Data Flow Programming

Data Flow Programming

Publication date 2019-06-07

Table of Contents

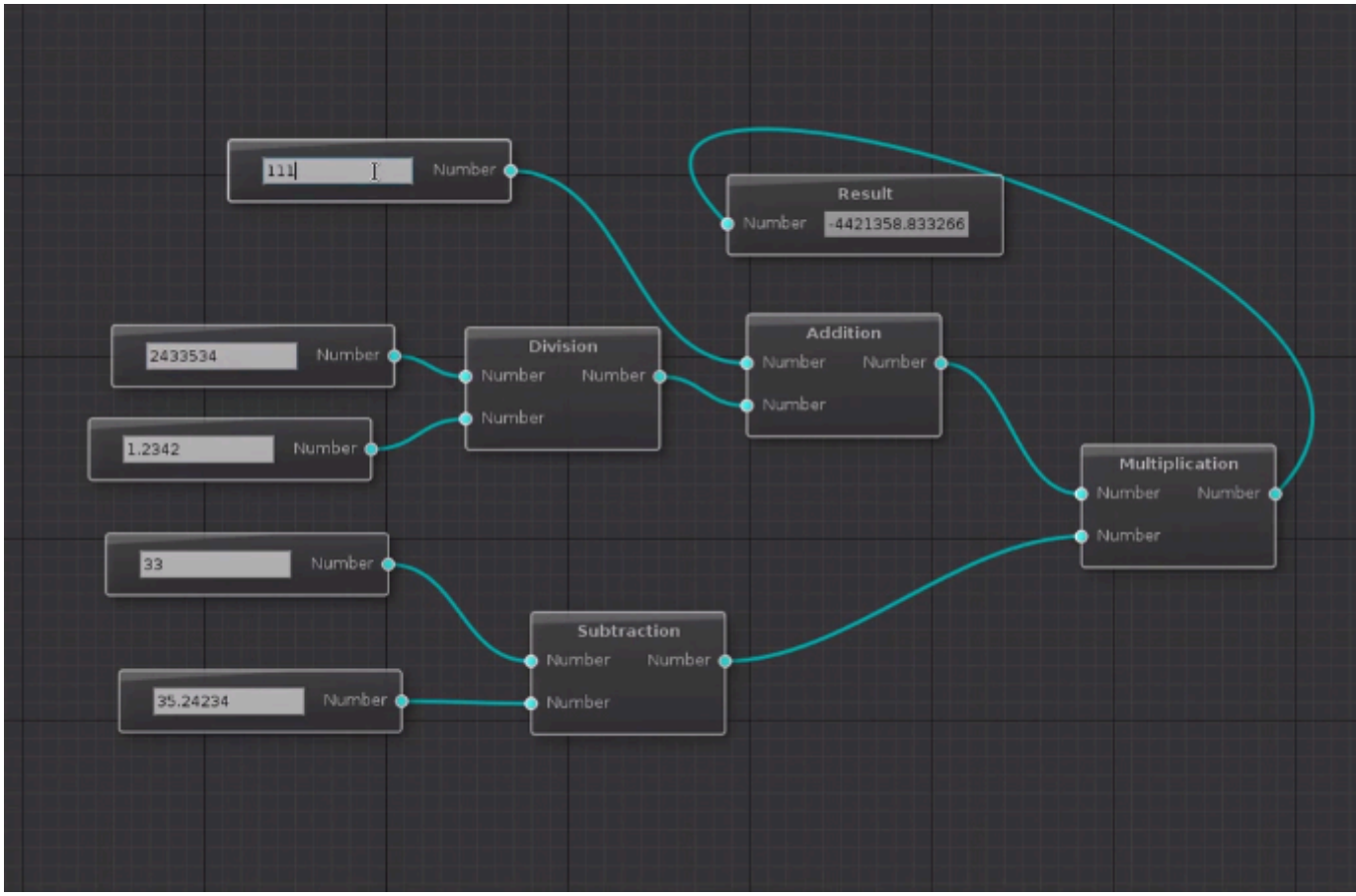
List of Figures

Chapter 1. Introduction

1. What is data flow?

- Business programming works with data and concentrates on how this data is transformed, combined and separated, to produce the desired outputs and modify stored data according to business requirements. Broadly speaking, whereas the conventional approaches to programming (referred to as "control flow") start with process and view data as secondary, business applications are usually designed starting with data and viewing process as secondary - processes are just the way data is created, manipulated and destroyed. We'll call this approach "data flow"
- More formally, speaking, data flow programming is a programming paradigm that models a program as a directed graph of the data flowing between operations.Operations are called nodes and the links between them are called arcs.Operations are executed only when data is available to them

Figure 1.1. Calculator



- Dataflow programming was pioneered by Jack Dennis and his graduate students at MIT in the 1960s.

  Dataflow has always been closely related to hardware. It is essentially the same way electronic engineers think about circuits, just in the form a programming language. There have been many attempts to design processors based on dataflow as opposed to the common Von Neumann architecture. MIT’s Tagged Token architecture, the Manchester Prototype Dataflow Computer, Monsoon and The WaveScalar architecture were all dataflow processor designs. They never gained the popularity that Intel’s Von Neumann microprocessors did, not because they wouldn’t work, but because it was impossible for them to keep pace with the ever increasing clock speeds that Intel, Zilog and others mass market manufactures were able to provide.

- Samples of data flow programming:

- Spreadsheets
- Unix pipes
- Reactive programming
- Futures and promises
- Flow based programming
- ETL

2. Control Flow vs Data Flow

- In control flow programming focus is on process and data is viewed as secondary while in data flow programming focus is on data and process is viewed as secondary - processes are just the way data is created, manipulated and destroyed.
- Control Flow Programming is related to von Neumann architecture which that has evolved to mean any stored-program computer in which an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is referred to as the von Neumann bottleneck and often limits the performance of the system

Figure 1.2. Von Neumann Architecture

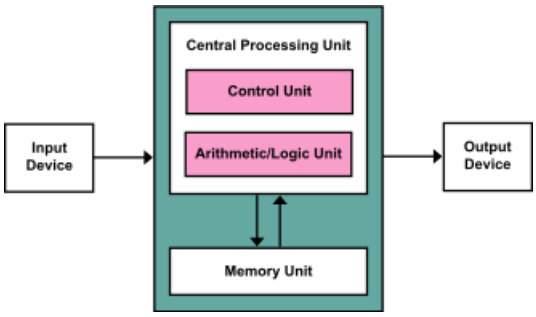
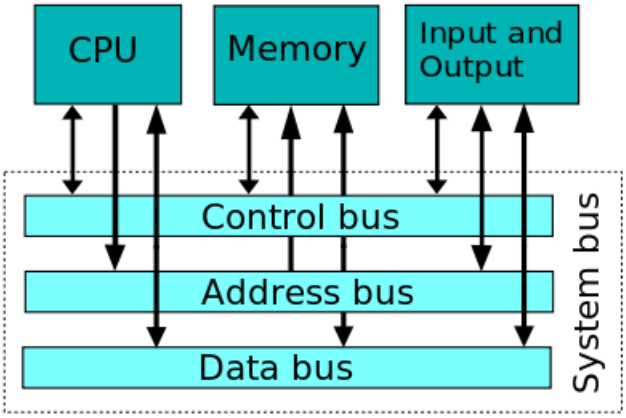


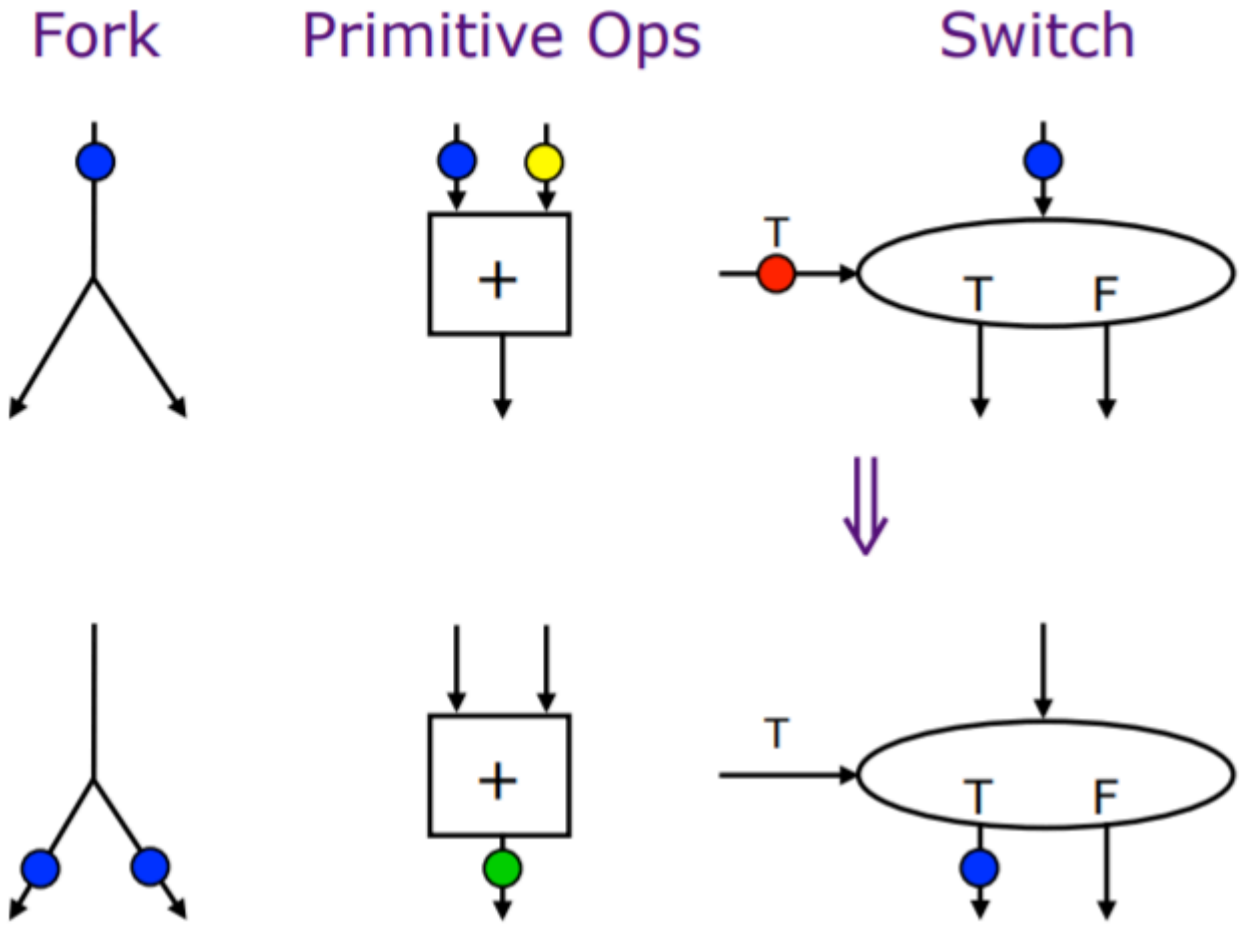
Figure 1.3. System Bus



- In a von Neumann machine, we have a stream of instructions which operate on external data. Conditional execution, jumps and function calls change the instruction/operation stream to be executed. This could be seen as instructions flowing through data (eg instructions operate on registers which are loaded with data by instructions - the data is static unless the instruction stream moves it). A control flow "IF" statement jumps to the correct branch in the instruction stream, but the data does not get moved.

- The shared memory design of the Von Neumann architecture poses no problems for sequential, single threaded programs. Parallel programs with multiple components trying to access a shared memory location, on the other hand, requires the use locks, semaphores, mutexes and other coordination/synchronization methods.
- In a data flow machine, you have a stream of data which is passed from instruction to instruction to be processed. Conditional execution, jumps and procedure calls route the data to different instructions. This could be seen as data flowing through otherwise static instructions like how electrical signals flow through circuits or water flows through pipes. A data flow "IF" statement would route the data to the correct branch.
- A small set of data flow operators can be used to define a general programming language:

Figure 1.4. Dataflow Nodes

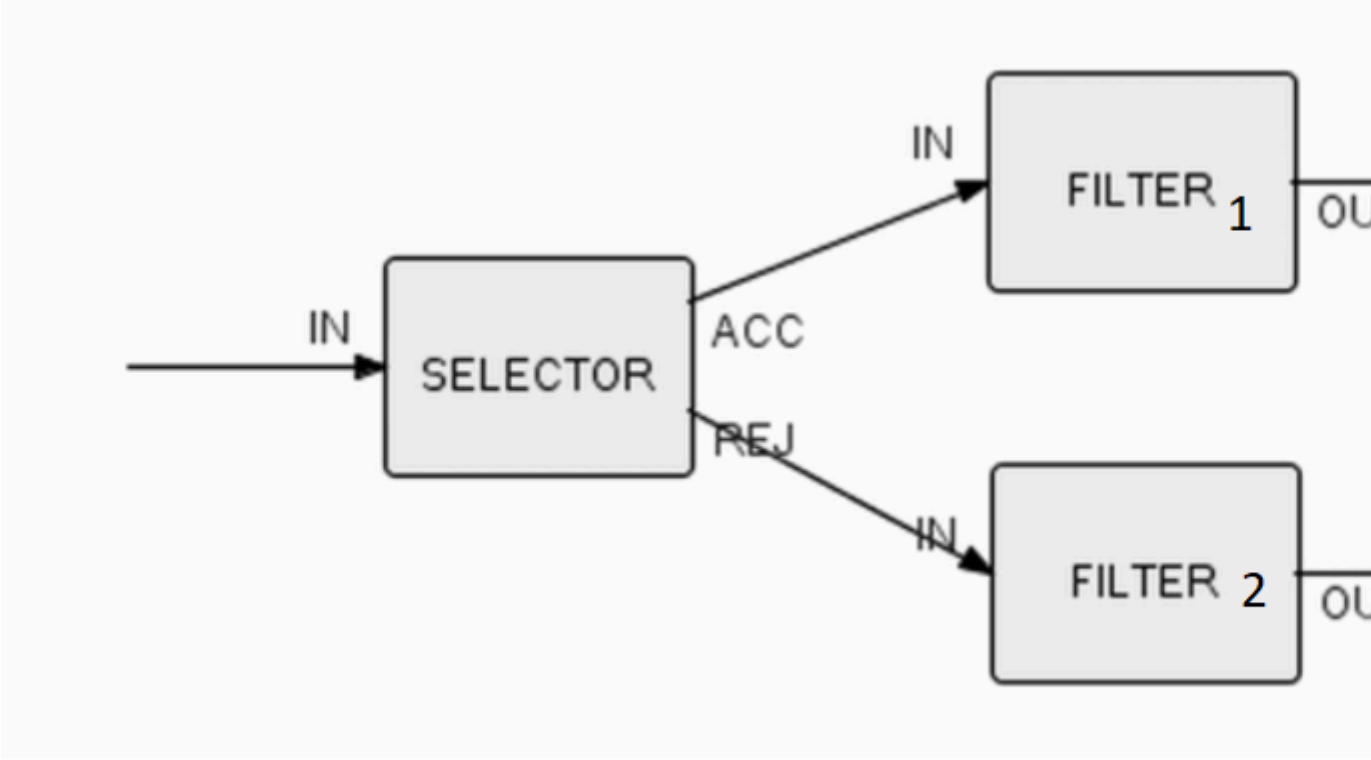


Chapter 2. Dataflow Architecture

1. Nodes

- A “node” is a processing element that takes inputs, does some operation and returns the results on its outputs.We can see a node as a black box where the only way to interact with it is through the inputs and outputs.
- A node is a unit of computation. The actual computation doesn’t matter because data flow is primarily concerned about moving data around.
- The only way for a node to send and receive data is through “ports.” A port is the connection point between an arc and a node.The ports are the only view we have inside the box.
- When nodes have multiple ports, they are given names to distinguish one from another.
- A node can be seen as a named instance of a "Component" or Component type, and a running node will be called "Process". We can have many nodes of the same type in data flow program.

Figure 2.1. Component Instances



- Nodes that have only input ports are called output nodes
- Nodes that have only output ports are called input nodes or Sink Nodes
- Basic functionality of a node: There are three basic operations a node can implement: "receive", "send" and "drop". A node that receives a packet has to send or drop the packet. There is no automatic disposal (No Garbage Collector)

  receive from IN using a

  if c is true

send a to OUT

else

drop a

endif

Note

A sink node must have a "create" functionality. It has to create nodes in order to feed the other down-stream nodes.

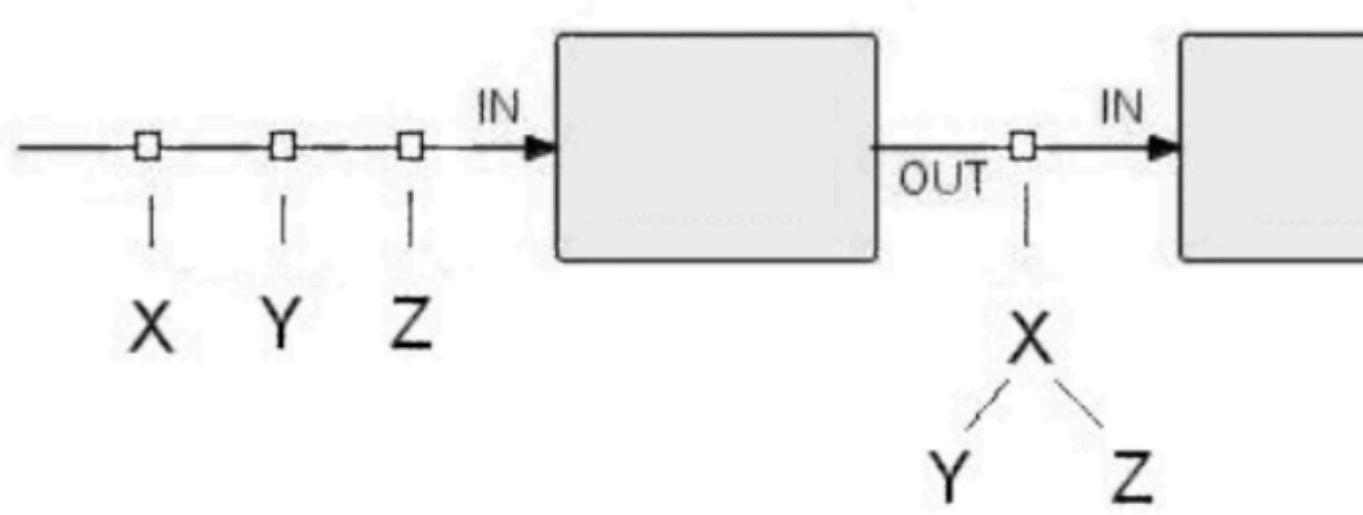
A sink node will have "create" functionality

- Sample Nodes:
  - MySql_RecordReader
  - Transformers
  - SugarCRM_BeanWriter

2. Data

- Data is treated the same without regard to its value or type
- Data is referred to as “tokens" or packets also known as Information Packets (IPs) (in Flow Based Programming)
- Usually an IP has a header and a payload. The header contains information about packet type and payload, usually expressed as attributes. The payload may contain any type data starting from primitive data like numeric, char, numbers to more structured data like records or trees.
- There are two types of Information Packets
  - Control Packets. They provide information about the data in the stream, like grouping or End of Data
  - Data Packets. They actually contain data that will processed by nodes
- The layouts of all IP types that a component can handle and the IP types that it can generate become part of the specification of that component.
- There can be nodes that can change the layout of a packet

Figure 2.2. Tree Structures



- The environment must provide a Runtime Type System compatible with Type Systems from all accessible data sources.
- Mutability of the data is important because it affects the semantics of the data flow implementation.The traditional and prudent choice is for data to be immutable, especially if parallelism is important.
- Samples:

3. Arcs

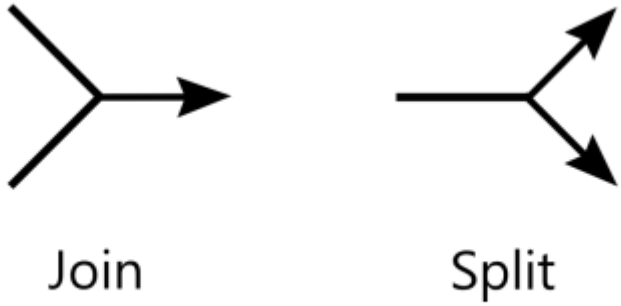
- Arcs are paths for data to flow between nodes.They are also called channels, connections or links.
- Data typically flows from the output port to the input port of nodes. Although it is very uncommon, the reverse is also possible (Push or Pull).
- We can think of an arc as a pipe/queue that can only hold so much data at any one time.
- The amount of data that an arc can hold is called the “arc capacity.”
- Sample Arcs:
  - Concurrent Queues
  - SEDA (Staged Event Driven Architecture) Queues. See Camel SEDA Components
  - Persistent Queues (Database Tables, Files)
  - External Queues (Rabbit MQ etc)

4. Ports

- A port is the connection point between an arc and a node
- Usually a port has a unique name with respect to a component

- A port has two states: Closed and Open
- Sometimes ports can be numbered.
- An input port of a component can be connected to many output ports of another component (join) and the reverse(split).

Figure 2.3. Simple Join



An arc join and split

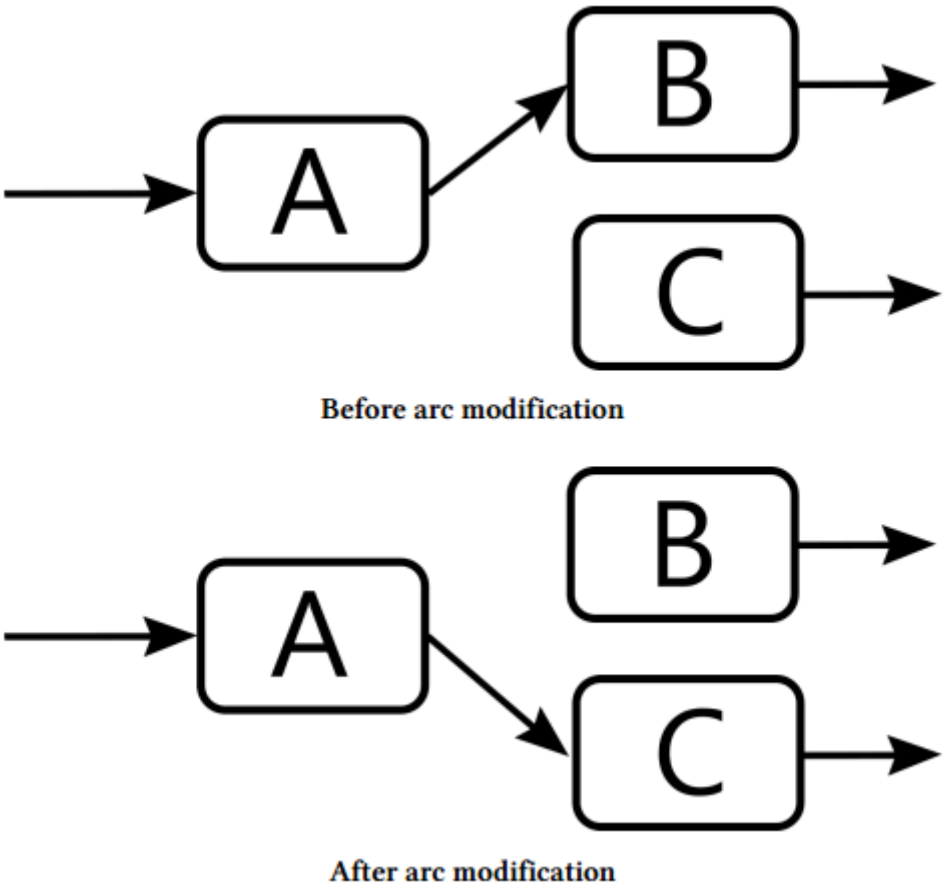
5. Network

- A data flow network is a directed graph containing nodes and the arcs between them.
- A network represents a data flow program
- Dynamic vs Static Networks.
  - Dynamic. Changes in the graph and/or node definitions can be made at run-time.

    Many programming frameworks and almost all interpreted languages, can load logic at runtime.(Java, C#, Php etc), or they allow functions to accept other functions as arguments (we call these higher-order-functions). Dynamic data flow allows us to do the same thing.

    For example, we could design a generic filter node and specialize its filtering at run-time by passing it another node that has the same interface (input and output ports).

Figure 2.4. Node Replacement



- Static. Graph and node definitions cannot be changed at run-time.

  The graph and the nodes are fixed. Its the same situation with compiled languages like C or C++.

  A special case of static data flow is Static Dataflow in Hardware

6. Executing a Network. The Scheduler

- The subsystem responsible with network execution is called Scheduler. In case of distributed network we can have a set of peer schedulers.
- Node execution is called "Activation" or "Firing" .
- Activation rules:
  - Preconditions: A Node is activated when there are data packets at least at one input port and there is space on the output channels(Arc capacity has not been reached). There might be cases when a Node can be fired in other conditions (IPs at all of its input ports or other conditions). In that case we have a custom activation rule.
  - Nodes without input ports(Sink Nodes) are activated by default at system initialization.
  - Once activated, a node can maintain control until it consumes all input packets(greedy) or it may decide to yield after each packet.

- Synchronous vs Asynchronous Activation. They are two ways to do the same thing: activate nodes in time. The asynchronous method has no preexisting knowledge of what's going to happen and just manages the now. While the synchronous method is very ordered, structured and plans everything first.

  - Asynchronous. A a node fires any time that its activation preconditions are met, possibly at the same time as other nodes.

    Asynchronous activation it is often combined with an arc capacity > 1. Nodes may have bursts of activity and thus the IP production rate may go up and down. If there is not enough space on the output arcs the node can't activate. Arcs need to have a capacity > 1 to buffer these bursts of IPs.

  - Synchronous.Synchronous execution is when the nodes fires on a pre-calculated, fixed schedule. The schedule determines which nodes have to fire before others, how often they have to fire and which nodes can fire simultaneously, if any.

  - Hybrid.Asynchronous and a synchronous engine combined in one package.

- Executing nodes or processes may thus be thought of as simple machines which can be in one of a small number of different "run states":

  - **Not yet initiated**. It means that the process has never received control. All processes start off in this state at the beginning.

  - **Terminated**. It means that the process will never receive control again. This can only happen if all of a process's upstream arcs/connections have been closed - each of the input connections of a process can be closed explicitly by that process, or it will be closed automatically if all of the processes feeding it have terminated

    The underlying idea here is that a process only becomes terminated if it can never be activated again. It can never be activated again if there is nowhere for more data IPs to come from. Note that, while a component's logic decides when to deactivate, termination is controlled by factors outside of the process. There is one exception to this: a component can decide not to be reactivated again, as a result of some error condition.

    There is a different way that a component can decide to terminate itself, and that is by closing its input ports. Some processes don't have input ports, but, for those that do, this has the same effect as terminating with a high return code.

    Sample: Suppose we have a Reader process which is reading a file or database table of a few million records, and a downstream process crashes: under normal data flow rules, the Reader keeps reading all the records, and sending them to its output port. As each send finds the output port closed, the Reader has to drop the undeliverable IP. So it has to read all the records, requiring a few million of "Create" and "Send" packet

    It is good programming style for components always to test for unsuccessful sends. If this condition is detected, they must decide whether to continue executing, or whether to just close down.

  - **Active**. A node/process is active when it has control and it executes its logic. If we had a separate CPU for each process, a normal Active state would be enough, although a component waiting to send to a full channel, waiting to receive from an empty one, or waiting on an external event, would have to spin waiting for the desired condition. To avoid this, we'll introduce a suspended state, so we can split the active state into:

    - Active normal

    - Active suspended

  - **Inactive**/**Idle**. When a process gives up control, by executing its "end" statement, or by explicitly doing a EXIT , or the equivalent, it "deactivates", and its state becomes "inactive".

- A simple sample:

Figure 2.5. Network Execution

Assume that initially there are no IPs on any of the arcs in the graph. If a single IP appears on the input arc of node A, then it can fire. The result is that it consumes the IP from the input, computes a new IP and puts it on the output.Now, node B has an IP available on its input channel, so it can fire. Node A doesn't have any more input IPs waiting, thus it goes into an idle/inactive state. Node B consumes the IP, computes a new one and puts it on the output.Node C follows the same execution pattern while node A and B are idle. Since no other IPs exist in the graph, none of the nodes can fire; therefore the execution of the graph is done.

Let's evaluate the graph again with one change. Instead of only putting a single IP on the input arc to A, this time let's supply it with a steady stream of data. Just as before, node A activates first, passing a new IP to B. Node B has data so it can fire just like before but now node A also has data so it can also fire.

Node A and B can both execute at the same time. Since the two nodes don't share any data we can safely execute them in parallel. Once B is done, it will put a new IP on its output for node C to consume. When node A is done, it will also put a new IP on the output arc for B to consume. The graph now has an IP on C's input, so node C can fire, an IP on B's input, so node B can fire, and since node A always has a constant stream of data on its input it can fire too. All three nodes can execute at the same time. This continues until we stop supplying data to node A.

- Push or Pull. Push and Pull refer to the way IPs move through the system.
  - Push - Producer is in control. With push, nodes send IPs to other nodes whenever they are available. The data producer is in control and initiates transmissions. Clicking on a form's summit button is an example of push.The browser initiates the conversation with the server and sends the form's data.
  - Pull - Consumer is in control. The consumer node must first request data from the producer. If the producer needs data from other upstream nodes it will also request data from those nodes. This continues until there is a producer node that doesn't need data from any other node. It will transmit its data to the requesting node. That consumer node can now send data to any node that requested data from it too. Pull is very rarely used in data flow systems but can be advantageous when a node's action is costly. It allows a node to lazily produce an output only when needed.
- .

7. Pipeline programming and a simple data flow implementation

- Pipeline Dataflow only has one input and one output port per node
- Network is executed sequencial.
- wTransit as a synchronous and sequencial data flow implementation. Task Calls will
  - Nodes have at most one input and output
  - Nodes are executed sequentially
  - There is only one Sink node per flow, called Row Generator, but this can be changed using subflows
  - Recursion is allowed
  - The IPs are tree based structures. There is no header or metadata available.
  - A custom activation Condition can be defined for each node using a DSL

- The network is dynamic. There are special types of nodes that can load a sub-network(sub-flow)
- The application model adds the concept of data connections used to interact with data from outside world.

Figure 2.6. Simple Flow

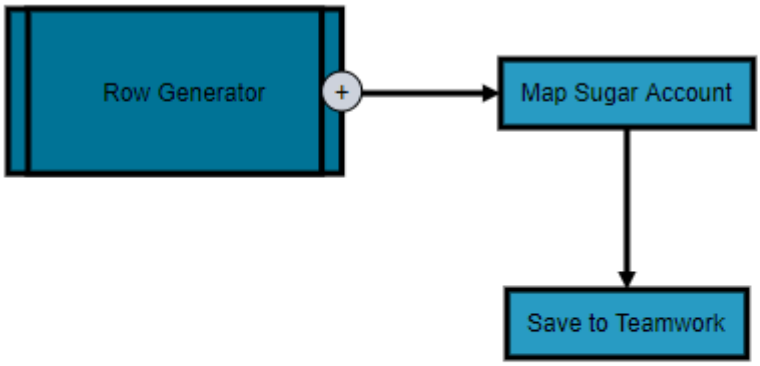
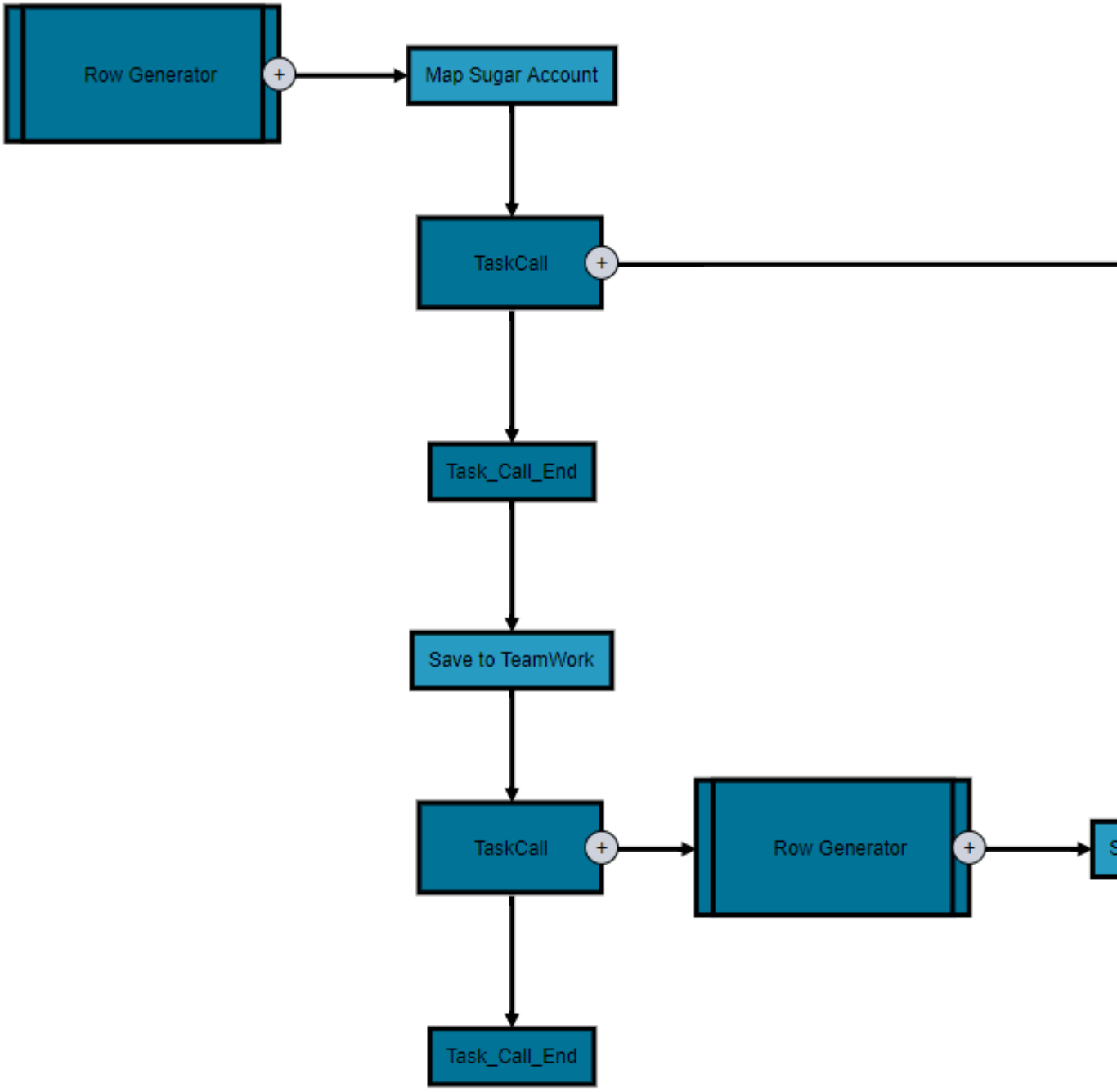


Figure 2.7. Flow with subflow



Add documentation....

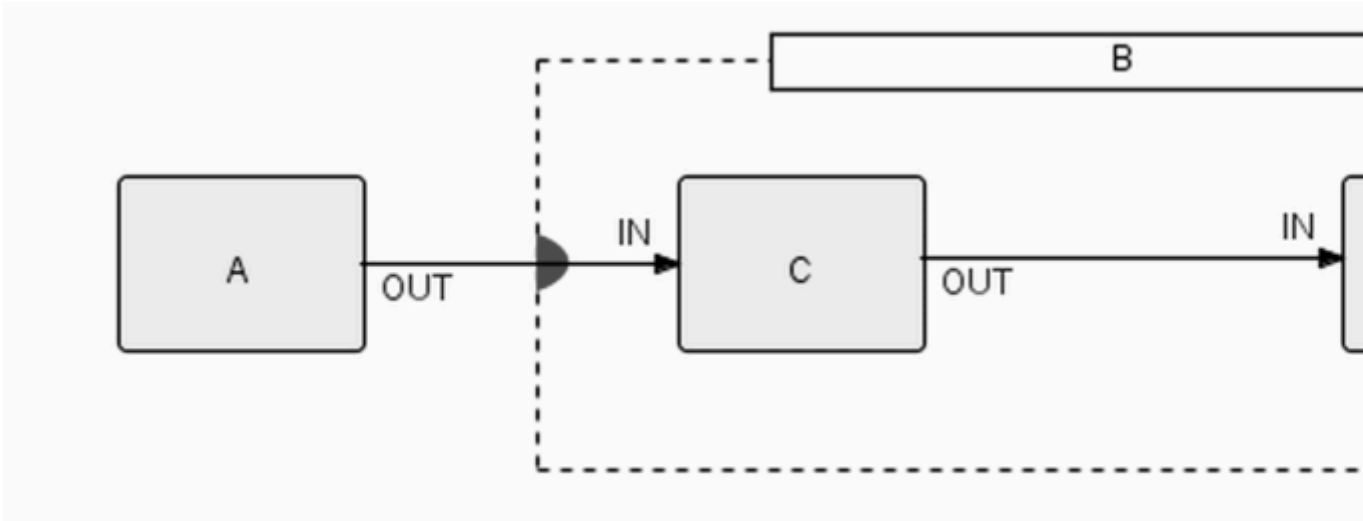
Chapter 3. Flow Based Programming and Dataflow

- Flow-Based Programming (FBP) is a version of data flow created by J. Paul Morrison in the early 1970s. It is a field-tested implementation that has spawned many recent developments such as NoFlo, Pypes, MicroFlow, DSPatch and many others. FBP is an asynchronous data flow model that allows for mutli-port and compound nodes.
- FBP makes use of both visual and textual representations. The textual form varies from one implementation to the next but is typically defined in the form of a domain specific language for graphs.
- FBP allow only arc joins. This is ensured by the queue-like functionality of an arc.

1. Composition

- A composite Node can be defined as a network of other nodes.
- Each free input port of the inner subnet should be exposed by the outer component
- There is no mandatory to expose all free out ports of the inner subnet the outer component

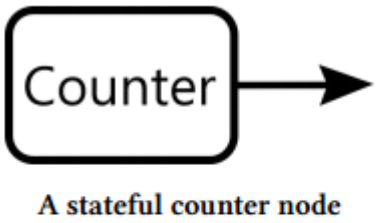
Figure 3.1. Composition



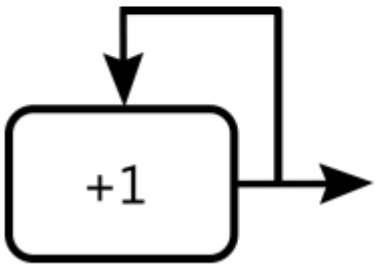
2. Node state

- Although this is not mandatory, nodes can have state that can be preserved between activations.
- A node that does not preserve his state between activations is a functional node.
- A node that preservers its state is a stateful node
- Any stateful node can be made into a functional node just by having an output arc that loops back to an input of the same node (a self arc, or self loop). Instead of storing state internally in the node, it would send its state data on the output so that in the next activation it can read it back in.

Figure 3.2. Stateful to Stateless



A stateful counter node



A stateless counter node

3. Special ports

- OPTIONS/PARAMTERS ports and IIPs.

Figure 3.3. Options Port - Configuration

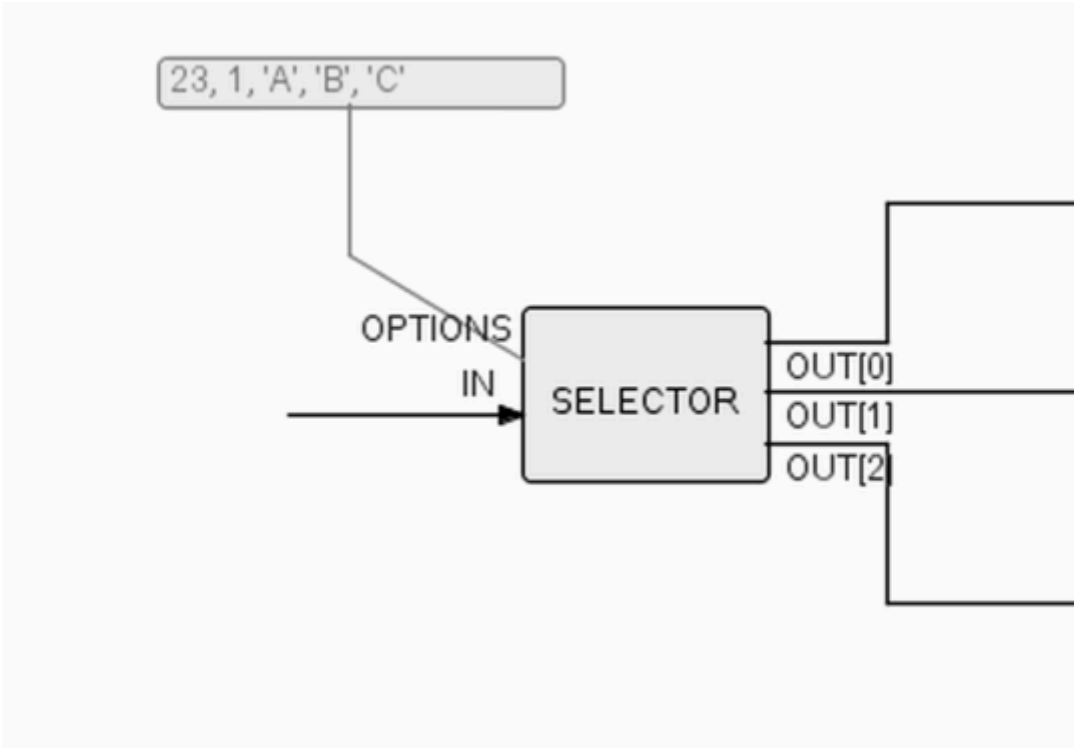
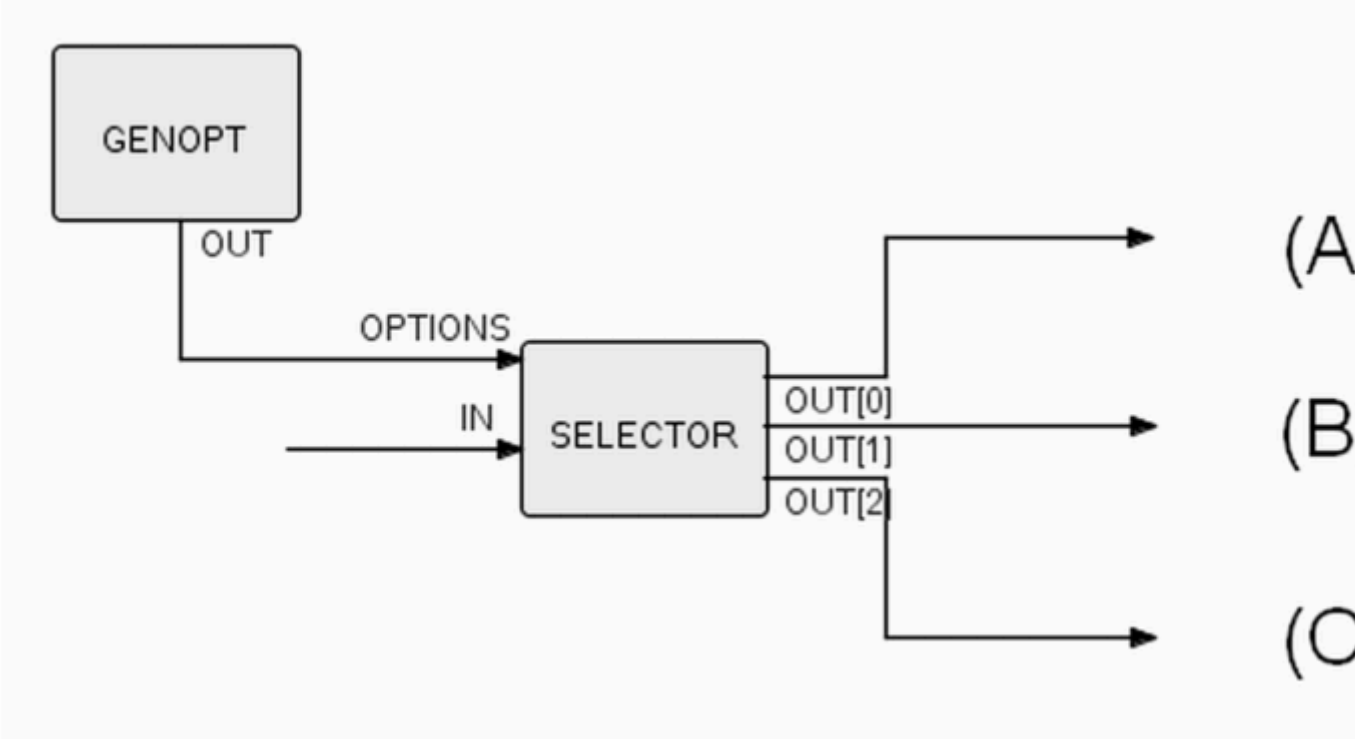


Figure 3.4. Options Port - Connected to other Component



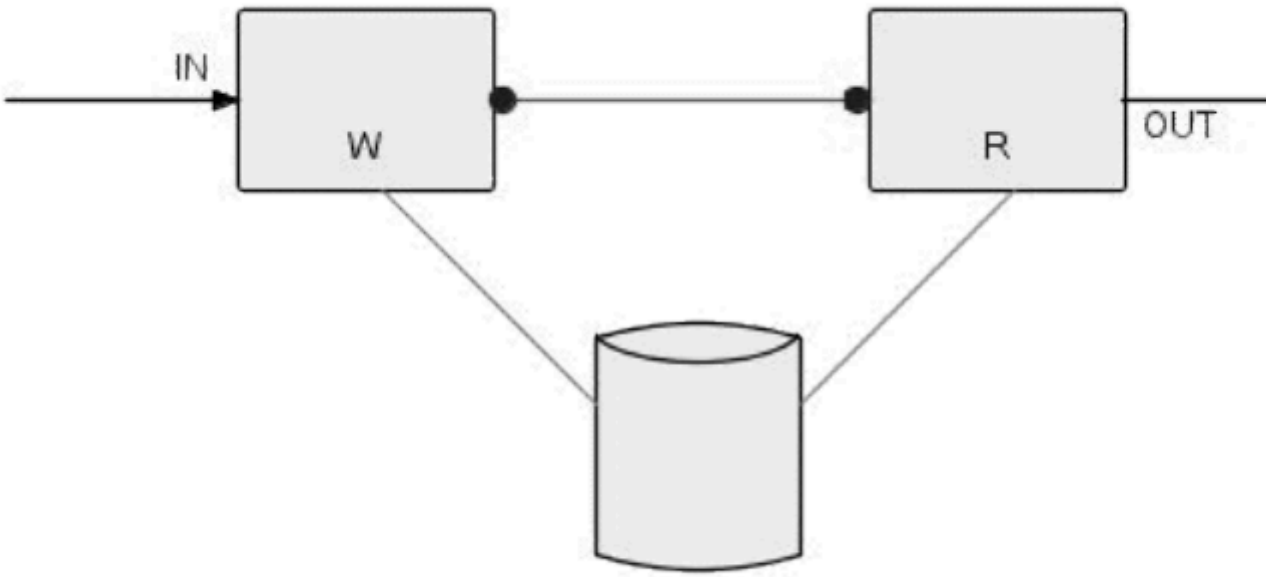
- Automatic ports.
  - Are ports are not under the component control
  - Not all ports need to be known to the components they are attached to: sometimes it is desirable to be able to specify connections in the network which the processes themselves don't know about
  - These are especially useful for introducing timing constraints into an application without having to add logic to the components
  - Consider two processes, one writing a file and one reading it. The network designer wants to interlock the two components so that the reader cannot start until the writer finishes. To do this, you figure that if you connect an input port to the reader, the reader will be prevented from starting until an IP arrives on that port (by the above scheduling rules). On the other hand, readers don't usually have input ports, and if you add one, the reader will have to have some additional code to dispose of incoming IPs.

    Similarly, you would also have to add code to the Writer at termination time to send an "I'm finished" IP to an appropriate output port.

    Now, to avoid having to add seldom used code (to put out and receive these special signals) to every component in the entire system, the software should provide two optional ports for each process which the implementing component doesn't know about: an automatic input port and an automatic output port. If the automatic output port is connected, the Scheduler closes it at termination time. The automatic input works like this: if there is an automatic input port connected, process activation is delayed until an IP is received on that port, or until the port is closed. This assumes that no data has arrived at another input port

    Here is a picture of the Writer/Reader situation:

Figure 3.5. Automatic Ports



4. Loop-Type Networks

5. Other Concerns

- Synchronization and Checkpoints
- Error Handling
- Auditing
- Debugging
- Recursion

6. FBP vs OOP

Chapter 4. An Extended Application Model

- Configuration. Hierarchical configuration based on Sections and Settings
- DSL. A Domain Specific Language to handle expression in different parts of the application(Parameters, Connection Strings)
- Data Connections. They ensure connectivity with outside world and are Connection String based.
- Triggers. Are fired when an external event related to the trigger occurs. (SugarCRM add record, TeamWork delete company etc)
- Flows. Are represented by data flow networks that are executed when the application or on a specific Trigger.

Chapter 5. Sample Application - Entity Sync

Chapter 6. Some Conclusions

- **Component based**. Dataflow programming is a component based programming methodology
- **Reactive**.Dataflow programming reactive programming. It is responsive to changing data and can be used to automatically propagate GUI events to other subsystems or components
- **Common points with Control flow programming**.Synchronous pipeline based data flow programming can be represent as Control Flow Programming. Also, internal logic of a component can be implemented using CF prorgamming.
- **Parallelization.**Dataflow has an inherent ability for parallelization. It doesn't guarantee parallelism, but makes it much easier.
- **Scalability**. Dataflow scales up from tightly coordinated processes within a single processor to largely independent cooperating processes, perhaps on different machines.
- **Orchestration**. Dataflow is a high-level coordination language that assists in combining different programming languages into one architecture. How nodes are programmed is entirely left up to the developer (although implementations may put constraints on it, the definition of data flow does not). Dataflow can be used to combine code from distant locations and written in different languages into one application

Chapter 7. References

- Data Flow Programming - Wiki
- Flow Based Programming, JP Morrison
- Dataflow and Reactive Programming Systems
- NoFlow for JavaScript