BPFlow: A Graphical Model-Driven Engineering Language for Reactive Systems

November 27, 2022

Abstract

1 Introduction

2 An example-driven demo

3 The proposed language

3.1 Syntax

3.2 Semantics

4 A proof-of-concept tool

5 Case Studies

5.1 Tic-Tac-Toe

5.2 Snake

5.3 Level Crossing

5.4 CASIO Watch

5.5 Smart LSC Home

6 Evaluation

6.1 Compare to diagrammatic BP-languages

6.2 Compare to diagrammatic non-BP-languages

7 The proposed modeling language

In this section, we show how flow diagrams can be used to define sequences of events by describing the flow of information units called tokens through nodes. The basic running cycle of a diagram is as follows:

1. Tokens are placed at the initial nodes according to the parameters of these nodes.
2. A cycle begins driven by nodes that change, delete, and duplicate tokens and transfer them to other nodes through ports and to sync. The cycle ends when all the tokens are in sync.
3. Once all tokens arrive in sync, an event is selected based on the requests and the blocking specified by the tokens. The selected event is triggered, and tokens that wait for it are released to the nodes.
4. The last two steps are repeated until there is no event to select.

We show in this section how we define building blocks and use them to describe diagrams that specify the processes of token creation, updates, and transfer between the nodes and sync. The semantics of a diagram is, thus, formally defined as the set of sequences of events that could have been generated. This establishes a new way for defining regular languages that take inspiration from automata composition techniques such as behavioral programming and data flow languages such as MATLAB Simulink.

7.1 Defining node and container libraries through types

We start with two definitions that describe how we define "types" that we will later use as templates for nodes and containers in diagrams. Our intention in this definition structure is to allow an expandable library of types with user-defined meanings. To enable future extensions, the definitions are built in a way that will let us define the semantics of a general diagram abstractly, separately from the meaning of the specific semantics for each type.

We first define the concept of the node type, which serves as a template of a node that modelers can drag into a diagram. The definition of the type includes its name, the names of the exit ports from it, a set of initial tokens, two functions that define how it changes tokens and passes them on, and two procedures that are hooks that enable context-dependent behavior, as we will expand on later.

Definition 1. *A node is a dictionary with the following fields:*

- $NAME$
- $PORTS$
- $INITIAL\_TOKENS$

- *Functions that depend on the type of the node:*
  - $tokensToSendForward\colon 2^{TOKEN} \times PORT \to 2^{TOKEN}$
  - $tokensToSync\colon 2^{TOKEN} \to 2^{TOKEN}$
- *Procedures that depend on the type of the node and its containment hierarchy:*
  - beforeEventSelection($diagram\_state$, $diagram\_vertex$)
  - afterEventSelection($diagram\_state$, $diagram\_vertex$, $event$)

The role of the first three fields is to define the static structure of the node. The *tokensToSendForward* function defines how a node of the specified type forwards tokens to its output ports. It takes, as a first parameter, the set of tokens that are currently positioned at a node of a certain type and a port identifier and returns a set of tokens (not necessarily a subset) that the node passes on the port. The *tokensToSync* function maps the set of tokens at the node to tokens that it syncs (again, not necessarily a subset). As said above, we will discuss the role of the beforeEventSelection and afterEventSelection later.

The following definition is designed to enable container libraries. The role of the containers is to change the behavior of the nodes placed in them through a hierarchical mechanism defined by the beforeEventSelection and afterEventSelection functions.

Definition 2. *A container is a dictionary with the following fields:*

- NAME
- beforeEventSelection($diagram\_state$, $diagram\_vertex$)
- afterEventSelection($diagram\_state$, $diagram\_vertex$, $event$)

Specifically, these procedures are called before and after each event selection when all tokens are in sync. The parameters for the procedures contain complete information about the context in terms of inclusion in the containers of the node and its contents and the collection of tokens that are placed in it at a given moment. The goal is to enable a stack of use in order of containment so that the procedures of each container invoke the procedures of the container that contains them. This allows us to define containers with a lateral effect on all the nodes and containers in them.

7.2 Diagrams and states

Using the types defined with the tools described in the previous section, we can, in this section, define diagrams and diagram states. A diagram is a graph whose nodes are mapped to types and contained in containers that can be contained within each other:

Definition 3. *A diagram is a dictionary with the following fields:*

- $G = (V, E)$ *is a directed graph.*
- C *is a set of containers.*
- PLACE: $V \cup C \to C \cup \{\perp\}$
- TYPE: $C \cup V \to \text{CONTAINER\_TYPE} \cup \text{NODE\_TYPE}$

The set C defines the containers in the diagram. The $PLACE$ function tells in which container each node and each container is placed. This function can map some elements to $\perp$ to express that these elements are not in any container (they are in the main, hidden container of the diagram). The TYPE function determines the type of each node and each container in the diagram.

Next, we define a diagram state that will be used later to define semantics as a series of states and events. In particular, a state contains a diagram and two functions describing the tokens present at each node and each node's sync.

Definition 4. *A diagram-state is a dictionary with the following fields:*

- DIAGRAM *is an instance of a diagram.*
- TKN: $V \to 2^{TOKEN}$
- SYN: $V \to 2^{TOKEN}$

7.3 Hierarchy based pre- and post- event-selection state manipulation

Given a state and a vertex, two procedures that may change the state of the diagram are defined hierarchically, according to the node's position in the diagram:

$$v.\mathit{beforeEventSelection}() = \mathit{state}.\,\text{DIAGRAM}\,.\,\text{TYPE}(v).\mathit{beforeEventSelection}(\mathit{state}, v)$$

and

$$v.\mathit{afterEventSelection}(\mathit{event}) = \mathit{state}.\,\text{DIAGRAM}\,.\,\text{TYPE}(v).\mathit{afterEventSelection}(\mathit{state}, v, \mathit{event}).$$

These procedures will be called while all tokens are in sync before and after selecting an event. Since the procedures know the containment hierarchy, they will run the procedures of the containers according to the hierarchy. This allows containers to have a lateral effect on all the containers and nodes contained in them, as we will demonstrate latter.

The way we achieved the lateral effect is by connecting bout hierarchy of object inheritance and decorator object. Each node in the graph inherent from the base node, or from her sons, so in the inherent stuck call the base node function will call, The function of the base node is as given in the next algorithm:

In line 2, we see the connection between the two inherent type, from object inherent we switch to place inherent.

7.4 Execution

The proposed language - Semantics:

The main algorithm[1] start the run of the BP-program by creating all the b-thread, one for each token that start on node of type start. The function RunInNewBT[2]create a b-thread for the token she as an argument, and call the functionality of running a token in node, RunInSameBT[3], thet run the execute function of the node.

Algorithm 1: Run diagram

```
  Input: diagram,bsync
1 $startNodes \leftarrow \{s \in diagram.states\colon s.type = \text{“}start\text{”}\}$
   $state \leftarrow getInitialState(diagram)$
2 Print $state$
3 For Each $node \in startNodes$ :
4     For Each $token \in node.initial\_token$ :
5         RunInNewBT(node,token,bsync)
```

Algorithm 2: RunInNewBT

```
  Input: node,context,bsync
1 bp.registerBThread(()→ RunInNewBT(node, copy(context), bsync))
```

We can see that at RunInSameBT[3] we wrapped the execute function with try-catch mechanism, the purpose of the try-catch mechanism will explained in the section about the region.

Algorithm 3: RunInSameBT

```
  Input: node,context,bsync
1 followers← $\emptyset$
2 try:
3     followers← node.execute(context,bsync)
4 catch move_exception:
5     follower ← move_exception.followers
6 goToFollowers(node,context,bsync,followers)
```

Algorithm 4: goToFollowers

```
Input: node,context,bsync,followers
1 $tickets \leftarrow \{\langle n, token\rangle : \langle n, tokens\rangle \in followers \wedge token \in tokens\}$
2 For $i \leftarrow 1;\ i < |tickets|;\ i \leftarrow i+1$ :
3     $\langle n, ctx\rangle tickets[1]$
4     RunInNewBT(n,ctx,bsync)
5 if $|tickets| >= 1$ then
6     $\langle n, ctx\rangle tickets[1]$
7     RunInSameBT(n,ctx,bsync)
```

8 the node library

The build of the nodes and make them versatile and modifiable for adding sets of functionality for the language, e.g., the node libraries. Under the role that each node define as one function, execute.

8.1 The base library

The base library gives the language the basic operators for demonstrating a flow of the system, loops, if-else, sync-node...

Definition 5. *Start node*

The start node use to start the flow of the token the equivalent of the b-thread in BP-language.

Algorithm 5: Start-Node: execute

```
Input: node, token, bsync
Output: followers
1 $followers \leftarrow \{\langle n, token\rangle : n \in diagram.sucssesor(node)\};$
2 return $followers$
```