Journal of Logical and Algebraic Methods in Programming 92 (2017) 19–44

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Functional BIP: Embedding connectors in functional programming languages

Romain Edelmann[a], Simon Bliudze[a,*], Joseph Sifakis[b]

[a] École polytechnique fédérale de Lausanne, Station 14, CH-1015 Lausanne, Switzerland
[b] Verimag, Université Grenoble Alpes, 700, avenue centrale, 38401 Saint Martin d'Hères, France

ARTICLE INFO

Article history:
Received 20 December 2016
Accepted 23 June 2017
Available online 12 August 2017

Keywords:
BIP
Connectors
Dynamicity
Functional programming
Haskell
Scala

ABSTRACT

This paper presents a theoretical foundation for functional language implementations of Behaviour–Interaction–Priority (BIP). We introduce a set of connector combinators describing synchronisation, data transfer, priorities and dynamicity in a principled way. A static type system ensures the soundness of connector semantics.

Based on this foundation, we implemented BIP as an embedded domain specific language (DSL) in Haskell and Scala. The DSL embedding allows programmers to benefit from the full expressive power of high-level languages. The clear separation of behaviour and coordination inherited from BIP leads to systems that are arguably simpler to maintain and reason about, compared to other approaches.

1. Introduction

When building large concurrent systems, one of the key difficulties lies in coordinating component behaviour and, in particular, management of the access to shared resources of the execution platform. Our approach relies on the BIP framework [1] for component-based design of correct-by-construction applications. BIP provides a simple, but powerful mechanism for the coordination of concurrent components by superposing three layers: Behaviour, Interaction, and Priority. First, component *behaviour* is described by Labelled Transition Systems (LTS) having transitions labelled with *ports* and extended with data stored in local variables. Ports form the interface of a component and are used to define its interactions with other components. They can also export part of the local variables, allowing access to the component's data. Second, *interaction models*, i.e. sets of interactions, define the component coordination. Interactions are sets of ports that define allowed synchronisations between components. Interaction models are specified in a structured manner by using connectors [2]. Third, *priorities* are used to impose scheduling constraints and to resolve conflicts when multiple interactions are enabled simultaneously. Interaction and Priority layers are collectively called *Glue*.

The strict separation between behaviour—i.e. stateful components—and coordination—i.e. stateless connectors and priorities—allows the design of modular systems that are easy to understand, test and maintain. Hierarchical combination of interactions and priorities provides a very expressive coordination mechanism [3,4]. The BIP language has been implemented as a coordination language for C/C++ [1] and Java [5,6]. It is supported by a tool-set including translators from

* Corresponding author.
E-mail addresses: romain.edelmann@epfl.ch (R. Edelmann), simon.bliudze@epfl.ch (S. Bliudze), joseph.sifakis@univ-grenoble-alpes.fr (J. Sifakis).

http://dx.doi.org/10.1016/j.jlamp.2017.06.003
2352-2208/© 2017 Elsevier Inc. All rights reserved.

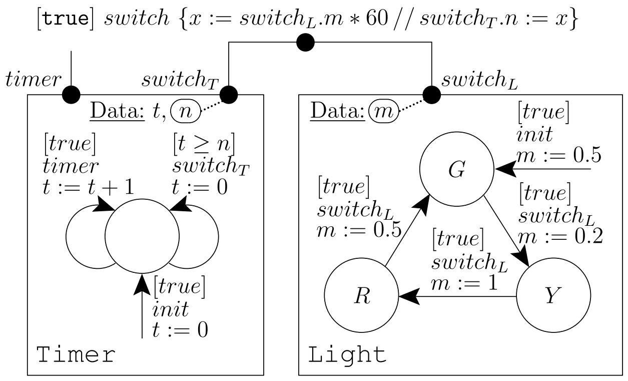


Fig. 1. Traffic light in BIP.

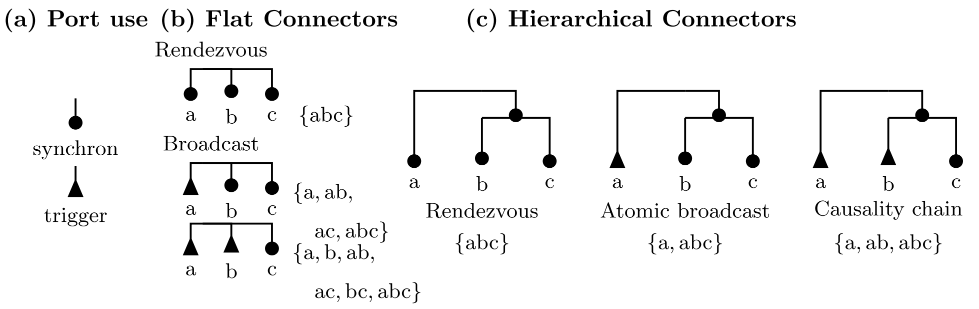


Fig. 2. Flat and hierarchical BIP connectors.

various programming models into BIP, source-to-source transformers, as well as a number of back-ends for the generation of code executable by dedicated engines.[1]

Atoms BIP systems are composed of atoms (atomic components) that have communication *ports* used for coordination. Atoms have disjoint state spaces; their behaviour is specified as a system of transitions labelled with ports. Fig. 1 shows a simple traffic light controller system modelled in BIP. It is composed of two atomic components `Timer` and `Light`, modelling, respectively, a timer and the light-switching behaviour. The `Timer` atom has one state with two self-loop transitions. The incoming arrow, labelled *init*, denotes the initialisation event. It is guarded by the constant predicate `true` and has an associated update function $t := 0$, which initialises the internal data variable t, used to keep track of the time spent since the last change of colour. This component also has a data variable n used in the guard $[t \geq n]$ of the transition labelled by the port $switch_T$ to determine when this transition can be fired. The variable n is exported through the port $switch_T$, which allows its value to be updated upon synchronisation with the other atomic component. The `Light` atom determines the colour of the traffic light and the duration (in minutes) that the light must stay in one of the three states, corresponding to the three colours.

Interactions Interactions between components are defined by hierarchically structured connectors [2,7].[2] The system in Fig. 1 has two connectors: a singleton connector with one port *timer* and no data transfer and a binary connector, synchronising the ports $switch_T$ and $switch_L$ of the two components. The first, singleton connector is necessary, since, in BIP, only ports that belong to at least one connector can fire. The second connector has an exported (top) port, called *switch*, and an associated variable x used for the data transfer. The guard is the constant predicate `true`, the upward and downward data-flows are defined, respectively, by the assignments $x := switch_L.m * 60$ and $switch_T.n := x$. Thus, upon each synchronisation, `Light` informs `Timer` about the amount of time to spend in the next location, converting it from minutes to seconds.

In [2], we have introduced the Algebra of Connectors. Connectors define sets of interactions based on the synchronisation attributes of the connected ports, which may be either *trigger* or *synchron* (Fig. 2a). If all connected ports are synchrons, then synchronisation is by *rendezvous*, i.e. the defined interaction may be executed only if all the connected components

[1] http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html.

[2] Our presentation of connectors combines elements of the Algebra of Connectors [2] (trigger and synchron port typing, hierarchical composition) for structuring the interactions and of interaction expressions [7] for data manipulation aspects (upward and downward data-flows). The notions of *top* and *bottom* ports were introduced in [7] to formalise the principle of the classical BIP language implementation, whereby connectors can *export* interactions for use by higher-level connectors.

allow the transitions of those ports (Fig. 2b). If a connector has at least one trigger, the synchronisation is by *broadcast*,[3] i.e. the allowed interactions are all non-empty subsets of the connected ports comprising at least one of the trigger ports (Fig. 2b). More complex connectors can be built hierarchically (Fig. 2c).

We have shown [2,4] that such hierarchical connectors can express any coordination operator definable by a certain class of Structural Operational Semantics (SOS) [8] rules. While detailed discussion of SOS rules and the corresponding coordination operators is beyond the scope of this paper,[4] it should be noted that this class encompasses parallel composition operators used in classical concurrency models, such as CSS [9] and CSP [10]. In particular, for a given set of ports P, any set of interactions $\Gamma \in 2^P$ can be precisely represented by a structured connector.

In the textual notation used for the Algebra of Connectors in [2], triggers are marked by primes. Thus, the three connectors in Fig. 2b (from top to bottom) are written, respectively, as abc, $a'bc$ and $a'b'c$. In hierarchical connectors, square brackets are used to denote sub-connectors. Thus, the three connectors in Fig. 2c (from left to right) are written, respectively, as $a[bc]$, $a'[bc]$ and $a'[b'c]$.

Notice that, abc and $a[bc]$ are equivalent, i.e. define the same set of interactions (here the singleton set $\{abc\}$), whereas $a'bc$ and $a'[bc]$ are not equivalent, defining, respectively, the interaction sets $\{a, ab, ac, abc\}$ and $\{a, abc\}$. Notice, furthermore, that this equivalence relation is not a congruence. Indeed, although connectors abc and $a[bc]$ are equivalent, this is not the case for $abcd'$ and $a[bc]d'$, since the latter does not allow the interactions involving only one of the ports b and c, i.e. bd, cd, abd and acd. A sound and complete axiomatisation is provided in [2]. In particular, it is shown that, if two connectors γ_1 and γ_2 are *equivalent*, then $[\gamma_1]$ and $[\gamma_2]$ are *congruent*, i.e. either of them, used as a sub-connector of a larger one, can be substituted by the other.

The description of each (sub-)connector consists of three parts:

1. A Boolean guard determining the enabledness of each interaction allowed by the connector, depending on the values of the provided data: an interaction is only enabled if the data provided by the components satisfies the guard [7].
2. A control part specifying a relation between a set of bottom ports and a single top port. In hierarchical connectors, sub-connector top ports are used as higher-level bottom ports. Each port of an atom can be used in several connectors. However, a CONNECTOR can involve at most one port per atomic component. The same applies to top ports of connectors.
3. A data-flow part specifying the computation associated with each interaction. The computation can affect variables associated with the ports. It consists of an upstream computation followed by a downstream computation. The former is specified by a function that takes as arguments the values exported through the ports involved in the interaction. The computed value is exported through the top port. The downstream computation produces values associated with the synchronised ports from the value received at the top port. This allows bidirectional exchange of information upon synchronisations among components.

Priorities Finally, priorities are used to impose scheduling constraints and to resolve conflicts when multiple interactions are enabled simultaneously. Interaction and Priority layers are collectively called *Glue*.

For instance, notice that, when $t \geq n$, both transitions, *timer* and $switch_T$, of the `Timer` atom in Fig. 1 are enabled. Since all other guards in the system are constant predicates `true`, this means that both connectors can fire. Imposing the priority $timer < switch$ resolves this choice, so that switching is performed whenever possible. In general, it is not necessary to impose priorities in all conflict situations: according to the BIP semantics, one of the enabled maximal priority interactions is chosen non-deterministically [2].

Although, the theoretical presentation in [2] completely separates interactions and priorities, in all practical applications and BIP implementations, one always implicitly assumes *maximal progress*, which defines the priority $\alpha < \beta$ for two interactions allowed *by the same connector*, provided that the inclusion $\alpha \subset \beta$ holds on α and β considered as sets of ports. Thus, among the interactions defined by the connector $a'bc$ (Fig. 2b), the priority relations $a < ab < abc$ and $a < ac < abc$ are always enforced in all practical applications. Unless additional priorities are provided explicitly, interactions ab and ac are incomparable. Thus, when all three ports a, b and c are enabled, but the interaction abc is disabled because of a data guard, one of the interactions ab and ac will be chosen non-deterministically.

This paper adapts the BIP coordination mechanisms to the context of functional programming languages. This adaptation constitutes the core of the results obtained by Edelmann in his Master thesis [11], where further additional material can be found. The main contributions presented in the paper are the following:

- We present a set of combinators to build connectors which can describe synchronisation, data transfer, priorities and dynamicity. We introduce the formal semantics and typing rules of those combinators and present some of their algebraic properties.

[3] Although we use the term “broadcast” by analogy with message passing—trigger ports initialise interactions, whereas synchrons join if they are enabled—, connectors *synchronise* ports, i.e. no messages passing is involved.

[4] Detailed presentation can be found in [4].

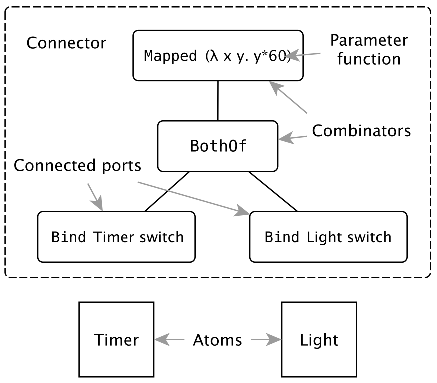


Fig. 3. A Functional BIP view of the *switch* connector from the example in Fig. 1.

- We present implementations of the concepts developed in this paper in two functional programming languages, Haskell [12] and Scala [13]. In each case, we show how the concepts can be transposed using well known idioms of the language. The two resulting frameworks are released under open source licenses and are freely available for download and use.[5]

The rest of this paper is structured as follows. Section 2 provides a brief introductory overview of Functional BIP. Section 3 introduces the typing and semantic framework for connector combinators. Section 4 introduces the core, priority, data and dynamic connector combinators. Section 5 discusses their algebraic properties. Section 6 describes an implementation of the presented concepts in Haskell and Scala. Section 7 discusses the related work. Section 8 concludes and summarises the paper.

2. Overview of Functional BIP

In Functional BIP, systems are composed of atoms, ports and a *single static* connector. As in the original BIP implementations, atoms have their own private memory, which is not shared with other atoms. Thus, atoms can communicate only by interacting through the ports of the system.

There are three main differences between Functional BIP and the original implementations. First of all, Functional BIP abandons code generation, introducing instead the primitives necessary to programmatically describe both the atomic behaviour and the glue. In particular, ports offer a very simple interface which consists of a single operation, called `await`. When invoking `await`, the atom sends a value to the port and blocks until the port sends a value back to the atom. Atoms may wait on multiple ports simultaneously. Ports are typed to describe the type of values that can be sent and received through them.

The second—most important—difference is that, in Functional BIP, ports are not directly associated with atoms. On the contrary, they are globally accessible entities used for communication and coordination. We introduce the construct `Bind` to attach a port to a specific atom. Such dissociation of ports from atoms allows us to introduce forms of dynamicity, which are not currently available in any of the original BIP implementations: some atoms are created at the system initiation, while others may be spawned by existing atoms at run time; we introduce the construct `Dynamic`, which binds all existing atoms, including those created at run time, to a given port.

Finally, as mentioned above, a Functional BIP system comprises only one connector, as opposed to a set of connectors in the original BIP approach. Notice, however, that this does not reduce the expressiveness of the BIP coordinating mechanism, since any number of connectors can be combined into a single one using the `OneOf` combinator (see Section 4.1.3). The connector of a system links atom–port pairs to define the possible interactions. The connector is unique and fixed throughout the system lifespan. Also note that connectors, contrary to atoms, are completely stateless in Functional BIP.

We introduce a number of combinators to describe connectors. Connector combinators combine connectors into larger connectors. They are used to hierarchically build complex connectors, starting from a set of basic ones [7]. Fig. 3 shows the Functional BIP representation of the *switch* connector from the example in Fig. 1. This is realised using two combinators: `BothOf` (see Section 4.1.4) and `Mapped` (see Section 4.3.1). The former synchronises the *switch* ports of the two atoms, whereas the latter is used to convert the data value sent by the first atom from minutes to seconds. Notice that, since, as discussed above, ports in Functional BIP are dissociated from atoms, we use only one port *switch* instead of the two ports $switch_T$ and $switch_L$ in the example in Fig. 1.

[5] Haskell version: https://github.com/redelmann/bip-in-haskell; Scala version: https://github.com/redelmann/bip-in-scala.

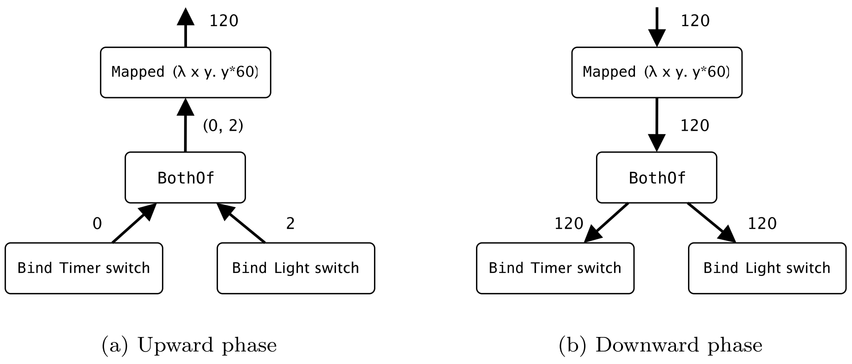


Fig. 4. Data transfer within connectors.

In Functional BIP, the connector is represented as a tree, with port–atom bindings at the leaves and combinators in the nodes. The arity of each node depends on the corresponding combinator.

Data transfer within the connector occurs in two phases: upward and downward. During the upward phase, the values sent through the ports by the atoms are collected and propagated up through the connector hierarchy. Once a value reaches the top of the hierarchy, it is transferred downward, back to the atoms involved in the interaction. Both up- and downward data transfers at each node of the tree are defined by the corresponding combinator. Fig. 4 illustrates the data transfer within the connector in Fig. 3, assuming that the atoms Timer and Light have sent, respectively, the values 2 and 0 on the port *switch*.

A connector ensures that atoms may only receive a value on the ports on which they are waiting. To maintain this invariant, we ensure that only the combinators which provided a value during the upward computation phase may receive a value during the downward phase. Some additional discussion about the implementation of Functional BIP is provided in Section 6.

The listings in Fig. 5 and Fig. 6 show Haskell and Scala implementations of the system presented in Fig. 1 using the Functional BIP framework. In order to define a Functional BIP system one would usually proceed in the following steps:

1. define the ports (lines 2–3 in Fig. 5 and 3–4 in Fig. 6);
2. define the behaviour of atoms to be executed in concurrent threads (lines 4–13, 14–22, 26–28 in Fig. 5 and 6–15, 17–21, 23–25 in Fig. 6);
3. create the atom instances (lines 24–26 in Fig. 5 and 27–29 in Fig. 6);
4. bind ports from these atom instances to those defined in step 1 and combine them using connector combinators (lines 30–34 in Fig. 5 and 31–35 in Fig. 6).

3. Semantic framework

Before we introduce the connector combinators, we introduce notation and concepts that will be used to describe their semantics and types.

3.1. Partial functions

We denote by $A \rightharpoonup B$ the set of partial functions from the set A to the set B. We will sometimes use the fact that partial functions can be represented as sets of pairs from $A \times B$, in particular when we will construct such partial functions. For instance, we will denote the empty partial function by $\emptyset$ and the singleton partial function that maps x to y by $\{x \mapsto y\}$.

3.2. Universe of values

We denote by $\mathcal{V}$ the set of values which can be handled and exchanged by the atoms. We assume that this set contains at least the Boolean values `true` and `false`, integers, all pairs of values and all total functions from values to values. Connectors are also part of the universe of values.

3.3. Types

The values are equipped with a polymorphic type system, similar to Haskell's type system. We denote by $\mathcal{T}$ the set of all types and by $v : t$ (respectively $x : t$) the predicate "value (respectively variable) $v \in \mathcal{V}$ has type $t \in \mathcal{T}$". For each type $t \in T$, we denote by $sem(t) \subseteq \mathcal{V}$ its *semantic domain*, i.e. the set of values ranged over by variables $x : t$. Booleans are given the type `bool`, integers the type `int` while pairs of values of type a and b are given the type $a \times b$. Total functions from values

```haskell
runSystem $ do
  switch <- newPort  -- Port with upward and downward value of type Double.
  tick <- newPort    -- Port with upward and downward value of type Unit.
  let behaviourTimer t n = do  -- Behaviour of a timer.
    let transitions =
      -- Send () on port tick, wrap received value in a Left.
      [ fmap Left $ onPort tick () ] ++
      -- Send t on port switch, wrap received value in a Right.
      [ fmap Right $ onPort switch t | t >= n && n >= 0.0 ]
    x <- awaitAny transitions  -- Wait to receive value x.
    case x of
      Left _ -> behaviourTimer (t + 1.0) n
      Right n' -> behaviourTimer 0.0 n'
  let behaviourLight = do  -- Behaviour of a light.
    -- In the green state.
    await switch 0.5  -- Send the value 0.5 on the port switch.
    -- In the yellow state.
    await switch 0.2
    -- In the red state.
    await switch 1.0
    -- Loop.
    behaviourLight
  -- Creation of the atoms.
  light <- newAtom behaviourLight  -- Creates an atom for the light.
  timer <- newAtom $ behaviourTimer 0.0 (-1.0)  -- Creates an atom for the time.
  clock <- newAtom $ forever $ do  -- Creates an atom for the clock.
    threadDelay 1000000 -- Sleep at least one second.
    await tick ()  -- Send the value () on the port tick.
  -- Definition of the system's connector.
  registerConnector $ anyOf
    -- Either timer and light sync on 'switch',
    [ (\ x y -> y * 60.0) <$> bind timer switch <*> bind light switch,
    -- Or timer and clock sync on 'tick'.
    bind timer tick <> bind clock tick ]
```

Fig. 5. Implementation in Haskell of the example from Fig. 3.

of type a to values of type b are given the type $a \to b$. Depending on the context, we will use both the "type" notation, $f : t_1 \to t_2$, and the "set" notation, $f : sem(t_1) \to sem(t_2)$. The main difference is that, without the function name, $t_1 \to t_2$ denotes the type, whereas $sem(t_1) \to sem(t_2)$ denotes the set of functions of this type, i.e. $sem(t_1 \to t_2) = sem(t_1) \to sem(t_2)$.

Connectors propagating values of type u during the upward phase and receiving values of type d during the downward phase have the type $u \uparrow\downarrow d$. The type system dictates how the connector combinators can be used. A connector of type $u \uparrow\downarrow d$ can only send values of type u during the upward phases, and expect values of type d during the downward phases.

3.4. Atoms

We denote by $\mathcal{A}$ the set of atom identifiers of a system.

```
val system = new System
// Ports of the system.
val switch = system.newPort[Double, Double]
val tick = system.newPort[Unit, Unit]

def behaviourTimer(t: Double, n: Double) {  // Behaviour of a timer.
  if (t >= n && n >= 0.0) {
    // Send () on the port tick, and t on the port switch. Wait for an answer on any of the two.
    awaitAny(tick.withValue(()).map(Left(_)), switch.withValue(t).map(Right(_))) {
      case Left(_) => behaviourTimer(t + 1.0, n)  // In case of tick.
      case Right(newN) => behaviourTimer(0.0, newN)  // In case of switch.
    }
  } else {
  // Send () on the port tick and wait.
    await(tick) { case _ => behaviourTimer(t + 1.0, n) }}}

def behaviourLight() {
  await(switch, 0.5) {  // Send 0.5 on the port switch.
    case _ => await(switch, 0.2) {
      case _ => await(switch. 1.0) {
        case _ => behaviourLight() }}}}

def behaviourClock() {
  Thread.sleep(1000)
  await(tick) { case _ => behaviourClock() }}

val light = system.newAtom { behaviourLight() }  // Creates the light atom.
val timer = system.newAtom { behaviourTimer() }  // Creates the timer atom.
val clock = system.newAtom { behaviourClock() }  // Creates the clock atom.

system.registerCo3nector {  // Sets the co3nector of the system.
  // Either timer and light sync on ‘switch‘
  anyOf(timer.bind(switch).and(light.bind(switch)).map { case (x, y) -> y * 60.0 },
    // Or timer and clock sync on ‘tick‘.
    timer.bind(tick).and(clock.bind(tick)).sending(()))}
```

Fig. 6. Implementation in Scala of the example from Fig. 3.

3.5. Ports

We denote by P the set of ports of a system. Each port $p \in P$ is associated with two types: p^{up} and p^{down}. Atoms can only send values of type p^{up} through the port p and are guaranteed that the value eventually received from the port, if any, has type p^{down}.

3.6. Connectors

We denote by C the set of connectors c for which there exists at least two types u and d such that $c : u \uparrow\downarrow d$. A connector of type $u \uparrow\downarrow d$ sends values of type u during the upward phases and expects values of type d during the downward phases.

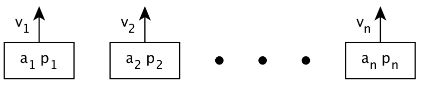


Fig. 7. System state.

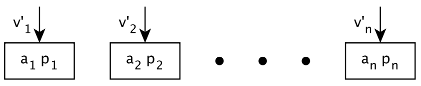


Fig. 8. Assignment.

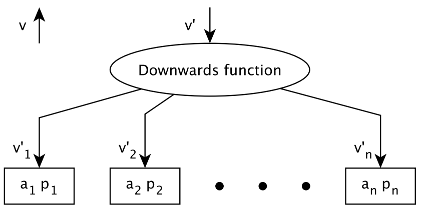


Fig. 9. Open interaction.

3.7. System states

A system state is as partial function, mapping atom–port pairs to corresponding values sent by the atom through the port. The set of states is:

$$\mathcal{S} = \{ f \in (\mathcal{A} \times P) \rightharpoonup \mathcal{V} \mid \forall \big((a, p) \mapsto x\big) \in f\ ,\ x : p^{up} \}\,. \tag{1}$$

In a given system state, all atoms are waiting on a (possibly empty) subset of ports. When a port is active for an atom, a value of the appropriate type has been sent through it. In the above definition, the predicate of the set comprehension ensures that only the states in which the values sent are of the appropriate type are considered. Fig. 7 shows a system state where atoms a_i have sent values v_i on ports p_i.

3.8. Assignments

Similarly, an assignment is a partial function, mapping atom–port pairs to corresponding values received by the atom on the port. The set of assignments is:

$$\mathcal{R} = \{ f \in (\mathcal{A} \times P) \rightharpoonup \mathcal{V} \mid \forall \big((a, p) \mapsto x\big) \in f\ ,\ x : p^{down} \}\,.$$

An assignment maps each atom–port pair to at most one value. The value assigned, if any, is the value received by the atom on the given port. Fig. 8 represents an assignment where atoms a_i receive values v_i on the ports p_i.

3.9. Open interactions

We denote by $\mathcal{O} = \mathcal{V} \times \bigcup_{u \in \mathcal{T}} (sem(u) \to \mathcal{R})$ the set of open interactions. An open interaction consists of an *upward value* in $\mathcal{V}$ and a *downward function* in $sem(u) \to \mathcal{R}$ that, given a *downward value* of some type u returns an assignment. Intuitively, open interactions describe the value that flows out of the connector (upward direction), and, given the value that flows into the connector (downward direction), describe what values, if any, are assigned to the ports of atoms. Those interactions are called *open* as the values exchanged are exposed. Fig. 9 shows the schematic representation of an open interaction.

When the upward value is in the domain of the downward function, the open interaction can be closed to obtain a valid assignment. In this case, the upward value is used as a downward value:

$$\begin{aligned} &close : \mathcal{O} \rightharpoonup \mathcal{R} \\ &close((v, f)) = f(v),\ \text{if } v \in domain(f)\,. \end{aligned} \tag{2}$$

3.10. Downward compatibility

We consider two assignments $R_1, R_2 \in \mathcal{R}$ to be *downward compatible* if and only if:

$$\{a \mid \exists p \in P.(a, p) \in domain(R_1)\} \cap \{a \mid \exists p \in P.(a, p) \in domain(R_2)\} = \emptyset\,.$$

Intuitively, two assignments are downward compatible if they involve distinct sets of atoms. We extend this notion to downward functions. Downward functions f_1, f_2 are downward compatible if and only if they produce downward compatible assignments for all possible downward values. Notice that, if two assignments $R_1, R_2 \in \mathcal{R}$ are downward compatible, then $R_1 \cup R_2$ is also an assignment. Indeed, downward compatibility ensures that two interactions can be fused [2] without violating the requirement that atoms are only involved once in an interaction. For two downward compatible functions, $f_1, f_2 : sem(u) \to \mathcal{R}$, we will denote $f_1 \cup f_2 : sem(u) \to \mathcal{R}$ the downward function defined by $(f_1 \cup f_2)(v) \overset{def}{=} f_1(v) \cup f_2(v)$.

4. Connector combinators

Using the semantic framework introduced in Section 3, we can now introduce the set of combinators used to describe connectors. For the sake of clarity, we will introduce combinators progressively, in related groups. First, we introduce core combinators, which correspond to the Algebra of Interactions [2]. Then, we introduce combinators describing priorities, data manipulation and dynamicity.

For each connector combinator, we will provide the type inference rule defining the type of the resulting connector from the types of the parameters and children connectors. We also provide formal semantics of each combinator. To this end, we define the following semantic function:

$$[\cdot] : \mathcal{C} \to \mathcal{S} \to 2^{\mathcal{O}}$$

This semantic function gives, for each connector and system state, the set of possible open interactions. Intuitively, this function will define, for each connector and system state, what are the possible values sent out of the connector during the upward phase and what are the corresponding assignments of values to the different atoms and ports for a given downward value. We will define this function progressively as we encounter the different set of connector combinators. We will define it recursively for each of the possible combinators.

4.1. Core combinators

4.1.1. `Bind`

This combinator takes a port and an atom and binds them together. It is equivalent to the port of an atomic component in BIP. The type of the resulting connector depends on the send and receive types of the port:

$$\frac{}{(\texttt{Bind}\, a\ p) : p^{up} \uparrow\downarrow p^{down}} \tag{3}$$

The resulting connector provides a single interaction if the atom is currently waiting on the port and none otherwise. The value propagated upward by this connector is the value, if any, that was sent through the port by the atom. When this connector receives a value during the downward phase, it is transmitted to the atom via the given port. Expressed in terms of the open semantics function, the behaviour of the resulting connector is defined, for all $S \in \mathcal{S}$, by

$$[\texttt{Bind}\, a\ p](S) = \begin{cases} \{(S(a,p), \{v \mapsto \{(a,p) \mapsto v\} \mid v : p^{down}\})\}, & \text{if } (a,p) \in domain(S)\,, \\ \emptyset\,, & \text{otherwise.} \end{cases} \tag{4}$$

If the system state has an entry for the given atom–port pair, then a single open interaction is possible. The upward value of the open interaction is the value sent by the port, while the downward function propagates the downward value to the atom and port.

4.1.2. `Success` and `Failure`

These two combinators do not involve any atoms and ports. They do not take any connectors as parameters and thus can only be found at the leaves of the connector tree.

Regardless of the system state, `Success` v always provides a single open interaction, whose upward value is v. `Failure`, on the other hand, represents a connector that is never enabled. While by themselves, these combinators do not present any practical interest, they can be combined with others to build more useful connectors. They correspond to the elements 1 and 0 of the Algebra of Interactions [2]. The types of connectors defined by `Success` and `Failure` are given by the following inference rules:

$$\frac{v : u}{(\texttt{Success}\ v) : u \uparrow\downarrow d}\,, \qquad \frac{}{\texttt{Failure} : u \uparrow\downarrow d}\,. \tag{5}$$

Notice that `Success` is polymorphic in the downward type and `Failure` in both the upward and downward types. Let us denote $\texttt{Success}^{u\uparrow\downarrow d}\ v$, with $v : u$, the instantiation of `Success` v for the type $u \uparrow\downarrow d$. The open-interaction semantics of the two connectors is defined, for all $S \in \mathcal{S}$, by

$$[\texttt{Success}^{u\uparrow\downarrow d}\ v](S) = \big\{\big(v, \{w \mapsto \emptyset \mid w \in sem(d)\}\big)\big\}\,, \tag{6}$$

$$[\texttt{Failure}](S) = \emptyset\,. \tag{7}$$

4.1.3. `OneOf`

This combinator expresses the non-deterministic choice between two connectors. The connector `OneOf` c_1 c_2 behaves as either c_1 or c_2. This combinator corresponds to the union operation in the Algebra of Interactions [2]. The type and open interaction semantics of `OneOf` c_1 c_2 are defined as follows:

$$\frac{c_1 : u \uparrow\downarrow d \qquad c_2 : u \uparrow\downarrow d}{(\texttt{OneOf}\ c_1\ c_2) : u \uparrow\downarrow d}\,, \tag{8}$$

$$[\texttt{OneOf}\ c_1\ c_2](S) = [c_1](S) \cup [c_2](S)\,, \qquad \text{for all } S \in \mathcal{S}. \tag{9}$$

Thus, the interactions possible in this connector are all interactions that are possible in either c_1 or c_2.

4.1.4. `BothOf`

This last core combinator represents the fusion of two connectors. It corresponds to the fusion operator in the Algebra of Interactions [2]. The corresponding type inference rule is

$$\frac{c_1 : u_1 \uparrow\downarrow d \qquad c_2 : u_2 \uparrow\downarrow d}{(\texttt{BothOf}\ c_1\ c_2) : (u_1 \times u_2) \uparrow\downarrow d}\,. \tag{10}$$

Thus, `BothOf` c_1 c_2 propagates upward the pair of upward values coming from c_1 and c_2. In the presence of non-determinism, this connector returns all possible combinations of interactions from c_1 and c_2, i.e., for all $S \in \mathcal{S}$, we let

$$[\texttt{BothOf}\ c_1\ c_2](S) = \Big\{ \big((x_1, x_2), f_1 \cup f_2\big) \,\Big|\, (x_1, f_1) \in [c_1](S)\,, (x_2, f_2) \in [c_2](S) \text{ and } f_1 \text{ and } f_2 \text{ are downward compatible} \Big\}. \tag{11}$$

Interactions that would lead to atoms receiving more than one value during the downward phase are filtered out by the downward compatibility check.

4.1.5. Examples

Example 4.1. The `Optional` combinator, which takes as parameters a value v and a connector c, and returns a connector that provides any interaction provided by c, plus an extra interaction whose upward value is v, is derived as follows:

$$\texttt{Optional}\ v\ c = \texttt{OneOf}\ c\ (\texttt{Success}\ v)\,.$$

Note that the extra interaction introduced by this combinator does not involve any atoms.

Example 4.2. The `AnyOf` and `AllOf` combinators, which generalise, respectively, the `OneOf` and the `BothOf` combinators to an arbitrary number of underlying connectors, are recursively built as follows:

$$\texttt{AnyOf} <> = \texttt{Failure}$$
$$\texttt{AnyOf} <c_1, \ldots> = \texttt{OneOf}\ c_1\ (\texttt{AnyOf} <\ldots>)$$

and

$$\texttt{AllOf} <> = \texttt{Success} <>$$
$$\texttt{AllOf} <c_1, \ldots> = \texttt{BothOf}\ c_1\ (\texttt{AllOf} <\ldots>)\,,$$

where <> denotes the empty sequence.

The `AllOf` combinator allows us to define an arbitrary rendezvous connector as shown in Fig. 2b. The following example shows how to define broadcast connectors.

Example 4.3. Recall the textual notation of the Algebra of Connectors (Section 1) and consider a connector $[t_1]' \ldots [t_n]' [s_1] \ldots [s_m]$ with n triggers and m synchrons. It is easy to see [2] that this connector can be equivalently rewritten as $\big[[t_1]' \ldots [t_n]'\big]' \big[[s_1]' \ldots [s_m]'\big]$.

We define the following derived combinators:

$$\texttt{Trigger}\ t\ s = \texttt{OneOf}\ (\texttt{BothOf}\ t\ s)\ t\,, \tag{12}$$

$$\texttt{ManyOf}\ c_1\ c_2 = \texttt{OneOf}\ (\texttt{BothOf}\ c_1\ c_2)\ (\texttt{OneOf}\ c_1\ c_2)\,, \tag{13}$$

$$\texttt{ManyOf}\ c_1\ c_2 < \cdots > = \texttt{ManyOf}\ c_1\ (\texttt{ManyOf}\ c_2\ < \cdots >)\,, \tag{14}$$

which allow us to encode arbitrary broadcast connectors $[t_1]' \ldots [t_n]' [s_1] \ldots [s_m]$ as follows:

$$\texttt{Broadcast}\ (t_1 \ldots t_n)\ (s_1 \ldots s_m) = \texttt{Trigger}\ (\texttt{ManyOf}\ t_1\ \ldots\ t_n)\ (\texttt{ManyOf}\ s_1\ \ldots\ s_m)\,. \tag{15}$$

4.2. Priority combinators

In this sub-section, we introduce the combinators used to specify priorities. Priorities are used to inhibit the execution of certain interactions when interactions of higher priority are possible. Contrary to the classical BIP syntax, where priorities are defined by a separate syntactic construction and can be applied only at the top level of a connector within a compound component, priority combinators can be applied at any level in the connector hierarchy.

4.2.1. `FirstOf`

This combinator imposes fixed-order priority among the sub-connectors. In the connector `FirstOf` c_1 c_2, interactions from c_1 will be preferred over interactions from c_2, whenever the former are available. The type and semantics of the resulting connector are defined as follows

$$\frac{c_1 : u \uparrow\downarrow d \qquad c_2 : u \uparrow\downarrow d}{(\texttt{FirstOf}\, c_1\, c_2) : u \uparrow\downarrow d}, \tag{16}$$

and, for all $S \in \mathcal{S}$,

$$[\texttt{FirstOf}\, c_1\, c_2](S) = \begin{cases} [c_1](S)\,, & \text{if } [c_1](S) \neq \emptyset\,, \\ [c_2](S)\,, & \text{otherwise}\,. \end{cases} \tag{17}$$

4.2.2. `Maximal`

Given a partial ordering on the upward data domain encoded by a predicate g, the connector `Maximal` g c returns all interactions from the connector c whose upward values are maximal. The parameter function g should return `true` when its first parameter is strictly less than its second parameter, and `false` otherwise. The type of the resulting connector is defined by the inference rule

$$\frac{g : (u \times u) \to \texttt{bool} \qquad c : u \uparrow\downarrow d}{(\texttt{Maximal}\, g\, c) : u \uparrow\downarrow d}, \tag{18}$$

whereas its semantics is defined by letting, for all $S \in \mathcal{S}$,

$$[\texttt{Maximal}\, g\, c](S) = \left\{ (x, f) \in [c](S) \,\middle|\, \nexists (x_1, f_1) \in [c](S).g(x, x_1) = \texttt{true} \right\}. \tag{19}$$

4.2.3. Examples

Example 4.4. Assume a system composed of two agents A_1 and A_2 with behaviour modelled by the automaton in Fig. 10a. The connector in Fig. 10b ensures that only one of A_1 and A_2 may be in the state *work* at the same time.

The use of `FirstOf` in this example ensures that the *finish* transitions are taken as soon as possible. If the `OneOf` combinator was used instead, the mutual exclusion property could be violated, as the connector would allow *begin* transitions to fire alone.

To see that more concretely, let us examine the different cases where the mutual exclusion property could potentially be violated:

1. Atoms A_1 and A_2 are both waiting on the port *begin*. In this case, the connector ensures that only one of the atoms may take the transition and enter the critical section. As `OneOf` may only select one of the underlying connector, both *begin* transitions cannot be part of the same interaction.
2. A_1 is waiting on the *finish* port and A_2 on the *begin* port. A problem would occur if A_2 entered the *work* state while A_1 stayed in the same state. The use of `FirstOf` within the connector ensures that A_1 takes a transition from the *work* state to the *idle* state, thereby avoiding the problematic situation.
3. The case when A_2 is waiting on the *finish* port and A_1 on the *begin* port is symmetrical to the previous case.

Thus, since in the initial state both atoms are idle, they cannot end up being in the *work* state simultaneously, i.e. the mutual exclusion property is, indeed, ensured.

Example 4.5. In Example 4.3, we have shown how to encode arbitrary broadcast connectors using the core combinators. In order to implement the maximal progress assumption, we replace two of the occurrences of `OneOf` in (12) and (13) by `FirstOf`:

$$\texttt{TriggerMP}\, t\, s = \texttt{FirstOf}\; (\texttt{BothOf}\; t\; s)\; t\,,$$

$$\texttt{ManyOfMP}\, c_1\, c_2 = \texttt{FirstOf}\; (\texttt{BothOf}\; c_1\; c_2)\; (\texttt{OneOf}\; c_1\; c_2)\,,$$

$$\texttt{ManyOfMP}\, c_1\, c_2 < \cdots > = \texttt{ManyOfMP}\, c_1\; (\texttt{ManyOfMP}\, c_2\; < \cdots >)\,.$$

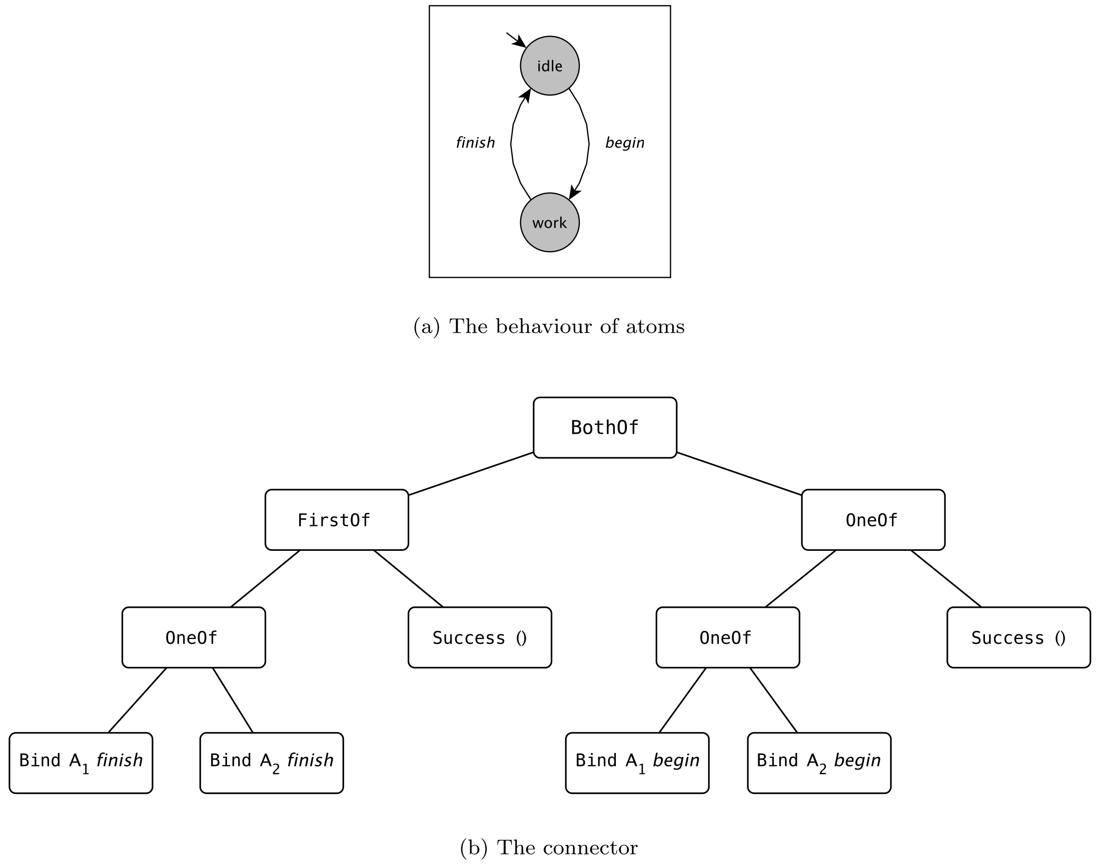


Fig. 10. Mutual exclusion using `FirstOf`.

The definition of the broadcast combinator stays the same:

$$\texttt{BroadcastMP}\,(t_1 \dots t_n)\,(s_1 \dots s_m) = \texttt{TriggerMP}\,(\texttt{ManyOfMP}\,t_1\ \dots\ t_n)\,(\texttt{ManyOfMP}\,s_1\ \dots\ s_m)\,.$$

4.3. Data combinators

We now introduce the combinators for data manipulation in the connectors.

4.3.1. `Mapped` and `ContraMapped`

These two combinators apply a parameter function to, respectively, the upward and downward values propagated by the underlying connector. The types of the resulting connectors are defined by the inference rules:

$$\frac{g : v \to u \qquad c : v \uparrow\downarrow d}{(\texttt{Mapped}\,g\,c) : u \uparrow\downarrow d}\,, \qquad \frac{g : d \to e \qquad c : u \uparrow\downarrow e}{(\texttt{ContraMapped}\,g\,c) : u \uparrow\downarrow d}\,. \tag{20}$$

The two combinators never modify the number of possible interactions. Their semantics is defined by letting, for all $S \in \mathcal{S}$,

$$[\texttt{Mapped}\,g\,c](S) = \left\{(g(x), f) \,\middle|\, (x, f) \in [c](S)\right\}, \tag{21}$$

$$[\texttt{ContraMapped}\,g\,c](S) = \left\{(x, f \circ g) \,\middle|\, (x, f) \in [c](S)\right\}. \tag{22}$$

4.3.2. `Guarded`

This combinator is used to filter out open interactions whose upward values fail to satisfy a predicate passed as the argument. The connector type is defined by the inference rule

$$\frac{g : u \to \texttt{bool} \qquad c : u \uparrow\downarrow d}{(\texttt{Guarded}\,g\,c) : u \uparrow\downarrow d}\,. \tag{23}$$

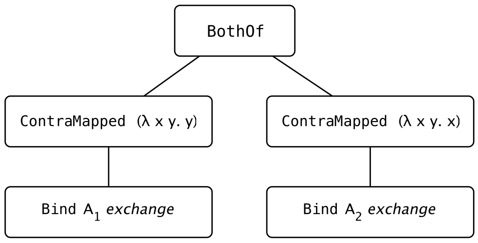


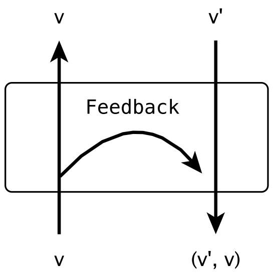
Fig. 11. Example use of `ContraMapped`.

This combinator allows restricting non-determinism, ensuring that upward values all satisfy a given predicate. Its semantics is defined by letting, for all $S \in \mathcal{S}$,

$$[\texttt{Guarded}\, g\, c](S) = \left\{(x, f) \in [c](S) \,\middle|\, g(x) = \texttt{true}\right\}. \tag{24}$$

4.3.3. `Feedback`

This last data manipulation combinator allows feeding the upward value back into the downward propagation phase. The typing inference rule and the semantics are defined by



$$\frac{c : u \uparrow\downarrow (d \times u)}{(\texttt{Feedback}\, c) : u \uparrow\downarrow d}, \tag{25}$$

$$[\texttt{Feedback}\, c](S) = \left\{(x, f \circ tag_x) \,\middle|\, (x, f) \in [c](S)\right\}, \text{ for all } S \in \mathcal{S}, \tag{26}$$

where $tag_x(y) \overset{def}{=} (y, x)$ (see the figure on the left).

While this combinator does not strictly augment the expressiveness of the language, it is useful to build connectors in a more modular fashion. In ensures that the upward value is always known in the downward phase regardless of the context in which the connector is used.

4.3.4. Examples

Example 4.6. Consider the connector in Fig. 11. This connector synchronises two atoms, A_1 and A_2, on the *exchange* port. The value sent back to each atom is the value that was sent by the other atom.

Example 4.7. The ability to manipulate data allows us to encode `FirstOf` by combining `Mapped`, `OneOf` and `Maximal`:

$$\texttt{FirstOf}\ c_1\ c_2 = \texttt{Mapped}\ untag\ \left(\texttt{Maximal}\ cmp_2\ \left(\texttt{OneOf}\ (\texttt{Mapped}\ tag_2\ c_1)\ (\texttt{Mapped}\ tag_1\ c_2)\right)\right),$$

where the function *untag*, cmp_2 and tag_n are defined as by putting

$$\begin{aligned} untag(x, n) &= x, \\ cmp_2((x, n), (y, m)) &= n < m, \\ tag_n(x) &= (x, n). \end{aligned}$$

Example 4.8. In Examples 4.3 and 4.5, we have shown how one can encode arbitrary broadcast connectors with (Example 4.3) and without (Example 4.5) the maximal progress assumption. However, both of these encodings potentially incur a huge overhead at run time, due to the duplication of operand connectors in (12) and (13). We now provide an alternative, efficient encoding of broadcast without maximal progress. To this end, we define the following derived combinators (denoting by *cs*, *ts* and *ss*, respectively, the lists of connectors, triggers and synchrons):

$$\begin{aligned} \texttt{OptionalD}\, c &= \texttt{OneOf}\ (\texttt{Mapped}\, singleton\, c)\ (\texttt{Success}\ <>), \\ \texttt{ManyOfD}\, cs &= \texttt{Mapped}\, flatten\ (\texttt{AllOf}\ < \texttt{OptionalD}\, c \,|\, c \in cs >) \end{aligned}$$

and

$$\begin{aligned}\texttt{BroadcastD}\,ts\;ss = \texttt{Mapped}\,\mathit{flatten}\,\big(&\texttt{BothOf}\\ &\big(\texttt{Guarded}\,\mathit{notEmpty}\;(\texttt{ManyOfD}\,ts)\big)\\ &\texttt{ManyOfD}\,ss\big)\,,\end{aligned}$$

where *singleton* wraps its parameter value into a singleton list, *flatten* transforms a structured tuple of lists into one flat list and *notEmpty* checks whether a list is empty or not. We omit the definitions of these functions.

Notice that this encoding does not use any priority combinators, relying on the guard to ensure that at least one trigger is present in every allowed interaction.

In order to obtain a similar encoding of a broadcast with maximal progress, one has to tag the lists produced by `ManyOfD` with their lengths and replace `Guarded` *notEmpty* by `Maximal` cmp_2 similarly to the previous example.

4.4. Dynamic combinators

Finally, we introduce a set of combinators that allow dynamicity, i.e. creation and deletion of atoms at run time and dynamic reconfiguration of the set of possible interactions among the atoms. Since the connector of a system is set at system initialisation time, it is not possible to explicitly refer to atoms which have not been created yet, and therefore it is not possible to make those atoms participate in any interactions described by the connector. The combinators we introduce in this section alleviate this problem.

4.4.1. `Dynamic`

Given a port p, the `Dynamic` combinator, binds all existing atoms to the port p. Atoms that are created at run time are also bound to the port. This combinator can be thought of as `Bind` with an atom chosen non-deterministically. Hence, the corresponding type inference rule is

$$\frac{}{(\texttt{Dynamic}\;p) : p^{up} \uparrow\downarrow p^{down}} \tag{27}$$

and the semantics is defined by letting, for all $S \in \mathcal{S}$,

$$[\texttt{Dynamic}\;p](S) = \Big\{ \Big(S(a,p), \big\{v \mapsto \{(a,p) \mapsto v\} \,\big|\, v : p^{down}\big\}\Big) \,\Big|\, (a,p) \in \mathit{domain}(S) \Big\}\,. \tag{28}$$

Note that the `Dynamic` combinator does not impose any restriction on the atom involved. It is however possible to constrain its choices using `Guarded` above it in the connector hierarchy.

4.4.2. `Joined`

This last combinator that we introduce allows dynamic connector reconfiguration by accepting connectors as values. The connector `Joined` c acts as a placeholder for connectors passed as upward values by the connector c. The type of this connector is thus given by the inference rule

$$\frac{c : (u \uparrow\downarrow d) \uparrow\downarrow d}{(\texttt{Joined}\,c) : u \uparrow\downarrow d}\,. \tag{29}$$

The connector is named after the natural transformation μ of monads, called *join* or *multiply*, in the context of category theory and `join` in Haskell. The corresponding semantic function is defined, for all $S \in \mathcal{S}$ by

$$[\texttt{Joined}\,c](S) = \Big\{ \big(x, f_1 \cup f_2\big) \,\Big|\, (c_1, f_1) \in [c](S),\, (x, f_2) \in [c_1](S) \text{ and } f_1 \text{ and } f_2 \text{ are downward compatible} \Big\}\,. \tag{30}$$

This connector is very expressive and can be used to derive some of the combinators that we have introduced previously, such as `Guarded` and `BothOf`.

4.4.3. Examples

Example 4.9. The `Joined` combinator allows us to redefine `Guarded` and `BothOf` as derived combinators:

$$\texttt{Guarded}\;p\;c = \texttt{Joined}\,\big(\texttt{Mapped}\,(\lambda x.\text{ if } p(x) \text{ then } \texttt{Success}\,x \text{ else } \texttt{Failure})\;c\big)\,,$$

$$\texttt{BothOf}\;c_1\;c_2 = \texttt{Joined}\,\big(\texttt{Mapped}\,\big(\lambda x_1.\,\texttt{Mapped}\,\big(\lambda x_2.(x_1,x_2)\big)\;c_2\big)\;c_1\big)\,.$$

In both cases, the basic idea is to build a connector from the upward value provided by the underlying connectors using `Mapped`, then using `Joined` on this connector.

As we will discuss later in Section 5, `Joined`, `Mapped`, `Success` and `Failure` form interesting algebraic structures, yielding many combinators such as `Guarded` and `BothOf`, in a completely generic fashion.

4.5. Closed semantic function

Based on the semantic function we have defined throughout this section, we now introduce the partial function $[\![\cdot]\!] : \mathcal{C} \rightharpoonup \mathcal{S} \to 2^{\mathcal{R}}$ that associates to a connector $c \in \mathcal{C}$ and a system state $S \in \mathcal{S}$, the set of possible assignments defined by the interactions allowed by c (see (2) for the definition of *close*):

$$[\![c]\!](S) \overset{def}{=} \{close(o) \mid o \in [c](S)\}\,, \text{ if } c : a \uparrow\downarrow a \text{ for some type } a\,.$$

This function gives to a connector its meaning as a function from system states to possible assignments.

The following two lemmata provide basic soundness results that we will use to prove the subsequent propositions that show that the closed semantics of any well-typed connector is well-defined and the resulting behaviour satisfies the BIP consistency constraints.

Lemma 4.10. *For any connector $c : u \uparrow\downarrow d$ and any system state $S \in \mathcal{S}$, we have $x \in sem(u)$, for all $(x, f) \in [c](S)$.*

Lemma 4.11. *For any connector $c : u \uparrow\downarrow d$ and any system state $S \in \mathcal{S}$, we have $domain(f) = sem(d)$, for all $(x, f) \in [c](S)$.*

The proofs of the lemmata are straightforward by structural induction and are relegated to the Appendix.

Proposition 4.12. *For any type u, the function $[\![\cdot]\!]$ is well-defined on connectors of type $u \uparrow\downarrow u$.*

Proof. For any connector $c \in \mathcal{C}$ and system state $S \in \mathcal{S}$, we have:

$$[\![c]\!](S) = \{close(o) \mid o \in [c](S)\} = \{f(x) \mid (x, f) \in [c](S)\}\,.$$

Consider some $(x, f) \in [c](S)$. By Lemma 4.10, we have $x \in sem(u)$, whereas, by Lemma 4.11, $sem(u) = domain(f)$. If follows immediately that $f(x)$ is well-defined. Hence $close(\cdot)$ and $[\![\cdot]\!]$ are also well-defined. □

Proposition 4.13. *For any type u, connector $c \in \mathcal{C}$ of type $u \uparrow\downarrow u$, system state $S \in \mathcal{S}$ and assignment $R \in [\![c]\!](S)$, holds the inclusion $domain(R) \subseteq domain(S)$.*

Proof sketch. This proposition is a straightforward consequence of the construction of the semantics function. It follows directly from the fact that new points can be added to the domain of an assignment only through the `Bind` and `Dynamic` combinators. Indeed, all other combinators either leave the assignments untouched or simply combine them. In the `Bind` and `Dynamic` combinators, a new atom–port pair is only added to the domain of the assignment if it belongs to the domain of the system state.

Thus, when an atom–port pair (a, p) is part of the domain of an assignment $R \in [\![c]\!](S)$, it must be, by construction, the case that $(a, p) \in domain(S)$. □

Proposition 4.13 means that assignments resulting from closing the connector only send values to atoms through the ports that these atoms are waiting on.

Proposition 4.14. *For any type u, connector $c \in \mathcal{C}$ of type $u \uparrow\downarrow u$ and system state $S \in \mathcal{S}$, all assignments in $[\![c]\!](S)$ involve each of the atoms at most once.*

Proof sketch. The proof is direct by structural induction on the connector combinators. The only two cases where special care must be taken, in order to ensure that the assignment does not involve an atom more than once, are `BothOf` and `Joined`. In those two cases, the downward compatibility check ensures that this property is not violated. □

Notice that the closed semantics of connectors allows simulating the `Feedback` combinator. This can be achieved by modifying all the combinators higher in the connector tree hierarchy to propagate the corresponding value upward then downward back to the connector, to which `Feedback` has to be applied. Therefore, the `Feedback` combinator is not, strictly speaking primitive. However, it greatly simplifies this construction, by allowing feeding values back locally.

4.6. Additional remarks about expressiveness

Connector combinators defined above are very expressive. Indeed, the `AllOf` (Example 4.2) and `Broadcast` (Example 4.3) combinators encode precisely the rendezvous and broadcast synchronisation discussed in the introduction and illustrated in Fig. 2. Since hierarchical composition is an intrinsic property of combinators, these are sufficient to directly encode any connector of the Algebra of Connectors [2]. The same approach used in Example 4.7 to encode `FirstOf` using the `Maximal` and `Mapped` combinators, can also be used to encode any priority relation.

Thus, connector combinators have at least the same expressiveness as the classical BIP glue. As discussed in the introduction, BIP connectors are sufficiently expressive to encode classical parallel composition operators, such as the ones used in CCS [9] and CSP [10]. The expressiveness of BIP priorities is discussed in detail in [3].

Furthermore, the way data is handled by connector combinators, i.e. the separation into the upward and downward data transfer phases, in combination with the data combinators (Section 4.3), precisely mimic interaction expressions as defined in [7], which, in their turn, correspond exactly to the implementation in the BIP tool-set.

Finally, it should be noted that the dynamic combinators `Dynamic` and `Joined` (Section 4.4) are novel. Introduced in [11], they allow dynamic creation and deletion of components—a functionality that is not available in the classical BIP implementation and goes beyond the dynamic connectors as defined in [14].

5. Algebraic & categorical properties

Algebraic and categorical structures often form powerful abstractions in functional programming languages. Such structures, such as monoids, functors and monads [15], as well as many others [16], have been successfully used in the context of functional programming.

In Haskell, those structures form typeclasses such as `Monoid`, `Functor` and `Monad` and can be used very effectively to build high-level expressions. The `Monad` typeclass is so prevalent that syntactic sugar (the *do notation*) is present in the language to ease its use.

In Scala, those structures are not explicitly present. However, many methods, such as `map` and `flatMap` mirror the functor and monad operations. Syntactic sugar (the *for-notation*) is present in the language and, as in Haskell, makes the use of multiple `map` and `flatMap` operations syntactically lighter.

In this section, we show that connector combinators form such structures. This allows connectors to be made instances of the corresponding typeclasses and allows programmers to use well-known abstractions and syntactic sugar to build connectors.

We only provide sketches of proofs—some complete proofs are provided in the appendix, while others can be found in [11].

5.1. Algebraic properties

Below, we provide two lemmata that show that connector combinators form two monoidal structures. However, first we have to define an equality relation on combinators. This is achieved by a canonical lifting of value and function equalities:

Definition 5.1 *(Connector equality).* Two connectors $c_1, c_2 \in \mathcal{C}$ are *equal* if and only if they define the same open interactions, for any system state, i.e.

$$c_1 = c_2 \overset{def}{\iff} \forall S \in \mathcal{S}, [c_1](S) = [c_2](S)\,.$$

Lemma 5.2. *Connector equality is a congruence w.r.t. all connector combinators introduced in Section 4.*

Proof sketch. The statement of the lemma follows straightforwardly from the observation that, for a given state S, the open-interaction semantics, $[\cdot](S)$, of every combinator involving a sub-connector c—i.e. every combinator other than `Bind`, `Success`, `Failure` and `Dynamic`—is defined per element of $[c](S)$ and without relying on the structure of c. $\square$

Lemma 5.3. $(\mathcal{C}, \texttt{OneOf}, \texttt{Failure})$ *is a commutative monoid, that is* `OneOf` *is associative and commutative, with the identity element* `Failure`.

Proof sketch. The lemma trivially follows from the properties of set-theoretical operations. $\square$

Lemma 5.4. *Given a commutative monoid* $(\mathcal{V}, \times, 1)$, *the structure* $\big(\mathcal{C}, \texttt{Mapped}\ (\cdot\times\cdot)\ (\texttt{BothOf}\ \cdot\ \cdot), \texttt{Success}\ 1\big)$ *is also a commutative monoid, that is* $\texttt{Mapped}\ (\cdot\times\cdot)\ (\texttt{BothOf}\ \cdot\ \cdot)$ *is associative and commutative, with the identity element* $\texttt{Success}\ 1$.

Proof sketch. The lemma follows directly from the properties of set-theoretical operations and the fact that $(\mathcal{V}, \times, 1)$ is a commutative monoid. $\square$

Notice that, taking $\times$ to be the usual Cartesian product, the resulting structure $\big(\mathcal{C}, \texttt{BothOf}, \texttt{Success}\ 1\big)$ is a commutative monoid *up to isomorphism*.

5.2. Categorical properties

Notice that the values $\mathcal{V}$ (comprising data values, functions and connectors) and types introduced in Section 3 form a category, whose objects are types, whereas arrows between objects a and b are functions $f : a \to b \in \mathcal{V}$. We denote this category T.

Given types u and d, we define two mappings $F^d_{up}, F^u_{down} : T \to T$ by putting, for each type a and each function $f : a \to b$,

$$F^d_{up}(a) = a \uparrow\downarrow d\,, \qquad F^d_{up}(f) = (\texttt{Mapped}\ f) : (a \uparrow\downarrow d) \to (b \uparrow\downarrow d)\,,$$
$$F^u_{down}(a) = u \uparrow\downarrow a\,, \qquad F^u_{down}(f) = (\texttt{ContraMapped}\ f) : u \uparrow\downarrow b \to u \uparrow\downarrow a\,.$$

The following propositions establish the basic properties of these mappings, showing that F^d_{up} is a functor—symmetrically, F^u_{down} is a contravariant functor—and that F^d_{up} along with `Success` and `Joined` form a monad. The proofs are straightforward and are provided in the appendix.

Proposition 5.5. *For any type d, the mapping F^d_{up} is a functor from the category T to itself, that is*

$$\texttt{Mapped}\ identity_a = identity_{(a\uparrow\downarrow d)}\,,$$
$$\texttt{Mapped}\ (g \circ f) \ = \texttt{Mapped}\ g \circ \texttt{Mapped}\ f\,.$$

Symmetrically, for any type u, the mapping F^u_{down} is a contravariant functor from the category T to itself, that is

$$\texttt{ContraMapped}\ identity_a = identity_{(a\uparrow\downarrow d)}\,,$$
$$\texttt{ContraMapped}\ (g \circ f) \ = \texttt{ContraMapped}\ f \circ \texttt{ContraMapped}\ g\,.$$

Proposition 5.6. *For any type d, the functor F^d_{up} along with* `Success` *and* `Joined` *form a monad, i.e. the following four properties hold:*

- `Success` *is a natural transformation from 1_T to F^d_{up}:*

$$\texttt{Success} \circ f = \texttt{Mapped}\ f \circ \texttt{Success}\,,$$

- `Joined` *is a natural transformation from $F^d_{up} \circ F^d_{up}$ to F^d_{up}:*

$$\texttt{Joined} \circ \texttt{Mapped}\ (\texttt{Mapped}\ f) = \texttt{Mapped}\ f \circ \texttt{Joined}\,,$$

- *First monad law:*

$$\texttt{Joined} \circ \texttt{Mapped}\ \texttt{Joined} = \texttt{Joined} \circ \texttt{Joined}\,,$$

- *Second monad law:*

$$\texttt{Joined} \circ \texttt{Mapped}\ \texttt{Success} = \texttt{Joined} \circ \texttt{Success} = identity\,.$$

6. Implementation

We have implemented the concepts presented in this paper in two functional programming languages, Haskell[6] and Scala.[7] These implementations allow programmers to build concurrent systems following the principles of BIP in very expressive high-level languages. Thus programmers can separately describe the behaviour and coordination of their system.

The choice of using an embedded DSL approach was motivated by the fact that the various combinators we have defined correspond to well-known constructs in both Haskell and Scala. Connectors can be built using already existing combinators and syntactic sugar (`do`-notation and `for`-notation). However, the approach has some drawbacks. Since parts of the code are opaque to the framework, the embedded DSL approach offers less opportunities for analysis and optimisations, compared to an external DSL approach. Meta-programming techniques could be used to alleviate this problem, but were not explored during this work.

Both implementations allow programmers to describe concurrent systems with a set of instructions, which are used to declare the atoms and ports of the system. Special instructions, such as `await` and `spawn` are provided for atoms to respectively wait on ports and spawn children atoms. Note that the automata and their description are not directly present in the frameworks. Instead, the behaviour of atoms is expressed directly in the host language, in a fashion that is opaque to

[6] https://github.com/redelmann/bip-in-haskell.
[7] https://github.com/redelmann/bip-in-scala.

```haskell
data Connector s d u
```

(a) Connector type in Haskell

```scala
class Connector[-D, +U]
```

(b) Connector type in Scala

Fig. 12. The connector type in Haskell and Scala.

the execution engine. The only actions that can be observed by the engine are calls to special instructions such as `await` and `spawn`.

Connectors are described using functions and methods which correspond directly to the connector combinators we have introduced in Section 4. A library of derived combinators is also made available in both languages. The connector type indicates the types of upward and downward values propagated by the connector. In both cases, connectors are encoded as Generalised Algebraic Data Types (GADTs) as illustrated in Fig. 12. Notice that the Haskell connector type has a type parameter `s`. This parameter is a phantom type, used to ensure that identifiers do not escape the scope of the system in which they are defined.

As shown in Section 5, connectors follow the laws of many interesting algebraic structures. This allows us to make connectors instances of many Haskell typeclasses, such as `Monoid`, `Functor`, `ProFunctor`, `Applicative`, `Alternative`, `Monad` and `MonadPlus`. These typeclasses allow programmers to build connectors using concepts and functions they already are familiar with.

In Scala, the `Connector` class implements common methods such as `map`, `flatMap` and `filter`. These methods often mirror those of Haskell typeclasses.

The execution engine is composed of three distinct parts:

The pool of threads is used to concurrently execute the behaviour of atoms.

The system state records information about waiting atoms. It is encoded as a mapping from atoms and ports to:

- nothing, if the atom is not currently waiting on the port;
- the value sent by the atom on the port, along with the continuation of the atom, if the atom is waiting on the port.

The continuations of atoms are stored to avoid needlessly blocking the execution thread of waiting atoms.

The engine core computes possible interactions from the system state and connector. This computation takes place when all atoms are either done with their execution or waiting on ports.

The engine recursively computes a stream of possible open interactions from the system's connector. The order in which open interactions are produced is arbitrary, which is in accordance with the non-deterministic semantics. For efficiency reasons, the first possible interaction produced at the top level of the connector is chosen.

In order to efficiently compute downwards compatibility checks, open interactions are extended to explicitly contain the set of all involved atoms. Two open interactions are downwards compatible if and only if the two sets of involved atoms are disjoint.

The execution engine runs the system in a lock-step fashion. The next interaction is only computed when the system has reached a stable state, that is only when all atoms are either waiting or have completed their execution. In Edelmann's Master thesis [11], we investigate a way to conservatively execute some interactions earlier, while some atoms are still executing.

We illustrate the implementations of Functional BIP by a concurrent system composed of a producer and 20 consumers. The producer repeatedly produces values that are then sent on the `send` port of the system. Consumers repeatedly wait to receive a value on the `receive` port and then directly consume it. The connector of the system states that the values sent by the producer on the `send` port may be transmitted to any atom waiting on the `receive` port. Fig. 13 shows a Haskell and a Scala implementations.

We also present a Haskell implementation of a Token Ring system. In this example, atoms are arranged to form a ring. An integer value is exchanged around this ring by the different atoms of the system. The value is incremented at each step until a given maximal value is reached. This example is presented in Fig. 14.

7. Related work

We show that an expressive component framework such as BIP can be defined as a DSL based on general purpose functional programming languages. Writing DSL's in functional languages allows their programmers to use powerful modelling concepts and tools with minimal effort, while still using a domain specific interface. Similar work can be found in [17,18]. Approaches such as [19] also use combinators to describe coordination, but do so locally at each process. These approaches have no notion of a global coordination object such as the connectors we present.

Dynamicity in BIP has been studied by several authors [14,20,21]. In [14], the authors present the Dy-BIP framework that allows dynamic reconfiguration of connectors among the ports of the system. They use *history variables* to allow sequences of interactions with the same instance of a given component type. Functional BIP can emulate history variables using data. Dynamic combinators allow reconfiguration of connectors, but also creation and deletion of atoms, thereby extending

```haskell
main = runSystem $ do

  -- Creation of the two ports.
  -- Port on which to send the values
  -- produced.
  send <- newPort

  -- Port on which to receive the values
  -- to consume.
  receive <- newPort

  -- Creation of 20 consumers.
  replicateM 20 $ newAtom $ forever $ do
    value <- await receive ()
    consume value

  -- Creation of the producer.
  producer <- newAtom $ forever $ do
    value <- produce
    await send value

  -- The connector of the system
  registerConnector $
    -- The producer on its send port.
    bind producer send
    <*
    -- Any atom on the request port.
    dynamic receive
```

(a) Haskell

```scala
object ProducerConsumers {
  def main(args: Array[String]) {
    // Creation of the system.
    val system = new System

    // Creation of the two ports of the system.
    val send    = system.newPort[Any, Int]
    val receive = system.newPort[Int, Unit]

    // Creation of the consumers.
    for (_ <- 1 to 20) yield system.newAtom {
      def act() {
        await(receive) { (value: Int) =>
          consume(receive)
          act()
        }
      }
      act()
    }
    // Creation of the producer.
    val producer = system.newAtom {
      def act() {
        val value = produce()
        await(send, value) { (_: Any) =>
          act()
        }
      }
      act()
    }
    // The connector of the system.
    system.registerConnector {
      producer.bind(send).
              andLeft(dynamic(receive))
    }
    system.run()
  }
}
```

(b) Scala

Fig. 13. Producer–Consumers example implemented in Haskell and Scala.

the expressivity with respect to Dy-BIP. From this perspective, the approach in [20] is closest to the one we adopted in Functional BIP. The authors define two extensions of the BIP model without priorities, which they abbreviate to BI(P): reconfigurable BI(P)—similar to Dy-BIP—and dynamic BI(P), allowing dynamic replication of components. They focus on the operational semantics of the two extensions and their properties, by studying their encodability in BIP and Place/Transition Petri nets (P/T Nets). Composition is defined through interaction models, without considering structured connectors. In

```
tokenRing nAtoms nExchanges = runSystem $ mdo

    send <- newPort     -- Port on which to send the token and its recipient.
    receive <- newPort  -- Port on which to receive the token.

    -- Behaviour of the i^th atom.
    let atom i = do
          n <- await receive ()
          liftIO $ putStrLn $ "Atom " ++ show i ++ " received " ++ show n
          let n' = succ n
          when (n' <= nExchanges) $ do
              await send (n', next i)
              atom i

    -- Creating the first atom.
    a <- newAtom $ do
        liftIO $ putStrLn "Atom 0 ready to send."
        await send (1, next 0)
        atom 0  -- From now on, acts as any other node.

    -- Creating the n - 1 next atoms.
    as <- forM [ 1 .. pred nAtoms ] $ \ i ->
        newAtom $ atom i

    -- Vector of atom identifiers.
    let v = fromList as
    let next i | i == pred nAtoms = a
               | otherwise = v ! i

    registerConnector $ do
        (value, destination) <- dynamic send  -- Accepts any atom on the port send.
        bind destination receive  -- Binds the receiver on the receive port.
        return value  -- Only propagate upward the value sent.
```

Fig. 14. Token Ring example implemented in Haskell.

contrast, the present paper focuses exclusively on connectors structured by combinators, disregarding the details of the operational semantics of components. As opposed to [20], we have defined a complete set of combinators, including data and priority. Although, the expressiveness of the two approaches w.r.t. dynamicity seems very close, we leave the formal investigation for future work.

In [21], the authors revisit the BIP expressiveness, by introducing simple behaviour, such as prefixing, in the BIP glue operators. They show that such minor modifications can rapidly lead to Turing completeness of glue. In contrast to [21] and other frameworks, such as [22,23], Functional BIP relies on a clear distinction between behaviour and coordination expressed by memoryless connectors obtained by combinator composition.

In contrast to other formalisms such as [24] our framework supports full dynamism and is rooted in rigorous abstract semantics. In [25], dynamic architectures are defined as a set of global transitions between global configurations. These transitions are expressed in a first order logic extended with architecture-specific predicates. The same logic is used in [26, 27] but global configurations are computed at run time from the local constraints of each component. [28] provides an operational semantics based on the composition of global configurations from local ones. These express three forms of dependencies between services (mandatory, optional and negative). Nonetheless, dynamism is supported only at the installation phase.

8. Conclusion

The paper shows how the BIP component framework can be embedded in functional host languages. The embedding consists in defining sets of connector combinators that can describe the coordination mechanisms of BIP. The definition is progressive and incremental. We first define combinators to express synchronisation and associated data transfer between the components of a system. Then, we have introduced combinators for the application of priority policies allowing conflict resolution between enabled interactions. Finally, we have presented combinators to deal with the dynamic creation of components and connectors.

We have shown that the set of the defined combinators enjoy interesting algebraic properties and form well-known algebraic structures which are of particular importance for the implementation of connectors in functional programming languages.

The two implementations show how these concepts can be transposed, respectively, into Haskell and Scala. For both languages, we have released an open source framework which programmers can use to build concurrent systems using the high-level coordination primitives offered by BIP.

Appendix A. Additional proofs

Proof of Lemma 4.10: For any connector $c : u \uparrow\downarrow d$ and any system state $S \in \mathcal{S}$, we have $x \in sem(u)$, for all $(x, f) \in [c](S)$.

Proof. We prove the lemma by structural induction on the typing rules.

Case `Bind` *a p*

Let $c = \texttt{Bind}\ a\ p$, with some atom $a \in \mathcal{A}$ and port $p \in P$. Then, by the typing rule of `Bind` (3), $u = p^{up}$ and $d = p^{down}$. If $(a, p) \in domain(S)$, by (4), $x = S(a, p)$. Furthermore, by the definition of state (1),

$$S(a, p) \in sem(p^{up}) = sem(u)\,.$$

If $(a, p) \notin domain(S)$, then the proposition trivially holds, as, by (4), $[\texttt{Bind}\ a\ p](S) = \emptyset$.

Case `Success`

Let $c = \texttt{Success}\ v$, with some value $v \in \mathcal{V}$. Then, by the typing rule of `Success` (5), it must be the case that $v : u$ and thus that $v \in sem(u)$. By (6), we have $x = v$. Thus the proposition trivially holds.

Case `Failure`

Let $c = \texttt{Failure}$. Then, by (7), $[c](S) = \emptyset$ and the proposition trivially holds.

Case `OneOf`

Let $c = \texttt{OneOf}\ c_1\ c_2$. Then, by the typing rule of `OneOf` (8), it must be the case that $c_1 : u \uparrow\downarrow d$ and $c_2 : u \uparrow\downarrow d$. By induction hypothesis, the proposition holds for c_1 and c_2. Thus, the proposition holds for c, as, by (9), $[c](S) = [c_1](S) \cup [c_2](S)$.

Case `BothOf`

Let $c = \texttt{BothOf}\ c_1\ c_2$. Then, by the typing rule of `BothOf` (10) it must be the case that:

- $c_1 : v \uparrow\downarrow d$ for some type v,
- $c_2 : w \uparrow\downarrow d$ for some type w,
- $u = v \times w$.

Furthermore, by (11), for any $(x, f) \in [c](S)$, we have $x = (x_1, x_2)$ and $f = f_1 \cup f_2$, such that $(x_1, f_1) \in [c_1](S)$ and $(x_2, f_2) \in [c_2](S)$. By the induction hypothesis, the proposition holds for c_1 and c_2, that is $x_1 \in sem(v)$ and $x_2 \in sem(w)$. Hence, $x = (x_1, x_2) \in sem(u)$.

Case `Mapped`

Let $c = \texttt{Mapped}\ g\ c_1$. Then, by the typing rule of `Mapped` (20), we have that $c_1 : v \uparrow\downarrow d$, for some type v, and $g : v \to u$. Furthermore, by (21), for any $(x, f) \in [c](S)$, we have $x = g(x_1)$, such that $(x_1, f_1) \in [c_1](S)$, for some downward function f_1. By induction hypothesis, we have $x_1 \in sem(v)$ and, consequently, $x = g(x_1) \in sem(u)$.

Case `ContraMapped`

Let $c = \texttt{ContraMapped}\ g\ c_1$. Then, by the typing rule of `ContraMapped` (20), it must be the case that $c_1 : u \uparrow\downarrow e$ for some type e. Furthermore, by (22), for any $(x, f) \in [c](S)$, we have $(x, f_1) \in [c_1](S)$, with some downward function f_1, such that $f = f_1 \circ g$. Thus $x \in sem(u)$ by the induction hypothesis.

Case `Guarded`

Let $c = \texttt{Guarded}\ g\ c_1$. Then, by the typing rule of `Guarded` (23), we have that:

- $g : u \to \texttt{bool}$ and
- $c_1 : u \uparrow\downarrow d$.

Furthermore, by (24), for any $(x, f) \in [c](S)$, we have $(x, f) \in [c_1](S)$ and $g(x) = \texttt{true}$. Thus, $x \in sem(u)$, by the induction hypothesis.

Case `Feedback`

Let $c = \texttt{Feedback}\ c_1$. By the typing rule of `Feedback` (25), we have $c_1 : u \uparrow\downarrow (d \times u)$. By (26), for any $(x, f) \in [c](S)$, we have $(x, g) \in [c_1](S)$, with some downward function g. Thus, the proposition holds trivially by the induction hypothesis.

Case `FirstOf`

Let $c = \texttt{FirstOf}\ c_1\ c_2$. Then, by the typing rule of `FirstOf` (16), it must be the case that $c_1 : u \uparrow\downarrow d$ and $c_2 : u \uparrow\downarrow d$. Furthermore, by (17), for any $(x, f) \in [c](S)$, we have either $(x, f) \in [c_1](S)$ or $(x, f) \in [c_2](S)$. Thus, the proposition trivially holds by the induction hypothesis.

Case `Maximal`

Let $c = \texttt{Maximal}\ g\ c_1$, then, by the typing rule of `Maximal` (18), we must have that $c_1 : u \uparrow\downarrow d$ and $g : (u \times u) \to \texttt{bool}$. Furthermore, by (19), $[c](S) \subseteq [c_1](S)$. Thus, for any $(x, f) \in [c](S)$, we also have $(x, f) \in [c_1](S)$ and the proposition holds by the induction hypothesis.

Case `Dynamic`

Let $c = \texttt{Dynamic}\ p$ for some port $p \in P$. By the typing rule of `Dynamic` (27), we have $u = p^{up}$. Furthermore, by (28), for any $(x, f) \in [c](S)$, we have $x = S(a, p)$, for some atom a, such that $(a, p) \in domain(S)$. By the definition of state (see Section 3.7), we have $x : p^{up}$, hence $x \in sem(u)$.

Case `Joined`

Let $c = \texttt{Joined}\ c_1$. Then, by the typing rule of `Joined` (29), it must be the case that $c_1 : (u \uparrow\downarrow d) \uparrow\downarrow d$. Furthermore, by (30), for any $(x, f) \in [c](S)$, we have $f = f_1 \cup f_2$, with $(x, f_1) \in [c_2](S)$ for some connector $c_2 : u \uparrow\downarrow d$, such that $(c_2, f_2) \in [c_1](S)$. Notice that, by the induction hypothesis, we, indeed, have $c_2 \in sem(u \uparrow\downarrow d)$. Hence, by a second application of the induction hypothesis, $x \in sem(u)$. □

Proof of Lemma 4.11: For any connector $c : u \uparrow\downarrow d$ and any system state $S \in \mathcal{S}$, we have $domain(f) = sem(d)$, for all $(x, f) \in [c](S)$.

Proof. We prove the lemma by structural induction on the typing rules.

Case `Bind` *a p*

Let $c = \texttt{Bind}\ a\ p$, for some atom $a \in \mathcal{A}$ and port $p \in P$. Then, by the typing rule of `Bind` (3), it must be the case that $c : p^{up} \uparrow\downarrow p^{down}$. Furthermore, by (4), if $(a, p) \in domain(S)$, then, for any $(x, f) \in [c](S)$, the domain of f is $sem(p^{down})$, i.e. $domain(f) = sem(d)$. If $(a, p) \notin domain(S)$, then the proposition trivially holds, as $[\texttt{Bind}\ a\ p](S) = \emptyset$.

Case `Success`

Let $c = \texttt{Success}\ v$, with some value $v : u$. Then, by (6), for any $(x, f) \in [c](S)$ we have $domain(f) = sem(d)$. Thus, the proposition trivially holds.

Case `Failure`

Let $c = \texttt{Failure}$. Then, by (7), $[c](S) = \emptyset$ and the proposition trivially holds.

Case `OneOf`

Let $c = \texttt{OneOf}\ c_1\ c_2$. Then, by the typing rule of `OneOf` (8), it must be the case that $c_1 : u \uparrow\downarrow d$ and $c_2 : u \uparrow\downarrow d$. Furthermore, by (9), for any $(x, f) \in [c](S)$, we either have $(x, f) \in [c_1](S)$ or $(x, f) \in [c_2](S)$. In both cases, we have $domain(f) = sem(d)$ by the induction hypothesis.

Case `BothOf`

Let $c = \texttt{BothOf}\ c_1\ c_2$. Then, by the typing rule of `BothOf` (10), it must be the case that:

- $c_1 : v \uparrow\downarrow d$ for some type v.
- $c_2 : w \uparrow\downarrow d$ for some type w.

Furthermore, by (11), for any $(x, f) \in [c](S)$, we have $x = (x_1, x_2)$ and $f = f_1 \cup f_2$, for some $(x_1, f_1) \in [c_1](S)$ and $(x_2, f_2) \in [c_2](S)$. By the induction hypothesis, $domain(f_1) = domain(f_2) = sem(d)$. Hence, also $domain(f) = sem(d)$.

Case `Mapped`

Let $c = \texttt{Mapped}\ g\ c_1$. Then, by the typing rule for `Mapped` (20), it must be the case that $c_1 : v \uparrow\downarrow d$ and $g : v \to u$. Furthermore, by (21), for any $(x, f) \in [c](S)$, we have $(x, f) = (g(y), f)$, for some $(y, f) \in [c_1](S)$ and the proposition holds, by the induction hypothesis.

Case `ContraMapped`

Let $c = \texttt{ContraMapped}\ g\ c_1$, then by the typing rule of `ContraMapped` (20), we have that $c_1 : u \uparrow\downarrow e$, for some type e, and $g : d \to e$, i.e. $domain(g) = sem(d)$. Furthermore, for any $(x, f) \in [c](S)$, by (22), we have $(x, f) = (x, f_1 \circ g)$, for some $(x, f_1) \in [c_1](S)$.

By the induction hypothesis, we have $sem(e) = domain(f_1)$. Thus, as $range(g) \subseteq sem(e) = domain(f_1)$, the composition $f_1 \circ g$ is well-defined and we have $domain(f_1 \circ g) = domain(g) = sem(d)$.

Case `Guarded`

Let $c = \texttt{Guarded}\ g\ c_1$. Then, by the typing rule of `Guarded` (23), we have that:

- $g : u \to \texttt{bool}$ and
- $c_1 : u \uparrow\downarrow d$.

Furthermore, by (24), for any $(x, f) \in [c](S)$, we have $(x, f) \in [c_1](S)$ and $g(x) = \texttt{true}$. Thus, $domain(f) = sem(d)$, by the induction hypothesis.

Case `Feedback`

Let $c = \texttt{Feedback}\ c_1$. Then, by the typing rule of `Feedback` (25), it must be the case that $c_1 : u \uparrow\downarrow (d \times u)$. Furthermore, for any $(x, f) \in [c](S)$, by (26), we have $(x, f) = (x, g \circ tag_x)$, for some $(x, g) \in [c_1](S)$, with $tag_x(y) \overset{def}{=} (y, x)$.

By Lemma 4.10, we have $x \in sem(u)$. Hence, $tag_x : d \to (d, u)$, i.e. $range(tag_x) \subseteq sem(d \times u)$. By the induction hypothesis, we have $sem(d \times u) = domain(g)$. Hence, the composition $g \circ tag_x$ is well-defined and $domain(f) = domain(tag_x) = sem(d)$.

Case `FirstOf`

Let $c = \texttt{FirstOf}\ c_1\ c_2$. Then, by the typing rule of `FirstOf` (16), it must be the case that $c_1 : u \uparrow\downarrow d$ and $c_2 : u \uparrow\downarrow d$. Furthermore, by (17), for any $(x, f) \in [c](S)$, we have either $(x, f) \in [c_1](S)$ or $(x, f) \in [c_2](S)$. Thus, the proposition trivially holds by the induction hypothesis.

Case `Maximal`

Let $c = \texttt{Maximal}\ g\ c_1$, then, by the typing rule of `Maximal` (18), we must have that $c_1 : u \uparrow\downarrow d$ and $g : (u \times u) \to \texttt{bool}$. Furthermore, by (19), $[c](S) \subseteq [c_1](S)$. Thus, for any $(x, f) \in [c](S)$, we also have $(x, f) \in [c_1](S)$ and the proposition holds by the induction hypothesis.

Case `Dynamic` *p*

Let $c = \texttt{Dynamic}\ p$, with some port $p \in P$. Then, by the typing rule for `Dynamic` (27), we have $d = p^{down}$. Furthermore, by (28), for any $(x, f) \in [c](S)$, we have $domain(f) = sem(p^{down})$. Thus, the proposition trivially holds.

Case `Joined`

Let $c = \texttt{Joined}\ c_1$. Then, by the typing rule of `Joined` (29), it must be the case that $c_1 : (u \uparrow\downarrow d) \uparrow\downarrow d$. Furthermore, by (30), for any $(x, f) \in [c](S)$, we have $f = f_1 \cup f_2$, with $(x, f_1) \in [c_2](S)$ for some connector c_2, such that $(c_2, f_2) \in [c_1](S)$. Hence, $domain(f) = domain(f_1) = domain(f_2)$.

By Lemma 4.10, $c_2 \in sem(u \uparrow\downarrow d)$, that is, indeed, $c_2 : u \uparrow\downarrow d$. Hence, by the induction hypothesis, $domain(f_1) = domain(f_2) = sem(d)$. Hence, $domain(f) = domain(f_1) = domain(f_2) = sem(d)$. □

Proof of Proposition 5.5: For any type d, the mappings F^d_{up} and F^u_{down} are, respectively, covariant and contravariant functors from the category T to itself.

We will only prove the proposition statement for F^d_{up}. The statement for F^u_{down} is symmetrical.

Proof. To show that F^d_{up} is a functor, we show that identity and composition are preserved.

Identity We have to prove that, for any two types u and d, $F^d_{up}(1_u) = 1_{F^d_{up}(u)}$. Let u by any type. By definition, 1_u is the identity function $identity_u : u \to u$. Therefore, we have that:

$$F^d_{up}(1_u) = \texttt{Mapped}\ identity_u\,.$$

We observe that $1_{F^d_{up}(u)}$ is the identity function that maps connectors of type $u \uparrow\downarrow d$ to themselves. Therefore, for the identity preservation property to hold, we must have that for any connector c of the correct type the following holds:

$$\texttt{Mapped}\ identity_u\ c = c\,.$$

By definition, the above equality is equivalent to the following statement:

$$\forall S \in \mathcal{S}.[\texttt{Mapped}\ identity_u\ c](S) = [c](S)\,.$$

Let S be any system state. We have that the above statement is trivially true as:

$$\begin{aligned}[\texttt{Mapped}\ identity_u\ c](S) &= \{(identity_u(x), f) \mid (x, f) \in [c](S)\}\\ &= \{(x, f) \mid (x, f) \in [c](S)\}\\ &= [c](S)\,.\end{aligned}$$

Composition We are left to show that indeed $F^d_{up}(g \circ h) = F^d_{up}(g) \circ F^d_{up}(h)$, i.e. that

$$\texttt{Mapped}\ (g \circ h) = (\texttt{Mapped}\ g) \circ (\texttt{Mapped}\ h)\,,$$

that is, for any connector c of appropriate type, we must have that:

$$\texttt{Mapped}\ (g \circ h)\ c = \texttt{Mapped}\ g\ (\texttt{Mapped}\ h\ c)\,.$$

By Definition 5.1, this equality reduces to:

$$\forall S \in \mathcal{S}.[\texttt{Mapped}\ (g \circ h)\ c](S) = [\texttt{Mapped}\ g\ (\texttt{Mapped}\ h\ c)](S)\,.$$

Let S be any system state. We trivially have that:

$$\begin{aligned}[\texttt{Mapped}\ (g \circ h)\ c](S) &= \{(g(h(x)), h) \mid (x, h) \in [c](S)\}\\ &= \{(g(y), h) \mid (y, h) \in [\texttt{Mapped}\ h\ c](S)\}\\ &= [\texttt{Mapped}\ g\ (\texttt{Mapped}\ h\ c)](S)\,. \quad \square\end{aligned}$$

Proof of Proposition 5.6: For any type d, the functor $F^d_{up} : T \to T$ along with `Success` *and* `Joined` *form a monad.*

In order to prove the proposition, we consider two families of mappings $\eta : 1_T \to F^d_{up}$ and $\mu : F^d_{up} \circ F^d_{up} \to F^d_{up}$, defined as follows (recall that the objects of T are types):

- for each type u, the mapping $\eta_u : u \to F^d_{up}(u)$ associates, to each $v \in sem(u)$, the connector $(\texttt{Success}\ v) : u \uparrow\downarrow d$;
- for each type u, the mapping $\mu_u : (u \uparrow\downarrow d) \uparrow\downarrow d \to u \uparrow\downarrow d$ associates, to each $c : (u \uparrow\downarrow d) \uparrow\downarrow d$, the connector $(\texttt{Joined}\ c) : u \uparrow\downarrow d$.

We will show that both these families are natural transformations and that, together with F^d_{up}, they satisfy the necessary coherence conditions.

Proof. $\eta : 1_T \to F^d_{up}$ *is a natural transformation.* We have to prove that the following equality holds for any function $g : a \to b$:

$$(\eta_b \circ 1_T)(g) = F^d_{up}(g) \circ \eta_a$$

Which, in this context, translates to:

$$\texttt{Success} \circ g = (\texttt{Mapped}\ g) \circ \texttt{Success}$$

Or, for any value $v \in sem(a)$:

$$\texttt{Success}\ g(v) = \texttt{Mapped}\ g\ (\texttt{Success}\ v)$$

Which is trivially true by definition of the semantics.

$\mu : F^d_{up} \circ F^d_{up} \to F^d_{up}$ *is a natural transformation.* We have to prove that the following equality holds, for any function $g : a \to b$,

$$\big(\mu_b \circ (F^d_{up} \circ F^d_{up})\big)(g) = F_{up}(g) \circ \mu_a\,.$$

Which, in this context, translates to

$$\texttt{Joined} \circ \texttt{Mapped}\ (\texttt{Mapped}\ g) = (\texttt{Mapped}\ g) \circ \texttt{Joined}$$

or, for any connector $c : (u \uparrow\downarrow d) \uparrow\downarrow d$,

$$\texttt{Joined}\ (\texttt{Mapped}\ (\texttt{Mapped}\ g)\ c) = \texttt{Mapped}\ g\ (\texttt{Joined}\ c)\,.$$

By definition of the equality on connectors, this is equivalent to the following statement:

$$\forall S \in \mathcal{S}.[\texttt{Joined}\ (\texttt{Mapped}\ (\texttt{Mapped}\ g)\ c)](S) = [\texttt{Mapped}\ g\ (\texttt{Joined}\ c)](S)$$

Let S be any system state. We have that:

$$\begin{aligned}&[\texttt{Joined}\ (\texttt{Mapped}\ (\texttt{Mapped}\ g)\ c)](S)\\ &\quad = \Big\{(x, f_1 \cup f_2) \,\Big|\, (c_1, f_1) \in [\texttt{Mapped}\ (\texttt{Mapped}\ g)\ c](S),\ (x, f_2) \in [c_1](S) \text{ and}\\ &\qquad\qquad f_1 \text{ and } f_2 \text{ are downward compatible}\Big\}\\ &\quad = \Big\{(x, f_1 \cup f_2) \,\Big|\, (c_1, f_1) \in [c](S),\ (x, f_2) \in [\texttt{Mapped}\ g\ c_1](S) \text{ and}\\ &\qquad\qquad f_1 \text{ and } f_2 \text{ are downward compatible}\Big\}\\ &\quad = \Big\{(g(x), f_1 \cup f_2) \,\Big|\, (c_1, f_1) \in [c](S),\ (x, f_2) \in [c_1](S) \text{ and}\\ &\qquad\qquad f_1 \text{ and } f_2 \text{ are downward compatible}\Big\}\\ &\quad = [\texttt{Mapped}\ g\ (\texttt{Joined}\ c)](S)\,.\end{aligned}$$

Identity We have to show that, for any type u and any connector $c : u \uparrow\downarrow v$,

$$(\mu_u \circ F^d_{up}(\eta_u))(c) = (\mu_u \circ \eta_{F^d_{up}(u)})(c) = c\,,$$

which translates to:

$$\texttt{Joined}\,(\texttt{Mapped}\,\texttt{Success}\,c) = \texttt{Joined}\,(\texttt{Success}\,c) = c\,.$$

For any system state S, we have

$$\begin{aligned}
&[\texttt{Joined}\,(\texttt{Mapped}\,\texttt{Success}\,c)](S)\\
&\quad = \big\{(x, f_1 \cup f_2)\,\big|\,(c_1, f_1) \in [\texttt{Mapped}\,\texttt{Success}\,c](S), (x, f_2) \in [c_1](S) \text{ and}\\
&\qquad\qquad f_1 \text{ and } f_2 \text{ are downward compatible}\big\}\\
&\quad = \big\{(x, f_1 \cup f_2)\,\big|\,(c_1, f_1) \in [c](S), (x, f_2) \in [\texttt{Success}\,c_1](S) \text{ and}\\
&\qquad\qquad f_1 \text{ and } f_2 \text{ are downward compatible}\big\}\\
&\quad = \big\{(c_1, f_1)\,\big|\,(c_1, f_1) \in [c](S)\big\}\\
&\quad = [c](S)\\
&[\texttt{Joined}\,(\texttt{Success}\,c)](S)\\
&\quad = \big\{(x, f_1 \cup f_2)\,\big|\,(c_1, f_1) \in [\texttt{Success}\,c](S), (x, f_2) \in [c_1](S) \text{ and}\\
&\qquad\qquad f_1 \text{ and } f_2 \text{ are downward compatible}\big\}\\
&\quad = \big\{(x, f_2)\,\big|\,(x, f_2) \in [c](S)\big\}\\
&\quad = [c](S)\,.
\end{aligned}$$

Associativity We have to show that, for any type u and any connector $c : ((u \uparrow\downarrow d) \uparrow\downarrow d) \uparrow\downarrow d$,

$$(\mu_u \circ F^d_{up}(\mu_u))(c) = (\mu_u \circ \mu_{F^d_{up}(u)})(c)\,,$$

which translates to The first law translates in this context to:

$$\texttt{Joined}\,(\texttt{Mapped}\,\texttt{Joined}\,c) = \texttt{Joined}\,(\texttt{Joined}\,c)$$

For any system state S, we have

$$\begin{aligned}
&[\texttt{Joined}\,(\texttt{Mapped}\,\texttt{Joined}\,c)](S)\\
&\quad = \big\{(x, f_1 \cup f_2)\,\big|\,(c_1, f_1) \in [\texttt{Mapped}\,\texttt{Joined}\,c](S), (x, f_2) \in [c_1](S) \text{ and}\\
&\qquad\qquad f_1 \text{ and } f_2 \text{ are downward compatible}\big\}\\
&\quad = \big\{(x, f_1 \cup f_2)\,\big|\,(c_1, f_1) \in [c](S), (x, f_2) \in [\texttt{Joined}\,c_1](S) \text{ and}\\
&\qquad\qquad f_1 \text{ and } f_2 \text{ are downward compatible}\big\}\\
&\quad = \big\{(x, f_1 \cup f_2 \cup f_3)\,\big|\,(c_1, f_1) \in [c](S), (c_3, f_3) \in [c_1](S), (x, f_2) \in [c_3](S) \text{ and}\\
&\qquad\qquad f_1, f_2 \text{ and } f_3 \text{ are downward compatible}\big\}\\
&\quad = \big\{(x, f_2 \cup f_3)\,\big|\,(c_3, f_3) \in [\texttt{Joined}\,c](S), (x, f_2) \in [c_3](S) \text{ and}\\
&\qquad\qquad f_2 \text{ and } f_3 \text{ are downward compatible}\big\}\\
&\quad = [\texttt{Joined}\,(\texttt{Joined}\,c)](S) \quad \square
\end{aligned}$$

References

[1] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, J. Sifakis, Rigorous component-based system design using the BIP framework, IEEE Softw. 28 (3) (2011) 41–48.

[2] S. Bliudze, J. Sifakis, The algebra of connectors—structuring interaction in BIP, IEEE Trans. Comput. 57 (10) (2008) 1315–1330.

[3] E. Baranov, S. Bliudze, A note on the expressiveness of BIP, in: Proceedings Combined 23rd International Workshop on Expressiveness in Concurrency and 13th Workshop on Structural Operational Semantics, EXPRESS/SOS 2016, in: EPTCS, vol. 222, 2016, pp. 1–14.

[4] S. Bliudze, J. Sifakis, A notion of glue expressiveness for component-based systems, in: CONCUR 2008, Springer, 2008, pp. 508–522.

[5] S. Bliudze, A. Mavridou, R. Szymanek, A. Zolotukhina, Coordination of software components with BIP: application to OSGi, in: Proceedings of the 6th International Workshop on Modeling in Software Engineering, MiSE 2014, ACM, New York, NY, USA, 2014, pp. 25–30.

[6] S. Bliudze, A. Mavridou, R. Szymanek, A. Zolotukhina, Exogenous coordination of concurrent software components with JavaBIP, Softw. Pract. Exp. (2017), http://dx.doi.org/10.1002/spe.2495.

[7] S. Bliudze, J. Sifakis, M.D. Bozga, M. Jaber, Architecture internalisation in BIP, in: Proceedings of the 17th International ACM SIGSOFT Symposium on Component-based Software Engineering, CBSE '14, ACM, 2014, pp. 169–178.

[8] G.D. Plotkin, A Structural Approach to Operational Semantics, Tech. Rep. DAIMI FN-19, University of Aarhus, 1981, http://citeseer.ist.psu.edu/plotkin81structural.html.

[9] R. Milner, Communication and Concurrency, Prentice Hall International Series in Computer Science, Prentice Hall, 1989.

[10] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall International Series in Computer Science, Prentice Hall, 1985.

[11] R. Edelmann, Behaviour–Interaction–Priority in Functional Programming Languages: Formalisation and Implementation of Concurrency Frameworks in Haskell and Scala, Master's thesis, Ecole polytechnique fédérale de Lausanne (EPFL), Jan. 2015, http://infoscience.epfl.ch/record/215767.

[12] S. Marlow (Ed.), Haskell 2010 Language Report, Haskell.org, 2010, https://www.haskell.org/definition/haskell2010.pdf.

[13] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, An Overview of the Scala Programming Language, Tech. Rep., EPFL, 2004, http://infoscience.epfl.ch/record/52656.

[14] M. Bozga, M. Jaber, N. Maris, J. Sifakis, Modeling dynamic architectures using Dy-BIP, in: Software Composition, SC 2012, in: LNCS, vol. 7306, Springer, 2012, pp. 1–16.

[15] P. Wadler, Monads for functional programming, in: Advanced Functional Programming, Springer, 1995, pp. 24–52.

[16] B. Yorgey, The Typeclassopedia, The Monad.Reader 13 (2009) 17–68.

[17] P. Haller, M. Odersky, Scala actors: unifying thread-based and event-based programming, Theor. Comput. Sci. 410 (2) (2009) 202–220.

[18] J. Launchbury, T. Elliott, Concurrent orchestration in Haskell, SIGPLAN Not. 45 (11) (2010) 79–90, http://dx.doi.org/10.1145/2088456.1863534.

[19] K. Donnelly, M. Fluet, Transactional events, SIGPLAN Not. 41 (9) (2006) 124–135, http://dx.doi.org/10.1145/1160074.1159821.

[20] R. Bruni, H.C. Melgratti, U. Montanari, Behaviour, interaction and dynamics, in: Specification, Algebra, and Software – Essays Dedicated to Kokichi Futatsugi, in: LNCS, vol. 8373, Springer, 2014, pp. 382–401.

[21] C. Di Giusto, J.-B. Stefani, Revisiting glue expressiveness in component-based systems, in: COORDINATION 2011, Springer, 2011, pp. 16–30.

[22] P. Inverardi, A.L. Wolf, Formal specification and analysis of software architectures using the chemical abstract machine model, IEEE Trans. Softw. Eng. 21 (4) (1995) 373–386.

[23] D. Le Métayer, Describing software architecture styles using graph grammars, IEEE Trans. Softw. Eng. 24 (7) (1998) 521–533.

[24] J. Magee, J. Kramer, Dynamic structure in software architectures, SIGSOFT Softw. Eng. Notes 21 (6) (1996) 3–14, http://dx.doi.org/10.1145/250707.239104.

[25] J.S. Kim, D. Garlan, Analyzing architectural styles, J. Syst. Softw. 83 (7) (2010) 1216–1235.

[26] I. Georgiadis, J. Magee, J. Kramer, Self-organising software architectures for distributed systems, in: Self-Healing Systems, ACM, 2002, pp. 33–38.

[27] M.H. Kacem, M. Jmaiel, A.H. Kacem, K. Drira, Evaluation and comparison of ADL based approaches for the description of dynamic of software architectures, in: ICEIS, vol. 3, 2005, pp. 189–195.

[28] M. Belguidoum, F. Dagnat, Dependency management in software component deployment, Electron. Notes Theor. Comput. Sci. 182 (2007) 17–32.