Introduction

The following diagram 1 shows a simple box-and-arrow diagram.

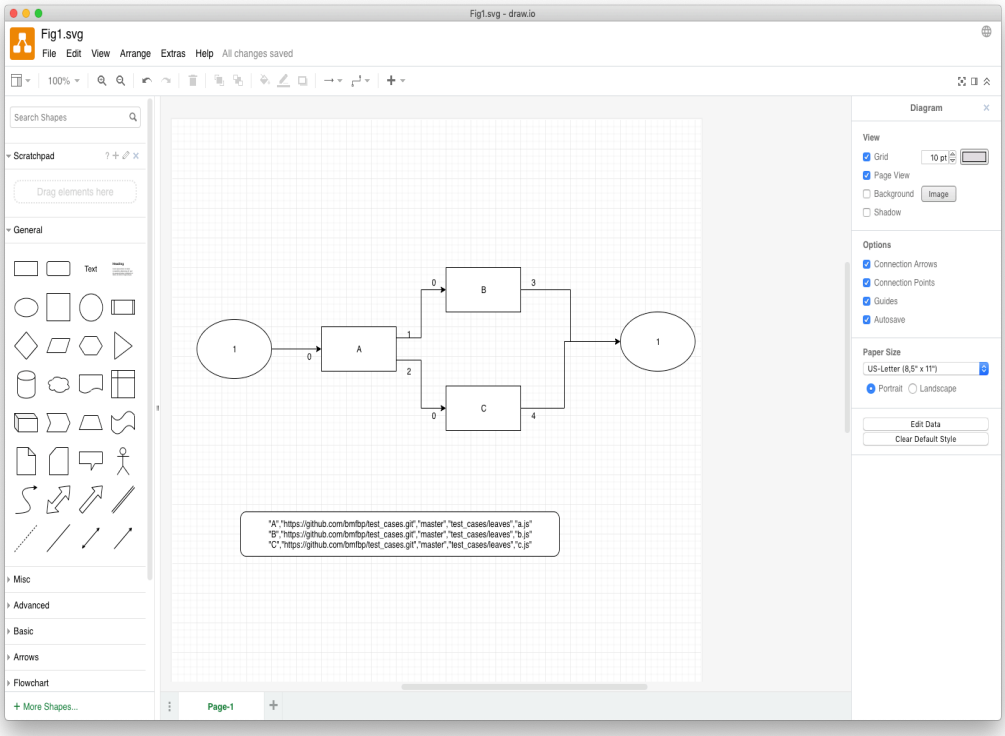

Figure: 1

A general diagram can be composed of many graphical items.

To build a diagram compiler, we create a *convention* to specify which graphical items will be accepted and what they mean to the compiler.

In a Composition diagram convention, I wish to describe the composition of software components which I call *parts*.

I will call this *particular* kind of diagram a "Schematic Diagram". The reader may see that a Schematic Diagram is very similar to a Network Diagram that we see on white-boards. A Schematic Diagram is different from a Network Diagram in that the software parts (software components) are very small[1] and

[1] about the size of a function

the parts are run as parts of an *app* instead of in separate *processes* scheduled by the operating system(s).

The Schematic Diagram is akin to *schematics* that represent electronics assemblies. In fact, many of the ideas from electronics schematics have been "borrowed" (aka "reused"), e.g.

- Parts are concurrent.
- Parts have named input and output pins, forming both, an input API and an output API.
- Messages are one-way only (no return is implied).
- Lines with arrow-heads represent *wires* which allow information to flow in only one direction.
- Diagrams are hierarchical - sub-assemblies are elided and shown on other diagrams connected via off-page connectors.

The example diagram is read as:

- An event comes from the outside into pin 1 (left oval) of the diagram.
- The incoming event (from pin 1) is routed to pin 0 of Part A.
- Part A *may* produce new events and send them to its pins 1 and 2.
- If Part A produces an event on its output pin 1, this event is routed to input pin 0 of Part B.
- If Part A produces an event on its output pin 2, this event is routed to input pin 0 of Part C.
- If Part B produces an event on it output pin 3, this event is routed to the outside via output pin 1 of the Schematic (rightmost oval).
- If Part C produces an event on it output pin 4, this event is routed to the outside via output pin 1 of the Schematic (rightmost oval).[2]

[2] This might actually be hard to discern using this drawing convention. The wires from B[3] and from C[4] both connect to the output oval 1. The wires are drawn overlapped and appear as a single wire. Likewise the right-pointing arrows overlap and appear as one arrow. In the future, we will add *Dots*, as used in electronics schematics to disambiguate this situation.

The use of the rounded rectangle in Schematic Diagrams will be discussed later.

Rules for Diagramming

I use diagrams to express Software Architecture.

As such, *every* diagram must "make sense" without the need to refer to other diagrams. This is the main rule of drawing software diagrams (IMO).

N.B. the above *rule* implies several features, which are not common in today's software:

- Child diagrams cannot *override* the operation(s) of a parent diagram. We employ *Composition* not *inheritance.*
- All parts on a diagram must be independent from all other parts. This implies that all parts are *concurrent.*
- All names (and indices) on a diagram are local to the diagram. As long as every Part on a diagram is uniquely named, there is no namespace pollution.
- "Exported" names (and indices) appear in the input and output off-page connectors (ovals) on a diagram. A diagram is fully specified by it's name, its input API and its output API.
- Input pins and output pins can have the same names (indices) without conflict.
- Wires (lines) are completely local to the diagram and "controlled" by the diagram.
- A *child* part cannot name or otherwise "see" its peers on a diagram. The Schematic Diagram controls all wires and all routing of messages between parts on the diagram.[3]
- Only the parent "knows" about the routing table between Parts on a

[3] I.E. "routing" is indirect. A child part places an event on its output. The parent Schematic decides to which other parts the event is routed too.

diagram. In fact, an event sent by a part to its output pin might not be connected to any other part (this is call NC - no connection).

The Parts of a Schematic Diagram

A Composition Diagram is composed of three main graphical objects:

- Rectangles
- Text
- Lines.[4]

Our *convention* for Schematic Diagrams requires us to add two more graphical primitives, before the diagram can be understood in a stand-alone manner (see "Rules for Diagramming"):

- Ovals and,
- Rounded Rectangles.

This example Schematic Diagram contains five piece of semantic information:

1) Parts. Rectangles with a single piece of text inside them (their "kind" - much like a type or class).
2) Wires between parts (lines with arrowheads).
3) Pins, input and output. Represented as numeric text.[5]
4) Metadata. Represented as rounded rectangles containing text. The text contains five comma-separated strings per part.
5) Off-page connectors. Represented as ovals containing one piece of text - a pin index.[6]

[4] With arrowheads.

[5] Currently, we use only numeric pin (index) names. Later, we intend to add string names for pins.

[6] Again, in the future, we will allow pins to have string names.

Draw.IO

Currently, we use a free tool called Draw.IO, to create diagrams.[7]

IO've marked up the previous diagram in 1, showing which Draw.IO tools must be used to create the various elements of Schematic Diagrams.

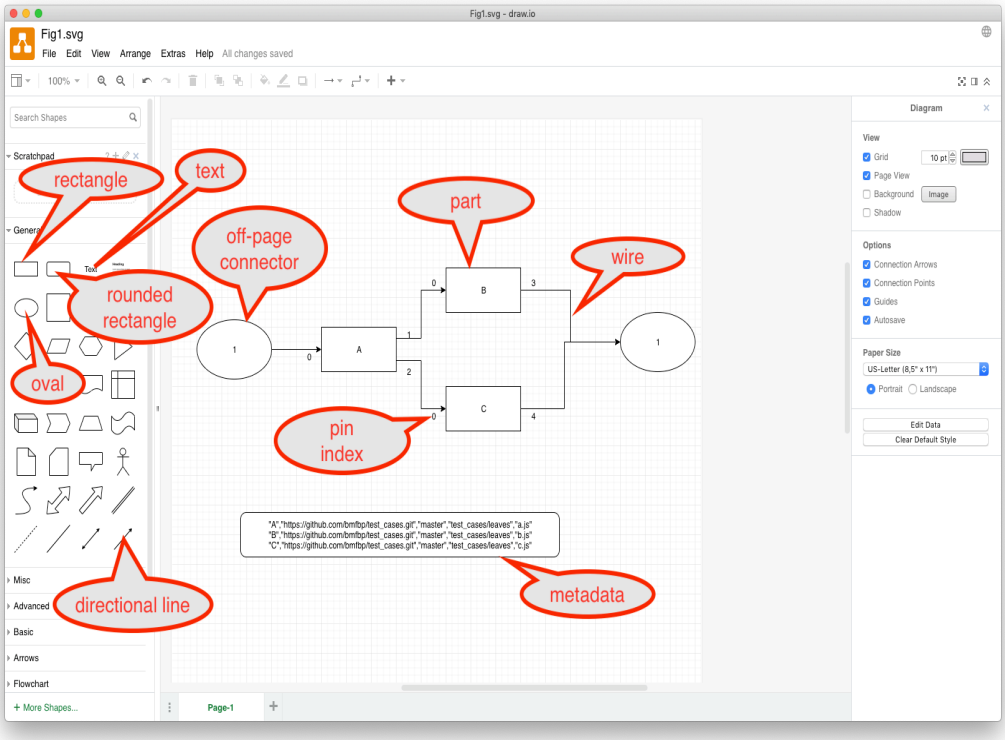

Figure: 2

The compiler recognises a small subset of diagramming commands. Schematics created using Draw.IO must use the tools indicated by this marked up diagram and must be saved as .SVG files.

Version Management

[7] Draw.IO is far from a perfect DaS (Diagrams as Syntax) editor, but it exists and can be used. The issues of creating a better DaS editor are discussed elsewhere.

We use GitHub and metadata for version management.

Every Schematic diagram contains a rounded rectangle with metadata information in it.

Every Part on the schematic is described by one line of metadata information in the rounded rectangle. Every line of metadata consists of five, comma-separated strings[8]:

- The Part's name, as it appears on the diagram.
- The GitHub repository name.
- The GitHub repository branch name.
- The directory path[9] to where the file will be found.
- The actual file name[10].

Using this information, the builder can extract every Part from the local directory system. The builder refreshes the local directory system by using "git pull" on the repository & branch specified in the metadata.

In this way, every Schematic diagram is self-consistent and specifies the version of every part used.

A Schematic Diagram must not know how a part is implemented, e.g. whether it is implemented as a Leaf part (code, JS for example) or by another Schematic Diagram.

Software That Cannot Be Usefully Expressed as Text

In 3, I show a diagram that is ambiguous if translated to text, while being

[8] Surrounded by double-quotes.

[9] Absolute or relative to the project.

[10] With appropriate extension, e.g. ".js" for JavaScript, ".schem" for Schematic Diagrams, etc.

unambiguous as a Schematic Diagram. I have removed metadata and off-page connectors for clarity.

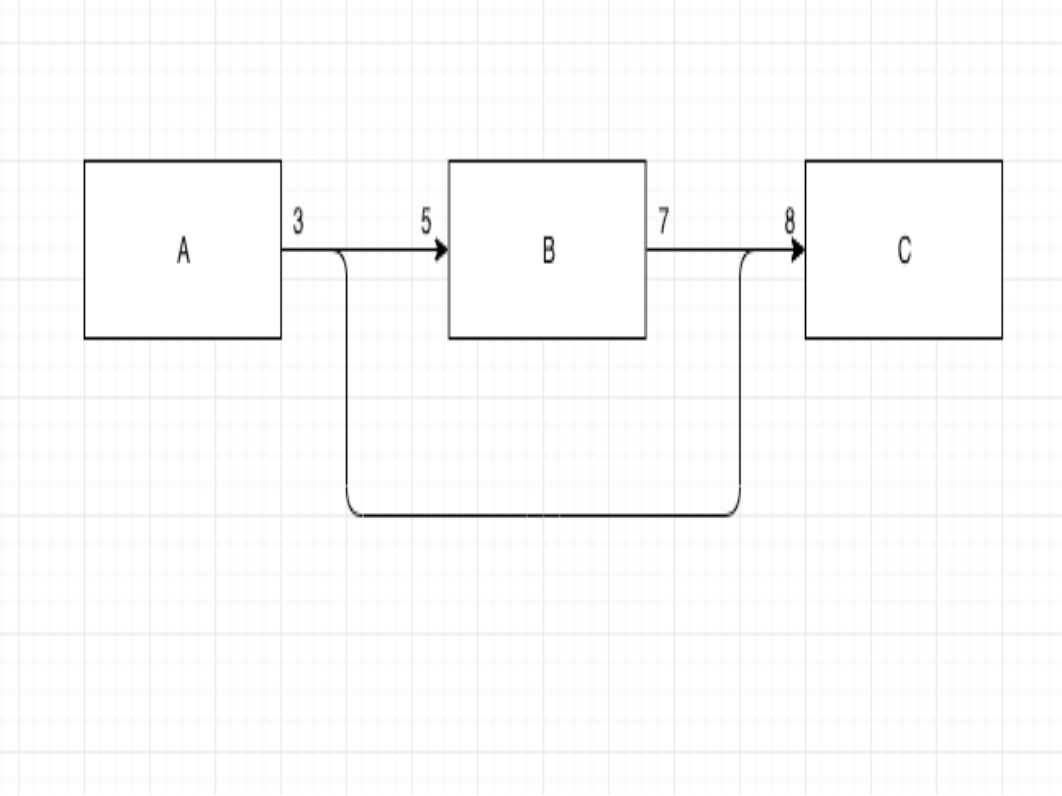

Figure: 3

This diagram consists of three parts and shows that part A outputs to parts B and to part C, while part B outputs to part C.

Expressed in text, the diagram requires two lists:

1. A list of part instances vs. their part "kinds". For example, part A is actually an instance of the A kind. We need to give that part a unique instance ID, say "Instance1".
2. A wiring list between the part instances.

For example, the graph in text might be:[11]

{

[11] In fact, many layouts for the graph are possible and the current compiler uses a layout different from this one.

```
    "instances" : [
                  {"Instance1" : "A"},
                  {"Instance2" : "B"},
                  {"Instance3" : "C"}
                 ]
    "numberOfWires" : 4,
    "wiringList" : [
                    //[ from     pin] , [to         pin]
                    { ["Instance1", 3], ["Instance2", 5] },
                    { ["Instance2", 7], ["Instance3", 8] },
                    { ["Instance1", 3], ["Instance3", 8] }
                  ]
}
```

In my opinion, the diagram shows the layout better than the textual representation.

N.B. The ambiguity, in text, occurs when pin 3 of A[12] fires an event. Is the event sent to pin 5 of B first, or to pin 8 of part C, first?

In most state-of-the-art programming languages, the order of event delivery matters. In one case, the code might execute the route:

- A[3] calls B[5]
- B[7] calls C[8]
- C returns to B
- B returns to A
- A[3] calls C[8]
- C returns to A

whereas in another case, the code might execute the route:

- A[3] calls C[8]
- C returns to A
- A[3] calls B[5]
- B[7] calls C[8]
- C returns to B
- B returns to A.

The text does not specify - without further labelling - which route is taken. In the first case, C gets an event (as a parameter to the call) before it gets and event from A, and, in the second case, it sees A's event before receiving B's.

[12] More exactly, Instance1, of kind A.

The diagram is unambiguous, though - it states that an event output at A[3] is sent to B[5] and simultaneously to C[8], then (and only then) an event from B[7] is sent to C[8]. Let's say that A[3] outputs *eventX* and B[7] outputs *eventY*. C[8] will see *eventX* followed by *eventY* always.

Most state-of-the-art programming languages[13] follow a 2-way synchronous protocol. The caller calls a function/method, then waits for a corresponding return before proceeding.

Schematic diagrams follow only a 1-way, asynchronous message-sending protocol. The Part that generates an event, sends a message to the input queues of all parts connected to a given output pin. None of the receivers is invoked until the message has been delivered to all input queues of connected parts.

This may seem to be an insignificant difference, but, in practice I've found that using one-way message sending and writing all parts as concurrent components,[14] gets rid of a great deal of multi-tasking "baggage"[15] that we have inherited. This method also removes many dependencies.[16]

Graph Specification

(TBD)

Leaf Node Specification

---

[13] Even those languages that use the phrase "message sending".

[14] Parts cannot know when they will be dispatched and the message ordering is guaranteed.

[15] I assert that much of the belief in the idea that "multi-tasking is hard" is based on our multi-decade use of multi-tasking kernels (aka faked time-sharing) built on top of synchronous architectures and the belief in full-preemption.

[16] See Chapter 1 of https://www.amazon.com/Event-Based-Programming-Taking-Events-2006-04-30/dp/B01JXTVFOS/ref=sr_1_1?keywords=faison+events&qid=1559076702&s=gateway&sr=8-1, which contains a good description of dependencies and how asynchronous events eliminate many problems

(TBD)