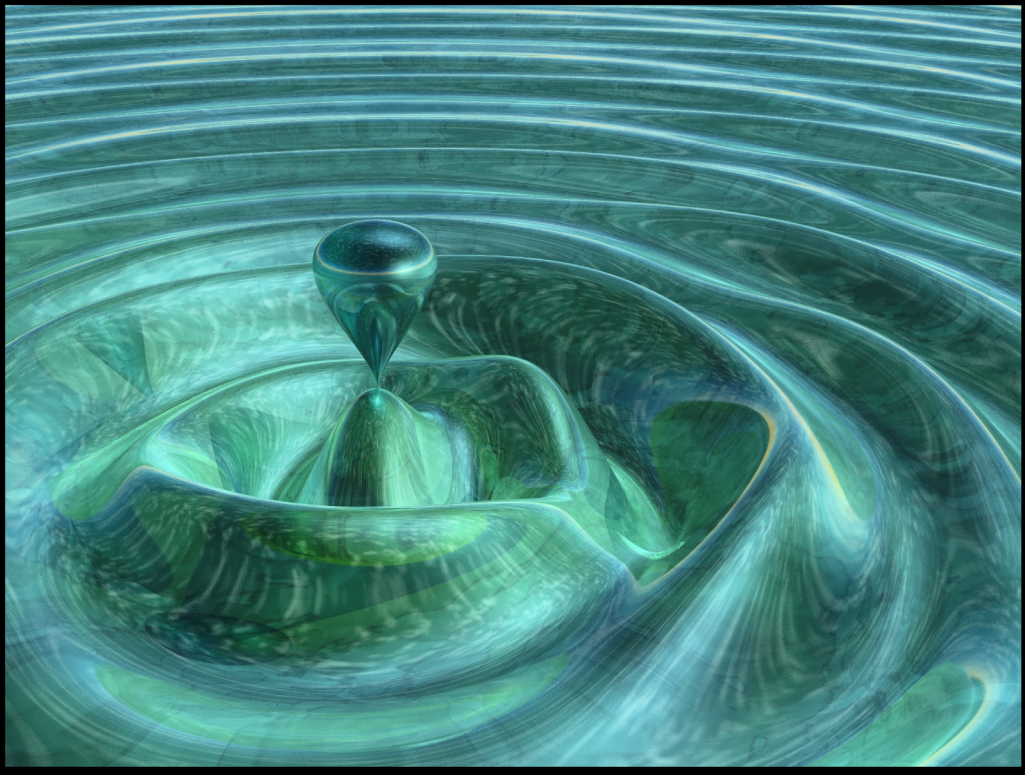
Dataflow & Reactive Programming Systems



A Practical Guide

Theory • Application • Examples

Matt Carkci

Dataflow and Reactive Programming Systems

A Practical Guide to Developing Dataflow and Reactive Programming Systems

Matt Carkci

This book is for sale at http://leanpub.com/dataflowbook

This version was published on 2014-05-29

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1 Introduction

Dataflow is a method of implementing software that is very different to the prevailing Von Neumann method that the software industry has been based on since inception.

At the lowest level, dataflow is both a programming style and a way to manage parallelism. At the top, dataflow is an overarching architecture that can incorporate and coordinate other computational methods seamlessly.

Dataflow is a family of methods that all share one important fact, data is king. The arrival of data causes the system to activate. Dataflow reacts to incoming data without having to be specifically told to do so. In traditional programming languages, the developer specifies exactly what the program will do at any moment.

1.1 Overview of the Book

It is important to understand the *concepts* of dataflow and not just the specifics of one library so that you can quickly adapt to any new library encountered. There are many varieties of dataflow with subtle differences yet they all can be considered dataflow. Sometimes very slight changes in the dataflow implementation can drastically change how you design programs. This book will explain the whole landscape of dataflow.

You'll learn dataflow from the software perspective. How it is an architecture and a way to think about building programs.

We'll start by covering it in its simplest form, Pipeline Dataflow, and then move on to the many features and variations you'll encounter in existing implementations. Three of the most common

styles of dataflow are explained in detail using code of a working implementation to bring theory into practice.

You should already have a little programming experience under your belt but you don't need to be an expert to understand what this book covers.

1.2 Reactive Programming is Dataflow

"Reactive Programming" is a term that has become popular recently but its origin stretches back to at least 1985. The paper, *"On the Development of Reactive Systems"* by David Harel and Amir Pnueli was the first to define "reactive systems":

> "Reactive systems... are repeatedly prompted by the outside world and their role is to continuously respond to external inputs."[1]

The paper specifies that reactive systems are not restricted to software alone. They were discussing ways to develop any type of reactive system, software or hardware. A few years later in 1989 Gerard Berry focuses on the software aspects in his paper, *"Real Time Programming: Special Purpose or General Purpose Languages"*:

> "It is convenient to distinguish roughly between three kinds of computer programs. *Transformational programs* compute results from a given set of inputs; typical examples are compilers or numerical computation programs. *Interactive programs* interact at their own speed with users or with other programs; from a user point of view a time-sharing system is interactive.

[1] Harel, D., & Pnueli, A. (1985). "On the development of reactive systems" (pp. 477-498). Springer Berlin Heidelberg. Chicago

> *Reactive programs* also maintain a continuous interaction with their environment, but at a speed which is determined by the environment, not by the program itself. Interactive programs work at their own pace and mostly deal with communications, while reactive programs only work in response to external demands and mostly deal with accurate interrupt handling.
>
> Real-time programs are usually reactive. However, there are reactive program that are not usually considered as being real-time, such as protocols, system drivers or man-machine interface handlers. All reactive programs require a common programming style.
>
> Complex applications usually require establishing cooperation between the three kinds of programs. For example, a programmer uses a man-machine interface involving menus, scroll bars and other reactive devices. The reactive interface permits him to tell the interactive operating systems to start transformational computations such as program compilations."[2]

From the preceding quotes we can say that reactive programs…

- Activate in response to external demands
- Mostly deal with handling parallelism
- Operate at the rate of incoming data
- Often work in cooperation with transformational and interactive aspects

The definition of dataflow is a little more vague. Any system where the data moves between code units and triggers execution of the code could be called dataflow, which includes reactive systems.

[2]Gerard Berry (1989). "Real Time Programming: Special Purpose or General Purpose Languages" (pp.11-17) IFIP Congress

Thus, I consider Reactive Programming to be a subset of dataflow but a rather large subset. In casual use, Reactive Programming it is often a synonym for dataflow.

1.3 Von Neumann Architecture

The reason parallel programming is so hard is directly related to the design of the microprocessors that sit in all of our computers.

The Von Neumann architecture is used in the common microprocessors of today. It is often described as an architecture where data does not move. A global memory location is reserved and given a name (the variable name) to store the data. Its contents can be set or changed but the location is always the same. The processor commands, in general, deal with assigning values to memory locations and what command should execute next. A "program-counter" contains the address of the next command to execute and is affected by statements like `goto` and `if`.

Our programs are simply statements to tell the microprocessor what to do… in excruciating detail. Any part of the program can mutate any memory location at any time.

In contrast, dataflow has the data move from one piece of code to another. There is no program-counter to keep track of what should be executed next, data arrival triggers the code to execute. There is no need to worry about locks because the data is local and can only be accessed by the code it was sent to.

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, has forced us to use locks and other coordination methods with little success. Applications of this style are not scalable and puts too much burden on developers to get it right. Unfortunately we are probably stuck with Von Neumann

processors for a long time. There's too much software already written for them and it would be crazy to reproduce the software for a new architecture.

Even our programming languages are influenced by the Von Neumann architecture. Most current programming languages are based directly or indirectly on the C language which is not much more than a prettier form of assembly language. Since C uses Von Neumann principals, by extension all derivative languages are also Von Neumann languages.

It seems our best hope is to emulate a parallel friendly architecture on top of the Von Neumann base. That's where this book comes in. All dataflow implementations that run on Von Neumann machines must translate dataflow techniques to Von Neumann techniques. I will show you how to build those systems and understand the ones you will encounter.

1.4 Benefits of Dataflow

Some of the benefits of dataflow that we'll cover in this book are…

- Dataflow has an inherent ability for parallelization. It doesn't guarantee parallelism, but makes it much easier.
- Dataflow is responsive to changing data and can be used to automatically propagate GUI events to all observers.
- Dataflow is a fix for "callback hell."
- 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 dataflow does not). Dataflow can be used to combine code from distant locations and written in different languages into one application.

> "Contrary to what was popularly believed in the early 1980s, dataflow and Von Neumann techniques were not mutually exclusive and irreconcilable concepts, but simply the two extremes of a continuum of possible computer architectures."[3]

- For those visual thinkers out there, dataflow graphs lend themselves to graphical representations and manipulation. Yet there's no requirement that it **must** be displayed graphically. Some dataflow languages are text only, some are visual and the rest allow both views.

1.5 History

The first description of dataflow techniques was in the 1961 paper, *"A Block Diagram Compiler"*[4] that developed a programming language (BLODI) to describe electronic circuits. The paper established the concepts of signal processing blocks communicating over interblock links encoded as a textual computer language. "BLODI was written to lighten the programming burden in problems concerning the simulation of signal processing devices"[5].

In 1966 William Robert Sutherland wrote, *"The On-Line Graphical Specification of Computer Procedures"*[6] that heavily influenced the visual representation of dataflow. He proposed a purely visual programming language where the user interacted with computer using the new technology of video displays and drawing tablets. Objects were drawn and then given a meaning. In a historical video,

[3] Johnston, W. M., Hanna, J. R., & Millar, R. J. (2004). Advances in dataflow programming languages. ACM Computing Surveys (CSUR), 36(1), 1-34. Chicago

[4] John L. Kelly Jr., Carol Lochbaum, V. A. Vyssotsky (1961). "A Block Diagram Compiler". Bell System Technical Journal, pages 669-678

[5] ibid

[6] Sutherland, W. R. (1966). ON-LINE GRAPHICAL SPECIFICATION OF COMPUTER PROCEDURES (No. TR-405). LINCOLN LAB MASS INST OF TECH LEXINGTON. Chicago

Sutherland is shown drawing a square-root block and then defines its operation by drawing its constituent blocks.

Jack B. Dennis continued the evolution of dataflow by explaining the exact steps that must be taken to execute a dataflow program in his 1974 paper, *"First Version of a Data Flow Procedure Language"*[7]. Many consider this paper to be the first definition of how a dataflow implementation should operate.

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.

From the 1990s until the early 2000s, less research went into dataflow because there was no pressing need. Every 18 months a new, faster microprocessor came out and no one felt a need to change the way things were done. Due to the increasingly graphical capabilities of computers, most of the advances during this period were concentrated in the visual aspects of dataflow. LabView is one of notable developments of this period.

Then we reached the limits of silicon. Starting around 2005, processor speed stopped increasing and the only option was to just add more cores to the chip. Parallelism became important again. Developers began looking around for solutions and created a resurgence in the 40+ year old concept of dataflow and reactive programming.

[7]Dennis, J. B. (1974, January). First version of a data flow procedure language. In Programming Symposium (pp. 362-376). Springer Berlin Heidelberg. Chicago

1.6 The Purpose of this Book

Dataflow is difficult to learn. Not due to inherent complexity but due to the number of variations dataflow can take on and the lack of a standardized language. Take for example the most common of all elements of dataflow, the node. Some call it a node while others call it a “block”, a “process”, an “action”, an “actor” and any number of other names. Extend this renaming to other basic elements and sometimes youâ€™re not sure what you are reading about. Half of the work in reading about dataflow is learning the author’s terminology.

Additionally, dataflow does not have a single set of features and capabilities. It is like ordering from a Chinese restaurant. Mix and match as you want but some things just don’t taste right together.

My goal is to describe all of the possible variations in easy to understand terms. Your goal should be to understand the general concepts of dataflow. Then you will be able to apply that knowledge to specific problems with possibly different semantics than those I describe. The purpose of this book is to give you the tools and understanding to work with a multitude of dataflow systems.

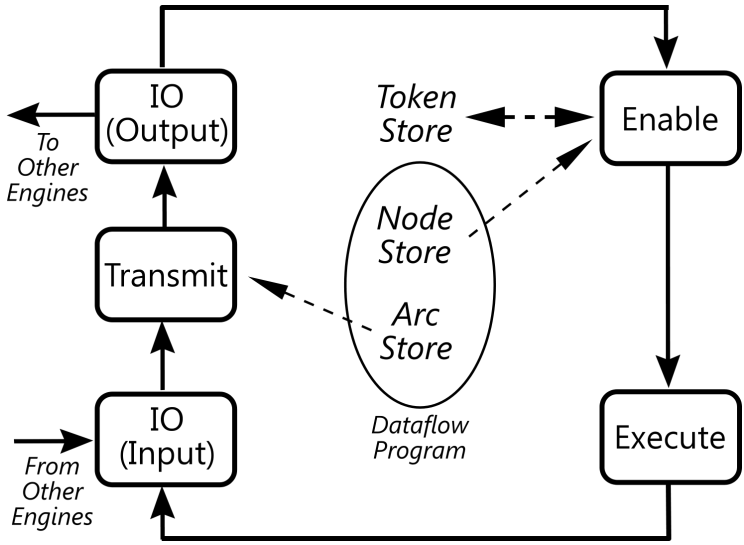
2 Asynchronous Dataflow Implementation

Asynchronous Dataflow is characterized by nodes that fire whenever one of their fire rules are satisfied.

Implementations commonly have a few standard components. An activation unit that determines what nodes can fire, an execution unit that controls how the nodes are executed, a token transmission unit that moves tokens from one node to another and finally storage for nodes and tokens.

This is just the bird's eye view of asynchronous dataflow. There are many different ways to design the system. In the rest of this chapter we will look at the design of a typical asynchronous implementation that you can you as a reference to understand the details of asynchronous dataflow systems.

2.1 Architecture Overview



Top Level Architecture of the Example Implementation

This figure shows the architecture of our asynchronous dataflow system. It uses a simple pipeline dataflow architecture to define a system that runs other, dynamic, asynchronous dataflow programs. It is modeled after a dataflow processor from the early 1980's, the Manchester Prototype Dataflow Computer.

Most asynchronous dataflow processors use some version of this basic architecture. The Manchester Processor design had demands on its design due to the physical nature of computer processors. As software doesn't share in those burdens, I have changed the design to be more general purpose.

Using the features of dataflow we examined earlier in the book, this implementation can be described as:

- Dynamic

- Asynchronous Activations
- Multiple Inputs and Outputs
- Cycles Allowed
- Functional Nodes
- Uses Per-Node Fire Patterns
- Pushes Data
- Arc Capacity > 1
- Arcs May Join and Split
- Single Token per Arc per Activation

2.2 Implementation Walk-Through

The four main components are:

- IO: Communication with other engines and the world
- Transmit: "Moves" a token by changing its location address to the next port's address
- Enable: Determines what nodes can fire
- Execute: Executes nodes

Tokens come into the system through the input side of the IO unit. Its job is to keep any tokens with addresses inside this engine and to pass on all other tokens.

Tokens then are sent to the Transmit unit. It looks at the token's address and compares it to a look-up table of connections in the system. If it finds that the new token's address is on an output port, then it will make a copy of the token and give it the address of the input port(s). Effectively, moving the token from one output port to another input port. If the token's address is already on an input port, then it keeps the address the same.

The tokens with the new addresses are sent from the Transmit unit to the output side of the IO unit. Its job is to send any tokens with external addresses and to keep those with internal addresses.

Our local tokens then move to the Enable unit. It looks at the incoming tokens and compares them to a store of waiting tokens to see if any nodes can now fire due to the new token. If not, it will save the new token in the Token Store for later use. If a node can now be activated, it creates a new token called an Execute Token. It packages together all the data tokens for the inputs of the node and a way to invoke the node itself.

The Execute unit receives these Execute Tokens and runs the node. Any output tokens are passed back around to the input IO unit again to start over.

2.3 Main Data Types

Besides the Fire Pattern data type below, all of these can be implemented as classes in an Object Oriented language, a `struct` in C or the equivalent in your language of choice. The itemized elements under each data type are the members of the type.

2.3.1 Port Address

Defines a unique location (combination of node ID and port ID) of a port within this engine

- Node Id - Unique to engine
- Port Id - Unique to node only

2.3.2 Data Token

A container for the data value and its location

- Value - Any data value
- Port Address - Current location (port) of token

2.3.3 Execute Token

Combines everything needed to fire a node

- Data Tokens - A collection of all tokens on inputs of node
- Node - A means to activate the node

2.3.4 Node

A run-time node combines the Node Definition and a unique Node ID

- Node Definition - Defines how to activate the node
- Node Id - Engine wide, unique id

2.3.5 Node Definition

A node declaration and definition. A single Node Definition may be used to define many run-time nodes that all act the same as the Node Definition – just with different Node IDs.

- Node's activation function - Function that does the real work
- List of Ports - All the ports on this node
- Fire Patterns - A collection of Fire Patterns

2.3.6 Arc

An Arc is a connection between to two ports

- Source Port Address - Address of the output port
- Sink Port Address - Address of the input port

2.3.7 Fire Pattern

This is a union type in C/C++, a sum type in Haskell, or, in object oriented languages, a base class of Fire Pattern with one sub-class for Exists, one for Empty and so on. This could also be implemented as an enumeration.

A Fire Pattern for a single port is one of the following:

- Exists - A token must exist on this input
- Empty - Input must be empty
- Don't Care - Doesn't matter if a token is available or not
- Always - Ignores all other patterns, and forces the whole fire pattern to match

The pattern for the whole node is simply a collection of the all the Fire Patterns for that node.

2.3.8 Token Store

A collection of all the Data Tokens in the system. Read and written to by the Enable Unit only. The tokens in here represent the complete state of the program.

2.3.9 Node Store

A collection of all the nodes in the system. Note changing this at run-time allows for dynamic node redefinitions.

2.3.10 Arc Store

A collection of all the connections in the system. Note changing this at run-time allows for dynamic graphs.

2.3.11 Dataflow Program

The Node Store and Arc Store together are everything needed to define a dataflow program. This is loaded into the engine before execution.

- Node Store - A collection of all the nodes in the program.
- Arc Store - A collection of all the arcs in the program.

2.4 Implementation Components

2.4.1 IO Unit

Input: Data Token
Local Output: Data Token
External Output: Data Token

The IO Unit is the interface to the engine. Tokens arriving at the input port with internal addresses are directed to the "local" port of the component and those with external addresses are directed to the "external" port.

2.4.2 Transmit Unit

Input: Data Token
Output: Data Token

Token movement along arcs are implemented with this unit. The Transmit Unit looks at the tokens address and compares it to a look-up table of connections in the system (Arc Store). If it finds that the new token's address is on an output port, then it will make one copy of the token for each input port and give it the address of that input port. This action is equivalent to moving the token along the arc and sending a copy down each path.

The look-up table is an encoding of all the connections in the program. Changing the values in this table changes the graph at run-time.

2.4.3 Enable Unit

Input: Data Token
Output: Execute Token

The Enable Unit looks at the incoming tokens and compares them to a store of waiting tokens. The Token Store holds all the tokens currently moving through the program. By comparing the waiting tokens with the node's fire pattern (pulled from the Token Store), the Enable Unit can determine if a node can fire.

For activated nodes, it creates and sends an Execute Token that packages together all the data tokens for the inputs of the node and a way to invoke the node itself. The tokens are removed from the Token Store and the node definition is copied from the Node Store.

If the incoming token does not cause a node to activate, then it will save the new token in the Token Store for later use.

This implementation allows for per-node firing patterns. The original Manchester Processor design had one firing pattern for every node… all inputs must have tokens waiting before the node can activate. And since all nodes in the original design only had 1 or 2 inputs, the Manchester architecture's Enable Unit didn't need access to the node's definition.

Due to the addition of per-node firing patterns and more than 2 inputs allowed per node, this design requires Enable to connect to the Node Store and the Token Store while the Manchester design only needed a connection to the Token Store.

The Enable Unit is the only component that writes to the Token Store so no special locking is needed for multithreaded operation.

2.4.4 Execute Unit

Input: Execute Token
Output: Data Token

With the contents of the Execute Token, this unit has everything it needs fire nodes.

It takes the collection of data tokens from the Execute Token and passes it to the node definition for evaluation. In this implementation the node definition is simply a function that takes a collection of tokens and returns another (possibly empty) collection of output tokens.

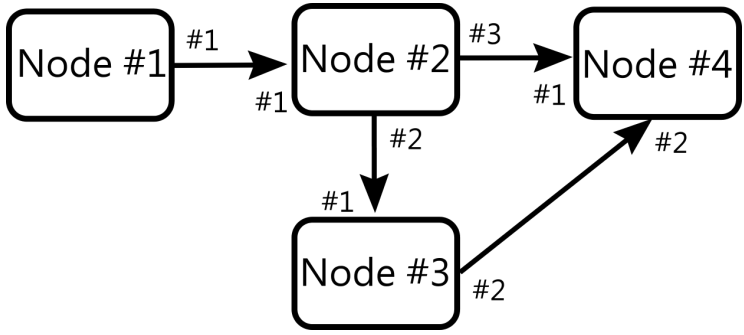
The node's function only deals with what I call, "local tokens." They are just like the regular data tokens (that I refer to in this context as a "global token") without the Node ID field. Nodes should be isolated and not have to know anything about the outside world. The node's ID is external to the definition of the node itself. It doesn't matter if there are 10 nodes, all with the same definition and different node IDs, they should all operate exactly the same. What the node does know about is its Port IDs. Port IDs are unique to the node definition. The node's function returns a collection of local tokens with addresses of output ports that exist on the node.

The Execute Unit must first convert a global token to a local token. It does this by simply stripping off the node's ID but retaining the port ID and data value. It calls the function with the locals (input tokens) and gets back a collection of locals (output tokens). The unit converts these to globals by adding the current node's ID back to the tokens along with the port ID and data value.

In the original Manchester design, nodes were defined by an op-code in the same way that assembly language instructions in a typical microprocessor are given numeric identifiers. The Execute Unit knew how to execute an op-code it received so the Execute Token only needed to include an op-code number instead of the full node definition like this implementation requires. In software

it costs the same to pass an integer as it does to pass the full node definition and makes the design more general.

2.5 Program Execution Example



Example Program. The number next to the port is the Port Id

We will assume that the dataflow program is already loaded into the engine. Node #1 is the first node to activate. When it is done, the node pushes a new token to its output port (#1) with the address of (Node #1, Port #1). This states that the token is currently on the output end of the arc between nodes #1 and #2. Then Node #2 fires since it has a token on its input.

For that sequence to happen, the engine has to do the following…

Immediately after starting the engine with the example dataflow program, there are no tokens in the program. The Enable Unit normally first looks at its incoming tokens to see if a node can be executed due to the new token. In this case, there are no input tokens to the Enable Unit but it finds that Node #1 is a source node so that means it can fire all the time. So the Enable Unit creates a new Execution Token with Node #1's definition as its contents and sends it to the Execute Unit.

The Execute unit sees that it has a new Execute Token waiting so it consumes the token and fires the Node Definition found

in the Execute Token. As mentioned above, activating Node #1 pushes a new token to its output. The Execute Unit gets the token, produced from Node #1, and sends that to the input IO Unit which immediately sends it to the Transmit Unit.

Remember that the token's address is (Node #1, Port #1)... The Transmit Unit looks in the collection of all arcs in the system to find where the token should be sent. Node #1's output arc connects to Node #2's input port #1. So the Transmit Unit changes the address of the token to (Node #2, Port #1) saying that the token has been moved to the input port of Node #2. The token with the updated address is passed to the output IO Unit. Since the address of the token is local to this engine, it sends the token to the Enable Unit.

Now the Enable Unit will look at the incoming token and see that now Node #2 can fire because a token is waiting on its input. It takes the token and places it into an Execute Token along with the Node Definition for Node #2.

The Execute Unit then activates Node #2 just like it did before with Node #1 and the cycles continue.

2.6 Preparing a Program for Execution

This example implementation does not include any means to convert a human friendly format (program text) to the engine's representation. Besides parsing and validating the program text, which is the same for every programming language, the only other thing required is to generate a unique ID for every node in the program.

This is the run-time identifier that uniquely identifies every node in the engine.

The best choice is to use Universally Unique IDs (UUIDs also called GUIDs). Second best is to use unique integers. UUIDs take up 128 bits each so space could be an issue for some designs.

UUIDs are best because they only need to be defined once and allow us to change the graph at run-time without worrying about generating duplicate integer IDs. They also can be used to refer to a specific version of a node. If you generate a new UUID anytime a breaking change is made to a node, all existing code referring to the old UUID will continue to work as expected.

Choose wisely because the type of ID impacts the maximum number of nodes you can have in the engine at any one time and thus restricts maximum program size.

The end result of the preparation phase is a filled in Node Store and Arc Store that is passed to the engine to execute.

2.7 Multiple Dataflow Engines

The Manchester architecture was designed to be easy to combine with other processors. Simply connect the IO units of a few of them to a bus so they can communicate. The tokens will be sent anywhere the address specifies.

This design does not handle multi-engine configurations as well as it could. The addition of an engine ID to the Port Address type would allow you to easily move nodes around to other engines to balance the load. It only takes changing the addresses on the arcs and a few other minor changes.

With multiple engines, some sort of overseer is necessary to balance the load and move code from one engine to another. This implementation was designed as an example of an asynchronous dataflow engine so no effort was spent on external components to make it easy to combine with other engines.