

Typed Connector Families and Their Semantics

José Proença^{a,*}, Dave Clarke^b

^a*INESC TEC & University of Minho, Dep. Informática, 4710-057 Braga, Portugal*

^b*Uppsala University, Box 337, SE-751 05 Uppsala, Sweden*

Abstract

Typed models of connector/component composition specify interfaces describing ports of components and connectors. Typing ensures that these ports are plugged together appropriately, so that data can flow out of each output port and into an input port. These interfaces typically consider the direction of data flow and the type of values flowing. Components, connectors, and systems are often parameterised in such a way that the parameters affect the interfaces. Typing such *connector families* is challenging. This paper takes a first step towards addressing this problem by presenting a calculus of connector families with integer and boolean parameters. The calculus is based on monoidal categories, with a dependent type system that describes the parameterised interfaces of these connectors. We use families of Reo connectors as running examples, and show how this calculus can be applied to Petri Nets and to BIP systems. The paper focusses on the structure of connectors—*well-connectedness*—and less on their behaviour, making it easily applicable to a wide range of coordination and component-based models. A type-checking algorithm based on constraints is used to analyse connector families, supported by a proof-of-concept implementation.

1. Introduction

Software product lines provide the flexibility of concisely specifying a family of software products, by identifying common features of functionality among these products and automatising the creation of products from a selection of relevant features. Interesting challenges in this domain include how to specify families and combinations of features, how to automatise the creation process, how to identify features from a collection of products, and how to reason about (e.g., verify) whole families of products.

This paper investigates such variability in coordination languages, i.e., it studies *connector families* that exogenously describe how (families of) components are connected. The key problem is that different connectors from a single family can have different interfaces, i.e., different ways of connecting to other

*Corresponding author

connectors. Hence, specifying and composing such families of connectors while guaranteeing that interfaces still match becomes non-trivial.

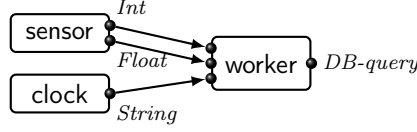


Figure 1: Composition of components: a sensor produces a pair of values, a clock produces a time stamp, and the worker produces a database query to store the sensor values.

Consider, for example, the simple component composition on Figure 1. In our calculus we can write this composition as $\text{sensor} \otimes \text{clock}; \text{worker}$, where ‘ \otimes ’ has higher precedence and denotes parallel composition, and ‘ $;$ ’ denotes sequential composition. In this scenario type-checking means guaranteeing that the interfaces match when plugging these 3 components together.

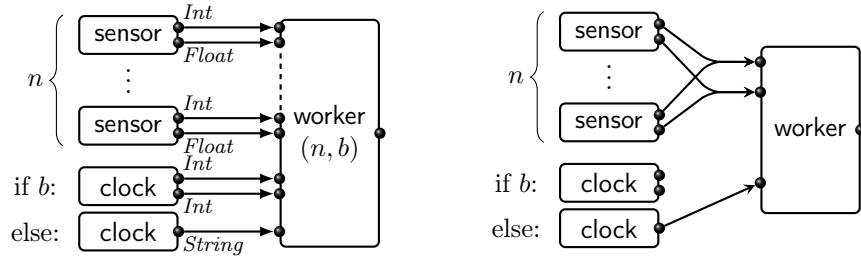


Figure 2: Composition of families of components: combinations of sensors and clocks are composed with a parameterised worker (left) and with a simpler worker after some extra coordination (right).

The case becomes more complex when components can be configured to have varying interfaces. For example, if the clock can chose to produce a pair of integers or a string based on a configuration parameter b , or if the sensor components is replaced by a number n of sensors running in parallel. This scenario is depicted in Figure 2, where reasoning about composition of such families is more complex than that of instantiated components. One can, for example, either (1) define a parameterised worker (left side of the figure) and require the interfaces to match for at least some instances of the families, or (2) replace the connections by a (parameterised) connector that combines the values of the sensors into an interface that matches the one from the worker (right side of the figure).

The first option (left of Figure 2) can be written in our calculus as $\lambda n : \mathbb{N}, b : \mathbb{B} \cdot (\text{sensor}^n \otimes \text{clock}(b); \text{worker}(n, b))$, where parameters are written as lambda abstractions, and exponentials replicate a given component or connector. An alternative and more modular expression for this scenario, also expressible in our calculus, could be $(\lambda n : \mathbb{N} \cdot \text{sensor}^n) \otimes (\lambda b : \mathbb{B} \cdot \text{clock}(b)); (\lambda n' : \mathbb{N}, b' : \mathbb{B} \cdot \text{worker}(n', b'))$, where each composition block is itself parameterised. This modular version is

implicitly imposing the parameters n and b to match the parameters n' and b' of the worker, hence the resulting composition is restricted by constraints over the parameters that make the composition be well-typed.

The second option (right of [Figure 2](#)) can be written in our calculus as $\lambda n : \mathbb{N}, b : \mathbb{B} \cdot (\text{sensor}^n \otimes \text{clock}(b); (\text{unzip}(n); (\nabla_n \otimes \nabla_n))) \otimes \text{id}_{\text{String}}; \text{worker}$, where $\text{unzip}(n)$ is a connector that swaps the orders of $2*n$ connections (that will be formally specified later in this paper) and ∇_n is a connector that merges n dataflows into one. The type-system of our calculus yields that this connector is well-typed exactly when b is false and for any possible n , i.e., the clock component must be configured with $b = \text{false}$ in order to have a *String* outgoing interface, so it can be composed with $\text{id}_{\text{String}}$. This constraint is explicit in our types; more specifically, the type of this connector is $\forall x : \mathbb{N}, b : \mathbb{B} \cdot (0 \rightarrow \text{DB-query})|_{b=\text{false}}$, indicating that it has no inputs, it has a single output with type *DB-query*, and it is restricted by the constraint $b = \text{false}$.

Summarising, the main contributions of this paper are:

- a calculus for families of connectors with constraints;
- a type system to describe well-defined compositions of such families;
- a semantics for families of connectors given by the Tile Model [\[1\]](#); and
- a constraint-based type-checking algorithm for this type system.

This paper extends a previous publication at FACS 2015 [\[2\]](#) mainly by (1) formalising the semantics of connector families using the Tile Model, which was previously used briefly for the basic connector calculus, and by (2) instantiating the typed connector calculus with Petri Nets and with BIP systems.

Connectors are defined incrementally. We start by defining a basic connector calculus for composing connectors inspired by Bruni et al.’s connector algebra [\[3, 4\]](#) ([Section 2](#)). This calculus is then extended with parameters and expressions, over both integers and booleans ([Section 3](#)), being now able to specify connectors (and interfaces) that depend on input parameters. Both the basic and the extended calculus are accompanied by a type system; the latter is an extension of the former, allowing integer and boolean parameters (and effectively becoming a dependent type system). [Section 4](#) introduces *connector families*, by explicitly incorporating constraints over the parameters, and by lifting the composition of connectors to the composition of constrained and parameterised connectors. [Section 5](#) describes an algorithm to type-check connector families with untyped ports, i.e., when the type flowing over each port is not relevant, and presents our prototype implementation. Typed connector calculus is used to model families of Petri Nets and BIP connectors in [Section 6](#). This paper wraps up with related work ([Section 7](#)), conclusions and future work ([Section 8](#)).

2. Basic Connector Calculus

This section describes an algebraic approach to specify connectors (or components) with a fixed interface, that is, with a fixed sequence of input and

output ports that are used to send and receive data. The main goal of this algebraic approach is to describe the structure of connectors and not so much their behaviour. We illustrate the usage of this algebra by using Reo connectors [5], which have well-defined semantics, although our approach can be applied to any connector-like model that connects entities with input and output interfaces.

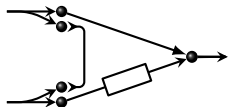
We start by presenting an overview of how to specify connectors using our calculus. We then describe the syntax of the basic connector calculus and a type system to verify if connectors are well-connected, followed by a brief discussion on how to describe the semantics of connectors orthogonally to this calculus.

2.1. Overview

Our *basic connector calculus* is based on monoidal categories—more specifically on traced symmetric monoidal categories [6]—where connectors are morphisms, “;” is the composition of morphisms with identity id , and “ \otimes ” is the tensor product. The operator “ \otimes ” composes connectors in parallel, while the operator “;” connects the ports of the given connectors. Objects of this category are *interfaces*, which correspond to ports in our connectors and include the unit of the tensor product represented by 0 . The commutativity of the tensor product is captured by a family of symmetries that swap the order of ports in parallel. Loops can be represented via *traces*, which plug part of the right interface to the left interface of the same connector.

The connector in Table 1 helps understanding the intuition behind our algebra of connectors. Our algebra is inspired by the graphical notation used for monoidal categories (see, e.g., Selinger’s survey [6]), and by Bruni et al.’s connector algebra [3, 4]. The Reo connector on the left is composed out of smaller subconnectors, connected with each other via shared ports (\bullet). The second column describes a possible representation of the same connector, writing the names of each subconnector parameterised by its ports. For example, the connector ‘ \searrow ’ is written as $\text{sdrain}(a, b)$ to mean that it has two ports named a and b . Composing connectors is achieved via the \bowtie operator, which connects ports with the same names – this is the most common way to compose Reo connectors in the literature. In this paper we will use instead the algebraic representation on the right of Table 1, where port names are not necessary. The connector $\Delta \otimes \Delta$, for example, puts two duplicator channels in parallel, yielding a new connector with 2 input ports and 4 output ports. This can be composed via “;” with $\text{id} \otimes \text{sdrain} \otimes \text{fifo}$ because this connector has 4 input ports: both the id and the fifo channels have one input port and the sdrain has 2 input ports.

Table 1: Specification of the alternator connector with port names and algebraically.

Graphical	With port names	Algebraic term
	$\Delta(a, a_1, a_2) \bowtie \Delta(b, b_1, b_2) \bowtie$ $\text{sdrain}(a_2, b_1) \bowtie$ $\text{id}(a_1, c_1) \bowtie \text{fifo}(b_2, c_2) \bowtie$ $\nabla(c_1, c_2, c)$	$\Delta \otimes \Delta;$ $\text{id} \otimes \text{sdrain} \otimes \text{fifo};$ ∇

2.2. Syntax

The syntax of connectors and interfaces of our basic connector calculus is presented in [Figure 3](#). Each connector has a signature $I \rightarrow J$ consisting of an input interface I and an output interface J . For example, the identity connector id_I has the same input and output interface I , written $\text{id}_I : I \rightarrow I$. Ports of an interface are identified simply with a capital letter, such as A , which capture the type of messages that can be sent via that port. In our examples we assume that A can only be the type 1, which represents any port type. This more specific model is known in the literature as PROP [\[7\]](#), which is a symmetric monoidal category generated by a single object, is also exploited in our algorithm for constraint solving later in [Section 5](#).¹

$c ::= c_1 ; c_2$	sequential composition	$p \in \mathcal{P} ::= \Delta_I$	duplicator with output I
$c_1 \otimes c_2$	parallel composition	∇_I	merger with input I
id_I	identity connectors	sdrain	synchronous drain
$\gamma_{I,J}$	symmetries	fifo	buffer
$\text{Tr}_I(c)$	traces	...	user-defined connectors
$p \in \mathcal{P}$	primitive connectors	$I, J ::= I \otimes J$	tensor
		0	interface with no ports
		A	port type

Figure 3: Connectors (left), primitive connectors (top-right), interfaces (bottom-right).

The intuition of these connectors becomes clearer with the visual representations exemplified in [Figure 4](#). All connectors are depicted with their input interface on the left side and the output interface on the right side. Each *identity* connector id_I has the same input and output interface I ; each *symmetry* $\gamma_{I,J}$ swaps the top interface I with the bottom interface J , hence it has input interface $I \otimes J$ and output interface $J \otimes I$; and each *trace* $\text{Tr}_I(c)$ creates a loop from the bottom output interface I of c with the bottom input interface I of c , hence if c has input interface $I' \otimes I$ and output interface $J' \otimes I$ then the trace has input and output interfaces I' and J' , respectively.

Parallelism is represented by tensor products, plugging of connectors by morphism composition, swapping order of parameters by symmetries, and loops by traces. Connectors and types obey a set of *Equations for Connectors* that allow their algebraic manipulation and capture the intuition behind the above mentioned representations. [Figure 5](#) presents *some* of these equations, which reflect properties of traced monoidal categories. For example, the fact that two symmetries in sequence with swapped interfaces are equivalent to the identity connector, or how the trace of the symmetry $\gamma_{1,1}$ is also equivalent to the identity. We chose not to present a full axiomatisation because this paper focuses on the

¹This connection with PROP was the motivation to denote as 0 the identity of the tensor and 1 the singleton object, instead of using the more commonly used notation of 1 for the identity of the tensor.

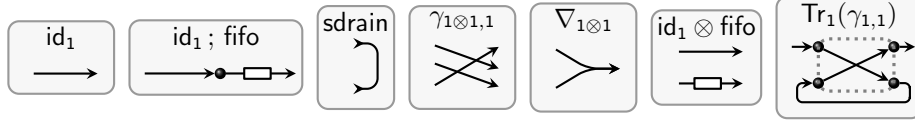


Figure 4: Visual representation of simple connectors.

type-checking of families of connectors, and leaves as future work the analysis of such axiomatisation.

$$\begin{array}{ll}
\text{if } c : I \rightarrow J \text{ then} & c_1 ; \text{Tr}_I(c_2) = \text{Tr}_I(c_1 \otimes \text{id}_I ; c_2) \quad (7) \\
\text{id}_I ; c = c ; \text{id}_J & \text{Tr}_I(c_1) ; c_2 = \text{Tr}_I(c_1 ; c_2 \otimes \text{id}_I) \quad (8) \\
\gamma_{I,J} ; \gamma_{J,I} = \text{id}_{I \otimes J} & (2) \\
(c_1 \otimes c_2) \otimes c_3 = c_1 \otimes (c_2 \otimes c_3) & (3) \quad 0 \otimes I = I = I \otimes 0 \quad (9) \\
\text{Tr}_I(\gamma_{I,I}) = \text{id}_I & (4) \quad (I_1 \otimes I_2) \otimes I_3 = I_1 \otimes (I_2 \otimes I_3) \quad (10) \\
\text{Tr}_0(c) = c & (5) \\
\text{Tr}_I(\text{Tr}_J(c)) = \text{Tr}_{I \otimes J}(c) & (6) \quad \text{if } I = I', J = J' \text{ then} \\
& I \rightarrow J = I' \rightarrow J' \quad (11)
\end{array}$$

Figure 5: Equations for connectors, interfaces, and types – based on properties of traced symmetrical monoidal categories.

2.3. Type rules

Every connector c has an input interface I and an output interface J , written $c : I \rightarrow J$. We call these two interfaces the *type* of the connector. Every primitive has a fixed type, for example, $\text{fifo} : \mathbf{1} \rightarrow \mathbf{1}$ and $\nabla_{\mathbf{1} \otimes \mathbf{1}} : \mathbf{1} \otimes \mathbf{1} \rightarrow \mathbf{1}$. The typing rules for connectors (Figure 6) reflect the fact that two connectors can only be composed sequentially if the output interface of the first connector matches the input interface of the second one. A connector is well-connected if and only if it is well-typed.

$$\begin{array}{c}
\begin{array}{ccc}
\text{(sequence)} & \text{(parallel)} & \text{(trace)} \\
\frac{\vdash c_1 : I_1 \rightarrow J \quad \vdash c_2 : J \rightarrow J_2}{\vdash c_1 ; c_2 : I_1 \rightarrow J_2} & \frac{\vdash c_1 : I_1 \rightarrow J_1 \quad \vdash c_2 : I_2 \rightarrow J_2}{\vdash c_1 \otimes c_2 : I_1 \otimes I_2 \rightarrow J_1 \otimes J_2} & \frac{\vdash c : I_1 \otimes J \rightarrow I_2 \otimes J}{\vdash \text{Tr}_J(c) : I_1 \rightarrow I_2} \\
\text{(sym)} & \text{(id)} & \text{(prim)} \\
\frac{}{\vdash \gamma_{I,J} : I \otimes J \rightarrow J \otimes I} & \frac{}{\vdash \text{id}_I : I \rightarrow I} & \frac{p : I \rightarrow J \in \mathcal{P}}{\vdash p : I \rightarrow J}
\end{array}
\end{array}$$

Figure 6: Type rules for basic connectors.

For example, using these type rules it is possible to infer the type of the connector $\text{Tr}_{\mathbf{1} \otimes \mathbf{1}}(\gamma_{\mathbf{1} \otimes \mathbf{1}, \mathbf{1}} ; (\text{fifo} \otimes \text{fifo} \otimes \text{fifo}))$ to be $\mathbf{1} \rightarrow \mathbf{1}$, but no type could be inferred

after removing one occurrence of `fifo`. This connector is chaining in sequence 3 parallel `fifo` connectors (i.e., it is equivalent to the connector `fifo; fifo; fifo`).

The type rules from Figure 6 rely on the syntactic comparison of interfaces, e.g., rule **(sequence)** allows c_1 and c_2 to be composed only if the output interface J of c_1 is syntactically equivalent to the input interface of c_2 . To support more complex notions of interfaces we use the constraint-based type rules from Figure 7, which explicitly compare interfaces that must be *provably equivalent* (based on the equations for connectors and interfaces, and other properties of traced symmetrical monoidal categories) instead of syntactically comparing them. Rules **(sym)**, **(id)**, and **(prim)** remain the same, only with the context. The typing judgements now include a context $\Gamma \mid \phi$ consisting both of a set of typed variables Γ (that will only be used in the next section) and a set of constraints ϕ that must hold for the connector to be well-typed. The context must be always well-formed, i.e., Γ cannot have repeated variables and ϕ must have at least one solution.

Observe that the **(trace)** rule introduces variables X_I and X_J representing interfaces, which are not in the syntax of interfaces. These variables are used to decompose an interface into the (tensor) composition with another constant interface.

$$\begin{array}{c}
 \text{(sequence)} \\
 \frac{\Gamma \mid \phi \vdash c_1 : I_1 \rightarrow J_1 \quad \Gamma \mid \phi \vdash c_2 : I_2 \rightarrow J_2}{\Gamma \mid \phi, J_1 = I_2 \vdash c_1 ; c_2 : I_1 \rightarrow J_2} \\
 \text{(trace)} \\
 \frac{\Gamma \mid \phi \vdash c : J_1 \rightarrow J_2}{\Gamma \mid \phi, J_1 = X_I \otimes I, J_2 = X_J \otimes I \vdash \text{Tr}_I(c) : X_I \rightarrow X_J}
 \end{array}$$

Figure 7: Constraint-based type rules.

Type preservation of equations for connectors The equations for connectors in Figure 5 preserve types. We prove this property for 4 equations, and the other proofs follow a similar approach.

Identity. We show that, when $c : I \rightarrow J$, the types of $\text{id}_J ; c$, c and $c ; \text{id}_J$ are the same up to the interface equalities. The type rules yield the type judgements $[I=I \vdash \text{id}_I ; c : I \rightarrow J]$, $[\vdash c : I \rightarrow J]$ and $[J=J \vdash c ; \text{id}_J : I \rightarrow J]$, which are equivalent since $I = I$ and $J = J$ always hold.

Associativity. We show that, for any $c_i : I_i \rightarrow J_i$ with $i \in \{1, 2, 3\}$, the type of $(c_1 \otimes c_2) \otimes c_3$ is the same as the type of $c_1 \otimes (c_2 \otimes c_3)$ up to the interface equalities. Indeed, type rules yield the types $(I_1 \otimes I_2) \otimes I_3 \rightarrow (J_1 \otimes J_2) \otimes J_3$ and $I_1 \otimes (I_2 \otimes I_3) \rightarrow J_1 \otimes (J_2 \otimes J_3)$, respectively, which are indeed equal.

Trace yanking. We show that, for any interface I , $\text{Tr}_I(\gamma_{I,I})$ and id_I have the same type. The type rules yield the type judgements $[I \otimes I = X_I \otimes I, I \otimes I = X_J \otimes I \vdash \text{Tr}_I(\gamma_{I,I}) : X_I \rightarrow X_J]$ and $[\vdash \text{id}_I : I \rightarrow I]$, respectively. These types

are indeed the same because $I \otimes I = X \otimes I$ has a unique solution (up to equality defined for interfaces) where $X = I$.

Trace naturality. We show that, for any $c_1 : I_1 \rightarrow J_1$ and $c_2 : J_1 \otimes I \rightarrow J_2 \otimes I$, the types of c_1 ; $\text{Tr}_I(c_2)$ and $\text{Tr}_I(c_1 \otimes \text{id}_I$; $c_2)$ are the same. The type rules yield, respectively, the type judgements $[J_1 = J_1, J_1 \otimes I = X_I \otimes I, J_2 \otimes I = X_J \otimes I \vdash c_1; \text{Tr}_I(c_2) : I_1 \rightarrow X_J]$ and $[I_1 \otimes I = X'_I \otimes I, J_2 \otimes I = X'_J \otimes I, J_1 \otimes I = J_1 \otimes I \vdash c_1; \text{Tr}_I(c_2) : X'_I \rightarrow X'_J]$. These are indeed the same because the only solutions for the constraints in the type judgements are $X_I = J_1$, $X_J = J_2$, $X'_I = I_1$, and $X'_J = J_2$, making both types the same as $I_1 \rightarrow J_2$ (after applying the solution).

2.4. Connector behaviour

Semantics for the behaviour of connectors can be given in various ways. For this paper we use the Tile Model [1], as it aligns closely with the algebraic presentation of connectors. We also use the Reo coordination language—more specifically its context independent semantics [4]—as the behaviour of our primitive connectors, whose visual representation has been being used.

We use the same ideas from the Tile Model proposed for Reo [4], using a variation of the category used to describe connectors. Each connector in the Tile Model consists of a set of tiles, one for each possible behaviour, as exemplified in Figure 8. Each of these tiles contains 4 objects, which belong to two monoidal categories with the same objects, and 4 morphisms between pairs of objects. Visually, a tile is depicted as a square with an object in each corner and with morphisms on the sides of this square. These morphisms go from left to right and from top to bottom: horizontal morphisms are from one category \mathcal{H} , describing the construction of a connector, and the vertical morphisms are from another category \mathcal{V} , describing the evolution in time of the connector. More specifically, horizontal morphisms are connectors as specified in Figure 3, and objects are interfaces. Vertical morphisms are either `flow`, `noFlow`, or a tensor product of these (with identity 0), representing a step where data flows over the ports where the `flow` morphism is applied, and data does not flow over the ports where `noFlow` is applied. Other vertical monoidal categories can be used, inducing a different semantics for connectors; for example one could include a different morphism `flowd` for each data value d being sent, allowing the semantics to depend on data being sent, or could include two different notions of `noFlow` to model context dependency as in connector colouring’s semantics [8].

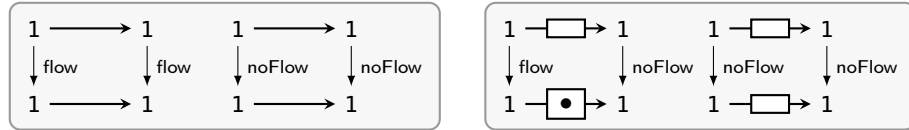


Figure 8: Tiles for the behaviour of the `id1` (left) and the empty `fifo` (right) connectors.

Primitive tiles The examples in Figure 8 describe the tiles for the `id1` and the `fifofull` primitive. For simplicity, we write $c_1 \xrightarrow[v_2]{v_1} c_2$ (following [1, 4]) to

denote a tile with horizontal morphisms (i.e., connectors) c_1 and c_2 , and with vertical morphisms (i.e., evolutions) v_1 and v_2 . In the `fifo` example, its 2 tiles are `fifo` $\xrightarrow[\text{noFlow}]{\text{flow}}$ `fifofull` and `fifo` $\xrightarrow[\text{noFlow}]{\text{noFlow}}$ `fifo`. The behaviour of `fifofull`, in turn, is given by $\left\{ \text{fifofull} \xrightarrow[\text{flow}]{\text{noFlow}} \text{fifo}, \text{fifofull} \xrightarrow[\text{noFlow}]{\text{noFlow}} \text{fifofull} \right\}$, meaning it can either have dataflow in its sink port (and becoming a `fifo`) or no dataflow at all. The behaviour (set of possible tiles) of other primitives used in this paper can be found in [Figure 9](#).

$$\begin{aligned} \gamma_{1,1} &= \left\{ \gamma_{1,1} \xrightarrow[\text{noFlow}]{\text{flow}} \gamma_{1,1}, \gamma_{1,1} \xrightarrow[\text{flow}]{\text{noFlow}} \gamma_{1,1}, \gamma_{1,1} \xrightarrow[\text{flow}]{\text{flow}} \gamma_{1,1}, \gamma_{1,1} \xrightarrow[\text{noFlow}]{\text{noFlow}} \gamma_{1,1} \right\} \\ \Delta_{1 \otimes 1} &= \left\{ \Delta_{1 \otimes 1} \xrightarrow[\text{flow} \otimes \text{flow}]{\text{flow}} \Delta_{1 \otimes 1}, \Delta_{1 \otimes 1} \xrightarrow[\text{noFlow} \otimes \text{noFlow}]{\text{noFlow}} \Delta_{1 \otimes 1} \right\} \\ \nabla_{1 \otimes 1} &= \left\{ \nabla_{1 \otimes 1} \xrightarrow[\text{flow}]{\text{flow} \otimes \text{noFlow}} \nabla_{1 \otimes 1}, \nabla_{1 \otimes 1} \xrightarrow[\text{flow}]{\text{noFlow} \otimes \text{flow}} \nabla_{1 \otimes 1}, \nabla_{1 \otimes 1} \xrightarrow[\text{noFlow}]{\text{noFlow} \otimes \text{noFlow}} \nabla_{1 \otimes 1} \right\} \\ \text{Tr}_I(c_1) &= \left\{ \text{Tr}_I(c_1) \xrightarrow[v_2]{v_1} \text{Tr}_I(c_2) \mid \exists (v : I \rightarrow I) \in \mathcal{V} \cdot c_1 \xrightarrow[v_2 \otimes v]{v_1 \otimes v} c_2 \text{ exists} \right\} \\ \text{id}_1 &= \left\{ \text{id}_1 \xrightarrow[\text{flow}]{\text{flow}} \text{id}_1, \text{id}_1 \xrightarrow[\text{noFlow}]{\text{noFlow}} \text{id}_1 \right\} \\ \text{sdrain} &= \left\{ \text{sdrain} \xrightarrow[\text{flow}]{\text{flow}} \text{sdrain}, \text{sdrain} \xrightarrow[\text{noFlow}]{\text{noFlow}} \text{sdrain} \right\} \\ \text{lossy} &= \left\{ \text{lossy} \xrightarrow[\text{flow}]{\text{flow}} \text{lossy}, \text{lossy} \xrightarrow[\text{noFlow}]{\text{flow}} \text{lossy}, \text{lossy} \xrightarrow[\text{noFlow}]{\text{noFlow}} \text{lossy} \right\} \end{aligned}$$

Figure 9: Behaviour of primitive connectors using tiles.

Tile composition Tiles can be composed horizontally, vertically, and in parallel [[3](#), [4](#), [1](#)]. Two tiles can be composed horizontally if their right and left morphisms match, respectively; can be composed vertically if their down and up morphisms match, respectively; and can always be composed in parallel by combining with \otimes all their morphisms and objects. Formally, let \circ be the composition of morphisms in the vertical category \mathcal{V} , then:

$$c_1 \xrightarrow[v]{v_1} c_2; c'_1 \xrightarrow[v_2]{v} c'_2 = (c_1; c'_1) \xrightarrow[v_2]{v_1} (c_2; c'_2) \quad (12)$$

$$c_1 \xrightarrow[v_2]{v_1} c \circ c \xrightarrow[v'_2]{v'_1} c_2 = c_1 \xrightarrow[v'_2 \circ v_2]{v'_1 \circ v_1} c_2 \quad (13)$$

$$c_1 \xrightarrow[v_2]{v_1} c_2 \otimes c'_1 \xrightarrow[v'_2]{v'_1} c'_2 = c_1 \otimes c_2 \xrightarrow[v_2 \otimes v'_2]{v_1 \otimes v'_1} c'_1 \otimes c'_2 \quad (14)$$

Examples Consider the simple connector obtained from composing a `fifo` with a $\Delta_{1 \otimes 1}$ (that we will simply denote as Δ). The semantics of `fifo`; Δ is given by combining the possible tiles of `fifo` and Δ . Figure 10 depicts the possible horizontal combinations of tiles.

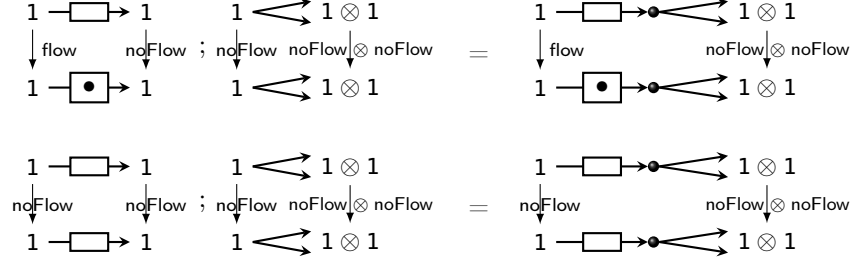


Figure 10: Combining tiles to give the semantics of `fifo`; Δ . No other pairs of tiles can be combined with ‘;’.

The vertical composition of tiles is illustrated in Figure 11, exemplifying a possible composition of (4) tiles that describe a possible evolution of the connector `fifo`; `fifo`, where data initially flows into an empty FIFO buffer, and later it flows from that buffer to a second FIFO buffer.

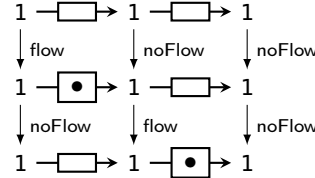


Figure 11: Horizontal and vertical composition of tiles yielding a possible execution of `fifo`; `fifo`.

Type preservation for Reo We showed in Section 2.3 that the non-exhaustive list of equations of connectors preserves types. We now repeat this process for the tile semantics of Reo. More specifically we show that, given the Reo primitives presented above (Figure 10 and the `fifo` connector), a well-typed connector c can only evolve to well-typed connectors, i.e., if $c \xrightarrow[v_2]{v_1} c'$ and c has type T , then c' also has type T . We do this by induction on the structure of the tiles.

Stateless primitives. Stateless primitives are connectors c that have only tiles with shape $c \xrightarrow[v_2]{v_1} c$, i.e., that never change after an evolution step. These include most of the connectors: id_I , $\gamma_{I,J}$, Δ_I , ∇_I , sdrain , and lossy . In this case types are trivially preserved, as the connector is not modified.

Fifo connector. The types of the connectors `fifo` and `fifofull` are also trivially preserved, because they both have the fixed type $1 \rightarrow 1$ and only use the tiles `fifofull` $\xrightarrow[\text{flow}]{\text{noFlow}}$ `fifo` and `fifofull` $\xrightarrow[\text{noFlow}]{\text{noFlow}}$ `fifofull`.

Traces. We show that any tile $\text{Tr}_I(c) \xrightarrow[v_2]{v_1} \text{Tr}_I(c')$ preserves types, i.e., the types of $\text{Tr}_I(c)$ and $\text{Tr}_I(c')$ are the same. If $\text{Tr}_I(c)$ is well-typed, then by rule (trace) $\text{Tr}_I(c) : X_I \rightarrow X_J$ if $c : J_1 \rightarrow J_2$, $J_1 = X_I \otimes I$ and $J_2 = X_J \otimes I$. By the definition of the tile we conclude that $c \xrightarrow[v_2 \otimes v]{v_1 \otimes v} c'$ must exist, for some $v : I \rightarrow I$. By induction we now conclude that $c' : J_1 \rightarrow J_2$, and using again the type rule (trace) (and knowing that $J_1 = X_I \otimes I$ and $J_2 = X_J \otimes I$) we conclude that $\text{Tr}_I(c') : X_I \rightarrow X_J$.

Tile horizontal composition - ; . Following Eq. (12), $(c_1; c'_1)$ can evolve via tiles with shape $c_1; c'_1 \xrightarrow[v_2]{v_1} c_2; c'_2$ when the smaller tiles $c_1 \xrightarrow[v]{v_1} c_2$ and $c'_1 \xrightarrow[v_2]{v} c'_2$ exist, for some vertical morphism v . We show that, when $(c_1; c'_1) : I_1 \rightarrow J'_1$, then also $(c_2; c'_2) : I_1 \rightarrow J'_1$. The type rule (sequence) states that $c_1 : I_1 \rightarrow J_1$, $c'_1 : I'_1 \rightarrow J'_1$, and $J_1 = J'_1$. From the smaller tiles, we conclude by induction that $c_2 : I_1 \rightarrow J_1$ and $c'_2 : I'_1 \rightarrow J'_1$. Finally, using again the type rule (sequence) and knowing that $J_1 = J'_1$ we conclude that $(c_2; c'_2) : I_1 \rightarrow J'_1$.

Tile vertical composition - o . Following Eq. (13), every tile $c_1 \xrightarrow[v'_2 \circ v_2]{v'_1 \circ v_1} c_2$ can be decomposed into the tiles $c_1 \xrightarrow[v_2]{v_1} c$ and $c \xrightarrow[v'_2]{v'_1} c_2$, for some morphism c . By induction, we can directly conclude that, if $c_1 : T$, then also $c : T$ and $c_2 : T$, hence c_1 and c_2 have the same type.

Tile parallel composition - \otimes . Following Eq. (14), $(c_1 \otimes c_2)$ can evolve via tiles with shape $c_1 \otimes c'_1 \xrightarrow[v_2 \otimes v'_2]{v_1 \otimes v'_1} c_2 \otimes c'_2$ when the smaller tiles $c_1 \xrightarrow[v_2]{v_1} c_2$ and $c'_1 \xrightarrow[v'_2]{v'_1} c'_2$ exist. The type rule (parallel) states that $c_1 \otimes c'_1 : I_1 \otimes I'_1 \rightarrow J_1 \otimes J'_1$ if $c_1 : I_1 \rightarrow J_1$ and $c'_1 : I'_1 \rightarrow J'_1$. By induction, we conclude that also $c_2 : I_2 \rightarrow J_2$ and $c'_2 : I'_2 \rightarrow J'_2$. Finally, using again the type rule (parallel) we conclude that $c_2 \otimes c'_2 : I_1 \otimes I'_1 \rightarrow J_1 \otimes J'_1$ (the same type as $c_1 \otimes c'_1$).

3. Parameterised Connector Calculus

Connectors are now extended in two ways: (i) by adding integer and boolean expressions to control n -ary replication and conditional choice, and (ii) by adding free variables that can be instantiated with either natural numbers or booleans. These variables are also used in the connector types, previously written as $I \rightarrow J$, which are now given by the grammar:

$$T ::= I \rightarrow J \mid \forall x : P \cdot T$$

where x is a variable and $P \in \{\mathbb{N}, \mathbb{B}\}$ represents a primitive type that can be either the natural numbers (\mathbb{N}) or booleans (\mathbb{B}).

This section introduces the extended syntax and some of its properties, describes motivating examples, and extends the type rules for the connector types described above with boolean and integer parameters.

3.1. Syntax

The extended syntax of connectors and interfaces with integers and booleans is defined in [Figure 12](#). We write c^α (resp. I^α) instead of $c^{x \leftarrow \alpha}$ (resp. $I^{x \leftarrow \alpha}$) when x is not a free variable in c .

$c ::= \dots$	connectors	$I ::= \dots$	interfaces
	$c^{x \leftarrow \alpha}$		$I^{x \leftarrow \alpha}$
	n -ary parallel replication		n -ary parallel replication
	$c_1 \oplus^\phi c_2$		$I \oplus^\phi J$
	conditional choice		conditional choice
	$\lambda x : P \cdot c$		
	parameterised connector		
	$c(\phi)$		α, β
	bool-instantiation		integer expressions
	$c(\alpha)$		ϕ, ψ
	int-instantiation		boolean expressions

Figure 12: Extended syntax of connectors (left) and interfaces (right).

This paper does not formalise integer and boolean expressions with typed variables, since the details of these expressions are not relevant. The semantics of the n -ary parallel replication, the conditional choice, and the instantiation of parameters² is captured by the new equations in [Figure 13](#). These equations include a new notation— $c[v/x]$ and $I[v/x]$ —that stands for the substitution of all variables x in c and I that appear freely (i.e., not bounded by a λ or a \forall quantifier) by the expression v .³ This paper does not formalise free variables nor substitution, which follow the standard definitions.

$$\begin{aligned}
 c^{x \leftarrow \alpha} &= c[0/x] \otimes \dots & I^{x \leftarrow \alpha} &= I[0/x] \otimes \dots \\
 &\dots \otimes c[\alpha-1/x] & &\dots \otimes I[\alpha-1/x] & (15) & & (20) \\
 c_1 \oplus^{true} c_2 &= c_1 & I \oplus^{true} J &= I & (16) & & (21) \\
 c_1 \oplus^\phi c_2 &= c_2 \oplus^{-\phi} c_1 & I \oplus^\phi J &= J \oplus^{-\phi} I & (17) & & (22) \\
 c \oplus^\phi c &= c & I \oplus^\phi I &= I & (18) & & (23) \\
 (\lambda x : P \cdot c)(v) &= c[v/x] & I \oplus^\phi J &= I \oplus^{\phi'} J \text{ (if } \phi \leftrightarrow \phi') & (19) & & (24) \\
 & & \forall x : P \cdot T &= \forall x : P \cdot T' \text{ (if } T = T') & & & (25)
 \end{aligned}$$

Figure 13: Equations for connectors, interfaces, and types for the parameterised calculus.

Examples Although this extension allows an n -ary composition in parallel of connectors and not in sequence, n -ary sequences of connectors can also be expressed by using traces, as exemplified in the general sequence of fifo connectors in [Figure 14](#). We write expressions such as $n - 1$ instead of the interface 1^{n-1} for simplicity, when it is clear that these expressions represent interfaces. Observe that this example has been mentioned in the end of [Section 2.3](#), for

²Known as β -reduction in lambda calculus.

³Note that, when $\alpha \leq 0$, $c^{x \leftarrow \alpha} = \text{id}_0$ and $I^{x \leftarrow \alpha} = 0$.

the specific case of 3 fifos in sequence, already defined using traces and parallel replication.

The example in Figure 15 is more complex, and is based on the sequencer connector found in Reo-related literature [5]. This connector forces n (synchronous) streams of data to alternate between which one has dataflow. A sequencer with size n has three inputs and three outputs: it starts by allowing data from the first input to flow to the first output atomically, it then allows the second input to flow to the first output, and so on. After the n^{th} input and output it starts over by allowing the first input and output to flow data. It uses the *zip* and *unzip* connectors to combine γ connectors (symmetries) in order to regroup sequences of pairs into a pair of sequences and vice-versa. The top part of the figure defines the zip connector based on the function *seq* introduced in Figure 14, and instantiates this function with $n = 3$. This instantiation provides a better intuition about the zip connector, and its visual representation unfolds the trace for readability.

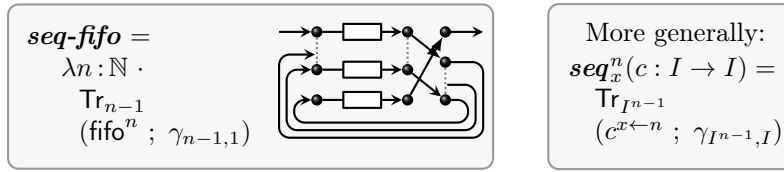


Figure 14: A sequence of n fifo connectors, and a function that generalises the sequence of connectors equal input and output interfaces.

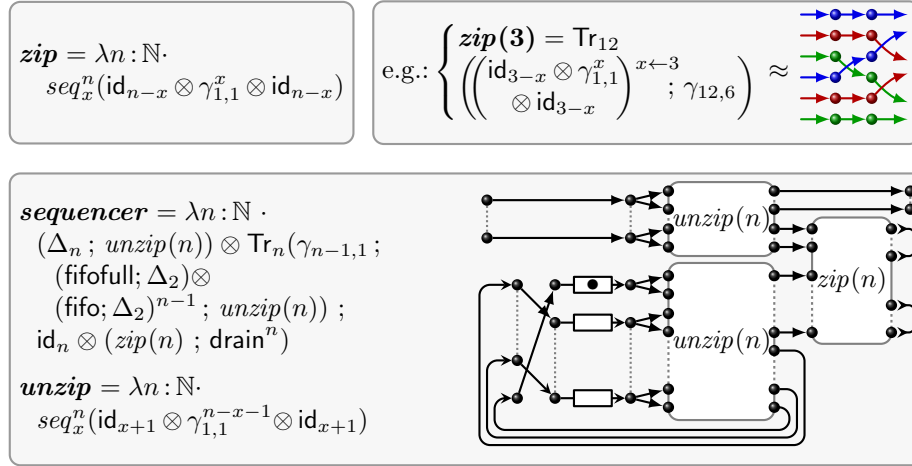


Figure 15: Zip connector and one of its instance (top), and an n -ary sequencer (bottom).

The details about the behaviour of the sequencer connector are out of the scope of this paper. However, observe that the visual representation is no longer precise enough, since the dotted lines only help to provide intuition but do not

specify completely the connector. The parameterised calculus, on the other hand, precisely describes how to build a n -ary sequencer for any $n \geq 0$.

3.2. Parameterised type rules

The extended type rules are presented in Figure 16, which now use the context Γ consisting of a set of variables and their associated primitive type (\mathbb{B} or \mathbb{N}).

$$\begin{array}{c}
\text{(parameterisation)} \\
\frac{\Gamma, x : P \mid \phi \vdash c : T}{\Gamma \mid \phi \vdash \lambda x : P . c : \forall x : P . T} \\
\\
\text{(choice)} \\
\frac{\Gamma \mid \phi \vdash \psi : \mathbb{B} \quad \Gamma \mid \phi \vdash c_1 : I_1 \rightarrow J_1 \quad \Gamma \mid \phi \vdash c_2 : I_2 \rightarrow J_2}{\Gamma \mid \phi \vdash c_1 \oplus^\psi c_2 : I_1 \oplus^\psi I_2 \rightarrow J_1 \oplus^\psi J_2} \\
\\
\text{(replication)} \qquad \text{(instantiation)} \\
\frac{\Gamma \mid \phi \vdash \alpha : \mathbb{N} \quad \Gamma, x : \mathbb{N} \mid \phi \vdash c : I \rightarrow J}{\Gamma \mid \phi \vdash c^{x \leftarrow \alpha} : I^{x \leftarrow \alpha} \rightarrow J^{x \leftarrow \alpha}} \quad \frac{\Gamma \mid \phi \vdash v : P \quad \Gamma \mid \phi \vdash c : \forall x : P . T}{\Gamma \mid \phi \vdash c(v) : T[v/x]}
\end{array}$$

Figure 16: Parameterised type rules— $x \notin \phi$ means x does not appear in ϕ . Previous type rules remain unchanged.

The actual verification of the type of the boolean and integer variables is done during the type-checking of boolean and integer expressions, which is well known and not defined in this paper. Hence the new type rules have some gray premises, corresponding to the type rules for booleans and integer expressions. The typing judgement $\Gamma \mid \phi \vdash c : T$ means that, given a set of typed variables Γ and a constraint ϕ with variables that do not need to be in Γ , the type of c must be T , the free variables in c and T must be in Γ , and ϕ must be satisfiable. The typing judgement $\Gamma \mid \phi \vdash e : P$ for integer and boolean expressions means that $\Gamma \vdash e : P$ (i.e., the variables in the boolean or integer expression e have the type specified in Γ) in a context where ϕ is satisfiable. The notation $T[e/x]$ denotes the substitution of free occurrences of x in T by the expression e , similarly to the substitution in connectors, also not formalised here. Observe that the constraint ψ in the (choice) rule does not influence the typing of c_1 and c_2 . Intuitively, if ψ and $\neg\psi$ was to be added to the context when typing c_1 and c_2 , respectively, then very likely one of these branches would have *false* in the context, meaning it could not be typed.

Example We illustrate the usage of these type rules by building the *derivation tree* for the *seq-fifo* connector (Figure 17), where we illustrate how to calculate the type of this connector by consecutively applying type rules. For simplicity, we write Tr_n and $\gamma_{n,m}$ to denote Tr_{1^n} and $\gamma_{1^n, 1^m}$. At every step of this derivation tree the context is well-formed (Γ has no repeated variables and ϕ is always satisfiable). From the existence of this derivation tree one can conclude that the *seq-fifo* connector is well-typed, and by further analysing the constraints in the context using the equations for interfaces and types it is possible to simplify the type to $\forall n : \mathbb{N} . \mathbf{1} \rightarrow \mathbf{1}$.

$$\begin{array}{c}
\emptyset \mid 1^n = 1^{n-1} \otimes 1, 1^{n-1} \otimes 1 = X_I \otimes 1^{n-1}, 1^n = X_J \otimes 1^{n-1} \\
\vdash \lambda n : \mathbb{N} \cdot \text{Tr}_{n-1}(\text{fifo}^n ; \gamma_{n-1,1}) : \forall n : \mathbb{N} \cdot X_I \rightarrow X_J \\
\left[\begin{array}{l} \text{parameterisation} \\ \text{trace} \\ \text{sequence} \\ \text{repl.} \end{array} \right. \left[\begin{array}{l} n : \mathbb{N} \mid 1^n = 1^{n-1} \otimes 1, 1^{n-1} \otimes 1 = X_I \otimes 1^{n-1}, 1^n = X_J \otimes 1^{n-1} \\ \vdash \text{Tr}_{n-1}(\text{fifo}^n ; \gamma_{n-1,1}) : X_I \rightarrow X_J \\ n : \mathbb{N} \mid 1^n = 1^{n-1} \otimes 1 \\ \vdash \text{fifo}^n ; \gamma_{n-1,1} : 1^n \rightarrow 1 \otimes 1^{n-1} \\ n : \mathbb{N} \mid \emptyset \\ \vdash \text{fifo}^n : 1^n \rightarrow 1^n \\ n : \mathbb{N} \mid \emptyset \vdash n : \mathbb{N} \\ n : \mathbb{N} \mid \emptyset \vdash \text{fifo} : 1 \rightarrow 1 \\ n : \mathbb{N} \mid \emptyset \vdash \gamma_{n-1,1} : 1^{n-1} \otimes 1 \rightarrow 1 \otimes 1^{n-1} \end{array} \right.
\end{array}$$

Figure 17: Derivation tree for the *seq-fifo* connector; contexts are represented in grey.

Type preservation of equations for connectors The equations for connectors, also including the extension in Figure 13, preserve the type rules for parameterised connectors. We prove this for only a couple of cases—the remainder can be proven in a similar way.

Replication. We show that, for any $c : I \rightarrow J$ with a free integer variable x and any integer expression α , the types of $c^{x \leftarrow \alpha}$ and $c[0/x] \otimes \dots \otimes c[\alpha - 1/x]$ are the same. Let the type rules yield the type judgement for c of $x : \mathbb{N} \mid \phi \vdash c : I \rightarrow J$. The type rules of the two expressions yield, respectively, the types $I^{x \leftarrow \alpha} \rightarrow J^{x \leftarrow \alpha}$ and $I[0/x] \otimes \dots \otimes I[\alpha - 1/x] \rightarrow J[0/x] \otimes \dots \otimes J[\alpha - 1/x]$. These types are the same based on Eq. (20) from Figure 13

Choice negation. We show that, for any c_1 and c_2 , $c_1 \oplus^\phi c_2$ and $c_2 \oplus^{-\phi} c_1$ have the same type. This can be quickly verified by applying the type rules and verifying that $I \oplus^\phi J = J \oplus^{-\phi} I$.

Instantiation. We show that, for any connector $c : T$ with a possible free variable $x : P$ and value $v : P$, the types of $(\lambda x : P \cdot c)(v)$ and $c[v/x]$ match. On one hand the type rules yield the typing judgement $\emptyset \mid \phi \vdash \lambda x : P \cdot c : \forall x : P \cdot T$, and applying the (instantiation) rule $\emptyset \mid \phi \vdash (\lambda x : P \cdot c)(v) : T[x/v]$. On the other hand the typing judgement for c is $x : P \mid \phi \vdash c : T$. Hence one must still prove that $\emptyset \mid \phi \vdash c[v/x] : T[v/x]$ can be inferred. This can be proved by induction on the structure of c : for every type rule one can show that, when some connector c' has type T' , then also $c'[v/x]$ will have type $T'[v/x]$. The key observation is that x can occur in explicit interfaces in c' (id_I , $\text{Tr}_I(\cdot)$, and $\gamma_{I,J}$), in integer expressions ($c^{x \leftarrow \alpha}$), and in boolean expressions ($c_1 \oplus^\phi c_2$). In any of these cases the expected type equality can be verified.

3.3. Behaviour of parameterised connectors

This section introduced so far parameters and (boolean and integer) expressions to the syntax of connectors and interfaces. We now show that these

parameterised connectors and interfaces still form a monoidal category, after the necessary adjustments. Furthermore, we present a new vertical monoidal category to describe the behaviour of parameterised Reo connectors.

The type system introduced above gives to each connector c a type with the shape $\forall x_1 : P_1 \cdot \forall x_n : P_n \cdots I \rightarrow J$. When looking at these connectors as morphisms, the objects can no longer be standard interfaces (as in [Figure 12](#)), but they need to include the scope of the variables, given the grammar below.

$$IP ::= I \mid \forall x : P \cdot IP$$

Similarly to types in [Figure 13](#), we extend the equations for interfaces with the equality $\forall x : P \cdot I = \forall x : P \cdot I'$ whenever $I = I'$. It can be easily verified that the category with objects IP and morphisms given by parameterised connectors is still a traced symmetric monoidal category, as with the basic connector calculus ([Section 2](#)). In this category, the composition of 2 parameterised connectors requires their parameters and interfaces to match, and every connector $c : \forall x : P \cdot I \rightarrow J$ will connect objects $\forall x : P \cdot I$ and $\forall x : P \cdot J$ with common parameters. For example, composing $\lambda n : \mathbb{N} \cdot c_1 ; \lambda n : \mathbb{N} \cdot c_2$ yields the connector $\lambda n : \mathbb{N} \cdot (c_1 ; c_2)$. Observe that this connector composition is very restrictive in practice, since it forces the matching of interfaces to use the same variables, and is not captured (yet) by the type rules. The following section ([Section 4](#)) will present a more relaxed approach to compose parameterised connectors whose interfaces are *similar enough*, i.e., when they can be restricted until their interfaces match.

The vertical monoidal category presented in [Section 2.4](#) is adapted to also include parameters. This new category has again the objects IP , and the morphisms are not only the ones shown before (`noFlow`, `flow`, and the tensor product of morphisms) but also new morphisms that instantiate variables. More specifically, for every number v_n , boolean v_b , and variables $n : \mathbb{N}$ and $b : \mathbb{B}$, the morphism $\text{inst}(n \mapsto v_n)$ is the unique arrow $(\forall n : \mathbb{N} \cdot I) \rightarrow I[v_n/n]$ and the morphism $\text{inst}(b \mapsto v_b)$ is the unique arrow $(\forall b : \mathbb{B} \cdot I) \rightarrow I[v_b/b]$. Intuitively, this notion of behaviour means that families of connectors can only be executed after being instantiated, and this instantiation is made via morphisms in the same category that describes the evolution of connectors.

For every parameterised connector c with type $\forall x : P \cdot (I \rightarrow J)$, we define one tile for each value v of P (a number or a boolean), as depicted on the left of [Figure 18](#). For example, the semantics of the parameterised connector fn with n parallel fifos is given by a set of tiles, one for each natural number i , that instantiate the n -ary connector to the more concrete connector $fn(i)$. When $i = 2$, the corresponding tile is the one on the right of [Figure 18](#).

Type refinement (not preservation) for parameterised connectors We showed in [Section 3.3](#) that the tile semantics for Reo preserves types, i.e., that for every tile $c \xrightarrow[v']{v} c'$ the type of c and c' are the same. We introduced above a new set of vertical morphisms that break type preservation: every time a connector is instantiated, its type is refined to a more particular one. Hence we need to use a weaker notion of type preservation. Our weaker notion says that, if c has type T and can evolve to c' , then the type T' of c' must be a *refinement*

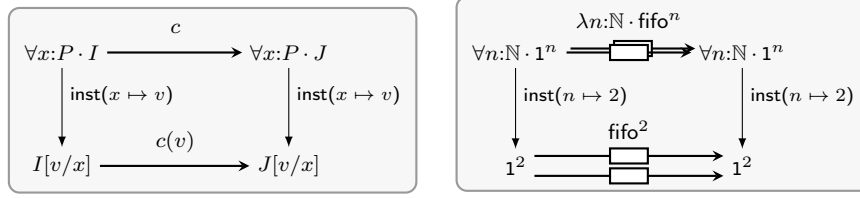


Figure 18: Tile for a parameterised connector c that instantiates x as v (left), and example of a tile for a family of n parallel fifo connectors by instantiating n with 2 (right).

of T , which we define below.

Here *refinement* has a similar (but stricter) meaning to its usage the context of refinement types [9], where a type T' is a refinement of T if it is subject to constraints that restricts its variables. Formally, we say a parameterised type $\overline{\forall x'}: P' \cdot T'$ refines the type $\overline{\forall x}: P \cdot T$ if there is a substitution σ —assigning variables from \overline{x} to values or variables—that makes $T'[\sigma]$ equal to T . I.e., a type can be refined by instantiating and renaming some of its variables.

To show that the evolution of families of Reo connectors, under the tile semantics, refines types, we only need to observe the following two points.

- Tiles that preserve types, such as the ones used in Section 3.3, also refine types (since any type T refines T).
- If $c \xrightarrow{\text{inst}(x \mapsto v)} c(v)$ then there exists x, P, I, J such that c is an arrow from $(\forall x: P \cdot I)$ to $(\forall x: P \cdot J)$, and c' is an arrow from $I[v/x]$ to $J[v/x]$. This means that c has type $\forall x: P, \overline{x''}: \overline{P''} \cdot I \rightarrow J$ and c' has type $(\forall \overline{x''}: \overline{P''} \cdot I \rightarrow J)[v/x]$, for some $\overline{x''}: \overline{P}$. Therefore, by the definition of refinement presented above, the type of $c(v)$ refines the type of c .

4. Connector families

This section introduces *connector families*: parameterised connectors that can (i) be *restricted* by given constraints ψ , written $c|_\psi$, and (ii) be *composed* with each other—sequentially, in parallel, via the choice or replication operators, or within traces. These restricted and composable connector families represent families in the same sense as software families in the context of software product lines (SPL) engineering [10]. The added constraints represent the family, which in the SPL community are often derived from feature models.

4.1. Restricted connectors and types

Connectors can now be written as $c|_\psi$, meaning that the connector c is restricted by the constraint ψ . For example, the connector with at most 5 fifo connectors in parallel can be written as $\lambda n: \mathbb{N} \cdot (\text{fifo}^n |_{n \leq 5})$. The type of this connector is written similarly as $\forall n: \mathbb{N} \cdot n \rightarrow n |_{n \leq 5}$. More formally, types now

$$c \mid_{true} = c \quad (26)$$

$$\begin{aligned} c_1 \mid_{\psi_1} = c_2 \mid_{\psi_2} \text{ iff} \\ \forall \sigma_1 \cdot \sigma_1 \models \psi_1 \Rightarrow \left(\exists \sigma_2 \cdot \sigma_2 \models \psi_2 \Rightarrow (c_1[\sigma_1] = c_2[\sigma_2]) \right) \text{ and} \\ \forall \sigma_2 \cdot \sigma_2 \models \psi_2 \Rightarrow \left(\exists \sigma_1 \cdot \sigma_1 \models \psi_1 \Rightarrow (c_1[\sigma_1] = c_2[\sigma_2]) \right) \end{aligned} \quad (27)$$

Figure 19: Equations for connectors and types – only the rules for connectors are included above because the rules for types are analogous to these above, simply replacing the connectors (c_1, c_2) by types (T_1, T_2) .

$$\begin{array}{c} \text{(restriction)} \\ \frac{\Gamma \mid \phi \vdash \psi \quad \Gamma \mid \phi, \psi \vdash c : T}{\Gamma \mid \phi \vdash c \mid_{\psi} : T \mid_{\psi}} \\ \text{(parallel)} \\ \frac{\Gamma \mid \phi \vdash c_1 : I_1 \rightarrow J_1 \mid_{\psi_1} \quad \Gamma \mid \phi \vdash c_2 : I_2 \rightarrow J_2 \mid_{\psi_2}}{\Gamma \mid \phi \vdash c_1 \otimes c_2 : I_1 \otimes I_2 \rightarrow J_1 \otimes J_2 \mid_{\psi_1, \psi_2}} \end{array}$$

Figure 20: Adding restrictions to types. Other rules remain almost the same, adapted in a similar way to the **(parallel)** rule.

include these constraints, also called *type families*, given by TF in the following grammar.

$$\begin{aligned} TF & ::= tf \mid \forall x : P \cdot TF \\ tf & ::= I \rightarrow J \mid tf \mid_{\psi} \end{aligned}$$

For readability we will continue to use the letter T for types instead of TF . The new equation that captures the semantics of the restriction operator is shown in Figure 19. The second equation (Eq. (27)), capturing the general case, states that two restricted connector are equivalent if any (valid) substitution on the first connector can be matched by a (valid) substitution on the second, and vice-versa. The main type rules are presented in Figure 20. The new rule **(restriction)** introduces a constraint ψ from the connector to the context. All other rules are adapted in a similar way to the **(parallel)** rule, by simply collecting the restriction constraints in the conclusions of the rules. For readability we write ‘ ψ_1, ψ_2 ’ to denote ‘ $\psi_1 \wedge \psi_2$ ’. A connector c is now *well-typed* if there is a derivation tree $\emptyset \mid \phi \vdash c : \forall x : P \cdot T \mid_{\psi}$ such that $\phi \wedge \psi$ is satisfiable, i.e., ψ has at least one solution that does not contradict at least one solution of ϕ . This approach resembles Jones’s qualified types [11], where types can be qualified with general predicates; in our work predicates can include only integer and boolean variables, and are not over type variables.

Example Recall the parameterised sequence of fifos from Figure 17. They can be restricted to sequences of at most 5 fifos, yielding the typing judgement:

$$\begin{aligned} \emptyset \mid \mathbf{1} \otimes (n-1) = \mathbf{1}^n \quad , \quad (n-1) \otimes \mathbf{1} = X_I \otimes (n-1) \quad , \quad \mathbf{1}^n = X_J \otimes (n-1) \\ \vdash \lambda n : \mathbb{N} \cdot (\text{Tr}_{n-1}(\gamma_{n-1,1} ; \text{fifo}^n) \mid_{n \leq 5}) \quad : \quad \forall n : \mathbb{N} \cdot X_I \rightarrow X_J \mid_{n \leq 5} \end{aligned}$$

$$\begin{array}{c}
\text{(fam-parallel)} \\
\frac{\Gamma \mid \phi \vdash c_1 : \forall \overline{x_1 : T_1} \cdot I_1 \rightarrow J_1 \mid_{\psi_1} \quad \Gamma \mid \phi \vdash c_2 : \forall \overline{x_2 : T_2} \cdot I_2 \rightarrow J_2 \mid_{\psi_2} \quad \overline{x_1} \cap \overline{x_2} = \emptyset}{\Gamma \mid \phi \vdash c_1 \otimes c_2 : \forall \overline{x_1 : T_1, x_2 : T_2} \cdot I_1 \otimes I_2 \rightarrow J_1 \otimes J_2 \mid_{\psi_1, \psi_2}} \\
\text{(fam-sequence-1)} \\
\frac{\Gamma \mid \phi \vdash c_1 : \forall \overline{x : T} \cdot I_1 \rightarrow J_1 \mid_{\psi} \quad \Gamma \mid \phi \vdash c_2 : \forall \overline{x : T} \cdot I_2 \rightarrow J_2 \mid_{\psi}}{\Gamma \mid \phi \vdash c_1 ; c_2 : \forall \overline{x : T} \cdot I_1 \rightarrow J_2 \mid_{\psi, J_1=I_2}} \\
\text{(fam-sequence-2)} \\
\frac{\Gamma \mid \phi \vdash c_1 : \forall \overline{x_1 : T_1} \cdot I_1 \rightarrow J_1 \mid_{\psi_1} \quad \Gamma \mid \phi \vdash c_2 : \forall \overline{x_2 : T_2} \cdot I_2 \rightarrow J_2 \mid_{\psi_2} \quad \overline{x_1} \cap \overline{x_2} = \emptyset}{\Gamma \mid \phi \vdash c_1 \odot c_2 : \forall \overline{x_1 : T_1, x_2 : T_2} \cdot I_1 \rightarrow J_2 \mid_{\psi_1, \psi_2, J_1=I_2}} \\
\text{(fam-replication)} \qquad \text{(fam-choice)} \\
\frac{\Gamma \mid \phi \vdash \alpha : \mathbb{N} \quad \Gamma, x : \mathbb{N} \mid \phi \vdash c : \forall \overline{x' : P} \cdot I \rightarrow J \mid_{\psi}}{\Gamma \mid \phi \vdash c^{x \leftarrow \alpha} : \forall \overline{x' : P} \cdot I^{x \leftarrow \alpha} \rightarrow J^{x \leftarrow \alpha} \mid_{\psi}} \quad \frac{\Gamma \mid \phi \vdash \psi : \mathbb{B} \quad \Gamma \mid \phi \vdash c_1 : \forall \overline{x_1 : T_1} \cdot I_1 \rightarrow J_1 \mid_{\psi_1} \quad \Gamma \mid \phi \vdash c_2 : \forall \overline{x_2 : T_2} \cdot I_2 \rightarrow J_2 \mid_{\psi_2}}{\Gamma \mid \phi \vdash c_1 \oplus^{\psi} c_2 : \forall \overline{x_1 : T_1, x_2 : T_2} \cdot I_1 \oplus^{\psi} I_2 \rightarrow J_1 \oplus^{\psi} J_2 \mid_{\psi \rightarrow \psi_1, \neg \psi \rightarrow \psi_2}} \\
\text{(fam-trace)} \\
\frac{\Gamma \mid \phi \vdash c : \forall \overline{x : P} \cdot J_1 \rightarrow J_2 \mid_{\psi}}{\Gamma \mid \phi \vdash \text{Tr}_I(c) : \forall \overline{x : P} \cdot X_I \rightarrow X_J \mid_{\psi, J_1=X_I \otimes I, J_2=X_J \otimes I}}
\end{array}$$

Figure 21: Type rules for the lifted composition operators of connectors.

The conjunction of the above constraints is satisfiable: the possible solutions map X_I and X_J to $\mathbf{1}$, and map n to any value between 0 and 5. Hence the connector is well-typed.

4.2. Family composition

Parameterised connectors (Section 3) allow integer and boolean expressions to influence the final connector. However, the existing type rules for composing connectors do not describe how to compose connectors with parameters. The type rules in Figure 21 add support for composing connector families via the operators ‘;’ and ‘ \odot ’. The composition of two parameterised connectors with ‘;’ requires the variables and the restrictions of matching interfaces to be the same, making this composition closer to the categorical composition of morphisms. The composition of two parameterised connectors with \odot produces a new connector parameterised by the parameters of both connectors and restricted by both restrictions. We write $\forall \overline{x : P}$ to represent a (possibly empty) sequence of nested pairs $\forall x : P$. Note that connectors without parameters and restrictions are specific instances of connector families; indeed, the new rules (fam-*) coincide with their simpler counterparts whenever the set of parameters and restrictions are empty.

Examples The two connectors below have the same type: $\forall x_1 : \mathbb{N}, x_2 : \mathbb{N}, x_3 : \mathbb{N} \cdot \mathbf{1}^{x_1} \rightarrow \mathbf{1}^{x_2} \otimes \mathbf{1}^{x_3}$, under a context where $\mathbf{1}^{x_1} = \mathbf{1}^{x_2} \otimes \mathbf{1}^{x_3}$. The first composes 3

connector families, while the second is a family that composes 3 connectors.

$$\begin{aligned} (\lambda x_1 : \mathbb{N} \cdot \text{id}_1^{x_1}) \odot (\lambda x_2 : \mathbb{N} \cdot \text{id}_1^{x_2}) \otimes (\lambda x_3 : \mathbb{N} \cdot \text{id}_1^{x_3}) & \quad (\text{composition of families}) \\ \lambda x_1 : \mathbb{N}, x_2 : \mathbb{N}, x_3 : \mathbb{N} \cdot (\text{id}_1^{x_1} ; \text{id}_1^{x_2} \otimes \text{id}_1^{x_3}) & \quad (\text{family of compositions}) \end{aligned}$$

Observe that the modularity gain with the composition of families is achieved by serialising all input arguments. I.e., it is possible to compose λ -connectors by grouping all of their arguments in a single sequence. As a consequence the tensor product \otimes does not obey the distributive property $(c_1 \odot c_2) \otimes (c_3 \odot c_4) = (c_1 \otimes c_3) \odot (c_2 \otimes c_4)$ whenever the left hand side exists when composing connector families, since the serialisation of the arguments produces different orders. For example, the composition $(\lambda x : \mathbb{N} \cdot f \odot \lambda y : \mathbb{N} \cdot g) \otimes (\lambda z : \mathbb{N} \cdot h \odot \text{id})$ has type $\forall x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \cdot T$, for some type T , while $(\lambda x : \mathbb{N} \cdot f \otimes \lambda z : \mathbb{N} \cdot z) \odot (\lambda y : \mathbb{N} \cdot h \otimes \text{id})$ has type $\forall x : \mathbb{N}, z : \mathbb{N}, y : \mathbb{N} \cdot T$, where the order of the parameters is different. However, this distributed property is preserved for the categorical composition $(;)$ of connectors. This is because $\lambda x : \mathbb{N} \cdot f \odot \lambda y : \mathbb{N} \cdot g$ cannot be well typed, since it requires both terms to be over the same variable.

Type preservation of equations for connectors Each connector type denotes a (possibly empty) family of types, in the sense that it includes a restriction constructor (as defined in [Section 4.1](#)) that may have unsatisfiable constraints. Hence some of the equalities used previously to show type preservation need to be adapted to this family setting. Furthermore, the role of the constraints in the context changed, since the type rules for families allow such constraints to refer to variables (which must be in the context). Hence the type preservation argument is now reformulated: for any variable context Γ such that $\Gamma | \phi_1 \vdash c_1 : T_1$ and $\Gamma | \phi_2 \vdash c_2 : T_2$, for some $c_1, c_2, \phi_1, \phi_2, T_1, T_2$, it must hold that

$$c_1 = c_2 \Rightarrow (T_1 = T_2 \text{ and } \phi_1 \leftrightarrow \phi_2).$$

We sketch the key arguments to prove the equality of restrictions introduced in [Figure 19](#), and leave as future work the exploration of properties (i.e., type preserving equations) of the composition operator \odot , since it does not obey the properties of a composition of morphisms in the category.

Let $c_1 |_{\psi_1} = c_2 |_{\psi_2}$; from where we get the two type judgements $\Gamma | \phi_1 \vdash c_1 : T_1 |_{\psi_1}$ and $\Gamma | \phi_2 \vdash c_2 : T_2 |_{\psi_2}$. By definition of equality one gets that

$$\forall \sigma_1 \cdot \sigma_1 \models \psi_1 \Rightarrow (\exists \sigma_2 \cdot \sigma_2 \models \psi_2 \Rightarrow (c_1[\sigma_1] = c_2[\sigma_2]))$$

and its symmetric property (by swapping 1 and 2). To show that $T_1 = T_2$ one must show that

$$\forall \sigma_1 \cdot \sigma_1 \models \psi_1 \Rightarrow (\exists \sigma_2 \cdot \sigma_2 \models \psi_2 \Rightarrow (T_1[\sigma_1] = T_2[\sigma_2])).$$

and its symmetric property. In the following we omit the arguments for the symmetric property because they do not change. To show the type equality we select σ_1 and σ_2 such that $\sigma_1 \models \psi_1$, $\sigma_2 \models \psi_2$, and $c_1[\sigma_1] = c_2[\sigma_2]$. Furthermore, we can assume without loss of generality that σ_1 and σ_2 instantiate all variables in Γ . The rest of the proof continues by applying the substitutions over the derivation trees of c_1 and c_2 to produce $T_1[\sigma_1]$ and $T_2[\sigma_2]$ (and also $\phi_1[\sigma_1]$ and $\phi_2[\sigma_2]$), and showing that these terms with no variables are indeed the same. These are omitted in this paper.

4.3. Behaviour of connector families

Recall that the behaviour of parameterised connectors was given by tiles over two monoidal categories that shared the objects IP (Section 3.3), where IP extends interfaces with scope variables to match the types of parameterised connectors. Similarly, this section (1) extends interfaces a second time with constraints, to match the types of connector families, and (2) presents a new vertical monoidal category to describe the behaviour of Reo connector families.

Objects - extended interfaces The objects of our Tile Model, previously defined as IP (Section 3.3), are now defined as family of interfaces with parameters (as before) and with *restrictions*, defined by IF in the grammar below.

$$\begin{aligned} IF & ::= if \quad | \quad \forall x : P \cdot IF \\ if & ::= I \quad | \quad if |_{\psi} \end{aligned}$$

Furthermore, we require objects to have non-empty instances, i.e., the conjunction of all restrictions ψ must be satisfiable. This follows the same principle as requiring connector families to have a derivation tree with a satisfiable constraint in order to be well-typed, i.e., the requiring connector families to have instances. We say that two interfaces in IF are equal if their possible instantiations (replacing variables by values that obey the restrictions) are the same, i.e., the rules in Figure 19 also apply to IF after replacing the connectors (c_1, c_2) by extended interfaces (IF_1, IF_2).

Morphisms - evolving families The temporal evolution (execution semantics) of connectors is given by the vertical morphisms of our tile semantics. In Section 3.3 we presented a vertical category with morphisms flow , noFlow , and $\text{inst}(x \mapsto v)$, for any variable x and value v . We now extend these to the new objects of the vertical category IF . More concretely, we leave flow and noFlow to be reflexive morphisms over simple interfaces, i.e., without parameters nor restrictions, and redefine the instantiation morphism for IF . Given a variable $x : P$ we define $\text{inst}(x \mapsto v)$ as the only morphism with signature $(\forall x : P \cdot IF) \rightarrow IF[v/x]$, defined only if $IF[v/x]$ is either a simple interface or if it has at least one element of the family. Intuitively, the instantiation morphism cannot produce interfaces with unsatisfiable restrictions.

For our tile semantics we define a tile for every parameterised connector c with type $\forall x : P \cdot (I \rightarrow J)$ and for every value v of P , exactly as in Section 3.3 (left of Figure 18). The only exception is that the instantiation morphism is the one described above, for families of interfaces (parameterised and with restriction).

Example Consider a simple variation of the n -ary FIFO_1 s in parallel $\text{bn-fifo} = \lambda b : \mathbb{B}, n : \mathbb{N} \cdot (\text{fifo}^n |_{n < 5}) \oplus^b \text{fifo}^n$. This variation is parameterised by a boolean b , indicating whether this connector has to be constrained, and a natural number n with the size of the connector. The bn-fifo connector has type $\forall b : \mathbb{B}, n : \mathbb{N} \cdot (\mathbf{1}^n \oplus^b \mathbf{1}^n) \rightarrow (\mathbf{1}^n \oplus^b \mathbf{1}^n) |_{b \rightarrow n < 5, -b \rightarrow \text{true}}$, or more simply after applying the equations for connectors $\forall b : \mathbb{B}, n : \mathbb{N} \cdot \mathbf{1}^n \rightarrow \mathbf{1}^n |_{b \rightarrow n < 5}$. The only 2 tiles that can be used to evolve the connector are presented on the left of Figure 22, each capturing

a different instantiation of the parameter b . The bottom connector when after the instantiation of the left side has a restriction: $n < 5$. Consequently, the vertical category does not have the morphism $\text{inst}(n \mapsto 6)$ because the resulting interfaces are $6 < 5 \mid_1$, which does not have any instance because $6 < 5$ is false.

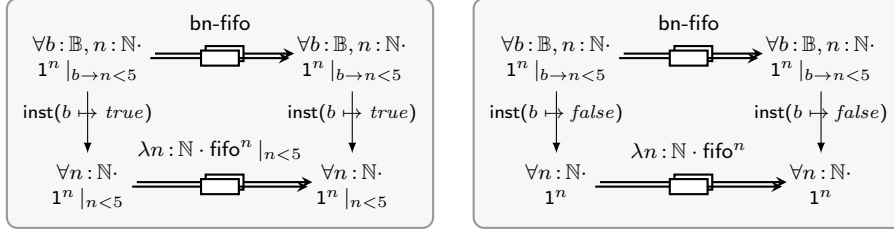


Figure 22: Two only tiles that can be applied to the connector bn-fifo .

Type refinement for connector families Recall that [Section 3.3](#) showed that, when a parameterised connector c evolves via a tile $c \xrightarrow[v']{v} c'$, then the type of c' refines the one of c . We now show that this result extends from *parameterised connectors* to *connector families*. It is enough to observe that the tile semantics remains practically the same, with only two differences: (1) the objects of the tiles can now include constraints, which does not affect the semantics, and (2) the tiles that use the instantiation morphism must have a valid outgoing interfaces (i.e., with satisfiable constraints). In both cases the type of evolving a connector is always a refinement of the original type, since the constraints in the types are either preserved or instantiated (without becoming unsatisfiable).

5. Solving type constraints

This section describes an algorithm to check if the constraints produced by the type rules are satisfiable; if so, this algorithm also provides an assignment of variables to values or to other variables.

Constraint-based approaches to type-checking are well-known, for example, for the lambda calculus [[12](#), Chapter 22], where constraints are solved using an unification algorithm. However, the unification algorithm used for the lambda calculus is not enough for our calculus, because interfaces can include complex expressions that cannot be just syntactically compared. Hence our algorithm performs algebraic rewritings, uses an unification algorithm (for the simpler cases), and invokes a constraint solver (for the more complex cases).

We focus only on untyped ports, represented by $\mathbf{1}$, which mean that any data can go through these ports.⁴ Consequently, interfaces are interpreted as integer expressions, denoting the number of ports, as we will shortly explain.

⁴More precisely, this means that we move from generic traced symmetric monoidal categories to a more concrete PROP category [[7](#)], where we develop our type-checking algorithm.

5.1. Overview

In our type-checking algorithm interfaces are interpreted as integers, by mapping constructors of interfaces to integer operations. For example, $(\mathbb{1}) = 1$, $(I \otimes J) = (I) + (J)$, $(I^\alpha) = (I) * \alpha$, and $(I^{x \leftarrow \alpha}) = \sum_{x=0}^{\alpha-1} (I)$, where (I) represents the interpretation of I as an integer. Both the constraints that appear in the context and the constraints that appear in the type are combined, hence producing a type $\forall x: \overline{P} \cdot I \rightarrow J \mid_\psi$, where ψ represents the conjunction of these constraints.

We exemplify our approach using the *zip* connector (Figure 15), under the restriction that n must be smaller than 5. This example was analysed automatically using our prototype implementation, which will be presented in Section 5.3 and follows closely our type rules. The type rules produce the type $\forall n: \mathbb{N} \cdot \mathbf{1}^{x_1} \rightarrow \mathbf{1}^{x_2} \mid_\psi$, where ψ is defined below (after interpreting the interfaces as integer expressions), and x_1, x_2 are variables introduced by the (trace) rule (Figure 7). We do not present the associated derivation tree for simplicity.

$$\psi = \begin{cases} x_1 + (2*n) * (n - 1) = \sum_{x=0}^{n-1} (n - x) + (2*x) + (n - x) , \\ x_2 + (2*n) * (n - 1) = (2*n) + ((2*n) * (n - 1)) , \\ ((2*n) * (n - 1)) + (2*n) = \sum_{x=0}^{n-1} (n - x) + (2*x) + (n - x) , n < 5 \end{cases}$$

Using algebraic laws such as distributivity, commutativity, and associativity of sums and multiplications, the constraints are simplified as follows.

$$x_1 = 2n , x_2 = 2n , n < 5$$

The unification algorithm then produces the substitution below, leaving the $n < 5$ constraint to be handled in a later phase.

$$[2n/x_1] \circ [2n/x_2]$$

The final step is to verify that the remaining constraint ($n < 5$) is satisfiable using a constraint solver, allowing us to conclude that the connector is well-typed. Furthermore, applying the substitution above to the type produced by the type rules gives the most general type $\forall n: \mathbb{N} \cdot 2n \rightarrow 2n \mid_{n < 5}$. The constraint solver provides a solution, say $\{n \mapsto 0\}$, which can be used to produce an instance of the general type: $0 \rightarrow 0$.

This three-phase approach (simplification, unification, and constraint solving) can be reduced to only constraint solving. I.e., to know if a connector c is well-typed it is enough to collect the constraints using the type rules, and to directly use a constraint solver without any simplification or unification. However we found this very limited in practice. For example, knowing that the type of the *zip* connector is $\forall n: \mathbb{N} \cdot \mathbf{1}^{x_1} \rightarrow \mathbf{1}^{x_2} \mid_\psi$ under the constraints mentioned above is not fully satisfactory: it is very difficult to understand how the interfaces vary with n . The extra simplification and unification phases reuse known type-checking techniques that can handle most simple cases (without constraints and with basic expressions). In our example, the type $\forall n: \mathbb{N} \cdot 2n \rightarrow 2n \mid_{n < 5}$ already provides a much clearer description of the interfaces of *zip*. Hence, using this phased approach to type-check it is possible to combine the simplicity of

the resulting types obtained via unification with the power of more complex constraints that need to be delegated to constraint solvers.

Towards typed ports This section focus on untyped ports, because the general constraint solving problem is then reduced to integer constraint solving problem, which is easier to tackle, and because it already covers a large range of connectors. However a similar approach could have been followed for typed ports, at the cost of performance, as we briefly explain. Port types could be seen as elements from an enumerable and finite global set T (instead of the singleton type 1). In this setting, interfaces could be represented as lists of port types (instead of integers), such that $(I \otimes J) = ([I]) \cdot ([J])$ (where \cdot concatenates 2 lists) and $(I^{x \leftarrow \alpha}) = ([I[0/x]]) \cdot \dots \cdot ([I[\alpha - 1/x]])$. Consequently simplifying connector types and constraints would be more complex, since the algebraic manipulation is more limited when working with lists. The constraint solving phase would no longer be handled by an integer constraint solver, but it would require a more general solver. For example, a rewriting engine (such as Prolog) could be used to solve equations over lists based on unification and backtracking.

5.2. Three-phase solver

This section explains in more detail the three-phase algorithm used to reason about constraints, exemplified in the previous subsection. These phases are performed in sequence, and consist of the *simplification* phase, the *unification* phase, and the *constraint-solving* phase, explained below.

Simplification This first phase prepares the constraints obtained by the type rules to be used by the unification phase. More specifically, it rewrites the constraints by applying algebraic laws of sums and multiplications, building a polynomial and manipulating the coefficients. For example, sums like $\sum_{x=n1}^{n2-1} (5 * x)$, where $5 * x$ is linear on x , are rewritten into $(5 * n2 + 5 * n1) * (n2 - n1) / 2$; to avoid integer divisions the denominator 2 is dropped and the other coefficients are multiplied by 2. Equalities are rewritten to match, if possible, the pattern $x = \alpha$, which is exploited by the unification phase.

Note that the type rules, apart from (*restriction*), only produce equalities of integer expressions. Our choice of rewrites included in the prototype implementation took into account the constraints generated by the type rules using a range of different connectors. These rewrites are able to simplify all the examples presented in this paper that do not use inequalities, most of which involve only linear expressions or are reduced to linear expressions, to a point where the constraint solving phase was not needed. Furthermore, other fast off-the-shelf technologies, such as computer algebra systems, could be used to quickly manipulate and simplify more complex expressions.

Unification The second phase consists of a traditional unification algorithm [12, Chapter 22] adapted to our type system, which produces both an *unification* and a set of constraints postponed to the constraint solving phase. An unification is formally a sequence of substitutions $\sigma_1 \circ \dots \circ \sigma_n$, and applying a unification to a connector or interface t consists of applying the substitutions

$$\begin{aligned}
\text{unify}(\phi) &= \text{unify}(\phi; \text{true}) \\
\text{unify}(\text{true} \quad ; \psi) &= (\emptyset; \psi) & (\sigma; \psi) \bar{\circ} \sigma' &= (\sigma \circ \sigma'; \psi) \\
\text{unify}(\text{true}, \phi \quad ; \psi) &= \text{unify}(\phi; \psi) \\
\text{unify}(\alpha = \alpha', \phi; \psi) &= \\
&\begin{cases} \text{unify}(\phi \quad ; \psi) & \text{if } \alpha \equiv \alpha' \\ \text{unify}(\phi[\alpha'/x]; \psi[\alpha'/x]) \bar{\circ} [\alpha'/x] & \text{if } \alpha \equiv x \text{ and } x \notin \text{fv}(\alpha') \\ \text{unify}(\phi[\alpha/x]; \psi[\alpha'/x]) \bar{\circ} [\alpha/x] & \text{if } \alpha' \equiv x \text{ and } x \notin \text{fv}(\alpha) \\ \text{unify}(\phi \quad ; \psi, \alpha = \alpha') & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 23: Unification algorithm for constraints over boolean and integer variables.

in sequence $((t \sigma_1) \dots) \sigma_n$. For example, unifying the constraints $x = 2 + y, z = 3 + x, y = w$ produces the sequence of substitutions $[2 + y/x] \circ [3 + 2 + y/z] \circ [w/y]$. Applying this unification to an interface means first substituting x by $2 + y$, followed by the substitutions of z and y . The resulting interface is guaranteed to have no occurrences of x, y , nor z , and not to have w bound by any constraint.

The unification algorithm is described by the `unify` function (Figure 23) that, given a set of constraints ϕ to be solved, returns a pair with a unification and a set of postponed constraints. The core of `unify` is defined in the right side of Figure 23. For every equality $\alpha = \alpha'$, it first checks if they are syntactically equivalent (using \equiv). It then checks if either the left or the right side is a variable that does not occur on the other side; if so, it adds the equality to the resulting unification. If none of these cases apply, it postpones the analysis of the constraint for the third phase, by using the second argument of `unify` as an accumulator.

Constraint solving The last phase consists of collecting the constraints postponed by the unification phase and use an off-the-shelf constraint solver. This will tell us if the constraints are satisfiable, producing a concrete example of a substitution that satisfies the constraints. In the example of the sequence of fifos with at most 5 fifos (Section 5.1), a possible solution for the constraints is $\{n \mapsto 4, x_1 \mapsto 1, x_2 \mapsto 1\}$. This substitution, when applied to the type obtained for *seq-fifo*, yields a concrete type instance *seq-fifo* : $\mathbf{1} \rightarrow \mathbf{1}$. In this example the concrete type instance matches its general type $(\forall n : \mathbb{N} \cdot \mathbf{1} \rightarrow \mathbf{1})$, since the value of n does not influence the type of the connector.

Note that a wide variety of approaches for solving constraints exist. One can use, for example, numerical methods to find solutions, or SMT solvers over some specific theory. The expressive power supported by the constraint solver dictates the expressivity of the expressions α and ϕ used in the connector, which we are abstracting away in this paper. The choices made in our proof-of-concept implementation, briefly explained in the next subsection, are therefore not strict and can be rethought if necessary. For example, one could use a Prolog-like engine to search for solutions based on backtracking, or a more dedicated mathematical engine if the restrictions involve more complex operations such as derivatives or trigonometric calculations. The rewriting rules included in our prototype tool are for simple arithmetic, the solver is a Java library for integer constraints, ba-

```

import paramConnectors.DSL...
val x = "x":I ; val n = "n":I ; val b = "b":B

//-----  $\lambda x:\mathbb{N} \cdot (\text{fifo}^x \mid_{x>5})$  -----//
typeOf( lam(x, (fifo^x) | (x>5)) )
// returns  $\forall x:I . x \rightarrow x \mid x > 5$ 
typeInstance( lam(x, (fifo^x) | (x>5)) )
// returns  $\odot 6 \rightarrow 6$ 
typeSubstitution( lam(x, (fifo^x) | (x>5)) )
// returns  $\odot [x:I \rightarrow 6]$ 

//----- seq-fifo -----//
typeOf( lam(x, Tr(x-1, sym(x-1,1) & (fifo^x))) )
// returns  $\forall x:I . 1 \rightarrow 1$  [type obtained only after constraint solving]
typeTree( lam(x, Tr(x-1, sym(x-1,1) & (fifo^x))) )
// returns  $\forall x:I . x_1 \rightarrow x_2 \mid ((x_1 + (x - 1)) == ((x - 1) + 1))$ 
//  $\& ((x_2 + (x - 1)) == x) \& ((1 + (x - 1)) == x) \& (x_1 \geq 0) \& (x_2 \geq 0)$ 

//----- composing families -----//
typeOf( lam(x, fifo^x | x<5) & lam(y, id^y) )
// returns  $\forall x:I, y:I . y \rightarrow y \mid (y < 5) \& (x == y)$ 

//----- zip and sequencer -----//
val zip = /*...*/ ; sequencer = /*...*/
typeOf( zip )
// returns  $\forall n:I . 2 * n \rightarrow 2 * n$ 
typeOf( sequencer )
// returns  $\forall n:I . n \rightarrow n$ 

```

Listing 1: Calculating the type of connectors using our tools.

sic interfaces are not typed, and a small fixed set of operators is used for integer and boolean expressions. With more complex interfaces or more complex set of operators we would need to use different tools and/or rewriting rules.

5.3. Implementation

We developed a proof-of-concept implementation in the Scala programming language that covers all the examples described in this paper, which can be found online.⁵ Listing 1 exemplifies the usage of this library—more examples can be also found online.

This implementation includes a simple domain specific language to specify connectors, making them similar to the syntax used throughout this paper. Composition of (families of) connectors is performed via the $\&$ operator, which corresponds to the composition operator \odot for connector families. It provides four main top-level functions: `typeTree`, `typeOf`, `typeInstance`, and `typeSubstitution`. The first creates the derivation tree (if it exists); `typeOf` simplifies the constraints, uses the unification algorithm, invokes the constraint

⁵<https://github.com/joseproenca/parameterised-connectors>

solver, and returns the most general type that it found (i.e., it simplifies the type using term rewriting and unification, and instantiates variables that have unique solutions); and `typeInstance` and `typeSubstitution` perform the same steps as `typeOf`, but the former returns the result of the constraint solving phase (even if the type is not the most general one) and the latter returns the substitutions obtained by the unification and the constraint solver phases. Hence the result of `typeInstance` never includes constraints. Finally, a function `debug` prints all intermediate calculations when type-checking a connector (family). The constraint solving phase uses the Choco solver⁶ to search for solutions of the constraints.

Observe that the resulting type instance and substitution of the first connector start with \odot —this means that the resulting type is a concrete instance of a type, i.e., the constraint solving phase found more than one solution for the variables of the inferred type (after unification). However, if we would ask for a type instance of $(\lambda x : \mathbb{N} \cdot \text{fifo}^x | x > 5)(7)$, for example, the result would be also its (general) type $7 \rightarrow 7$, without the \odot . Typing the connector $(\lambda x : \mathbb{N} \cdot \text{fifo}^x | x > 5)(2)$ gives a type error, because the constraints are not satisfied.

6. Modelling more Connector Families

Our calculus for families is exemplified in this paper using Reo’s notation and semantics. We claim, however, that the usefulness of our calculus extends Reo. To support this claim, we use a different set of primitives to model Petri Nets [13, 14] and BIP connectors [15].

6.1. Families of Petri Nets

Petri Nets are a formalism with a well-known graphical notation that capture the movement of resources. More precisely, it is depicted as a graph of places (\circ) and transitions (\blacksquare), combined by linking places to transitions and transitions to places. Two simple examples can be found in Figure 24.

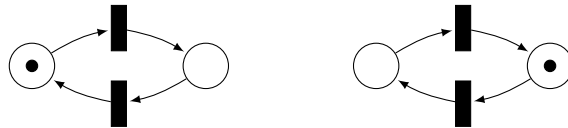


Figure 24: Example of a simple Petri Net in two different configurations.

Places can contain tokens, denoted by \bullet . We will consider the simple variant of Petri Nets whose places can contain at most one token. The semantics of such Petri Net is given by sequences of transitions that *fire*. A transition t is fired when all its incoming places (places linked to t) have a token, and when all its outgoing places (places linked from t) have no token. After t is fired the

⁶<http://choco-solver.org>

incoming places become empty, and the outgoing places become full (with a token). In the example from Figure 24, the Petri Net on the left can fire the top transition and evolve to the Petri Net on the right, which in turn can fire the bottom transition and evolve back to the transition on the left.

Basic connector calculus for Petri Nets Recall the basic connector calculus described in Section 2. Petri Nets can be modelled using the same calculus with a different set of primitive connectors \mathcal{P} and port types A , introduced in Figure 25. The resulting calculus roughly coincides with the *Petri calculus* introduced by Sobociński [16, 17], where he also extends (stateless) connector algebras [3] to model Petri Nets, exploring in more detail the semantics of Petri Nets and of the Petri calculus.

$$\begin{array}{l}
 p \in \mathcal{P} ::= \textcircled{} : 1 \rightarrow 1 \quad | \quad \nabla_I : I \rightarrow 1 \\
 \quad | \textcircled{\bullet} : 1 \rightarrow 1 \quad | \quad \Delta_I : 1 \rightarrow I \\
 \quad | \blacksquare : 1 \rightarrow 1 \quad | \quad \vee_I : I \rightarrow 1 \\
 \quad | \uparrow : 0 \rightarrow 1 \quad | \quad \wedge_I : 1 \rightarrow I \\
 \quad | \downarrow : 1 \rightarrow 0
 \end{array}
 \quad
 \begin{array}{l}
 I, J ::= I \otimes J \text{ tensor} \\
 \quad | \mathbf{0} \quad \text{empty interface} \\
 \quad | \mathbf{1} \quad \text{singleton port type}
 \end{array}$$

Figure 25: Primitives and interfaces for connector calculus for Petri Nets

With this new set of primitives and interfaces, the Petri Nets in Figure 24 can be encoded as $\text{Tr}_1(\textcircled{\bullet}; \blacksquare; \textcircled{}; \blacksquare)$ and $\text{Tr}_1(\textcircled{}; \blacksquare; \textcircled{\bullet}; \blacksquare)$, respectively. The trace constructor binds the output of the last transition to the first place, building a loop between a two places. The type rules confirm that these two connectors have type $0 \rightarrow 0$, i.e., they have no input and no outputs. The parameters of both traces have the type $1 \rightarrow 1$.

The semantics of our calculus of Petri Nets can be given using the same tile model as described in Section 2.4, using the tiles in Figure 26 for the newly introduced primitives. For simplicity we included only $\Delta_{1 \otimes 1}$, $\nabla_{1 \otimes 1}$, $\wedge_{1 \otimes 1}$, and $\vee_{1 \otimes 1}$, but these can be easily extrapolated for arbitrary interfaces. Note that the notation for these mergers and splitters differ from the one used for Reo. Namely, the connector ∇_I was previously used to merge two dataflows exclusively (either one or the other), while here it merges dataflows inclusively (both one and the other) following [16].

Families of Petri Nets The power of parameters and restrictions in connector families can now be applied to this basic calculus of Petri Nets, using the same definition as in Section 4. A simple example of an n -ary Petri Net can be found in Figure 27, where a token alternates between being in a (left) place and replicated in n places (right). The type of this parameterised connector is $\forall n : \mathbb{N} \cdot 0 \rightarrow 0$, i.e., given any natural number n , it has no open inputs nor outputs, as one would expect.

Modelling Feature Nets Feature Nets [14] are Petri Nets parameterised on a selection of features that determine which transitions are active (i.e., can be fired). More concretely, transitions are marked with a so-called *application condition*, which is a propositional formula over feature names. Figure 28 depicts

$$\begin{aligned}
\circ &= \left\{ \circ \xrightarrow[\text{noFlow}]{\text{flow}} \bullet, \circ \xrightarrow[\text{noFlow}]{\text{noFlow}} \circ \right\} & \blacksquare &= \left\{ \blacksquare \xrightarrow[\text{flow}]{\text{flow}}, \blacksquare \xrightarrow[\text{noFlow}]{\text{noFlow}} \blacksquare \right\} \\
\bullet &= \left\{ \bullet \xrightarrow[\text{flow}]{\text{noFlow}} \circ, \bullet \xrightarrow[\text{noFlow}]{\text{noFlow}} \bullet \right\} & \uparrow &= \left\{ \uparrow \xrightarrow[\text{noFlow}]{0} \uparrow \right\} & \downarrow &= \left\{ \downarrow \xrightarrow[0]{\text{noFlow}} \downarrow \right\} \\
\Delta_{1 \otimes 1} &= \left\{ \Delta_{1 \otimes 1} \xrightarrow[\text{flow} \otimes \text{flow}]{\text{flow}} \Delta_{1 \otimes 1}, \Delta_{1 \otimes 1} \xrightarrow[\text{noFlow} \otimes \text{noFlow}]{\text{noFlow}} \Delta_{1 \otimes 1} \right\} \\
\nabla_{1 \otimes 1} &= \left\{ \nabla_{1 \otimes 1} \xrightarrow[\text{flow}]{\text{flow} \otimes \text{flow}} \nabla_{1 \otimes 1}, \nabla_{1 \otimes 1} \xrightarrow[\text{noFlow}]{\text{noFlow} \otimes \text{noFlow}} \nabla_{1 \otimes 1} \right\} \\
\wedge_{1 \otimes 1} &= \left\{ \wedge_{1 \otimes 1} \xrightarrow[\text{flow} \otimes \text{noFlow}]{\text{flow}} \wedge_{1 \otimes 1}, \wedge_{1 \otimes 1} \xrightarrow[\text{noFlow} \otimes \text{flow}]{\text{flow}} \wedge_{1 \otimes 1}, \wedge_{1 \otimes 1} \xrightarrow[\text{noFlow} \otimes \text{noFlow}]{\text{noFlow}} \wedge_{1 \otimes 1} \right\} \\
\vee_{1 \otimes 1} &= \left\{ \vee_{1 \otimes 1} \xrightarrow[\text{flow}]{\text{flow} \otimes \text{noFlow}} \vee_{1 \otimes 1}, \vee_{1 \otimes 1} \xrightarrow[\text{flow}]{\text{noFlow} \otimes \text{flow}} \vee_{1 \otimes 1}, \vee_{1 \otimes 1} \xrightarrow[\text{noFlow}]{\text{noFlow} \otimes \text{noFlow}} \vee_{1 \otimes 1} \right\}
\end{aligned}$$

Figure 26: Behaviour of Petri Net primitives using tiles.

$$\lambda n : \mathbb{N} \cdot \text{Tr}_1(\bullet; \blacksquare; \Delta_{1^n}; \circ^n; \nabla_{1^n}; \blacksquare) = \left. \begin{array}{c} \bullet \\ \swarrow \quad \searrow \\ \blacksquare \quad \blacksquare \\ \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \circ \quad \circ \quad \vdots \quad \circ \quad \circ \end{array} \right\} n \text{ times}$$

Figure 27: N -ary Petri Net example: connector (left) and its visual representation (right).

an example of a Feature Net borrowed from [14], where transitions are labelled (above) with a propositional formula over the feature names *Coffee* and *Milk*. When the *Coffee* feature is selected and the *Milk* feature is not, the Feature Net behaves as if only the two left transitions could be fired, because the application condition $Coffee \wedge Milk$ evaluates to false.

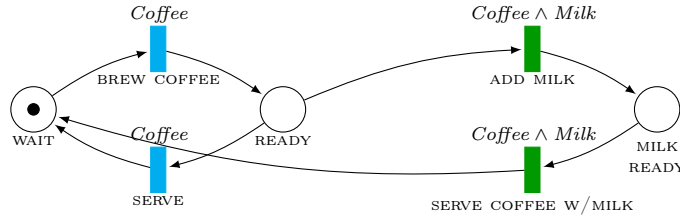


Figure 28: Feature Net for a coffee machine with two features: *Coffee* and *Milk*.

Feature Nets can be easily modelled using boolean variables in our parame-

terised calculus, and introducing a new primitive $\square : 1 \rightarrow 1$ with behaviour given by a single tile $\square \xrightarrow[\text{noFlow}]{\text{noFlow}} \square$. I.e., introducing a new transition construct that can never be fired. The example in Figure 28 can then be specified in our calculus as follows.

$$\lambda \text{Coffee} : \mathbb{B}, \text{Milk} : \mathbb{B} \cdot \text{Tr}_1(\odot; \text{cf}; \circ; \Delta; \text{cf} \otimes (\text{mi}; \circ; \text{mi}); \vee)$$

$$\text{where } \text{cf} = \blacksquare \oplus^{\text{Coffee}} \square \quad \text{and} \quad \text{mi} = \blacksquare \oplus^{\text{Coffee} \wedge \text{Milk}} \square$$

This example uses the choice operator \oplus to decide on whether each transition is active (using \blacksquare) or inactive (using \square), with a boolean predicate given precisely by the application condition. The variables of these predicates are then used as variables under lambda abstractions in our parameterised calculus. The same approach can be used to any Feature Net with transitions marked by application conditions.⁷

Feature models can also be added to our connectors via restrictions. A feature model describes valid combinations of features, often specified as a feature diagram [18]. Such feature models are typically formalised as boolean constraints over the set of features. For example, a feature model for the coffee machine family may impose that $\text{Coffee} \rightarrow \text{Milk}$, i.e., that the *Milk* feature can only be selected when the *Coffee* feature is selected. This fits our calculus of connector families using restrictions; the example above with the restriction $\text{Coffee} \rightarrow \text{Milk}$ can be written as $\lambda \text{Coffee} : \mathbb{B}, \text{Milk} : \mathbb{B} \cdot \text{Tr}_1(\odot; \text{cf}; \circ; \Delta; \text{cf} \otimes (\text{mi}; \circ; \text{mi}); \vee) \mid_{\text{Coffee} \rightarrow \text{Milk}}$.

6.2. BIP families

A coordinated system in BIP, following e.g. [19], is comprised of a set of *primitive components* connected by an *interaction model*. We do not consider priority in BIP, as most papers on BIP also disregard it, and because it has little influence in our contribution to express families. Each primitive component B_i is a labelled transition system with an associated set P_i of labels (ports), and where sets of ports P_i are pairwise disjoint. An interaction model γ is a set of non-empty sets of ports called *interactions*, where each of these interactions describes a set of ports that performs a multi-way synchronisation, i.e., either all ports execute at the same time or none can execute. The example in Figure 29 depicts a BIP system, borrowed from [19, Section 6], where a blackboard component Blb with ports b1 and b2 interacts with a controller Contr and with 3 services Srv_i via 4 possible interactions. Each of these interactions is depicted with a different colour, binding ports that are expected to occur simultaneously.

Basic connector calculus for BIP BIP can be modelled in our basic calculus by representing interaction models using connectors that connect ports of

⁷The authors [14] also present a semantics where arcs—and not transitions—are marked with application conditions; the encoding of such nets can be modelled in our parameterised calculus using an analogous approach to what we presented.

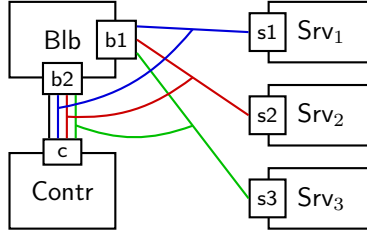


Figure 29: Five BIP components interacting with each other via 4 possible interactions: synchronising either $[b2,c]$ (black), $[b1,s1,b2,c]$ (blue), $[b1,s2,b2,c]$ (red), or $[b1,s3,b2,c]$ (green).

primitive components. These connectors use merger and splitters in a similar way to our encoding of Petri Nets, indicating what ports can have dataflow together. The idea of using mergers, splitters, and other primitive connectors to describe interactions is analogue to the ideas behind *interaction formulas* [19], which are propositional logic formulas whose solutions describe interaction models. More specifically, these connectors compose ports by combining a primitive set of connectors, whereas each contribute to a global set of constraints over the behaviour of these ports. Our approach also follows the ideas from Bruni et al. [17], where BIP is encoded as a Petri Net variation, and in turn described using the *Petri calculus* (an algebra of connectors with buffers mentioned in the previous subsection). The authors encode each component as a connector with type $0 \rightarrow n$, where n is its number of ports. Our representation of BIP as a connector algebra differs in that it distinguishes between left and right ports, where left and right could be interpreted as input ports and output ports, for example. I.e., a component with n left ports and m out ports has type $n \rightarrow m$, instead of $0 \rightarrow (n+m)$. Consequently it provides a more compact representation at the cost of distinguishing between left (input) and right (output) ports.

$$\begin{array}{l|l|l}
 p \in \mathcal{P} ::= B_i : I_i \rightarrow J_i & \nabla_I : I \rightarrow \mathbf{1} & I, J ::= I \otimes J \text{ tensor} \\
 | \uparrow : \mathbf{0} \rightarrow \mathbf{1} & \Delta_I : \mathbf{1} \rightarrow I & | \mathbf{0} \quad \text{empty interface} \\
 | \downarrow : \mathbf{1} \rightarrow \mathbf{0} & \vee_I : I \rightarrow \mathbf{1} & | \mathbf{1} \quad \text{singleton port type} \\
 | \uparrow : \mathbf{0} \rightarrow \mathbf{1} & \wedge_I : \mathbf{1} \rightarrow I & \\
 | \downarrow : \mathbf{1} \rightarrow \mathbf{0} & &
 \end{array}$$

Figure 30: Primitives and interfaces for connector calculus for BIP

The chosen set of primitive connectors for our BIP calculus is introduced in Figure 30, which overlaps with the previous calculus for Petri Nets. These connectors, instead of places and transitions, include a set of primitive components B_i , for some indexing set K and $i \in K$, with left ports I_i and right ports J_i . The distinction between left and right port is not related to direction of dataflow, although it could be used for such task.

The tile semantics of the primitive connectors that are not defined for Petri Nets (Figure 26) is specified in Figure 31. The connectors \uparrow and \downarrow represents connectors with one ports, which is always ready to have dataflow. Components

can evolve by having flow on some of its ports (left or right) and no-flow on the remaining ports. This is modelled by including one tile for each transition, using the vertical morphisms `flow` and `noFlow` to represent ports that have dataflow. For example, if a component B has two left ports a, b and a right port c , and it has a transition $q \xrightarrow{a,c} q'$, then we include the tile $B_q \xrightarrow[\text{flow}]{\text{flow} \otimes \text{noFlow}} B_{q'}$. Here we write B_q to represent the labelled transition system B with initial state q , and assume ports have a fixed order.

$$B_i = \left\{ \begin{array}{l} B_i \xrightarrow[\text{flow}]{a_{i1} \otimes \dots \otimes a_{in}} B'_i \\ b_{i1} \otimes \dots \otimes b_{im} \end{array} \right. \left. \begin{array}{l} B \text{ has a transition } q \xrightarrow{A} q', \\ q \text{ is the initial state of } B_i, \\ B'_i \text{ is the same as } B_i \text{ with initial state } q', \\ \text{each } a \in \{a_{i1}, \dots, a_{in}\} \text{ is a left port,} \\ \text{each } b \in \{b_{i1}, \dots, b_{im}\} \text{ is a right port,} \\ a = \text{flow if } a \in A, \text{ otherwise } a = \text{noFlow, and} \\ b = \text{flow if } b \in A, \text{ otherwise } b = \text{noFlow.} \end{array} \right\}$$

$$\uparrow = \left\{ \uparrow \xrightarrow[\text{flow}]{0} \uparrow, \uparrow \xrightarrow[\text{noFlow}]{0} \uparrow \right\} \quad \downarrow = \left\{ \downarrow \xrightarrow[0]{\text{flow}} \downarrow, \downarrow \xrightarrow[0]{\text{noFlow}} \downarrow \right\}$$

Figure 31: Behaviour of BIP primitive connectors using tiles.

Using this calculus for BIP systems, the example from Figure 29 can be written using basic connector calculus as presented in Figure 32. Here we assume that the primitive components have type $\text{Blb} : 0 \rightarrow 1 \otimes 1$, $\text{Contr} : 0 \rightarrow 1$, and for each $i \in \{1, 2, 3\}$, $\text{Srv}_i : 1 \rightarrow 0$.

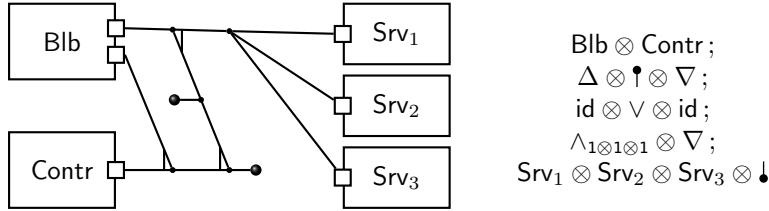


Figure 32: Re-specification of the BIP system in Figure 29 using connector calculus; the left side presents a visual representation of the expression on the right side.

Families of BIP connectors The basic connector calculus for BIP systems presented above is now extended for connector families. A simple example of n -ary BIP systems is described in Figure 33, which adapts the system in Figure 32 to support an arbitrary number of services. It uses a single parameter n that is used to determine how many right ports the (exclusive) splitter \wedge has, and how many instances of Srv are used.

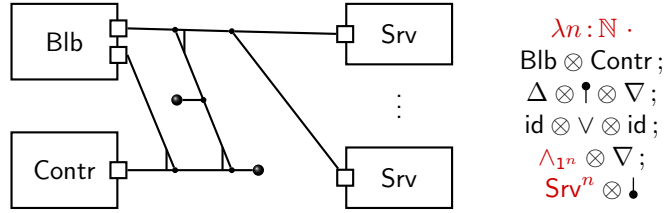


Figure 33: N -ary BIP system: connector family (right) and its visual representation (left).

7. Related Work

Algebras of connectors The usage of symmetric monoidal categories to represent Reo connectors (and others) has been introduced by Bruni et al. [3], where they introduce an algebra of stateless connectors with an operational semantics expressed using the Tile Model [1]. The authors focus on the behavioural aspects, exploiting normalisation and axiomatisation techniques. An extension of this work dedicated to Reo connectors [4] investigates more complex semantics of Reo (with context dependent connectors) using the Tile Model. Other extensions to connector algebras exist. For example, Sobocinski [16], and more recently Bonchi et al. [20], present stateful extensions to model and reason about the behaviour of Petri Nets and of Signal Flow Graphs, respectively. The former was introduced in Section 6.1. The latter also describes the usage of traces (Tr) as a possible way to specify loops in their algebra. In all these approaches, interfaces (objects of the categories) can be either input or output ports, independently of being on the left or right side of the connector (morphism), focusing on the behaviour of connectors instead of how to build families of these connectors.

In our work we do not distinguish input from output ports, assuming data always flows from left to right, and use traces to support loops and achieve the same expressivity. As a result, we found the resulting connectors to be easier to read and understand. For example the connector fifo has type $\bullet \circ \rightarrow 0$ in Bruni et al.’s algebra, meaning that the left side has two ports: an input \bullet and an output \circ one. Composing two fifos in sequence uses extra connectors (called nodes) and has type $0 \rightarrow \circ \bullet$ —for a more complete explanation see [1]. Indeed, our algebra has stronger resemblances with lambda calculus (and with pointfree style in functional programming [21]), facilitating the extension to families of connectors, which is the main novelty of this work.

Analysis of software product lines In the context of software product lines Kästner et al. [22], for example, investigated how to lift a type-checking algorithm from programs to families of programs. They use featherweight Java annotated with constraints used during product generation, and present a type-checking approach that preserves types during this product generation. Their focus is on keeping the constraints being solved as small as possible, unlike previous approaches in the generative programming community (e.g., by Thaker et

al. [23]) that compile a larger global set of constraints. Many other verification approaches for software product lines have been investigated [24, 25, 26, 27]. Post and Sinz [24] verify families of Linux device drivers using the CBMC bounded model checker, and Apel et al. [25] verify more general families of C programs using the CPAchecker symbolic checker. More recently Thüm et al [26] presents an approach to use the KeY theorem prover to verify a feature-oriented dialect of Java with JML annotations. They encode such annotated families of Java programs into new (traditional) Java programs with new JML annotations that can be directly used by KeY to verify the family of products. Dimovski et al [27] take a more general view and provide a calculus for modular specification of variability abstractions, and investigate tradeoffs between precision and time when analysing software product lines and abstractions of them.

Our approach targets connector and component interfaces instead of typed languages, and explicitly uses parameters that influence the connectors. Consequently, feature models can contribute not only with feature selections but also with values used to build concrete connectors. Our calculus is simpler than other more traditional programming languages since it has no statements, no notion of heap or memory, nor tables of fields or methods.

8. Conclusion and Future Work

This paper formalises a calculus for connector families, i.e., for connectors (or components) with an open number of interfaces and restricted to given constraints. A dependent type system guarantees well-connectedness of such families, i.e., that interfaces of subconnectors can be composed as long as the parameters obey the constraints in the type. These constraints are reducible to nonlinear constraints on integers when considering untyped ports (only the type `1`), in which case arithmetic properties and integer constraint solvers can be used to check the constraints under which a connector family is well-connected. The runtime semantics of families of connectors is based on the Tile Model, using a different monoidal category over interfaces (other than the one given by connectors and interfaces) to model both the instantiation of parameters and the execution of discrete steps of the connector.

In the future we will explore more deeply the equations for families of connectors, providing a better insight on different categories of (families of) connector calculus and their relation with the type rules. From a more practical perspective, we will investigate approaches to type check connector families where the type of the data passing through the ports is also checked. Finally, we also plan to investigate how to reduce the size of the constraints being solved, by using the more dedicated contexts while building the type tree instead of collecting the constraints for a follow-up phase, similarly to the work of Kästner et al. [22].

Acknowledgements

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826. This work is also partially funded by the personal grant from FCT – Fundação para a Ciência e a Tecnologia – with reference SFRH/BPD/91908/2012.

- [1] F. Gadducci, U. Montanari, [The tile model](#), in: G. D. Plotkin, C. Stirling, M. Tofte (Eds.), *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, The MIT Press, 2000, pp. 133–166.
URL <http://dl.acm.org/citation.cfm?id=345868.345889>
- [2] J. Proença, D. Clarke, Typed connector families, in: C. Braga, P. C. Ölveczky (Eds.), *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, Vol. 9539 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 294–311. doi:[10.1007/978-3-319-28934-2_16](https://doi.org/10.1007/978-3-319-28934-2_16).
- [3] R. Bruni, I. Lanese, U. Montanari, A basic algebra of stateless connectors, *Theor. Comput. Sci.* 366 (1-2) (2006) 98–120. doi:[10.1016/j.tcs.2006.07.005](https://doi.org/10.1016/j.tcs.2006.07.005).
- [4] F. Arbab, R. Bruni, D. Clarke, I. Lanese, U. Montanari, Tiles for Reo, in: A. Corradini, U. Montanari (Eds.), *Recent Trends in Algebraic Development Techniques*, Vol. 5486 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 37–55. doi:[10.1007/978-3-642-03429-9_4](https://doi.org/10.1007/978-3-642-03429-9_4).
- [5] F. Arbab, Reo: A channel-based coordination model for component composition, *Mathematical Structures in Computer Science* 14 (3) (2004) 329–366. doi:[10.1017/S0960129504004153](https://doi.org/10.1017/S0960129504004153).
- [6] P. Selinger, A survey of graphical languages for monoidal categories, in: B. Coecke (Ed.), *New Structures for Physics*, Vol. 813 of *Lecture Notes in Physics*, Springer Berlin Heidelberg, 2011, pp. 289–355. doi:[10.1007/978-3-642-12821-9_4](https://doi.org/10.1007/978-3-642-12821-9_4).
- [7] S. Lack, [Composing PROPs.](#), *Theory and Applications of Categories* [electronic only] 13 (2004) 147–163.
URL <http://eudml.org/doc/124613>
- [8] D. Clarke, D. Costa, F. Arbab, Connector colouring I: Synchronisation and context dependency, *Science of Computer Programming* 66 (3) (2007) 205–225. doi:[10.1016/j.scico.2007.01.009](https://doi.org/10.1016/j.scico.2007.01.009).
- [9] P. M. Rondon, M. Kawaguchi, R. Jhala, [Liquid types](#), in: R. Gupta, S. P. Amarasinghe (Eds.), *Proceedings of the ACM SIGPLAN 2008 Conference*

- on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008, ACM, 2008, pp. 159–169. doi:[10.1145/1375581.1375602](https://doi.org/10.1145/1375581.1375602). URL <http://doi.acm.org/10.1145/1375581.1375602>
- [10] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering, Springer, 2005. doi:[10.1007/3-540-28901-1](https://doi.org/10.1007/3-540-28901-1).
 - [11] M. P. Jones, A theory of qualified types, Science of Computer Programming 22 (3) (1994) 231 – 256. doi:[10.1016/0167-6423\(94\)00005-0](https://doi.org/10.1016/0167-6423(94)00005-0).
 - [12] B. C. Pierce, Types and Programming Languages, MIT Press, 2002.
 - [13] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580. doi:[10.1109/5.24143](https://doi.org/10.1109/5.24143).
 - [14] R. Muschevici, J. Proença, D. Clarke, Feature Nets: behavioural modelling of software product lines, Software and Systems Modeling (2015) 1–26. doi:[10.1007/s10270-015-0475-z](https://doi.org/10.1007/s10270-015-0475-z).
 - [15] S. Bliudze, J. Sifakis, Synthesizing Glue Operators from Glue Constraints for the Construction of Component-Based Systems, in: S. Apel, E. Jackson (Eds.), Software Composition, LNCS, Springer, Berlin / Heidelberg, 2011, pp. 51–67. doi:[10.1007/978-3-642-22045-6_4](https://doi.org/10.1007/978-3-642-22045-6_4).
 - [16] P. Sobocinski, Representations of Petri net interactions, in: P. Gastin, F. Laroussinie (Eds.), CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings, Vol. 6269 of Lecture Notes in Computer Science, Springer, 2010, pp. 554–568. doi:[10.1007/978-3-642-15375-4_38](https://doi.org/10.1007/978-3-642-15375-4_38).
 - [17] R. Bruni, H. C. Melgratti, U. Montanari, Connector algebras, Petri Nets, and BIP, in: E. M. Clarke, I. Virbitskaite, A. Voronkov (Eds.), Ershov Memorial Conference, Vol. 7162 of Lecture Notes in Computer Science, Springer, 2011, pp. 19–38. doi:[10.1007/978-3-642-29709-0_2](https://doi.org/10.1007/978-3-642-29709-0_2).
 - [18] D. Batory, Feature models, grammars, and propositional formulas, in: Proceedings of the 9th International Conference on Software Product Lines, SPLC’05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 7–20. doi:[10.1007/11554844_3](https://doi.org/10.1007/11554844_3).
 - [19] A. Mavridou, E. Baranov, S. Bliudze, J. Sifakis, Configuration logics: Modelling architecture styles, in: C. Braga, P. C. Ölveczky (Eds.), Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers, Vol. 9539 of Lecture Notes in Computer Science, Springer, 2015, pp. 256–274. doi:[10.1007/978-3-319-28934-2_14](https://doi.org/10.1007/978-3-319-28934-2_14).
 - [20] F. Bonchi, P. Sobocinski, F. Zanasi, Full abstraction for signal flow graphs, in: Proceedings of the 42nd Annual Symposium on Principles of Programming Languages, POPL ’15, ACM, New York, NY, USA, 2015, pp. 515–526. doi:[10.1145/2676726.2676993](https://doi.org/10.1145/2676726.2676993).

- [21] J. Gibbons, A pointless derivation of radix sort, *Journal of Functional Programming* 9 (3) (1999) 339–346. doi:[10.1017/s0956796899003354](https://doi.org/10.1017/s0956796899003354).
- [22] C. Kastner, S. Apel, Type-checking software product lines - a formal approach, in: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08*, IEEE Computer Society, Washington, DC, USA, 2008, pp. 258–267. doi:[10.1109/ASE.2008.36](https://doi.org/10.1109/ASE.2008.36).
- [23] S. Thaker, D. Batory, D. Kitchin, W. Cook, Safe composition of product lines, in: *Proceedings of the 6th International Conference on Generative Programming and Component Engineering, GPCE '07*, ACM, 2007, pp. 95–104. doi:[10.1145/1289971.1289989](https://doi.org/10.1145/1289971.1289989).
- [24] H. Post, C. Sinz, Configuration lifting: Verification meets software configuration, in: *Proceedings of the 2008 23rd International Conference on Automated Software Engineering, ASE '08*, IEEE Computer Society, 2008, pp. 347–350. doi:[10.1109/ASE.2008.45](https://doi.org/10.1109/ASE.2008.45).
- [25] S. Apel, H. Speidel, P. Wendler, A. von Rhein, D. Beyer, Detection of feature interactions using feature-aware verification, in: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11*, IEEE Computer Society, Washington, DC, USA, 2011, pp. 372–375. doi:[10.1109/ASE.2011.6100075](https://doi.org/10.1109/ASE.2011.6100075).
- [26] T. Thüm, I. Schaefer, S. Apel, M. Hentschel, Family-based deductive verification of software product lines, in: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering, GPCE '12*, ACM, New York, NY, USA, 2012, pp. 11–20. doi:[10.1145/2371401.2371404](https://doi.org/10.1145/2371401.2371404).
- [27] A. S. Dimovski, C. Brabrand, A. Wasowski, Variability abstractions: Trading precision for speed in family-based analyses, in: J. T. Boyland (Ed.), *29th European Conference on Object-Oriented Programming, ECOOP 2015*, July 5-10, 2015, Prague, Czech Republic, Vol. 37 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 247–270. doi:[10.4230/LIPIcs.ECOOP.2015.247](https://doi.org/10.4230/LIPIcs.ECOOP.2015.247).