Chapter 8. BDF Domain

Authors: *Joseph T. Buck*

Other Contributors: *Edward A. Lee*

8.1 Introduction

Boolean-controlled dataflow (BDF) is a domain that can be thought of as a generalization of synchronous dataflow (SDF). It supports dynamic flow of control but still permits much of the scheduling work to be performed at compile time. The dynamic dataflow (DDF) domain, by contrast, makes all scheduling decisions at run time. Thus, while BDF is a generalization of SDF, DDF is still more general. Accordingly, the BDF domain permits SDF actors to be used, and the DDF domain permits BDF actors to be used. This chapter will assume that the reader is familiar with the SDF domain.

The BDF domain can execute any actor that falls into the class of Boolean-controlled dataflow actors. For an actor to be SDF, the number of particles read by each input porthole, or written by each output porthole, must be constant. Under BDF, a generalization is permitted: the number of particles read or written by a porthole may be either a constant or a two-valued function of a particle read on a control porthole for the same star. One of the two values of the function must be zero. The effect of this is that a porthole might read tokens only if the corresponding control particle is zero (`FALSE`) or nonzero (`TRUE`). The control porthole is always of type integer, and it must read or write exactly one particle. Although the particles on the control porthole are of integer type, we treat them as Booleans, using the C/C++ convention that zero is false and nonzero is true.

We say that a porthole that conditionally transfers data based on a control token is a conditional porthole. A conditional input porthole must be controlled by a control input. A conditional output porthole may be controlled by either a control input or a control output. These restrictions permit the run-time flow of control to be determined by looking only at the values of particles on control ports. The compile-time scheduler determines exactly how the flow of control will be altered at run time by the values of these particles. It constructs what we call an *annotated schedule*, which is a compile-time schedule where each firing is annotated with the run-time conditions under which the firing should occur.

The theory that describes graphs of BDF actors and their properties is called the token flow model. Its properties are summarized in [Buc93b] and developed in much more detail in [Buc93c].

The BDF scheduler performs the following functions. First, it performs a consistency check analogous to the one performed by the SDF scheduler to detect certain types of errors corresponding to mismatches in particle flow rates [Lee91a]. Assuming that no error is detected, it then applies a clustering algorithm to the graph, attempting to map it into traditional control structures such as if-then-else and do-while. If this clustering process succeeds in reducing the entire graph to a single cluster, the graph is then executed with the quasi-static schedule corresponding to the clusters. (It is not completely static since some actors will be conditionally executed based on control particle values, but the result is "as static as possible.") If the clustering does not succeed, then the resulting clusters may optionally be executed by the same dynamic scheduler as is used in the DDF domain. Dynamic execution of clusters is enabled or disabled by setting the "allowDynamic" parameter of the default-BDF target.

8.2 The default-BDF target

At this time, there is only one BDF target. The parameters of the target are:

logFile (`STRING`) Default =
The default is the empty string. The filename to which to report various information about a run. If this parameter is empty (the default), there will be no reporting. If the parameter is "<cerr>" or "<cout>", messages will go to the Unix standard error or standard output, respectively.

allowDynamic (`INT`) Default = `NO`
If `TRUE` or `YES`, then dynamic scheduling will be used if the compile-time analysis fails to completely cluster the graph. As shown in [Buc93c], there will always be some valid graphs that cannot be clustered.

requireStronglyConsistent
(`INT`) Default = `NO`
If `TRUE` or `YES`, then a graph will be rejected if it is not "strongly consistent" [Lee91a]. This will cause some valid systems, even systems that can be successfully statically scheduled, to be rejected.

schedulePeriod (`FLOAT`) Default = `10000.0`
This defines the amount of time taken by one iteration of the BDF schedule. The notion of "iteration" is defined in the SDF chapter, in the section 5.1.3.

8.3 An overview of BDF stars

The "open-palette" command in pigi ("O") will open a checkbox window that you can use to open the standard palettes in all of the installed domains. At the current time, the BDF star library is small enough that it is contained entirely in one palette, shown in figure 8-1.

`CondGate` If the value on the "control" input is nonzero, the input particle is copied to output. Otherwise, no input is consumed (except the control particle) and no output is produced. This is effectively one half of a `Select`.

`Fork` (Two icons.) Copy the input particle to each output. The SDF fork is not used here because the BDF domain requires some extra steps to assert that each output of a fork is logically equivalent if the input is a Boolean signal.

`Not` Output the logical inverse of the Boolean input. Again, the equivalent SDF logic block is not adequate because extra steps are needed to assert the logical relationship between the input and the output.

`Select` If the value on the “control” porthole is nonzero, *N* tokens (from the parameter *N*) from “trueInput” are copied to the output; otherwise, *N* tokens from “falseInput” are copied to the output.

`Switch` Switch input particles to one of two outputs, depending on the value of the control input. The parameter *N* gives the number of particles read in one firing. If the particle read from the control input is `TRUE`, then the values are written to “trueOutput”; otherwise they are written to “falseOutput”.

The Higher Order Functions icon leads to the HOF palette that contains HOF stars that can be used within BDF.

8.4 An overview of BDF demos

The demos with icons shown in figure 8-2 illustrate Boolean-controlled dataflow principles. A useful way to understand these principles when running BDF demos is to display the schedule after a run. This can be done from pigi using the display-schedule command under

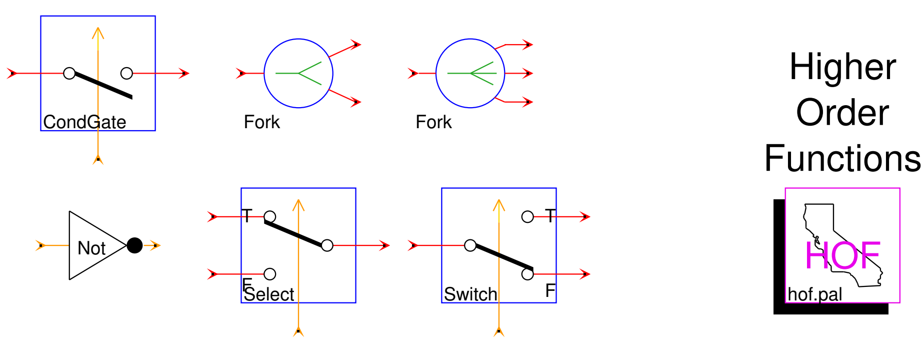

FIGURE 8-1: The palette of stars for the BDF domain. All SDF stars may also be used.

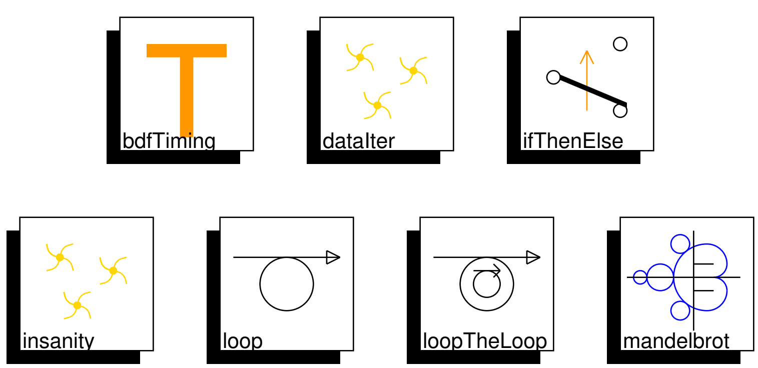

FIGURE 8-2: The BDF demos.

the Exec menu. It must be done before the control panel is dismissed, because dismissing the control panel destroys the scheduler.

`bdfTiming` This demo is identical to the DDF timing demo, except that it uses BDF `Switch` and `Select` stars instead of DDF `Case` and `EndCase`. The static schedule has some simple if-then constructs to implement conditional firing.

`dataIter` This simple system, which does nothing interesting, is surprisingly difficult to schedule statically. It requires nesting an if-then within a do-while within a manifest iteration.

`ifThenElse` This simple system uses `Switch` and `Select` stars to construct an if-then-else.

`insanity` This peculiar system applies two functions, log and cosine, but the order of application is chosen at random. The BDF clustering algorithm fails to complete on this graph. If the *allowDynamic* parameter of the target is set to `YES`, then the scheduler will construct four SDF subschedules, which must then be invoked dynamically.

`loop` This system illustrates the classic dataflow mechanism for implementing data-dependent iteration (a do-while). A sequence of integers (a ramp) is the overall input. Each input value gets multiplied by 0.5 inside the loop until its magnitude is smaller than 0.5. Then that smaller result is sent to the output.

`loopTheLoop` This system is similar to the `loop` demo, except that a second do-while loop is nested within the first.

`mandelbrot` This system calculates the Mandelbrot set and uses Matlab to plot the output. Matlab must be installed on the local workstation to view the output of this demo, or Matlab must be available on a machine that is accessible via the Unix rsh command. See “Matlab stars” on page 5-26 for more information.