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/

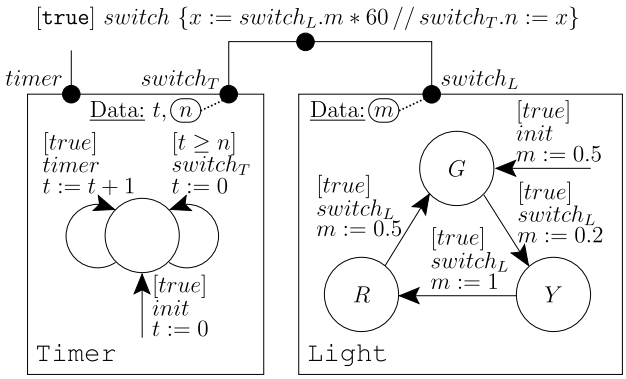

Fig. 1. Traffic light in BIP.

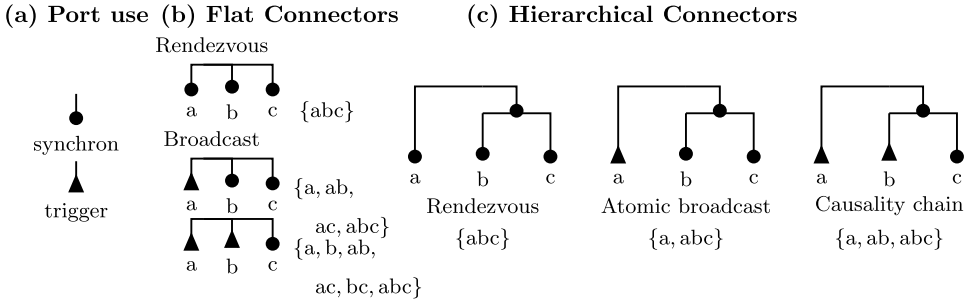

Fig. 2. Flat and hierarchical BIP connectors.

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.