Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# 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

ARTICLE INFO

*Article history:*
Received 9 April 2016
Received in revised form 9 February 2017
Accepted 1 March 2017
Available online xxxx

*Keywords:*
Calculus of connectors
Variability in connectors
Composition of families
Type system
Tile Model

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

Consider, for example, the simple component composition on Fig. 1. In our calculus we can write this composition as $\mathsf{sensor} \otimes \mathsf{clock}\,;\, \mathsf{worker}$, 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.

* Corresponding author.
*E-mail addresses:* jose.proenca@di.uminho.pt (J. Proença), dave.clarke@it.uu.se (D. Clarke).

http://dx.doi.org/10.1016/j.scico.2017.03.002
0167-6423/

Please cite this article in press as: J. Proença, D. Clarke, Typed connector families and their semantics, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.03.002

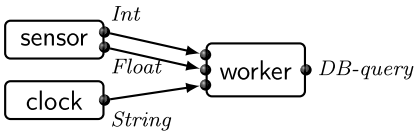


**Fig. 1.** Composition of components: a sensor produces a pair of values, a clock produces a time stamp, and the worker produces a database query to store the sensor values.

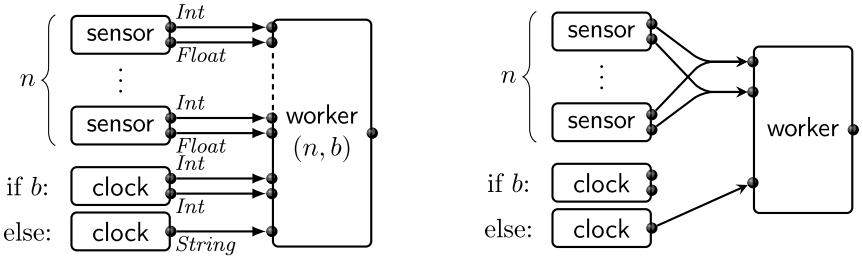


**Fig. 2.** Composition of families of components: combinations of sensors and clocks are composed with a parameterised worker (left) and with a simpler worker after some extra coordination (right).

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 Fig. 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 Fig. 2) can be written in our calculus as $\lambda n : \mathbb{N}, b : \mathbb{B} \cdot (\mathsf{sensor}^n \otimes \mathsf{clock}(b)\,;\, \mathsf{worker}(n, b))$, 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 $(\lambda n : \mathbb{N} \cdot \mathsf{sensor}^n) \otimes (\lambda b : \mathbb{B} \cdot \mathsf{clock}(b))\,;\, (\lambda n' : \mathbb{N}, b' : \mathbb{B} \cdot \mathsf{worker}(n', b'))$, 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 Fig. 2) can be written in our calculus as $\lambda n : \mathbb{N}, b : \mathbb{B} \cdot (\mathsf{sensor}^n \otimes \mathsf{clock}(b)\,;\, (\mathit{unzip}(n); (\nabla_n \otimes \nabla_n)) \otimes \mathsf{id}_{\mathit{String}}\,;\, \mathsf{worker})$, where $\mathit{unzip}(n)$ is a connector that swaps the orders of $2 * n$ connections (that will be formally specified later in this paper) and $\nabla_n$ 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 $b = \mathit{false}$ in order to have a *String* outgoing interface, so it can be composed with $\mathsf{id}_{\mathit{String}}$. This constraint is explicit in our types; more specifically, the type of this connector is $\forall x : \mathbb{N}, b : \mathbb{B} \cdot (0 \to \mathit{DB\text{-}query})\,|_{b=\mathit{false}}$, indicating that it has no inputs, it has a single output with type *DB-query*, and it is restricted by the constraint $b = \mathit{false}$.

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).

Please cite this article in press as: J. Proença, D. Clarke, Typed connector families and their semantics, Sci. Comput. Program. (2017), http://dx.doi.org/10.1016/j.scico.2017.03.002