Compositional Separation of Control Flow and Data Flow

Damian Arellanes
Lancaster University, United Kingdom

Abstract

Every Model of High-Level Computation (MHC) has an underlying composition mechanism for combining simple computing devices into more complex ones. Composition can be done by (explicitly or implicitly) defining control flow, data flow or any combination thereof. Control flow specifies the order in which individual computations are activated, whereas data flow defines how data is exchanged among them. Unfortunately, traditional MHCs either mix data and control or only consider one dimension explicitly, which makes it difficult to reason about data flow and control flow separately. Reasoning about these dimensions orthogonally is a crucial desideratum for optimisation, maintainability and verification purposes. In this paper, we introduce a novel MHC that explicitly treats data flow and control flow as separate dimensions, while providing modularity. As the model is rooted in category theory, it provides category-theoretic operations for compositionally constructing sequential, parallel, branching or iterative composites. Compositionality entails that a composite exhibits the same properties as its respective constituents, including separation of concerns and modularity. We conclude the paper by demonstrating how our proposal can be used to model high-level computations in two different application domains: software engineering and artificial intelligence.

1. Introduction

In the context of theoretical computer science, *compositionality* refers to the property of Models of High-Level Computation (MHCs) that allows the inductive definition of complex computing devices from simpler ones [1, 2, 3, 4]. MHCs raise the level of abstraction of their classical, low-level counterpart (e.g., Turing Machines) by giving a birds-eye-view of multiple interacting devices which can individually correspond to low- or even high-level computations *per se*. As such devices are treated as black boxes, their internal details are irrelevant. What matters is how to compose/glue them into high-level abstractions, e.g., a sequential process to trigger the computation of devices A and B, in that order. Apart from moving up the ladder of abstraction, another characteristic defining difference with respect to their classical counterpart is that MHCs can be open in the sense they can compute on data streams coming from the external world, which make them suitable to be useful in the actual construction or simulation of complex computing systems [5]. Examples of MHCs include component models [6], workflow languages [7] and process algebras [8].

When composition is done algebraically, the resulting computation structures (known as *composites*) exhibit the same characteristics as their constituents [6]. Algebraic compositionality can be realised by the composition of *control flow* [9] or *data flow* [10]. Control flow defines the order in which individual computing devices are computed, whereas data flow defines how data is passed among them. Traditionally, MHCs do not support algebraic composition and they allow the definition of computations in which data follows control. This sort of coupling makes it difficult (i) to (formally) reason about computation order and data production/consumption separately and (ii) to explicitly distinguish between control and data dependencies [11]. Consequently, it is hard to (1) verify these dimensions independently [12, 13], or (2) modify/optimise/transform control flow without affecting data flow (or vice versa) [14, 15]. As separating control from data addresses (1) for increased reliability and (2) for enhanced maintenance, enabling such a separation within the foundational semantics of any MHC is a crucial

desideratum. Facilitating this in a compositional manner is far from trivial but it can provide additional benefits such as modularity for reusing high-level computations at scale (i.e., functional scalability [9]) and compositional verification towards soundness-by-construction [16]. For example, we can verify termination compositionally through the analysis of control flow only, i.e., without considering data flow at all.

While *control-based composition* approaches define explicit control flow for the coordination of computing devices, *data-based composition* defines implicit control in the collaborative exchange of data [17]. Thus, the notion of control flow has higher precedence than that of data flow because it is always present in any composition mechanism (and not the other way round).[1] In fact, it is possible to compose complex computations by control flow only and without the need of passing data at all (cf. actuator composition [18]).[2] More generally, in every model of computation, control flow is always present no matter if it is implicit or explicit. For instance, a finite state automaton can describe the finite control of a Turing machine when data is abstracted away. Here, control structures (e.g., sequencing, branching or looping) are not explicitly given but they rather emerge from state transitions [3].

As control flow is ever present, we believe that the right way of constructing complex, high-level computational behaviours is through a control-based composition mechanism that does not neglect the role of data passing. Accordingly, in this paper we propose a model in which fundamental units of composition (known as *computons*) are passive open systems able to interact with their environment via an interface which consists of input and output ports. As a port is a structural construct that can exclusively buffer either data or control, computons exchange data and control separately. Our model is compositional in the sense computons can be inductively composed into larger ones via well-defined control-based composition operations. Such operations are rooted in category theory and allow the formal construction of sequential, parallel, branching or iterative computons from simpler ones. As a result of composition semantics, composites preserve the structure of the composed entities, so they also separate data and control and their port-based interface is inductively constructed from the composed computons. Remarkably, unlike existing compositional approaches, any two computons can always be composed sequentially or in parallel, regardless of the data they require or produce. Rather than focusing on specific operational semantics, the focus of this paper is to provide formal category-theoretic operators, built upon colimit semantics, for the compositional construction of computing devices (i.e., computons) that separate data flow and control flow. Although different ways of expressing computon behaviour are possible, because the model is independent of concrete execution semantics, in this paper we use the *token game* from the theory of classical P/T Petri nets for this purpose, as shown in Figure 1.

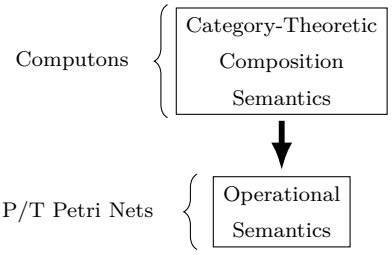


Figure 1: This paper is primarily focused on (category-theoretic) composition semantics to study control-based composition in its pure form, without adhering to specific execution details. Separating composition from operation allows us to describe computon behaviour through different execution models. Although here we use the well-known token game from classical P/T Petri nets, other formalisms can be used for the same purpose such as natural semantics [20], small-step semantics [21] or even timed Petri nets [22], just to name a few.

[1] A potential explanation is that the notion of control flow is tightly linked to the arrow of time, whereas data is just a piece of information that is sent from one place to another in some specific time-dependent order.

[2] Even if we argue that control is a piece of information/data, the act of sending it from one computing device to another is still governed by the grandiose, apparently unavoidable, notion of control flow (cf. interleaving semantics for global execution traces in concurrent systems [19]).

By focusing on the fundamentals of control-based composition while capturing the core mechanisms that underlie all MHCs (i.e., control flow and data flow), the model we propose is intended to be a universal meta-framework for the formal reasoning of every possible high-level computation. This generality, coupled with the fact that we do not subscribe to concrete execution semantics or implementation details, allows the model's foundations to serve as the basis for the future development of concrete specification languages, implementation frameworks or even standard tools that can be used by software engineers to compositionally construct complex software systems that exhibit separation of control and data flow. For example, the proposed model can be implemented via the Alloy specification language [23] so as to support model checking. If less assurances are preferred or visual modelling is required, mainstream languages can be used instead such as Java or JavaScript.

Offering generality also enables the combination of distinct models of computation. For example, the behaviour of a (non-composite) computon can be expressed as a Turing machine, whereas the behaviour of another (non-composite) can be defined as a (typed) lambda abstraction. Then, using the proposed sequencing operator, we can form a composite computon for executing the Turing machine before supplying its output to the lambda abstraction in order to perform a reduction operation.[3] If a parallelising operator is used instead, then these two computing devices can be triggered at the same time.

Apart from enabling generality, orthogonalising data and control and separating operational from composition semantics, our model facilitates modularity and encapsulation as a result of realising compositionality. Encapsulation allows treating computons unifiedly, and is realised by the fact that a composite defines explicit control and data flow structures that can only be accessed through a well-defined interface. Thus, a computon can be perceived as an encapsulated black-box that can only interact with other computons via its visible ports. Hiding internal structure in this way enable us to build computons of considerable complexity.

As control is what governs computation, one of the primary objectives of the computon model is to facilitate the compositional formation of explicit control flow structures. As such structures are orthogonal to data, data flow can be "disregarded" to independently reasoning about execution order, i.e., unlike models where control is implicit, execution order does not need to be "discovered". In Section 2, we formalise functors for this purpose, which can be useful for static analysis or verification purposes.

The rest of the paper is structured as follows. Section 2 presents the definition of computons by treating them as set-valued functors in a category we introduce which we refer to as the *category of computons and computon morphisms*. Although this paper is mainly focused on the compositional construction of such entities and not on their operational semantics, Section 3 discusses the notion of computon execution via the well-known *token game* that has been used in the context of classical P/T Petri nets for many years. Sections 4 and 5 present the most elementary classes of computons that serve as building blocks for constructing complex composites. Section 6 describes formal category-theoretic operations to form sequential, parallel, branching or iterative composites. Section 7 provides two applications of the proposed model in the domains of software engineering and artificial intelligence, by discussing compositional system construction and showing how the separation of control and data flow can be exploited for model transformation purposes. Section 8 presents and analyses related work, and Section 9 outlines the conclusions and future directions of our work. A mapping from Petri net syntax to the graphical notation we use to represent computons is provided in Appendix A.

[3] Of course, in practice, the Turing machine's output might need to be encoded to make it compatible with the input expected by the lambda abstraction, either by a run-time environment or by introducing an intermediate computon for doing the encoding.

2. Computons

Intuitively, a *computon* is a bipartite graph with two types of nodes: *computation units* and *ports*.[4] A computation unit is a construct that receives information in ports, performs some computation and produces new information in other ports. A port that is connected between two computation units is known as *internal port* (or *i-port*), whereas a port that is exclusively connected to or from a computation unit is called *external* (or *e-port*). As ports and computation units are connected via edges, edges represent information flow ranging from control signals to complex data values.[5]

The interface of a computon towards the outside world is determined by its collection of e-ports each being an external control inport (*ec-inport*), an external control outport (*ec-outport*), an external data inport (*ed-inport*) or an external data outport (*ed-outport*). An ec-inport is where control flow originates, an ec-outport is where control flow terminates, an ed-inport stores data coming from the external world whereas an ed-outport stores data resulting from the computon's operation. A port that is ec-inport and ec-outport is called *ec-inoutport*. Similarly, if it is both ed-inport and ed-outport, it is called *ed-inoutport*.

Figure 2 presents the naming system we employ to derive port names where it is clear that, just as there are *ec-ports* and *ed-ports*, there are also internal control ports (*ic-ports*) and internal data ports (*id-ports*).

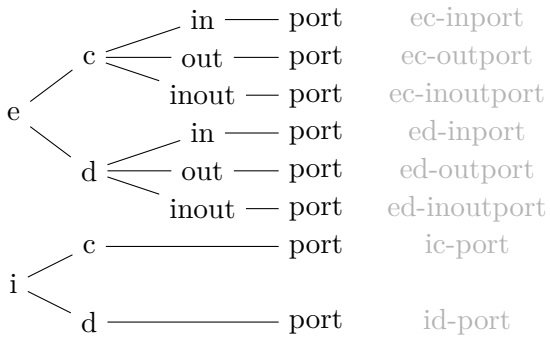


Figure 2: Taxonomy of port types. The letter *e* stands for external, *i* for internal, *c* for control and *d* for data. External ports serve as inputs, outputs or both for facilitating interaction with the external world. Internal ports lack this characteristic as they solely mediate communication within the internals of a computon. For the sake of conciseness, we write dashes only after *c* or *d*, rather than writing them for each level of the hierarchy. For example, the path *e-c-out-port* is abbreviated as *ec-outport*. In total, there are eight possible paths, i.e., eight possible port types.

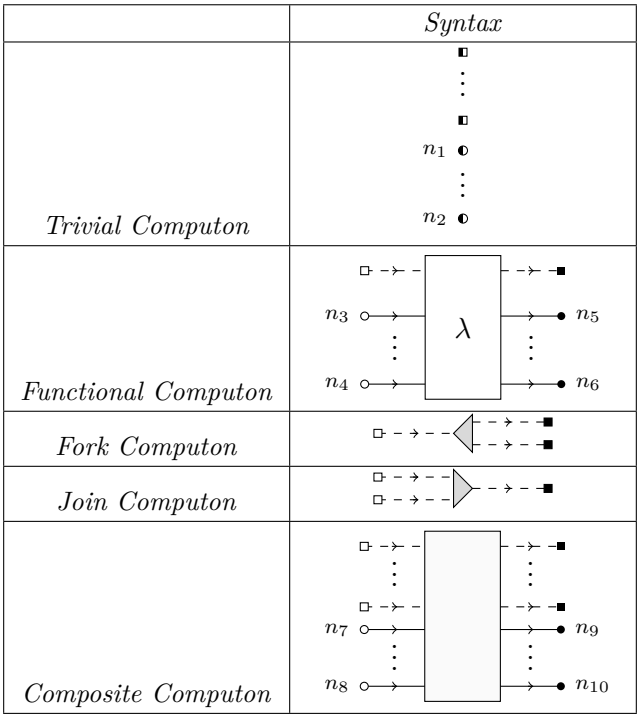
The dichotomy between data and control in our proposal entails that a computon is a unit of control-driven computation wherein control signals and data values travel independently via edges. Ports are differentiated by colours which, in practice, can be abstract data types such as the type of booleans or the type of integers. In this paper, in order to provide a general "type distinction framework", ports are deliberatively coloured with natural numbers. Although non-concrete sets could be used in principle, $\mathbb{N}$ is particularly convenient because it simplifies the presentation and interpretation of our approach by offering familiar, ready-to-use elements (i.e., colours) that can later be associated with concrete types. For example, in one type system, 2 may represent the type of booleans, 3 the type of floats and 4 the type of strings; whereas in another system, such numbers may denote characters, strings and integers, correspondingly. Such a description is straightforward precisely because we work with $\mathbb{N}$; using a fully abstract set would make this kind of explanations considerably less intuitive. In any case, we require a designated colour for control ports so we use 0 for that purpose. Any other natural number is used for data ports, as shown in Figure 3(a).

[4]The word *computon* derives from the Latin root for computation (i.e., *computatio*) and the Greek suffix *-on*. In Physics, such a suffix is traditionally used to designate subatomic particle names [24].

[5]In the context of resource theories [25], data ports correspond to resource wires.

		Syntax	*Colour Set*	*Incoming Edges*	*Outgoing Edges*
Control	*ec-inport*	▫	$\{0\}$	Never	Always
	ec-outport	▪	$\{0\}$	Always	Never
	ec-inoutport	◧	$\{0\}$	Never	Never
	ic-port	◧	$\{0\}$	Always	Always
	control flow	- -➛ -			
Data	*ed-inport*	○	$\mathbb{N}^+$	Never	Always
	ed-outport	●	$\mathbb{N}^+$	Always	Never
	ed-inoutport	◐	$\mathbb{N}^+$	Never	Never
	id-port	◐	$\mathbb{N}^+$	Always	Always
	data flow	⟶			

(a) Syntax for ports and flows.



(b) Syntax for computons. Here, $n_1, \ldots, n_{10} \in \mathbb{N}^+$ and the λ-box and triangles denote computation units. The other box corresponds to a composite computon whose internals depend on the composition operator being used. More details on such operators are provided in Section 6.

Figure 3: Graphical syntax of the computon model.

A glance at Figure 3(a) reveals that control ports are associated with square shapes, whereas data ports are displayed as circles. As control ports are always zero-labelled, we will omit their colour for clarity purposes, and we just display the colour of data ports. Figure 3(a) shows that e-inports and e-outports are depicted on white and black backgrounds, respectively. As e-inoutports are a combination of e-inports and e-outports, their background is half black and half white. It is important to mention that e-inports do not have any incoming edges but just outgoing ones, while e-outports only have incoming edges. Ec-inoutports have no edges at all and ic-ports have adjacent edges on both ends. So, even if they share the same graphical representation, ic-ports and ec-inoutports can be distinguished by their connected edges (the same is true for id-ports and ed-inoutports). We decide to use the same syntax for them because, intuitively, both i-ports and e-inoutports receive and forward information.

Figure 3(b) displays the rest of the syntax we will be using throughout this paper to

discuss the computon model. We acknowledge that the semantics of these diagrams is not obvious at this stage so we refer the reader to Sections 4, 5 and 6 for further details on this matter. The use of boxed diagrams with port-based interfaces is getting increasingly popular in the literature on compositionality. Like existing notations, our graphical syntax maintains a clear distinction between computation units and ports, and differentiates between inputs (i.e., e-inports), outputs (i.e., e-outports) and input-outputs (i.e., i-ports). The difference lies in the support to distinguishing among different types of (high-level) computing devices which we refer to as computons (viz., trivial, functional, fork, join and composites). Distinguishing between them is of vital importance for immediately recognising individual computational behaviour while providing a clear visualisation of the structural parts of a composite. Structural clarity would be less evident if all computon types used identical syntax.

Another key difference with respect to existing notations is that our syntax offers a clear syntactic separation between control and data ports and between control and data flows; thus, emphasising the explicit separation of control flow and data flow provided by our model. Furthermore, we introduce syntactic constructs for expressing input-output ports that are never connected to computation units, which we refer to as e-inoutports.

A glance at Figure 3(b) reveals that a collection of e-inoutports gives rise to what we call trivial computons (see Section 4). This figure also shows that functional computons are able to receive data and exactly one control signal before producing further data and a new control signal (see Section 5). Figure 3(b) also shows that fork and join computons do not require or produce any data, but just control (see Section 5). All these properties can be immediately devised by just looking at our graphical syntax, without the need of delving into formal definitions. Trivial, functional, fork and join computons can be used to form even more complex entities which we refer to as composite computons (see Section 6).

Given the above reasons, we believe that our graphical syntax is more suitable than existing ones for discussing compositional construction over a wide range of diverse computons that explicitly separate control flow and data flow. Our syntax is indeed ready to be used by a visual programming language built on top of the computon model theory, which we intend to develop in the near future.

In Section 5, we will see that functional, fork and join computons belong to the class of *primitive computons*. We say they are primitive because each of them captures a (potentially low-level) computation in a single unit that can be composed into high-level computations (i.e., composite computons) via the operators we propose in Section 6. Although different operational rules can be given to computation units due to the separation of execution from composition semantics, the expectation is that a unit must encapsulate a halting computation that can only be triggered when all the unit's e-inports have information. As a unit always has at least one ec-inport and ed-inports are optional, execution is necessarily driven by control. Upon termination, a unit must produce information in all its e-outports (which always include ec-outports), no matter whether computation is functional or relational. This operation strictly guarantees that composites do not diverge and that the intended semantics of sequencing, parallelising, branching and iteration are consistently preserved. Although we adhere to this strict operation scheme in Section 3, the computon model is flexible enough to accommodate other execution semantics for additional features such as time, costs, weights, pre-/post-conditions, probabilistic randomness or any combination thereof.

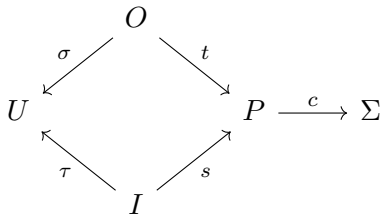
Regardless of the chosen operation scheme, the internal structure of a computation unit must never be made visible from a high-level perspective, in order to support modularity of primitive computons and, by extension, of the composite computons built upon them. Composition does not depend on such internal details, but only on defined interfaces. Computon interfaces use natural numbers as an abstraction of data types to specify a contract that tells what inputs to supply and what outputs to expect from computation, without exposing internal workings such as internal control or data routing.

The way control and data are consumed within a computation unit depends on the concrete interpretation given to computation units themselves. For example, if a unit is a Turing

machine for realising an n-ary computable function, the unit shall have at least one ec-inport (for receiving control signals) and exactly n ed-inports (one for each argument). As previously stated, even if all the ed-inports have data, it is expected that the unit will not be triggered until receiving a control signal in each ec-inport. Once all e-inports have information, the corresponding Turing machine can be executed on the n data inputs so input control signals can be safely disregarded.[6] Upon termination, a control signal can be assigned to each unit's ec-outport to indicate that the machine's computation has terminated and that output data (stored in ed-outports) is ready to be consumed by other units.[7] Consumption order is dictated by control flow within composite computons.

2.1. Formal Definition

Formally, a computon is a functor from **Comp** to **Set** (see Definition 1) where **Set** is the category of finite sets and total functions and **Comp** is the free category generated by the following diagram:[8] [9]



which consists of five objects and twelve morphisms (including identity and composite morphisms given by trivial paths and path concatenation, respectively).

Definition 1 (Computon)**.** A computon λ is a functor **Comp** $\to$ **Set** that maps:

- U to a (possibly empty) set $\lambda(U)$ of computation units,
- P to a (non-empty) set $\lambda(P)$ of ports,
- O to a (possibly empty) set $\lambda(O)$ of edges,
- I to a (possibly empty) set $\lambda(I)$ of edges,
- Σ to a (non-empty) set $\lambda(\Sigma) \subset \mathbb{N}$ of colours where $0 \in \lambda(\Sigma)$,
- σ to a surjective function $\lambda(\sigma) : \lambda(O) \twoheadrightarrow \lambda(U)$ that specifies the outgoing edges of each computation unit,
- τ to a surjective function $\lambda(\tau) : \lambda(I) \twoheadrightarrow \lambda(U)$ that specifies the incoming edges of each computation unit,
- t to a function $\lambda(t) : \lambda(O) \to \lambda(P)$ that specifies the incoming edges of each port,
- s to a function $\lambda(s) : \lambda(I) \to \lambda(P)$ that specifies the outgoing edges of each port, and
- c to a surjective function $\lambda(c) : \lambda(P) \twoheadrightarrow \lambda(\Sigma)$ that assigns to each port a colour

such that $\sigma \restriction_{(c \circ t)^{-1}(0)}$ is surjective, $\tau \restriction_{(c \circ s)^{-1}(0)}$ is surjective and there is:

[6] Recall that any Turing machine can use special delimiters in the tape to process multiple inputs.

[7] A Turing machine's output can be read from the tape upon termination.

[8] Following the notation of function abstraction in Lambda Calculus, we use λ to denote computons.

[9] Lifting conditions can be established to enforce (non-)emptiness, injectivity, surjectivity and even uniqueness, among others. Although such constraints can be straightforwardly applied on **Comp**, we do not present them for the clarity of argument. We simply assume that computons are well-defined if and only if the conditions imposed by Definition 1 are met (e.g., the surjectivity of the colouring function). A similar approach has been used in the context of databases for constraining database schemas [26].

- an identity function $1_{\lambda(x)}$ in **Set** for each object x of **Comp**,
- a composite function $\lambda(g) \circ \lambda(f)$ in **Set** for each pair (f, g) of composable morphisms in **Comp**,
- at least one port $p \in [\lambda(P) \setminus Im(\lambda(s))]$ with $\lambda(c)(p) = 0$ and
- at least one port $q \in [\lambda(P) \setminus Im(\lambda(t))]$ with $\lambda(c)(q) = 0$.

As a computon λ is a set-valued functor, it can be expressed in the form of a tuple $(U, P, I, O, \Sigma, \sigma, \tau, t, s, c)$. Without loss of generality, we took the liberty of simplifying the expression in order to reduce clutter, e.g., we write U for $\lambda(U)$. For the rest of the paper, the reader must bear in mind that each component of λ is an actual set or a function, not an object or a morphism in **Comp**. To distinguish between computons, we use natural numbers as subscripts which carry over computon components. If the symbol for a computon has no subscript, the computon components have no subscript either.

In Definition 1, we write $\sigma \restriction_{(c \circ t)^{-1}(0)}$ and $\tau \restriction_{(c \circ s)^{-1}(0)}$ to express the restriction of the functions σ and τ to the fibers $(c \circ t)^{-1}(0)$ and $(c \circ s)^{-1}(0)$, respectively. The surjectivity condition on these two functions ensures that every computation unit (if any) has at least one incoming edge and at least one outgoing edge connected from and to a 0-coloured port, respectively. As every function we deal with is total and σ and τ are surjective in general, every edge always runs from a port to a computation unit or viceversa. That is, a computon has neither dangling edges nor dangling computation units. When the set of units is empty, a computon is necessarily made up of coloured ports only. In Section 4, we will see that such a class of entities, referred to as trivial computons, is needed for the coherence of our theory.

Definition 2 (Computon Interface)**.** The interface of a computon λ towards the external world is a tuple (P^+, P^-) where $P^+ := P \setminus Im(t)$ is the set of e-inports of λ and $P^- := P \setminus Im(s)$ is the set of e-outports of λ. A port $p \in Im(s) \cap Im(t)$ is called an i-port of λ.

Notation 1. Given a computon λ, C^+ denotes its set of ec-inports, C^- its set of ec-outports, D^+ its set of ed-inports and D^- its set of ed-outports. These sets are defined as follows:

$$C^\square := \{p \in P^\square \mid c(p) = 0\} \text{ with } \square \in \{+, -\}$$

$$D^\square := \{p \in P^\square \mid c(p) > 0\} \text{ with } \square \in \{+, -\}$$

As the last two conditions of Definition 1 state that computons must have at least one ec-inport and at least one ec-outport, it trivially follows that $C^+ \neq \emptyset \neq C^-$. Data ports are optional so D^+ and D^- can be empty. Fork computons are an example where $D^+ = \emptyset = D^-$. Particularly, Figure 3(b) shows that they always possess only one element in P^+ and exactly two elements in P^-. It also shows that this sort of computons have no i-ports either, i.e., $Im(s) \cap Im(t) = \emptyset$. For more technical details on fork computons, see Section 5.

Notice in Definition 2 that the sets P^+ and P^- are not necessarily disjoint so a port can be e-inport and e-outport at the same time. If $p \in C^+ \cap C^-$, then p is an *ec-inoutport*. If $p \in D^+ \cap D^-$, it is an *ed-inoutport*. Figure 3(b) shows that trivial computons have all e-inoutports, i.e., $P = P^+ \cap P^-$. For more technical details on them, see Section 4.

In Figure 3(a), it is indicated that e-inports and e-outports only possess outgoing and incoming edges, respectively. This property follows from Proposition 1. Using Definition 2, it is easy to additionally show that e-inoutports have no edges at all and that i-ports have both incoming and outgoing edges. Thus, even if ic-ports and ec-inoutports share the same graphical representation, they can be distinguished by their connected edges, with the same being true for id-ports with respect to id-inoutports.

Proposition 1. If λ is a computon, then $P^+ \setminus P^- = P^+ \cap Im(s)$ and $P^- \setminus P^+ = P^- \cap Im(t)$.

Proof. Let $p \in P^+ \setminus P^-$ be a port of a computon λ. Then, $p \in P^+ \setminus P^- \iff p \in P^+ \wedge p \notin P^- \iff p \in P^+ \wedge \neg(p \in P \wedge p \notin Im(s))$ by Definition 2 $\iff p \in P^+ \wedge (p \notin P \vee p \in Im(s)) \iff p \in P^+ \wedge p \in Im(s)$ because $p \in P$ is always true $\iff p \in P^+ \cap Im(s)$. The proof of $P^- \setminus P^+ = P^- \cap Im(t)$ is completely analogous. □

As control ports and data ports are identified as separate entities, information movement within a computon corresponds to either data flow or control flow. Particularly, we say that any control port is connected to or from a computation unit via a control flow edge, whereas a data port is connected analogously but with a data flow edge (see Definitions 3 and 4). The collection of ports receiving information from a computation unit u and sending information to u are denoted $u\bullet$ and $\bullet u$, respectively. Similarly, $\bullet p$ and $p\bullet$ denote the source and target computation units of a port p, respectively (see Definition 5). When there is information flow from every e-inport or i-port to some e-outport, we say that the computon is connected (see Definition 6).

Definition 3 (Information Flow)**.** Given a computon λ, let $p \in P$ and $u \in U$. We say there is information flow from p to u if there is an edge $i \in I$ such that $s(i) = p$ and $\tau(i) = u$. This is denoted $p \xrightarrow{i} u$. If there is an edge $o \in O$ with $\sigma(o) = u$ and $t(o) = p$, we say there is information flow from u to p, written $u \xrightarrow{o} p$. We use $p_1 \xrightarrow{\exists} p_n$ to denote the existence of $p_1 \xrightarrow{i_1} u_1 \xrightarrow{o_1} p_2 \xrightarrow{i_2} u_2 \xrightarrow{o_2} \cdots \xrightarrow{i_{n-1}} u_{n-1} \xrightarrow{o_{n-1}} p_n$ for $p_1, \ldots, p_n \in P$, $u_1, \ldots, u_{n-1} \in U$, $o_1, \ldots, o_{n-1} \in O$, $i_1, \ldots, i_{n-1} \in I$ and $n \geq 2$.

Definition 4 (Control Flow and Data Flow Edges)**.** Given a computon λ and an edge $e \in I \cup O$, we say e represents control flow if $c(s(e)) = 0$ or $c(t(e)) = 0$; otherwise, it represents data flow.

Definition 5 (Pre- and Post-Sets)**.** For a computation unit $u \in U$ of a computon λ, $\bullet u$ and $u\bullet$ denote the sets $\{p \in P \mid (\exists i \in I)(p \xrightarrow{i} u)\}$ and $\{p \in P \mid (\exists o \in O)(u \xrightarrow{o} p)\}$, respectively. Similarly, for a port $p \in P$, $\bullet p$ and $p\bullet$ denote the sets $\{u \in U \mid (\exists o \in O)(u \xrightarrow{o} p)\}$ and $\{u \in U \mid (\exists i \in I)(p \xrightarrow{i} u)\}$, respectively.

Definition 6 (Connected Computon)**.** We say that a computon λ is connected if and only if for each $p \in Im(s) \cup P^+$ there exists some $q \in P^-$ such that $p \xrightarrow{\exists} q$ holds.

A glance at Figure 3(b) reveals that functional computons satisfy Definition 6, whereas trivial ones do not. In general, the existence of at least one e-inoutport implies that the corresponding computon is not connected, as captured by the contrapositive of Proposition 2. When a computon is connected, there always are computation units and there is information flow from every e-outport or i-port to some e-inport (see Propositions 3 and 4).

Proposition 2. If λ is a connected computon, $P^+ \cap P^- = \emptyset$.

Proof. Assume for contradiction that λ is connected with some $p \in P^+ \cap P^-$ so that $p \notin Im(t)$ and $p \notin Im(s)$. Using Definition 6, we know there must be some $q \in P^-$ where $p \xrightarrow{\exists} q$, i.e., $s(p) = i$ for some $i \in I$ as per Definition 3. As this clearly contradicts $p \notin Im(s)$, we conclude $P^+ \cap P^- = \emptyset$. □

Proposition 3. Every connected computon has at least one computation unit.

Proof. Assume for contradiction λ is a connected computon with $U = \emptyset$, meaning σ and τ are well-defined only if $I = \emptyset = O$. Since λ is connected, for each $p \in Im(s) \cup P^+$ there is some $q \in P^-$ for which $p \xrightarrow{\exists} q$ holds.

- If $p \in Im(s)$, there must be some $i \in I$ where $s(i) = p$; thereby, contradicting $I = \emptyset$.
- If $p \in P^+$, we have two possibilities:

– There is some $i \in I$ where $s(i) = p$ which also contradicts $I = \emptyset$.
– There is no $i \in I$ where $s(i) = p$ so there is no information flow from p to any other port, including e-outports; thus, contradicting the fact that λ is connected.

Therefore, we conclude $U \neq \emptyset$. □

Proposition 4. If λ is a connected computon, for each $q \in P^- \cup Im(t)$ there is a port $p \in P^+$ where $p \xrightarrow{\exists} q$.

Proof. If λ is a connected computon with $q \in P^- \cup Im(t)$, we know by Proposition 2 that $q \in P^- \cap P^+$ cannot hold. So, if $q \in P^-$, then $q \notin P^+$; in other words, $q \notin Im(s)$ and $q \in Im(t)$. So, it suffices to show $q \in Im(t)$ only. For this, we perform the following recursive procedure by initially considering the set $V := \{q\}$ of visited ports:

As $q \in Im(t)$, there is some $o \in O$ where $t(o) = q$. Applying the surjectivity of σ and τ and the totality of s, we derive $p \xrightarrow{i} u \xrightarrow{o} q$ for some $u \in U$, some $i \in I$ and some $p \in P \setminus V$. If $p \in P^+$, then the proof is complete. Otherwise, $p \in Im(t)$ so we simply repeat this procedure with $V \cup \{p\}$ in place of V and p in place of q.

The above procedure will eventually terminate since the sets U, I, O and P are finite, and because we use $V \subseteq P$ to explore new ports at each step. It is guaranteed that an e-inport will be chosen for exploration at some point since every computon has at least one ec-inport and there are no dangling computation units (see Definition 1). □

At this stage, we have provided sufficient details about the general structure of computons by treating them as set-valued functors. Defining computons in this way gives rise to a functor category which we refer to as $\mathbf{Set}^{\mathbf{Comp}}$.

2.2. The Category of Computons

$\mathbf{Set}^{\mathbf{Comp}}$ is a category whose objects and morphisms are computons and computon morphisms, respectively (see Definition 7).

Definition 7 (Computon morphism)**.** If λ_1 and λ_2 are two computons, a computon morphism $\alpha \colon \lambda_1 \to \lambda_2$ is a natural transformation whose components are the total functions $\alpha_U \colon U_1 \to U_2$, $\alpha_P \colon P_1 \to P_2$, $\alpha_O \colon O_1 \to O_2$, $\alpha_I \colon I_1 \to I_2$ and $\alpha_\Sigma \colon \Sigma_1 \hookrightarrow \Sigma_2$ such that the diagrams of Figure 4 commute and $\vec{i}(\alpha) \cup \vec{o}(\alpha) \subseteq P_1^+ \cup P_1^-$. Here, $\vec{i}(\alpha)$ and $\vec{o}(\alpha)$ denote $\{p_1 \in P_1 \mid \bullet\alpha(p_1) \setminus \alpha(\bullet p_1) \neq \emptyset\}$ and $\{p_1 \in P_1 \mid \alpha(p_1) \bullet \setminus \alpha(p_1\bullet) \neq \emptyset\}$, respectively.

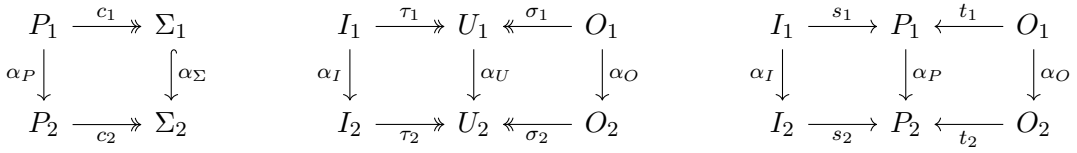


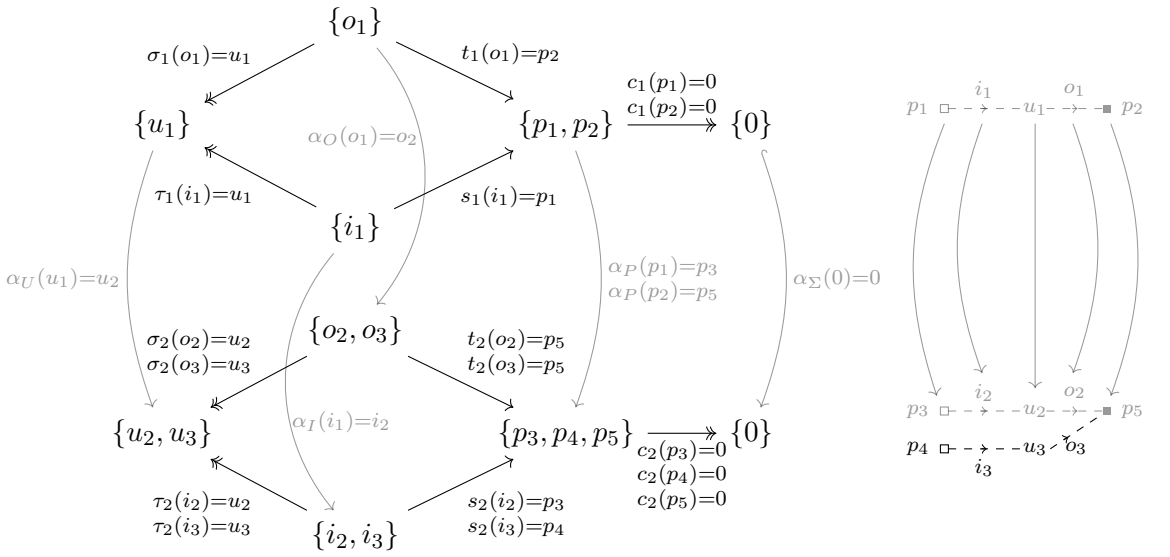
Figure 4: A computon morphism is a natural transformation $\alpha \colon \lambda_1 \to \lambda_2$.

Notation 2. To simplify notation when referring to the components of a computon morphism $\alpha \colon \lambda_1 \to \lambda_2$, we write $\alpha(u)$ for $\alpha_U(u)$, $\alpha(p)$ for $\alpha_P(p)$, $\alpha(o)$ for $\alpha_O(o)$ and $\alpha(i)$ for $\alpha_I(i)$. For the rest of the paper, we also write $\alpha(A)$ to denote $Im(\alpha_P|_A)$ if $A \subseteq P_1$ or $Im(\alpha_U|_A)$ if $A \subseteq U_1$. Likewise, we use $\alpha(B)^{-1}$ to denote $\{p_1 \in P_1 \mid \alpha(p_1) \in B\}$ if $B \subseteq P_2$ or $\{u_1 \in U_1 \mid \alpha(u_1) \in B\}$ if $B \subseteq U_2$.

Remark 1. Naturally, composition of computon morphisms α and β is defined componentwise:

$$(\beta_U, \beta_P, \beta_I, \beta_O, \beta_\Sigma) \circ (\alpha_U, \alpha_P, \alpha_I, \alpha_O, \alpha_\Sigma) = (\beta_U \circ \alpha_U, \beta_P \circ \alpha_P, \beta_I \circ \alpha_I, \beta_O \circ \alpha_O, \beta_\Sigma \circ \alpha_\Sigma)$$

Figure 5(a) describes a computon morphism α from λ_1 to λ_2. The top-level diamond specifies that λ_1 has ports $p_1 \in P_1$ and $p_2 \in P_1$ connected to and from a computation unit $u_1 \in U_1$ through the edges $i_1 \in I_1$ and $o_1 \in O_1$, respectively. Thereby, forming the information flows $p_1 \xrightarrow{i_1} u_1 \xrightarrow{o_1} p_2$. As p_1 has no incoming edges and p_2 has no outgoing edges, we use Definition 2 to deduce $P_1^+ := \{p_1\}$ and $P_1^- := \{p_2\}$. Since both p_1 and p_2 are zero-coloured and they are the only ports in λ_1, we further deduce $P_1^+ = C_1^+$, $P_1^- = C_1^-$ and $D_1^+ = \emptyset = D_1^-$. Therefore, by Definition 4, both i_1 and o_1 denote control flow. The top-level diagram of Figure 5(b) shows the graphical representation of λ_1 using computon syntax.[10]



(a) Using commutative diagrams.

(b) Using computon syntax:
▫ Ec-inport ▪ Ec-outport
- ➛ - Control flow edge

Figure 5: Example of a computon morphism $\alpha\colon \lambda_1 \to \lambda_2$.

The diamond at the bottom of Figure 5(a) specifies that λ_2 has ports $p_3 \in P_2$ and $p_5 \in P_2$ connected to and from a computation unit $u_2 \in U_2$ via the edges $i_2 \in I_2$ and $o_2 \in O_2$, respectively. This computon also includes the edge $i_3 \in I_2$ for connecting the port $p_4 \in P_2$ to a computation unit $u_3 \in U_2$ which, in turn, is connected to p_5 via the edge $o_3 \in O_2$. Thereby, forming the information flow $p_3 \xrightarrow{i_2} u_2 \xrightarrow{o_2} p_5 \xleftarrow{o_3} u_3 \xleftarrow{i_3} p_4$. Similar to λ_1, we observe that p_3 and p_4 have no incoming edges and that p_5 has no outgoing edges. So, by Definition 2, $P_2^+ := \{p_3, p_4\}$ and $P_2^- := \{p_5\}$. As all ports of λ_2 are also zero-coloured, it follows that $P_2^+ = C_2^+$, $P_2^- = C_2^-$ and $D_2^+ = \emptyset = D_2^-$. By Definition 4, this means all the edges of λ_2 represent control flow. The bottom-level diagram of Figure 5(b) displays the graphical structure of λ_2 through the use of computon syntax.

The components of the computon morphism $\alpha\colon \lambda_1 \to \lambda_2$ are displayed in gray on Figure 5(a), in order to distinguish them from the diamond diagrams that define computons. Figure 5(b) shows that this morphism maps the unique computation unit of λ_1 to u_2, the sole ec-inport of λ_1 to the ec-inport p_3, the unique ec-outport of λ_1 to the unique ec-outport of λ_2, the edge i_1 to the edge i_2 and the edge o_1 to the edge o_2. This mapping is sound since it ensures the diagrams of Figure 5(a) commute, meaning that the structure of λ_1 is preserved within λ_2.

Observe in λ_2 that u_2 and u_3 are the only computation units connected to p_5 so that

[10] Recall that ports are coloured with natural numbers and edges are not coloured at all. The diagram on the right-hand side of Figure 5 just displays port and flow labels for illustrative purposes. For now, we just display computation units as labels but, in upcoming sections, we will use specific syntax to distinguish among different types of such units.

$\bullet p_5 = \{u_2, u_3\}$ by Definition 5. Considering $\alpha_P(p_2) = p_5$, as shown in Figure 5(a), we have $\bullet p_5 = \{u_2, u_3\} = \bullet\alpha_P(p_2)$. A further inspection of Figure 5(a) reveals that the set of all computation units mapped from u_1 is $\{u_2\}$ because $\alpha_U(u_1) = u_2$ is the only mapping given by α_U, i.e., $\alpha_U(\{u_1\}) = Im(\alpha_U\,|_{\{u_1\}}) = \{u_2\}$, according to Notation 2. Definition 5 allows us to deduce $\{u_1\} = \bullet p_2$ because u_1 is the only computation unit connected to p_2. So, $\alpha_U(\{u_1\}) = Im(\alpha_U\,|_{\{u_1\}}) = \{u_2\} = \alpha_U(\bullet p_2)$. As $\bullet\alpha_P(p_2) \setminus \alpha_U(\bullet p_2) = \{u_2, u_3\} \setminus \{u_2\} \neq \emptyset$, we use Definition 7 to conclude $p_2 \in \vec{i}(\alpha)$. Basically, $\vec{i}(\alpha)$ denotes the set ports in λ_1 that are mapped to ports in λ_2 connected to computation units not included in the α-embedding (in this case, u_3). The set $\vec{o}(\alpha)$ is similar but contains λ_1-ports that are mapped to λ_2-ports connected to computation units excluded from the α-embedding. As we did to verify $p_2 \in \vec{i}(\alpha)$, we can show $\vec{o}(\alpha) = \emptyset$ in our example. This is because the information flow $p_1 \xrightarrow{i_1} u_1 \xrightarrow{o_1} p_2$ of λ_1 is entirely inserted into the information flow $p_3 \xrightarrow{i_2} u_2 \xrightarrow{o_2} p_5$ of λ_2, and there are no flows of the form $p_3 \to u$ or $p_5 \to u$ in λ_2.

The example we just described allows us to intuitively perceive a computon morphism as an embedding (or an insertion) of a computon into a (potentially more complex) one, while preserving ports (with their respective colours, incoming edges and outgoing edges) and computation units (with their respective incoming and outgoing edges). As a result of this preservation, an e-inport is mapped to an e-inport or an i-port (e.g., $\alpha_P(p_1) = p_3 \in P_2^+$ in Figure 5) — see Propositions 5, 6 and 7. Similarly, an e-outport is mapped to an e-outport or an i-port (e.g., $\alpha_P(p_2) = p_5 \in P_2^-$ in Figure 5) — see Propositions 5, 6 and 7. While a computon morphism can map external ports to internal ones, the latter can never be mapped to external ports due to the commutative diagrams presented in Figure 4.

Proposition 5. If $\alpha\colon \lambda_1 \to \lambda_2$ is a computon morphism, $\alpha^{-1}(P_2^+) \subseteq P_1^+$ and $\alpha^{-1}(P_2^-) \subseteq P_1^-$.

Proof. By letting $\alpha\colon \lambda_1 \to \lambda_2$ be a computon morphism, we only prove $\alpha^{-1}(P_2^+) \subseteq P_1^+$ by contrapositive, since the proof of $\alpha^{-1}(P_2^-) \subseteq P_1^-$ is completely analogous.

If $p_1 \in P \setminus P_1^+$, then $(\exists o_1 \in O_1)[t_1(o_1) = p_1]$ (see Definition 2). By commutativity, we know $t_2(\alpha(o_1)) = \alpha(t_1(o_1)) = \alpha(p_1)$ which implies $\alpha(p_1) \notin P_2^+$ (see Definition 2) and, consequently, $p_1 \notin \alpha^{-1}(P_2^+)$. As the implication $p_1 \notin P_1^+ \implies p_1 \notin \alpha^{-1}(P_2^+)$ is logically equivalent to $p_1 \in \alpha^{-1}(P_2^+) \implies p_1 \in P_1^+$, we conclude $\alpha^{-1}(P_2^+) \subseteq P_1^+$, as required. □

Proposition 6. If $\alpha\colon \lambda_1 \to \lambda_2$ is a computon morphism, $P_1^+ \cap \vec{i}(\alpha) = \emptyset \implies \alpha^{-1}(P_2^+) = P_1^+$ and $P_1^- \cap \vec{o}(\alpha) = \emptyset \implies \alpha^{-1}(P_2^-) = P_1^-$.

Proof. Let $\alpha\colon \lambda_1 \to \lambda_2$ be a computon morphism and assume $P_1^+ \cap \vec{i}(\alpha) = \emptyset$. This assumption says that if $p_1 \in P_1^+$ then $p_1 \notin \vec{i}(\alpha)$ so that $\bullet\alpha(p_1) \setminus \alpha(\bullet p_1) = \emptyset$ which is true when $\bullet\alpha(p_1) = \alpha(\bullet p_1)$. As $\bullet p_1 = \emptyset$ because $p_1 \in P_1^+$, we have $\bullet\alpha(p_1) = \emptyset = \alpha(\bullet p_1)$. The fact $\bullet\alpha(p_1) = \emptyset$ implies $\alpha(p_1) \in P_2^+$, i.e., $p_1 \in \alpha^{-1}(P_2^+)$. Thus, proving $P_1^+ \subseteq \alpha^{-1}(P_2^+)$. Since $\alpha^{-1}(P_2^+) \subseteq P_1^+$ also holds by Proposition 5, we conclude $P_1^+ \cap \vec{i}(\alpha) = \emptyset \implies \alpha^{-1}(P_2^+) = P_1^+$.

The proof of $P_1^- \cap \vec{o}(\alpha) = \emptyset \implies \alpha^{-1}(P_2^-) = P_1^-$ follows analogously. □

Proposition 7. If λ_1 is a connected computon and $\alpha\colon \lambda_1 \to \lambda_2$ is a computon morphism, $(P_1^+ \cap \vec{i}(\alpha)) \cup (P_1^- \cap \vec{o}(\alpha)) \subseteq \alpha^{-1}(Im(t_2) \cap Im(s_2))$.

Proof. Let $\alpha\colon \lambda_1 \to \lambda_2$ be a computon morphism from a connected computon λ_1 to an arbitrary computon λ_2. If $p_1 \in P_1^+ \cap \vec{i}(\alpha)$, then there is some $u_2 \in \bullet\alpha(p_1) \setminus \alpha(\bullet p_1)$. By Definition 5, there must also be some $o_2 \in O_2$ where $\sigma_2(o_2) = u_2$ and $t_2(o_2) = \alpha(p_1)$. That is, $\alpha(p_1) \in Im(t_2)$.

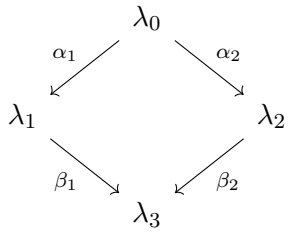
Now, since λ_1 is a connected computon and $p_1 \in P_1^+$, there is some $i_1 \in I_1$ and some $u_1 \in U_1$ where $p_1 \xrightarrow{i_1} u_1$ holds (see Definition 6 and Proposition 3). By commutativity and because $s_1(i_1) = p_1$, $s_2(\alpha(i_1)) = \alpha(s_1(i_1)) = \alpha(p_1)$. That is, $\alpha(p_1) \in Im(s_2)$.

As having $\alpha(p_1) \in Im(t_2) \cap Im(s_2)$ implies $p_1 \in \alpha^{-1}(Im(t_2) \cap Im(s_2))$, we have just proved $P_1^+ \cap \vec{i}(\alpha) \subseteq \alpha^{-1}(Im(t_2) \cap Im(s_2))$. The proof of $P_1^- \cap \vec{o}(\alpha) \subseteq \alpha^{-1}(Im(t_2) \cap Im(s_2))$ is completely analogous. □

2.3. Colimits in the Category of Computons

Although computons are set-valued functors, general colimits in $\mathbf{Set}^{\mathbf{Comp}}$ do not always exist because morphisms need to satisfy the special conditions imposed by Definition 7 and there are no initial objects, i.e., $\mathbf{Set}^{\mathbf{Comp}}$ is not cocomplete. When colimits exist, they are canonically computed component-wise in **Set** so they constitute formal category-theoretic operations to glue multiple computons together, according to the instructions given by the morphisms of a certain diagram. For example, a pushout construction glues computons by identifying their common parts in the form of an *apex computon*. This notion is formalised in Definition 8.

Definition 8 (Pushout Construction)**.** Given a span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms, the pushout of the corresponding diagram in $\mathbf{Set}^{\mathbf{Comp}}$:



denoted $(\beta_1 \colon \lambda_1 \to \lambda_3, \lambda_3, \beta_2 \colon \lambda_2 \to \lambda_3)$ or $\lambda_1 +_{\lambda_0} \lambda_2$, is obtained by computing the pushout in **Set** of each individual computon component, except Σ which simply is the union of Σ_1 and Σ_2:

$$P_3 = P_1 +_{P_0} P_2$$
$$U_3 = U_1 +_{U_0} U_2$$
$$I_3 = I_1 +_{I_0} I_2$$
$$O_3 = O_1 +_{O_0} O_2$$
$$\Sigma_3 = \Sigma_1 \cup \Sigma_2$$

with τ_3, σ_3, s_3, t_3 and c_3 being defined in the obvious way:

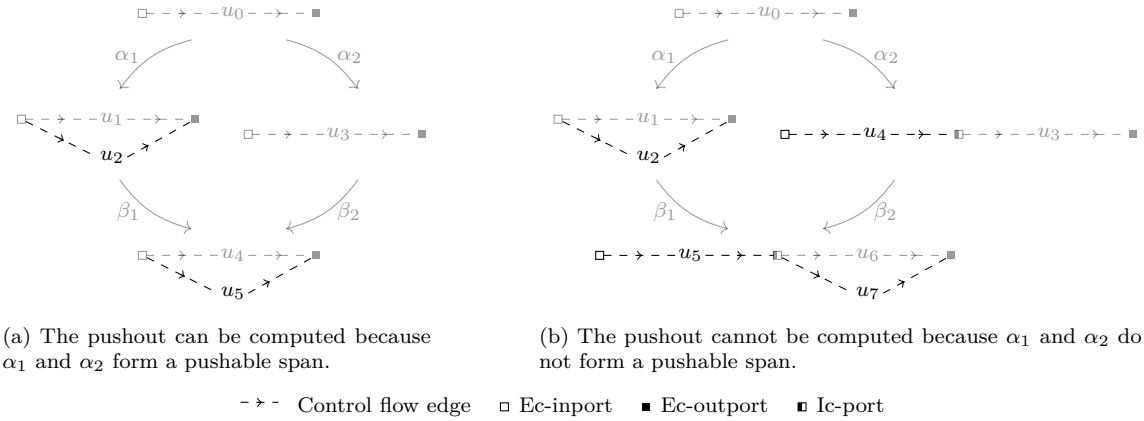
$$\tau_3 \colon I_1 +_{I_0} I_2 \twoheadrightarrow U_1 +_{U_0} U_2$$
$$\sigma_3 \colon O_1 +_{O_0} O_2 \twoheadrightarrow U_1 +_{U_0} U_2$$
$$s_3 \colon I_1 +_{I_0} I_2 \to P_1 +_{P_0} P_2$$
$$t_3 \colon O_1 +_{O_0} O_2 \to P_1 +_{P_0} P_2$$
$$c_3 \colon P_1 +_{P_0} P_2 \twoheadrightarrow \Sigma_1 +_{\Sigma_0} \Sigma_2$$

Unfortunately, a pushout operation cannot be computed for every span of computon morphisms. To understand why, let us recall that Definition 7 enforces a computon morphism to insert a computon into another only at the boundaries, so that computons can only "interact" through their e-ports. Four examples of valid computon morphisms are depicted in Figure 6(a).

Figure 6(b) shows that, unfortunately, defining a span of valid computon morphisms is not sufficient to compute a pushout in $\mathbf{Set}^{\mathbf{Comp}}$. A pushout only exists for spans that adhere to Definition 9.

Definition 9 (Pushable Span)**.** A span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms is pushable if $\alpha_1(\vec{i}(\alpha_2)) \cup \alpha_1(\vec{o}(\alpha_2)) \subseteq P_1^+ \cup P_1^-$ and $\alpha_2(\vec{i}(\alpha_1)) \cup \alpha_2(\vec{o}(\alpha_1)) \subseteq P_2^+ \cup P_2^-$.

Basically, Definition 9 says that a span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms is pushable when, for every port $p \in P_0$, the following holds:



(a) The pushout can be computed because α_1 and α_2 form a pushable span.

(b) The pushout cannot be computed because α_1 and α_2 do not form a pushable span.

Figure 6: An example to illustrate the semantics of a pushable span of computon morphisms.

1. If $\alpha_1(p)$ is connected to/from a computation unit of λ_1 that does not form part of the α_1-embedding, then $\alpha_2(p)$ must be either e-inport or e-outport in λ_2.
2. If $\alpha_2(p)$ is connected to/from a computation unit of λ_2 that does not form part of the α_2-embedding, then $\alpha_1(p)$ must be either e-inport or e-outport in λ_1.

To understand these conditions, consider the span formed by α_1 and α_2 in Figure 6(a), which embeds the apex computon into the parts highlighted in gray. Here, α_1 maps the e-inport p_0 of the apex to the e-inport of the left leg. As the e-inport of the left leg is connected to the computation unit u_2, which does not form part of the α_1-embedding, it follows that $p_0 \in \vec{o}(\alpha_1)$ (see Definition 7). According to the Condition 1 presented above, $\alpha_2(p_0)$ must be either e-inport or e-outport. As in this case $\alpha_2(p_0)$ is indeed an e-inport and the other conditions of Definition 9 are analogously satisfied, we conclude that our span is pushable.

Figure 6(b) presents a counterexample in which α_2 does not map the e-inport of the apex computon to an external port of the right leg but to an i-port that lies between the computation units u_3 and u_4. As $p_0 \in \vec{o}(\alpha_1)$ and $\alpha_2(p_0)$ is neither e-inport nor e-outport, we have that the Condition 1 presented above is not satisfied. Hence, the span from Figure 6(b), although valid, is not pushable.

In particular, the pushout of such a span cannot be computed because the induced computon morphism β_2 violates Definition 7, i.e., $\alpha_2(p_0) \in \vec{o}(\beta_2)$ but $\alpha_2(p_0)$ is neither e-inport nor e-outport. The fact a pushout construction can only be computed for pushable spans (see Proposition 8) entails that $\mathbf{Set}^{\mathbf{Comp}}$ does not have all pushouts. Nevertheless, when such a construction exists for a span whose legs are connected computons, the result of the corresponding operation is a connected computon (see Proposition 9).

Proposition 8. Let $\alpha_1 \colon \lambda_0 \to \lambda_1$ and $\alpha_2 \colon \lambda_0 \to \lambda_2$ be two computon morphisms. The pushout of α_1 and α_2 exists $\iff$ $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ is pushable.

Proof. ($\implies$) Assuming that the pushout $(\beta_1 \colon \lambda_1 \to \lambda_3, \lambda_3, \beta_2 \colon \lambda_2 \to \lambda_3)$ of $\alpha_1 \colon \lambda_0 \to \lambda_1$ and $\alpha_2 \colon \lambda_0 \to \lambda_2$ exists in $\mathbf{Set}^{\mathbf{Comp}}$, we just prove $\alpha_1(\vec{i}(\alpha_2)) \subseteq P_1^+ \cup P_1^-$ since the other conditions of Definition 9 follow analogously.

Supposing there is some $p_1 \in \alpha_1(\vec{i}(\alpha_2)) \setminus (P_1^+ \cup P_1^-)$, we know there is a port $p_0 \in \vec{i}(\alpha_2)$ where $\alpha_1(p_0) = p_1$. As the pushout $(\beta_1, \lambda_3, \beta_2)$ exists in $\mathbf{Set}^{\mathbf{Comp}}$, the equation $\beta_1(\alpha_1(p_0)) = \beta_1(p_1) = \beta_2(\alpha_2(p_0))$ holds. Since $p_0 \in \vec{i}(\alpha_2)$, there is some $u_2 \in \bullet\alpha_2(p_0) \setminus \alpha_2(\bullet p_0)$ so that $\beta_1(p_1) = \beta_2(\alpha_2(p_0)) \in \beta_2(u_2)\bullet$. As $p_1 \notin P_1^+ \cup P_1^-$, $p_1 \notin \vec{i}(\beta_1)$ (see Definition 7), meaning that there is some $u_1 \in \bullet p_1$ where $\beta_1(u_1) = \beta_2(u_2)$. Using commutativity, we deduce the existence of $u_0 \in U_0$ such that $\alpha_1(u_0) = u_1$ and $\alpha_2(u_0) = u_2$. As this contradicts the fact $u_2 \in \bullet\alpha_2(p_0) \setminus \alpha_2(\bullet p_0)$, we conclude $\alpha_1(\vec{i}(\alpha_2)) \subseteq P_1^+ \cup P_2^-$.

($\impliedby$) Assuming $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ is a pushable span of computon morphisms (as per Definition 9), we prove that the pushout $(\beta_1 \colon \lambda_1 \to \lambda_3, \lambda_3, \beta_2 \colon \lambda_2 \to \lambda_3)$ of α_1 and α_2 can be

constructed via Definition 8. For this, we first prove that β_1 and β_2 are computon morphisms. Below we provide the proof for β_1 only, since the other is completely analogous.

As **Set** has all pushouts, the existence of each component of β_1 and β_2 can be directly deduced. For example, the β_1-component embedding ports of P_1 into P_3 exists because $P_3 = P_1 +_{P_0} P_2$ can always be computed in **Set**. Consequently, the equations $\beta_i \circ c_1 = c_3 \circ \beta_i$, $\beta_i \circ \tau_1 = \tau_3 \circ \beta_i$, $\beta_i \circ \sigma_1 = \sigma_3 \circ \beta_i$, $\beta_i \circ s_1 = s_3 \circ \beta_i$ and $\beta_i \circ t_1 = t_3 \circ \beta_i$ hold for $i = 1, 2$ (see the commutative diagrams of Definition 7).

For the Σ-component of β_1 to be an inclusion (as required by Definition 7), we simply choose the canonical function $f : \Sigma_1 \to \Sigma_1 \cup \Sigma_2$ given by $f(x) = x$ for all $x \in \Sigma_1$. We now show $\vec{i}(\beta_1) \cup \vec{o}(\beta_1) \subseteq P_1^+ \cup P_1^-$ holds.

If $p_1 \in \vec{i}(\beta_1)$ then $\bullet\beta_1(p_1) \setminus \beta_1(\bullet p_1) \neq \emptyset$ so there exists some $u_3 \in \bullet\beta_1(p_1) \setminus \beta_1(\bullet p_1)$ and no $u_1 \in \bullet p_1$ where $\beta_1(u_1) = u_3$. Since $U_3 = U_1 +_{U_0} U_2$ (by Definition 8), there must be some $u_2 \in U_2$ where $\beta_2(u_2) = u_3 \in \bullet\beta_1(p_1) \setminus \beta_1(\bullet p_1)$ and, consequently, some $o_3 \in O_3$ where $\beta_2(u_2) \xrightarrow{o_3} \beta_1(p_1)$ (see Definition 5). As there is no $o_1 \in O_1$ satisfying $\beta_1(\sigma_1(o_1)) = \sigma_3(\beta_1(o_1)) = \sigma_3(o_3) = u_3$ because there is no $u_1 \in U_1$ satisfying $\beta_1(u_1) = u_3$, $O_3 = O_1 +_{O_0} O_2$ implies that there must be some $o_2 \in O_2$ for which $\beta_2(o_2) = o_3$ is true. As $\beta_2(u_2) \xrightarrow{\beta_2(o_2)} \beta_1(p_1)$ holds, we use the commutativity property to deduce the existence of $p_2 \in P_2$ such that $\beta_2(p_2) = \beta_1(p_1)$ and $u_2 \in \bullet p_2$. As $\beta_2(p_2) = \beta_1(p_1)$ and $P_3 = P_1 +_{P_0} P_2$, we use again commutativity to deduce there also is a port $p_0 \in P_0$ with $\alpha_1(p_0) = p_1$ and $\alpha_2(p_0) = p_2$ so that $u_2 \in \bullet\alpha_2(p_0)$. Since $U_3 = U_1 +_{U_0} U_2$ and $(\nexists u_1 \in U_1)[\beta_1(u_1) = u_3 = \beta_2(u_2)]$, we have that there is no $u_0 \in U_0$ where $\alpha_2(u_0) = u_2$. That is, $u_2 \notin \alpha_2(\bullet p_0)$ because $\bullet p_0 \subseteq U_0$. Thus, it is true that $u_2 \in \bullet\alpha_2(p_0) \setminus \alpha_2(\bullet p_0)$ and, therefore, that $p_0 \in \vec{i}(\alpha_2)$ (see Definition 7).

Since $\alpha_1(\vec{i}(\alpha_2)) \subseteq P_1^+ \cup P_1^-$ (by Definition 9) and $p_0 \in \vec{i}(\alpha_2)$ (by the above), we have $\alpha_1(p_0) = p_1 \in P_1^+ \cup P_1^-$. A similar approach can be used to show that $q_1 \in \vec{o}(\beta_1)$ implies $q_1 \in P_1^+ \cup P_1^-$. So, $\vec{i}(\beta_1) \cup \vec{o}(\beta_1) \subseteq P_1^+ \cup P_1^-$.

Having proved β_1 is a computon morphism, we now assume $\gamma_1 \colon \lambda_1 \to \lambda_4$ and $\gamma_2 \colon \lambda_2 \to \lambda_4$ are computon morphisms with $\gamma_1 \circ \alpha_1 = \gamma_2 \circ \alpha_2$, in order to show there is a unique computon morphism $\gamma_3 \colon \lambda_3 \to \lambda_4$ such that the corresponding diagram commutes. As it is obvious that the Σ-component of γ_3 is an inclusion function because the Σ-components of β_i and γ_i also are (for $i = 1, 2$) and because $\Sigma_3 = \Sigma_1 \cup \Sigma_2$, we just prove $\vec{i}(\gamma_3) \cup \vec{o}(\gamma_3) \subseteq P_3^+ \cup P_3^-$.

Let $p_3 \in \vec{i}(\gamma_3)$ so $\bullet\gamma_3(p_3) \setminus \gamma_3(\bullet p_3) \neq \emptyset$. As $P_3 = P_1 +_{P_0} P_2$, we observe $p_3 = \beta_j(p_j)$ for some $p_j \in P_j$ with $j = 1, 2$. With this in mind, we perform the following operations:

$$
\begin{aligned}
\emptyset \neq \bullet\gamma_3(p_3) \setminus \gamma_3(\bullet p_3) &= \bullet\gamma_3(\beta_j(p_j)) \setminus \gamma_3(\bullet\beta_j(p_j)) \\
&= \bullet\gamma_j(p_j) \setminus \gamma_3(\bullet\beta_j(p_j)) && \text{because } \gamma_3 \circ \beta_j = \gamma_j \\
&\subseteq \bullet\gamma_j(p_j) \setminus \gamma_3(\beta_j(\bullet p_j)) && \text{because } \beta_j(\bullet p_j) \subseteq \bullet\beta_j(p_j) \\
&= \bullet\gamma_j(p_j) \setminus \gamma_j(\bullet p_j) && \text{because } \gamma_3 \circ \beta_j = \gamma_j
\end{aligned}
$$

By the above, we deduce $p_j \in \vec{i}(\gamma_j)$ and, consequently, $p_j \in P_j^+ \cup P_j^-$ (because γ_j is a computon morphism with $\vec{i}(\gamma_j) \subseteq P_j^+ \cup P_j^-$ — see Definition 7). Using the facts $p_3 = \beta_j(p_j) \in \vec{i}(\gamma_3)$ and $p_j \in P_j^+ \cup P_j^-$, we further deduce $\beta_j^{-1}(p_3) \subseteq P_j^+ \cup P_j^-$. Hence, $p_3 \in P_3^+ \cup P_3^-$ by Proposition 5. The proof of $q_3 \in \vec{o}(\gamma_3) \implies q_3 \in P_3^+ \cup P_3^-$ is completely analogous.

As the Σ-component of γ_3 is an inclusion function and $\vec{i}(\gamma_3) \cup \vec{o}(\gamma_3) \subseteq P_3^+ \cup P_3^-$, we conclude that γ_3 is a computon morphism. Such a morphism is unique because each of its components are unique (by the fact that a pushout in **Set** satisfies the universal property). □

Proposition 9. Let $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ be a pushable span of computon morphisms. If λ_1 and λ_2 are connected computons, then the pushout of α_1 and α_2 is a connected computon.

Proof. Let $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ be a pushable span of computon morphisms and assume λ_1 and λ_2 are connected computons. By Proposition 8, $(\beta_1 \colon \lambda_1 \to \lambda_3, \lambda_3, \beta_2 \colon \lambda_2 \to \lambda_3)$ can be constructed from α_1 and α_2.

To prove λ_3 is a connected computon, let $p_3 \in Im(s_3) \cup P_3^+$. Since $P_3 = P_1 +_{P_0} P_2$, we know p_3 is identified with a port from (1) λ_1, (2) λ_2 or (3) both. Before considering these three scenarios, let V_i be a sequence of visited e-outports of λ_i for $i \in \{1, 2\}$. The sequence V_i is initially empty and is iteratively updated through the following process.

1. If p_3 is exclusively identified with a port from λ_1, i.e., there is a port $p_1 \in P_1$ where $\beta_1(p_1) = p_3$, we have that $p_1 \in P_1^-$ cannot hold because:
   - If $\beta_1(p_1) = p_3 \in Im(s_3)$, then there is some $i_3 \in I_3$ where $s_3(i_3) = \beta_1(p_1) = p_3$. As there is no $i_1 \in I_1$ with $s_1(i_1) = p_1$ because $p_1 \in P_1^-$, $I_3 = I_1 +_{I_0} I_2$ implies there must be some $i_2 \in I_2$ for which $\beta_2(i_2) = i_3$ holds. By the totality of s_2, there must also be some $p_2 \in P_2$ where $s_2(i_2) = p_2$. Using the commutative equations derived from β_2, we obtain: $\beta_2(s_2(i_2)) = s_3(\beta_2(i_2)) = s_3(i_3) = \beta_1(p_1) = p_3$ which violates our assumption that p_3 is exclusively identified with a λ_1-port.
   - If $\beta_1(p_1) = p_3 \in P_3^+$, we use Proposition 5 to deduce $p_1 \in P_1^+$. As Proposition 2 says $P_1^+ \cap P_1^- = \emptyset$ because λ_1 is connected, we have that $p_1 \in P_1^+ \cap P_1^-$ cannot hold.

   Now, if $p_1 \notin P_1^-$, then there is some $i_1 \in I_1$ where $s_1(i_1) = p_1$, i.e., $p_1 \in Im(s_1)$, meaning $p_1 \xrightarrow{\exists} q_1$ must hold for some $q_1 \in P_1^-$ with $(\nexists k)[V_1[k] = q_1]$ (because λ_1 is a connected computon). By the commutativity property of β_1, $\beta_1(p_1) \xrightarrow{\exists} \beta_1(q_1)$ must also hold. We now append $\langle q_1 \rangle$ to V_1 and consider the following two cases:
   - $\beta_1(q_1) \in P_3^-$ when there is no $p_2 \in P_2$ such that $\beta_1(q_1) = \beta_2(p_2)$ (i.e., $\beta_1(q_1)$ is exclusively identified with a λ_1-port) or when $\beta_1(q_1)$ is identified only with e-outports of λ_2. As $\beta_1(p_1) = p_3 \in Im(s_3) \cup P_3^+$ and $\beta_1(q_1) \in P_3^-$, λ_3 must be a connected computon.
   - $\beta_1(q_1) \notin P_3^-$ when there is some $p_2 \in P_2 \setminus P_2^-$ with $\beta_1(q_1) = \beta_2(p_2)$. As λ_2 is a connected computon, $p_2 \xrightarrow{\exists} q_2$ for some $q_2 \in P_2^-$ and, therefore, $\beta_2(p_2) \xrightarrow{\exists} \beta_2(q_2)$ must also hold by the commutativity property of β_2. Considering $\beta_1(p_1) = p_3$ and $\beta_1(q_1) = \beta_2(p_2)$, we have $p_3 \xrightarrow{\exists} \beta_2(p_2) \xrightarrow{\exists} \beta_2(q_2)$. If $\beta_2(q_2) \in P_3^-$, then λ_3 is a connected computon. If not, perform 1, 2 or 3, whichever holds for $\beta_2(q_2)$.
2. The proof when p_3 is exclusively identified with a λ_2-port is symmetric to that of (1).
3. The port p_3 is identified with a port from both λ_1 and λ_2, i.e., $p_3 = \beta_1(p_1) = \beta_2(p_2)$ for some $p_1 \in P_1$ and some $p_2 \in P_2$. In this case, if $p_1 \in Im(s_1) \cup P_1^+$, there is some $q_1 \in P_1^-$ for which $p_1 \xrightarrow{\exists} q_1$ and $(\nexists k)[V_1[k] = q_1]$ because λ_1 is a connected computon; consequently, we append $\langle q_1 \rangle$ to V_1 and infer $\beta_1(p_1) \xrightarrow{\exists} \beta_1(q_1)$ from the commutativity property of β_1. If $p_2 \in Im(s_2) \cup P_2^+$, there is some $q_2 \in P_2^-$ such that $(\nexists k)[V_2[k] = q_2]$ and $p_2 \xrightarrow{\exists} q_2$ because λ_2 is a connected computon; hence, we append $\langle q_2 \rangle$ to V_2 and infer $\beta_2(p_2) \xrightarrow{\exists} \beta_2(q_2)$ from the commutativity property of β_2.

   Now, perform the following check nondeterministically for each $j = 1, 2$: If $\beta_j(p_j) \xrightarrow{\exists} \beta_j(q_j)$ and $\beta_j(q_j) \in P_3^-$, there is information flow from $p_3 = \beta_1(p_1) = \beta_2(p_2)$ to an e-outport of λ_3, meaning λ_3 is a connected computon. If $\beta_j(p_j) \xrightarrow{\exists} \beta_j(q_j)$ and $\beta_j(q_j) \notin P_3^-$, perform 1, 2 or 3, whichever holds for $\beta_j(q_j)$.

Checking from 1 to 3 is an iterative process which is repeated until yielding $p_3 \xrightarrow{\exists} \cdots \xrightarrow{\exists} q_3$ for the initial p_3 and some $q_3 \in P_3^-$. Termination is guaranteed because (i) the number of ports, edges and computation units of λ_3 is finite; (ii) V_i contains all the visited e-outports of λ_i so each iteration extends the information flow pipeline to a new e-outport; and (iii) λ_1 and λ_2 are both connected computons. As the iterative process will eventually devise information flow from p_3 to an e-outport of λ_3, we conclude λ_3 must be a connected computon. □

Pushouts are not the only colimits that can be computed in $\mathbf{Set}^{\mathbf{Comp}}$. Another useful operation for describing our theory of computons is that of coproduct which intuitively allows the definition of a side-by-side computon. As per Proposition 10, this operation can always be computed in $\mathbf{Set}^{\mathbf{Comp}}$ so that such a category has all coproducts. Computing the coproduct of two connected computons results in another connected computon, as shown in Proposition 11.

Proposition 10. The coproduct $\lambda_1 + \lambda_2$ of computons λ_1 and λ_2 always exists in $\mathbf{Set}^{\mathbf{Comp}}$.

Proof. The coproduct λ_3 of a computon λ_1 and a computon λ_2, written $\lambda_1 + \lambda_2$, is obtained by computing the following in **Set**: $P_3 = P_1 + P_2$, $U_3 = U_1 + U_2$, $I_3 = I_1 + I_2$, $O_3 = O_1 + O_2$ and $\Sigma_3 = \Sigma_1 \cup \Sigma_2$. Particularly, the operation to obtain Σ_3 can always be computed since the cospan $\Sigma_1 \hookrightarrow \Sigma_1 \cup \Sigma_2 \hookleftarrow \Sigma_2$ of unique inclusion functions always exists in **Set** (because $\Sigma_1, \Sigma_2 \subset \mathbb{N}$). This operation also satisfies the universal property in the sense that, for any set $\Sigma_4 \subset \mathbb{N}$ with inclusions $\Sigma_1 \hookrightarrow \Sigma_4$ and $\Sigma_2 \hookrightarrow \Sigma_4$, there is a unique inclusion $\Sigma_1 \cup \Sigma_2 \hookrightarrow \Sigma_4$. Consequently, the function c_3 is canonically identified with the mapping $P_1 + P_2 \twoheadrightarrow \Sigma_1 \cup \Sigma_2$ which is surjective by the fact $(\forall j \in \{1,2\})(\forall p \in P_j)(\exists! x \in \Sigma_j)[c_j(p) = x]$. All the functions of λ_3, including c_3, are defined in the obvious way to make the corresponding squares commute. For example: $\forall o_3 \in O_3, \sigma_3(o_3) = \begin{cases} (\beta_1 \circ \sigma_1)(o_1) & \text{if } o_3 = \beta_1(o_1) \text{ for some } o_1 \in O_1 \\ (\beta_2 \circ \sigma_2)(o_2) & \text{if } o_3 = \beta_2(o_2) \text{ for some } o_2 \in O_2 \end{cases}$ where $\beta_k \colon \lambda_k \to \lambda_3$ is the canonical injection into the coproduct λ_3.

The existence of each component of β_k follows directly from the fact that **Set** has all coproducts. Particularly, the Σ-component of β_k is the unique inclusion $\Sigma_k \hookrightarrow \Sigma_1 \cup \Sigma_2$ while the others are obvious morphisms of the form $A_k \to A_1 + A_2$ such as $U_k \to U_1 + U_2$. Computing U_3 as the disjoint union of U_1 and U_2 implies $\vec{i}(\beta_k) \cup \vec{o}(\beta_k) = \emptyset \subseteq P_k^+ \cup P_k^-$.

As the coproduct of each component of λ_3 is computed in **Set** and **Set** has all coproducts, it is true that coproduct in $\mathbf{Set}^{\mathbf{Comp}}$ satisfies the universal property. This means that, if there is a computon λ_4 with morphisms $\gamma_1 \colon \lambda_1 \to \lambda_4$ and $\gamma_2 \colon \lambda_2 \to \lambda_4$, there is a unique morphism $\gamma_3 \colon \lambda_3 \to \lambda_4$ such that $\gamma_3 \circ \beta_1 = \gamma_1$ and $\gamma_3 \circ \beta_2 = \gamma_2$. As the Σ-components of γ_1 and γ_2 are both inclusion functions it is easy to see that the Σ-component of γ_3 is the unique inclusion $\Sigma_3 \hookrightarrow \Sigma_4$. The other components of γ_3 are given in the obvious way. For example:

$$\forall p_3 \in P_3, \gamma_3(p_3) = \begin{cases} \gamma_1(p_1) & \text{if } p_3 = \beta_1(p_1) \text{ for some } p_1 \in P_1 \\ \gamma_2(p_2) & \text{if } p_3 = \beta_2(p_2) \text{ for some } p_2 \in P_2 \end{cases}$$

To meet the rest of the requirements of Definition 7 for γ_3, we just now have to prove $\vec{i}(\gamma_3) \cup \vec{o}(\gamma_3) \subseteq P_3^+ \cup P_3^-$. Below we provide the proof of $\vec{i}(\gamma_3) \subseteq P_3^+ \cup P_3^-$ since the other is completely analogous.

Let $p_3 \in \vec{i}(\gamma_3)$ so $\bullet\gamma_3(p_3) \setminus \gamma_3(\bullet p_3) \neq \emptyset$. As $P_3 = P_1 + P_2$, we observe that $p_3 = \beta_n(p_n)$ for some $p_n \in P_n$ $(n = 1, 2)$. Using a similar reasoning as the proof of Proposition 8, we deduce $p_n \in \vec{i}(\gamma_n)$ which implies $p_n \in P_n^+ \cup P_n^-$ because γ_n is a computon morphism with $\vec{i}(\gamma_n) \subseteq P_n^+ \cup P_n^-$ (see Definition 7). As $\vec{i}(\beta_n) \cup \vec{o}(\beta_n) = \emptyset$, $P_n \cap \vec{i}(\beta_n) = \emptyset = P_n \cap \vec{o}(\beta_n)$ and, consequently, $\beta_n^{-1}(P_3^+) = P_n^+$ and $\beta_n^{-1}(P_3^-) = P_n^-$ (see Proposition 6). That is, $p_n \in P_n^+ \cup P_n^- \iff p_n \in \beta_n^{-1}(P_3^+) \cup \beta_n^{-1}(P_3^-)$. Using the fact $p_3 = \beta_n(p_n)$, we conclude $p_3 \in P_3^+ \cup P_3^-$. □

Proposition 11. The coproduct of two connected computons is a connected computon.

Proof. By Proposition 10, we know the coproduct λ_3 of connected computons λ_1 and λ_2 exists. To prove λ_3 is also connected, consider a port $p_3 \in Im(s_3) \cup P_3^+$ which is necessarily identified with a port from either λ_1 or λ_2 by the definition of coproduct. Assuming $p_3 = \beta_k(p)$ for some $p \in P_k$ and $k \in \{1, 2\}$, we prove by cases:

- If $p_3 \in Im(s_3)$, there is some $i_3 \in I_3$ where $s_3(i_3) = p_3 = \beta_k(p)$. By commutativity, there also is some $i \in I_k$ for which $\beta_k(i) = i_3$ and $s_k(i) = p$, i.e., $p \in Im(s_k)$. As λ_k is a

connected computon, $p \xrightarrow{\exists} q$ holds for some $q \in P_k^-$ (see Definition 6). Again, by commutativity, $\beta_k(p) \xrightarrow{\exists} \beta_k(q)$ so that $p_3 \xrightarrow{\exists} \beta_k(q)$. Using coproduct definition and $q \in P_k^-$, we deduce $\beta_k(q) \in P_3^-$.

- If $p_3 \in P_3^+$, then $p \in P_k^+$ by Proposition 5. As λ_k is a connected computon, we have some $q \in P_k^-$ for which $p \xrightarrow{\exists} q$. Hence, $\beta_k(p) \xrightarrow{\exists} \beta_k(q)$ by commutativity and, therefore, $p_3 \xrightarrow{\exists} \beta_k(q)$. Using coproduct definition and $q \in P_k^-$, we deduce $\beta_k(q) \in P_3^-$.

Having information flow from an arbitrary port in $Im(s_3) \cup P_3^+$ to an e-outport of λ_3 entails that λ_3 is a connected computon. □

2.4. Control Flow and Data Flow Structures

The control flow structure of a computon can be expressed as an object in the category of directed labelled graphs and graph homomorphisms, $\mathbf{Set}^{\mathbf{Gr}}$, which is formalised below.

Definition 10. Let **Gr** be the category freely generated by the following diagram:

$$E \underset{\vec{t}}{\overset{\vec{s}}{\rightrightarrows}} V \xrightarrow{\ l\ } L$$

which gives rise to the functor category $\mathbf{Set}^{\mathbf{Gr}}$ of directed labelled graphs and graph homomorphisms [27]. In this category, composition is defined component-wise and the components of every identity morphism are all identity functions (which map domain elements to themselves). Even though a directed labelled graph G is a functor $\mathbf{Gr} \to \mathbf{Set}$, we simplify notation by writing V for the set $G(V)$ of vertices, E for the set $G(E)$ of edges and L for the set $G(L)$ of labels. We similarly write $\vec{s}$ for the source function $G(\vec{s})$, $\vec{t}$ for the target function $G(\vec{t})$ and l for the labelling function $G(l)$. Whenever there is a subindex for a graph, we use the same subindex for its components, e.g., V_1 for G_1. The notation $\vec{s}$ and $\vec{t}$ is used to avoid confusion with the s and t functions of a computon λ.

As the control flow structure of a computon is an object in $\mathbf{Set}^{\mathbf{Gr}}$, we refer to it as a Control Flow Graph (CFG) which is constructed via the application of the functor described in Definition 11.

Definition 11 (Computon CFG)**.** The functor $\mathscr{C}\colon \mathbf{Set}^{\mathbf{Comp}} \to \mathbf{Set}^{\mathbf{Gr}}$ maps each computon λ to its underlying CFG $\mathscr{C}(\lambda)$ as follows:

- The set V of vertices of $\mathscr{C}(\lambda)$ is $U \cup \{p \in P \mid c(p) = 0\}$.
- The set E of edges of $\mathscr{C}(\lambda)$ is $\{i \in I \mid c(s(i)) = 0\} \cup \{o \in O \mid c(t(o)) = 0\}$.
- The set L of labels of $\mathscr{C}(\lambda)$ is $\{0, \kappa\}$.
- The source function $\vec{s}\colon E \to V$ of $\mathscr{C}(\lambda)$ is given by $\vec{s}(e) = \begin{cases} s(e) & \text{if } e \in I \\ \sigma(e) & \text{if } e \in O \end{cases}$
- The target function $\vec{t}\colon E \to V$ of $\mathscr{C}(\lambda)$ is given by $\vec{t}(e) = \begin{cases} t(e) & \text{if } e \in O \\ \tau(e) & \text{if } e \in I \end{cases}$
- The labelling function $l\colon V \to L$ of $\mathscr{C}(\lambda)$ is given by $l(v) = \begin{cases} c(v) & \text{if } v \in P \\ \kappa & \text{if } v \in U \end{cases}$

For a computon morphism $\alpha\colon \lambda_1 \to \lambda_2$, there is a graph homomorphism $\mathscr{C}(\alpha)\colon \mathscr{C}(\lambda_1) \to \mathscr{C}(\lambda_2)$ such that:

- $\mathscr{C}(\alpha)_V\colon V_1 \to V_2$ is given by $\mathscr{C}(\alpha)_V(v) = \begin{cases} \alpha_U(v) & \text{if } v \in U_1 \\ \alpha_P(v) & \text{if } v \in P_1 \end{cases}$

- $\mathscr{C}(\alpha)_E \colon E_1 \to E_2$ is given by $\mathscr{C}(\alpha)_E(e) = \begin{cases} \alpha_I(e) & \text{if } e \in I_1 \\ \alpha_O(e) & \text{if } e \in O_1 \end{cases}$
- $\mathscr{C}(\alpha)_L \colon L_1 \to L_2$ is given by $\mathscr{C}(\alpha)_L(x) = \begin{cases} \alpha_\Sigma(x) & \text{if } x \in \Sigma_1 \\ \kappa & \text{otherwise} \end{cases}$

Definition 11 indicates that a computon CFG is obtained by just considering control flow edges, control ports and computation units, while ignoring data elements. As computation units are not labelled within computons, all of them take the constant label κ in the corresponding CFG. The construction given by Definition 11 is functorial in the sense $\mathscr{C}$ preserves the structure of $\mathbf{Set}^{\mathbf{Comp}}$, including composition and identities (see Proposition 12).

Proposition 12. The process of building a computon CFG is functorial, i.e., $\mathscr{C}$ is a functor.

Proof. Given Definition 11, we first prove that, for every computon morphism $\alpha \colon \lambda_1 \to \lambda_2$, $\mathscr{C}(\alpha) \colon \mathscr{C}(\lambda_1) \to \mathscr{C}(\lambda_2)$ is a graph homomorphism that preserves labels and oriented incidence. To verify this, we show that the following diagram commutes:

$$\begin{array}{ccccc} E_1 & \underset{\vec{t}_1}{\overset{\vec{s}_1}{\rightrightarrows}} & V_1 & \xrightarrow{l_1} & L_1 \\ {\scriptstyle \mathscr{C}(\alpha)_E}\downarrow & & \downarrow{\scriptstyle \mathscr{C}(\alpha)_V} & & \downarrow{\scriptstyle \mathscr{C}(\alpha)_L} \\ E_2 & \underset{\vec{t}_2}{\overset{\vec{s}_2}{\rightrightarrows}} & V_2 & \xrightarrow[l_2]{} & L_2 \end{array}$$

For our proof, we simplify notation by omitting the (α) part of a $\mathscr{C}(\alpha)$-component; for example, we write $\mathscr{C}_E$ for $\mathscr{C}(\alpha)_E$. We only verify the equation $l_2 \circ \vec{s}_2 \circ \mathscr{C}_E = \mathscr{C}_L \circ l_1 \circ \vec{s}_1$ since the equation $l_2 \circ \vec{t}_2 \circ \mathscr{C}_E = \mathscr{C}_L \circ l_1 \circ \vec{t}_1$ can be showed analogously. By letting $e \in E_1$, we have two cases:

1. $e \in I_1$:

$$\begin{aligned} l_2(\vec{s}_2(\mathscr{C}_E(e))) &= l_2(\vec{s}_2(\alpha_I(e))) && \text{by the definition of } \mathscr{C}(\alpha)_E \text{ and considering } e \in I_1 \\ &= l_2(s_2(\alpha_I(e))) && \text{by the definition of } \vec{s}_2 \text{ and considering } \alpha_I(e) \in I_2 \\ &= l_2(\alpha_P(s_1(e))) && \text{because } s_2 \circ \alpha_I = \alpha_P \circ s_1 \\ &= c_2(\alpha_P(s_1(e))) && \text{by the definition of } l_2 \text{ and considering } \alpha_P(s_1(e)) \in P_2 \\ &= \alpha_\Sigma(c_1(s_1(e))) && \text{because } c_2 \circ \alpha_P = \alpha_\Sigma \circ c_1 \\ &= \mathscr{C}_L(c_1(s_1(e))) && \text{by the definition of } \mathscr{C}(\alpha)_L \text{ and considering } c_1(s_1(e)) \in \Sigma_1 \\ &= \mathscr{C}_L(l_1(s_1(e))) && \text{by the definition of } l_1 \text{ and considering } s_1(e) \in P_1 \\ &= \mathscr{C}_L(l_1(\vec{s}_1(e))) && \text{by the definition of } \vec{s}_1 \text{ and considering } e \in I_1 \end{aligned}$$

2. $e \in O_1$:

$$\begin{aligned} l_2(\vec{s}_2(\mathscr{C}_E(e))) &= l_2(\vec{s}_2(\alpha_O(e))) && \text{by the definition of } \mathscr{C}(\alpha)_E \text{ and considering } e \in O_1 \\ &= l_2(\sigma_2(\alpha_O(e))) && \text{by the definition of } \vec{s}_2 \text{ and considering } \alpha_O(e) \in O_2 \\ &= \kappa && \text{by the definition of } l_2 \text{ and considering } \sigma_2(\alpha_O(e)) \in U_2 \\ &= \mathscr{C}_L(\kappa) && \text{by the definition of } \mathscr{C}_L \text{ and considering } \kappa \in L_1 \setminus \Sigma_1 \\ &= \mathscr{C}_L(l_1(\sigma_1(e))) && \text{by the definition of } l_1 \text{ and considering } \sigma_1(e) \in U_1 \\ &= \mathscr{C}_L(l_1(\vec{s}_1(e))) && \text{by the definition of } \vec{s}_1 \text{ and considering } e \in O_1 \end{aligned}$$

As the above diagram commutes, it follows that $\mathscr{C}(\alpha)$ preserves labels, sources and targets. To verify $\mathscr{C}$ preserves identities, we have to show $\mathscr{C}(1_\lambda) = 1_{\mathscr{C}(\lambda)}$ for any computon λ, which is trivially true since both identity computon morphisms and identity graph homomorphisms are built upon identity functions. Considering the computon morphisms $\alpha_1 \colon \lambda_1 \to \lambda_2$ and $\alpha_2 \colon \lambda_2 \to \lambda_3$, we now show that composition is preserved too.

If $v \in V_1$, then either $v \in U_1$ or $v \in P_1$. We only prove the first case since the proof of the other is symmetric:

$$\begin{aligned}
\mathscr{C}(\alpha_2 \circ \alpha_1)_V(v) &= (\alpha_2 \circ \alpha_1)_U(v) && \text{by the definition of } \mathscr{C}(\alpha_2 \circ \alpha_1)_V \text{ and considering } v \in U_1 \\
&= \alpha_2(\alpha_1(v)) && \text{by Notation 2} \\
&= (\mathscr{C}(\alpha_2)_V \circ \alpha_1)(v) && \text{by the definition of } \mathscr{C}(\alpha_2)_V \text{ and considering } \alpha_1(v) \in U_2 \\
&= (\mathscr{C}(\alpha_2)_V \circ \mathscr{C}(\alpha_1)_V)(v) && \text{by the definition of } \mathscr{C}(\alpha_1)_V \text{ and considering } v \in U_1
\end{aligned}$$

A similar approach can be used to show $\mathscr{C}(\alpha_2 \circ \alpha_1)_E = \mathscr{C}(\alpha_2)_E \circ \mathscr{C}(\alpha_1)_E$ and $\mathscr{C}(\alpha_2 \circ \alpha_1)_L = \mathscr{C}(\alpha_2)_L \circ \mathscr{C}(\alpha_1)_L$ so $\mathscr{C}(\alpha_2 \circ \alpha_1) = \mathscr{C}(\alpha_2) \circ \mathscr{C}(\alpha_1)$ in general. As $\mathscr{C}$ preserves structure, identities and composition (the proof of associativity of composition is similar to the above), we conclude that the construction presented in Definition 11 is functorial. □

Interestingly, the control flow structure of any computon can always be embodied by some other computon; thus, giving rise to the endofunctor described in Definition 12.

Definition 12 (Control Flow Endofunctor)**.** The endofunctor $\mathfrak{E}\colon \mathbf{Set}^{\mathbf{Comp}} \to \mathbf{Set}^{\mathbf{Comp}}$ maps a computon λ to a computon $\mathfrak{E}(\lambda)$ as follows:

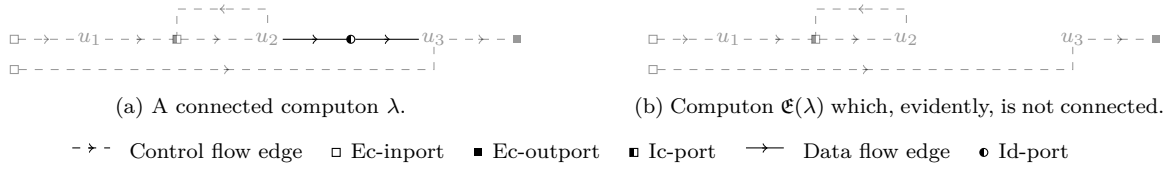
- The set $\mathfrak{E}(U)$ of computation units is U,
- The set $\mathfrak{E}(O)$ of edges is given by $\{o \in O \mid c(t(o)) = 0\}$,
- The set $\mathfrak{E}(I)$ of edges is given by $\{i \in I \mid c(s(i)) = 0\}$,
- The set $\mathfrak{E}(P)$ of ports is given by $\{p \in P \mid c(p) = 0\}$,
- The set $\mathfrak{E}(\Sigma)$ of colours is $\{0\}$,
- The function $\mathfrak{E}(\sigma)$ is given by $\sigma \restriction_{\mathfrak{E}(O)}$,
- The function $\mathfrak{E}(t)$ is given by $t \restriction_{\mathfrak{E}(O)}$,
- The function $\mathfrak{E}(\tau)$ is given by $\tau \restriction_{\mathfrak{E}(I)}$,
- The function $\mathfrak{E}(s)$ is given by $s \restriction_{\mathfrak{E}(I)}$ and
- The function $\mathfrak{E}(c)$ is given by $c \restriction_{\mathfrak{E}(P)}$.

Given a computon morphism $\alpha\colon \lambda_1 \to \lambda_2$, the components of $\mathfrak{E}(\alpha)\colon \mathfrak{E}(\lambda_1) \to \mathfrak{E}(\lambda_2)$ are defined as follows:

- $\mathfrak{E}(\alpha)_U\colon \mathfrak{E}(U_1) \to \mathfrak{E}(U_2)$ by $\mathfrak{E}(\alpha)_U = \alpha_U$,
- $\mathfrak{E}(\alpha)_O\colon \mathfrak{E}(O_1) \to \mathfrak{E}(O_2)$ by $\mathfrak{E}(\alpha)_O = \alpha_O \restriction_{\mathfrak{E}(O_1)}$,
- $\mathfrak{E}(\alpha)_I\colon \mathfrak{E}(I_1) \to \mathfrak{E}(I_2)$ by $\mathfrak{E}(\alpha)_I = \alpha_I \restriction_{\mathfrak{E}(I_1)}$,
- $\mathfrak{E}(\alpha)_P\colon \mathfrak{E}(P_1) \to \mathfrak{E}(P_2)$ by $\mathfrak{E}(\alpha)_P = \alpha_P \restriction_{\mathfrak{E}(P_1)}$, and
- $\mathfrak{E}(\alpha)_\Sigma\colon \mathfrak{E}(\Sigma_1) \to \mathfrak{E}(\Sigma_2)$ by $\mathfrak{E}(\alpha)_\Sigma = \alpha_\Sigma \restriction_{\mathfrak{E}(\Sigma_1)}$.

Remark 2. A glance at Definition 12 reveals that a computon $\mathfrak{E}(\lambda)\colon \mathbf{Comp} \to \mathbf{Set}$ is a subfunctor of its source computon $\lambda\colon \mathbf{Comp} \to \mathbf{Set}$. That is, there exists a natural transformation $\mathfrak{E}(\lambda) \to \lambda$ whose components are monic.

Although the structure of any computon can be expressed as another in terms of computation units and control flow/ports only, connectivity is not always preserved by the construction presented in Definition 12. As an example, consider the computon λ displayed in Figure 7(a), which clearly is connected because there is a sequence of information flows from each ec-inport to the only ec-outport. Figure 7(b) shows the resulting object $\mathfrak{E}(\lambda)$ which, although valid, does not satisfy Definition 6. This follows from the fact that the upper ec-inport does not have any sequence of information flows that can take it to the only ec-outport.[11]



(a) A connected computon λ.

(b) Computon $\mathfrak{E}(\lambda)$ which, evidently, is not connected.

Figure 7: An example to show that the endofunctor $\mathfrak{E}$ does not preserve connectivity in general. Like in Figure 5, for now we just display computation units as labels. In upcoming sections, we will use specific syntax to distinguish among different types of units. We use opacity to illustrate the mapping done by $\mathfrak{E}$ and, thereby, highlighting that the data elements of λ are "forgotten" in $\mathfrak{E}(\lambda)$.

Despite not preserving connectivity, $\mathfrak{E}$ retains all control ports and computation units, together with their flow adjacency and port colouring. More generally speaking, such endofunctor is functorial because it preserves the structure of $\mathbf{Set}^{\mathbf{Comp}}$, including composition and identities. Checking functoriality can be trivially done in a similar manner as the proof of Proposition 12. We just need to check that the objects and morphisms in the image of $\mathfrak{E}$ are indeed computons and computon morphisms in the sense of Definitions 1 and 7. For this, we have Propositions 13 and 14.

Proposition 13. If λ is a computon, then $\mathfrak{E}(\lambda)$ is also a computon.

Proof. By Definitions 1 and 12, it is clear that $\mathfrak{E}(\lambda)$ must have a (possibly empty) set $\mathfrak{E}(U)$ of computation units, a non-empty set $\mathfrak{E}(P)$ of ports, a possibly empty set $\mathfrak{E}(O)$ of outgoing edges, a possibly empty set $\mathfrak{E}(I)$ of incoming edges and a non-empty set $\mathfrak{E}(\Sigma) = \{0\} \subset \mathbb{N}$ of colours. As $\mathfrak{E}(s) = s \restriction_{\mathfrak{E}(I)}$ and s is total (by Definitions 12 and 1), $\mathfrak{E}(s)$ is a total function. Analogously, $\mathfrak{E}(t)$, $\mathfrak{E}(\sigma)$ and $\mathfrak{E}(\tau)$ are total too.

To show $\mathfrak{E}(\tau)$ is surjective in addition, simply use Definition 12 and the fiber of $c \circ s$ over 0 to derive $\mathfrak{E}(\tau) = \tau \restriction_{\mathfrak{E}(I)} = \tau \restriction_{\{i \in I \mid c(s(i))=0\}} = \tau \restriction_{(c \circ s)^{-1}(0)}$. As $\tau \restriction_{(c \circ s)^{-1}(0)}$ is necessarily surjective according to Definition 1, it follows that $\mathfrak{E}(\tau)$ also is. An analogous approach can prove surjectivity for $\mathfrak{E}(\sigma)$. Since $\mathfrak{E}(c)(p) = 0$ for all $p \in \mathfrak{E}(P)$, it is obvious $\mathfrak{E}(c)$ is total surjective.

Now, if we consider an arbitrary unit $u \in \mathfrak{E}(U)$, there must be some $i \in \mathfrak{E}(I)$ because $\mathfrak{E}(\tau)$ is onto, i.e., $i \in \{j \in I \mid c(s(j)) = 0\}$ as per Definition 12. By noticing $s \restriction_{\mathfrak{E}(I)}$ and $c \restriction_{\mathfrak{E}(P)}$ respectively map input flows to control ports and control ports to 0, we deduce $c \restriction_{\mathfrak{E}(P)} \circ s \restriction_{\mathfrak{E}(I)}$ must map input flows to 0. In other words, $i \in (c \restriction_{\mathfrak{E}(P)} \circ s \restriction_{\mathfrak{E}(I)})^{-1}(0)$ or, in line with Definition 12, $i \in (\mathfrak{E}(c) \circ \mathfrak{E}(s))^{-1}(0)$. Hence, $\mathfrak{E}(\tau) \restriction_{(\mathfrak{E}(c) \circ \mathfrak{E}(s))^{-1}(0)}$ is a surjective function.

Finally, to show $\mathfrak{E}(\lambda)$ has at least one ec-inport and at least one ec-outport, let $q \in P^+$ with $c(q) = 0$ (i.e., $q \in \mathfrak{E}(P)$) and consider $q \in P^+ \implies q \notin Im(t) \implies q \notin Im(\mathfrak{E}(t)) \implies q \in \mathfrak{E}(P)^+$. Since $q \in \mathfrak{E}(P)^+$ and $\mathfrak{E}(c)(q) = c(q) = 0$, q must be an ec-inport of $\mathfrak{E}(\lambda)$. An analogous reasoning can additionally show that $\mathfrak{E}(\lambda)$ has at least one ec-outport.

Having satisfied all the conditions from Definition 1, we conclude $\mathfrak{E}(\lambda)$ is a well-formed computon. □

[11] From the example presented in Figure 7, we can observe that the endofunctor $\mathfrak{E}$ could be useful to detect potential non-termination. Our concept of termination is formalised in Section 3.

Proposition 14. If $\alpha\colon \lambda_1 \to \lambda_2$ is a computon morphism, then $\mathfrak{E}(\alpha)\colon \mathfrak{E}(\lambda_1) \to \mathfrak{E}(\lambda_2)$ is a computon morphism.

Proof. Assuming $\alpha : \lambda_1 \to \lambda_2$ is a computon morphism, we first check $\vec{i}(\mathfrak{E}(\alpha)) \cup \vec{o}(\mathfrak{E}(\alpha)) \subseteq \mathfrak{E}(P_1)^+ \cup \mathfrak{E}(P_1)^-$. For this, let $p \in \vec{i}(\mathfrak{E}(\alpha)) \cup \vec{o}(\mathfrak{E}(\alpha))$ so that $\bullet\mathfrak{E}(\alpha)(p) \setminus \mathfrak{E}(\alpha)(\bullet p) \neq \emptyset$ or $\mathfrak{E}(\alpha)(p) \bullet \setminus \mathfrak{E}(\alpha)(p\bullet) \neq \emptyset$. By Definition 12, we have $\mathfrak{E}(\alpha)(p) = \alpha(p)$ because $p \in \mathfrak{E}(P_1)$. So, $\bullet\alpha(p) \setminus \alpha(\bullet p) \neq \emptyset$ or $\alpha(p) \bullet \setminus \alpha(p\bullet) \neq \emptyset$. Using Definitions 7 and 2, $p \in \vec{i}(\alpha) \cup \vec{o}(\alpha) \implies p \in P_1^+ \cup P_1^-$ so $p \notin Im(t_1)$ or $p \notin Im(s_1)$. As $\mathfrak{E}(t_1) = t_1 \restriction_{\mathfrak{E}(O_1)}$ and $\mathfrak{E}(s_1) = s_1 \restriction_{\mathfrak{E}(I_1)}$, $p \notin Im(\mathfrak{E}(t_1))$ or $p \notin Im(\mathfrak{E}(s_1))$. Hence, $p \in \mathfrak{E}(P_1)^+ \cup \mathfrak{E}(P_1)^-$.

Now, notice α_U, α_P, α_O and α_I are necessarily total functions by Definition 7. As $\mathfrak{E}(\alpha)_U = \alpha_U$, $\mathfrak{E}(\alpha)_P = \alpha_P \restriction_{\mathfrak{E}(P_1)}$, $\mathfrak{E}(\alpha)_O = \alpha_O \restriction_{\mathfrak{E}(O_1)}$ and $\mathfrak{E}(\alpha)_I = \alpha_I \restriction_{\mathfrak{E}(I_1)}$, we use the definition of function restriction to deduce $\mathfrak{E}(\alpha)_U$, $\mathfrak{E}(\alpha)_P$, $\mathfrak{E}(\alpha)_O$ and $\mathfrak{E}(\alpha)_I$ must be total too. As $\mathfrak{E}(\alpha)_\Sigma$ is necessarily a function $\{0\} \to \{0\}$ given by $0 \mapsto 0$ (see Definition 12), $\mathfrak{E}(\alpha)_\Sigma$ is trivially an inclusion. Therefore, we conclude $\mathfrak{E}(\alpha)$ satisfies all the conditions from Definition 7, i.e., $\mathfrak{E}(\alpha)$ is a computon morphism. □

Unfortunately, the data flow structure of a computon cannot be fully embodied by another computon, since Definition 1 enforces objects in $\mathbf{Set}^{\mathbf{Comp}}$ to always have control ports. Nevertheless, such a structure can still be presented as a Data Flow Graph (DFG) which is just a directed labelled graph obtained via the application of the functor from Definition 13. Again, as computation units are not labelled within computons, all of them take the constant label κ in the corresponding DFG.

Definition 13 (Computon DFG)**.** The functor $\mathscr{D}\colon \mathbf{Set}^{\mathbf{Comp}} \to \mathbf{Set}^{\mathbf{Gr}}$ maps a computon λ to its underlying DFG $\mathscr{D}(\lambda)$ as follows:

- The set V of vertices of $\mathscr{D}(\lambda)$ is given by $U \cup \{p \in P \mid c(p) > 0\}$.
- The set E of edges of $\mathscr{D}(\lambda)$ is given by $\{i \in I \mid c(s(i)) > 0\} \cup \{o \in O \mid c(t(o)) > 0\}$.
- The set L of labels of $\mathscr{D}(\lambda)$ is given by $\Sigma \cup \{\kappa\}$.
- The source function $\vec{s}\colon E \to V$ of $\mathscr{D}(\lambda)$ is given by $\vec{s}(e) = \begin{cases} s(e) & \text{if } e \in I \\ \sigma(e) & \text{if } e \in O \end{cases}$
- The target function $\vec{t}\colon E \to V$ of $\mathscr{D}(\lambda)$ is given by $\vec{t}(e) = \begin{cases} t(e) & \text{if } e \in O \\ \tau(e) & \text{if } e \in I \end{cases}$
- The labelling function $l\colon V \to L$ of $\mathscr{D}(\lambda)$ is given by $l(v) = \begin{cases} c(v) & \text{if } v \in P \\ \kappa & \text{if } v \in U \end{cases}$

For a computon morphism $\alpha\colon \lambda_1 \to \lambda_2$, we define the components of the corresponding graph homomorphism $\mathscr{D}(\alpha)\colon \mathscr{D}(\lambda_1) \to \mathscr{D}(\lambda_2)$ as follows:

- $\mathscr{D}(\alpha)_V\colon V_1 \to V_2$ by $\mathscr{D}(\alpha)_V(v) = \begin{cases} \alpha_U(v) & \text{if } v \in U_1 \\ \alpha_P(v) & \text{if } v \in P_1 \end{cases}$
- $\mathscr{D}(\alpha)_E\colon E_1 \to E_2$ by $\mathscr{D}(\alpha)_E(e) = \begin{cases} \alpha_I(e) & \text{if } e \in I_1 \\ \alpha_O(e) & \text{if } e \in O_1 \end{cases}$
- $\mathscr{D}(\alpha)_L\colon L_1 \to L_2$ by $\mathscr{D}(\alpha)_L(x) = \begin{cases} \alpha_\Sigma(x) & \text{if } x \in \Sigma_1 \\ \kappa & \text{otherwise} \end{cases}$

Definition 13 indicates that a computon DFG $\mathscr{D}(\lambda)$ is constructed in a similar fashion as its counterpart. The only difference is that, rather than considering control elements, $\mathscr{D}(\lambda)$ considers data ports, data flows and computation units only. That is, Definitions 11 and 13 just differ in how vertices, edges and labels are constructed. By Proposition 15, Definition 13 also yields a functorial construction.

Proposition 15. The process of building a computon DFG is functorial, i.e., $\mathscr{D}$ is a functor.

Proof. Definitions 11 and 13 define the labelling, source and target functions in the same way. Similarly, the components of a graph homomorphism are defined analogously. Therefore, the proof of this proposition is similar to that of Proposition 12. □

3. Operational Semantics

The operational behaviour of a computon can be described via *token game semantics* because the structure of a computon can be expressed as a classical P/T Petri net (S, T, in, out) where S is a set of places, T is a set of transitions and $in, out : T \to S^{\oplus}$ are functions assigning to each transition its corresponding pre- and post-set.[12] Here, $S^{\oplus}$ denotes the free commutative monoid generated by S. As each element of $S^{\oplus}$ is a formal finite linear combination with natural number coefficients, $\oplus$ denotes addition of linear combinations. Given this notion, Definition 14 formalises the category of Petri nets.

Definition 14 (Category of Petri Nets)**. Petri** is the category of Petri nets, where each morphism $f : (S_1, T_1, in_1, out_1) \to (S_2, T_2, in_2, out_2)$ consists of a total function $f_T : T_1 \to T_2$ and a monoid homomorphism $f^{\oplus} : S_1^{\oplus} \to S_2^{\oplus}$ that make the following diagram commute:

$$\begin{array}{ccc} T_1 & \overset{in_1}{\underset{out_1}{\rightrightarrows}} & S_1^{\oplus} \\ {\scriptstyle f_T}\downarrow & & \downarrow{\scriptstyle f^{\oplus}} \\ T_2 & \overset{in_2}{\underset{out_2}{\rightrightarrows}} & S_2^{\oplus} \end{array}$$

As it is a sound category, **Petri** satisfies the identity law and is closed under associative composition (which is defined component-wise) [29]. For monoidal structure preservation, $f^{\oplus}$ leaves the identity fixed while respecting addition of linear combinations. This homomorphism, which is the unique extension of a function $f : S_1 \to S_2$, is given by $f^{\oplus}(\bigoplus_{s \in S_1} n_s s) = \bigoplus_{s \in S_1} n_s f(s)$ where $n_s \in \mathbb{N}$ denotes the multiplicity of a place $s \in S_1$. In other words, $f^{\oplus}$ respects the combinatorial structure of S_1 in S_2.

Using Definition 14, we now specify a functor to map each computon to its underlying Petri net, i.e., we provide operational semantics for computons in the theory of classical Petri nets.

Definition 15 (Computons as Petri Nets)**.** The functor $\mathcal{N} : \mathbf{Set}^{\mathbf{Comp}} \to \mathbf{Petri}$ sends any computon λ to its underlying Petri net $\mathcal{N}(\lambda)$ such that:

- $\mathcal{N}(\lambda)$ has the set P of ports as its set of places.
- $\mathcal{N}(\lambda)$ has the set U of computation units as its set of transitions.
- The pre-set function $in : U \to P^{\oplus}$ is given by $in(u) = \bigoplus_{p \in {}^{\bullet}u} 1p$ for all $u \in U$.
- The post-set function $out : U \to P^{\oplus}$ is given by $out(u) = \bigoplus_{p \in u^{\bullet}} 1p$ for all $u \in U$.

[12] In this paper, we use the term classical (P/T) Petri net to refer to a categorical Petri net, which are equivalent in expressive power [28].

On mappings, $\mathcal{N}$ takes a computon morphism $\alpha : \lambda_1 \to \lambda_2$ to a net morphism $\mathcal{N}(\alpha) : \mathcal{N}(\lambda_1) \to \mathcal{N}(\lambda_2)$ as follows:

- $\mathcal{N}(\alpha)_T : U_1 \to U_2$ is a function given by $\mathcal{N}(\alpha)_T(u) = \alpha_U(u)$ for all $u \in U_1$, and
- $\mathcal{N}(\alpha)^{\oplus} : P_1^{\oplus} \to P_2^{\oplus}$ is a monoid homomorphism which leaves the identity fixed and respects the monoid operation $\oplus$ such that, for every combination $p_1 \oplus \cdots \oplus p_n$ given by $P_1^{\oplus}$, we have $\mathcal{N}(\alpha)^{\oplus}(p_1 \oplus \cdots \oplus p_n) = \alpha_P(p_1) \oplus \cdots \oplus \alpha_P(p_n)$.

Here, for $j = 1, 2$ we clearly abuse notation by the use of U_j and $P_j^{\oplus}$ to respectively denote the set of transitions of $\mathcal{N}(\lambda_j)$ and the free commutative monoid on $\mathcal{N}(\lambda_j)$-places.

A glance at Definition 15 reveals that the functor $\mathcal{N}$ preserves the structure of the category of computons in **Petri**, including composition of computon morphisms, the identity law and associativity of composition. The functoriality of $\mathcal{N}$ can be trivially checked by noticing that a net morphism $\mathcal{N}(\alpha)$ is completely built upon the U- and P-components of a computon morphism α, and that a Petri net $\mathcal{N}(\lambda)$ includes all the ports (i.e., places) and computation units (i.e., transitions) from λ. Particularly, the pre- and post-set functions of $\mathcal{N}(\lambda)$ only consider unitary coefficients with no repeated places. So, there is a one-to-one correspondence between the input places of a transition u and the ports in $\bullet u$, and between the output places of u and the ports in $u\bullet$ (see Definition 5). That is, the pre- and post-set of u can directly be treated as sets of places rather than bags. This formalisation can be done without any consequences since multiplicity of inputs/outputs is explicitly specified in a computon, as a result of operating directly on individual port elements and individual edges. Controlling multiplicity outside Petri nets is an example of how composition semantics can dictate operational aspects. Although we decide to use Petri nets because they fit naturally with the structure of computons, other formalisms can be used to specify operational semantics for computons (e.g., timed Petri nets [22]).[13] This flexibility is due to the separation of structure/composition (in computons) from operation (in P/T Petri nets in our case).

Although the categorical structure of $\mathbf{Set}^{\mathbf{Comp}}$ is totally preserved in **Petri**, the structure of individual objects (computons) is not fully preserved because port colouring is not taken into account. Despite of forgetting port colouring, control and data flow is implicit in the underlying net of any computon. To explicitly present control flow as a net, the proof of Proposition 16 says we can construct a functor from $\mathcal{N}$ that does not operate on the whole category of computons but on the image of the endofunctor $\mathfrak{E}$ from Definition 12.

Proposition 16. There is a composite functor $\mathcal{C} \circ \mathfrak{E} : \mathbf{Set}^{\mathbf{Comp}} \to \mathbf{Petri}$ to present the control flow structure of any computon as a Petri net.

Proof. The image $\mathfrak{E}(\mathbf{Set}^{\mathbf{Comp}})$ of the category $\mathbf{Set}^{\mathbf{Comp}}$ under the endofunctor $\mathfrak{E}$ is trivially a full subcategory of $\mathbf{Set}^{\mathbf{Comp}}$ (see Definition 12). Then, there is an obvious functor $\mathcal{C} : \mathfrak{E}(\mathbf{Set}^{\mathbf{Comp}}) \to \mathbf{Petri}$ given by the restriction of $\mathcal{N}$ to $\mathfrak{E}(\mathbf{Set}^{\mathbf{Comp}})$. That is, $\mathcal{C}$ uses the mapping from Definition 15 to send the control flow structure $\mathfrak{E}(\lambda)$ of a computon λ to a Petri net $\mathcal{C}(\mathfrak{E}(\lambda))$ and a computon morphism $\mathfrak{E}(\lambda_1) \to \mathfrak{E}(\lambda_2)$ to a net morphism $\mathcal{C}(\mathfrak{E}(\lambda_1)) \to \mathcal{C}(\mathfrak{E}(\lambda_2))$.

Evidently, the composite functor $\mathcal{C} \circ \mathfrak{E} : \mathbf{Set}^{\mathbf{Comp}} \to \mathbf{Petri}$ is well-defined because the image of $\mathfrak{E}$ is the domain of $\mathcal{C}$. □

Unfortunately, presenting the data flow structure of a computon as a net cannot be done as elegantly as we did for Proposition 16, since computons always require control ports and, therefore, there is no way of solely representing data flow in the form of a computon. Despite of this, Proposition 17 shows that data flow can indeed be presented as a net through a functorial construction akin to Definition 15.

[13] A same computon can behave differently depending on the chosen operational semantics.

Proposition 17. There is a functor $\mathcal{D} : \mathbf{Set}^{\mathbf{Comp}} \to \mathbf{Petri}$ to present the data flow structure of any computon as a Petri net.

Proof. Proving this proposition requires the construction of a functor $\mathcal{D} : \mathbf{Set}^{\mathbf{Comp}} \to \mathbf{Petri}$ which, given a computon λ, defines a Petri net $\mathcal{D}(\lambda)$ as follows:

- $\mathcal{D}(\lambda)$ has $\{p \in P \mid c(p) > 0\}$ as its set S of places, i.e., $S^{\oplus}$ is the free commutative monoid on the set of data ports of λ.
- $\mathcal{D}(\lambda)$ has the set U of computation units as its set of transitions.
- The pre-set function $in : U \to S^{\oplus}$ is given by $in(u) = \bigoplus_{p \in {}^\bullet u \cap S} 1p$ for all $u \in U$.
- The post-set function $out : U \to S^{\oplus}$ is given by $out(u) = \bigoplus_{p \in u^\bullet \cap S} 1p$ for all $u \in U$.

On mappings, the functor $\mathcal{D}$ takes each computon morphism $\alpha : \lambda_1 \to \lambda_2$ to a net morphism $\mathcal{D}(\alpha) : \mathcal{D}(\lambda_1) \to \mathcal{D}(\lambda_2)$ as follows:

- $\mathcal{D}(\alpha)_T : U_1 \to U_2$ is a function given by $\mathcal{D}(\alpha)_T(u) = \alpha_U(u)$ for all $u \in U_1$, and
- $\mathcal{D}(\alpha)^{\oplus} : S_1^{\oplus} \to S_2^{\oplus}$ is a monoid homomorphism which leaves the identity fixed and respects the monoid operation $\oplus$ such that, for every combination $p_1 \oplus \cdots \oplus p_n$ given by $S_1^{\oplus}$, we have $\mathcal{D}(\alpha)^{\oplus}(p_1 \oplus \cdots \oplus p_n) = \alpha_P(p_1) \oplus \cdots \oplus \alpha_P(p_n)$.

Here, for $j = 1, 2$ we clearly abuse notation by the use of U_j and $S_j^{\oplus}$ to respectively denote the set of transitions of $\mathcal{D}(\lambda_j)$ and the free commutative monoid on $\mathcal{D}(\lambda_j)$-places.

Consider the net morphism $\mathcal{D}(\alpha) : \mathcal{D}(\lambda_1) \to \mathcal{D}(\lambda_2)$ and without loss of generality assume $u_1 \in U_1$ is a transition of $\mathcal{D}(\lambda_1)$ where $in_1(u_1) = p_1 \oplus \cdots \oplus p_n$. As each $p_j \in {}^\bullet u_1 \cap S_1$ has a unitary coefficient and occurs only once, we can treat $p_1 \oplus \cdots \oplus p_n$ and the set $\{p_1, \ldots, p_n\}$ interchangeably. Leveraging $\{p_1, \ldots, p_n\} \subseteq {}^\bullet u_1$, we use the commutativity property of α to deduce $\alpha_U(u_1) \in U_2$ and $\{\alpha_P(p_1), \ldots, \alpha_P(p_n)\} \subseteq {}^\bullet \alpha_U(u_1)$. With this in mind, we now prove $\mathcal{D}(\alpha)^{\oplus} \circ in_1 = in_2 \circ \mathcal{D}(\alpha)_T$:

$$\begin{aligned}
\mathcal{D}(\alpha)^{\oplus}(in_1(u_1)) &= \mathcal{D}(\alpha)^{\oplus}(p_1 \oplus \cdots \oplus p_n) && \text{since } in_1(u_1) = p_1 \oplus \cdots \oplus p_n \\
&= \alpha_P(p_1) \oplus \cdots \oplus \alpha_P(p_n) && \text{by the definition of } \mathcal{D}(\alpha)^{\oplus} \\
&= \{\alpha_P(p_1), \ldots, \alpha_P(p_n)\} && \text{treating the combination as a set (see above)} \\
&\subseteq {}^\bullet \alpha_U(u_1) && \text{by the commutativity of } \alpha \text{ (see above)}
\end{aligned}$$

Having $\{\alpha_P(p_1), \ldots, \alpha_P(p_n)\} \subseteq {}^\bullet \alpha_U(u_1)$ allows us to use Definition 5 to deduce $\alpha_P(p_j) \in P_2$ and $(\exists i \in I_2)[\alpha_P(p_j) \xrightarrow{i} \alpha_U(u_1)]$ for all $j = 1, \ldots, n$. Since $c_1(p_j) > 0$ because $p_j \in S_1$, the colour preservation of α says $c_2(\alpha_P(p_j)) > 0$ for all $j = 1, \ldots, n$, i.e., $\{\alpha_P(p_1), \ldots, \alpha_P(p_n)\} \subseteq S_2$. Hence:

$$\begin{aligned}
\{\alpha_P(p_1), \ldots, \alpha_P(p_n)\} &= \alpha_P(p_1) \oplus \cdots \oplus \alpha_P(p_n) && \text{treating the set as a combination} \\
&= in_2(\alpha_U(u_1)) && \text{because } \{\alpha_P(p_1), \ldots, \alpha_P(p_n)\} \subseteq {}^\bullet \alpha_U(u_1) \cap S_2 \\
&= in_2(\mathcal{D}(\alpha)_T(u_1)) && \text{by the definition of } \mathcal{D}(\alpha)_T
\end{aligned}$$

Thus, showing $\mathcal{D}(\alpha)_S^{\oplus} \circ in_1 = in_2 \circ \mathcal{D}(\alpha)_T$ in general. Since the proof of $\mathcal{D}(\alpha)_S^{\oplus} \circ out_1 = out_2 \circ \mathcal{D}(\alpha)_T$ is completely analogous, we conclude that the functor $\mathcal{D}$ preserves transitions (i.e., computation units) and place adjacency (i.e., data port adjacency). To check $\mathcal{D}$ also preserves composition, it suffices to observe that the T-component of a net morphism $\mathcal{D}(\alpha)$ corresponds to α_U and that the other component is entirely built upon α_P. Therefore, $\mathcal{D}(\alpha_2 \circ \alpha_1) = \mathcal{D}(\alpha_2) \circ \mathcal{D}(\alpha_1)$ for any pair of computon morphisms $\alpha_1 : \lambda_1 \to \lambda_2$ and $\alpha_2 : \lambda_2 \to \lambda_3$. As $1_{\mathcal{D}(\lambda_2)} \circ \mathcal{D}(\alpha) = \mathcal{D}(\alpha) = \mathcal{D}(\alpha) \circ 1_{\mathcal{D}(\lambda_1)}$ holds component-wise, the identity law also follows trivially.

For transitions, associativity of composition holds because associativity is satisfied in $\mathbf{Set}^{\mathbf{Comp}}$ and the T-component of a net morphism corresponds to the U-component of a computon morphism. We just need to verify associativity for the corresponding monoid homomorphism by considering the net morphisms $\mathcal{D}(\alpha_1) : \mathcal{D}(\lambda_1) \to \mathcal{D}(\lambda_2)$, $\mathcal{D}(\alpha_2) : \mathcal{D}(\lambda_2) \to \mathcal{D}(\lambda_3)$ and $\mathcal{D}(\alpha_3) : \mathcal{D}(\lambda_2) \to \mathcal{D}(\lambda_3)$. Assuming $q_1 \oplus \cdots \oplus q_m$ is a linear combination from $S_1^{\oplus}$, we obtain:

$$
\begin{aligned}
&\mathcal{D}(\alpha_3)^{\oplus}[(\mathcal{D}(\alpha_2)^{\oplus} \circ \mathcal{D}(\alpha_1)^{\oplus})(q_1 \oplus \cdots \oplus q_m)] \\
=&\mathcal{D}(\alpha_3)^{\oplus}[(\mathcal{D}(\alpha_2 \circ \alpha_1)^{\oplus}(q_1 \oplus \cdots \oplus q_m)] && \text{by composition preservation} \\
=&\mathcal{D}(\alpha_3)^{\oplus}[(\alpha_2 \circ \alpha_1)(q_1) \oplus \cdots \oplus (\alpha_2 \circ \alpha_1)(q_m)] && \text{by the definition of } \mathcal{D}(\alpha_2 \circ \alpha_1)^{\oplus} \\
=&\alpha_3[(\alpha_2 \circ \alpha_1)(q_1)] \oplus \cdots \oplus \alpha_3[(\alpha_2 \circ \alpha_1)(q_m)] && \text{by the definition of } \mathcal{D}(\alpha_3)^{\oplus} \\
=&(\alpha_3 \circ \alpha_2)[\alpha_1(q_1)] \oplus \cdots \oplus (\alpha_3 \circ \alpha_2)[\alpha_1(q_m)] && \text{by associativity of computon morphisms} \\
=&\mathcal{D}(\alpha_3 \circ \alpha_2)^{\oplus}[\alpha_1(q_1) \oplus \cdots \oplus \alpha_1(q_m)] && \text{by the definition of } \mathcal{D}(\alpha_3 \circ \alpha_2)^{\oplus} \\
=&\mathcal{D}(\alpha_3 \circ \alpha_2)^{\oplus}[\mathcal{D}(\alpha_1)^{\oplus}(q_1 \oplus \cdots \oplus q_m)] && \text{by the definition of } \mathcal{D}(\alpha_1)^{\oplus} \\
=&[\mathcal{D}(\alpha_3)^{\oplus} \circ \mathcal{D}^{\oplus}(\alpha_2)][\mathcal{D}(\alpha_1)^{\oplus}(q_1 \oplus \cdots \oplus q_m)] && \text{by composition preservation}
\end{aligned}
$$

Thus, showing $\mathcal{D}(\alpha_3)^{\oplus} \circ [\mathcal{D}(\alpha_2)^{\oplus} \circ \mathcal{D}(\alpha_1)^{\oplus}] = [\mathcal{D}(\alpha_3)^{\oplus} \circ \mathcal{D}^{\oplus}(\alpha_2)] \circ \mathcal{D}(\alpha_1)^{\oplus}$ and, hence, demonstrating that the construction given by $\mathcal{D}$ is indeed functorial. That is, $\mathcal{D}$ preserves the structure of data ports (i.e., places) and computation units (i.e., transitions) together with identities and composition, while satisfying associativity of composition. Thus, we conclude $\mathcal{D}$ is a sound categorical construction to faithfully present the data flow structure of any computon as a Petri net. □

Definition 15 together with Propositions 16 and 17 constitute three alternative ways of studying information flow within a computon in the theory of classical P/T Petri nets. The former allows us to describe both control flow and data flow as a net, whereas the second and third one serve to respectively present control flow and data flow.[14] In any case, a net's behaviour corresponds to the classical token game:

1. A transition u is enabled at some state if and only if each input place of u has at least one token. This rule is a consequence of having a unitary coefficient for each element of $in(u)$ and is applicable to a net under $\mathcal{N}$, $\mathcal{C} \circ \mathfrak{E}$ or $\mathcal{D}$.
2. If u is enabled at some state, then it fires to reach a new state in which u consumes exactly one token from each corresponding input place and produces exactly one token in each corresponding output place. Token consumption and token production are unitary because all the elements of $in(u)$ and $out(u)$ have unitary coefficients. This rule is applicable to a net under $\mathcal{N}$, $\mathcal{C} \circ \mathfrak{E}$ or $\mathcal{D}$.

In the context of Petri nets, a state is just a distribution of tokens over places given by a marking function, as formalised in Definition 16.

Definition 16 (State of a Net)**.** The state of a Petri net (S, T, in, out) at some point in time is a marking function $M\colon S \to \mathbb{N}$ which assigns zero or more tokens to each place. We say M is initial if $M(p) > 0$ for every input place $p \in S$ and $M(q) = 0$ for every non-input place $q \in S$. Dually, M is final when $M(r) > 0$ for every output place $r \in S$ and $M(s) = 0$ for every non-output place $s \in S$. We write M_i and M_f for the initial and final states, respectively.

As the functors $\mathcal{N}$, $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$ map to Petri nets, Definition 16 can be used to determine the state of computon nets at some point in time. Passing from one marking to another constitutes an evolution of states in which control and/or data flow occurs implicitly. This

[14] Although $\mathcal{N}$ defines execution semantics that include data flow, one might prefer to use the composite functor $\mathcal{C} \circ \mathfrak{E}$ to exclusively model execution since it is well known that control flow comprehensively captures computational behaviour.

operational aspect is a consequence of deriving a net from the category of computons which explicitly indicate which ports store control and which ones buffer data. Particularly, for nets under $\mathcal{N}$, if a computation unit u has m control ports and n data ports connected to it, the corresponding transition will have m input places to buffer incoming control and n input places to store incoming data. Consequently, by Rule 1, u will fire with m "control tokens" and n "data tokens". Assuming u has j control ports and k data ports connected from it, Rule 2 says u must produce j "control tokens" and k "data tokens" upon firing. We use quotation marks to indicate token colouring is implicitly defined in a computon's net. [15]

Apart from controlling multiplicity of places and token colouring, separating composition from operation allows us to dictate crucial execution aspects by construction. For instance, connected computons provide their underlying nets with enhanced coverability for better reachability analysis. Evidently, by Propositions 9 and 11, this advantage extends to pushouts and coproducts, respectively.

As there is a path from every input place to some output place in the net of a connected computon (under $\mathcal{N}$ or $\mathcal{C} \circ \mathfrak{E}$), a state marking all the input places of such a net has an option to reach a state marking all the corresponding output places (in which no further transitions can fire). Although this provides connected computons with a weak potential to complete, there is no guarantee all execution paths will lead to successful completion. For stronger termination guarantees, we need to verify additional properties such as deadlock-freedom. In Section 6, we demonstrate how this additional operational aspect can be statically enforced by our composition operators, i.e., from the (structural) composition dimension and without any domain knowledge. Our precise notion of deadlock-freedom is formalised in Definition 17.

Definition 17 (Deadlock-Freedom)**.** A Petri net is deadlock-free if, for every marking state M reachable from M_i with $M \neq M_f$, there exists some enabled transition at M.

Remark 3. Recall that the reachability relation on Petri net markings satisfies the reflexive property. So, every marking is trivially reachable from itself by zero transition firings.

Remark 4. By Definition 16, a final state M_f does not enable any transitions. Consequently, if the condition $M \neq M_f$ were removed from Definition 17, the net corresponding to a terminating computon would necessarily exhibit deadlocks. Although M_f would constitute a deadlock state under a classical notion of deadlock-freedom, Definition 17 does not classify it as such since our goal is to prevent unwanted blocking during execution only. Similar notions of deadlock-freedom have been adopted in other works [7, 30].

Remark 5. In Section 6, we are interested in verifying deadlock-freedom only for nets that have an initial and a final state in the sense of Definition 17. As Definition 1 enforces computons to have ec-inports and ec-outports, nets under $\mathcal{N}$ or $\mathcal{C} \circ \mathfrak{E}$ satisfy these two conditions. For $\mathcal{D}$-nets, we only check deadlock-freedom when such conditions are satisfied. This is because some of them do not necessarily have input and output places (to buffer data) by the fact Definition 1 does not require computons to have ed-inports or ed-outports.

Enforcing deadlock-freedom at the level of composition semantics contributes towards satisfying termination-by-construction and, hence, to building large-scale complex systems with predictable behaviour and enhanced reliability. Having a weak option to complete together with deadlock-freedom constitute necessary conditions for termination in some cases, but they are not sufficient in general. Hence, it is important to reasoning about other crucial operational properties such as livelocks and boundedness. Given the compositional nature of the proposed model, such additional assurances can be verified compositionally using existing Petri net tools.

[15] Computon nets do not explicitly colour places or tokens to distinguish between control and data. However, if token/port colouring is explicitly needed, one can define functors from $\mathbf{Set}^{\mathbf{Comp}}$ to the category of coloured Petri nets.

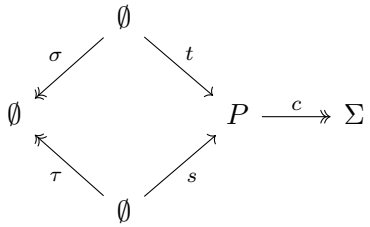
Although this paper is primarily focused on the static (structural) dimension given by composition, this section briefly discussed how operational aspects (i.e., dynamics) can be dictated by construction. As our main focus is composition, we did not discuss algebraic operations on Petri nets (e.g., refinement). Petri nets just serve as a medium to express computon behaviour. Our overarching tenet is that operations for manipulating behaviour must be realised at the composition-level and then propagated to the operational-level through the functors we propose in this section. In Section 6, we further discuss how execution can be dictated by construction. In the future, we plan to study additional advantages derived from the separation of composition from operation such as using formalisms beyond Petri nets to describe and analyse computon behaviour. Using functors on **Petri** to endow computons with execution semantics entails that all the theory of categorical Petri nets [31] is applicable to our work.

In the future, we intend to explore computon processes in more detail, e.g., by studying computon behaviour within symmetric monoidal categories [32].[16] For now, we would just like to highlight that in the *token game for computons*, due to the inherent concurrent nature of Petri nets, "control tokens" can arrive before "data tokens" (or viceversa). By implicitly having "control places" separated from "data places" in a net under $\mathcal{N}$, a transition only fires when both "control" and "data" tokens are placed in their respective input places. This means a transition is a passive construct with blocking behaviour which implicitly synchronises data and control before firing. After firing, it produces exactly one token in each corresponding output place.

4. Trivial Computons

A *trivial computon* has no computation units, no edges and no i-ports at all, but just a number of e-inoutports (see Definition 18). Up to isomorphism, it is the only object in $\mathbf{Set}^{\mathbf{Comp}}$ with no computation units (see Proposition 18); consequently, it is not connected in the sense of Definition 6 (see Proposition 19).

Definition 18 (Trivial Computon)**.** A trivial computon λ is a computon whose diagram in **Set** has the form:



Proposition 18. A computon has no computation units if and only if it is a trivial computon.

Proof. ($\Longrightarrow$) Let λ be a computon with $U = \emptyset$. By the definition of empty function, we have $U = \emptyset$ only if $I = \emptyset = O$ so that σ and τ are surjective. Definition 1 states that any computon is required to have at least one coloured port so $|P| \geq 1$ and $|\Sigma| \geq 1$. As s and t are not surjective by the definition of empty function, we have $p \in P^+ \cap P^-$ for all $p \in P$ (see Definition 2). Particularly, if $c(p) = 0$, then $p \in C^+ \cap C^-$; otherwise, $p \in D^+ \cap D^-$. As the function c is surjective by Definition 1, we conclude that λ is a trivial computon.

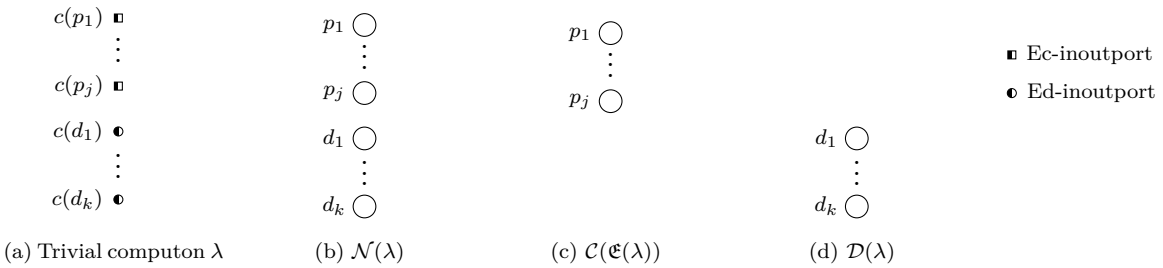
($\Longleftarrow$) This follows directly from Definition 18. □

Proposition 19. A trivial computon is not connected.

Proof. If λ is a trivial computon, then $U = \emptyset$. Therefore, λ is not connected as per Proposition 3. □

[16] The exploitation of symmetric monoidal categories to study concurrent behaviours in Petri nets started long before [32]. Early works include [29], [33] and [34].

The general structure of a trivial computon with j ec-inoutports and k ed-inoutports is depicted in Figure 8, together with its corresponding Petri nets. Definition 19 states that, in our theory, there is a distinguished computon of this sort consisting of a single ec-inoutport, which we refer to as the *unit computon*.



(a) Trivial computon λ (b) $\mathcal{N}(\lambda)$ (c) $\mathcal{C}(\mathfrak{E}(\lambda))$ (d) $\mathcal{D}(\lambda)$

Figure 8: A trivial computon λ and its corresponding Petri nets. By Definition 18, λ does not have any computation units but just e-inoutports; consequently, $\mathcal{N}(\lambda)$, $\mathcal{C}(\mathfrak{E}(\lambda))$ and $\mathcal{D}(\lambda)$ have places only so they are behaviourless. The net $\mathcal{N}(\lambda)$ embodies the whole structure of λ, whereas $\mathcal{C}(\mathfrak{E}(\lambda))$ and $\mathcal{D}(\lambda)$ have places for control and data only, respectively. Labels on places are just for reference purposes since we deal with non-labelled Petri nets, as discussed in Section 3.

Remark 6. By Definition 1, a trivial computon λ always includes control ports (i.e., $j \geq 1$) and can optionally possess data ports (i.e., $k \geq 0$). Therefore, $\mathcal{N}(\lambda)$ and $\mathcal{C}(\mathfrak{E}(\lambda))$ always have at least one place, whereas $\mathcal{D}(\lambda)$ can have no places at all.

Definition 19 (Unit Computon)**.** The *unit computon* is a trivial computon with $|P| = |\Sigma| = 1$. We use Λ to denote it.

The existence of Λ can be proven by the fact that, according to Definition 1, the set of ports and the set of colours are never empty. By the same definition, we can observe that a computon can have no computation units and no edges at all. Proposition 20 uses this observation to show that Λ is unique up to unique isomorphism.

Proposition 20. Λ is unique up to unique isomorphism.

Proof. By letting λ_1 and λ_2 be two unit computons, we construct a computon morphism $\alpha : \lambda_1 \to \lambda_2$. As $U_1 = U_2 = O_1 = O_2 = I_1 = I_2 = \emptyset$, the only choice we have for α_U, α_O and α_I is the empty function. For α_P, Definition 19 forces us to exclusively consider the unique function given by $\alpha_P(p_1) = p_2$ for the unique ports $p_1 \in P_1$ and $p_2 \in P_2$. By this mapping, p_1 can never be in $\vec{i}(\alpha) \cup \vec{o}(\alpha)$ because $\bullet\alpha(p_1) = \bullet p_2 = \alpha(p_1)\bullet = p_2\bullet = \emptyset$ (as per Definition 18) so $\vec{i}(\alpha) \cup \vec{o}(\alpha) = \emptyset \subseteq P_1^+ \cup P_1^-$. The inclusion $\alpha_\Sigma(0) = 0$ is the only option we have for α_Σ, given that $\Sigma_1 = \{0\} = \Sigma_2$ (see Definitions 1 and 19). As this construction satisfies Definition 7, α is indeed a computon morphism. More specifically, α is a computon isomorphism because its inverse is necessarily constructed in the same way, and it is unique because the components of α are unique functions in **Set** (i.e., empty functions or trivial injections).

Given that most of the α-components are empty functions, the only equation that needs to be verified from the commutative squares of Definition 7 is $\alpha_\Sigma \circ c_1 = c_2 \circ \alpha_P$. Having $\alpha_\Sigma(c_1(p_1)) = \alpha_\Sigma(0) = 0 = c_2(p_2) = c_2(\alpha_P(p_1))$, we conclude that the naturality condition of the unique isomorphism α holds. Thus, proving that the unit computon is unique up to unique isomorphism. □

Since a computon is required to have at least one coloured port, Λ can be perceived as the "simplest" object in $\mathbf{Set}^{\mathbf{Comp}}$, which corresponds to ◘ in graphical notation. In spite of this structural feature, Λ is not an initial object in such a category since there are $k \geq 1$ computon morphisms from it to any other computon λ, where k is the number of control ports in λ.[17] Under this premise, as mentioned in Section 2, $\mathbf{Set}^{\mathbf{Comp}}$ has no initial objects.

[17] This can be easily proved by induction on the number of control ports of an arbitrary computon.

In $\mathbf{Set}^{\mathbf{Comp}}$, there are distinguished morphisms that respectively embed a trivial computon into all the e-inports or all the e-outports of some computon. These morphisms are referred to as in- and out-markers, respectively. The intuition behind these notions is captured in Definition 20.

Definition 20 (In- and Out-markers)**.** An *in-marker* of a computon λ is a computon monomorphism $\alpha : \lambda_0 \to \lambda$ where λ_0 is a trivial computon with $\alpha(P_0) = P^+$. If $\alpha(P_0) = P^-$, then α is an *out-marker* of λ.

Notation 3. For convenience, we write λ^+ and λ^- for the respective in- and out-markers of a computon λ. We use the word "the" because all the in-markers of a computon are isomorphic to each other, with the same being true for the corresponding out-markers. When the context is clear, we simply use the word "markers" to unifiedly refer to such morphisms which by, Proposition 21, always exist for any computon.

Proposition 21. Every computon has markers.

Proof. The proof follows directly from the fact that every computon has at least one e-inport and at least one e-outport (see Definition 1). □

As the image of a marker covers all the e-inports or all the e-outports of a computon, it is easy to show that every span of markers is pushable. For convenience, we capture this truth in Proposition 22. Also, as only e-ports are identified in the pushout of a span of markers, it is true that the induced morphisms of the pushout of in- or out-markers preserves e-inports or e-outports, correspondingly (see Proposition 23).

Proposition 22. Every span of markers is pushable.

Proof. The proof follows directly from Definitions 9 and 20. □

Proposition 23. Let $\square \in \{+, -\}$ and $j = 1, 2$. If $(\beta_1 : \lambda_1 \to \lambda_3, \lambda_3, \beta_2 : \lambda_2 \to \lambda_3)$ is the pushout of a span $\lambda_1 \xleftarrow{\lambda_1^\square} \lambda_0 \xrightarrow{\lambda_2^\square} \lambda_2$, then $\beta_j(\lambda_j^\square(P_0)) = P_3^\square$.

Proof. Suppose $(\beta_1 : \lambda_1 \to \lambda_3, \lambda_3, \beta_2 : \lambda_2 \to \lambda_3)$ is the pushout of a span $\lambda_1 \xleftarrow{\lambda_1^+} \lambda_0 \xrightarrow{\lambda_2^+} \lambda_2$ of in-marker morphisms, and assume for contradiction there is some $p_3 \in \beta_j(\lambda_j^+(P_0)) \triangle P_3^+$ for $j \in \{1, 2\}$. If $p_3 \in \beta_j(\lambda_j^+(P_0)) \setminus P_3^+$, then there is some $u_3 \in \bullet p_3$ and, by pushout commutativity, some $u_j \in \bullet p_j$ for $p_j \in \lambda_j^+(P_0)$ such that $\beta_j(u_j) = u_3$ and $\beta_j(p_j) = p_3$. As Definition 20 says $\lambda_j^+(P_0) = P_j^+$, we have $p_j \in \lambda_j^+(P_0) \iff p_j \in P_j^+$, i.e., a contradiction to the assumption $u_j \in \bullet p_j$. The case $p_3 \in P_3^+ \setminus \beta_j(\lambda_j^+(P_0))$ never holds since this would violate the commutativity property of pushout constructions.

Proving the statement $\gamma_j(\lambda_j^-(P_4)) = P_5^-$ for the pushout $(\gamma_1 : \lambda_1 \to \lambda_5, \lambda_5, \gamma_2 : \lambda_2 \to \lambda_5)$ of a span $\lambda_1 \xleftarrow{\lambda_1^-} \lambda_4 \xrightarrow{\lambda_2^-} \lambda_2$ is completely analogous. Therefore, we conclude that our initial proposition is true. □

The existence of marker morphisms gives rise to the notion of dual computons. Informally, the dual of a computon λ is constructed by structurally swapping e-inports with e-outports and vice versa, so the domains of λ^+ and λ^- precisely correspond to the domains of the out- and in-markers of its dual, respectively. This is formalised in Definition 21.

Definition 21 (Dual Computons)**.** If there are spans $\lambda_2 \xleftarrow{\lambda_2^+} \lambda_0 \xrightarrow{\lambda_3^-} \lambda_3$ and $\lambda_2 \xleftarrow{\lambda_2^-} \lambda_1 \xrightarrow{\lambda_3^+} \lambda_3$, we say that λ_2 is a dual of a computon λ_3 and vice versa.

A computon λ can have multiple duals (with different structure each) because Definition 21 only requires a dual to swap the interface of λ, without imposing any constraints on internal structure. More precisely, duals are not unique up to isomorphism, so a dual of a dual is

not necessarily isomorphic to the original computon. Although uniqueness is not satisfied in general, Proposition 24 states that for a connected computon, it is possible to have a dual with inverted information flows. Therefore, when connectivity is guaranteed, the construction given in the corresponding proof can be applied twice to obtain a connected computon that is isomorphic to the original one.

Proposition 24. If λ is a connected computon, there exists a connected computon which is a dual of λ.

Proof. Assuming λ_2 is a connected computon, we construct a computon λ_3 by letting $P_2 = P_3$, $U_2 = U_3$, $I_2 = O_3$, $O_2 = I_3$ and $c_2 = c_3$. For each $o \in O_3$, we set $\sigma_3(o) = \tau_2(o)$ and $t_3(o) = s_2(o)$ to yield $Im(\sigma_3) = Im(\tau_2)$ and $Im(t_3) = Im(s_2)$. Similarly, for each $i \in I_3$, we let $s_3(i) = t_2(i)$ and $\tau_3(i) = \sigma_2(i)$ to have $Im(s_3) = Im(t_2)$ and $Im(\tau_3) = Im(\sigma_2)$. That is, λ_3 has the same coloured ports and computation units as λ_2, but with inverted information flows. So, $p \overset{\exists}{\to} q$ for λ_2 iff $q \overset{\exists}{\to} p$ for λ_3. Moreover, $P_2^+ = P_3^-$ and $P_2^- = P_3^+$ because:

$$p_2 \in P_2^+ \iff p_2 \notin Im(t_2) \iff p_2 \notin Im(s_3) \iff p_2 \in P_3^-$$

$$p_2 \in P_2^- \iff p_2 \notin Im(s_2) \iff p_2 \notin Im(t_3) \iff p_2 \in P_3^+$$

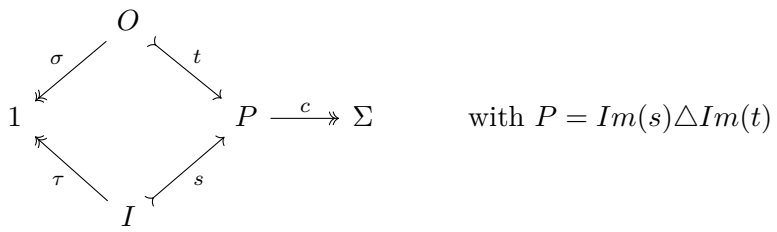
By the above, $Im(s_3) \cup P_3^+ = Im(t_2) \cup P_2^-$ is evident. As λ_2 is connected, we simply use Proposition 4 to deduce that, for each $q_3 \in Im(t_2) \cup P_2^-$, there is a port $p_3 \in P_2^+$ where $p_3 \overset{\exists}{\to} q_3$. Equivalently, for each $q_3 \in Im(s_3) \cup P_3^+$, there is a port $p_3 \in P_3^-$ where $q_3 \overset{\exists}{\to} p_3$. That is, by Definition 6, λ_3 is also connected.

To finalise our proof, we construct a trivial computon λ_0 and a trivial computon λ_1 by letting $P_0 = P_2^+$, $c_0 = c_2 \restriction P_2^+$, $P_1 = P_2^-$ and $c_1 = c_2 \restriction P_2^-$. As $P_0 = P_2^+ = P_3^-$, there evidently is an in-marker $\lambda_0 \to \lambda_2$ and an out-marker $\lambda_0 \to \lambda_3$. Analogously, having $P_1 = P_2^- = P_3^+$ implies the existence of an out-marker $\lambda_1 \to \lambda_2$ and an in-marker $\lambda_1 \to \lambda_3$. By Definition 21, we conclude that λ_3 is a connected computon and a dual of λ_2, as required. □

5. Primitive Computons

Like a trivial computon, a primitive one has no i-ports. The difference is that there is exactly one computation unit to which all ports are attached via edges (see Definition 22). So, every port can be either e-inport or e-outport, never both (see Proposition 25). This implies a primitive computon is connected, i.e., it has neither dangling ports nor dangling computation units (see Proposition 26). For these reasons, the underlying net of any primitive computon is deadlock-free (see Proposition 27 and Remark 8). In this paper, we consider three classes of primitive computons, namely *fork computons*, *join computons* and *functional computons*.

Definition 22. A primitive computon is a computon whose diagram in **Set** has the form:[18]



with $P = Im(s) \triangle Im(t)$

Remark 7. A glance at Definition 22 reveals that a primitive computon has no i-ports because $Im(s) \cap Im(t) = \emptyset$ follows from $P = Im(s) \triangle Im(t)$. Consequently, $Im(s) \setminus Im(t) = Im(s)$ and $Im(t) \setminus Im(s) = Im(t)$ also hold.

[18] We use 1 and $\rightarrowtail$ to denote a singleton set and an injective function, respectively. The symbol $\triangle$ is the operator for symmetric difference given by $A \triangle B = (A \setminus B) \cup (B \setminus A)$ for sets A and B.

Proposition 25. If λ is a primitive computon, then $P = P^+ \triangle P^-$.

Proof. If λ is a primitive computon, then $p \in P \iff p \in Im(s) \triangle Im(t)$ (see Definition 22).

- If $p \in Im(s) \setminus Im(t)$, then $p \in P^+ \setminus P^-$ because $(\exists i \in I)[s(i) = p]$ and $(\nexists o \in O)[t(o) = p]$.
- If $p \in Im(t) \setminus Im(s)$, then $p \in P^- \setminus P^+$ because $(\exists o \in O)[t(o) = p]$ and $(\nexists i \in I)[s(i) = p]$.

By the above cases and by the definition of symmetric difference, we have that $p \in P \iff p \in Im(s) \triangle Im(t) \iff p \in (P^+ \setminus P^-) \cup (P^- \setminus P^+) \iff p \in P^+ \triangle P^-$. Hence, we conclude $P = Im(s) \triangle Im(t) = (P^+ \setminus P^-) \cup (P^- \setminus P^+) = P^+ \triangle P^-$, as required. □

Proposition 26. Every primitive computon is a connected computon.

Proof. Assuming λ is a primitive computon with $p \in Im(s) \cup P^+$, we first show $Im(s) = P^+$ as follows: $p \in Im(s) \iff p \notin P^-$ by Definition 2 $\iff p \in P^+$ by Proposition 25. Having $Im(s) = P^+$ implies we just have to prove for either $p \in Im(s)$ or $p \in P^+$.

If $p \in Im(s)$, we know there must be some $i \in I$ for which $s(i) = p$. If u is the only computation unit in U (see Definition 22), $\tau(i) = u$ because τ is total and surjective (see Definition 1). As σ is also surjective, there exists some $o \in O$ where $\sigma(o) = u$. By the totality of t, there is some $q \in P$ where $t(o) = q$. Having $q \in Im(t)$ and $P = Im(s) \triangle Im(t)$ allows us to deduce $q \notin Im(s)$ so that $q \in P^-$ by Definition 2. As $p \xrightarrow{\exists} q$ holds for $p \in Im(s) \cup P^+$ and $q \in P^-$, we conclude that λ adheres to Definition 6 and it is therefore a connected computon. □

Proposition 27. If λ is a primitive computon, the nets $\mathcal{N}(\lambda)$ and $\mathcal{C}(\mathfrak{E}(\lambda))$ are deadlock-free.

Proof. The proof is obvious since every primitive computon's net has only one transition to which all input places (i.e., e-inports) and output places (i.e., e-outports) are attached (see Definition 22 and Proposition 25). □

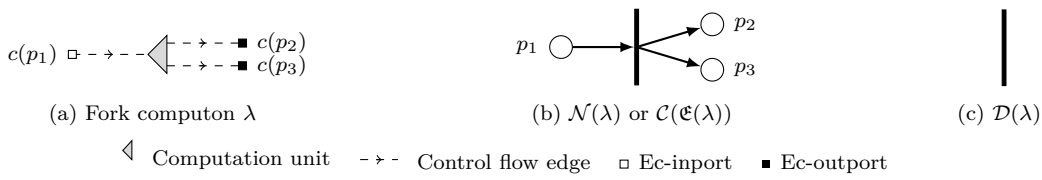
Remark 8. The only scenario in which $\mathcal{D}(\lambda)$ satisfies Remark 5 (i.e., initial and final states exist) is when λ has both ed-inports and ed-outports. When this occurs, the proof that $\mathcal{D}(\lambda)$ is deadlock-free is analogous to that of Proposition 27.

5.1. Fork Computons

A fork computon has exactly one ec-inport and two ec-outports, i.e., it has no data ports at all (see Definition 23). Intuitively, it just duplicates the control received in its unique ec-inport into all its ec-outports.

Definition 23. A fork computon λ is a primitive computon with $|O| = 2$ and $|I| = |\Sigma| = 1$.

From Definition 23, we can deduce a fork computon λ has exactly three ports because s and t are injective, $P = Im(s) \triangle Im(t)$, $|O| = 2$ and $|I| = 1$. Specifically, $|P^-| = 2$ and $|P^+| = 1$ because $P = P^+ \triangle P^-$ (see Definition 2 and Proposition 25). As $|\Sigma| = 1$ and c is total and $C^+ \neq \emptyset \neq C^-$ (see Definition 1), $\Sigma = \{0\}$ so $(\forall p \in P)[c(p) = 0]$. The general structure of a fork computon, together with its underlying Petri nets, is depicted in Figure 9.



(a) Fork computon λ (b) $\mathcal{N}(\lambda)$ or $\mathcal{C}(\mathfrak{E}(\lambda))$ (c) $\mathcal{D}(\lambda)$

Figure 9: A fork computon λ and its corresponding Petri nets. As λ has control ports only, the Petri nets $\mathcal{N}(\lambda)$ and $\mathcal{C}(\mathfrak{E}(\lambda))$ are isomorphic. By the same reason, $\mathcal{D}(\lambda)$ has no places at all but just a single transition. Labels on places are just for reference purposes since we deal with non-labelled Petri nets, as discussed in Section 3.

5.2. Join Computons

A join computon is the dual of a fork computon since it has exactly two ec-inports and one ec-outport (see Definition 24). Intuitively, it merges the control received in its ec-inports into its unique ec-outport.

Definition 24. A join computon λ is a primitive computon with $|O| = |\Sigma| = 1$ and $|I| = 2$.

The properties of a join computon λ are almost identical to that of a fork computon so it is true that $c(p) = 0$ for all $p \in P$. The only difference is in terms of the number of ec-inports and ec-outports. The general structure of a join computon, together with its underlying Petri nets, is depicted in Figure 10.

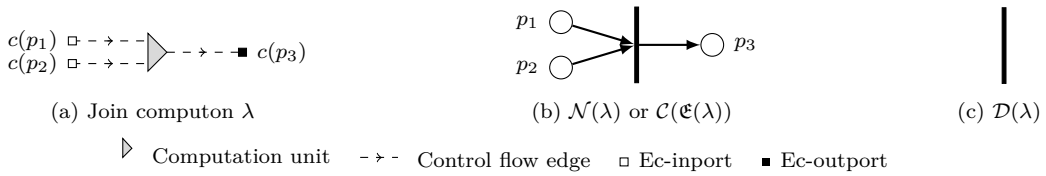


Figure 10: A join computon λ and its corresponding Petri nets. As λ has control ports only, the Petri nets $\mathcal{N}(\lambda)$ and $\mathcal{C}(\mathfrak{E}(\lambda))$ are isomorphic. By the same reason, $\mathcal{D}(\lambda)$ has no places at all but just a single transition. Labels on places are just for reference purposes since we deal with non-labelled Petri nets, as discussed in Section 3.

5.3. Functional Computons

A functional computon has exactly one ec-inport, one ec-outport, any number of ed-inports and any number of ed-outports, as formalised in Definition 25. Intuitively, the unique computation unit is a high-level representation of a (potentially halting) computation, triggered after receiving a control signal and a number of input data values. The successful termination of such a computation results in a single control signal and a number of output data values.

Definition 25. A functional computon λ is a primitive computon where $(\exists! p \in P^+)[c(p) = 0]$ and $(\exists! q \in P^-)[c(q) = 0]$.

The general structure of a functional computon with j ed-inports and k ed-outports is illustrated in Figure 11, together with its corresponding Petri nets.[19]

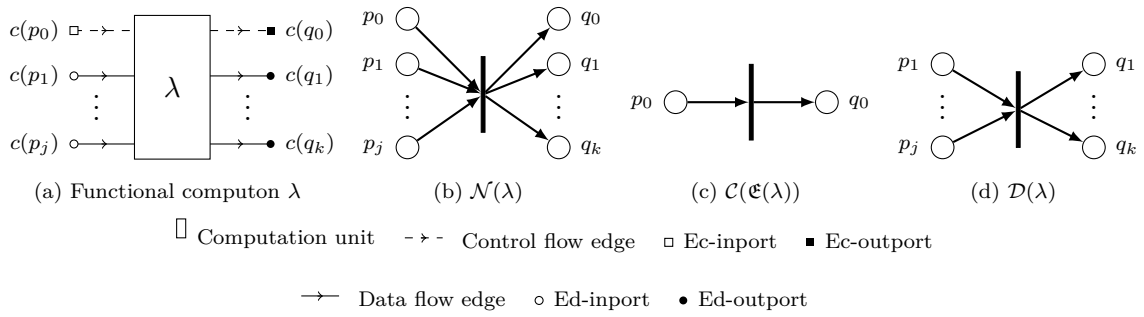


Figure 11: A functional computon λ and its corresponding Petri nets. Evidently, there always is a place to buffer incoming control and another one to buffer outgoing control, in order to respect the structural constraints given by Definition 25. Particularly, the net $\mathcal{N}(\lambda)$ embodies the whole structure of λ, whereas $\mathcal{C}(\mathfrak{E}(\lambda))$ and $\mathcal{D}(\lambda)$ have places for control and data only, respectively. As places to store data are optional as per Definition 1, we have $j, k \geq 0$. Labels on places are just for reference purposes since we deal with non-labelled Petri nets, as discussed in Section 3.

[19] We acknowledge that the definition of computons does not include labels for computation units. However, for increased clarity, we took the liberty of using the symbol of a functional computon for labelling its unique computation unit.

Up to isomorphism, there exists a particular functional computon which possesses only one ec-inport and only one ec-outport. This sort of computon, referred to as *the glue computon*, is explicitly defined when $|I| = |O| = 1$. Intuitively, it just echoes the control signal received in its unique ec-inport into its only ec-outport.

6. Composite Computons

A composite computon is algebraically formed via a composition operator which defines an explicit control flow structure for the execution of computons in some order. More formally, a composite computon is the finite colimit of some diagram in $\mathbf{Set}^{\mathbf{Comp}}$, whose construction is given by a composition operation characterised as a colimit computation. In this section, we describe separate operations to form sequential, parallel, branching and iterative computons. Table 1 summarises the colimit constructions each composition operation builds upon as well as resulting composites. This table also shows the subsection each operation is described in.

Table 1: Relationship between our composition operations and colimits in $\mathbf{Set}^{\mathbf{Comp}}$.

Composition operation	*Built upon*	*Result*	*Subsection*
Total sequencing	Pushout	Total sequential computon	6.1
Partial sequencing	Pushout	Partial sequential computon	6.1
Asynchronous parallel	Coproduct	P-async computon	6.2
Synchronous parallel	Pushout and coproduct	P-sync computon	6.2
Branching	Pushout and coproduct	Branching computon	6.3
Head-iteration	Pushout and coproduct	Head-iterative computon	6.4
Tail-iteration	Pushout and coproduct	Tail-iterative computon	6.4

For each operation, we describe their category-theoretic foundations, operational semantics (in **Petri**) and encapsulation. The latter is a property that allows hiding the internals of composites so as to treat them as self-contained, black-box units. In this section, we will focus on encapsulation of control and data flow and we we will show that, while control flow is explicitly defined, data flow is implicitly sewn through our colimit-based composition operations. Contrary to what might seem obvious, data does not always follow control, especially in the cases of partial sequencing and parallel composition.

Compositionality is a key term in this section, which refers to the property of constructing complex computons from simpler ones through the proposed operators, while ensuring closure and structure preservation of the simpler entities in the complex ones. By closure, we mean an operation from Table 1 remains within $\mathbf{Set}^{\mathbf{Comp}}$, i.e., the result of composing computons, each adhering to Definition 1, is another computon which also conforms to Definition 1. By preservation, we mean ensuring that the computation units, ports, colouring and flow adjacency of the composed entities are retained in the corresponding composite.

To realise preservation, our operators build upon pushout and coproduct which, in turn, rely on computon morphisms to (intuitively) "insert" a computon into another (see Definition 7). That is, these two constructions do not introduce new structures but just embed computons in different ways, as discussed in Section 2. In the case of pushouts, two computons are merged into another via an apex object which represents the common part between the entities being merged. When it comes to coproduct, two computons are put in a side-by-side structure without identifying any elements.

As $\mathbf{Set}^{\mathbf{Comp}}$ has all coproducts (see Proposition 10), it is true that any two computons can always be combined under this colimit. Pushouts only exist for pushable spans (see Proposition 8) so computons λ_1 and λ_2 can only be combined under this construction when there is a computon λ_0 that allows the formation of a pushable span $\lambda_1 \leftarrow \lambda_0 \rightarrow \lambda_2$.

In this section we prove that, although $\mathbf{Set}^{\mathbf{Comp}}$ does not have all pushouts, such a category is closed under the operators we propose since the colimits they define always exist,

i.e., they build upon coproduct, pushouts over pushable spans or any combination thereof. The respective proofs of existence are given for each operator in the corresponding subsection.

6.1. Sequential Computons

Sequential composition is an operation we characterise as a particular pushout in $\mathbf{Set}^{\mathbf{Comp}}$. It is particular because, intuitively, the common object needs to be a trivial computon λ_0 that can be "embedded" into some or all the e-outports of a computon λ_1 and into some or all the e-inports of a computon λ_2. Particularly, every port $p_0 \in P_0$ that is embedded into an e-outport $p_1 \in P_1^-$ needs to be embedded into an e-inport $p_2 \in P_2^+$ and vice versa (see Definition 26). This restriction, given by so-called *sequentiable spans*, enables a strict sequence in which λ_1 is computed before λ_2. A converse computation is possible and requires a different embedding since sequencing is a non-commutative operation in which order matters. The notion of a sequential computon is formalised in Definition 27 and its proof of existence is directly derivable from Lemma 1.

Definition 26 (Sequentiable Spans)**.** A span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms is sequentiable if (i) λ_0 is a trivial computon with $P_0 = \vec{i}(\alpha_1) \cap \vec{o}(\alpha_2)$, (ii) λ_1 and λ_2 are connected computons, (iii) $\alpha_1(\vec{o}(\alpha_2)) \subseteq P_1^-$ and (iv) $\alpha_2(\vec{i}(\alpha_1)) \subseteq P_2^+$. Particularly, if $\alpha_1(\vec{o}(\alpha_2)) = P_1^-$ and $\alpha_2(\vec{i}(\alpha_1)) = P_2^+$, then the span is totally sequentiable. Otherwise, if $\alpha_1(\vec{o}(\alpha_2)) \subset P_1^-$ or $\alpha_2(\vec{i}(\alpha_1)) \subset P_2^+$, the span is partially sequentiable.

Remark 9. If a span ρ is totally sequentiable, then neither $\alpha_1(\vec{o}(\alpha_2)) \subset P_1^-$ nor $\alpha_2(\vec{i}(\alpha_1)) \subset P_2^+$ can hold because equality forbids proper inclusion; conversely, if ρ is partially sequentiable then at least one of the equalities $\alpha_1(\vec{o}(\alpha_2)) = P_1^-$ or $\alpha_2(\vec{i}(\alpha_1)) = P_2^+$ fails. Therefore, totally sequentiable spans are not partially sequentiable (and vice versa), i.e., these two notions are mutually exclusive.

Lemma 1. Every (partially or totally) sequentiable span of computon morphisms is pushable.

Proof. Let $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ be a sequentiable span of computon morphisms. By Definition 26, $p_0 \in P_0 \iff p_0 \in \vec{i}(\alpha_1) \cap \vec{o}(\alpha_2)$ so $\alpha_1(p_0) \in P_1^-$ and $\alpha_2(p_0) \in P_2^+$. That is, the equation $\alpha_1(p_0)\bullet = \emptyset = \bullet\alpha_2(p_0)$ holds. As $p_0\bullet = \emptyset = \bullet p_0$ (because λ_0 is a trivial computon), it follows that $\alpha_1(p_0) \bullet \setminus \alpha_1(p_0\bullet) = \emptyset = \bullet\alpha_2(p_0) \setminus \alpha_2(\bullet p_0)$, meaning $p_0 \notin \vec{o}(\alpha_1) \cup \vec{i}(\alpha_2)$ and, hence, $\vec{o}(\alpha_1) \cup \vec{i}(\alpha_2) = \emptyset$. For $j \in \{1, 2\}$, we observe that $P_j^+ \neq \emptyset \neq P_j^-$ (by Definition 1) and that λ_j is not a trivial computon because it is a connected computon (see Proposition 3). Thus, $\alpha_1(\vec{i}(\alpha_2)) \cup \alpha_1(\vec{o}(\alpha_2)) \subseteq P_1^- \subset P_1^+ \cup P_1^-$ and $\alpha_2(\vec{i}(\alpha_1)) \cup \alpha_2(\vec{o}(\alpha_1)) \subseteq P_2^+ \subset P_2^+ \cup P_2^-$. □

Corollary 1. For a sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms, we have $\vec{o}(\alpha_1) \cup \vec{i}(\alpha_2) = \emptyset$.

Proof. See the proof of Lemma 1. □

Definition 27 (Sequential Computon)**.** Let ρ be a sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms. If ρ is partially sequentiable, then its pushout is called a partial sequential computon, written $\lambda_1 \rhd_\rho \lambda_2$. Otherwise, if ρ is totally sequentiable, then its pushout is called a total sequential computon, written $\lambda_1 \trianglerighteq_\rho \lambda_2$. In any case, λ_0 is called the apex computon, λ_1 the left operand and λ_2 the right operand.

When an apex computon is embedded into all the e-outports of the left operand and into all the e-inports of the right one, we say that the respective span is totally sequentiable; otherwise, we say it is partially sequentiable (see Definition 26). Basically, a sequentiable span chooses the e-outports of the left operand and the e-inports of the right one that will become i-ports in the corresponding sequential composite. Port renomination, which is realised by the induced pushout morphisms, follows Proposition 7. By Proposition 28, total sequentiality implies partial sequentiality in the general case. The only exception occurs when the left and right operands have exactly one e-outport and one e-inport, respectively, in which case total sequentiality is the only alternative.

Proposition 28. If ρ_1 is a totally sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms with $|P_1^-| > 1 \vee |P_2^+| > 1$, there exists a partial sequential computon $\lambda_1 \rhd_{\rho_2} \lambda_2$ where ρ_2 is the partially sequentiable span $\lambda_1 \xleftarrow{\alpha_1 \circ \alpha_0} \Lambda \xrightarrow{\alpha_2 \circ \alpha_0} \lambda_2$ with $\alpha_0 : \Lambda \to \lambda_0$.

Proof. Let $\lambda_1 \trianglerighteq_{\rho_1} \lambda_2$ be the pushout of a totally sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms. Since λ_0 is required to have at least one ec-inoutport (see Definition 1), we know there exists at least one computon morphism $\alpha_0 : \Lambda \to \lambda_0$ such that $\alpha_0(p) \in P_0 \iff \alpha_0(p) \in \vec{i}(\alpha_1) \cap \vec{o}(\alpha_2)$ (see Definition 26). Consequently, the span $\rho_2 := \lambda_1 \xleftarrow{\alpha_1 \circ \alpha_0} \Lambda \xrightarrow{\alpha_2 \circ \alpha_0} \lambda_2$ exists.

As $\alpha_0(p) \in \vec{o}(\alpha_2)$ and $\alpha_1(\vec{o}(\alpha_2)) = P_1^-$ (by Definition 26 of totally sequentiable spans), we have $\alpha_1(\alpha_0(p)) \in P_1^-$, i.e., $\alpha_1(\alpha_0(p))\bullet = \emptyset$. The fact that λ_1 is a connected computon implies there exists some computation unit $u_1 \in \bullet\alpha_1(\alpha_0(p))$ so that $\bullet\alpha_1(\alpha_0(p)) \neq \emptyset$ (see Proposition 3). As $\bullet p = \emptyset$ because Λ is a trivial computon, $\bullet\alpha_1(\alpha_0(p)) \setminus \alpha_1(\alpha_0(\bullet p)) \neq \emptyset$ holds, meaning $p \in \vec{i}(\alpha_1 \circ \alpha_0)$ (see Definition 7). A similar reasoning can be used to deduce $\alpha_2(\alpha_0(p)) \in P_2^+$ and $p \in \vec{o}(\alpha_2 \circ \alpha_0)$, i.e., $\bullet\alpha_2(\alpha_0(p)) = \emptyset$. Having $P = \{p\}$ together with $\bullet\alpha_2(\alpha_0(p)) = \alpha_1(\alpha_0(p))\bullet = \bullet p = p\bullet = \emptyset$ allows us to deduce $\vec{i}(\alpha_2 \circ \alpha_0) = \emptyset = \vec{o}(\alpha_1 \circ \alpha_0)$. That is, $P = \vec{i}(\alpha_1 \circ \alpha_0) \cap \vec{o}(\alpha_2 \circ \alpha_0)$.

The facts $\alpha_1(\alpha_0(p)) \in P_1^-$ and $p \in \vec{o}(\alpha_2 \circ \alpha_0)$ together imply $\alpha_1(\alpha_0(\vec{o}(\alpha_2 \circ \alpha_0))) \subseteq P_1^-$. Similarly, $\alpha_2(\alpha_0(p)) \in P_2^+$ and $p \in \vec{i}(\alpha_1 \circ \alpha_0)$ imply $\alpha_2(\alpha_0(\vec{i}(\alpha_1 \circ \alpha_0))) \subseteq P_2^+$. Hence, by Definition 26, ρ_2 is a sequentiable span.

The morphisms of such a span are particularly injective because $|P| = 1$. So, if $|P_1^-| > 1$ or $|P_2^+| > 1$, $\alpha_1(\alpha_0(\vec{o}(\alpha_2 \circ \alpha_0))) \subset P_1^-$ or $\alpha_2(\alpha_0(\vec{i}(\alpha_1 \circ \alpha_0))) \subset P_2^+$. That is, by Lemma 1, ρ_2 is a partially sequentiable span whose pushout forms a partial sequential computon $\lambda_1 \rhd_{\rho_2} \lambda_2$. □

Any computon operand can be put in any order within a sequential computon, since ec-inports and ec-outports always possess the same colour (i.e., zero). This property, combined with the fact that a computon always has at least one ec-inport and at least one ec-outport, allows us to compose any two connected computons sequentially regardless of the data they require or produce (see Theorem 1). Composing two connected computons sequentially always results in another connected computon (see Proposition 29).

Theorem 1. Let λ_1 and λ_2 be two computons. Then, there is a span ρ whose pushout is $\lambda_1 \rhd_\rho \lambda_2$ or $\lambda_1 \trianglerighteq_\rho \lambda_2 \iff \lambda_1$ and λ_2 are connected computons.

Proof. ($\implies$) This part of the proof follows directly from Definition 26.

($\impliedby$) Assuming λ_1 and λ_2 are connected computons, we first prove there exists a sequentiable span $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms. For this, we choose λ_0 to be Λ which is a trivial computon with a unique port $p \in P^+ \cap P^-$ and $c(p) = 0$ (see Definition 19). Below we construct computon morphisms $\alpha_1 : \Lambda \to \lambda_1$ and $\alpha_2 : \Lambda \to \lambda_2$ by only considering port mapping because $I = O = U = \emptyset$ and $\Sigma = \{0\}$.

Since any computon has at least one ec-inport and at least one ec-outport, $C_1^- \neq \emptyset \neq C_2^+$. If we trivially define $\alpha_1(p) = p_1 \in C_1^-$ and $\alpha_2(p) = p_2 \in C_2^+$, then $p \in \vec{i}(\alpha_1) \cap \vec{o}(\alpha_2)$ because λ_1 and λ_2 are connected computons (see Definition 6 and Proposition 3). As $P = \{p\}$, we have $P = \vec{i}(\alpha_1) \cap \vec{o}(\alpha_2)$ so $\alpha_1(\vec{o}(\alpha_2)) \subseteq C_1^- \subseteq P_1^-$ and $\alpha_2(\vec{i}(\alpha_1)) \subseteq C_2^+ \subseteq P_2^+$. That is, ρ is a sequentiable span of computon morphisms (by Definition 26), whose pushout is a sequential computon (by Lemma 1).

If $\alpha_1(\vec{o}(\alpha_2)) \subset C_1^-$ then $\alpha_1(\vec{o}(\alpha_2)) \subset P_1^-$ because $C_1^- \subseteq P_1^-$. Similarly, $\alpha_2(\vec{i}(\alpha_1)) \subset C_2^+$ implies $\alpha_2(\vec{i}(\alpha_1)) \subset P_2^+$ because $C_2^+ \subseteq P_2^+$. Thus, $\lambda_1 \rhd_\rho \lambda_2$ would be the pushout of ρ in both cases. Now, when $\alpha_1(\vec{o}(\alpha_2)) = C_1^-$, we have two possibilities: $C_1^- \subset P_1^-$ or $C_1^- = P_1^-$. If $C_1^- \subset P_1^-$, then $\alpha_1(\vec{o}(\alpha_2)) \subset P_1^-$ so that $\lambda_1 \rhd_\rho \lambda_2$ would also be the pushout of ρ. A similar approach can be used to prove that the partial sequential computon $\lambda_1 \rhd_\rho \lambda_2$ is the pushout of ρ when both $\alpha_2(\vec{i}(\alpha_1)) = C_2^+$ and $C_2^+ \subset P_2^+$ hold.

The only scenario in which $\lambda_1 \trianglerighteq_\rho \lambda_2$ is the pushout of ρ is when λ_1 posesses only one ec-outport with no ed-outports while λ_2 has only one ec-inport with no ed-inports. More precisely, $\lambda_1 \trianglerighteq_\rho \lambda_2$ can be formed exactly when $\alpha_1(\vec{o}(\alpha_2)) = C_1^- = P_1^-$ and $\alpha_2(\vec{i}(\alpha_1)) = C_2^+ = P_1^+$.

Therefore, we conclude that for every pair (λ_1, λ_2) of connected computons, either $\lambda_1 \rhd_\rho \lambda_2$ or $\lambda_1 \trianglerighteq_\rho \lambda_2$ exists. □

Proposition 29. A sequential computon is a connected computon.

Proof. We know that a sequential computon is the pushout of a sequentiable span ρ of computon morphisms (as per Definition 27) and that the legs of ρ are connected computons (as per Definition 26). By Proposition 9, the pushout of ρ must be a connected computon. □

Theorem 1 is important for our theory since it entails that any two connected computons can always be composed sequentially. Although an apex computon always exists, it is important to note that it does not need to correspond to the entire common part between the e-outports of the left operand and the e-inports of the right operand. By common, we mean ports sharing the same colour. Figure 12 depicts a scenario of this sort.

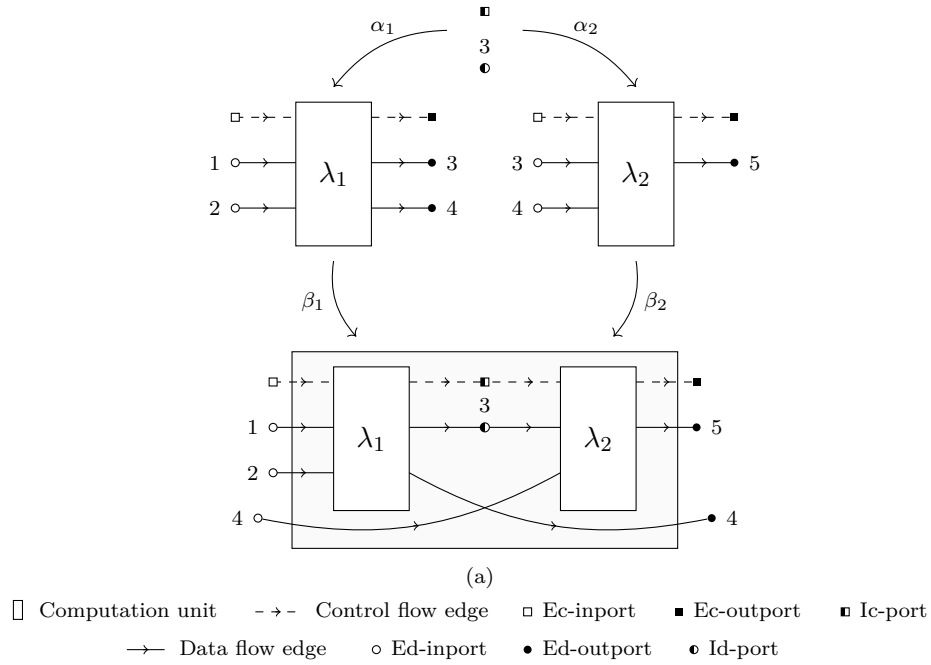


Figure 12: Constructing a partial sequential computon $\lambda_1 \rhd_\rho \lambda_2$ from a partially sequentiable span $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms.

Figure 12 shows that, when a partial sequential computon $\lambda_1 \rhd_\rho \lambda_2$ is constructed from a partially sequentiable span $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$, there is an implicit effect in which all the e-outports of λ_1 that are not in the image of α_1 become e-outports in $\lambda_1 \rhd_\rho \lambda_2$. Similarly, all the e-inports of λ_2 that are not in the image of α_2 become e-inports in $\lambda_1 \rhd_\rho \lambda_2$. Naturally, this generative effect does not occur in the case of total sequential composition since the images of the computon morphisms involved would cover every e-outport of the left operand and every e-inport of the right one. Instead, each $p_1 \in P_1^-$ and each $p_2 \in P_2^+$ would be mapped to an i-port of $\lambda_1 \trianglerighteq_\rho \lambda_2$. No matter whether a partial or a total sequential computon is formed, it is true that every e-inport of the left operand and every e-outport of the right one are preserved in the resulting sequential computon (see Propositions 30 and 31).

Proposition 30. If $(\beta_1 : \lambda_1 \to \lambda_3, \lambda_3, \beta_2 : \lambda_2 \to \lambda_3)$ is the pushout of a sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms, then $\beta_1(P_1^+) \subseteq P_3^+$ and $\beta_2(P_2^-) \subseteq P_3^-$.

Proof. By letting $(\beta_1 : \lambda_1 \to \lambda_3, \lambda_3, \beta_2 : \lambda_2 \to \lambda_3)$ be the pushout of a sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms, we just show $\beta_1(P_1^+) \subseteq P_3^+$ since the proof of $\beta_2(P_2^-) \subseteq P_3^-$ is completely analogous.

Assume for contrapositive that $p_3 \notin P_3^+$ so there is some $o_3 \in O_3$ where $t_3(o_3) = p_3$. The fact $O_3 = O_1 +_{O_0} O_2$ implies three possibilities: (i) there exclusively is some $o_1 \in O_1$ where $\beta_1(o_1) = o_3$, (ii) there exclusively is some $o_2 \in O_2$ where $\beta_2(o_2) = o_3$ or (iii) there are $o_4 \in O_1$ and $o_5 \in O_2$ such that $\beta_1(o_4) = o_3 = \beta_2(o_5)$. The third scenario never holds since λ_0 is a trivial computon by Definition 26. So, we just prove for (i) and (ii).

For (i), $\beta_1(t_1(o_1)) = t_3(\beta_1(o_1)) = t_3(o_3) = p_3$ implies there is some $p_1 \in P_1$ with $t_1(o_1) = p_1$ and $\beta_1(p_1) = p_3$. Consequently, by Definition 2, $p_1 \notin P_1^+$ so $p_3 \notin \beta_1(P_1^+)$. If (ii) holds, $\beta_2(t_2(o_2)) = t_3(\beta_2(o_2)) = t_3(o_3) = p_3$ implies there is some $p_2 \in P_2$ for which $t_2(o_2) = p_2$ and $\beta_2(p_2) = p_3$. If there is some $p_1 \in P_1$ where $\beta_1(p_1) = p_3 = \beta_2(p_2)$, there is some $p_0 \in P_0$ where $\alpha_1(p_0) = p_1$ and $\alpha_2(p_0) = p_2$. As $(\nexists o_1 \in O_1)[\beta_1(o_1) = o_3 = \beta_2(o_2)]$ and σ_2 is surjective, $p_0 \in \vec{i}(\alpha_2)$ which contradicts the fact $\vec{i}(\alpha_2) = \emptyset$ (see Corollary 1). So, there is no $p_1 \in P_1$ where $\beta_1(p_1) = p_3 = \beta_2(p_2)$. That is, $p_3 \notin \beta_1(P_1^+)$.

Proving $p_3 \notin P_3^+ \implies p_3 \notin \beta_1(P_1^+)$ in the above cases entails $\beta_1(P_1^+) \subseteq P_3^+$, as required. □

Proposition 31. If $(\beta_1 : \lambda_1 \to \lambda_3, \lambda_3, \beta_2 : \lambda_2 \to \lambda_3)$ is the pushout of a totally sequentiable span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms, then $\beta_1(P_1^+) = P_3^+$ and $\beta_2(P_2^-) = P_3^-$.

Proof. Assuming $(\beta_1 : \lambda_1 \to \lambda_3, \lambda_3, \beta_2 : \lambda_2 \to \lambda_3)$ is the pushout of a totally sequentiable span $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms, below we just show $\beta_1(P_1^+) = P_3^+$ since the proof of $\beta_2(P_2^-) = P_3^-$ is completely analogous.

As Proposition 30 says $\beta_1(P_1^+) \subseteq P_3^+$, we just have to prove $P_3^+ \subseteq \beta_1(P_1^+)$. If we let $p_3 \in P_3^+$, by Proposition 5 and by the fact $P_3 = P_1 +_{P_0} P_2$, we have three options: (i) there exclusively is some $p_1 \in P_1^+$ such that $\beta_1(p_1) = p_3$, (ii) there exclusively is some $p_2 \in P_2^+$ such that $\beta_2(p_2) = p_3$ or (iii) there are $p_4 \in P_1^+$ and $p_5 \in P_2^+$ such that $\beta_1(p_4) = p_3 = \beta_2(p_5)$. If (i) is true, then $p_3 \in \beta_1(P_1^+)$ follows directly. We now show that (ii) and (iii) cannot hold.

Supposing (ii) is true, we have $p_2 \in \alpha_2(\vec{i}(\alpha_1))$ because $\alpha_2(\vec{i}(\alpha_1)) = P_2^+$ (by the fact that ρ is totally sequentiable). Therefore, $(\exists p_0 \in P_0 \cap \vec{i}(\alpha_1))(\exists p_1 \in P_1)[\alpha_1(p_0) = p_1 \ \wedge \ \alpha_2(p_0) = p_2]$. As commutativity contradicts (ii), there is no $p_2 \in P_2^+$ such that $\beta_2(p_2) = p_3 \in P_3^+$. That is, $p_3 \notin \beta_2(P_2^+)$. To disprove (iii), we deduce by commutativity the existence of some $p \in P_0$ where $\alpha_1(p) = p_4$ and $\alpha_2(p) = p_5$. Since $p_4 \in P_1^+$ and $p_5 \in P_2^+$, Proposition 5 says $p \in P_0^+$. The fact that λ_2 is a connected computon and that $\alpha_2(p) = p_5 \in P_2^+$ entail $p \in \vec{o}(\alpha_2)$. Because ρ is totally sequentiable, it is true that $\alpha_1(\vec{o}(\alpha_2)) = P_1^-$ and, consequently, $\alpha_1(p) = p_4 \in P_1^-$. But λ_1 is also a connected computon, so $p_4 \in P_1^+ \cap P_1^-$ cannot hold because that would contradict Proposition 2. This contradiction implies there is no $p_5 \in P_2^+$ where $\beta_2(p_5) = p_3 \in P_3^+$, i.e., $p_3 \notin \beta_2(P_2^+)$.

Proving (i) and disproving (ii) and (iii) entails $p_3 \in P_3^+ \implies [p_3 \in \beta_1(P_1^+) \setminus \beta_2(P_2^+)]$, i.e., $P_3^+ \subseteq \beta_1(P_1^+)$, as required. □

Although Figure 12 shows an example of partial sequential composition, the same computon operands can be used to perform total sequential composition. This is because, in this case, there exists an apex computon that can be inserted into all the e-outports of λ_1 and into all the e-inports of λ_2. Such an apex does not always exist so partiality does not imply totality and, thus, the reverse of Proposition 28 does not hold. Proposition 28 combined with Theorem 1 states that if any two connected computons can be composed into a total sequential computon, they can also be composed into a partial sequential computon, only if the left operand has more than one e-outport or if the right operand has at least two e-inports; otherwise, such computons can only form a total sequential computon. While total sequential composition is an associative operation (see Proposition 32), partial sequential composition is not (see Proposition 33). In both cases, commutativity does not hold in the sense that the order of the operands matters (see Proposition 34).

Proposition 32 (Total sequential composition is associative)**.** There is an isomorphism between $\lambda_1 \trianglerighteq_{\rho_3} (\lambda_2 \trianglerighteq_{\rho_2} \lambda_3)$ and $(\lambda_1 \trianglerighteq_{\rho_1} \lambda_2) \trianglerighteq_{\rho_4} \lambda_3$ for any choice of total sequential computons $\lambda_1 \trianglerighteq_{\rho_1} \lambda_2$, $\lambda_2 \trianglerighteq_{\rho_2} \lambda_3$, $\lambda_1 \trianglerighteq_{\rho_3} (\lambda_2 \trianglerighteq_{\rho_2} \lambda_3)$ and $(\lambda_1 \trianglerighteq_{\rho_1} \lambda_2) \trianglerighteq_{\rho_4} \lambda_3$.

Proof. Let $\rho_1 := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ and $\rho_2 := \lambda_2 \xleftarrow{\alpha_3} \lambda_4 \xrightarrow{\alpha_4} \lambda_3$ be two totally sequentiable spans of computon morphisms. By Definition 27 and Lemma 1, we know that the pushouts of ρ_1 and ρ_2 are the total sequential computons $\lambda_1 \trianglerighteq_{\rho_1} \lambda_2$ and $\lambda_2 \trianglerighteq_{\rho_2} \lambda_3$, respectively. Accordingly, consider the following commutative diagram:

$$\begin{array}{ccccc}
 & & \lambda_4 & \xrightarrow{\alpha_4} & \lambda_3 \\
 & & \downarrow{\scriptstyle \alpha_3} & & \downarrow{\scriptstyle \beta_4} \\
\lambda_0 & \xrightarrow{\alpha_2} & \lambda_2 & \xrightarrow{\beta_3} & \lambda_2 \trianglerighteq_{\rho_2} \lambda_3 \\
\downarrow{\scriptstyle \alpha_1} & & \downarrow{\scriptstyle \beta_2} & & \downarrow{\scriptstyle \beta_6} \\
\lambda_1 & \xrightarrow[\beta_1]{} & \lambda_1 \trianglerighteq_{\rho_1} \lambda_2 & \xrightarrow[\beta_5]{} & \lambda_5
\end{array}$$

where $\lambda_1 \trianglerighteq_{\rho_1} \lambda_2 \xleftarrow{\beta_2} \lambda_2 \xrightarrow{\beta_3} \lambda_2 \trianglerighteq_{\rho_2} \lambda_3$ is a pushout-induced span of computon morphisms, which evidently is not sequentiable by the fact that λ_2 is not a trivial computon. As it is routine to show it is pushable, we have that its pushout λ_5 can be constructed. Consequently, each square of the above diagram is a pushout. Using categorical algebra to horizontally and vertically compose morphisms, we obtain the following diagrams:

$$\begin{array}{ccc}
\lambda_0 & \xrightarrow{\beta_3 \circ \alpha_2} & \lambda_2 \trianglerighteq_{\rho_2} \lambda_3 \\
\downarrow{\scriptstyle \alpha_1} & & \downarrow{\scriptstyle \beta_6} \\
\lambda_1 & \xrightarrow[\beta_5 \circ \beta_1]{} & \lambda_5
\end{array}
\qquad\qquad
\begin{array}{ccc}
\lambda_4 & \xrightarrow{\alpha_4} & \lambda_3 \\
\downarrow{\scriptstyle \beta_2 \circ \alpha_3} & & \downarrow{\scriptstyle \beta_6 \circ \beta_4} \\
\lambda_1 \trianglerighteq_{\rho_1} \lambda_2 & \xrightarrow[\beta_5]{} & \lambda_5
\end{array}$$

By letting ρ_3 be the (above) span $\lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\beta_3 \circ \alpha_2} \lambda_2 \trianglerighteq_{\rho_2} \lambda_3$, we now show its pushout λ_5 is the total sequential computon $\lambda_1 \trianglerighteq_{\rho_3} (\lambda_2 \trianglerighteq_{\rho_2} \lambda_3)$. For this, we first prove that ρ_3 is totally sequentiable: $p_0 \in P_0 \iff p_0 \in \vec{i}(\alpha_1) \cap \vec{o}(\alpha_2)$ (because ρ_1 is sequentiable) $\iff p_0 \in \vec{i}(\alpha_1)$ and $\alpha_2(p_0)\bullet \setminus \alpha_2(p_0\bullet) \neq \emptyset$ (by Definition 7) $\iff p_0 \in \vec{i}(\alpha_1)$ and $\beta_3(\alpha_2(p_0))\bullet \setminus \beta_3(\alpha_2(p_0\bullet)) \neq \emptyset$ (by the preservation of computation units) $\iff p_0 \in \vec{i}(\alpha_1) \cap \vec{o}(\beta_3 \circ \alpha_2)$ (by Definition 7). Therefore, $P_0 = \vec{i}(\alpha_1) \cap \vec{o}(\beta_3 \circ \alpha_2)$, i.e., Condition (i) of Definition 26 is met by ρ_3.

As λ_1 and $\lambda_2 \trianglerighteq_{\rho_2} \lambda_3$ are connected computons (by Definition 26 and Proposition 29), it follows that Condition (ii) of Definition 26 is also met by ρ_3. To prove $\alpha_1(\vec{o}(\beta_3 \circ \alpha_2)) = P_1^-$, consider the following chain of double implications: $p \in \alpha_1(\vec{o}(\beta_3 \circ \alpha_2)) \iff p \in \alpha_1(P_0)$ (because $P_0 = \vec{o}(\beta_3 \circ \alpha_2)$) $\iff p \in P_1^-$ (because $\alpha_1(P_0) = P_1^-$ by the fact that ρ_1 is totally sequentiable). To prove the last condition of totally sequentiable spans, we proceed as follows: $q \in \beta_3(\alpha_2(\vec{i}(\alpha_1))) \iff q \in \beta_3(P_2^+)$ (because $\alpha_2(\vec{i}(\alpha_1)) = P_2^+$ by the fact that ρ_1 is totally sequentiable) $\iff q$ is an e-inport of $\lambda_2 \trianglerighteq_{\rho_2} \lambda_3$ (by Proposition 31).

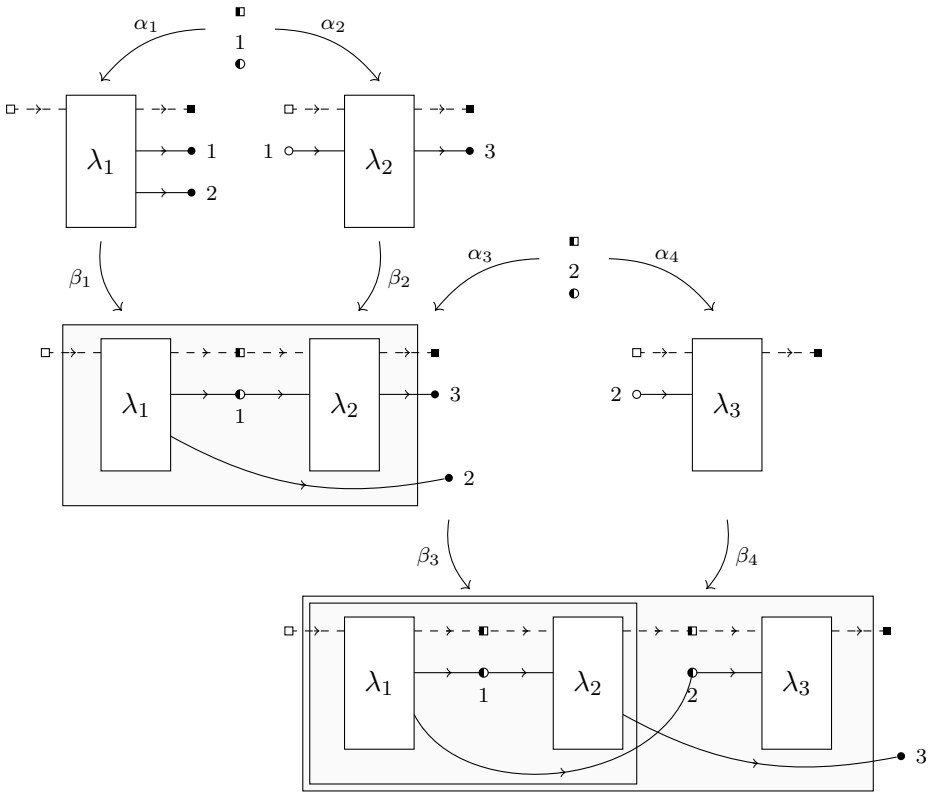
Above we proved that ρ_3 is a totally sequentiable span of computon morphisms where λ_0 is the apex computon, λ_1 the left operand and $\lambda_2 \trianglerighteq_{\rho_2} \lambda_3$ the right operand. Using Lemma 1 and Definition 27, we have that the pushout of ρ_3 is the total sequential computon $\lambda_1 \trianglerighteq_{\rho_3} (\lambda_2 \trianglerighteq_{\rho_2} \lambda_3)$. Deducing that $(\lambda_1 \trianglerighteq_{\rho_1} \lambda_2) \trianglerighteq_{\rho_4} \lambda_3$ is the pushout of the span $\rho_4 := \lambda_1 \trianglerighteq_{\rho_1} \lambda_2 \xleftarrow{\beta_2 \circ \alpha_3} \lambda_4 \xrightarrow{\alpha_4} \lambda_3$ can be done analogously.

As $\lambda_1 \trianglerighteq_{\rho_3} (\lambda_2 \trianglerighteq_{\rho_2} \lambda_3)$, λ_5 and $(\lambda_1 \trianglerighteq_{\rho_1} \lambda_2) \trianglerighteq_{\rho_4} \lambda_3$ are evidently isomorphic, we conclude that total sequential composition is an associative operation up to isomorphism. □

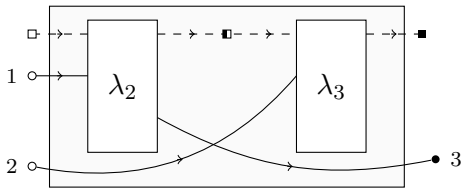
Proposition 33. Partial sequential composition is not associative.

Proof. We show there is no isomorphism between $(\lambda_1 \triangleright_{\rho_1} \lambda_2) \triangleright_{\rho_2} \lambda_3$ and $\lambda_1 \triangleright_{\rho_4} (\lambda_2 \triangleright_{\rho_3} \lambda_3)$ for some choice of partial sequential computons $\lambda_1 \triangleright_{\rho_1} \lambda_2$, $(\lambda_1 \triangleright_{\rho_1} \lambda_2) \triangleright_{\rho_2} \lambda_3$, $\lambda_2 \triangleright_{\rho_3} \lambda_3$ and $\lambda_1 \triangleright_{\rho_4} (\lambda_2 \triangleright_{\rho_3} \lambda_3)$.

For this, let us consider Figure 13(a) in which there are two partially sequentiable spans of computon morphisms: $\rho_1 := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ and $\rho_2 := (\lambda_1 \triangleright_{\rho_1} \lambda_2) \xleftarrow{\alpha_3} \lambda_4 \xrightarrow{\alpha_4} \lambda_3$.



(a) Constructing the partial sequential computon $(\lambda_1 \triangleright_{\rho_1} \lambda_2) \triangleright_{\rho_2} \lambda_3$.



(b) Partial sequential computon $\lambda_2 \triangleright_{\rho_3} \lambda_3$.

Computation unit Control flow edge Ec-inport Ec-outport Ic-port
Data flow edge Ed-inport Ed-outport Id-port

Figure 13: Counterexample that disproves the associativity of partial sequential composition.

Carefully observing Figure 13(a) reveals that, up to isomorphism, there is only one partially sequentiable span $\rho_3 := \lambda_2 \xleftarrow{\alpha_5} \lambda_5 \xrightarrow{\alpha_6} \lambda_3$ where λ_5 is a unit computon which can only be injected into the unique ec-outport of λ_2 and into the unique ec-inport of λ_3 to yield the partial sequential computon $\lambda_2 \triangleright_{\rho_3} \lambda_3$ depicted in Figure 13(b). Now, to construct a partial sequential computon $\lambda_1 \triangleright_{\rho_4} (\lambda_2 \triangleright_{\rho_3} \lambda_3)$ isomorphic to $(\lambda_1 \triangleright_{\rho_1} \lambda_2) \triangleright_{\rho_2} \lambda_3$, ρ_4 must necessarily be totally sequentiable. So, the proposition being proved is true. □

Proposition 34. Sequential composition is not commutative.

Proof. For this proposition, we show:

1. There is no isomorphism between $\lambda_1 \triangleright_{\rho_1} \lambda_2$ and $\lambda_2 \triangleright_{\rho_2} \lambda_1$ for some choice of partial sequential computons $\lambda_1 \triangleright_{\rho_1} \lambda_2$ and $\lambda_2 \triangleright_{\rho_2} \lambda_1$.
2. There is no isomorphism between $\lambda_3 \trianglerighteq_{\rho_3} \lambda_4$ and $\lambda_4 \trianglerighteq_{\rho_4} \lambda_3$ for some choice of total sequential computons $\lambda_3 \trianglerighteq_{\rho_3} \lambda_4$ and $\lambda_4 \trianglerighteq_{\rho_4} \lambda_3$.

For 1, let us consider the partially sequentiable span $\rho_1 := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ described in Figure 12 and the obvious partially sequentiable span $\rho_2 := \lambda_2 \xleftarrow{\gamma_2} \Lambda \xrightarrow{\gamma_1} \lambda_1$ given by $\gamma_2(p) = p_2 \in C_2^-$ and $\gamma_1(p) = p_1 \in C_1^+$. As it is trivial to check that $\lambda_1 \triangleright_{\rho_1} \lambda_2$ and $\lambda_2 \triangleright_{\rho_2} \lambda_1$ are not

isomorphic, we have just constructed an example that disproves the commutativity of partial sequential composition. For 2, consider the example depicted in Figure 14 which evidently shows that the total sequential computons (a) and (b) are not isomorphic. Hence, we conclude that total sequential composition is not a commutative operation either.

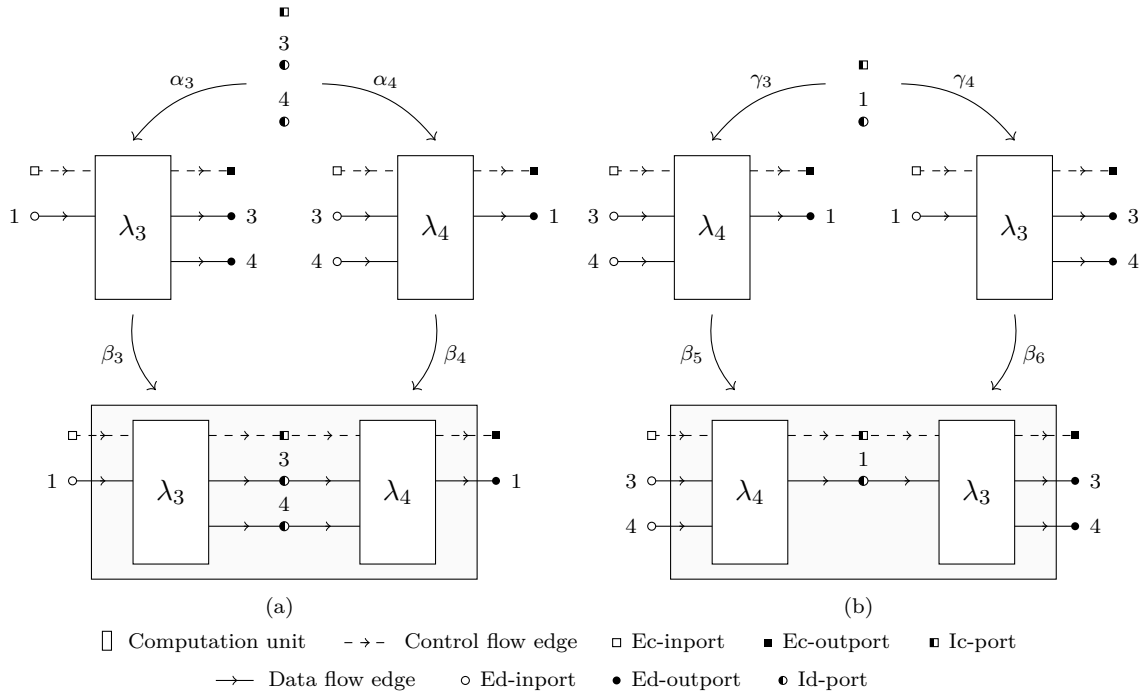
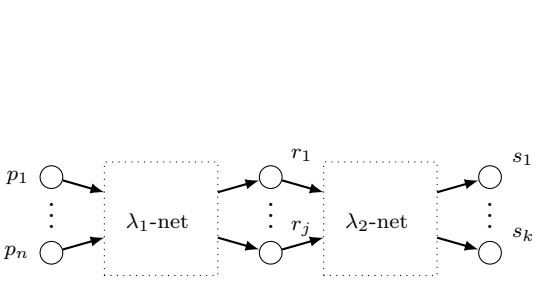


Figure 14: Counterexample that disproves the commutativity of total sequential composition.

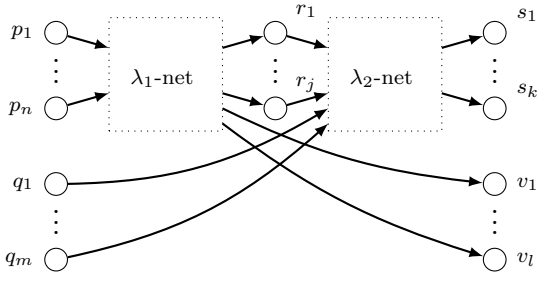
□

6.1.1. Operational semantics for sequential computons (in the theory of Petri nets)

No matter whether we use any of the three functors presented in Section 3, the Petri net of a sequential computon does not introduce any additional places or transitions, as a result of using a pushout operation on a sequentiable span of computon morphisms. In the case of a total sequential computon, the corresponding net takes all the e-inports from the left operand as input places and all the e-outports of the right operand as output places (see Figure 15(a) and Proposition 31). The net of a partial sequential computon has a similar structure, with the addition it has the unmatched e-inports of the right operand as input places and the unmatched e-ouports of the left operand as output places (see Figure 15(b)).



(a) Net of a total sequential computon $\lambda_1 \trianglerighteq_\rho \lambda_2$ constructed from a connected computon λ_1 that has n e-inports and j e-outports, and a connected computon λ_2 that has j e-inports and k e-outports.



(b) Net of a partial sequential computon $\lambda_1 \triangleright_\rho \lambda_2$ constructed from a connected computon λ_1 that has n e-inports and $j+l$ e-outports, and a connected computon λ_2 that has $j+m$ e-inports and k e-outports.

Figure 15: General structure of sequential computon nets. Both figures are applicable to all the functorial constructions from Section 3, namely $\mathcal{N}$, $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$. In the case of $\mathcal{D}$, $j, k, l, m, n \geq 0$ and for the others, $j, k, l, m, n > 0$ according to Definition 1.

Although there are no new places or transitions that could cause deadlocks in the net of a (partial or total) sequential computon, there is no guarantee such a net is deadlock-free, even though the nets of the composed computons are. To fully ensure deadlock-freedom, we have to synchronise the e-outports of the left operand with the e-inports of the right one, in order to prevent the net of the right operand from being executed before reaching its initial state. This can structurally be done by introducing a primitive computon that acts as a synchronisation point between the left and right operands. The formal notion of such a computon is given in Definition 28.

Definition 28 (In- and Out-Sync Computons)**.** A primitive computon λ is an in-sync of a connected computon λ_1 if the domain of λ_1^+ is the domain of both λ^+ and λ^-. It is an out-sync of λ_1 if the domain of λ_1^- is the domain of both λ^+ and λ^-.

Evidently, for any connected computon there always are in- and out-sync computons by the fact marker morphisms always exist (see Proposition 21). Propositions 35 and 36 show that the process of adapting an in- or an out-sync to a connected computon precisely corresponds to a total sequencing operation.

Proposition 35. If λ is the in-sync of a connected computon λ_1, there is a span ρ whose pushout is the total sequential computon $\lambda \trianglerighteq_\rho \lambda_1$.

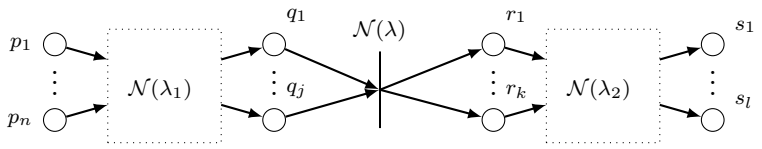
Proof. Assuming λ is the in-sync of a connected computon λ_1, Definition 28 says there is a trivial computon λ_2 that gives rise to the span $\rho := \lambda \xleftarrow{\lambda^-} \lambda_2 \xrightarrow{\lambda_1^+} \lambda_1$. Verifying ρ is totally sequentiable follows directly from Definition 20 and Proposition 26. So, by Definition 27, the pushout of ρ is the total sequential computon $\lambda \trianglerighteq_\rho \lambda_1$. □

Proposition 36. If λ is the out-sync of a connected computon λ_1, there is a span ρ whose pushout is the total sequential computon $\lambda_1 \trianglerighteq_\rho \lambda$.

Proof. The proof is analogous to that of Proposition 35. □

Proposition 37. Let $\lambda_1 \trianglerighteq_\rho \lambda_2$ be a total sequential computon and λ the in-sync of λ_2. If $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock-free, there are spans ρ_1 and ρ_2 such that $\mathcal{N}(\lambda_1 \trianglerighteq_{\rho_2} \lambda \trianglerighteq_{\rho_1} \lambda_2)$ is deadlock-free.

Proof. Let $\lambda_1 \trianglerighteq_\rho \lambda_2$ be a total sequential computon constructed from the totally sequentiable span $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms. If λ is the in-sync of λ_2, Proposition 35 says there is a trivial computon λ_3 such that the pushout of $\rho_1 := \lambda \xleftarrow{\lambda^-} \lambda_3 \xrightarrow{\lambda_2^+} \lambda_2$ is the total sequential computon $\lambda \trianglerighteq_{\rho_1} \lambda_2$. If $\beta : \lambda_2 \to \lambda \trianglerighteq_{\rho_1} \lambda_2$ is one of the morphisms induced by such a pushout, we deduce the existence of $\rho_2 := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\beta \circ \alpha_2} \lambda \trianglerighteq_{\rho_1} \lambda_2$ which trivially is totally sequentiable by the fact ρ_1 and ρ are. Consequently, by Definition 27, there is a total sequential computon $\lambda_1 \trianglerighteq_{\rho_2} \lambda \trianglerighteq_{\rho_1} \lambda_2$ whose underlying net $\mathcal{N}(\lambda_1 \trianglerighteq_{\rho_2} \lambda \trianglerighteq_{\rho_1} \lambda_2)$ has the following form when applying the functorial construction from Definition 15:



By Definition 16, the initial state M_i of the above net is a marking function where $M_i(p) > 0$ for all $p \in \{p_1, \ldots, p_n\}$ and no tokens for all the other places, including those inside $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$. As this marking corresponds to the initial marking of $\mathcal{N}(\lambda_1)$, only states from $\mathcal{N}(\lambda_1)$ are reachable from M_i in the next time step. Assuming $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock-free, we now have the following cases:

- If no state of $\mathcal{N}(\lambda_1)$ puts tokens in all the input places of $\mathcal{N}(\lambda)$, no state of $\mathcal{N}(\lambda)$ or $\mathcal{N}(\lambda_2)$ will ever be reached. Even though $\mathcal{N}(\lambda_1)$ will not terminate successfully, there is a guarantee $\mathcal{N}(\lambda_1 \trianglerighteq_{\rho_2} \lambda \trianglerighteq_{\rho_1} \lambda_2)$ will never be stuck because $\mathcal{N}(\lambda_1)$ is deadlock-free.
- If a state of $\mathcal{N}(\lambda_1)$ puts tokens in all the places in $\{q_1, \ldots, q_j\}$, the only transition of $\mathcal{N}(\lambda)$ will be fired to reach a state that marks all the places in $\{r_1, \ldots, r_k\}$, which evidently corresponds to the initial marking of $\mathcal{N}(\lambda_2)$. As $\mathcal{N}(\lambda_2)$ is deadlock-free, $\mathcal{N}(\lambda_1 \trianglerighteq_{\rho_2} \lambda \trianglerighteq_{\rho_1} \lambda_2)$ will not be stuck.

Hence, we conclude $\mathcal{N}(\lambda_1 \trianglerighteq_{\rho_2} \lambda \trianglerighteq_{\rho_1} \lambda_2)$ is deadlock-free, as required. □

Proposition 38. Let $\lambda_1 \trianglerighteq_\rho \lambda_2$ be a total sequential computon and λ the out-sync of λ_1. If $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock-free, there are spans ρ_1 and ρ_2 such that $\mathcal{N}(\lambda_1 \trianglerighteq_{\rho_1} \lambda \trianglerighteq_{\rho_2} \lambda_2)$ is deadlock-free.

Proof. Let $\lambda_1 \trianglerighteq_\rho \lambda_2$ be a total sequential computon constructed from the totally sequentiable span $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms. If λ is the out-sync of λ_1, Proposition 36 says there is a trivial computon λ_3 such that the pushout of $\rho_1 := \lambda_1 \xleftarrow{\lambda_1^-} \lambda_3 \xrightarrow{\lambda^+} \lambda$ is the total sequential computon $\lambda_1 \trianglerighteq_{\rho_1} \lambda$. If $\beta : \lambda_1 \to \lambda_1 \trianglerighteq_{\rho_1} \lambda$ is one of the morphisms induced by such a pushout, we deduce the existence of $\rho_2 := \lambda_1 \trianglerighteq_{\rho_1} \lambda \xleftarrow{\beta \circ \alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ which trivially is totally sequentiable by the fact ρ_1 and ρ are. Consequently, by Definition 27, there is a total sequential computon $\lambda_1 \trianglerighteq_{\rho_1} \lambda \trianglerighteq_{\rho_2} \lambda_2$.

Applying the functorial construction $\mathcal{N}$ from Definition 15 on $\lambda_1 \trianglerighteq_{\rho_1} \lambda \trianglerighteq_{\rho_2} \lambda_2$ results in a Petri net with the form shown in the proof of Proposition 37. Therefore, by that proposition, $\mathcal{N}(\lambda_1 \trianglerighteq_{\rho_1} \lambda \trianglerighteq_{\rho_2} \lambda_2)$ is deadlock-free. □

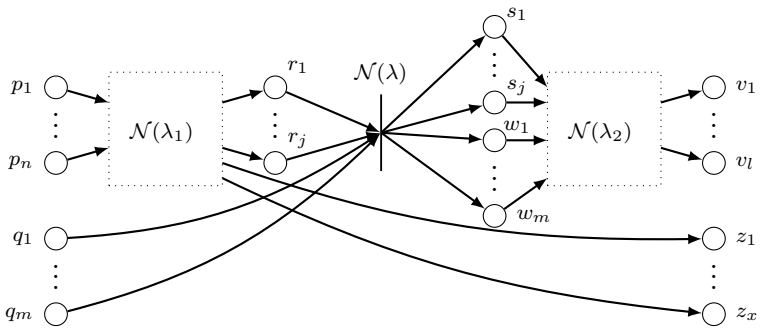
Corollary 2. Let $\lambda_1 \trianglerighteq_\rho \lambda_2$ be a total sequential computon, λ_3 the in-sync of λ_2 and λ_4 the out-sync of λ_1. Then, there are spans ρ_1, ρ_2, ρ_3 and ρ_4 such that $\mathcal{N}(\lambda_1 \trianglerighteq_{\rho_1} \lambda_3 \trianglerighteq_{\rho_2} \lambda_2) \cong \mathcal{N}(\lambda_1 \trianglerighteq_{\rho_3} \lambda_4 \trianglerighteq_{\rho_4} \lambda_2)$.

Proof. The proof follows directly from Propositions 37 and 38, with the observation that λ_3 must necessarily be isomorphic to λ_4. □

Propositions 37 and 38 together entail it is always possible to construct a deadlock-free net from any total sequential computon. To do so, it suffices to place a sync computon between the left and right operands. More precisely, we can attach either an in-sync computon to the right operand or an out-sync computon to the left one (see Corollary 2) via a total sequencing operation (see Propositions 35 and 36). The resulting composite can then be composed into a new total sequential computon that respects the mapping given by the original sequentiable span of computon morphisms. No matter whether we adapt the right or the left operand, the net of the resulting composite is deadlock-free by Propositions 37 and 38. For partial sequential computons, a similar approach enables deadlock-freedom, as described by Propositions 39 and 40.

Proposition 39. Let $\lambda_1 \triangleright_\rho \lambda_2$ be a partial sequential computon and λ the in-sync of λ_2. If $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock-free, there are spans ρ_1 and ρ_2 such that $\mathcal{N}(\lambda_1 \triangleright_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2))$ is deadlock-free.

Proof. Let $\lambda_1 \triangleright_\rho \lambda_2$ be a partial sequential computon constructed from the partially sequentiable span $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms. If λ is the in-sync of λ_2, Proposition 35 says there is a trivial computon λ_3 such that the pushout of $\rho_1 := \lambda \xleftarrow{\lambda^-} \lambda_3 \xrightarrow{\lambda_2^+} \lambda_2$ is the total sequential computon $\lambda \trianglerighteq_{\rho_1} \lambda_2$. If $\beta : \lambda_2 \to \lambda \trianglerighteq_{\rho_1} \lambda_2$ is one of the morphisms induced by such a pushout, we deduce the existence of $\rho_2 := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\beta \circ \alpha_2} \lambda \trianglerighteq_{\rho_1} \lambda_2$ which trivially is partially sequentiable by the fact ρ is. Consequently, by Definition 27, there is a partial sequential computon $\lambda_1 \triangleright_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2)$ whose underlying net $\mathcal{N}(\lambda_1 \triangleright_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2))$ has the following form when applying the functorial construction from Definition 15:



By Definition 16, the initial state M_i of the above net is a marking function where $M_i(p) > 0$ for all $p \in \{p_1, \ldots, p_n, q_1, \ldots, q_m\}$ and no tokens for all the other places, including those inside $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$. This initial marking can only reach states from $\mathcal{N}(\lambda_1)$ in the next time step, since the only transition of $\mathcal{N}(\lambda)$ is not yet enabled ($j > 0$ by Definition 1). Assuming $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock-free, we now have the following cases:

- If no state of $\mathcal{N}(\lambda_1)$ puts tokens in all the places in $\{r_1, \ldots, r_j\}$, the only transition of $\mathcal{N}(\lambda)$ will not be enabled. Consequently, no state of $\mathcal{N}(\lambda)$ or $\mathcal{N}(\lambda_2)$ will ever be reached. Even though $\mathcal{N}(\lambda_1)$ will not terminate successfully, there is a guarantee $\mathcal{N}(\lambda_1 \rhd_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2))$ will never be stuck because $\mathcal{N}(\lambda_1)$ is deadlock-free.
- If a state of $\mathcal{N}(\lambda_1)$ puts tokens in all the places in $\{r_1, \ldots, r_j\}$, the unique transition of $\mathcal{N}(\lambda)$ will be enabled since there are also tokens in $\{q_1, \ldots, q_m\}$ previously placed by M_i. As firing such a transition will evidently reach the initial marking of $\mathcal{N}(\lambda_2)$ and $\mathcal{N}(\lambda_2)$ is deadlock-free, $\mathcal{N}(\lambda_1 \rhd_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2))$ will not be stuck.

Hence, $\mathcal{N}(\lambda_1 \rhd_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2))$ is deadlock-free, as required. □

Remark 10. Due to the non-associativity property of partial sequential composition (see Proposition 33), the total sequential computon $\lambda \trianglerighteq_{\rho_1} \lambda_2$ must be defined before constructing the corresponding partial sequential computon. This is the reason we use parentheses for the syntactic expression $\lambda_1 \rhd_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2)$.

Remark 11. Having the same subindex for r- and s-places (and for q- and w-places) is not a coincidence. We did this to reflect the fact there is a one-to-one correspondence between the input and outputs places of the net of a sync computon. As we are adapting λ_2 rather than λ_1 and ρ_2 is partially sequentiable, there are some input places of $\mathcal{N}(\lambda)$ for which there is no match (i.e., $q_1, \ldots, q_m$) and some output places of $\mathcal{N}(\lambda_1)$ for which there is no match (i.e., $z_1, \ldots, z_x$). This evidently is a structural consequence of partial composition semantics.

Proposition 40. Let $\lambda_1 \rhd_\rho \lambda_2$ be a partial sequential computon and λ the out-sync of λ_1. If $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock-free, there are spans ρ_1 and ρ_2 such that $\mathcal{N}((\lambda_1 \trianglerighteq_{\rho_1} \lambda) \rhd_{\rho_2} \lambda_2)$ is deadlock-free.

Proof. The proof similar to that of Proposition 39. □

Remark 12. Although Propositions 37–40 and Corollary 2 are statements about the functor $\mathcal{N}$, they are applicable to the functors $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$ presented in Section 3. The proofs are valid for $\mathcal{C} \circ \mathfrak{E}$ since Proposition 16 says $\mathcal{C}$ is just a restriction of $\mathcal{N}$ to $\mathfrak{E}(\mathbf{Set}^{\mathbf{Comp}})$.

By Remark 5, we only need to check deadlock-freedom for $\mathcal{D}$-nets that have initial and final states. In the case of Proposition 37, this occurs when $n, l > 0$. As total sequencing connects all the ed-outports of the left operand with all the ed-inports of the right one, we only have to check the cases when $j = 0 = k$ and $j, k > 0$. In the first one, only states of $\mathcal{D}(\lambda_1)$ are reachable from the initial marking. If such a net is deadlock-free, $\mathcal{D}(\lambda_1 \trianglerighteq_{\rho_2} \lambda \trianglerighteq_{\rho_1} \lambda_2)$ is deadlock-free too. When $j, k > 0$, the proof is identical to that of Proposition 37. Proposition 38 and Corollary 2 follow analogously for $\mathcal{D}$.

For Proposition 39, we observe $\mathcal{D}(\lambda_1 \rhd_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2))$ has the following possibilities for j and m (considering $n, l > 0$ by Remark 5):

1. If $j = 0$ and $m = 0$, $\mathcal{D}(\lambda_1)$ and $\mathcal{D}(\lambda_2)$ do not share any data. In this case, M_i marks the places $p_1, \ldots, p_n$ so only states of $\mathcal{D}(\lambda_1)$ are reached. Since such a net is deadlock-free, $\mathcal{D}(\lambda_1 \rhd_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2))$ is deadlock-free too.
2. If $j = 0$ and $m > 0$, $\mathcal{D}(\lambda_1)$ and $\mathcal{D}(\lambda_2)$ do not share any data. In this case, M_i marks the places $p_1, \ldots, p_n, q_1, \ldots, q_m$ to reach states from $\mathcal{D}(\lambda_1)$ and $\mathcal{D}(\lambda)$ simultaneously. Since $j = 0$, the only transition of $\mathcal{D}(\lambda)$ is enabled by M_i and firing it results in tokens in $w_1, \ldots, w_m$ thereby reaching the initial state of $\mathcal{D}(\lambda_2)$. As $\mathcal{D}(\lambda_1)$ and $\mathcal{D}(\lambda_2)$ are both deadlock-free, $\mathcal{D}(\lambda_1 \rhd_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2))$ is deadlock-free too.
3. If $j > 0$ and $m = 0$, $\mathcal{D}(\lambda_1)$ and $\mathcal{D}(\lambda_2)$ do share data. In this case, M_i marks the places $p_1, \ldots, p_n$ to solely activate $\mathcal{D}(\lambda_1)$. If no state of $\mathcal{D}(\lambda_1)$ ever puts tokens in all the places in $\{r_1, \ldots, r_j\}$, no state of $\mathcal{D}(\lambda)$ or $\mathcal{D}(\lambda_2)$ will ever be reached. As $\mathcal{D}(\lambda_1)$ is deadlock-free, $\mathcal{D}(\lambda_1 \rhd_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2))$ will not get stuck. If a state of $\mathcal{D}(\lambda_1)$ puts tokens in all the places in $\{r_1, \ldots, r_j\}$, then states of $\mathcal{D}(\lambda_2)$ will be reached. Since $\mathcal{D}(\lambda_2)$ is deadlock-free too, $\mathcal{D}(\lambda_1 \rhd_{\rho_2} (\lambda \trianglerighteq_{\rho_1} \lambda_2))$ will not get stuck.
4. If $j > 0$ and $m > 0$, $\mathcal{D}(\lambda_1)$ and $\mathcal{D}(\lambda_2)$ do share data. In this case, the proof is identical to that of Proposition 39.

A similar reasoning applies to $\mathcal{D}((\lambda_1 \trianglerighteq_{\rho_3} \lambda) \rhd_{\rho_4} \lambda_2)$ with respect to Proposition 40.

6.1.2. Encapsulation of control flow and data flow in sequential computons

A total sequential computon encapsulates sequential control flow and up to sequential data flow, as a result of matching all the e-outports of the left operand with all the e-inports of the right one via a pushout operation on a totally sequentiable span of computon morphisms. Sequential control flow allows the strict execution of one computon after another, whilst sequential data flow serves to transfer all the computation results of the left operand to the right one. Figure 16 illustrates how the total sequential computon from Figure 14(a) encapsulates both sequential control flow for the invocation of λ_3 and λ_4 (in that order) and sequential data flow for passing two data items between them.

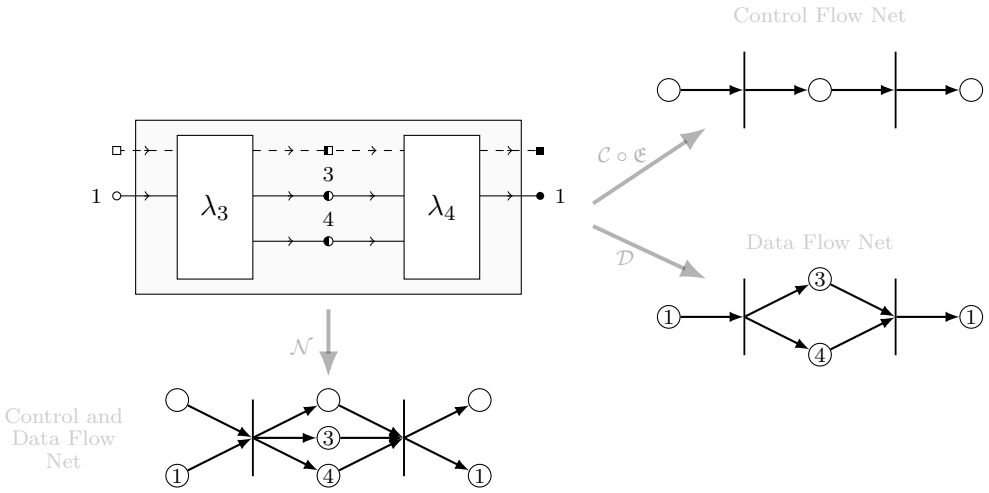


Figure 16: Sequential control flow and sequential data flow encapsulated by the total sequential computon from Figure 14(a). We label some places for mapping purposes even though Petri nets are not labelled (see Section 3).

We know a partial sequential computon $\lambda_1 \rhd_\rho \lambda_2$ is the pushout of a partially sequentiable span $\rho := \lambda_1 \xleftarrow{\alpha_1} \lambda_0 \xrightarrow{\alpha_2} \lambda_2$ of computon morphisms. By Definition 26, some of the e-outports of λ_1 are matched with some of the e-inports of λ_2, with the characteristic there always is at least one ec-outport in C_1^- identified with some ec-inport in C_2^+ (because the apex λ_0 is

necessarily a trivial computon with at least one ec-inoutport — see Definition 1). For this reason, $\lambda_1 \rhd_\rho \lambda_2$ also encapsulates sequential control flow and up to partial sequential data flow. By partial, we mean some data elements are passed from the left operand to the right one. To give a concrete example, Figure 17 shows the encapsulation given by the partial sequential computon that results from the pushout construction depicted in Figure 12.

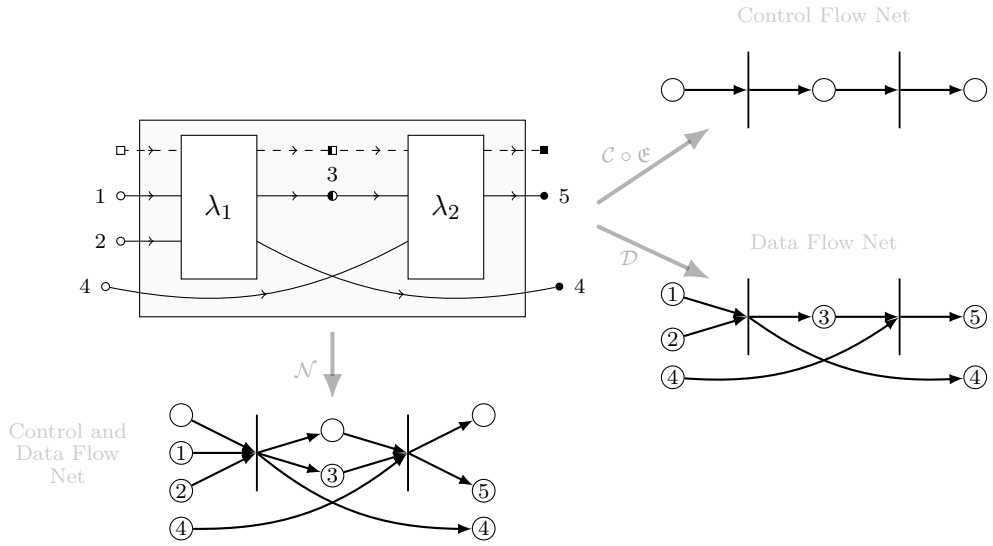


Figure 17: Sequential control flow and partial sequential data flow encapsulated by the partial sequential computon from Figure 12. We label some places for mapping purposes even though Petri nets are not labelled (see Section 3).

In Figure 17, it is easy to observe that not all data from λ_1 is passed to λ_2, even though control flow is necessarily sequential. In other words, data does not always follow control within a partial sequential computon, as a result of identifying some ed-outports of the left operand with some ed-inports of the right one. For example, in our scenario, only a 3-coloured data item is passed from λ_1 to λ_2, whilst λ_2/λ_1 consumes/produces a 4-coloured data item from/for the external environment.

6.2. Parallel Computons

In this subsection, we present two major classes of parallel computons, *p-async* and *p-sync*, which allow the asynchronous and synchronous execution of connected computons, respectively. These two classes serve to capture parallel processes that do not interfere with one another.

6.2.1. Asynchronous Parallel Computons

A *p-async computon* intutitively permits the independent, simultaneous execution of two connected computons, without the need of forking or synchronizing control. Its formal notion is given in Definition 29.

Definition 29. A p-async computon is the coproduct of two connected computons.

Definition 29 implies that a p-async composite puts two connected computons side by side by offering multiple ec-inports to trigger some control-driven computation concurrently. Figure 18 depicts a self-descriptive example for the construction of a p-async computon in which the connected computons being put in parallel are the same we used in Figure 12. This example demonstrates a particular feature of our theory, which is to allow the composition of the same connected computons into sequential or p-async composite structures, no matter the data such computons require or produce. Theorem 2 generalises this assertion by stating that two arbitrary connected computons are sufficient and necessary to form a p-async computon which, by Proposition 41, is always connected. As per Propositions 42 and 43, the operation for forming a p-async computon is both commutative and associative.

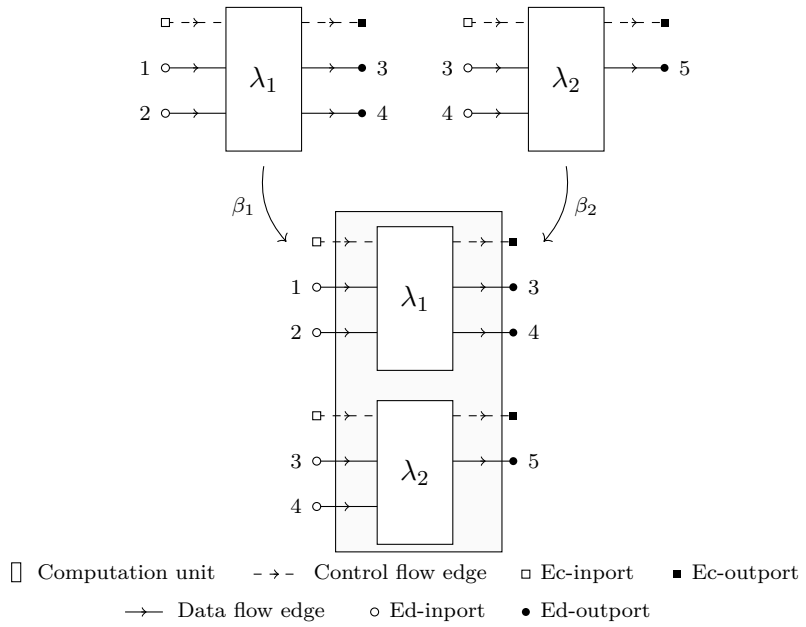


Figure 18: Constructing a p-async computon $\lambda_1 + \lambda_2$ where λ_1 and λ_2 are isomorphic to the respective left and right operands presented in Figure 12.

Theorem 2. λ_1 and λ_2 are connected computons $\iff$ the p-async computon $\lambda_1 + \lambda_2$ exists.

Proof. The implication and reverse implication follow from Proposition 10 and Definition 29, respectively. □

Proposition 41. A p-async computon is a connected computon.

Proof. The proof trivially follows from Definition 29 and Proposition 11. □

Proposition 42 (Asynchronous parallel composition is commutative)**.** There is an isomorphism between $\lambda_1 + \lambda_2$ and $\lambda_2 + \lambda_1$ for any (connected) computons λ_1 and λ_2.

Proof. The proof follows directly from the well-known fact that categorical coproduct is commutative up to unique isomorphism. □

Proposition 43 (Asynchronous parallel composition is associative)**.** There is an isomorphism between $(\lambda_1 + \lambda_2) + \lambda_3$ and $\lambda_1 + (\lambda_2 + \lambda_3)$ for any (connected) computons λ_1, λ_2 and λ_3.

Proof. The proof follows directly from the well-known fact that categorical coproduct is associative up to unique isomorphism. □

6.2.2. Operational semantics for p-async computons (in the theory of Petri nets)

No matter whether we use any of the three functors from Section 3, the Petri net of a p-async computon does not introduce any additional places or transitions and the nets of the operands do not interact in any way. This structural organisation, depicted in Figure 19, results from defining a p-async computon in the form of a coproduct construction. By Proposition 44 and Remark 13, any p-async's net is deadlock free when the nets of the composed computons are too.

Proposition 44. If $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock-free, $\mathcal{N}(\lambda_1 + \lambda_2)$ is deadlock-free.

Proof. If the net from Figure 19 corresponds to $\mathcal{N}(\lambda_1 + \lambda_2)$, Definition 16 says the initial state M_i of such a net is a marking function where $M_i(p) > 0$ for all $p \in \{p_1, \ldots, p_n, r_1, \ldots, r_m\}$ and no tokens for all the other places, including those inside $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$. As this marking evidently reaches states from $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ in parallel, we have that $\mathcal{N}(\lambda_1 + \lambda_2)$ is deadlock-free whenever $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are. □

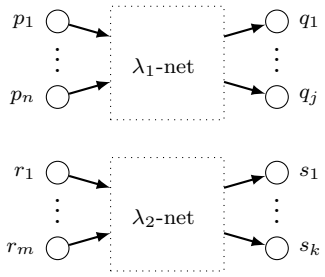


Figure 19: General structure of the Petri net of a p-async computon $\lambda_1 + \lambda_2$ constructed from a connected computon λ_1 with n e-inports and j e-outports, and a connected computon λ_2 with m e-inports and k e-outports. This structure is applicable to all the functorial constructions presented in Section 3, namely $\mathcal{N}$, $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$.

Remark 13. Although it is a statement about $\mathcal{N}$, Proposition 44 is applicable to the functors $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$ presented in Section 3. The proof is valid for $\mathcal{C} \circ \mathfrak{E}$ since Proposition 16 says $\mathcal{C}$ is just a restriction of $\mathcal{N}$ to $\mathfrak{E}(\mathbf{Set}^{\mathbf{Comp}})$.

For $\mathcal{D}$, we are only interested in nets with initial and final states (see Remark 5). Thus, considering the form depicted in Figure 19, we have the following cases for a net $\mathcal{D}(\lambda_1 + \lambda_2)$:

1. If $m = 0$ and $n > 0$, or $m > 0$ and $n = 0$, only states of $\mathcal{D}(\lambda_1)$ or $\mathcal{D}(\lambda_2)$ are reached from M_i. Since both nets are deadlock-free, $\mathcal{D}(\lambda_1 + \lambda_2)$ is deadlock-free.
2. If $m > 0$ and $n > 0$, states of $\mathcal{D}(\lambda_1)$ and $\mathcal{D}(\lambda_2)$ are simultaneously reached from M_i. As both nets are deadlock-free, $\mathcal{D}(\lambda_1 + \lambda_2)$ is deadlock-free.

6.2.3. Encapsulation of control flow and data flow in p-async computons

By Definition 29, we know a p-async computon results from a coproduct construction built upon disjoint union. Consequently, there are no ports of one computon identified with ports of the other, meaning there is no way of structurally exchanging either data or control. For this reason, a p-async computon encapsulates asynchronous parallel control flow and up to asynchronous parallel data flow. To give a concrete example, Figure 20 shows the encapsulation given by the p-async computon from Figure 18.

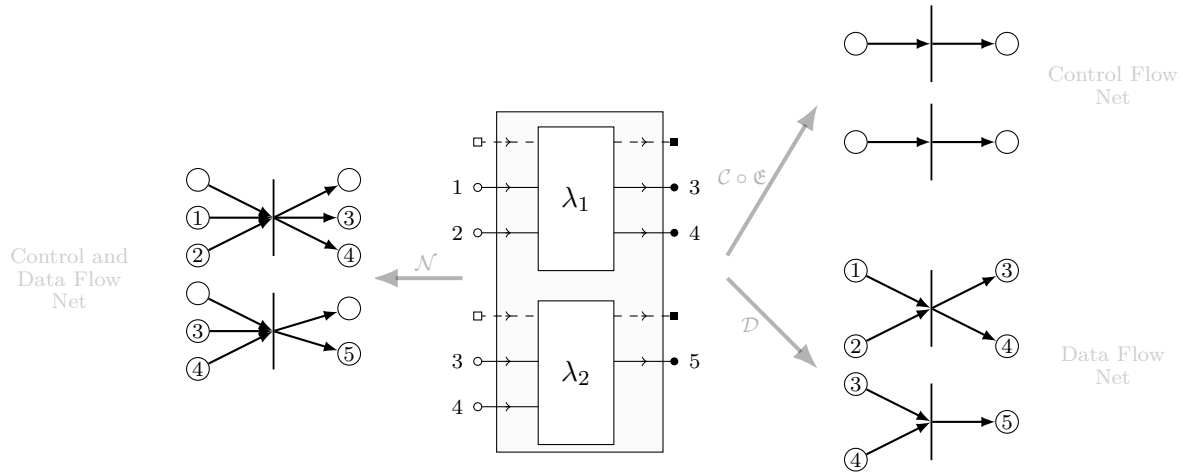


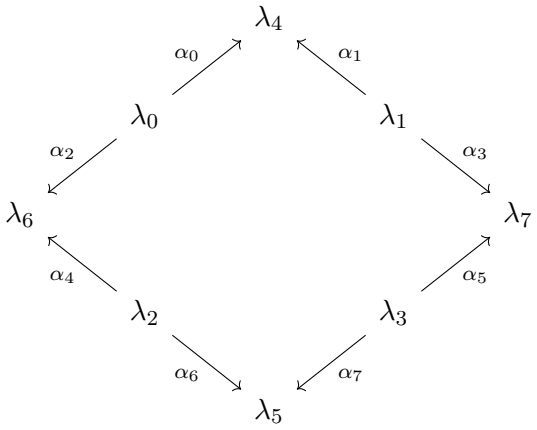
Figure 20: Asynchronous parallel control flow and asynchronous parallel data flow encapsulated by the p-async computon from Figure 18. We label some places for mapping purposes even though Petri nets are not labelled (see Section 3).

6.2.4. Synchronous Parallel Computons

Structurally, a *p-sync computon* consists of a fork computon, two arbitrary connected computons and a join computon. The roles of the fork and join are to split into and synchronise control from the connected computons, respectively. Formally, a p-sync computon is

constructed from a so-called *p-diagram* which satisfies the requirements imposed by Definition 30. Such a composite computon is defined as a colimit in $\mathbf{Set}^{\mathbf{Comp}}$ which, by Lemma 2, can always be computed via coproduct and pushout constructions.

Definition 30 (P-Diagram)**.** A p-diagram ρ is a diagram with the following shape in $\mathbf{Set}^{\mathbf{Comp}}$:



where:

1. λ_0, λ_1, λ_2 and λ_3 are unit computons (see Definition 19),
2. λ_4 and λ_5 are connected computons,
3. λ_6 is a fork computon,
4. λ_7 is a join computon,
5. $\lambda_6 \xleftarrow{\alpha_2} \lambda_0 \xrightarrow{\alpha_0} \lambda_4$, $\lambda_4 \xleftarrow{\alpha_1} \lambda_1 \xrightarrow{\alpha_3} \lambda_7$, $\lambda_5 \xleftarrow{\alpha_7} \lambda_3 \xrightarrow{\alpha_5} \lambda_7$ and $\lambda_6 \xleftarrow{\alpha_4} \lambda_2 \xrightarrow{\alpha_6} \lambda_5$ are partially sequentiable spans of computon morphisms,
6. $\alpha_2(P_0) \neq \alpha_4(P_2)$ and
7. $\alpha_3(P_1) \neq \alpha_5(P_3)$.

Definition 31 (P-Sync Computon)**.** A p-sync computon is the colimit of a p-diagram.

Notation 4. For convenience, we use a pipe to reflect the fact that two connected computons are being put into a synchronous parallel structure. For example, we write $\lambda_4 \mid_\rho \lambda_5$ for the p-sync computon obtained by computing the colimit of the p-diagram ρ shown in Definition 30.

Lemma 2. A p-sync computon can always be constructed in $\mathbf{Set}^{\mathbf{Comp}}$.

Proof. Considering the p-diagram shown in Definition 30, let $(\beta_1 : \lambda_6 \to \lambda_8, \lambda_8, \beta_2 : \lambda_4 \to \lambda_8)$, $(\beta_3 : \lambda_4 \to \lambda_9, \lambda_9, \beta_4 : \lambda_7 \to \lambda_9)$, $(\beta_5 : \lambda_6 \to \lambda_{10}, \lambda_{10}, \beta_6 : \lambda_5 \to \lambda_{10})$ and $(\beta_7 : \lambda_5 \to \lambda_{11}, \lambda_{11}, \beta_8 : \lambda_7 \to \lambda_{11})$ be the respective pushouts of the partially sequentiable spans $\lambda_6 \xleftarrow{\alpha_2} \lambda_0 \xrightarrow{\alpha_0} \lambda_4$, $\lambda_4 \xleftarrow{\alpha_1} \lambda_1 \xrightarrow{\alpha_3} \lambda_7$, $\lambda_6 \xleftarrow{\alpha_4} \lambda_2 \xrightarrow{\alpha_6} \lambda_5$ and $\lambda_5 \xleftarrow{\alpha_7} \lambda_3 \xrightarrow{\alpha_5} \lambda_7$. By Definition 27 and Lemma 1, we know such pushouts can be constructed to yield partial sequential computons. For example, λ_8 corresponds to the partial sequential computon $\lambda_6 \rhd_{\rho_1} \lambda_4$ constructed from the partially sequentiable span $\rho_1 := \lambda_6 \xleftarrow{\alpha_2} \lambda_0 \xrightarrow{\alpha_0} \lambda_4$.

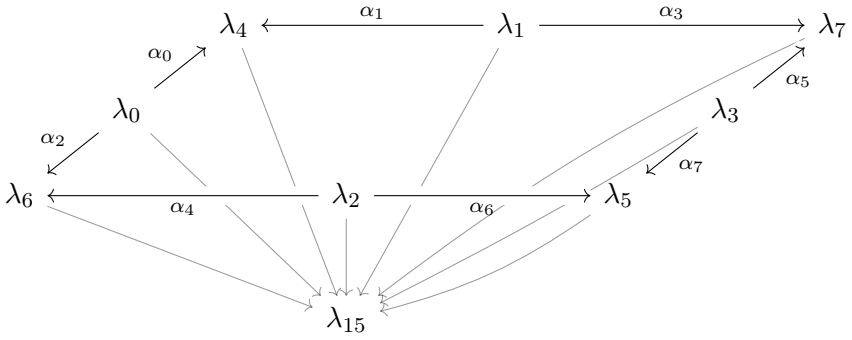
We now show that the induced span $\lambda_8 \xleftarrow{\beta_2} \lambda_4 \xrightarrow{\beta_3} \lambda_9$ is pushable. For this, let $p_8 \in \beta_2(\vec{o}(\beta_3))$ so there exists some $p_4 \in \vec{o}(\beta_3)$ where $\beta_2(p_4) = p_8$. Since $p_4 \in \vec{o}(\beta_3)$ and β_3 is induced by the pushout of the partially sequentiable span $\lambda_4 \xleftarrow{\alpha_1} \lambda_1 \xrightarrow{\alpha_3} \lambda_7$, we have $\alpha_1(p_1) = p_4 \in P_4^-$ for the unique port $p_1 \in P_1$ (recall λ_1 is a unit computon and λ_4 is an arbitary connected computon — see Conditions 1 and 2 of Definition 30). By Proposition 30, $\beta_2(P_4^-) \subseteq P_8^-$ because λ_8 is the partial sequential computon $\lambda_6 \rhd_{\rho_1} \lambda_4$. So, $\beta_2(p_4) = p_8 \in P_8^-$.

As the other conditions of Definition 9 follow analogously, we have that the pushout $(\beta_9 : \lambda_8 \to \lambda_{12}, \lambda_{12}, \beta_{10} : \lambda_9 \to \lambda_{12})$ of $\lambda_8 \xleftarrow{\beta_2} \lambda_4 \xrightarrow{\beta_3} \lambda_9$ can be constructed. A similar approach

can be used to prove the existence of the pushout $(\beta_{11} : \lambda_{10} \to \lambda_{13}, \lambda_{13}, \beta_{12} : \lambda_{11} \to \lambda_{13})$ of the induced span $\lambda_{10} \xleftarrow{\beta_6} \lambda_5 \xrightarrow{\beta_7} \lambda_{11}$.

Now, Proposition 10 says that $\lambda_6 + \lambda_7$ can be formed. By the universal property of coproducts, we deduce there are unique computon morphisms $(\beta_9 \circ \beta_1, \beta_{10} \circ \beta_4) : \lambda_6 + \lambda_7 \to \lambda_{12}$ and $(\beta_{11} \circ \beta_5, \beta_{12} \circ \beta_8) : \lambda_6 + \lambda_7 \to \lambda_{13}$. Considering Conditions 6 and 7 of Definition 30, it is routine to check that the pushout λ_{14} of $\lambda_{13} \xleftarrow{(\beta_{11} \circ \beta_5, \beta_{12} \circ \beta_8)} \lambda_6 + \lambda_7 \xrightarrow{(\beta_9 \circ \beta_1, \beta_{10} \circ \beta_4)} \lambda_{12}$ can be constructed.

To check that λ_{14} is indeed the colimit of the p-diagram shown in Definition 30, consider the following cone:



Since λ_8, λ_9, λ_{10} and λ_{11} are pushouts of the spans of the original p-diagram, by the universal property of pushouts, it is true that there are unique computon morphisms $\lambda_8 \to \lambda_{15}$, $\lambda_9 \to \lambda_{15}$, $\lambda_{10} \to \lambda_{15}$ and $\lambda_{11} \to \lambda_{15}$ that make the corresponding diagram commute. Likewise, as such morphisms exist, there also are unique computon morphisms from the respective pushouts of $\lambda_8 \xleftarrow{\beta_2} \lambda_4 \xrightarrow{\beta_3} \lambda_9$ and $\lambda_{10} \xleftarrow{\beta_6} \lambda_5 \xrightarrow{\beta_7} \lambda_{11}$. That is, $\lambda_{12} \to \lambda_{15}$ and $\lambda_{13} \to \lambda_{15}$ exist.

In the above cone, it is clear there are morphisms $\lambda_6 \to \lambda_{15}$ and $\lambda_7 \to \lambda_{15}$. Using the universal property of coproducts, we deduce the existence of a unique computon morphism $\lambda_6 + \lambda_7 \to \lambda_{15}$. Finally, as $\lambda_{12} \to \lambda_{15}$ and $\lambda_{13} \to \lambda_{15}$ exist, we use the universal property of pushouts again to deduce there is a unique computon morphism $\lambda_{14} \to \lambda_{15}$. As $\lambda_{14} \to \lambda_{15}$ makes everything commute in our construction, it is true that λ_{14} is the colimit of the original p-diagram. □

Corollary 3. A p-sync computon is a connected computon.

Proof. Consider the construction presented in the proof of Lemma 2. By Proposition 9, we have that λ_8, λ_9, λ_{10} and λ_{11} are connected computons because λ_4, λ_5, λ_6 and λ_7 also are (recall forks and joins are primitive computons which, by Proposition 26, adhere to Definition 6). Consequently, the pushouts λ_{12} and λ_{13} (of the induced spans $\lambda_8 \xleftarrow{\beta_2} \lambda_4 \xrightarrow{\beta_3} \lambda_9$ and $\lambda_{10} \xleftarrow{\beta_6} \lambda_5 \xrightarrow{\beta_7} \lambda_{11}$, respectively) are connected computons too. Using Proposition 9 again, we deduce that the pushout λ_{14} of the unique span $\lambda_{13} \xleftarrow{(\beta_{11} \circ \beta_5, \beta_{12} \circ \beta_8)} \lambda_6 + \lambda_7 \xrightarrow{(\beta_9 \circ \beta_1, \beta_{10} \circ \beta_4)} \lambda_{12}$ is a connected computon. As λ_{14} is the colimit of the original p-diagram (shown in Definition 30), we conclude that every p-sync computon is a connected computon. □

To elucidate the proof of Lemma 2, Figure 21 provides a complete, self-descriptive example for constructing a p-sync computon from the connected computons used as operands in one of our examples of partial sequential composition (see Figure 12). A glance at Figure 21 reveals that the initial p-diagram ρ (displayed in the middle) specifies the basic building blocks for constructing a p-sync computon, namely four unit computons, a fork computon, a join computon and two connected computons (i.e., the computons being put in parallel). The construction starts by computing four pushout operations that produce a partial sequential computon each as per Definitions 27 and 30 (see the squares marked with S). The induced morphisms of such pushouts form two spans whose respective pushouts freely behave as in Definition 8, i.e., they are pushouts that just "merge" computons via some common object (see the squares marked with M). In this case, such common objects are λ_1 and λ_2, respectively.

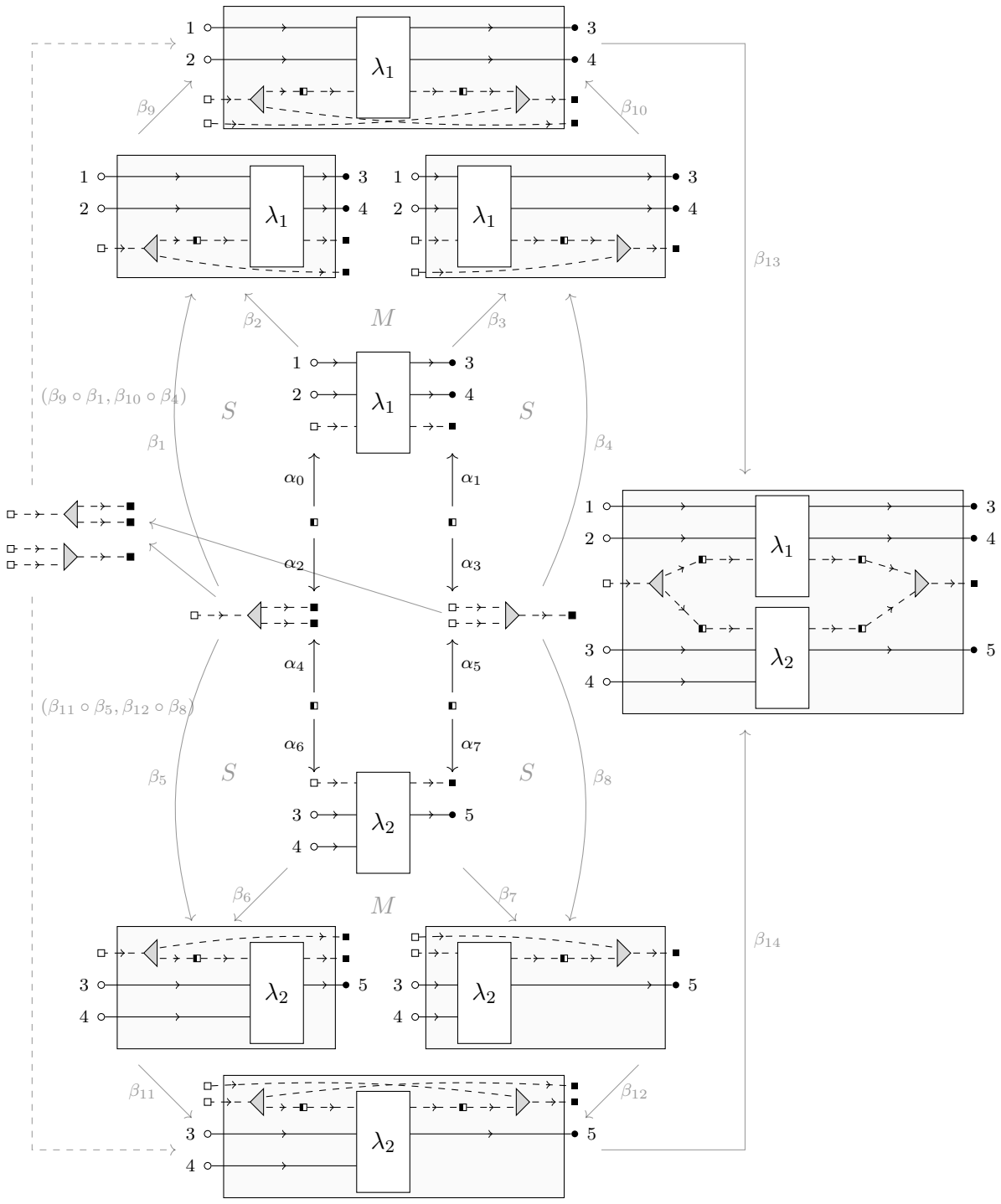


Figure 21: Constructing a p-sync computon $\lambda_1|_\rho\lambda_2$ where λ_1 and λ_2 are isomorphic to the operands presented in Figure 12 and ρ is the p-diagram in the middle (whose morphisms are displayed as black arrows). Intuitively, the pushouts marked with 'S' are first computed to form four partial sequential computons. Then, the pushouts marked with 'M' are computed to merge such partial sequential computons into the top- and bottom-level composites, respectively. Finally, the pushout of the span formed by $(\beta_{11} \circ \beta_5, \beta_{12} \circ \beta_8)$ and $(\beta_9 \circ \beta_1, \beta_{10} \circ \beta_4)$ is computed to yield the p-sync computon $\lambda_1 \mid_\rho \lambda_2$ (displayed on the rightmost part of this figure).

Our construction finalises by computing the pushout of the unique computon morphisms deduced from the universal property of coproducts. The coproduct, in this case, is the juxtaposition of a fork computon and a join computon (in fact a p-async computon), which serves as a common object for the pushout of the unique (induced) span of $(\beta_{11} \circ \beta_5, \beta_{12} \circ \beta_8)$ and $(\beta_9 \circ \beta_1, \beta_{10} \circ \beta_4)$, i.e., for constructing the p-sync computon $\lambda_1 \mid_\rho \lambda_2$ (see Notation 4).

As $\lambda_1 \mid_\rho \lambda_2$ is constructed from pushouts that rely on unit computons as apices, only an ec-inport $p_1 \in C_1^+$, an ec-outport $q_1 \in C_1^-$, an ec-inport $p_2 \in C_2^+$ and an ec-outport $q_2 \in C_2^-$

become i-ports in $\lambda_1 \mid_\rho \lambda_2$. The rest of e-inports and e-outports of the arbitrary connected computons become e-inports and e-outports in $\lambda_1 \mid_\rho \lambda_2$, respectively. This structural implication is derived from the fact that fork and join computons have control ports only; so, unlike sequential composition and like p-async computons, λ_1 and λ_2 do not have any structural means to exchange data when composed into a synchronous parallel structure. To ensure a consistent construction of the p-sync computon $\lambda_1 \mid_\rho \lambda_2$, Condition 6 of Definition 30 intuitively says that p_1 and p_2 cannot be mapped to the same ec-outport of the fork computon. A similar constraint is imposed by Condition 7 which states that q_1 and q_2 cannot be mapped to the same ec-inport of the join computon.

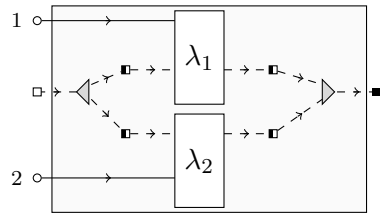
Another difference with respect to sequential composition is that the order of the computons being parallelised does not matter. So, even if λ_1 and λ_2 are interchanged in the p-diagram from Figure 21, we will always have the same colimit result, i.e., synchronous parallel composition is a commutative operation (see Proposition 45). Proposition 46 shows that, unlike total sequencing, synchronous parallel composition is not associative so that grouping matters. Although such an algebraic property is not satisfied, the result of synchronous parallel composition is always a connected computon (see Corollary 3). Also, any two connected computons can be put into a synchronous parallel structure regardless of the data they require or produce (see Theorem 3).

Proposition 45 (Synchronous parallel composition is commutative)**.** There is an isomorphism between $\lambda_1 \mid_{\rho_1} \lambda_2$ and $\lambda_2 \mid_{\rho_2} \lambda_1$ for any p-sync computons $\lambda_1 \mid_{\rho_1} \lambda_2$ and $\lambda_2 \mid_{\rho_2} \lambda_1$.

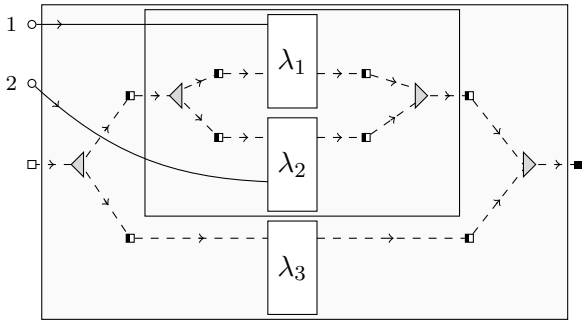
Proof. The proof is obvious. It follows from the fact that fork computons are trivially isomorphic, with the same being true for join and unit computons. □

Proposition 46 (Synchronous parallel composition is not associative)**.** There is no isomorphism between $(\lambda_1 \mid_{\rho_1} \lambda_2) \mid_{\rho_2} \lambda_3$ and $\lambda_1 \mid_{\rho_4} (\lambda_2 \mid_{\rho_3} \lambda_3)$ for some choice of p-sync computons $\lambda_1 \mid_{\rho_1} \lambda_2$, $(\lambda_1 \mid_{\rho_1} \lambda_2) \mid_{\rho_2} \lambda_3$, $\lambda_2 \mid_{\rho_3} \lambda_3$ and $\lambda_1 \mid_{\rho_4} (\lambda_2 \mid_{\rho_3} \lambda_3)$.

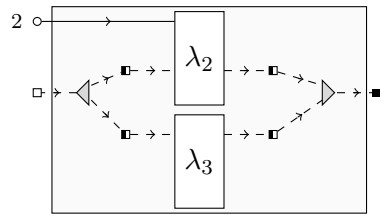
Proof. Suppose ρ_1, ρ_2, ρ_3 and ρ_4 are p-diagrams. Considering Figure 22, we let (a), (b), (c) and (d) be the colimits of ρ_1, ρ_2, ρ_3 and ρ_4, respectively. As it is clear there is no isomorphism from the p-sync computon (b) to the p-sync computon (d), we conclude that the proposition being proved is true.



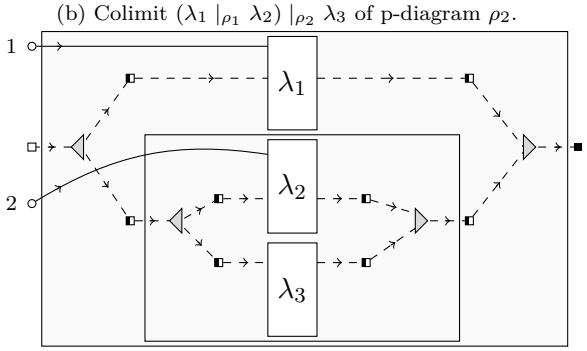
(a) Colimit $\lambda_1 \mid_{\rho_1} \lambda_2$ of p-diagram ρ_1.



(b) Colimit $(\lambda_1 \mid_{\rho_1} \lambda_2) \mid_{\rho_2} \lambda_3$ of p-diagram ρ_2.



(c) Colimit $\lambda_2 \mid_{\rho_3} \lambda_3$ of p-diagram ρ_3.



(d) Colimit $\lambda_1 \mid_{\rho_4} (\lambda_2 \mid_{\rho_3} \lambda_3)$ of p-diagram ρ_4.

Figure 22: Counterexample that disproves the associativity property of synchronous parallel composition.

□

Theorem 3. For every pair (λ_1, λ_2) of connected computons, there is a p-diagram ρ such that $\lambda_1 \mid_\rho \lambda_2$ exists.

Proof. Let λ_4 and λ_5 be two arbitrary connected computons, λ_6 a fork computon, λ_7 a join computon, and λ_j a unit computon for $j \in \{0, 1, 2, 3\}$. We first construct the following spans of computon morphisms: $\lambda_6 \xleftarrow{\alpha_2} \lambda_0 \xrightarrow{\alpha_0} \lambda_4$, $\lambda_4 \xleftarrow{\alpha_1} \lambda_1 \xrightarrow{\alpha_3} \lambda_7$, $\lambda_5 \xleftarrow{\alpha_7} \lambda_3 \xrightarrow{\alpha_5} \lambda_7$ and $\lambda_6 \xleftarrow{\alpha_4} \lambda_2 \xrightarrow{\alpha_6} \lambda_5$. As the common domain of each span is a unit computon, each morphism is a diagram of the form:

$$\begin{array}{ccccccccccccc}
1 & \longleftarrow & 1 & \longleftarrow & \emptyset & \longrightarrow & \emptyset & \longleftarrow & \emptyset & \longrightarrow & 1 & \longrightarrow & 1 \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
\Sigma & \longleftarrow & P & \longleftarrow & O & \longrightarrow & U & \longleftarrow & I & \longrightarrow & P & \longrightarrow & \Sigma
\end{array}$$

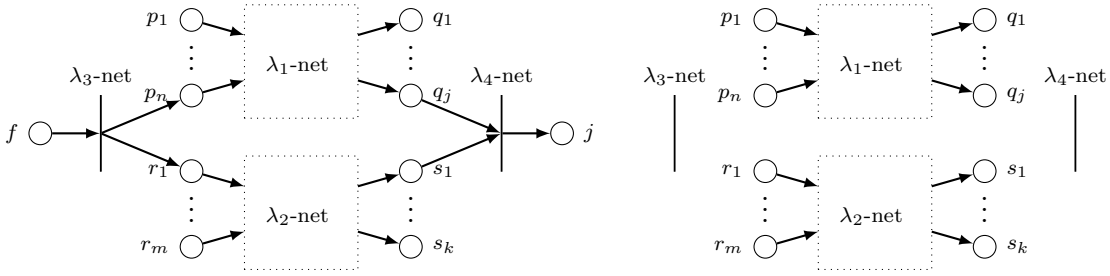
In the above diagram, it is evident that the only morphism components that are not empty functions are those mapping ports and colours, respectively. As the set of colours of a unit computon is always $\{0\}$ and 0 is in the set of colours of every computon (by Definition 1), the respective Σ-component of each morphism can be defined in the obvious way to yield an inclusion function. Now, if $p_j \in P_j$, the P-component of each morphism is given as follows: $\alpha_2(p_0) \in C_6^-$, $\alpha_0(p_0) \in C_4^+$, $\alpha_1(p_1) \in C_4^-$, $\alpha_3(p_1) \in C_7^+$, $\alpha_7(p_3) \in C_5^-$, $\alpha_5(p_3) \in C_7^+$, $\alpha_4(p_2) \in C_6^-$ and $\alpha_6(p_2) \in C_5^+$ such that $\alpha_2(p_0) \neq \alpha_4(p_2)$ and $\alpha_3(p_1) \neq \alpha_5(p_3)$.

Since λ_4 and λ_6 are connected computons and λ_0 is a trivial computon (see Definition 6 and Proposition 3), $p_0 \in \vec{i}(\alpha_2) \cap \vec{o}(\alpha_0)$ and, consequently, $P_0 = \vec{i}(\alpha_2) \cap \vec{o}(\alpha_0)$ because $P_0 = \{p_0\}$. The facts $p_0 \in \vec{i}(\alpha_2) \cap \vec{o}(\alpha_0)$, $\alpha_2(p_0) \in C_6^-$ and $\alpha_0(p_0) \in C_4^+$ allow us to further deduce $\alpha_2(\vec{o}(\alpha_0)) \subseteq C_6^- \subseteq P_6^-$ and $\alpha_0(\vec{i}(\alpha_2)) \subseteq C_4^+ \subseteq P_4^+$. In particular, $\alpha_2(\vec{o}(\alpha_0)) \subset P_6^-$ because $|P_0| = 1$ and $|P_6^-| = |C_6^-| = 2$ (see the above diagram and Definition 23). This means that, by Definition 26, $\lambda_6 \xleftarrow{\alpha_2} \lambda_0 \xrightarrow{\alpha_0} \lambda_4$ is partially sequentiable. Proving that the other spans are also partially sequentiable is completely analogous.

As our construction corresponds to a p-diagram ρ whose colimit can be computed by Lemma 2, it follows that the p-sync computon $\lambda_4 \mid_\rho \lambda_5$ exists in $\mathbf{Set}^{\mathbf{Comp}}$ (see Definition 31 and Notation 4). □

6.2.5. Operational semantics for p-sync computons (in the theory of Petri nets)

No matter whether we use any of the three functors presented in Section 3 (i.e., $\mathcal{N}$, $\mathcal{C} \circ \mathfrak{E}$ or $\mathcal{D}$), the Petri net of a p-sync computon does not introduce any additional places or transitions beyond those from the nets of the computons of the corresponding p-diagram. In the case of $\mathcal{N}$ and $\mathcal{C} \circ \mathfrak{E}$, the net of a p-sync has the form depicted in Figure 23(a); whereas for $\mathcal{D}$, the corresponding net has the form depicted in Figure 23(b).



(a) For $\mathcal{N} \circ \mathfrak{E}$ or $\mathcal{C}$.

(b) For $\mathcal{D}$, all places buffer data so the transitions for the fork and join computons do not have any input or output places (cf. Figures 9(c) and 10(c)).

Figure 23: General structure of the Petri net of a p-sync computon $\lambda_1 \mid_\rho \lambda_2$ constructed from a connected computon λ_1 with n e-inports and j e-outports, and a connected computon λ_2 with m e-inports and k e-outports. The places f and j correspond to the only ec-inport and the only ec-outport of the fork and join computons, respectively.

By Proposition 47 and Remark 14, the underlying net of any p-sync computon is deadlock-free only if the nets of the composed computons are deadlock-free too.

Proposition 47. A Petri net $\mathcal{N}(\lambda_1 \mid_\rho \lambda_2)$ is deadlock-free if $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock-free, and the fork transition of $\mathcal{N}(\lambda_1 \mid_\rho \lambda_2)$ is the only transition enabled in the initial state of $\mathcal{N}(\lambda_1 \mid_\rho \lambda_2)$.

Proof. If the net from Figure 23(a) corresponds to $\mathcal{N}(\lambda_1 \mid_\rho \lambda_2)$, we know by Definition 16 that its initial state M_i is a marking function that puts tokens in f together with the input places of $\mathcal{N}(\lambda_1)$ different than p_n and the input places of $\mathcal{N}(\lambda_2)$ different than r_1, while keeping no tokens in all the other places, including those inside $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$. If the unique transition of $\mathcal{N}(\lambda_3)$ is the only one enabled under M_i, firing it reaches a marking that corresponds to the initial states of $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$, i.e., only the places $p_1, \ldots, p_n$ and $r_1, \ldots, r_m$ have tokens. Thus, if $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock free, $\mathcal{N}(\lambda_1 \mid_\rho \lambda_2)$ must be deadlock-free too, considering it is evident that the places and transitions of $\mathcal{N}(\lambda_4)$ do not introduce any deadlocks. □

Remark 14. Although it is a statement about the functor $\mathcal{N}$, Proposition 47 is applicable to the functors $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$ presented in Section 3. The proof is valid for $\mathcal{C} \circ \mathfrak{E}$ since Proposition 16 says $\mathcal{C}$ is just a restriction of $\mathcal{N}$ to $\mathfrak{E}(\mathbf{Set}^{\mathbf{Comp}})$. As $\mathcal{D}(\lambda_1 \mid_\rho \lambda_2)$ has isolated transitions for fork and join computons (see Figure 23(b)), deadlock-freedom follows directly from Remark 13.

By restricting the fork transition of $\mathcal{N}(\lambda_1 \mid_\rho \lambda_2)$ to be the only one enabled under M_i, Proposition 47 disregards the possibility of triggering either the λ_1-net or the λ_2-net before/upon firing the unique transition of the λ_3-net. Should any of these two cases occur, there is a possibility of deadlock.

6.2.6. Encapsulation of control flow and data flow in p-sync computons

By Definition 31, we know a p-sync computon $\lambda_1 \mid_\rho \lambda_2$ is the colimit of a p-diagram ρ which, by Definition 30, integrates a fork computon, a join computon, two connected computons and four unit computons. As a result of the colimit construction described in Lemma 2, a p-sync computon connects the ec-outports of the fork and the ec-inports of the join with ec-inports and ec-outports of the connected computons, respectively. Thus, forming a composite that encapsulates synchronous parallel control flow. It is synchronous in the sense the fork enables the parallel invocation of λ_1 and λ_2, while the join waits for their termination. As no data ports are linked (through the colimit of ρ), there is no data exchange within a p-sync computon so $\lambda_1 \mid_\rho \lambda_2$ encapsulates up to asynchronous parallel data flow (just as p-async computons do). To give a concrete example, Figure 24 shows the encapsulation given by the p-sync computon resulting from the colimit construction depicted in Figure 21.

Having synchronous control and asynchronous data entails that data does not follow control within a p-sync computon. So, data items can be received before forking control or produced before joining control. Despite of this asynchronous behaviour, the connected computons being paralleised cannot consume data until receiving a control signal from the corresponding fork computon. This is enforced in nets under $\mathcal{N}$. For example, in the scenario depicted in Figure 24, $\mathcal{N}(\lambda_1)$ cannot perform any computation until receiving control from the fork as well as 1- and 2-coloured data items from the external environment. In other words, the transition representing λ_1 needs to be enabled by both the transition representing the fork computon and the external environment.

As computons interact via input/output ports, it might seem that the computon model is not yet ready to encode synchronised concurrent behaviours (cf. open automata [35]). We conjecture that an instance of such class of behaviours can be expressed as a p-sync computon to simultaneously trigger computons that encode the transitions being synchronised. Although constructing such composites might be infeasible in practice due to their potentially complex structure, proving this conjecture would support our thesis that control flow is ever present in any (concurrent or sequential) computation.

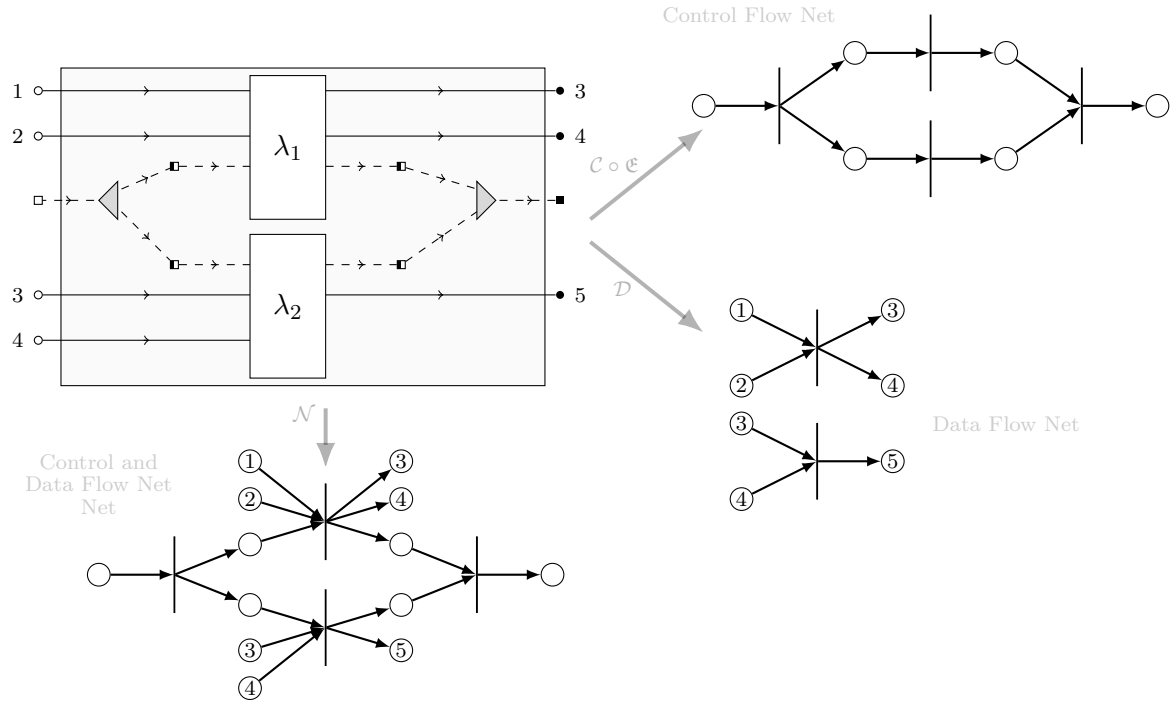
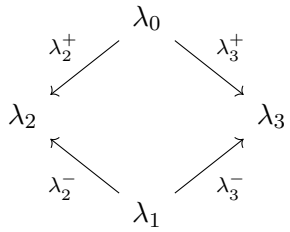


Figure 24: Synchronous parallel control flow and asynchronous parallel data flow encapsulated by the p-sync computon from Figure 21. We label some places for mapping purposes even though Petri nets are not labelled (see Section 3).

6.3. Branching Computons

A *branching computon* structurally consists of two connected computons whose e-inports and e-outports overlap, respectively. This overlapping restriction is captured by a so-called *b-diagram* whose morphisms are all markers (see Definition 32).

Definition 32 (B-Diagram)**.** A b-diagram is a diagram with the following shape in $\mathbf{Set}^{\mathbf{Comp}}$:



where:

1. λ_2 is a connected computon with an in-marker λ_2^+ and an out-marker λ_2^-, and
2. λ_3 is a connected computon with an in-marker λ_3^+ and an out-marker λ_3^-.

Evidently, by Definition 20, λ_0 and λ_1 are trivial computons, serving as domains for the markers involved in the b-diagram.

A branching computon operates in a non-deterministic manner by exclusively choosing a connected computon out of two possible ones. Its role is to capture the essential structure of decision-making by abstracting away from particular conditions or external environment influences which might be variable, unpredictable or too complex to enumerate exhaustively. Intuitively, a branching computon is like an *if-else* programming construct but without explicit conditions deciding computational paths.[20]

[20]Another reason for decoupling control flow from external influences is to facilitate formal verification of

To construct a branching composite, it suffices to compute the colimit of a b-diagram by performing a pushout operation along a coproduct construction. More specifically, if we consider the b-diagram from Definition 32, the operation is $\lambda_2 +_{\lambda_0+\lambda_1} \lambda_3$ such that λ_2 and λ_3 are the computons being branched (see Definition 33). As the apex of such a pushout is the coproduct of a trivial computon λ_0 (which can be injected into all the e-inports of both operands) and a trivial computon λ_1 (which can be identified with all the e-outports of both operands), a branching structure can be constructed only from computons with isomorphic interfaces. By Lemma 3, this pushout construction can always be computed in $\mathbf{Set}^{\mathbf{Comp}}$.

Definition 33 (Branching Computon)**.** A branching computon is the colimit of a b-diagram.

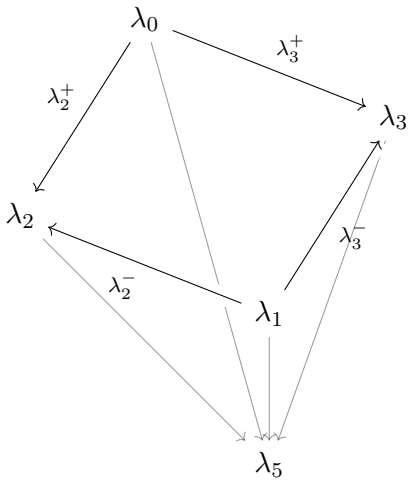
Notation 5. For convenience, we use a question mark to reflect the fact that computons are chosen non-deterministically. For example, we write $\lambda_2 ?_\rho \lambda_3$ for the colimit of the b-diagram ρ shown in Definition 32.

Lemma 3. A branching computon can always be constructed in $\mathbf{Set}^{\mathbf{Comp}}$.

Proof. Consider the b-diagram shown in Definition 32. By Proposition 10, we know that the coproduct $\lambda_0 + \lambda_1$ can be formed. As there are markers $\lambda_j^+ : \lambda_0 \to \lambda_j$ and $\lambda_j^- : \lambda_1 \to \lambda_j$ for $j \in \{2, 3\}$, we use the universal property of coproducts to deduce there also are unique computon morphisms $(\lambda_2^+, \lambda_2^-) : \lambda_0 + \lambda_1 \to \lambda_2$ and $(\lambda_3^+, \lambda_3^-) : \lambda_0 + \lambda_1 \to \lambda_3$. Assuming $\beta_1 : \lambda_0 \to \lambda_0 + \lambda_1$ and $\beta_2 : \lambda_1 \to \lambda_0 + \lambda_1$ are the canonical injections into $\lambda_0 + \lambda_1$, we now prove that the induced span $\lambda_2 \xleftarrow{(\lambda_2^+, \lambda_2^-)} \lambda_0 + \lambda_1 \xrightarrow{(\lambda_3^+, \lambda_3^-)} \lambda_3$ is pushable.

If $p_2 \in (\lambda_2^+, \lambda_2^-)(\vec{i}(\lambda_3^+, \lambda_3^-))$, there is some $q \in \vec{i}(\lambda_3^+, \lambda_3^-)$ for which $(\lambda_2^+, \lambda_2^-)(q) = p_2$, i.e., $\bullet(\lambda_3^+, \lambda_3^-)(q) \setminus (\lambda_3^+, \lambda_3^-)(\bullet q) \neq \emptyset$ which implies $(\lambda_3^+, \lambda_3^-)(q) \notin P_3^+$. Consequently, $(\lambda_3^+, \lambda_3^-)(q)$ is not in the image of the in-marker λ_3^+ so, by coproduct definition, there must be some $p_1 \in \vec{i}(\lambda_3^-)$ such that $\lambda_3^-(p_1) = (\lambda_3^+, \lambda_3^-)(q) \in P_3^-$. As $\lambda_3^- = (\lambda_3^+, \lambda_3^-) \circ \beta_2$ (by coproduct commutativity), we have $\lambda_3^-(p_1) = (\lambda_3^+, \lambda_3^-)(q) = (\lambda_3^+, \lambda_3^-)(\beta_2(p_1))$. That is, $q = \beta_2(p_1)$ given that the P-component of $(\lambda_3^+, \lambda_3^-)$ is injective. Injectivity follows from the fact that λ_3^+ and λ_3^- are both injective (see Definition 20) and that $P_3^+ \cap P_3^- = \emptyset$ because λ_3 is a connected computon (see Definition 32 and Proposition 2).

Now, observing $\lambda_2^- = (\lambda_2^+, \lambda_2^-) \circ \beta_2$ and considering that λ_2^- is an out-marker, we obtain $\lambda_2^-(p_1) = (\lambda_2^+, \lambda_2^-)(\beta_2(p_1)) = (\lambda_2^+, \lambda_2^-)(q) = p_2 \in P_2^-$. Hence, $(\lambda_2^+, \lambda_2^-)(\vec{i}(\lambda_3^+, \lambda_3^-)) \subseteq P_2^- \subseteq P_2^- \cup P_2^+$. As the other conditions of Definition 9 can be proved analogously, the pushout $(\beta_3 : \lambda_2 \to \lambda_4, \lambda_4, \beta_4 : \lambda_3 \to \lambda_4)$ of $\lambda_2 \xleftarrow{(\lambda_2^+, \lambda_2^-)} \lambda_0 + \lambda_1 \xrightarrow{(\lambda_3^+, \lambda_3^-)} \lambda_3$ can be constructed. To prove λ_4 is the colimit of the original b-diagram, suppose there is a cone:



crucial control flow properties such as deadlock absence. Without decoupling, a verification problem could become intractable since, in general, the external environment cannot be modelled completely. Even when assumptions are made, they might not capture the real-world accurately [7].

As there evidently are morphisms $\lambda_0 \to \lambda_5$ and $\lambda_1 \to \lambda_5$, we use the universal property of coproducts to deduce there is a unique computon morphism $\lambda_0 + \lambda_1 \to \lambda_5$ such that the corresponding diagram commutes. As there also are computon morphisms $\lambda_2 \to \lambda_5$ and $\lambda_3 \to \lambda_5$, we use the universal property of pushouts to deduce there is a unique morphism $\lambda_4 \to \lambda_5$ that makes everything commute in our construction. Therefore, λ_4 is the colimit of the original b-diagram. □

Corollary 4. A branching computon is a connected computon.

Proof. As a branching computon is the pushout of a pushable span whose legs are connected computons (see the proof of Lemma 3 and Definition 32), we simply use Proposition 9 to deduce that any branching computon is a connected computon. □

To clarify the construction presented in the proof of Lemma 3, Figure 25 presents a complete, self-descriptive example for the construction of a branching computon $\lambda_1 ?_\rho \lambda_2$ where ρ is the b-diagram displayed at the top. Particularly, for $j \in \{1, 2\}$, λ_j is a computon with an in-marker λ_j^+ and an out-marker λ_j^- and the computon morphisms β_1 and β_2 are canonical injections satisfying the universal property of coproducts. By this property, there are unique computon morphisms $(\lambda_1^+, \lambda_1^-)$ and $(\lambda_2^+, \lambda_2^-)$ whose pushout yields the branching computon $\lambda_1 ?_\rho \lambda_2$ together with induced morphisms γ_1 and γ_2.

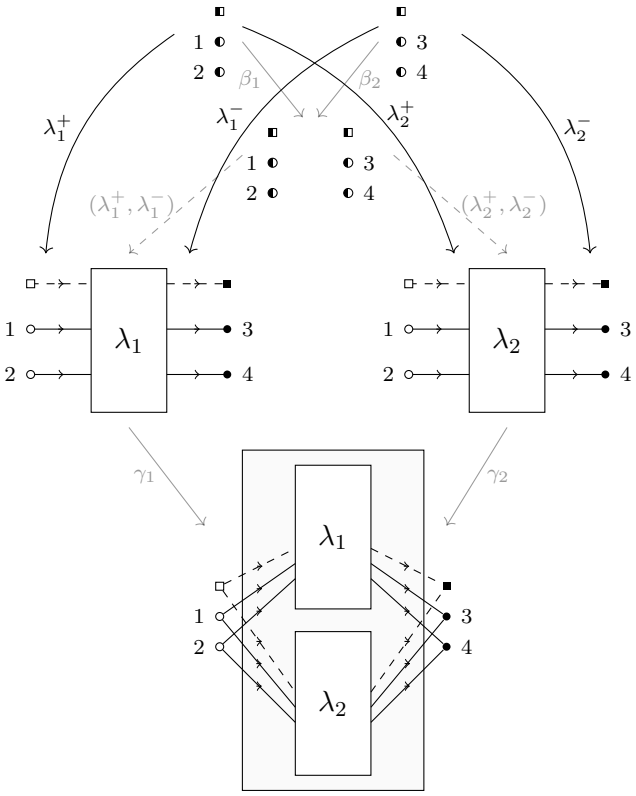


Figure 25: Constructing a branching computon $\lambda_1 ?_\rho \lambda_2$ where ρ is the b-diagram at the top (whose morphisms are displayed as black arrows). Here, λ_1 and λ_2 are both isomorphic to the left operand presented in the example of Figure 12, and the β-morphisms are the canonical injections into the coproduct of the trivial computons in ρ. The branching computon $\lambda_1 ?_\rho \lambda_2$ is simply the pushout of the unique morphisms induced from the universal property of coproducts, namely $(\lambda_1^+, \lambda_1^-)$ and $(\lambda_2^+, \lambda_2^-)$.

Branching is an operation that enables the non-deterministic selection of a computon out of two possible ones. So, even if we interchange λ_1 and λ_2 in the construction depicted in Figure 25, the colimit would be isomorphic, i.e., constructing a branching computon is a commutative operation (see Proposition 48). As grouping does not alter the colimit result either, branching is associative in addition (see Proposition 49).

Proposition 48 (Branching composition is commutative)**.** There is an isomorphism between $\lambda_1 ?_{\rho_1} \lambda_2$ and $\lambda_2 ?_{\rho_2} \lambda_1$ for any branching computons $\lambda_1 ?_{\rho_1} \lambda_2$ and $\lambda_2 ?_{\rho_2} \lambda_1$.

Proof. The proof follows directly from the well-known fact that categorical pushout is commutative up to unique isomorphism. □

Proposition 49 (Branching composition is associative)**.** There is an isomorphism between $(\lambda_1 ?_{\rho_1} \lambda_2) ?_{\rho_2} \lambda_3$ and $\lambda_1 ?_{\rho_4} (\lambda_2 ?_{\rho_3} \lambda_3)$ for any branching computons $\lambda_1 ?_{\rho_1} \lambda_2$, $(\lambda_1 ?_{\rho_1} \lambda_2) ?_{\rho_2} \lambda_3$, $\lambda_2 ?_{\rho_3} \lambda_3$ and $\lambda_1 ?_{\rho_4} (\lambda_2 ?_{\rho_3} \lambda_3)$.

Proof. The proof is similar to that of Proposition 32. □

Unfortunately, not every pair of connected computons is a candidate to define a branching composite. This is because the e-inports of one computon must totally match the e-inports of the other, with the same being true for e-outports (hence the retrictions imposed by the morphisms of a b-diagram — see Definition 32). Nevertheless, when a pair of computons meets such restrictions, Corollary 4 states that their corresponding branching composite is always a connected computon.

6.3.1. Operational semantics for branching computons (in the theory of Petri nets)

No matter whether we use any of the three functorial constructions presented in Section 3, the underlying Petri net of a branching computon does not have any additional places or transitions beyond those from the composed computon nets. The general structure of a branching computon's net is depicted in Figure 26.

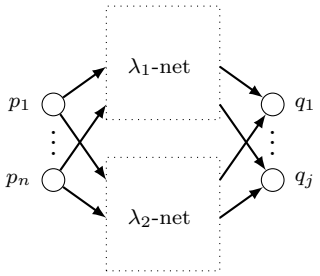


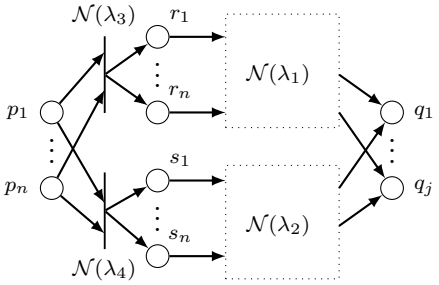
Figure 26: General structure of the Petri net of a branching computon $\lambda_1 ?_\rho \lambda_2$ constructed from connected computons λ_1 and λ_2 that have n e-inports and j e-outports each. This structure is applicable to all the functorial constructions from Section 3, namely $\mathcal{N}$, $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$.

Unfortunately, a net $\mathcal{N}(\lambda_1 ?_\rho \lambda_2)$ is not deadlock-free in general, especially when λ_1 or λ_2 rely on partial sequencing. To fully ensure deadlock-freedom, the composed computons need to be "adapted" by attaching an in-sync to each of them; thereby, forming total sequential computons that synchronise e-inports. These sequential entities can then composed into a branching structure which is guaranteed to be deadlock-free, provided that $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock-free too. Proposition 50 uses this adaptation process to show that it is possible to convert the net of any branching computon into a deadlock-free one.

Proposition 50. Let $\lambda_1 ?_\rho \lambda_2$ be a branching computon and assume λ_3 and λ_4 are in-syncs of λ_1 and λ_2, respectively. If $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are deadlock-free, there are spans ρ_1, ρ_2 and ρ_3 such that $\mathcal{N}((\lambda_3 \trianglerighteq_{\rho_1} \lambda_1) ?_{\rho_3} (\lambda_4 \trianglerighteq_{\rho_2} \lambda_2))$ is deadlock-free.

Proof. If $\lambda_1 ?_\rho \lambda_2$ is a branching computon, we know by Definition 32 that λ_1 and λ_2 are connected computons. If λ_3 and λ_4 are in-syncs of λ_1 and λ_2, respectively, Proposition 35 says there must be spans ρ_1 and ρ_2 such that $\lambda_3 \trianglerighteq_{\rho_1} \lambda_1$ and $\lambda_4 \trianglerighteq_{\rho_2} \lambda_2$ exist. By Proposition 31 and Definition 28, we deduce that the in-markers of $\lambda_3 \trianglerighteq_{\rho_1} \lambda_1$ and $\lambda_4 \trianglerighteq_{\rho_2} \lambda_2$ share domain with λ_1^+ and λ_2^+, respectively. Similarly, using Proposition 31, we further derive that the

out-markers of $\lambda_3 \trianglerighteq_{\rho_1} \lambda_1$ and $\lambda_4 \trianglerighteq_{\rho_2} \lambda_2$ share domain with λ_1^- and λ_2^-, respectively. Considering that the domains of λ_1^+ and λ_2^+ coincide in the b-diagram ρ (as per Definition 32) and that sequential computons are connected (see Proposition 29), we have just constructed a new b-diagram ρ_3 out of ρ. Simply using Lemma 3, we deduce $(\lambda_3 \trianglerighteq_{\rho_1} \lambda_1)?_{\rho_3}(\lambda_4 \trianglerighteq_{\rho_2} \lambda_2)$ exists. According to the functorial construction presented in Definition 15, we know the net $\mathcal{N}((\lambda_3 \trianglerighteq_{\rho_1} \lambda_1)?_{\rho_3}(\lambda_4 \trianglerighteq_{\rho_2} \lambda_2))$ must have the following form:



Definition 16 says that the initial state M_i of the above net is a marking function where $M_i(p) > 0$ for all $p \in \{p_1, \ldots, p_n\}$ and no tokens in all the other places, including those inside $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$. Although M_i enables the only transitions of $\mathcal{N}(\lambda_3)$ and $\mathcal{N}(\lambda_4)$, it is clear that only one of them will be chosen for firing due to mutual exclusion. Consequently, we have two possible execution paths:

- $\mathcal{N}(\lambda_3)$ fires to put tokens in each place in $\{r_1, \ldots, r_n\}$; thereby, reaching a correspondence with the initial state of $\mathcal{N}(\lambda_1)$.
- $\mathcal{N}(\lambda_4)$ fires to put tokens in each place in $\{s_1, \ldots, s_n\}$, thereby, reaching a correspondence with the initial state of $\mathcal{N}(\lambda_2)$.

If we assume that $\mathcal{N}(\lambda_1)$ and $\mathcal{N}(\lambda_2)$ are both deadlock-free, it is evident that the net $\mathcal{N}((\lambda_3 \trianglerighteq_{\rho_1} \lambda_1)?_{\rho_3}(\lambda_4 \trianglerighteq_{\rho_2} \lambda_2))$ cannot get stuck in any of the two possible execution paths. Therefore, $\mathcal{N}((\lambda_3 \trianglerighteq_{\rho_1} \lambda_1)?_{\rho_3}(\lambda_4 \trianglerighteq_{\rho_2} \lambda_2))$ is deadlock-free, as required. □

Remark 15. As $\lambda_1?_\rho\lambda_2$ identifies all the e-inports of λ_1 with all e-inports of λ_2, it is evident that the in-syncs λ_3 and λ_4 must be isomorphic. Accordingly, we deliberately use the same subindices for p-, r- and s-places so as to show that there is a one-to-one correspondence between the input and output places of $\mathcal{N}(\lambda_3)$ and $\mathcal{N}(\lambda_4)$.

Remark 16. Although it is a statement about the functor $\mathcal{N}$, Proposition 50 is applicable to the functors $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$ presented in Section 3. The proof is valid for $\mathcal{C} \circ \mathfrak{E}$ since Proposition 16 says $\mathcal{C}$ is just a restriction of $\mathcal{N}$ to $\mathfrak{E}(\mathbf{Set}^{\mathbf{Comp}})$.

As we are only interested in checking deadlock-freedom for $\mathcal{D}$-nets with initial and final states (see Remark 5), we only need to consider the form depicted in the proof of Proposition 50 for $j, n > 0$. As this is exactly what we have for any net under $\mathcal{N}$ (because any computon always has ec-inports and ec-outports), the proof that any $\mathcal{D}$-net (with initial and final states) is deadlock-free is analogous to that of Proposition 50.

6.3.2. Encapsulation of control flow and data flow in branching computons

A branching computon encapsulates branching control flow and up to branching data flow, as a result of matching all the e-inports/e-outports of one computon with all the e-inports/e-outports of another. It is branching in the sense a corresponding net chooses an execution path out of two possible ones. For instance, Figure 27 shows the encapsulation given by the branching computon resulting from the colimit construction depicted in Figure 25.

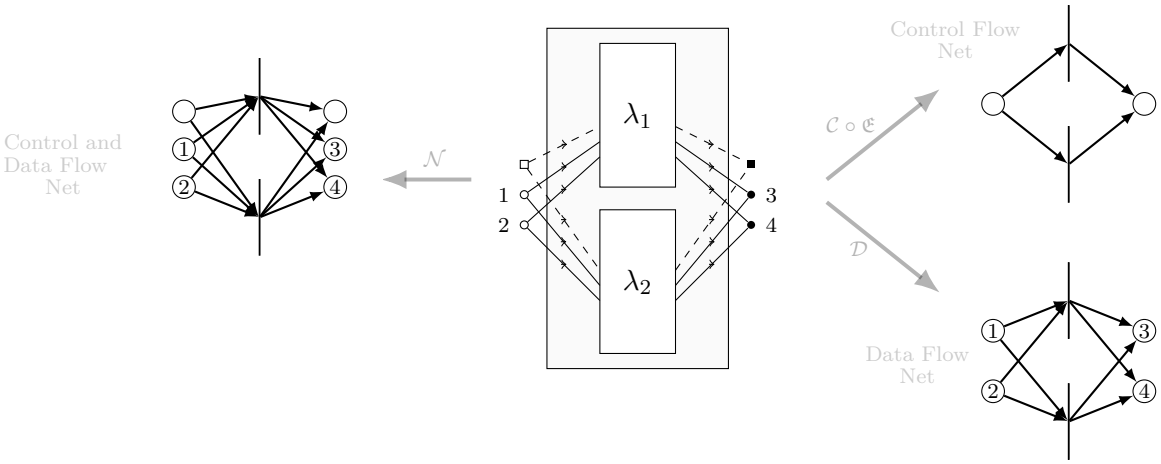


Figure 27: Branching control flow and branching data flow encapsulated by the branching computon from Figure 25. We label some places for mapping purposes even though Petri nets are not labelled (see Section 3).

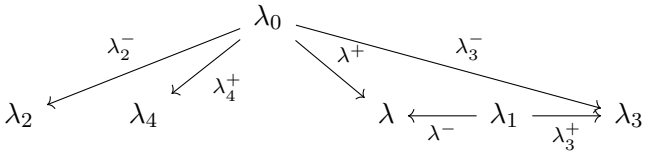
6.4. Iterative Computons

An *iterative computon* is structurally made up of four connected computons, one of which corresponds to the computon λ being iterated over while another allows the repeated invocation of λ. The other two connected computons serve as endpoints to respectively enter and exit the iterative computational structure being defined. In this subsection, we describe two classes of iterative computons: *head-iterative* and *tail-iterative*.

6.4.1. Head-Iterative Computons

A *head-iterative computon* decides to either continue or terminate an iterative process just before executing an arbitrary connected computon. To construct it, it suffices to compute the colimit of a so-called *h-diagram* which defines basic building blocks, namely four connected computons and two trivial computons, together with six marker morphisms (see Definitions 34 and 35). By Lemma 4, such a colimit always exists in the category $\mathbf{Set}^{\mathbf{Comp}}$ so a head-iterative computon can always be constructed. By Corollary 5, a head-iterative computon is a connected computon.

Definition 34 (H-Diagram)**.** An h-diagram is a diagram with the following shape in $\mathbf{Set}^{\mathbf{Comp}}$:



where:

- λ is a connected computon with an in-marker λ^+ and an out-marker λ^-,
- λ_2 is a connected computon with an out-marker λ_2^-,
- λ_3 is a connected computon with an in-marker λ_3^+ and an out-marker λ_3^-, and
- λ_4 is a connected computon with an in-marker λ_4^+.

Evidently, by Definition 20, λ_0 and λ_1 are trivial computons, serving as domains for the markers involved in the h-diagram.

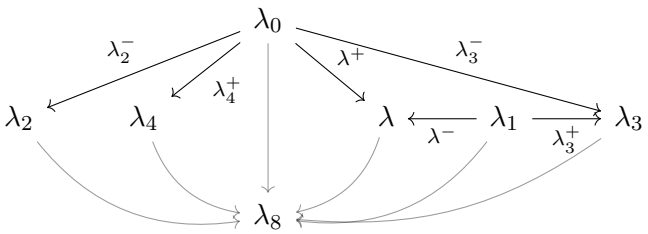
Definition 35 (Head-Iterative Computon)**.** A head-iterative computon is the colimit of an h-diagram.

Notation 6. For convenience, we write a star symbol before a computon symbol λ to indicate that the decision-making locus that terminates the iterative process is placed just before the computational structure of λ. For example, we write $*_\rho(\lambda)$ for the colimit of the h-diagram ρ shown in Definition 34.

Lemma 4. A head-iterative computon can always be constructed in $\mathbf{Set}^{\mathbf{Comp}}$.

Proof. Considering the h-diagram shown in Definition 34, by Propositions 8 and 22, we know that the pushouts of $\lambda_2 \xleftarrow{\lambda_2^-} \lambda_0 \xrightarrow{\lambda_3^-} \lambda_3$ and $\lambda_4 \xleftarrow{\lambda_4^+} \lambda_0 \xrightarrow{\lambda^+} \lambda$ can be constructed. Let us denote them $(\beta_1 : \lambda_2 \to \lambda_5, \lambda_5, \beta_2 : \lambda_3 \to \lambda_5)$ and $(\beta_3 : \lambda_4 \to \lambda_6, \lambda_6, \beta_4 : \lambda \to \lambda_6)$, respectively. By pushout commutativity, we deduce the existence of computon morphisms $f : \lambda_0 \to \lambda_5$ and $g : \lambda_0 \to \lambda_6$ such that $f = \beta_1 \circ \lambda_2^- = \beta_2 \circ \lambda_3^-$ and $g = \beta_3 \circ \lambda_4^+ = \beta_4 \circ \lambda^+$.

Now, by Proposition 10, we know that the coproduct $\lambda_0 + \lambda_1$ can be formed. Using the universal property of coproducts, we also know there must be unique computon morphisms $(f, \beta_2 \circ \lambda_3^+)$ and $(g, \beta_4 \circ \lambda^-)$. To show that $\lambda_5 \xleftarrow{(f,\beta_2\circ\lambda_3^+)} \lambda_0 + \lambda_1 \xrightarrow{(g,\beta_4\circ\lambda^-)} \lambda_6$ is pushable, assume $p_6 \in (g, \beta_4 \circ \lambda^-)(\vec{o}(f, \beta_2 \circ \lambda_3^+))$ so there is some $q \in \vec{o}(f, \beta_2 \circ \lambda_3^+)$ such that $(g, \beta_4 \circ \lambda^-)(q) = p_6$. As $(f, \beta_2 \circ \lambda_3^+)(q)\bullet \setminus (f, \beta_2 \circ \lambda_3^+)(q\bullet) \neq \emptyset$, it is true that $(f, \beta_2 \circ \lambda_3^+)(q) \notin P_5^-$. By coproduct definition and considering that λ_5 is the pushout of the span $\lambda_2 \xleftarrow{\lambda_2^-} \lambda_0 \xrightarrow{\lambda_3^-} \lambda_3$ of out-marker morphisms, we use Proposition 23 to deduce there is some $p_1 \in P_1$ where $\beta_2(\lambda_3^+(p_1)) = (f, \beta_2 \circ \lambda_3^+)(q)$. Again, by coproduct definition, we get $(g, \beta_4 \circ \lambda^-)(q) = p_6 = \beta_4(\lambda^-(p_1)) \in P_6^-$ (because λ^- is an out-marker morphism and β_4 is an induced morphism for the pushout of $\lambda_4 \xleftarrow{\lambda_4^+} \lambda_0 \xrightarrow{\lambda^+} \lambda$). As the other conditions of Definition 9 are proved analogously, it is true that the pushout λ_7 of $(f, \beta_2 \circ \lambda_3^+)$ and $(g, \beta_4 \circ \lambda^-)$ can be constructed. We now show that such a pushout satisfies the universal property of the colimit of the original h-diagram by supposing there is a cone:



Since λ_5 and λ_6 are the respective pushouts of $\lambda_2 \xleftarrow{\lambda_2^-} \lambda_0 \xrightarrow{\lambda_3^-} \lambda_3$ and $\lambda_4 \xleftarrow{\lambda_4^+} \lambda_0 \xrightarrow{\lambda^+} \lambda$, by the universal property of pushouts, there are unique computon morphisms $\lambda_5 \to \lambda_8$ and $\lambda_6 \to \lambda_8$ that make the corresponding diagrams commute.

In the above cone, there are computon morphisms $\lambda_0 \to \lambda_8$ and $\lambda_1 \to \lambda_8$. Using the universal property of coproducts, we deduce there is a unique computon morphism $\lambda_0 + \lambda_1 \to \lambda_8$. Finally, we use the existence of $\lambda_5 \to \lambda_8$ and $\lambda_6 \to \lambda_8$ and the universal property of pushouts to deduce there is a unique computon morphism $\lambda_7 \to \lambda_8$ that makes everything commute in our construction. Thus, proving that λ_7 is the colimit of the original h-diagram. □

Corollary 5. A head-iterative computon is a connected computon.

Proof. In the construction presented in the proof of Lemma 4, the pushouts λ_5 and λ_6 are connected computons by the fact that λ, λ_2, λ_3 and λ_4 also are (see Proposition 9). Consequently, the pushout λ_7 of the pushable span $\lambda_5 \xleftarrow{(f,\beta_2\circ\lambda_3^+)} \lambda_0 + \lambda_1 \xrightarrow{(g,\beta_4\circ\lambda^-)} \lambda_6$ is a connected computon. As λ_7 is the colimit of the (original) h-diagram shown in Definition 34, we conclude that every head-iterative computon is a connected computon. □

To clarify the construction presented in the proof of Lemma 4, Figure 28 illustrates a complete, self-descriptive example for the formation of a head-iterative computon $*_\rho(\lambda)$ over a functional computon λ where ρ is the h-diagram shown in the middle.

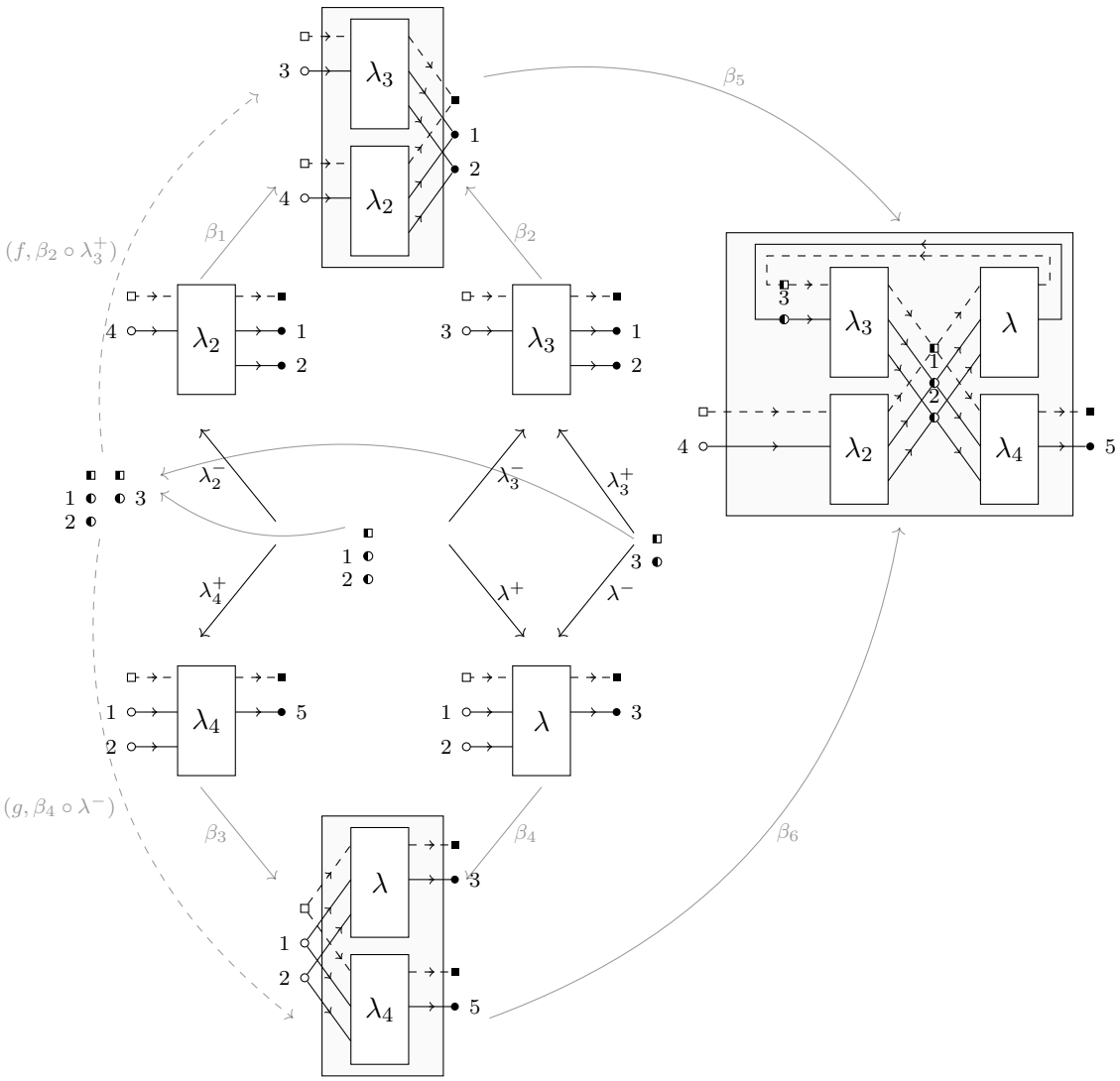


Figure 28: Constructing a head-iterative computon $*_\rho(\lambda)$ where ρ is the h-diagram shown in the middle (whose morphisms are displayed as black arrows). Here, $f = \beta_1 \circ \lambda_2^- = \beta_2 \circ \lambda_3^-$ and $g = \beta_3 \circ \lambda_4^+ = \beta_4 \circ \lambda^+$.

A glance at Figure 28 reveals that $*_\rho(\lambda)$ is constructed from three additional connected computons (i.e., λ_2, λ_3 and λ_4) and two trivial computons. One of the trivial computons serves as the domain for the in-markers λ_4^+ and λ^+ so the e-inports of λ_4 and λ match. This trivial computon also serves as the domain for the out-markers λ_2^- and λ_3^-. The other trivial computon is the domain of the in-marker λ_3^+ and the out-marker λ^-, i.e., the e-inports of λ_3 are identified with the e-outports of λ.

The right-most composite in Figure 28 shows that the connected computons λ_2 and λ_4 serve as the respective entry and exit points for the whole iterative structure of $*_\rho(\lambda)$. Particularly, λ_2 is needed because, without this, $*_\rho(\lambda)$ will enter into a closed loop with no entry points (i.e., no e-inports); thus, violating Definition 1. Beyond this, there is no special requirement for the e-inports of λ_2 or the e-outports of λ_4, as evidenced by the h-diagram shown in Definition 34. Not enforcing specific requirements on this matter enables a high degree of modelling flexibility. For instance, it is possible to deem λ_2 as a computon that replicates data (when its e-inports and e-outports are isomorphic) or as a computon that receives data of a certain type, performs some processing on that data and returns data of a different type. In our particular example, as its e-inports and e-outports do not coincide, we can treat λ_2 as a computon that pre-processes (or filters) information before sending it into the iterative computation defined over λ. By Theorem 4, a head-iterative computon can always be formed for any arbitrary connected computon, regardless of the data such an arbitrary computon requires or produces.

Theorem 4. λ is a connected computon $\iff$ a head-iterative computon $*_\rho(\lambda)$ exists for some h-diagram ρ.

Proof. ($\Longrightarrow$) Let λ be an arbitrary connected computon. By Proposition 21, we deduce the existence of an in-marker $\lambda^+ : \lambda_0 \to \lambda$ and an out-marker $\lambda^- : \lambda_1 \to \lambda$. Now, if λ_2 and λ_3 are connected computons and duals of λ (see Proposition 24), Definition 21 says there is an in-marker $\lambda_3^+ : \lambda_1 \to \lambda_3$ as well as out-markers $\lambda_3^- : \lambda_0 \to \lambda_3$ and $\lambda_2^- : \lambda_0 \to \lambda_2$. Finally, if λ_4 is a computon isomorphic to λ, λ_4 must be connected and there must evidently exists an in-marker $\lambda_4^+ : \lambda_0 \to \lambda_4$.

The above construction corresponds to that of an h-diagram ρ so we simply apply Lemma 4 to deduce that the colimit of ρ exists. Using Definition 35 and Notation 6, we conclude such a colimit is the head-iterative computon $*_\rho(\lambda)$.

($\Longleftarrow$) This part of the proof follows directly from Definition 34. □

6.4.2. Operational semantics for head-iterative computons (in the theory of Petri nets)

No matter whether we use any of the three functorial constructions presented in Section 3, the Petri net of a head-iterative computon has no additional places or transitions beyond those from the nets of the computons of the corresponding h-diagram. The general structure of a net of this sort is depicted in Figure 29.

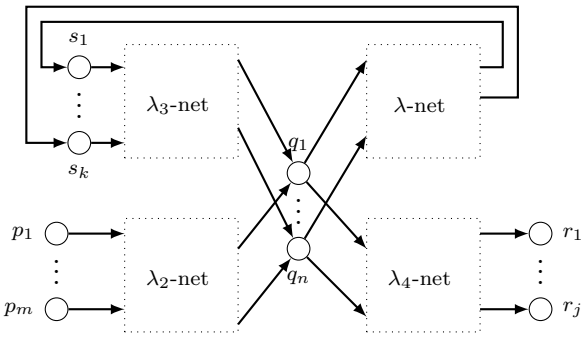


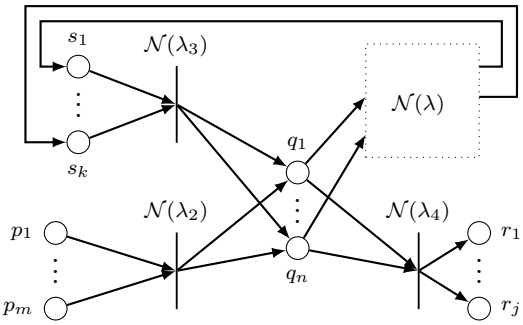
Figure 29: General structure of the Petri net of a head-iterative computon, considering the h-diagram from Definition 34. This structure is applicable to the functorial constructions $\mathcal{N}$, $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$ from Section 3.

Unfortunately, there is no guarantee every head-iterative's net is deadlock-free even when the nets of the computons from the corresponding h-diagram are. Despite of this, it is still possible to enforce deadlock-freedom by using primitive computons as entry, exit and iteration points. Proposition 27 and Remark 8 together say every primitive computon's net is deadlock-free. So, as long as the net of the computon being iterated over never gets stuck, the corresponding head-iterative's net will be deadlock-free (see Proposition 51 and Remark 17).

Proposition 51. Consider the h-diagram ρ depicted in Definition 34 and assume M_i and M_f are the initial and final markings of the net $\mathcal{N}(\lambda)$, respectively. The net $\mathcal{N}(*_\rho(\lambda))$ is deadlock-free if:

1. λ_j is a primitive computon for $j = 2, 3, 4$,
2. $\mathcal{N}(\lambda)$ is deadlock-free, and
3. for every marking M reachable from M_i, if $M(s) > 0$ for each output place s of $\mathcal{N}(\lambda)$ then $M = M_f$.

Proof. Consider the h-diagram ρ from Definition 34 and assume λ_j is a primitive computon for $j = 2, 3, 4$. By Proposition 26, ρ is a well-defined h-diagram because each λ_j is a connected computon. Using Lemma 4, we deduce the existence of $*_\rho(\lambda)$ whose underlying net $\mathcal{N}(*_\rho(\lambda))$ has the following form according to the functorial construction presented in Definition 15:



The above net evidently has the form depicted in Figure 29. The only difference is that, rather than black-boxing $\mathcal{N}(\lambda_j)$, we display its internals which consist of only one transition (because λ_j is primitive). By Definition 16, we know the initial state M_i of $\mathcal{N}(*_\rho(\lambda))$ is a marking function where $M_i(p) > 0$ for all $p \in \{p_1, \ldots, p_m\}$ and no tokens for all the other places, including those inside $\mathcal{N}(\lambda)$. This marking evidently enables the only transition of $\mathcal{N}(\lambda_2)$ and nothing else. Consequently, firing $\mathcal{N}(\lambda_2)$ reaches a state that marks each place in $\{q_1, \ldots, q_n\}$. Assuming $\mathcal{N}(\lambda)$ is deadlock-free and that only its final state can put tokens in each element from $\{s_1, \ldots, s_k\}$ (see Conditions 2 and 3), we now have two possible execution paths (as per mutual exclusion):

1. If $\mathcal{N}(\lambda_4)$ is executed, the final state of $\mathcal{N}(*_\rho(\lambda))$ will immediately be reached with tokens in $r_1, \ldots, r_j$. Therefore, $\mathcal{N}(*_\rho(\lambda))$ will not get stuck.
2. If $\mathcal{N}(\lambda)$ is executed, we have two options:
   (a) No state of $\mathcal{N}(\lambda)$ ever puts tokens in all the places in $\{s_1, \ldots, s_k\}$. In this case, even though $\mathcal{N}(\lambda)$ never terminates successfully, there is a guarantee $\mathcal{N}(*_\rho(\lambda))$ will not get stuck because $\mathcal{N}(\lambda)$ is deadlock-free.
   (b) A state of $\mathcal{N}(\lambda)$ puts tokens in all the places in $\{s_1, \ldots, s_k\}$. If so, the final marking of $\mathcal{N}(\lambda_3)$ will be reached, which simply puts a token in each place in $\{q_1, \ldots, q_n\}$. As we have the same two execution alternatives again, we simply repeat 1 or 2 whichever applies.

By the above, it is evident that all the execution paths lead to a deadlock-free execution. Therefore, we conclude $\mathcal{N}(*_\rho(\lambda))$ is deadlock-free, as required. □

Remark 17. Although it is a statement about the functor $\mathcal{N}$, Proposition 51 is applicable to the functors $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$ presented in Section 3. The proof is valid for $\mathcal{C} \circ \mathfrak{E}$ since Proposition 16 says $\mathcal{C}$ is just a restriction of $\mathcal{N}$ to $\mathfrak{E}(\mathbf{Set}^{\mathbf{Comp}})$.

Remark 5 says we are only interested in checking deadlock-freedom for $\mathcal{D}$-nets that have initial and final states. A glance at the figure depicted in the proof of Proposition 51 reveals this is satisfied when $j, m > 0$. Starting with the initial state M_i that puts tokens in $p_1, \ldots, p_m$, we have the following cases:

- If $n = 0$, M_i enables the only transition of $\mathcal{D}(\lambda_2)$ which, upon firing, makes $\mathcal{D}(*_\rho(\lambda))$ enter into a deadlock state.
- If $k = 0$ and $n > 0$, $\mathcal{D}(\lambda)$ will never reach its final state. Despite of this, $\mathcal{D}(*_\rho(\lambda))$ is guaranteed to be deadlock-free when $\mathcal{D}(\lambda)$ also is.
- If $k > 0$ and $n > 0$, the proof of deadlock-freedom for $\mathcal{D}(*_\rho(\lambda))$ is analogous to that of Proposition 51.

Therefore, to guarantee $\mathcal{D}(*_\rho(\lambda))$ is deadlock-free, we must consider an h-diagram ρ where the entry and iteration computons have ed-outports, apart from ensuring that $\mathcal{D}(\lambda)$ is deadlock-free and that satisfies Condition 3 of Proposition 51.

6.4.3. Encapsulation of control flow and data flow in head-iterative computons

By Definition 35, we know a head-iterative computon $*_\rho(\lambda)$ is the colimit of an h-diagram ρ which, by Definition 34, is formed by four connected computons and two trivial computons. One of the connected computons is λ (i.e., the computon being iterated over) whereas the others serve as entry, exit and iteration points. Thus, $*_\rho(\lambda)$ encapsulates cyclic control flow and up to cyclic data flow. By cyclic, we mean λ and the iteration entity are executed repeatedly. In a head-iterative computon, the decision whether to repeat λ is made before executing it. To give a concrete example, Figure 30 illustrates the encapsulation given by the head-iterative computon resulting from the colimit construction depicted in Figure 28.

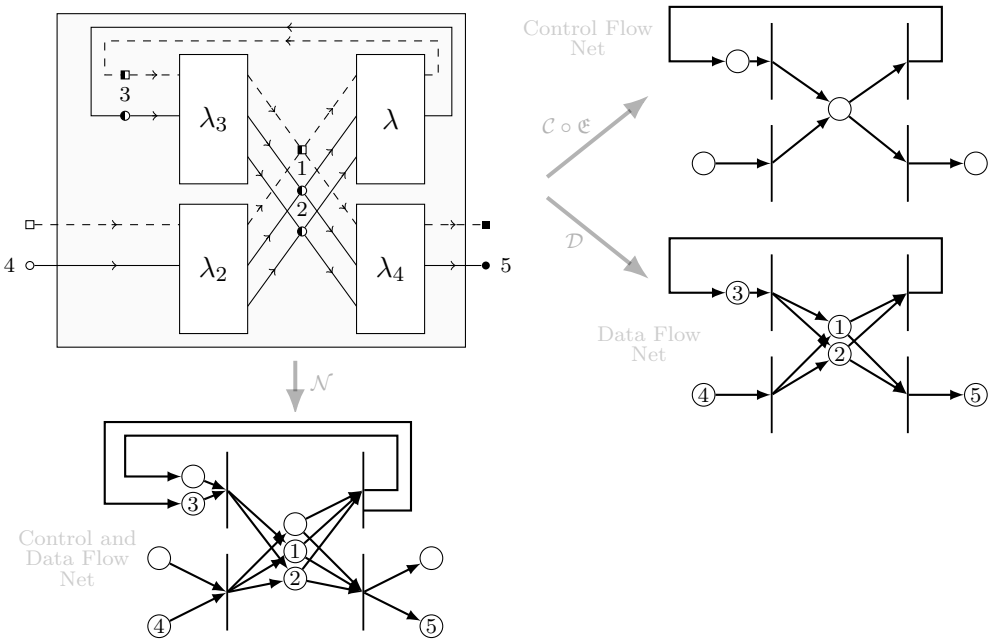
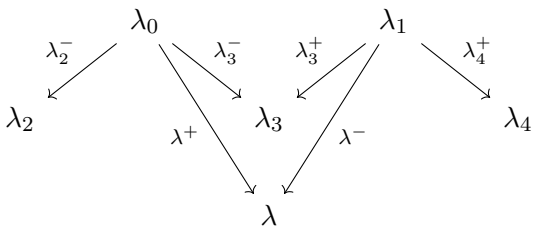


Figure 30: Cyclic control flow and cyclic data flow encapsulated by the head-iterative computon from Figure 28. We label some places for mapping purposes even though Petri nets are not labelled (see Section 3).

6.4.4. Tail-Iterative Computons

A *tail-iterative computon* is structurally similar to a head-iterative one in the sense it is formed from the same basic building blocks, namely four connected computons and two trivial computons, as specified by the notion of a *t-diagram* (see Definition 36). The difference lies in the position of the structure that non-deterministically chooses continuation or termination of the iterative computation. While a head-iterative computon defines such a structure just before the computon being iterated over, a tail-iterative one specifies it immediately after. Analogically, in the realm of imperative programming languages, a head-iterative computon corresponds to a *while* construct, whereas a tail-iterative is akin to a *do-while* statement. Like head-iterative computons, tail-iteratives are connected computons which can always be constructed in $\mathbf{Set}^{\mathbf{Comp}}$ (see Definition 37, Lemma 5 and Corollary 6).

Definition 36 (T-Diagram)**.** A t-diagram is a diagram with the following shape in $\mathbf{Set}^{\mathbf{Comp}}$:



where:

- λ is a connected computon with an in-marker λ^+ and an out-marker λ^-,
- λ_2 is a connected computon with an out-marker λ_2^-,
- λ_3 is a connected computon with an in-marker λ_3^+ and an out-marker λ_3^-, and
- λ_4 is a connected computon with an in-marker λ_4^+.

Evidently, by Definition 20, λ_0 and λ_1 are trivial computons serving as domains for the markers involved in the t-diagram.

Definition 37 (Tail-Iterative Computon)**.** A tail-iterative computon is the colimit of a t-diagram.

Notation 7. For convenience, we write a star symbol after a computon symbol λ to indicate that the decision-making locus that terminates the iterative process is placed immediately after the computational structure of λ. For example, we write $(\lambda)*_\rho$ for the colimit of the t-diagram ρ shown in Definition 36.

Lemma 5. A tail-iterative computon can always be constructed in $\mathbf{Set}^{\mathbf{Comp}}$.

Proof. Considering the t-diagram shown in Definition 36, by Propositions 8 and 22, we know that the pushouts of $\lambda_2 \xleftarrow{\lambda_2^-} \lambda_0 \xrightarrow{\lambda_3^-} \lambda_3$ and $\lambda_3 \xleftarrow{\lambda_3^+} \lambda_1 \xrightarrow{\lambda_4^+} \lambda_4$ can be constructed. Let us denote them $(\beta_1 : \lambda_2 \to \lambda_5, \lambda_5, \beta_2 : \lambda_3 \to \lambda_5)$ and $(\beta_3 : \lambda_3 \to \lambda_6, \lambda_6, \beta_4 : \lambda_4 \to \lambda_6)$, respectively.

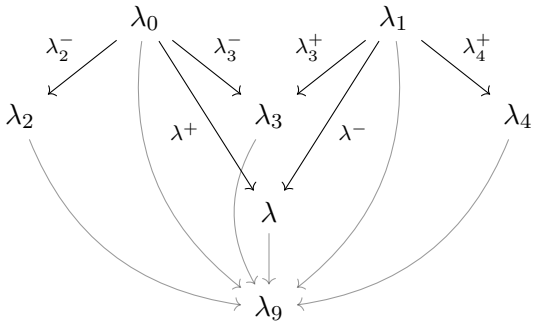
To show that the induced span $\lambda_5 \xleftarrow{\beta_2} \lambda_3 \xrightarrow{\beta_3} \lambda_6$ is pushable, we just prove $\beta_2(\vec{o}(\beta_3)) \subseteq P_5^+ \cup P_5^-$ since the other conditions of Definition 9 follow analogously. For this, assume $p_5 \in \beta_2(\vec{o}(\beta_3))$ so there is some $p_3 \in \vec{o}(\beta_3)$ where $\beta_2(p_3) = p_5$. As β_3 is a computon morphism, Definition 7 says $\vec{i}(\beta_3) \cup \vec{o}(\beta_3) \subseteq P_3^+ \cup P_3^-$. Supposing for contradiction $\beta_2(p_3) \notin P_5^+ \cup P_5^-$, we have $t_5(o_5) = \beta_2(p_3) = s_5(i_5)$ for some $o_5 \in O_5$ and some $i_5 \in I_5$. Accordingly, we consider the following cases:

1. When $p_3 \in P_3^+$, we use $O_5 = O_2 +_{O_0} O_3$ to determine the preimage of o_5. If $o_5 = \beta_1(o_2)$ for some $o_2 \in O_2$, $\beta_1 \circ t_2 = t_5 \circ \beta_1$ entails $\beta_1(t_2(o_2)) = t_5(\beta_1(o_2)) = t_5(o_5) = \beta_2(p_3)$. As β_1 and β_2 are morphisms induced by the pushout of a span of out-markers, $\beta_1(t_2(o_2)) = \beta_2(p_3)$ if and only if $t_2(o_2) \in P_2^-$ and $p_3 \in P_3^-$. But λ_3 is a connected computon as per Definition 36 so $p_3 \in P_3^- \cap P_3^+$ cannot hold as per Proposition 2. If $o_5 = \beta_2(o_3)$ for some $o_3 \in O_3$, $\beta_2 \circ t_3 = t_5 \circ \beta_2$ entails $\beta_2(t_3(o_3)) = t_5(\beta_2(o_3)) = t_5(o_5) = \beta_2(p_3)$. As $t_3(o_3) = p_3 \in P_3^+$ directly contradicts Definition 2, we focus on $t_3(o_3) \neq p_3$. By considering that the P-component of β_2 can only be non-injective on the image of the P-component of λ_3^-, we deduce $\beta_2(t_3(o_3)) = \beta_2(p_3)$ with $t_3(o_3) \neq p_3 \iff t_3(o_3), p_3 \in P_3^-$. Again, $p_3 \in P_3^- \cap P_3^+$ contradicts Proposition 2 because λ_3 is a connected computon.
2. When $p_3 \in P_3^-$, we use $I_5 = I_2 +_{I_0} I_3$ to determine the preimage of i_5. If $i_5 = \beta_1(i_2)$ for some $i_2 \in I_2$, $\beta_1 \circ s_2 = s_5 \circ \beta_1$ entails $\beta_1(s_2(i_2)) = s_5(\beta_1(i_2)) = s_5(i_5) = \beta_2(p_3)$. As β_1 and β_2 are morphisms induced by the pushout of a span of out-markers, $\beta_1(s_2(i_2)) = \beta_2(p_3)$ if and only if $s_2(i_2) \in P_2^-$ and $p_3 \in P_3^-$. But $s_2(i_2) \in P_2^-$ cannot be true because that would contradict Definition 2. If $i_5 = \beta_2(i_3)$ for some $i_3 \in I_3$, $\beta_2 \circ s_3 = s_5 \circ \beta_2$ entails $\beta_2(s_3(i_3)) = s_5(\beta_2(i_3)) = s_5(i_5) = \beta_2(p_3)$. As $s_3(i_3) = p_3 \in P_3^-$ directly contradicts Definition 2, we focus on $s_3(i_3) \neq p_3$. By considering that the P-component of β_2 can only be non-injective on the image of the P-component of λ_3^-, we deduce $\beta_2(s_3(i_3)) = \beta_2(p_3)$ with $s_3(i_3) \neq p_3$ if and only if $s_3(i_3), p_3 \in P_3^-$. Clearly, $s_3(i_3) \in P_3^-$ contradicts Definition 2.

Disproving the above scenarios implies that our initial assumption must be false. So, $\beta_2(p_3) \in P_5^+ \cup P_5^-$. Showing $\lambda_5 \xleftarrow{\beta_2} \lambda_3 \xrightarrow{\beta_3} \lambda_6$ is pushable allows us to use Proposition 8 to

construct its pushout, denoted $(\beta_5 : \lambda_5 \to \lambda_7, \lambda_7, \beta_6 : \lambda_6 \to \lambda_7)$. By pushout commutativity, we deduce the existence of computon morphisms $f : \lambda_0 \to \lambda_7$ and $g : \lambda_1 \to \lambda_7$ where $f = \beta_5 \circ \beta_1 \circ \lambda_2^- = \beta_5 \circ \beta_2 \circ \lambda_3^- = \beta_6 \circ \beta_3 \circ \lambda_3^-$ and $g = \beta_5 \circ \beta_2 \circ \lambda_3^+ = \beta_6 \circ \beta_3 \circ \lambda_3^+ = \beta_6 \circ \beta_4 \circ \lambda_4^+$.

Now, by Proposition 10, we know that the coproduct $\lambda_0 + \lambda_1$ can be formed. Using the universal property of coproducts, we also know there must be unique computon morphisms $(\lambda^+, \lambda^-) : \lambda_0 + \lambda_1 \to \lambda$ and $(f, g) : \lambda_0 + \lambda_1 \to \lambda_7$. To show that $\lambda_7 \xleftarrow{(f,g)} \lambda_0 + \lambda_1 \xrightarrow{(\lambda^+,\lambda^-)} \lambda$ is pushable, assume $p_7 \in (f, g)(\vec{i}(\lambda^+, \lambda^-))$ so there is some $q \in \vec{i}(\lambda^+, \lambda^-)$ where $(f, g)(q) = p_7$. As $\bullet(\lambda^+, \lambda^-)(q) \setminus (\lambda^+, \lambda^-)(\bullet q) \neq \emptyset$, we have $(\lambda^+, \lambda^-)(q) \notin P^+$. Consequently, by coproduct definition, $(\lambda^+, \lambda^-)(q)$ must be in the image of λ^-. That is, there must be some $p_1 \in P_1$ for which $\lambda^-(p_1) = (\lambda^+, \lambda^-)(q) \in P^-$. As λ_3^+ and λ_4^+ are in-markers, we can apply Proposition 23, while recognising that λ_5 does not identify P_3^+-ports, to further deduce $g(p_1) \in P_7^+$. By coproduct definition, we have $(f, g)(q) = p_7 = g(p_1) \in P_7^+$ and, therefore, $(f, g)(\vec{i}(\lambda^+, \lambda^-)) \subseteq P_7^+ \cup P_7^-$. As proving $(f, g)(\vec{o}(\lambda^+, \lambda^-)) \subseteq P_7^+ \cup P_7^-$ and $(\lambda^+, \lambda^-)(\vec{i}(f, g)) \cup (\lambda^+, \lambda^-)(\vec{o}(f, g)) \subseteq P^+ \cup P^-$ can be done analogously, the pushout $(\beta_7 : \lambda \to \lambda_8, \lambda_8, \beta_8 : \lambda_7 \to \lambda_8)$ of (λ^+, λ^-) and (f, g) can be constructed. To show such a pushout satisfies the universal property of the colimit of the original t-diagram, suppose there is a cone:



Since λ_5 and λ_6 are the pushouts of $\lambda_2 \xleftarrow{\lambda_2^-} \lambda_0 \xrightarrow{\lambda_3^-} \lambda_3$ and $\lambda_3 \xleftarrow{\lambda_3^+} \lambda_1 \xrightarrow{\lambda_4^+} \lambda_4$, respectively, we know there are unique computon morphisms $\lambda_5 \to \lambda_9$ and $\lambda_6 \to \lambda_9$ that make the corresponding triangles commute. Taking into account these morphisms and considering that λ_7 is the pushout of the induced span $\lambda_5 \xleftarrow{\beta_2} \lambda_3 \xrightarrow{\beta_3} \lambda_6$, we use the universal property of pushouts to deduce there is a unique computon morphism $\lambda_7 \to \lambda_9$ that also makes the corresponding diagram commute.

In the above cone, there are computon morphisms $\lambda_0 \to \lambda_9$ and $\lambda_1 \to \lambda_9$. Using the universal property of coproducts, we deduce there is a unique computon morphism $\lambda_0 + \lambda_1 \to \lambda_9$. As in the cone there also is $\lambda \to \lambda_9$, we use the universal property of pushouts to deduce the existence of a unique computon morphism $\lambda_8 \to \lambda_9$. Thus, proving that λ_8 is the colimit of the original t-diagram. □

Corollary 6. A tail-iterative computon is a connected computon.

Proof. Considering the construction presented in the proof of Lemma 5, we know that the pushouts λ_5 and λ_6 are connected computons because λ_2, λ_3 and λ_4 also are (see Proposition 9). Consequently, the pushout λ_7 of the induced span $\lambda_5 \xleftarrow{\beta_2} \lambda_3 \xrightarrow{\beta_3} \lambda_6$ is also a connected computon.

As λ_7 and λ are connected computons, we use again Proposition 9 to deduce that the pushout λ_8 of the unique (pushable) span $\lambda_7 \xleftarrow{(f,g)} \lambda_0 + \lambda_1 \xrightarrow{(\lambda^+,\lambda^-)} \lambda$ is a connected computon. As λ_8 is the colimit of the original t-diagram (shown in Definition 36), we conclude that every tail-iterative computon is a connected computon. □

Building upon the proof of Lemma 5, Figure 31 shows a detailed, self-descriptive example for the construction of a tail-iterative computon $(\lambda)*_\rho$ where λ is the same connected computon we use in Figure 28 and ρ is the t-diagram whose morphisms are displayed as black arrows.

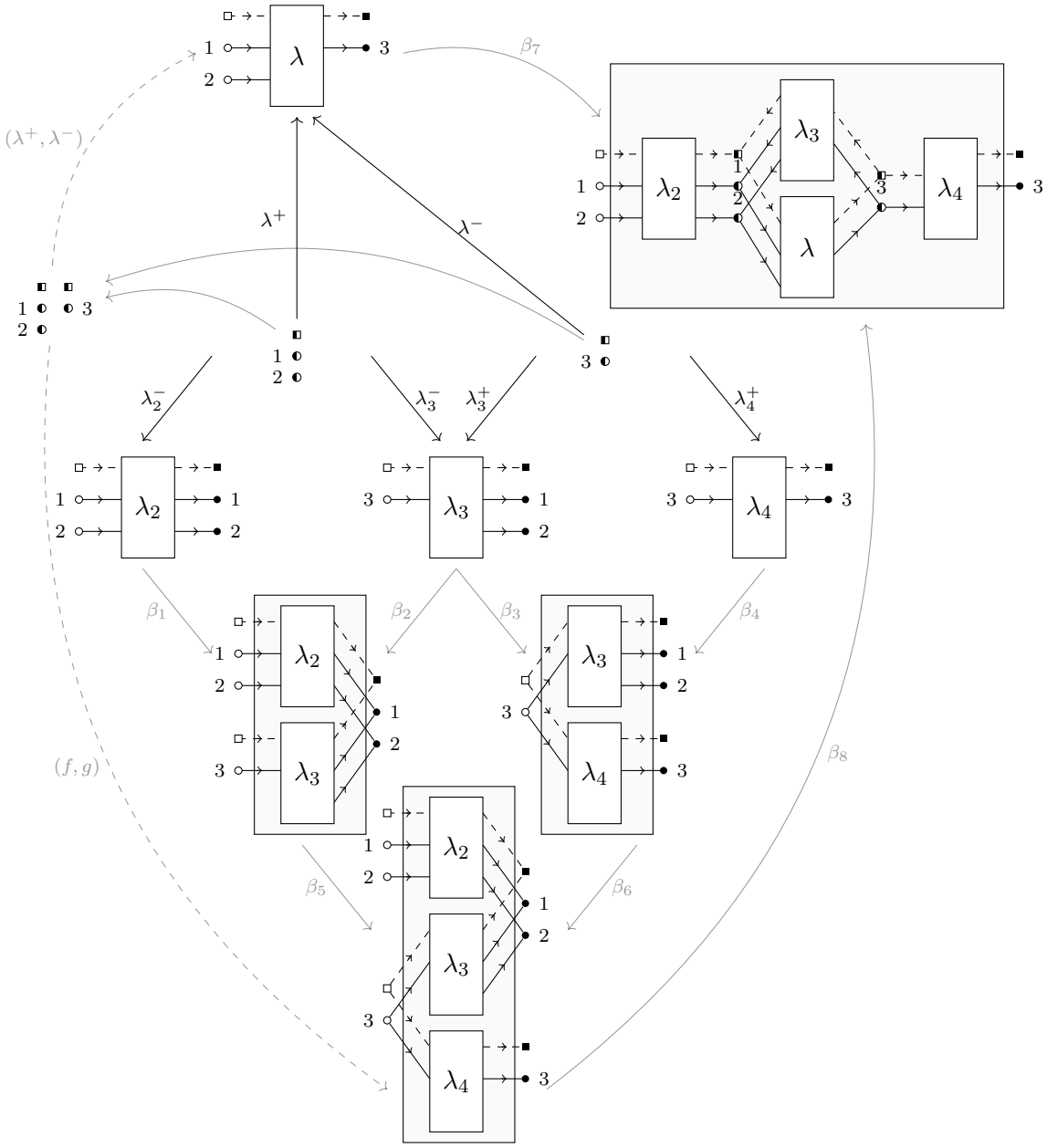


Figure 31: Constructing a tail-iterative computon $(\lambda)*_\rho$ where ρ is the t-diagram whose morphisms are displayed as black arrows. Here, $f = \beta_5 \circ \beta_1 \circ \lambda_2^- = \beta_5 \circ \beta_2 \circ \lambda_3^- = \beta_6 \circ \beta_3 \circ \lambda_3^-$ and $g = \beta_5 \circ \beta_2 \circ \lambda_3^+ = \beta_6 \circ \beta_3 \circ \lambda_3^+ = \beta_6 \circ \beta_4 \circ \lambda_4^+$.

Figure 31 shows that, like head-iterative computons, the connected computons λ_2 and λ_4 serve as entry and exit points for the iterative computation, respectively, while the connected computon λ_3 enables the repeated invocation of λ. Although in this example the e-inports and e-outports of λ_2 are isomorphic (the same for λ_4), there no strict requirement for enforcing this as there is no in-marker λ_2^+ and no out-marker λ_4^- in the corresponding t-diagram ρ. Not enforcing this structural feature enables a certain degree of flexibility in the sense the endpoints of a tail-iterative computon can or cannot expose the interface of the computon being iterated over (i.e., λ). Again, like head-iterative computons, it is possible to operationally implement λ_2 and λ_4 in different manners. For instance, in our particular scenario, λ_2 can be treated as a computon that either duplicates information or transforms data of the same type. As we are dealing with high-level computations, the internals of such functional computons are irrelevant. We just focus on structure from a "birds-eye viewpoint". By Theorem 5, a tail-iterative composite can always be constructed for any arbitrary connected computon, regardless of the data such an arbitrary computon requires or produces.

Theorem 5. λ is a connected computon $\iff$ a tail-iterative computon $(\lambda)*_\rho$ exists for some t-diagram ρ.

Proof. ($\Longrightarrow$) Let λ be an arbitrary connected computon. By Proposition 21, we deduce there is an in-marker $\lambda^+ : \lambda_0 \to \lambda$ and an out-marker $\lambda^- : \lambda_1 \to \lambda$. Now, if λ_2, λ_3 and λ_4 are connected computons and duals of λ (see Proposition 24), Definition 21 says there are in-markers $\lambda_3^+ : \lambda_1 \to \lambda_3$ and $\lambda_4^+ : \lambda_1 \to \lambda_4$ as well as out-markers $\lambda_2^- : \lambda_0 \to \lambda_2$ and $\lambda_3^- : \lambda_0 \to \lambda_3$.

As the above construction corresponds to that of a t-diagram ρ, by Lemma 5, we have that the colimit of ρ exists. Using Definition 37 and Notation 7, we conclude such a colimit is the tail-iterative computon $(\lambda)*_\rho$.

($\Longleftarrow$) This part of the proof follows directly from Definition 36. □

6.4.5. Operational semantics for tail-iterative computons (in the theory of Petri nets)

No matter whether we use any of the three functorial constructions from Section 3, the Petri net of a tail-iterative computon has no additional places or transitions beyond those from the nets of the computons of the corresponding t-diagram. The general structure of a net of this sort is depicted in Figure 32.

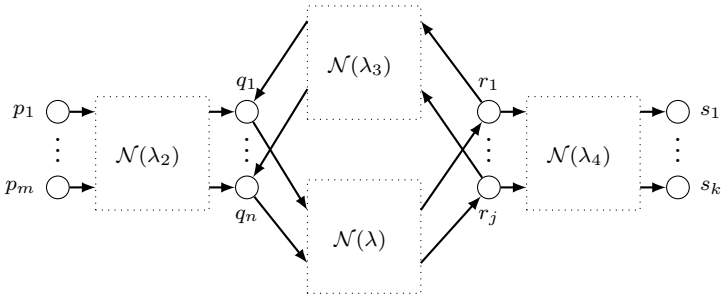


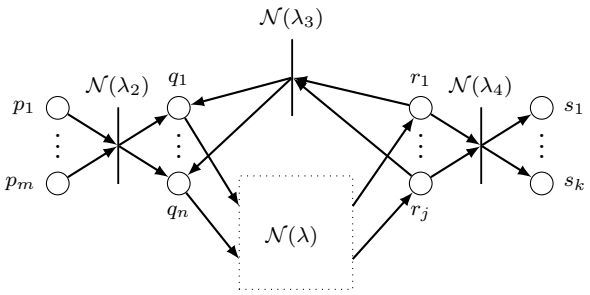
Figure 32: General structure of the Petri net of a tail-iterative computon, considering the t-diagram from Definition 36. This structure is applicable to all the functorial constructions from Section 3: $\mathcal{N}$, $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$.

Unfortunately, there is no guarantee every tail-iterative's net is deadlock-free even when the nets of the computons from the corresponding t-diagram are. Despite of this, it is still possible to enforce deadlock-freedom by employing primitive computons as entry, exit and iteration points. Proposition 27 and Remark 8 together entail every primitive computon's net is deadlock-free. So, as long as the net of the computon being iterated over never gets stuck, the net of the corresponding tail-iterative computon will be deadlock-free too (see Proposition 52 and Remark 18).

Proposition 52. Consider the t-diagram ρ depicted in Definition 36 and assume M_i and M_f are the initial and final markings of the net $\mathcal{N}(\lambda)$, respectively. The net $\mathcal{N}((\lambda)*_\rho)$ is deadlock-free if:

1. λ_j is a primitive computon for $j = 2, 3, 4$,
2. $\mathcal{N}(\lambda)$ is deadlock-free, and
3. for every marking M reachable from M_i, if $M(r) > 0$ for each output place r of $\mathcal{N}(\lambda)$ then $M = M_f$.

Proof. Consider the t-diagram ρ from Definition 36 and assume λ_j is a primitive computon for $j = 2, 3, 4$. By Proposition 26, we know ρ is a well-defined t-diagram because each λ_j is a connected computon. Then, using Lemma 5, we deduce the existence of $(\lambda)*_\rho$ whose underlying net $\mathcal{N}((\lambda)*_\rho)$ has the following form according to the functorial construction presented in Definition 15:



The above net evidently has the form depicted in Figure 32. The only difference is that, rather than black-boxing $\mathcal{N}(\lambda_j)$, we display its internals which consist of only one transition (since λ_j is primitive). By Definition 16, we know the initial state M_i of $\mathcal{N}((\lambda)*_\rho)$ is a marking function where $M_i(p) > 0$ for all $p \in \{p_1, \ldots, p_m\}$ and no tokens for all the other places, including those inside $\mathcal{N}(\lambda)$. This marking evidently enables the only transition of $\mathcal{N}(\lambda_2)$ and nothing else. Consequently, firing $\mathcal{N}(\lambda_2)$ reaches a state that marks each place in $\{q_1, \ldots, q_n\}$. This new marking evidently corresponds to the initial state of $\mathcal{N}(\lambda)$. Assuming $\mathcal{N}(\lambda)$ is deadlock-free and that only its final state can put tokens in each element from $\{r_1, \ldots, r_j\}$ (see Conditions 2 and 3), we now have the following cases:

1. If no state of $\mathcal{N}(\lambda)$ ever puts tokens in all the places in $\{r_1, \ldots, r_j\}$, $\mathcal{N}(\lambda)$ will never terminate successfully. Despite of this, there is a guarantee $\mathcal{N}((\lambda)*_\rho)$ will not get stuck because $\mathcal{N}(\lambda)$ is deadlock-free.
2. If a state of $\mathcal{N}(\lambda)$ puts tokens in all the places in $\{r_1, \ldots, r_j\}$, there are two possible execution paths because $\mathcal{N}(\lambda_3)$ and $\mathcal{N}(\lambda_4)$ are both enabled (due to mutual exclusion):
   (a) If $\mathcal{N}(\lambda_4)$ is triggered, the final state of $\mathcal{N}((\lambda)*_\rho)$ will be reached with tokens in $s_1, \ldots, s_k$. So, $\mathcal{N}((\lambda)*_\rho)$ is deadlock-free.
   (b) If $\mathcal{N}(\lambda_3)$ is triggered, only the places in $\{q_1, \ldots, q_n\}$ will be marked in the next state. As the initial state of $\mathcal{N}(\lambda)$ is reached once again, we simply repeat 1 or 2 whichever applies.

By the above, it is evident that all the execution paths lead to a deadlock-free execution. Therefore, we conclude $\mathcal{N}((\lambda)*_\rho)$ is deadlock-free, as required. □

Remark 18. Although it is a statement about the functor $\mathcal{N}$, Proposition 52 is applicable to the functors $\mathcal{C} \circ \mathfrak{E}$ and $\mathcal{D}$ presented in Section 3. The proof is valid for $\mathcal{C} \circ \mathfrak{E}$ since Proposition 16 says $\mathcal{C}$ is just a restriction of $\mathcal{N}$ to $\mathfrak{E}(\mathbf{Set}^{\mathbf{Comp}})$.

Remark 5 says we are only interested in checking deadlock-freedom for $\mathcal{D}$-nets that have initial and final states. A glance at the figure depicted in the proof of Proposition 52 reveals this is satisfied when $k, m > 0$. Starting with the initial state M_i that puts tokens in $p_1, \ldots, p_m$, we have following cases:

- If $n = 0$, M_i enables the only transition of $\mathcal{D}(\lambda_2)$ which, upon firing, makes $\mathcal{N}((\lambda)*_\rho)$ enter into a deadlock state.
- If $j = 0$ and $n > 0$, the net $\mathcal{D}(\lambda)$ will never reach its final state. Despite of this, $\mathcal{N}((\lambda)*_\rho)$ is guaranteed to be deadlock-free when $\mathcal{D}(\lambda)$ also is.
- If $j > 0$ and $n > 0$, the proof of deadlock-freedom for $\mathcal{N}((\lambda)*_\rho)$ is analogous to that of Proposition 52.

Hence, to guarantee $\mathcal{D}((\lambda)*_\rho)$ is deadlock-free, we must consider a t-diagram ρ where the entry and iteration computons have both ed-outports, apart from ensuring that $\mathcal{D}(\lambda)$ is deadlock-free and that satisfies Condition 3 of Proposition 52.

6.4.6. Encapsulation of control flow and data flow in tail-iterative computons

By Definition 37, we know a tail-iterative computon $(\lambda)*_\rho$ is the colimit of a t-diagram ρ which, by Definition 36, consists of four connected computons and two trivial computons. One of the connected computons is λ (i.e., the computon being iterated over) whereas the others serve as entry, exit and iteration points. Thus, like a head-iterative, $(\lambda)*_\rho$ encapsulates cyclic control flow and up to cyclic data flow. By cyclic, we mean λ and the iteration entity are executed repeatedly. In a tail-iterative computon, the decision whether to repeat λ is made after executing it. To give a concrete example, Figure 33 illustrates the encapsulation given by the tail-iterative computon resulting from the colimit construction depicted in Figure 31.

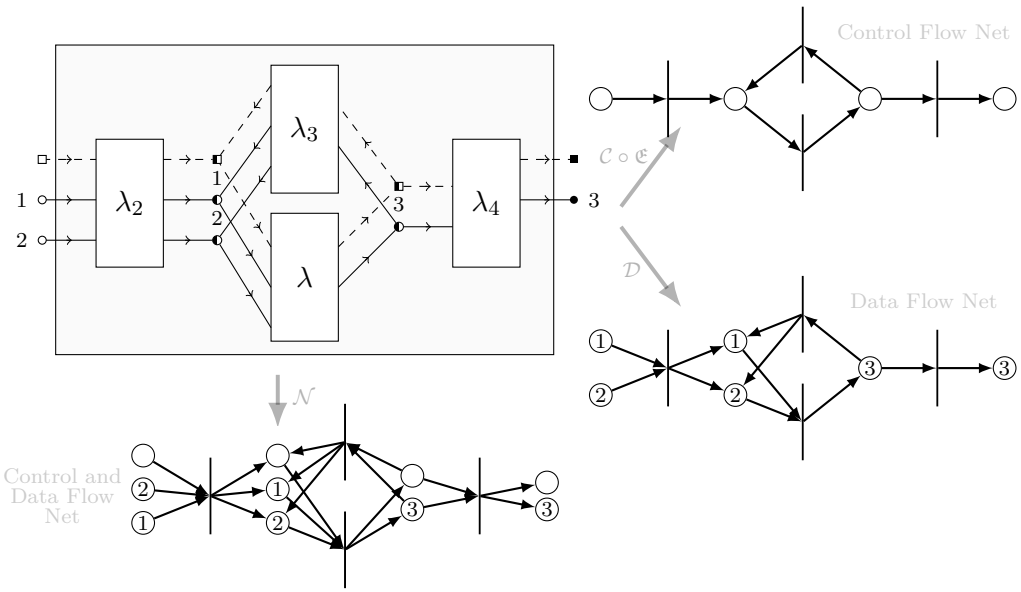


Figure 33: Cyclic control flow and cyclic data flow encapsulated by the tail-iterative computon from Figure 31. We label some places for mapping purposes even though Petri nets are not labelled (see Section 3).

7. Applications of the Proposed Model

Compositionality is not exclusive of a single domain, but it appears in many spheres, from physical [36] to artificial systems [9]. This section describes the application of the computon model in two different domains: software engineering and artificial intelligence. Although we are not proposing an end-user modelling language but just an MHC to capture the essence of high-level computations, this section serves to demonstrate the suitability of our model for the compositional construction of high-level computations that separate data flow and control flow.

For each case study, we describe the respective composite computons and show how the separation of concerns can be exploited to analyse control flow independently from data flow (and vice versa), in this case for model transformation. We particularly express control flow as a diagram in Business Process Model and Notation (BPMN) [37] which is the standard language that has been widely used for many years, in both academia and industry, to canonically model workflow control flow. This conversion process does not consider data flow at all and is done through the graph transformation system described in Section 7.1.1. For data flows, we rely upon DFGs in standard notation [38] where arrows represent data flow and circles denote consumer or producer computations. The conversion process is performed without considering control flow at all via the graph transformation system described in Section 7.1.2. Both BPMN diagrams and standard DFGs are far more expressive than Petri nets to express control and data flow within a system. For completeness, for each scenario, we display the Petri net under $\mathcal{C} \circ \mathfrak{E}$ that comprehensively captures system behaviour. Constructing the

corresponding nets under $\mathcal{N}$ or $\mathcal{D}$ can easily be done using the mapping from Appendix A, which adheres to the functorial constructions from Definition 15 and Proposition 17.

For clarity, we do not describe the corresponding colimiting diagrams of composite computons, but their definition is left to the reader as a matter of routine exercise. For each composite, we try to provide as much internal structure as possible. But when this is not possible due to space restrictions, we simply make reference to a previously created composite. Some composites are not shown independently so as to save space and minimise duplication.

7.1. Transformation System

Before presenting our concrete case studies, we describe our model transformation system which consists of two different modules, one for transforming a computon into its corresponding BPMN diagram and another for retrieving the respective DFG in standard notation. To simplify transformation, we operate on computon CFGs and DFGs rather than Petri nets. This is because such constructs are multidirected labelled graphs that embed all the necessary information about computon flows, without adhering to specific operational semantics (see Definitions 11 and 13).

7.1.1. Transforming a Computon CFG into a BPMN Diagram

For this, we propose a simple graph transformation system *ad hoc* to our specific case studies, whose aim is to realise the syntax mapping displayed in Figure 34. As our purpose is not to provide general rewriting rules but to demonstrate how the separation of control and data flow can be used for model transformation purposes in two concrete scenarios, we do not investigate theoretical properties such as confluence. Instead, we just ensure that rewriting terminates in a correct state. For this, we implemented our system in Groove [39], a well-established tool that supports state-space exploration, which enabled us to validate termination and confirm that, in each case study, all derivations lead to the intended terminal graph. In the future, we would like to investigate more general, application-independent rewriting rules as well as efficient rewriting algorithms.

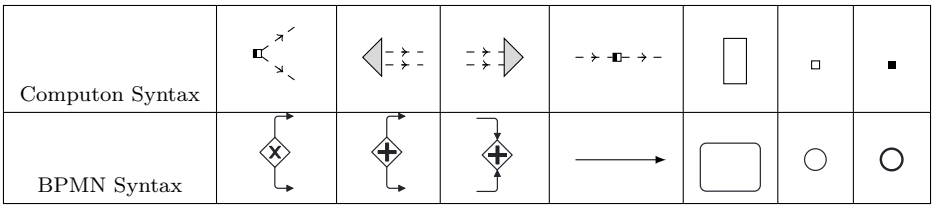


Figure 34: Mapping from computon syntax to BPMN syntax.

To understand the purpose of our transformation system, let us consider the example depicted in Figure 35 in which we first convert a composite computon into its corresponding CFG via the functorial construction $\mathscr{C}$ described in Section 2.4. This CFG serves as the input to our system which, in turn, performs the mapping from Figure 34 to produce a BPMN diagram that captures the control flow structure of our composite. Here, for instance, we can notice that the unique fork construct has been replaced with an AND gateway, whereas ic-ports (together with their adjacent flows) have been substituted with a simple arrow, known as sequence flow in the context of BPMN. These two rewritings are just application instances of the mapping shown in Figure 34.

Although it might seem trivial, realising such a mapping cannot be done directly since technical considerations need to be taken into account to maintain graph integrity while ensuring semantic correctness (e.g., avoiding dangling edges during rewriting). For that reason, we propose a graph transformation system which converts a computon CFG $\mathscr{C}(\lambda)$ into a BPMN diagram G via the sequential application of the injective rules from the set $R = \{r_1, \ldots, r_{12}\}$ displayed in Figure 36.

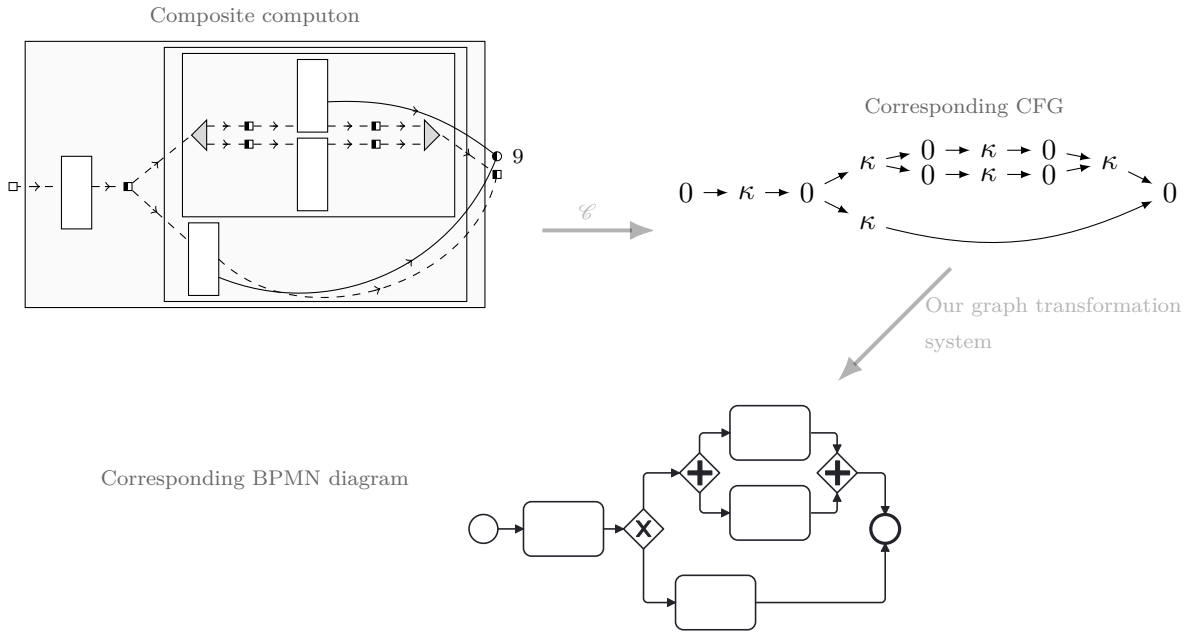


Figure 35: Example to illustrate the intended purpose of our graph transformation system.

Each of the R-rules from Figure 36 is applied individually until no further match is found, so our system produces a sequence of derivations of the form $\mathscr{C}(\lambda) \Rightarrow_{r_j} \cdots \Rightarrow_{r_k} G$ for $j, k \in \{1, \ldots, 12\}$ and $j \leq k$. As we rely on the Double-Pushout Approach for graph transformation [40], a derivation $G_1 \Rightarrow_{r_i} G_2$ exists for $i \in \{1, \ldots, 12\}$ if there is a context graph G_0 that makes the following two squares commute in $\mathbf{Set}^{\mathbf{Gr}}$:

$$\begin{array}{ccccc} \mathcal{L}_i & \longleftarrow & \mathcal{I}_i & \longrightarrow & \mathcal{R}_i \\ \downarrow & & \downarrow & & \downarrow \\ G_1 & \longleftarrow & G_0 & \longrightarrow & G_2 \end{array}$$

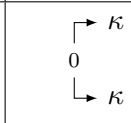
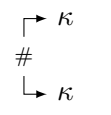
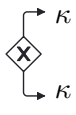
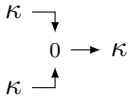
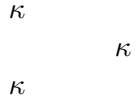
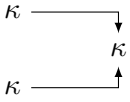
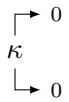
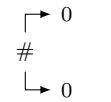
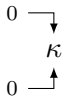
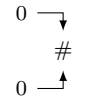
Here, $\mathcal{L}_i$, $\mathcal{R}_i$ and $\mathcal{I}_i$ denote the left-hand side, right-hand side and interface of a rule r_i.

The purpose of r_1 is to replace branching sources with XOR gateways (i.e., BPMN elements that denote alternative control flow) through the match of 0-nodes with exactly two outgoing edges (i.e., control ports with two salient control flows). In our rule set, XOR join gateways are not taken into account to facilitate transformation processes, a consideration that is completely valid since such constructs are not strictly required in BPMN diagrams [41, 42].

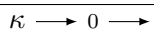
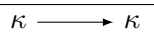
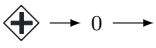
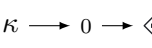
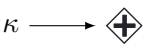
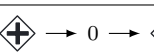
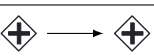
Rule r_2 removes intermediate branching sinks by matching 0-nodes connected from two κ-nodes to exactly one κ-node (i.e., control ports linked from two computation units to some other unit). Applying r_2 results in the removal of a 0-node (together with its adjacent edges) and in the addition of new edges from the sources of that node to its single target. Rules r_3 and r_4 rewrite parallel splits and synchronisation points, respectively. Splits correspond to κ-nodes with two outgoing edges (i.e., fork computons), whereas synchronisation points are κ-nodes with two incoming edges (i.e., join computons). Both of them are relabelled as AND gateways, i.e., BPMN constructs for splitting or merging concurrent execution paths.

Rules r_5 to r_8 manage sequencing by removing all intermediate 0-nodes (i.e., ic-ports) together with their adjacent edges. Whenever a node is removed, an arc is put instead, from the source of its left edge to the target of its right one. The purpose of rule r_9 is to replace k-nodes (i.e., computation units) with BPMN tasks which are atomic activities that represent a unit of computation performed by some computing device. Rules r_{10}, r_{11} and r_{12} simply relabel ec-inports and ec-outports as start and end events, respectively. In BPMN, start events denote control flow origin whilst end events represent control flow termination.[21]

[21] We are aware rules r_5, r_6, r_7 and r_8 can be merged into a single one by the use of node restrictions (the same

Rule	Left-Hand Side	Interface	Right-Hand Side
r_1			
r_2			
r_3			
r_4			

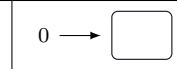
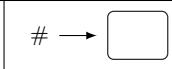
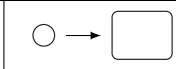
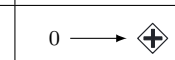
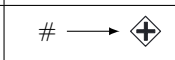
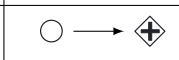
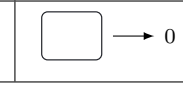
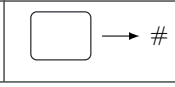
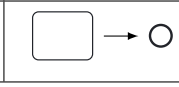
(a) Rules for rewriting branching and parallel constructs.

Rule	Left-Hand Side	Interface	Right-Hand Side
r_5			
r_6			
r_7			
r_8			

(b) Rules for rewriting sequential constructs.

Rule	Left-Hand Side	Interface	Right-Hand Side
r_9	κ	#	

(c) Rule for rewriting functional computons.

Rule	Left-Hand Side	Interface	Right-Hand Side
r_{10}			
r_{11}			
r_{12}			

(d) Rules for rewriting ec-inports and ec-outports.

Figure 36: Rewriting rules to transform a computon CFG into a BPMN diagram. Here, a hash symbol # is used as a wildcard to ensure that structure is preserved. This choice is arbitrary and any other label can be used instead, as long as it is different from the labels used for CFGs or BPMN diagrams. Labels are put in the place of nodes, and object mapping corresponds to graphical arrangement.

To validate the twelve rules from Figure 36 on the scenarios described in Sections 7.2 and 7.3, we implemented our graph transformation system in Groove [39] which is a reference tool for specifying and simulating such kind of systems. With the help of Groove, we verified that our rule set R satisfies dangling conditions and that it correctly produces BPMN diagrams for the CFG of the total sequential computon from Figure 41 and the memory cell from Figure 46(e). Particularly, the correct BPMN diagram for Figure 41 is produced after exploring

for rules r_{10} and r_{11}). Our purpose is not to provide a minimal set of rewriting rules but to demonstrate how the separation of control and data flow can be leveraged to convert a computon CFG into its equivalent BPMN diagram, without the need of analysing data flow at all. We believe that avoiding the use of node restrictions clarifies our transformation system and provides additional expressivity in terms of graph matching.

37 states and 36 transitions, whereas the BPMN diagram for Figure 46(e) is obtained after exploring 35 states and 34 transitions. For the simulation, we used linear state exploration which chooses one transition from each open state. For reproduction purposes, our source code is available at https://github.com/damianarellanes/cfg-transformation.[22]

7.1.2. Transforming a Computon DFG into a DFG in Standard Notation

For this, we propose a functor S from the category $\mathbf{Set}^{\mathbf{Gr}}$ to the category of graphs with labelled vertices and labelled edges, whose behaviour is formalised in Definition 38.

Definition 38. Given a computon DFG $\mathscr{D}(\lambda)$, the functor S constructs a graph $S(\mathscr{D}(\lambda))$ by letting:

- the set L' of vertex labels be L,
- the set M' of edge labels be $L \cup \{\epsilon\}$ (ϵ denotes the empty label),
- the set of vertices $V' \subseteq V$ be $\{v \mid v^- = 0 \ \vee \ v^+ = 0 \ \vee \ l(v) = \kappa\}$,
- the set of edges $E' \subseteq V' \times V' \times L'$ be $\{(v, w, \epsilon) \mid (\exists e \in E)[\vec{s}(e) = v \wedge \vec{t}(e) = w]\} \cup \{(v, w, x) \mid (\exists e_1, e_2 \in E)(\exists y \in V)[\vec{s}(e_1) = v \wedge \vec{t}(e_1) = y \wedge \vec{s}(e_2) = y \wedge \vec{t}(e_2) = w \wedge l(y) = x]\}$.
- the source and target functions be mappings $E' \to V'$ given by $s'(v, w, x) = v$ and $t'(v, w, x) = w$, respectively,
- the edge labelling function $m' : E' \to M'$ and the vertex labelling function $l' : V' \to L'$ be given by $m'(v, w, x) = x$ and $l'(v) = l(v)$, respectively.

For a graph homomorphism $\mathscr{D}(\alpha) : \mathscr{D}(\lambda_1) \to \mathscr{D}(\lambda_2)$, the components of $S(\mathscr{D}(\alpha)) : S(\mathscr{D}(\lambda_1)) \to S(\mathscr{D}(\lambda_2))$ are:

- $S(\mathscr{D}(\alpha))_V : V_1' \to V_2'$ given by $S(\mathscr{D}(\alpha))_V(v) = \mathscr{D}(\alpha)_V(v)$,
- $S(\mathscr{D}(\alpha))_E : E_1' \to E_2'$ given by $S(\mathscr{D}(\alpha))_E(v, w, x) = (\mathscr{D}(\alpha)_V(v), \mathscr{D}(\alpha)_V(w), x)$,
- $S(\mathscr{D}(\alpha))_L : L_1' \to L_2'$ given by $S(\mathscr{D}(\alpha))_L(x) = \mathscr{D}(\alpha)_L(x)$, and
- $S(\mathscr{D}(\alpha))_M : M_1' \to M_2'$ given by $S(\mathscr{D}(\alpha))_M(x) = x$.

Checking the functoriality of S is routine and is analogous to that of Proposition 12.

A glance at Definition 38 reveals that S is a functor that preserves all the boundary- and κ-vertices from a computon DFG $\mathscr{D}(\lambda)$, i.e., ed-inports, ed-outports and computation units. The only edges retained in $S(\mathscr{D}(\lambda))$ are those connected from a vertex with no incoming edges to a κ-node (i.e., data flows from ed-inports to computation units) and edges connected from a κ-node to a vertex with no outgoing edges (i.e., data flows from computation units to ed-outports). These preserved edges are all empty-labelled to meet the requirements of standard DFG notation. To fully satisfy such a notation, it suffices to replace κ vertices from $S(\mathscr{D}(\lambda))$ with $\bigcirc$ symbols. For example, the DFGs in standard notation of the total sequential computon from Figure 41 and of the memory cell from Figure 46(e) are displayed in Figures 43(b) and 48(b), respectively. All these diagrams are created using the functorial construction S from Definition 38.

Basically, our functor S leverages the separation of concerns of the computon model so as to optimise computon DFGs. On the one hand, S compresses $\mathscr{D}(\lambda)$-paths of the form $v \to y \to w$ through the preservation of v and w in $S(\mathscr{D}(\lambda))$ and the creation of an edge $v \to w$ with the label of y. Apart from *path compression*, S performs *multiplicity reduction*,

[22] Groove implicitly creates interfaces, determines context graphs and computes pushouts, given the left- and right-hand side of a rule as well as a host graph.

i.e., edges with the same source, target and label are all collapsed onto a single edge.[23] This behaviour is particularly important to simplify large and complex computon DFGs, leading to more efficient algorithms for data flow analysis and a clearer visualisation of data passing. Although multiplicity reduction is not relevant for any of the scenarios described in this section, we consider it to highlight the benefits of separating data and control towards data flow optimisations that do not consider control and preserve the order of data passing.

7.2. Case Study 1: Compositional AWS Infrastructure Deployment

AWS Step Functions [43] is a serverless orchestration framework by Amazon Web Services (AWS), which allows software developers the implementation and management of (multi-step) serverless application workflows in the cloud, using visual and interactive programming constructs. A step function is a workflow that defines a high-level computation for the invocation of web-services in some pre-defined order, with the aim of automating a specific task such as database provisioning, release management or serverless deployment. In this section, we focus on a step function for automatic infrastructure deployment, provided as a use case by the AWS team [44], which follows different execution paths depending on the state of an AWS CloudFormation stack and intermediate processing results. If the stack does not exist, a new stack is created and deployment succeeds. Otherwise, a change set is created before inspecting its resources and deciding whether the change set will be executed or removed. If the change set is executed, then deployment succeeds; otherwise, deployment fails. The step function we consider is presented in Figure 37.

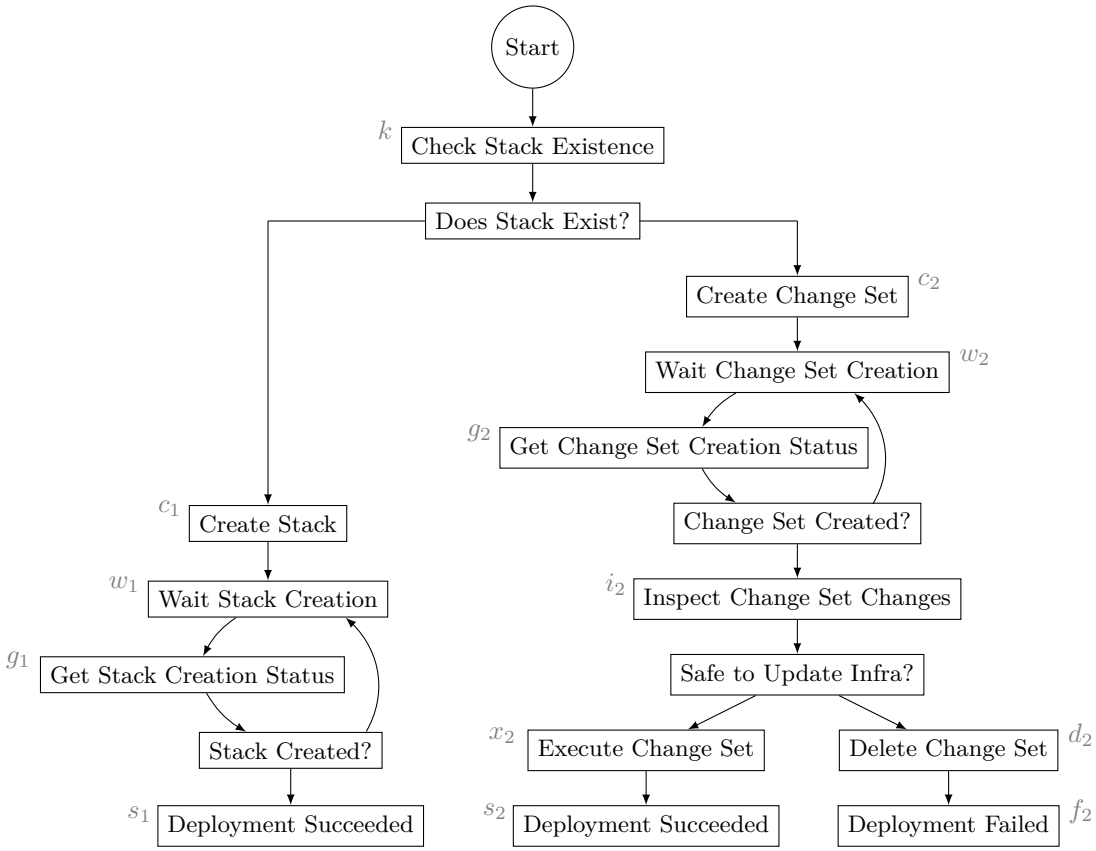


Figure 37: AWS step function to automate infrastructure deployment.

A step function does not allow the specification of data flow but data exchange is implicit in the processing of the steps involved in the workflow control flow being defined. Before executing the workflow, a JSON file is created to specify the initial input in the form of multiple

[23] As an edge is a triple (v, w, x) and a set cannot contain repeated elements, it follows that there cannot be multiples edges with the same source, same target and same label in $S(\mathscr{D}(\lambda))$.

properties and values. The JSON file is modified as the workflow execution progresses, by particularly appending the output of each intermediate web service invocation.

To compositionally construct the step function workflow, we use the model we propose in this paper by considering every service invocation as a functional computon which explicitly defines data required and produced (i.e., data flow and control flow are both explicit). The only processes we do not consider are those that branch control such as *Stack Created?*. This is because those processes are built-in AWS functions which directly correspond to branching computons in our model. Although, strictly speaking, multiple computon ports can be coloured in the same way to represent the same data type (e.g., a boolean), for clarity concerns and demonstration purposes we treat every deployment parameter as a unique colour. The description of each port colour is presented in Figure 38.

Colour	Description
1	Colour of an environment type on which the infrastructure code will be deployed (e.g., development, testing or production).
2	Colour of a name of an AWS CloudFormation stack.
3	Colour of a path to an AWS CloudFormation template.
4	Colour of an identifier of a revision S3 bucket.
5	Colour of a revision S3 key.
6	Colour of a flag that specifies whether the AWS CloudFormation stack exists or not.
7	Colour of a stack creation status.
8	Colour of a state which can be either success or fail.
9	Colour of a change set name.
10	Colour of a change set creation status.
11	Colour of a change set action which determines whether the stack can be safely updated or not.

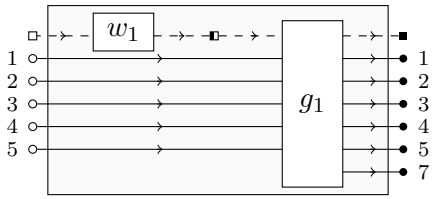
Figure 38: Colours for the AWS step function shown in Figure 37.

As compositionality enforces bottom-up construction, we start by defining the partial sequential computon $w_1 \rhd_{\rho_1} g_1$ where w_1 and g_1 respectively correspond to the processes *wait stack creation* and *get stack creation status* (see Figure 39(a)). This partial sequential computon, together with three functional computons (e_1, e_2 and e_3), serve as the basis to form the tail-iterative computon $(w_1 \rhd_{\rho_1} g_1)*_{\rho_2}$ which waits until the stack is created (see Figure 39(b)). The additional computons e_1, e_2 and e_3 echo data, remove data of colour 7 and discard all data, respectively. The only branching structure in $(w_1 \trianglerighteq_{\rho_1} g_1)*_{\rho_2}$ determines whether the stack has been created or not. If the stack has not been created, the loop continues; otherwise, e_3 is invoked to exit the tail-recursive composite and pass control to the external world.

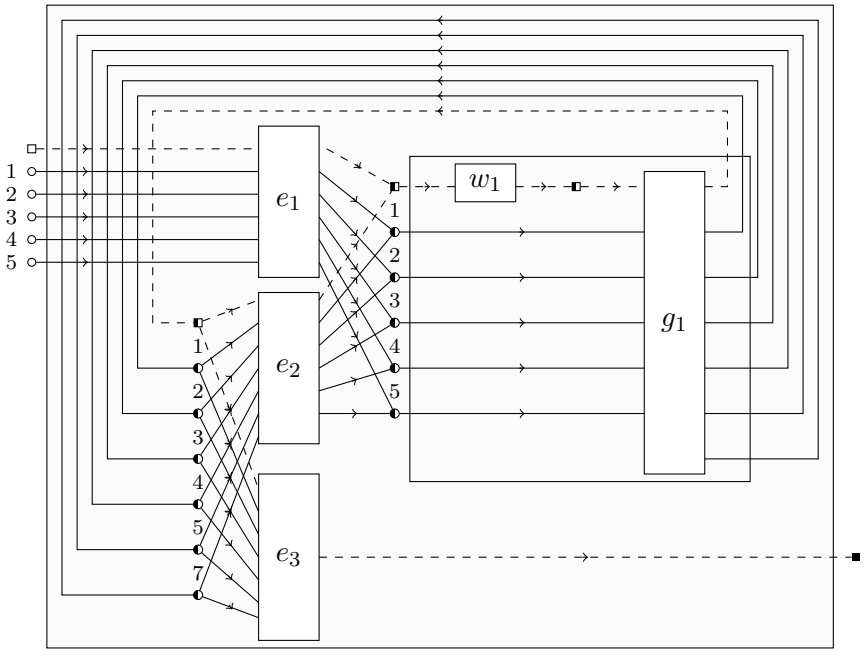
The tail-iterative composite $(w_1 \rhd_{\rho_1} g_1)*_{\rho_2}$ is then used to construct the total sequential computon $c_1 \trianglerighteq_{\rho_3} ((w_1 \rhd_{\rho_1} g_1)*_{\rho_2})$ where c_1 is a functional computon in charge of creating the stack. Such a sequential computon is then used as left operand to construct the (even more complex) total sequential computon $(c_1 \trianglerighteq_{\rho_3} ((w_1 \rhd_{\rho_1} g_1)*_{\rho_2})) \trianglerighteq_{\rho_4} s_1$ in which s_1 is a functional computon that marks deployment as successful. This complex total sequential composite, depicted in Figure 39(c), corresponds to the left path of the step function workflow shown in Figure 37.

For the other path, we first construct the branching computon $(x_2 \trianglerighteq_{\rho_5} s_2)?_{\rho_7}(d_2 \trianglerighteq_{\rho_6} f_2)$ whose only purpose is to succeed or fail deployment (see Figure 40(a)). In this composite, $x_2 \trianglerighteq_{\rho_5} s_2$ corresponds to the total sequential computon which *executes a change set* via the functional computon x_2, before marking deployment as *successful* via the functional computon s_2. The other part of the branching composite triggers the total sequential computon $d_2 \trianglerighteq_{\rho_6} f_2$ which interrupts deployment by first *deleting the change set* via the functional computon d_2 and then using the functional computon f_2 to indicate that deployment has *failed*.

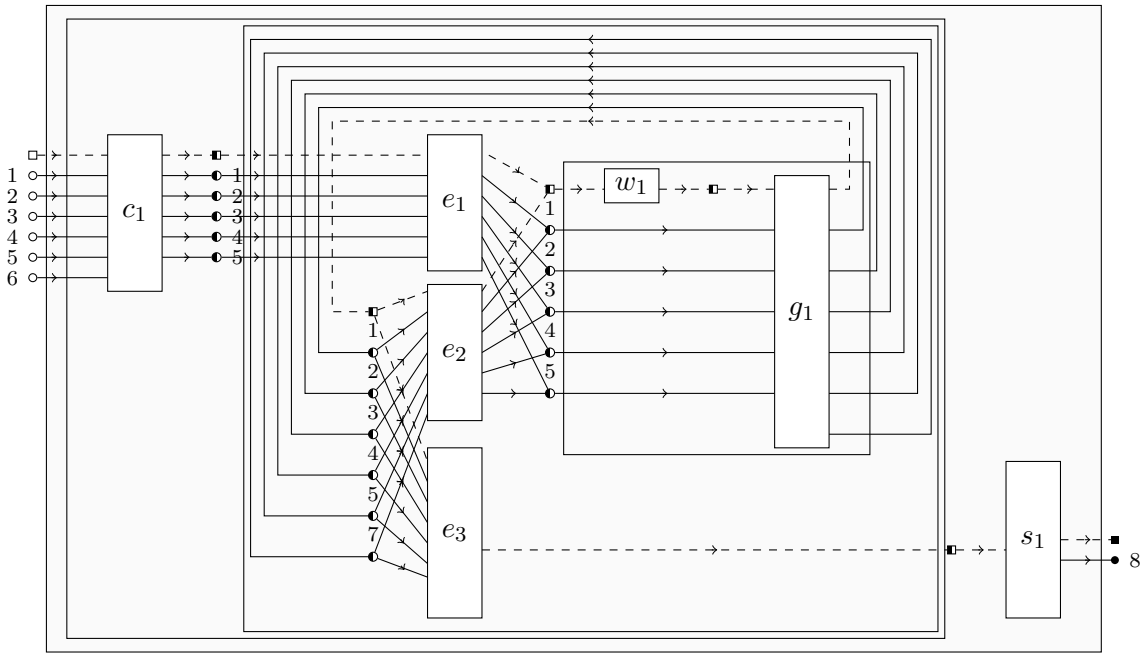
Returning to the innermost right part of the step function shown in Figure 37, we construct the partial sequential computon $w_2 \rhd_{\rho_8} g_2$ where w_2 and g_2 correspond to the proceses *wait change set creation* and *get change set creation status*, respectively. Such a sequential computon is then composed into a tail-iterative composite $(w_2 \rhd_{\rho_8} g_2)*_{\rho_9}$ which waits until the change set gets created, and whose structure is similar to that of $(w_1 \rhd_{\rho_1} g_1)*_{\rho_2}$ (see Figures 39(b) and 40(b)).



(a) Partial sequential computon to wait stack creation before getting a stack creation status: $w_1 \rhd_{\rho_1} g_1$

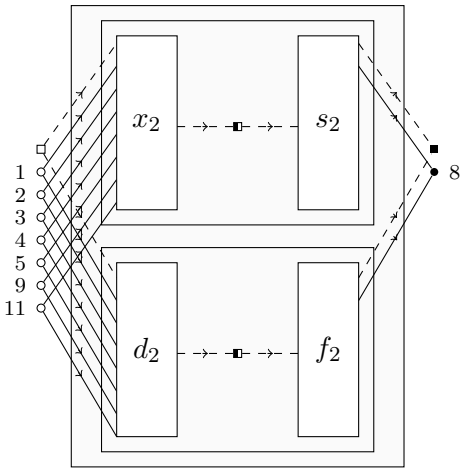


(b) Tail-iterative computon over (a): $(w_1 \rhd_{\rho_1} g_1)*_{\rho_2}$

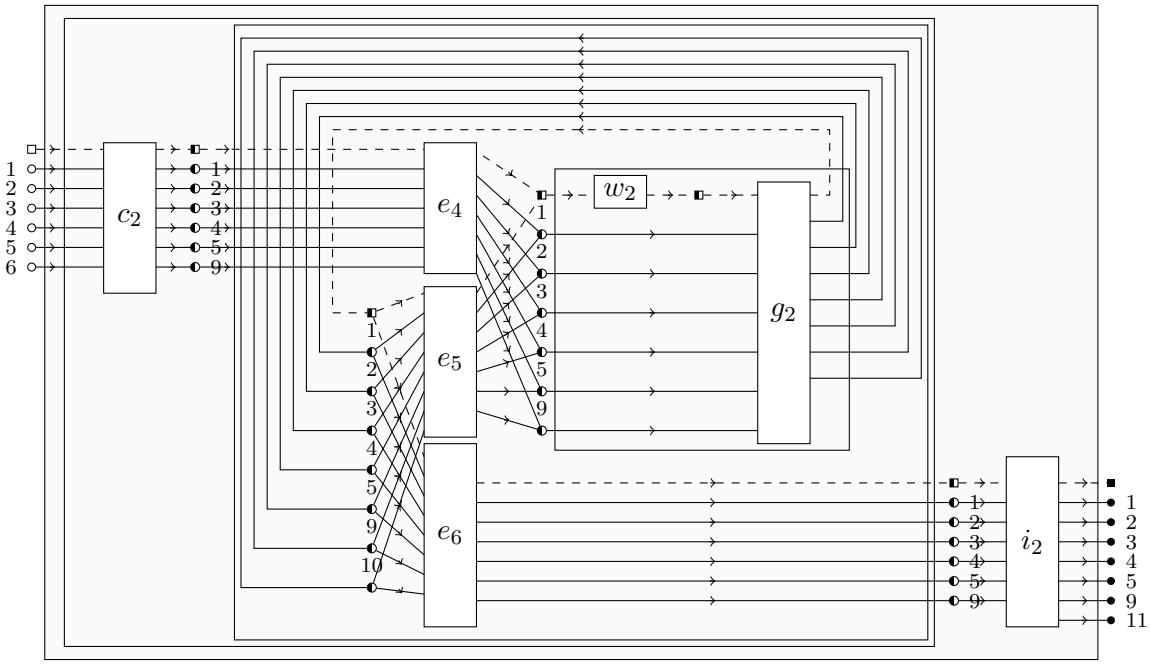


(c) Composite computon corresponding to the left path of the step function shown in Figure 37: $(c_1 \trianglerighteq_{\rho_3} ((w_1 \rhd_{\rho_1} g_1)*_{\rho_2})) \trianglerighteq_{\rho_4} s_1$

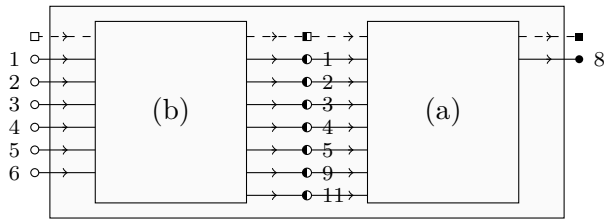
Figure 39: Constructing the composite computon for the left path of the step function shown in Figure 37.



(a) Branching computon that determines whether deployment succeeds or fails: $(x_2 \trianglerighteq_{\rho_5} s_2)?_{\rho_7}(d_2 \trianglerighteq_{\rho_6} f_2)$



(b) Total sequential computon that creates a change set, waits until the set is created and then inspects change set changes: $(c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})) \trianglerighteq_{\rho_{11}} i_2$



(c) Composite corresponding to the right path of the step function shown in Figure 37: $((c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})) \trianglerighteq_{\rho_{11}} i_2) \trianglerighteq_{\rho_{12}} ((x_2 \trianglerighteq_{\rho_5} s_2)?_{\rho_7}(d_2 \trianglerighteq_{\rho_6} f_2))$

Figure 40: Constructing the composite computon for the right path of the step function shown in Figure 37.

The tail-iterative computon $(w_2 \triangleright_{\rho_8} g_2)*_{\rho_9}$ is subsequently used as a right operand to define the total sequential computon $c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})$ wherein c_2 is a functional computon in charge of *creating the change set*. This newly constructed sequential composite is in turn used as left operand to construct the (even more complex) total sequential computon $(c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})) \trianglerighteq_{\rho_{11}} i_2$ wherein i_2 is a functional computon that *inspects change set changes* to determine whether any of the existing resources need to be deleted or whether the existing stack can be safely updated. The whole structure of $(c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})) \trianglerighteq_{\rho_{11}} i_2$ is shown in Figure 40(b).

The most complex composite for the right path of the step function from Figure 37 is constructed by taking the total sequential computon $(c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})) \trianglerighteq_{\rho_{11}} i_2$ and the branching computon $(x_2 \trianglerighteq_{\rho_5} s_2)?_{\rho_7}(d_2 \trianglerighteq_{\rho_6} f_2)$ as left and right operands, respectively, in order to yield $((c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})) \trianglerighteq_{\rho_{11}} i_2) \trianglerighteq_{\rho_{12}} ((x_2 \trianglerighteq_{\rho_5} s_2)?_{\rho_7}(d_2 \trianglerighteq_{\rho_6} f_2))$ which is the total sequential computon shown in Figure 40(c).

Once the left and right paths of the intended step function have been constructed, we compose them into the branching composite $((c_1 \trianglerighteq_{\rho_3} ((w_1 \triangleright_{\rho_1} g_1)*_{\rho_2})) \trianglerighteq_{\rho_4} s_1)?_{\rho_{13}}(((c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})) \trianglerighteq_{\rho_{11}} i_2) \trianglerighteq_{\rho_{12}} ((x_2 \trianglerighteq_{\rho_5} s_2)?_{\rho_7}(d_2 \trianglerighteq_{\rho_6} f_2)))$ which checks whether a new stack needs to be created (via the left path) or whether a change set needs to be created and inspected (via the right path). This branching computon is ultimately composed with k (i.e., a functional computon that determines whether a stack exists or not) into the total sequential computon $k \trianglerighteq_{\rho_{14}} ((c_1 \trianglerighteq_{\rho_3} ((w_1 \triangleright_{\rho_1} g_1)*_{\rho_2})) \trianglerighteq_{\rho_4} s_1)?_{\rho_{13}}(((c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})) \trianglerighteq_{\rho_{11}} i_2) \trianglerighteq_{\rho_{12}} ((x_2 \trianglerighteq_{\rho_5} s_2)?_{\rho_7}(d_2 \trianglerighteq_{\rho_6} f_2)))$ which captures the whole behaviour of the step function for infrastructure deployment shown in Figure 37. The structure of such a complex sequential composite is depicted in Figure 41 and its behaviour is shown in Figure 42.

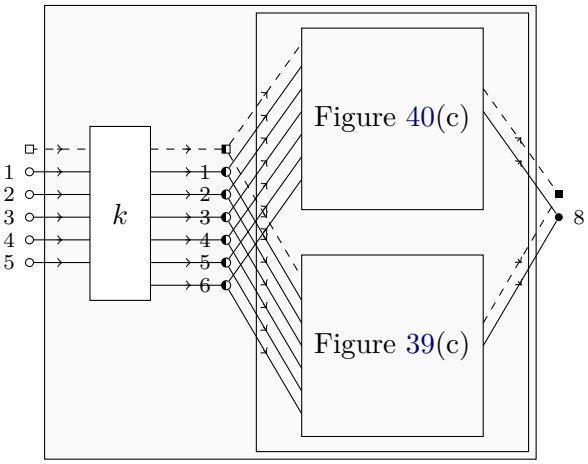


Figure 41: Total sequential computon corresponding to the step function shown in Figure 37: $k \trianglerighteq_{\rho_{14}} ((c_1 \trianglerighteq_{\rho_3} ((w_1 \triangleright_{\rho_1} g_1)*_{\rho_2})) \trianglerighteq_{\rho_4} s_1)?_{\rho_{13}}(((c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})) \trianglerighteq_{\rho_{11}} i_2) \trianglerighteq_{\rho_{12}} ((x_2 \trianglerighteq_{\rho_5} s_2)?_{\rho_7}(d_2 \trianglerighteq_{\rho_6} f_2)))$

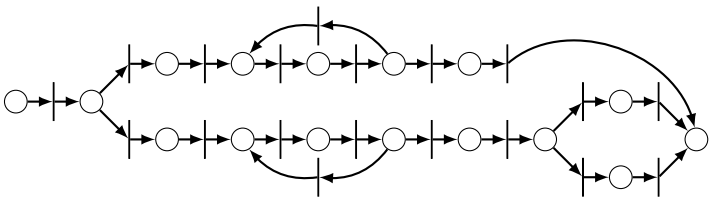


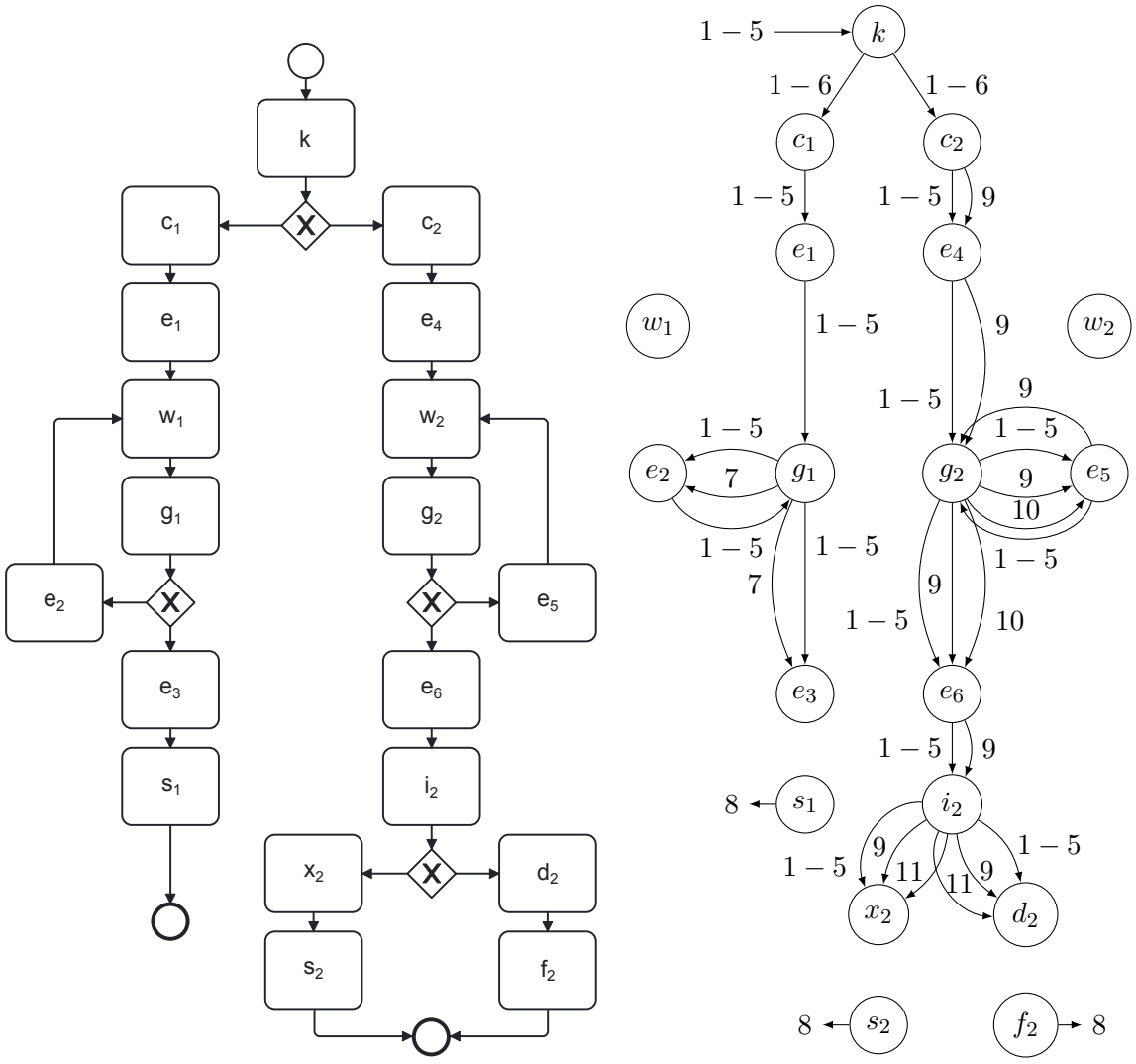
Figure 42: Behaviour of the step function shown in Figure 37, expressed as the Petri net $\mathcal{C}(\mathfrak{E}(k \trianglerighteq_{\rho_{14}} ((c_1 \trianglerighteq_{\rho_3} ((w_1 \triangleright_{\rho_1} g_1)*_{\rho_2})) \trianglerighteq_{\rho_4} s_1)?_{\rho_{13}}(((c_2 \trianglerighteq_{\rho_{10}} ((w_2 \triangleright_{\rho_8} g_2)*_{\rho_9})) \trianglerighteq_{\rho_{11}} i_2) \trianglerighteq_{\rho_{12}} ((x_2 \trianglerighteq_{\rho_5} s_2)?_{\rho_7}(d_2 \trianglerighteq_{\rho_6} f_2)))))$

Rather than presenting behaviour as a net under $\mathcal{N}$, we decide to use the functor $\mathcal{C} \circ \mathfrak{E}$ since control flow captures system behaviour comprehensively. Displaying the corresponding nets under $\mathcal{N}$ or $\mathcal{D}$ can be easily done using the mapping provided in Appendix A, which corresponds to the functorial descriptions from Definition 15 and Proposition 17. Here, we show the equivalent BPMN diagram (for control flow) and the corresponding DFG in standard notation (for data flow), using the graph transformation system described in Section 7.1. These models are far more expressive than Petri nets for our purpose which is just demonstrating the separation of control and data for model transformation.

By Proposition 29, the total sequential computon from Figure 41 is connected because there is information flow from every non-e-outport to either the unique ec-outport or the unique ed-outport. A glance at this figure reveals that computons are modular by construction, a consequence of compositionality that allows hiding the internals of complex composite

structures. For instance, Figure 41 hides the structure of the total sequential computon from Figure 40(c) which, in turn, hides the complexity of the composites 40(a) and 40(b). As per Proposition 29 and Corollaries 4 and 6, all the computons we deal with in this example are connected, including primitives (such as w_1) and composites (such as $w_1 \rhd_{\rho_1} g_1$).

Apart from modularity, another semantic consequence of our model is the separation of data and control which can be leveraged to analyse these two dimensions independently. For instance, we use the functor from Definition 11 to extract the CFG of the computon from Figure 41 which is then converted into its corresponding BPMN diagram via the graph transformation system proposed in Section 7.1.1 which, in turn, realises transformation without considering data flow at all (see Figure 43(a)). For data flow, we extract the DFG of the computon from Figure 41 through the functor from Definition 13, which is then converted into its equivalent DFG in standard notation via the functor described in Section 7.1.2 which do not consider control flow at all (see Figure 43(b)). Although the separation of concerns can be leveraged in other ways (e.g., to formally verify reachability of control flow only), the purpose of this section is just to demonstrate how control flow and data flow can be analysed independently for model transformation.



(a) BPMN Diagram (control flow).

(b) DFG in standard notation (data flow). For readability, we use a dash symbol between numbers n and m to indicate the presence of data flows $\xrightarrow{n}$, $\xrightarrow{n+1}$, ..., $\xrightarrow{m}$ such that $n \geq 1$ and $m > n$.

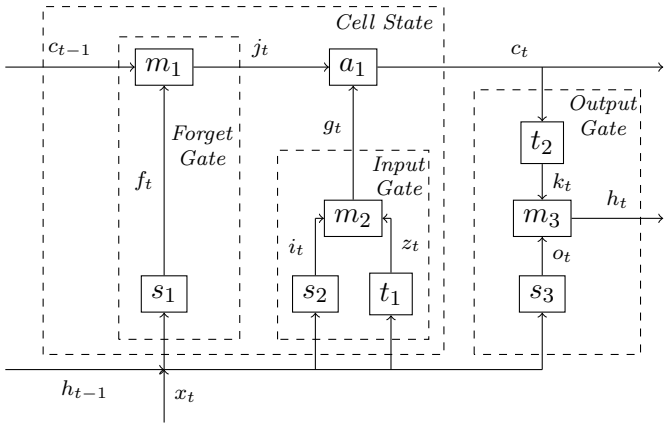
Figure 43: BPMN diagram and DFG in standard notation to respectively express the control flow and data flow structures of the composite shown in Figure 41.

In the actual implementation of the step function from Figure 37, data is appended at every step of the workflow execution [44]. This issue is derived from the fact that, apart from being non-compositional, AWS step functions do not separate control flow and data flow, so it is neccesary to pass a bundle of parameters as a single data item (i.e., as a JSON object). In other words, data flow is implicitly defined in the explicit workflow control flow.

Enabling separation of concerns through our model allows us to remove the data modelling issue of AWS step functions so as to pass only relevant data among computation units. For example, the functional computon w_1, which is just in charge of delaying computation, can be implemented as a sleep function to wait for a fixed amount of time, without processing any data at all. Also, the change set creation status can only be used to terminate the loop of the right path, without the need of passing it onto subsequent computations. Certainly, it is still possible to pass all data parameters at every step of the computation in the form of a single port colour (e.g., the JSON object colour), just as in the actual implementation. However, doing this could not be as expressive as the way we model data flow in our scenario.

7.3. Case Study 2: Compositional LSTMs

A Long Short-Term Memory (LSTM) [45] is a Recurrent Neural Network which has been widely used in the field of Deep Learning to learn long-term data dependencies. Typical applications of it include handwriting recognition, automatic language translation and writing generation. The key idea of a LSTM is to use a container to store and process information for an extended period of time, in order to allow for constant error flow during training. Such a container, known as a memory cell, is controlled by three types of interacting gates: an input gate, a forget gate and an output gate, which respectively decide what information can be added to, removed from and sent out of the cell. Particularly, the outcome of the forget and input gates is added so as to produce an updated cell state that can be further consumed by other memory cells. The abstract, high-level schematic representation of an individual memory cell is shown in Figure 44(a). Such a scheme is abstract because the specific behaviour of the components involved is not provided and it is high-level because each component can contain further internal components which are not exposed to the outside world.



(a) Abstract, high-level architecture.

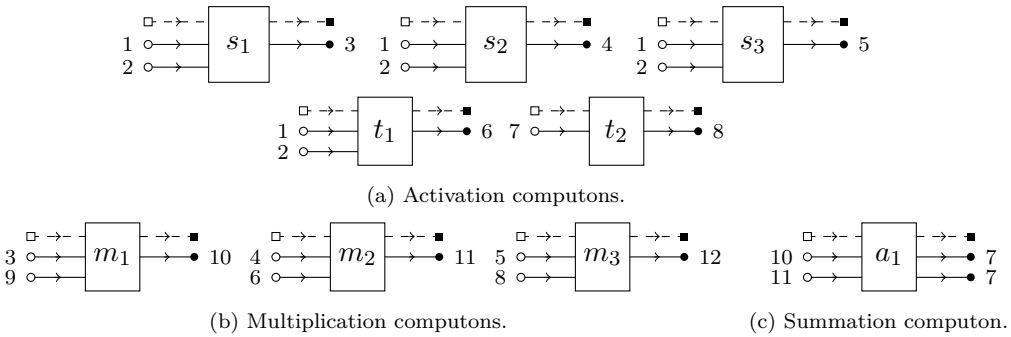
Colour	Variable
1	h_{t-1}
2	x_t
3	f_t
4	i_t
5	o_t
6	z_t
7	c_t
8	k_t
9	c_{t-1}
10	j_t
11	g_t
12	h_t

(b) Mapping from colours to variables.

Figure 44: Conceptual representation of a memory cell.

A memory cell can be constructed compositionally using the computon model. Such a construction is done in a bottom-up manner, starting from the most elementary units of computation of the memory cell, which are displayed as non-dashed squares in Figure 44(a). As we are dealing with a model of high-level computation, the specific computation details of such squares are not required, so non-dashed squares can naturally be defined as functional computons, i.e., as black boxes that can perform any kind of computation (see Figure 45). Typically, for a concrete memory cell, the functional computons s_i, t_j, m_i and a_1 would be required to compute sigmoid, hyperbolic tangent, Hadamard and element-wise summation

functions, respectively. For the sake of "high-levelness", we just say s_i and t_j are activation computons, while m_i and a_1 are multiplication and summation computons, respectively.



(a) Activation computons.

(b) Multiplication computons.

(c) Summation computon.

Figure 45: Functional computons for the compositional construction of a memory cell.

Strictly speaking, every ed-port shown in Figure 45 must have the same colour (i.e., the vector colour) since data being moved within a cell corresponds to a vector of the same type. But, for clarity and demonstration purposes, we consider 12 different colours, each corresponding to the type of each variable shown in Figure 44(a). The mapping from colours to variables is presented in Figure 44(b).

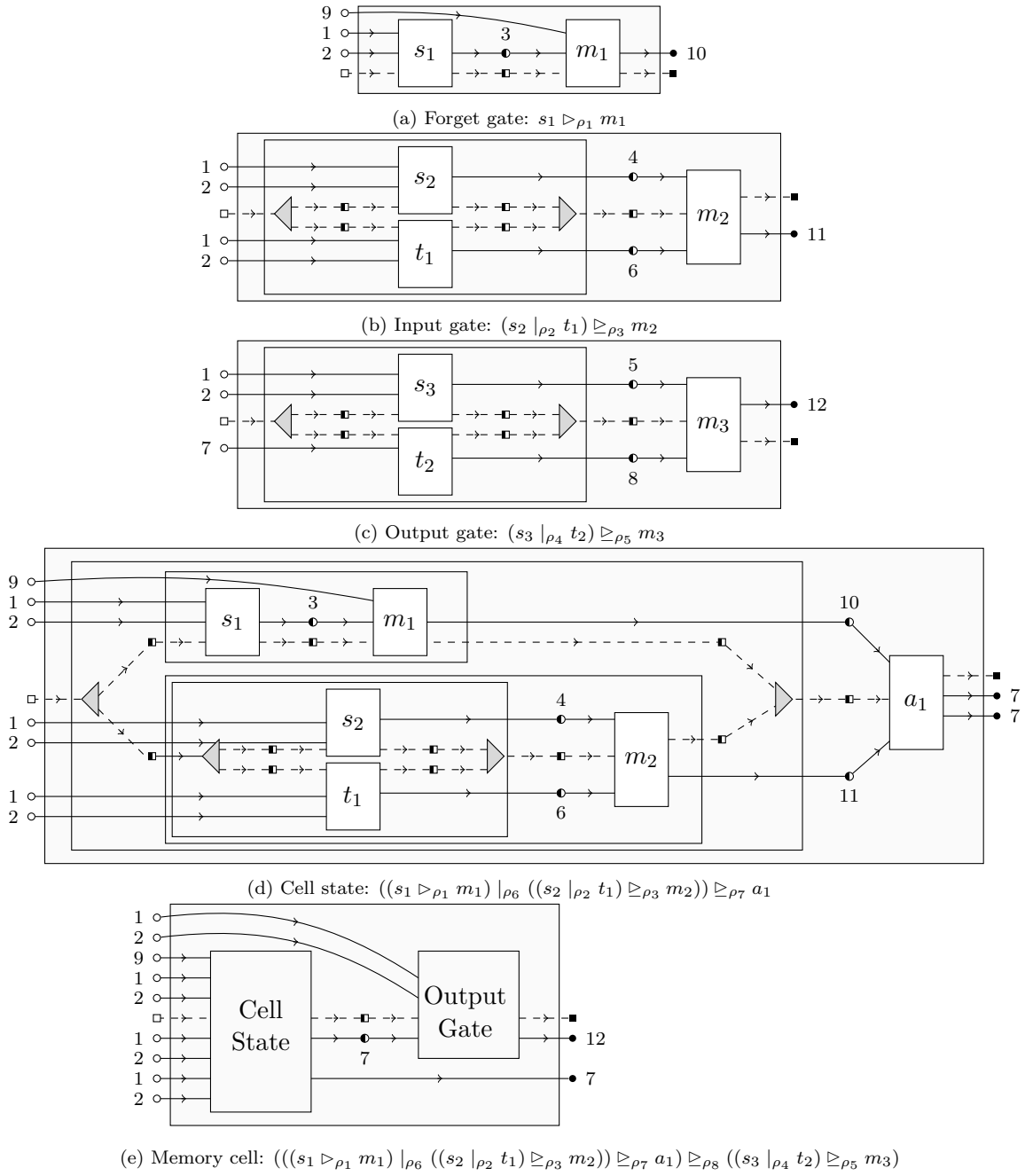
Using a long-term memory vector c_{t-1} (of colour 9), a short-term memory vector h_{t-1} (of colour 1) and an external input/predictor vector x_t (of colour 2), the whole memory cell performs some computation and then returns a new long-term memory vector c_t (of colour 7) and a new short-term memory vector h_t (of colour 12) which, in turn, can used by other memory cells to perform subsequent computations.

For the (high-level) computation of a memory cell, the inner forget gate computes the activation computon s_1 in terms of h_{t-1} and x_t to yield the vector f_t (of colour 3). The result f_t and the long-term memory vector c_{t-1} are then multipled via m_1 to obtain j_t (of colour 10). To capture this computation, Figure 46(a) shows that the forget gate is characterised as the partial sequential computon $s_1 \rhd_{\rho_1} m_1$.

Constructing the input gate is done differently since it requires two copies of h_{t-1} and two copies of x_t to simultaneously compute the activation computons s_2 and t_1, in order to produce the vector i_t (of colour 4) and the vector z_t (of colour 6). Such results are then multiplied through m_2 to produce the vector g_t (of colour 11). As s_2 and t_1 are computed in parallel before m_2, the input gate is naturally characterised as the total sequential computon $(s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2$ whose structure is depicted in Figure 46(b). Figure 46(c) shows that the structure of the output gate is similar to that of $(s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2$ so the output gate is precisely the total sequential computon $(s_3 \mid_{\rho_4} t_2) \trianglerighteq_{\rho_5} m_3$ whose left and right operands are the p-sync computon $s_3 \mid_{\rho_4} t_2$ and the multiplication computon m_3, respectively. As data is not shared within a p-sync computon, $s_3 \mid_{\rho_4} t_2$ has the same ed-outports as s_3 and t_2, namely the vector o_t (of colour 5) and the vector k_t (of colour 8). The only ed-outport of $(s_3 \mid_{\rho_4} t_2) \trianglerighteq_{\rho_5} m_3$ is h_t (of colour 12) which represents the new short-term memory value to be passed onto subsequent memory cells.

As they have no dependencies among them, the forget gate composite $s_1 \rhd_{\rho_1} m_1$ and the input gate composite $(s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2$ are composed into the p-sync computon $(s_1 \rhd_{\rho_1} m_1) \mid_{\rho_6} ((s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2)$ whose ed-ports are inherited from $s_1 \rhd_{\rho_1} m_1$ and $(s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2$. To enable the functional computon a_1 to element-wisely add j_t and g_t, the newly constructed p-sync and a_1 are composed into the total sequential computon $((s_1 \rhd_{\rho_1} m_1) \mid_{\rho_6} ((s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2)) \trianglerighteq_{\rho_7} a_1$ which, by Proposition 30, has the same ed-outports as a_1, i.e., the next cell state c_t (of colour 7) – see Figure 46(d). As c_t is sent outside the memory cell and is further required by the output gate, Figure 45(c) shows that a_1 produces two copies of it. [24]

[24] We decide to model the summation computon a_1 in this way in order to avoid the unnecessary burden of introducing extra functional computons to express data replication.



(a) Forget gate: $s_1 \rhd_{\rho_1} m_1$

(b) Input gate: $(s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2$

(c) Output gate: $(s_3 \mid_{\rho_4} t_2) \trianglerighteq_{\rho_5} m_3$

(d) Cell state: $((s_1 \rhd_{\rho_1} m_1) \mid_{\rho_6} ((s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2)) \trianglerighteq_{\rho_7} a_1$

(e) Memory cell: $(((s_1 \rhd_{\rho_1} m_1) \mid_{\rho_6} ((s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2)) \trianglerighteq_{\rho_7} a_1) \trianglerighteq_{\rho_8} ((s_3 \mid_{\rho_4} t_2) \trianglerighteq_{\rho_5} m_3)$

Figure 46: Subfigures (a)-(d) are composite computons that serve as building blocks to construct the memory cell depicted in (e). Each of them model a dotted box with the same name in Figure 44(a). For example, the composite (a) models the forget gate by providing ed-inports of colour 1, 2 and 9 which respectively correspond to h_{t-1}, x_t and c_{t-1} (according to Figure 44(b)). The functional computon s_1 uses h_{t-1} and x_t to produce a 3-coloured data item (i.e., f_t) which is sequentially passed to the functional computon m_1. The output of m_1 is the output of (a), viz., a 10-coloured data item corresponding to j_t. Evidently, (a) has a single ec-inport to allow the forget gate to be invoked from outside, and a ec-outport to pass control to other computons. For example, in (d), (a) passes control to the join computon used to synchronise the composites (a) and (b).

The final step in our compositional construction is forming the composite structure that represents the whole memory cell. Figure 46(e) shows that such a structure is precisely the partial sequential computon $(((s_1 \rhd_{\rho_1} m_1) \mid_{\rho_6} ((s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2)) \trianglerighteq_{\rho_7} a_1) \rhd_{\rho_8} ((s_3 \mid_{\rho_4} t_2) \trianglerighteq_{\rho_5} m_3)$ in which the composite $((s_1 \rhd_{\rho_1} m_1) \mid_{\rho_6} ((s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2)) \trianglerighteq_{\rho_7} a_1$ and the output gate $(s_3 \mid_{\rho_4} t_2) \trianglerighteq_{\rho_5} m_3$ are the left and right operands, respectively. For visualisation purposes, we treat such operands as black boxes.

The resulting memory cell composite can also be treated as a black box for defining even more complex composites such as a complete Recurrent Neural Network. Black boxing is

possible because computons are modular by construction so that their internals can be hidden without any side effects. In this scenario, the memory cell hides the complexity of the total sequential computons from Figures 46(c) and 46(d). By Proposition 29 and Corollary 3, all the composites and primitive computons we deal with are connected in the sense of Definition 6. This can be easily verified by observing in Figure 46 that there is information flow from every non-e-outport to either an ec-outport or an ed-outport. For example, there is information flow from the ec-inport of the p-sync computon of the output gate to the unique ec-outport of m_3. Although a purely sequentially-driven construction could have been used instead, we decide to rely upon p-sync computons so as to emphasise the parallel nature of computations occuring within a memory cell. Using p-sync composites instead of p-async constructions allows us to semantically express the fact that data needs to be synchronised (via control) before being consumed by subsequent computations.

The behaviour of the whole memory cell is displayed in Figure 47, which corresponds to the net under $\mathcal{C} \circ \mathfrak{E}$ of the composite from Figure 46(e). Like in the previous example, we decide to just display the net that encapsulates control flow since it comprehensively captures the computational behaviour of the memory cell. The corresponding nets under $\mathcal{N}$ and under $\mathcal{D}$ can be easily constructed using the mapping from Appendix A, which captures the mapping given by the functorial constructions from Definition 15 and Proposition 17. In Figure 48, we show the equivalent BPMN diagram (for control flow) and the DFG in standard notation (for data flow) using the graph transformation system described in Section 7.1.

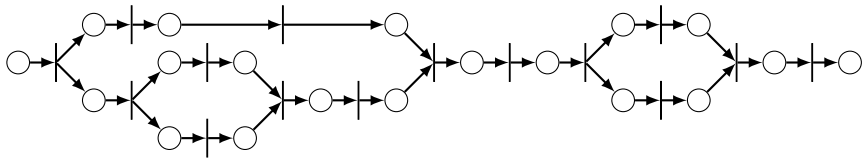
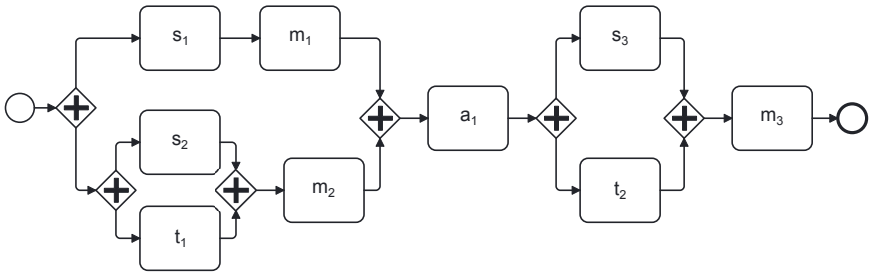
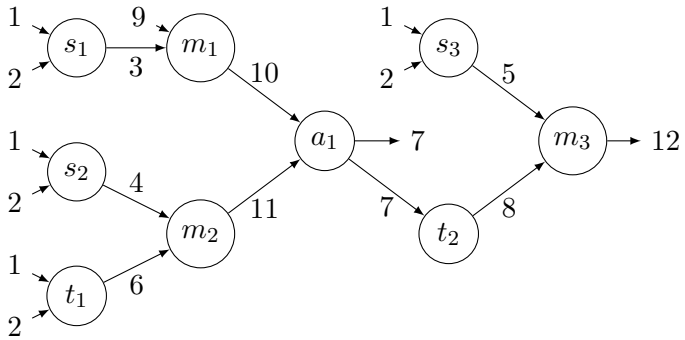


Figure 47: Behaviour of the memory cell computon presented in Figure 46(e), expressed as the Petri net $\mathcal{C}(\mathfrak{E}((((s_1 \rhd_{\rho_1} m_1) \mid_{\rho_6} ((s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2)) \trianglerighteq_{\rho_7} a_1) \trianglerighteq_{\rho_8} ((s_3 \mid_{\rho_4} t_2) \trianglerighteq_{\rho_5} m_3)))$



(a) BPMN Diagram (control flow).



(b) DFG in standard notation (data flow). For readability and since they do not produce or receive any data, we omit the nodes corresponding to fork or join computons.

Figure 48: BPMN diagram and DFG in standard notation to respectively express the control flow and data flow structures of the memory cell $(((s_1 \rhd_{\rho_1} m_1) \mid_{\rho_6} ((s_2 \mid_{\rho_2} t_1) \trianglerighteq_{\rho_3} m_2)) \trianglerighteq_{\rho_7} a_1) \rhd_{\rho_8} ((s_3 \mid_{\rho_4} t_2) \trianglerighteq_{\rho_5} m_3)$ depicted in Figure 46(e).

The process described in Section 7.1 for transforming a CFG into a BPMN diagram does not consider data flow at all, whereas the process for transforming a DFG does not consider control flow at all. Although the separation of concerns can be leveraged for other purposes (e.g., to formally verify termination of control flow only), the purpose of this section is just to demonstrate how control flow and data flow can be analysed independently for model transformation.

Figure 48 demonstrates that, as every composite computon captures both control flow and data flow within a single compositional structure, the proposed model can be perceived as a unification (or conciliation) of workflow control flow languages [7] and data flow languages [38] within a single compositional setting.

8. Related Work

In this section, we present the related work of our proposal, namely related compositional approaches and component models that separate concerns.

The CommUnity component model [46] separates computation from interaction and provides categorical semantics for architectural configurations. At its core, software components (called designs) are objects in a category, interrelated through morphisms. The most well-known structure preserving relationships are the so-called superposition morphisms, which allow embedding designs into more complex ones, similar to the action of computon morphisms. The difference is that our morphisms force computons to interact exclusively at the boundaries and that our colimit constructions define explicit control flow structures. CommUnity colimits serve to merge (or amalgamate) designs compositionally à la Definition 8, leaving control flow implicit without any support to separate it from data.

Prosave [47] is a design language built on top of the ProCom component model, which was inspired on [48] to allow the definition of nested structures of interconnected components. Like computons, Procom components are passive units of computation with explicit separation of control ports and data ports. Despite of this similarity, Procom is not compositional since it does not provide algebraic operators to perform control-based composition, but just informal programming constructs for connecting ports either directly or indirectly. Indirect connection is done through so-called connectors which establish control or data flow interaction between components via message passing. As the model is not compositional, Procom composites do not offer a clear separation of concerns like their internal components. They rather have data ports only where both control flow and data flow terminate.

SCADE [49] is a similar component model which integrates an imperative language (i.e., Esterel [50]) and a functional language (i.e., Lustre [51]) to define control flow and data flow, respectively. Particularly, so-called Safe-State Machines (SSMs) model the discrete control part of a system, whereas Lustre blocks serve to continuously process data. Like Prosave, SCADE does not provide formal operators for defining control-based composite blocks, but just programming constructs to non-compositionally assemble a system.

In the same line of work, [14, 52, 53] describe a component model that provides two orthogonal dimensions to manage control flow and data flow separately. The model encapsulates control since it offers composition operators to define sequential, parallel or branching composites in a hierarchical, bottom-up manner. Unfortunately, the semantics of the model is semi-formal [9, 54] so it is not possible to precisely determine whether the model is fully compositional or not. Also, components do not have separate ports for data and control, but just control ports. Consequently, the data dimension is implicitly defined in the underlying composition mechanism whose goal is to build complex workflows from simpler ones.

Workflow Nets (WF-nets) [7, 55] provide support for modelling workflow processes in the form of control-driven computations. As they offer well-founded semantics built upon Petri nets, WF-nets formalise the notion of workflow graphs which are traditionally specified through industry-oriented languages such as UML diagrams, Event-driven Process Chains or the BPMN notation. WF-nets do not separate control from data and do not provide formal

operators for explicitly and compositionally defining sequential, parallel, branching or iterative composites. The issue of the separation of concerns is resolved by RWFN-nets [56] which unify extended WF-nets and so-called resource nets for separating the process and resource perspectives of a workflow. Although the model provides a clear separation of concerns, there is not a clear distinction between input and output data, and composition is not algebraically defined. Therefore, RWFN-nets do not separate data and control compositionally. Other Petri net based approaches, for workflow construction, that non-compositionally separate control flow and data flow are the functor model [57], extended-time nets [58], the FunState model [59] and dual flow nets [12].

Existing compositional approaches built upon Petri net foundations rely on the notion of open interfaces to the external world. Specially designated open places are particularly used by open Petri nets (ONets) [2] to construct complex behaviours from simpler ones. In this framework, ONet composition is realised by gluing the output places of one net with the input places of another. As this composition mechanism is characterised as a pushout in a categorical setting [60], ONets are compositional. An ONet morphism resembles a computon morphism in the sense that input and output places can be preserved upon transformation (see Proposition 5). Nevertheless, like Definition 8, a pushout operation just serves for merging two ONets via a common object so that there are no specific operators for explicitly defining sequential, parallel, branching or iterative composites (i.e., ONets do not encapsulate explicit control flow). Petri box calculus [8], Open WF-nets [61], Petri nets with interface [62], nets with boundaries [63] and Petri net components [64] also rely on the notion of open interfaces. Like ONets, all these Petri-net-based approaches do not separate data from control.

Although they do not separate concerns, Whole-grain Petri nets [65] deserve a mention since, unlike classical Petri net theory and like computons, they abolish the traditional notion of multisets of places, typically expressed as a free commutative monoid $S^{\oplus}$ on a set S of places (cf. Definition 14). Accordingly, they also work upon a similar categorical scheme to **Comp**, in order to define concrete instances of Whole-grain nets (cf. [65, 66]). The difference is that **Comp** has objects that enable computons to have a clear distinction between control and data ports. Another difference is that our theory identifies particular classes of computon objects that can be used as building blocks to define more complex computons through sequencing, parallelising, branching or iteration operations. Although primitive computons are isomorphic to Whole-grain corollas, Whole-grain Petri nets do not distinguish between different types of corollas (e.g., join or fork corollas).

Within the realm of related compositional models, we also find string diagrams [67, 68, 69], whose origins trace back some 50 years to Penrose's graphical notation for tensor networks [70], later given a sound and complete categorical foundation by Joyal–Street [71]. To date, such diagrams have been widely applied across several domains, including quantum mechanics [72], electrical engineering [73], control theory [74] and natural language processing [75], just to name a few. String diagrams are becoming increasingly popular because they offer well-founded syntax to graphically represent symmetric monoidal categories, where boxes represent processes/morphisms and wires express inputs or outputs for those processes [76, 77, 78, 25, 79, 80, 81, 82, 83, 84, 85]. In this model, string diagrams can be composed sequentially via morphism composition [83], in parallel via monoidal product [76] and into iterative structures via traces [86, 85]. Extensions to support probabilistic branching have also been developed [87, 88]. Since sequential composition is done by totally matching outputs with inputs (or domain with codomain) and there are not distinguished wires for representing control, it follows that, unlike computons, not every string diagram can be composed sequentially with one another (cf. Theorem 1). Moreover, there is no distinction between control flow and data flow, as evidenced by the uniform treatment of wires to carry data only [89]. Some efforts have been made to colouring wires but not to explicitly separate concerns [90]. The separation of concerns has more recently been addressed through tape diagrams [91, 92, 93], which are similar in spirit to [94], but that allow the formation of nested string diagrams where an inner layer represents data flow and an outer part expresses control flow. Technically, the control/data interplay

is captured by the laws of rig categories with finite biproducts, featuring two symmetric monoidal structures on relations, $(Rel, \oplus, 0)$ and $(Rel, \otimes, 1)$, which respectively serve to model control and data flow. Both monoidal structures can admit traces to model iteration [92] and, like string diagrams, total sequencing, asynchronous parallelising and branching are supported. Unfortunately, the notions of synchronous parallelising and partial sequencing are not considered, since it is assumed that data always follows control [93]. As a result, tape diagrams lack the expressive power to model scenarios where asynchronous data can delay process execution independently of control. Moreover, no independent extraction of control flow and data flow graphs from tape diagrams has yet been attempted in a categorical setting. The computon model accommodates this through the functors presented in Section 2.4.

The colimit-based composition operators described in Section 6 might resemble works from category-theoretic cybernetics. For example, [95] proposes the use of general colimits to bind components of complex systems. Although binding is hierarchical and bottom-up, as in the computon model, the proposed operations do not define explicit control flow structures capable of operationally enacting what the authors describe as causal and informational interactions, which correspond to those enabled by computons. Composition machines [24] attempt to address this limitation by composing components into total sequential structures. Rather than using colimits, composition is realised by morphism composition, applied dynamically to generate spaces of sequential composites in discrete time. Unlike computons, there are no operations for forming branching, iterative or parallel composites. A glance at Figure 46(d) in Section 7 reveals that the structure of a composite computon is like a membrane in which other computons reside and that can be part of another membrane/composite. An edge connected to/from a composite's e-port is akin to a fibre which can traverse other membranes, as long as the e-port it is connected to/from does not become an i-port. This analogy resembles the structural organisation of a P-system [96] where membranes are delimiting compartments of multisets of objects that evolve according to bio-inspired rules. Other models resembling this structural analogy include Architectural Design Rewriting [97], Fractal [98] and Robin Milner's bigraphs [99]. Unfortunately, all these models do not separate data and control, and some of them just consider these dimensions implicitly.

9. Conclusions and Future Directions

In this paper, we presented a model of high-level computation in which computons are first-class semantic entities which structurally possess a number of computation units that can be connected to/from two types of ports: control ports and data ports. Computons are objects in a functor category, denoted $\mathbf{Set}^{\mathbf{Comp}}$, where two major classes of objects reside. The first class is that of trivial computons which have just ports and no computation units. The second class pertains to primitive computons which are fully connected entities in the sense they have a unique computation unit to which all ports are attached. These two classes serve as building blocks to define complex computons via category-theoretic operations. We particularly presented separate operations to inductively form sequential, parallel, branching and iterative composite computons. In Section 6, we proved that all of them are connected in the sense of Definition 6, meaning there is a sequence of information flows from every non e-outport to some e-outport. As the model is compositional, composites exhibit the same properties as their constituents, i.e., they have the same structure with a clear separation of control flow and data flow.

Generally speaking, both control flow and data flow are inextricably present in any classical high-level computation (e.g., a workflow process), so it is crucial to separately reason about them for verification, maintainability and optimisation purposes. For example, in Section 7, we leveraged the separation of concerns of the proposed model to show how control flow can be transformed into a BPMN diagram without analysing data flow at all, and how data flow can be converted into a DFG in standard notation without considering control flow at all. Evidently, model transformation is not the only way of exploiting the separation of concerns

of our proposal. By leveraging the fact that the behaviour of a computon can be expressed as a token game, it is also possible to use standard Petri net tools or relevant graph-based analysis techniques to separately verify computing properties, such as reachability of control flow only or data flow only. Taking advantage of graph-based techniques can also enable an optimal implementation in which functional computons exchange data decentrally while composites coordinate control flow hierarchically [14]. Although our model does not consider explicit structures for data processing (e.g., map-reduce or filter constructs), because data flow is ultimately governed by control flow, we acknowledge that introducing them is important to increase the expressivity of composite computons. However, doing this in a compositional manner requires further investigation.

Enabling compositionality is also important to induce modularity which is a well-known feature for reusing computations at scale. Modularity does not imply compositionality because modules can be constructed in many different ways (not necessarily algebraically). When an algebraic composition mechanism is used to realise this feature, computation properties are preserved across all composition levels. In our proposal, the separation of control flow and data flow is one of such properties. Thus, as computons only interact through their respective e-ports, composite computons can be perceived as modular black-boxes that encapsulate control and data flow structures. Although branching is supported by our theory, not every pair of computons is a candidate for defining a branching structure, as described in Section 6.3.

In Section 6.4, we showed that a head- or a tail-iterative structure can be formed for any connected computon (see Theorems 4 and 5). Likewise, in Sections 6.1 and 6.2, we proved that any pair of connected computons can always be composed sequentially (see Theorem 1) or in parallel (see Theorems 2 and 3) regardless of the data they require or produce. This is because, intuitively, a computon has at least one ec-outport that can always be matched with at least one ec-inport of another. Matching all the e-outports of one computon with all the e-inports of another one gives rise to total sequential composition which, to the best of our knowledge, is the *de facto* way of sequencing computations nowadays (cf. [100]).

In this paper, we argue that sequencing is a particular form of merging because the former can be expressed in terms of the latter. Particularly, in our proposal, merging corresponds to a pushout operation in $\mathbf{Set}^{\mathbf{Comp}}$ (see Definition 8), while sequencing is characterised as a pushout with restrictions in the same category (see Definition 27). As sequencing cannot only be done totally but also partially, our sequencing mechanism is more general than those prevailing in the existing literature. Partial composition entails that non-matching e-ports are preserved across every composition level (e.g., see Figures 12 and 21).

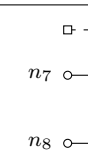
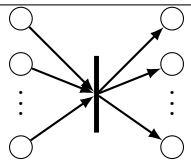
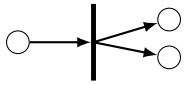
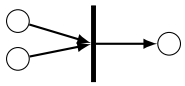
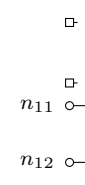
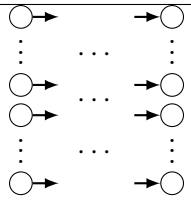
If computons are seen as relations from e-inports to e-outports, our composition mechanism provides the basis to redefine the current notion of composition of relations which states that the composite $S \circ R$ of $R \subseteq X \times Y$ and $S \subseteq Y \times Z$ is given by $\{(x, z) \mid \exists y[R(x, y) \ \wedge \ S(y, z)]\}$. Since $S \circ R$ is a subset of $X \times Z$, it is evident that some relations in $X \times Y$ and in $Y \times Z$ are lost. By resorting to the foundations laid in this paper, a more structure-preserving definition emerges: $(S \circ R) \cup [(X \times Y) \setminus (S \circ R)] \cup [(Y \times Z) \setminus (S \circ R)]$. Thus, rather than being a subset of $X \times Z$, a composite relation would be a subset of $(X \cup Y) \times (Y \cup Z)$. In the future, we would like to further investigate this new notion derived from the foundations of partial sequential composition.

Defining computons as preorders in a categorical setting can be achieved by borrowing ideas from resource theories [25]. We hypothesise there are symmetric monoidal categories in which computons are morphisms and ports are objects. Defining categories of this sort can be helpful to study the operational semantics of composite computons through the arrow of time. Particularly, *v-categories* and *v-profunctors* can provide theoretical underpinnings for formally answering specific questions about the execution of computons. Another potential direction is to study the operational semantics of computons from the lenses of polynomial-style finite-set configurations and etale maps in the context of Whole-grain Petri nets and processes. Studying operational semantics from different angles is possible due to the separation between composition and execution semantics of the proposed model.

Appendix A. Mapping Between Petri Net Syntax and Computon Syntax

Table A.1 presents a mapping from Petri net syntax to computon syntax, given by any of the three functorial constructions presented in Section 3, which is useful to discuss the operational semantics of computons. A glance at this table reveals that, in general, places with no incoming arrows correspond to e-inports, whereas places with no outgoing arrows correspond to e-outports. This reflects the fact that e-inports and e-outports receive and send information from and to the external world, respectively.

Table A.1: Mapping from computon syntax to Petri net syntax where n_j is a natural number greater than zero for all $j = 1, \ldots, 14$.

		Computon syntax	Petri net syntax
Control	ec-inport		
	ec-outport		
	ec-inoutport		
	ic-port		
Data	ed-inport		
	ed-outport		
	ed-inoutport		
	id-port		
Trivial Computon			
Functional Computon			
Fork Computon			
Join Computon			
Composite Computon			

References

[1] Stavros Tripakis, Christos Stergiou, Chris Shaver, and Edward A. Lee. A modular formal semantics for Ptolemy. *Mathematical Structures in Computer Science*, 23(4): 834–881, 2013.

[2] John C. Baez and Jade Master. Open Petri nets. *Mathematical Structures in Computer Science*, 30(3):314–341, 2020.

[3] Damian Arellanes. Models of High-Level Computation. *Frontiers in Computer Science*, 7, 2025.

[4] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, 2011.

[5] Farhad Arbab. Composition of Interacting Computations. In *Interactive Computation: The New Paradigm*, pages 277–321. Springer, Berlin, Heidelberg, 2006.

[6] Kung-Kiu Lau and Simone Di Cola. *An Introduction to Component-based Software Development*. World Scientific, Singapore, 1st edition, 2017.

[7] W. M. P. Van der Aalst. The application of Petri-nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[8] Eike Best, Raymond Devillers, and Maciej Koutny. The Box Algebra=Petri Nets+Process Expressions. *Information and Computation*, 178(1):44–100, 2002.

[9] Damian Arellanes and Kung-Kiu Lau. Evaluating IoT service composition mechanisms for the scalability of IoT systems. *Future Generation Computer Systems*, 108:827–848, 2020.

[10] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[11] Robin Message. Programming for humans: a new paradigm for domain-specific languages. Technical Report UCAM-CL-TR-843, University of Cambridge, 2013.

[12] Mauricio Varea, Bashir M. Al-Hashimi, Luis A. Cortés, Petru Eles, and Zebo Peng. Dual Flow Nets: Modeling the control/data-flow relation in embedded systems. *ACM Transactions on Embedded Computing Systems*, 5(1):54–81, 2006.

[13] Edmund Clarke, Anubhav Gupta, Himanshu Jain, and Helmut Veith. Model Checking: Back and Forth between Hardware and Software. In *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, Lecture Notes in Computer Science, pages 251–255. Springer, Berlin, Heidelberg, 2008.

[14] Damian Arellanes, Kung-Kiu Lau, and Rizos Sakellariou. Decentralized Data Flows for the Functional Scalability of Service-Oriented IoT Systems. *The Computer Journal*, 66 (6):1477–1506, 2023.

[15] Wim Vanderbauwhede. Separation of Data flow and Control flow in Reconfigurable Multi-core SoCs using the Gannet Service-based Architecture. In *2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pages 326–333, 2007.

[16] Tabea Bordis, Tobias Runge, Alexander Kittelmann, and Ina Schaefer. Correctness-by-Construction: An Overview of the CorC Ecosystem. *ACM SIGAda Letters*, 42(2):75–78, 2023.

[17] Damian Arellanes and Kung-Kiu Lau. Analysis and Classification of Service Interactions for the Scalability of the Internet of Things. In *IEEE ICIOT*, pages 80–87, 2018.

[18] Damian Arellanes and Kung-Kiu Lau. Algebraic Service Composition for User-Centric IoT Applications. In Dimitrios Georgakopoulos and Liang-Jie Zhang, editors, *Internet of Things – ICIOT 2018*, volume 10972 of *Lecture Notes in Computer Science*, pages 56–69. Springer International Publishing, Cham, 2018.

[19] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 12th edition, 2018.

[20] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39, Berlin, Heidelberg, 1987. Springer.

[21] Gordon D Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.

[22] Chander Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[23] Daniel Jackson. Alloy: a language and tool for exploring software designs. *Communications of the ACM*, 62(9):66–76, 2019.

[24] Damian Arellanes. Composition Machines: Programming Self-organising Software Models for the Emergence of Sequential Program Spaces. In *Theoretical Aspects of Software Engineering*, pages 19–37. Springer, 2024.

[25] Bob Coecke, Tobias Fritz, and Robert W. Spekkens. A mathematical theory of resources. *Information and Computation*, 250:59–86, 2016.

[26] David I. Spivak. Database queries and constraints via lifting problems. *Mathematical Structures in Computer Science*, 24(6):1–55, 2014.

[27] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1):181–224, 1993.

[28] Claudia Ermel and Alfio Martini. A Taste of Categorical Petri Nets. Technical Report 96-9, TU Berlin, 1996.

[29] José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, 1990.

[30] Luca Fossati, Raymond Hu, and Nobuko Yoshida. Multiparty Session Nets. In *9th International Symposium on Trustworthy Global Computing*, pages 112–127. Springer, 2014.

[31] John C. Baez, Fabrizio Genovese, Jade Master, and Michael Shulman. Categories of Nets. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2021.

[32] Jade Master. Petri nets based on Lawvere theories. *Mathematical Structures in Computer Science*, 30(7):833–864, 2020.

[33] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing net computations and processes. In *4th Annual Symposium on Logic in Computer Science*, pages 175–185, 1989.

[34] Vladimiro Sassone. On the category of Petri net computations. In *6th International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, pages 334–348, 1995.

[35] Piergiulio Katis, N. Sabadini, and R. F. C. Walters. Span(Graph): A categorical algebra of transition systems. In *International Conference on Algebraic Methodology and Software Technology*, pages 307–321, Berlin, Heidelberg, 1997. Springer.

[36] Bob Coecke. Compositionality as We See It, Everywhere Around Us. In *The Quantum-Like Revolution*, pages 247–267. Springer International Publishing, Cham, 2023.

[37] OMG . Business Process Model And Notation (BPMN), 2011. URL https://www.omg.org/spec/BPMN/2.0/.

[38] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[39] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, 2012.

[40] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.

[41] Bruce Silver. *BPMN Method and Style: With BPMN Implementer's Guide*. Cody-Cassidy Press, 2nd edition, 2011.

[42] Oracle Corporation. Developing Business Processes with Oracle Business Process Composer. Technical Report 12.2.1.1, 2016.

[43] Amazon Web Services. AWS Step Functions, 2023. URL https://aws.amazon.com/step-functions/.

[44] Marcilio Mendonca. Using AWS Step Functions State Machines to Handle Workflow-Driven AWS CodePipeline Actions, 2017. URL https://aws.amazon.com/blogs/devops/using-aws-step-functions-state-machines-to-handle-workflow-driven-aws-codepipeline-actions/.

[45] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[46] José L. Fiadeiro. *Categories for Software Engineering*. Springer-Verlag, Berlin Heidelberg, 2005.

[47] Tomas Bures, Jan Carlson, Ivica Crnkovic, Severine Sentilles, and Aneta Vulgarakis. ProCom — the Progress Component Model Reference Manual. 2008.

[48] Kaj Hanninen, Jukka Maki-Turja, Mikael Nolin, Mats Lindberg, John Lundback, and Kurt-Lennart Lundback. The Rubus component model for resource constrained real-time systems. In *3rd International Symposium on Industrial Embedded Systems*, pages 177–183, 2008.

[49] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. SCADE 6: A formal language for embedded critical software development (invited paper). In *11th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–11, 2017.

[50] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[51] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[52] Kung-Kiu Lau, Lily Safie, Petr Stepan, and Cuong Tran. A component model that is both control-driven and data-driven. In *14th International ACM Sigsoft Symposium on Component Based Software Engineering*, pages 41–50, New York, NY, USA, 2011. ACM.

[53] Petr Štěpán. Design pattern solutions as explicit entities in component-based software development. In *16th international workshop on Component-oriented programming*, pages 9–16, New York, NY, USA, 2011. ACM.

[54] Damian Arellanes and Kung-Kiu Lau. Exogenous Connectors for Hierarchical Service Composition. In *International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 125–132. IEEE, 2017.

[55] W. M. P. Van der Aalst, K. M. Van Hee, A. H. M. Ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn. Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011.

[56] Oana Otilia Prisecaru. Resource workflow nets: an approach to workflow modelling and analysis. *Enterprise Information Systems*, 2(2):101–120, 2008.

[57] M Ohba, Y Tanitsu, N Takimoto, and H Kadota. Functor: A higher-level co-operating program model. *Annual Review in Automatic Programming*, 11:21–28, 1981.

[58] Zebo Peng and K. Kuchcinski. Automated transformation of algorithms into register-transfer level implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(2):150–166, 1994.

[59] L. Thiele, K. Strehl, D. Ziegengein, R. Ernst, and J. Teich. FunState-an internal design representation for codesign. In *IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers*, pages 558–565, 1999.

[60] Paolo Baldan, Andrea Corradini, Hartmut Ehrig, and Reiko Heckel. Compositional semantics for open Petri nets based on deterministic processes. *Mathematical Structures in Computer Science*, 15(1):1–35, 2005.

[61] Karsten Wolf. Does My Service Have Partners? In *Transactions on Petri Nets and Other Models of Concurrency II: Special Issue on Concurrency in Process-Aware Information Systems*, Lecture Notes in Computer Science, pages 152–171. Springer, Berlin, Heidelberg, 2009.

[62] Paolo Baldan, Filippo Bonchi, Fabio Gadducci, and Giacoma Valentina Monreale. Modular encoding of synchronous and asynchronous interactions using open Petri nets. *Science of Computer Programming*, 109:96–124, 2015.

[63] Roberto Bruni, Hernan Melgratti, Ugo Montanari, and Pawel Sobocinski. Connector algebras for C/E and P/T nets' interactions. *Logical Methods in Computer Science*, Volume 9, Issue 3, 2013.

[64] Ekkart Kindler. A compositional partial order semantics for Petri net components. In *Application and Theory of Petri Nets 1997*, Lecture Notes in Computer Science, pages 235–252, Berlin, Heidelberg, 1997. Springer.

[65] Joachim Kock. Whole-grain Petri Nets and Processes. *Journal of the ACM*, 70(1):1–58, 2022.

[66] Evan Patterson, Owen Lynch, and James Fairbanks. Categorical Data Structures for Technical Computing. *Compositionality*, 4:5, 2022.

[67] Robin Piedeleu and Fabio Zanasi. *An Introduction to String Diagrams for Computer Scientists*. Elements in Applied Category Theory. Cambridge University Press, 1st edition, 2025.

[68] Ralf Hinze and Dan Marsden. *Introducing String Diagrams: The Art of Category Theory*. Cambridge University Press, Cambridge, 2023.

[69] Dusko Pavlovic. *Programs as Diagrams: From Categorical Computability to Computable Categories*. Theory and Applications of Computability. Springer, Cham, 2023.

[70] Roger Penrose. Applications of Negative Dimensional Tensors. *Combinatorial mathematics and its Applications*, pages 221–244, 1971.

[71] André Joyal and Ross Street. The geometry of tensor calculus, I. *Advances in Mathematics*, 88(1):55–112, 1991.

[72] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, Cambridge, 1st edition, 2017.

[73] John C. Baez and Brendan Fong. A Compositional Framework for Passive Linear Networks. *Theory and Applications of Categories*, 33(38):1158–1222, 2018.

[74] John C. Baez and Jason Erbele. Categories in Control. *Theory and Applications of Categories*, 30(24):836–881, 2015.

[75] B. Coecke, M. Sadrzadeh, and S. Clark. Mathematical Foundations for a Compositional Distributional Model of Meaning. *Linguistic Analysis*, 36:345–384, 2010.

[76] P. Selinger. A Survey of Graphical Languages for Monoidal Categories. In *New Structures for Physics*, pages 289–355. Springer, Berlin, Heidelberg, 2011.

[77] Bob Coecke and Ross Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New J. Phys.*, 13(4):1–85, 2011.

[78] Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. Full Abstraction for Signal Flow Graphs. In *42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, POPL '15, pages 515–526, Mumbai, India, 2015. ACM.

[79] Brendan Fong, Paweł Sobociński, and Paolo Rapisarda. A categorical approach to open and interconnected dynamical systems. In *31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 495–504, New York, NY, USA, 2016. ACM.

[80] Filippo Bonchi, Joshua Holland, Robin Piedeleu, Paweł Sobociński, and Fabio Zanasi. Diagrammatic algebra: from linear to concurrent systems. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[81] Brendan Fong and David Spivak. String Diagrams for Regular Logic (Extended Abstract). *Electronic Proceedings in Theoretical Computer Science*, 323:196–229, 2020.

[82] Robin Piedeleu and Fabio Zanasi. A String Diagrammatic Axiomatisation of Finite-State Automata. In *24th International Conference on Foundations of Software Science and Computation Structures*, pages 469–489. Springer, 2021.

[83] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory II: Rewriting with symmetric monoidal structure. *Mathematical Structures in Computer Science*, 32(4):511–541, 2022.

[84] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński, and Fabio Zanasi. String diagram rewrite theory III: Confluence with and without Frobenius. *Mathematical Structures in Computer Science*, 32(7):829–869, 2022.

[85] Dan R. Ghica and George Kaye. Rewriting Modulo Traced Comonoid Structure. In *8th International Conference on Formal Structures for Computation and Deduction (FSCD)*, Lecture Notes in Computer Science, pages 14:1–14:21. Springer, 2023.

[86] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

[87] Alejandro Villoria, Henning Basold, and Alfons Laarman. Enriching Diagrams with Algebraic Operations. In *28th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2024)*. Springer, 2025.

[88] Robin Piedeleu, Mateo Torres-Ruiz, Alexandra Silva, and Fabio Zanasi. A Complete Axiomatisation of Equivalence for Discrete Probabilistic Programming. In *34th European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, pages 202–229. Springer, 2025.

[89] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String Diagram Rewrite Theory I: Rewriting with Frobenius Structure. *Journal of the ACM*, 69(2):14:1–14:58, 2022.

[90] Matthew Earnshaw and Paweł Sobociński. String diagrammatic trace theory. In *48th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 272 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 43:1–43:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[91] Filippo Bonchi, Alessandro Di Giorgio, and Alessio Santamaria. Deconstructing the Calculus of Relations with Tape Diagrams. *Proceedings of the ACM on Programming Languages*, 7(POPL):64:1864–64:1894, 2023.

[92] Filippo Bonchi, Alessandro Di Giorgio, and Elena Di Lavore. A Diagrammatic Algebra for Program Logics. In *28th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 308–330, 2025.

[93] Filippo Bonchi, Cipriano Junior Cioffo, Alessandro Di Giorgio, and Elena Di Lavore. Tape Diagrams for Monoidal Monads. In *11th Conference on Algebra and Coalgebra in Computer Science (CALCO)*, volume 342 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:24, Dagstuhl, Germany, 2025.

[94] Cole Comfort, Antonin Delpeuch, and Jules Hedges. Sheet diagrams for bimonoidal categories, 2020. arXiv preprint arXiv:2010.13361.

[95] A. C. Ehresmann and J. P. Vanbremeersch. *Memory Evolutive Systems; Hierarchy, Emergence, Cognition*, volume 4. Elsevier, 1st edition, 2007.

[96] Gheorghe Păun. Computing with Membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[97] Roberto Bruni, Howard Foster, Alberto Lluch Lafuente, Ugo Montanari, and Emilio Tuosto. A Formal Support to Business and Architectural Design for Service-Oriented Systems. In *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, pages 133–152. Springer, Berlin, Heidelberg, 2011.

[98] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[99] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 1st edition, 2009.

[100] B. Fong. *The algebra of open and interconnected systems*. PhD Thesis, University of Oxford, 2016.