent properties make it an ideal language to reason about resources. CLL sequents (under polarity restrictions) can be viewed as resource-based specifications of processes. The CLL inference rules then describe the logically legal, but primitive ways to manipulate and compose such processes.

We presented algorithms that allow intuitive composition in parallel, conditionally, and in sequence. We call these composition actions `TENSOR`, `WITH`, and `JOIN` respectively, and they are implemented in HOL Light. We analysed how each action functions in different cases and examples.

As a result of the rigorous usage of CLL inference rules, the constructed compositions have guaranteed resource accounting, so that no resources disappear or are created out of nowhere. The proofs-as-processes paradigm and its recent

evolutions allow the extraction of process calculus terms from these proofs, for concurrent and guaranteed deadlock-free execution.

In the future, we intend to work towards relaxing identified limitations along 2 main lines: (a) functionality, by incorporating and dealing with increasingly more complex specifications including those requiring formulation of more complex filters, and (b) expressiveness, by extending the fragment of CLL we are using while keeping a balance in terms of efficiency.

Through this work, it is made obvious that intuitive process compositions in CLL require complex applications of a large number of inference rules. Our algorithms automate the appropriate deductions and alleviate this burden from the user. We have tied these with the diagrammatic interface of `WorkflowFM` [21], so that the user is not required to know or understand CLL or theorem proving, but merely sees inputs and outputs represented graphically. They can then obtain intuitive process compositions with the aforementioned correctness guarantees with a few simple clicks.

Acknowledgements

References

1. Van der Aalst, W.M.: The application of petri nets to workflow management. Journal of circuits, systems, and computers 8(01), 21–66 (1998)
2. Abramsky, S.: Proofs as processes. Theoretical Computer Science 135(1), 5–9 (1994)
3. Acay, C., Pfenning, F.: Refinements for session typed concurrency (2016)
4. Bellin, G., Scott, P.: On the π-calculus and linear logic. TCS 135(1), 11–65 (1994)
5. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. LNCS pp. 222–236 (2010)
6. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL) 1.1 (2001)
7. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. Submitted to FoSSaCS (2018)
8. Dardha, O., Pérez, J.A.: Comparing deadlock-free session typed processes. In: Proceedings of the 22th International Workshop on Expressiveness in Concurrency EXPRESS/SOS. pp. 1–15 (2015)
9. Girard, J.Y.: Linear logic: its syntax and semantics. In: Girard, J.Y., Lafont, Y., Regnier, L. (eds.) Advances in Linear Logic. No. 222 in London Mathematical Society Lecture Notes, Cambridge Univ. Press (1995)
10. Gooch, P., Roudsari, A.: Computerization of workflows, guidelines and care pathways: a review of implementation challenges for process-oriented health information systems. Journal of the American Medical Informatics Association 18(6), 738–748 (2011)
11. Harrison, J.: HOL Light: A tutorial introduction. LNCS pp. 265–269 (1996)

12. Kezadri Hamiaz, M., Pantel, M., Thirioux, X., Combemale, B.: Correct-by-construction model driven engineering composition operators. Formal Aspects of Computing 28(3), 409–440 (May 2016)
13. Laurent, O.: Etude de la polarisation en logique. Ph.D. thesis, Université de la Méditerranée-Aix-Marseille II (2002)
14. Lincoln, P., Mitchell, J., Scedrov, A., Shankar, N.: Decision problems for propositional linear logic. Annals of Pure and Applied Logic 56(1), 239 – 311 (1992)
15. Manataki, A., Fleuriot, J., Papapanagiotou, P.: A workflow-driven formal methods approach to the generation of structured checklists for intrahospital patient transfers. Journal of Biomedical and Health Informatics 21(4), 1156–1162 (2017)
16. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., et al.: OWL-S: Semantic markup for web services (2004)
17. McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL–the planning domain definition language (1998)
18. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge Univ Presss (1999)
19. OASIS: Web Services Business Process Execution Language, version 2.0, OASIS Standard (2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`
20. Object Management Group: Business Process Model and Notation (BPMN), version 2.0 (2011), `http://www.omg.org/spec/BPMN/2.0/PDF`
21. Papapanagiotou, P., Fleuriot, J.: WorkflowFM: A logic-based framework for formal process specification and composition. In: de Moura, L. (ed.) Automated Deduction – CADE 26. pp. 357–370. Springer (2017)
22. Rao, J., Küngas, P., Matskin, M.: Composition of semantic web services using linear logic theorem proving. Information Systems 31(4-5), 340–360 (2006)
23. Szpyrka, M., Nalepa, G.J., Ligęza, A., Kluza, K.: Proposal of formal verification of selected BPMN models with Alvis Modeling Language. In: 5th International Symposium on Intelligent Distributed Computing. pp. 249–255. Springer (2012)
24. Tammet, T.: Proof strategies in linear logic. Journal of Automated Reasoning 12(3), 273–304 (1994)
25. Toninho, B., Caires, L., Pfenning, F.: Dependent session types via intuitionistic linear type theory. In: 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming. pp. 161–172. PPDP '11, ACM (2011)
26. Tounsi, I., Hadj Kacem, M., Hadj Kacem, A.: Building correct by construction SOA design patterns: Modeling and refinement. In: Drira, K. (ed.) Software Architecture. pp. 33–44. Springer (2013)
27. Troelstra, A.S.: Lectures on Linear Logic. CSLI Lecture Notes 29, Stanford (1992)
28. Wadler, P.: Propositions as sessions. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming. pp. 273–286. ACM (2012)