DPL Syntax

DPL means *Diagrammatic Programming Language*. Programming in 2024 allows us to use syntax that goes beyond simple text characters. Most programming languages are based on the biases of 1950s, using only bitmaps that 1950s hardware could support - ASCII characters. We can do better. We can create languages that are hybrids of diagrams and characters (ASCII and Unicode).

This document describes a small, but, usable syntax based on simple graphic objects supported by many diagram editors, e.g. draw.io, and data formats such as SVG and XML.

DPL diagrammatic elements use *vector graphics* instead of simple bitmaps. A rectangle is parsed as a rectangle, regardless of whether it has been resized.

DPL diagrammatic elements can overlap, whereas characters are usually arranged on a strict grid of non-overlapping cells.

DPL diagrammatic elements do not imply a sequential meaning. Character-based programming languages are usually read from left-to-right and top-to-bottom, whereas DPL source code elements do not imply such ordering. Such lack-of-ordering allows more degrees of freedom for programming language design. More importantly, lack-of-ordering allows design of programming languages wherein program units are totally isolated from one another. Character-based languages tend to imply a sequential ordering, hence, subtly insert unwanted dependencies between program units. Code libraries built with character-based languages tend not to work like LEGO® blocks, since program units are not fully isolated.

Overview

In this section, we describe a small, but useful, subset of diagram elements that form one kind of DPL. This DPL compiles to code. At the moment, it compiles to Odin. Odin is not garbage-collected and is like a modern version of the C language.

The elements include blocks of text. This DPL is a hybrid of diagrams + text.

In general, the diagrams display a software program as a set of Components interconnected by Connection arrows.

There are two major classes of Component:

- Container
- Leaf.

Containers are drawings that can contain Components and Connections. Containers can contain other Container components and/or Leaf components. In `draw.io`, we represent each Container as a separate drawing on a separate *tab* in the `draw.io` editor.

Leaves represent code. Leaves are represented on diagrams as graduated blue rectangles. The actual code is found in the corresponding `main.odin` file. That's only a convention, not a requirement. The code must provide one entry point `instantiate`. This entry point creates a unique instance of a component with unique instance data and provides a pointer to a handler routine which takes two (2) parameters

1. The component data structure (like *self* in class-based languages).
2. The *message* to be handled.

In this implementation, the component data structure is in the shape of an `zd.Eh` struct. The details can be found in `0D/odin/0d.odin`.

Likewise, the shape of the zd.Message struct can be found in `0D/odin/message.odin`.

Typically, `handle` uses the `send` function to create output messages.

Components are *templates*, similar to *classes* in class-based languages. Components can be instantiated multiple times in a project. Each instantiation is unique and has a blob of unique storage (state) associated with it (similar to *self* in class-based languages).

Templates have unique names. Instances do not need to be explicitly named. The underlying 0D engine guarantees that each instance is unique, in a manner similar to *references* to *objects* in class-based languages.

Components have input and output *ports*. Input ports are drawn as white pills (small rounded rectangles), and output ports are drawn as dark-blue pills. Each port has a unique name. The names are scoped to be visible only within their parent component. The scope of input port names is unique from the scope of output port names within the same component, i.e. a component can use the same name for an input port as for an output port, but every input name must be different from every other input name within the same component and every output name must be different from every other output name within the same component. Different components can use the same names for ports as other components, for example, many components have input ports named “”, and, they have output ports named “”, too. Those port names do not clash, they are "locally scoped" within their parent components.

Containers also have special ports called *gates*. Gates represent the top-level inputs and outputs of a Container. Input gates are drawn as white rhombuses with a name, and, output gates are drawn as blue rhombuses with a name. As already hinted-at, the default input gate is called “” (no name - nothing) and the default output gate is called “” (no name - nothing). In UNIX, these would be called `stdin` and `stdout` respectively.

Connections within Containers are used to hook output ports to input ports of child components in a 1:many and many:1 manner. And, Connections hook *gates* to *ports*.

Components can only *send* messages to their own output ports and gates. This is similar to *input parameter lists* in other programming languages, except that in this DPL, we also include parameter lists for *outputs.* The set of input ports are like input parameter lists, the set of output ports are like output parameter lists. In other programming languages, a function cannot know where a particular input parameter came from. Likewise, in this DPL, a component cannot know where a particular output will go nor where an input came from.

Unlike procedures and functions in most programming languages, inputs to and outputs from a component can happen at any time, in any order. For example, if a component has two inputs A and B, it might be the case that inputs arrive on port B, and, some time later, inputs arrive on port A, or, the inputs might arrive very closely spaced together in time, or, multiple B inputs might arrive before any A inputs arrive, or, inputs *never* arrive on port A, and so on.

It turned out to be trivial to represent shell commands as Leaf Components. We call this *VSH* - for Visual SHell. It is possible to draw shell pipelines in `draw.io` and to execute the pipelines. Combinations beyond simple pipelines are easier to express visually than as pure text. *[Aside: an outcome of this approach is that it is convenient to visually express component programs that contain relatively heavy concepts. For example, a parser can be drawn and implemented as a single component with input ports and output ports. Likewise, an A.I./LLM component can be drawn and implemented as a single component.]*

Visual Overview

First, I list summaries for the various sections, then the details of each section...

Visual Syntax Summary

- bare components
  - Bare Container
  - Bare Leaf
  - Bare VSH component
  - Bare string
- Container
- Leaf
- input gates
  - Noname input gate (aka `stdin` for a complete diagram)
  - Named input gate
- output gate
  - Noname output gate (aka `stdout` for a complete diagram)
  - Named output gate
- input port
  - Noname input port (aka `stdin` for a single component)
  - Named input port
- output port

    - Noname output post (aka `stdout` for a single component)
    - Named output port
- connection
- comment

Visual Semantics Summary

- down
- up
- across
- through
- fan-out (split)
- fan-in (join)

Style, Readability Summary

- opacity
- line style
- line thickness

Idioms Summary

- feedback
- sequential
- parallel

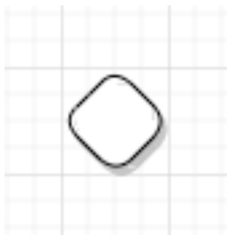
- concurrency
- errors, exceptions

Low Level Technicalities Summary

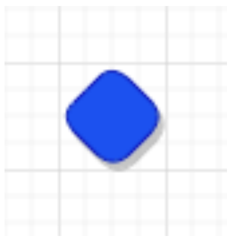
- kickoff inject
- `handle` ()
- `send` ()
- message copying
- active and idle
- notes

Palette

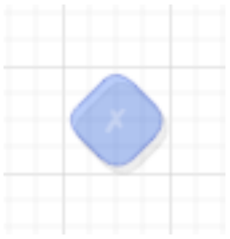
- **Gates**
- **Input gate**



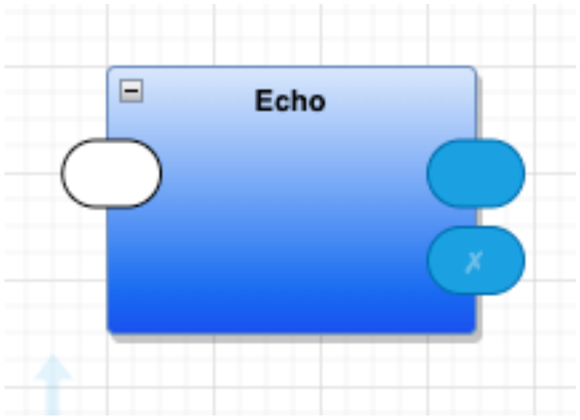
- **Output gate**



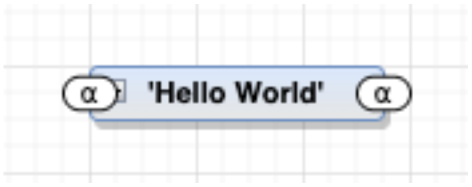
- **Failure output gate**



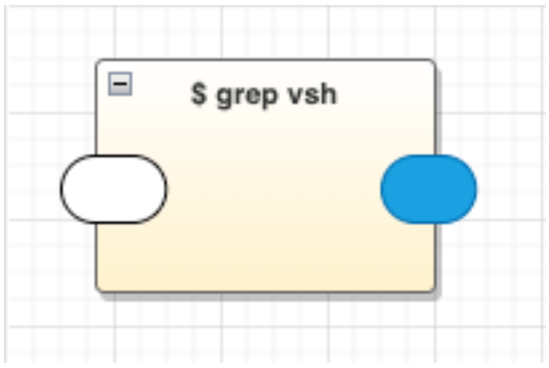
- **Components**
  - Theoretically, it's not necessary (in fact, forbidden) to know how a component is implemented, but, distinct sizes and colours help during a learning curve. Note that the DPL compiler ignores size and colour - these are, but, comments to help human readers.



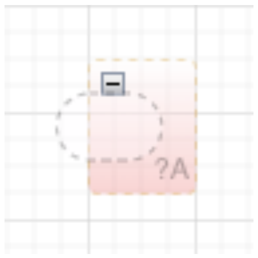
- **String constant**



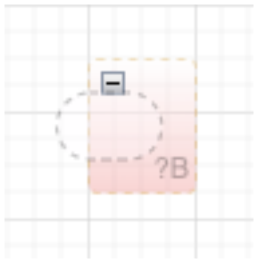
- **VSH component**



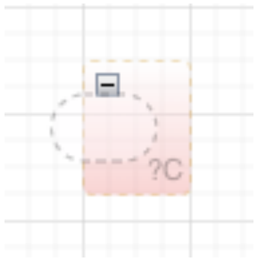
- **Probe A**



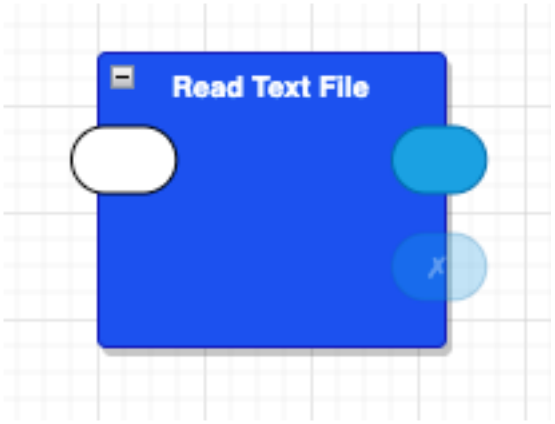
- **Probe B**



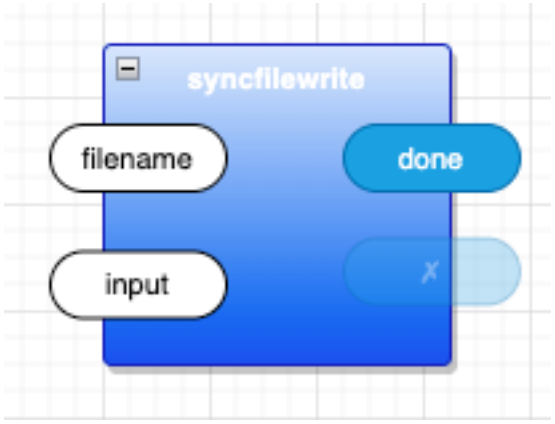
- **Probe C**



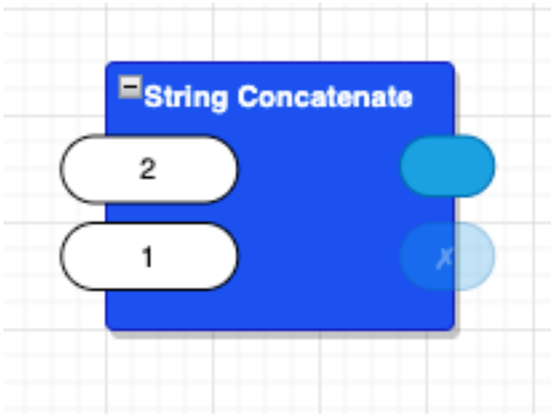
- **Read Text File**



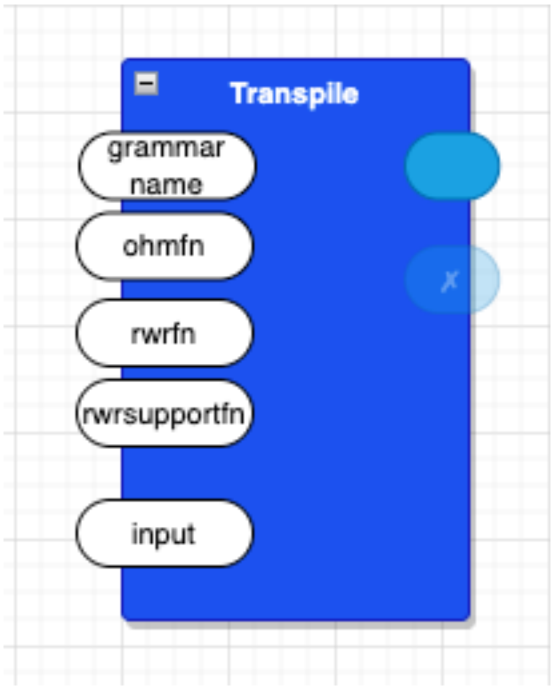
- **SyncFileWrite**



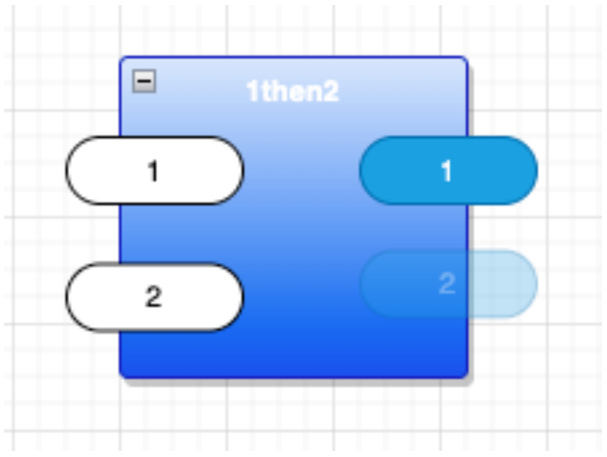
- **String Concatenate**



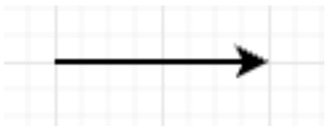
- **Transpile**



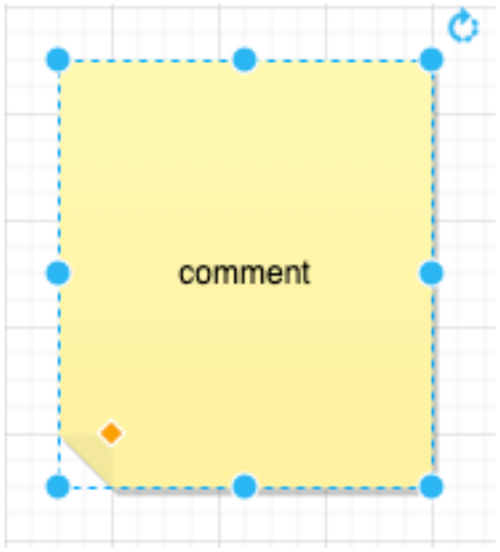
- **1then2**



Connection

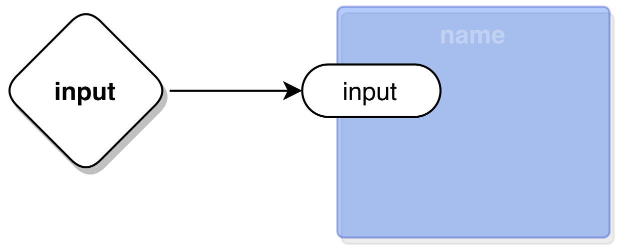


Comment



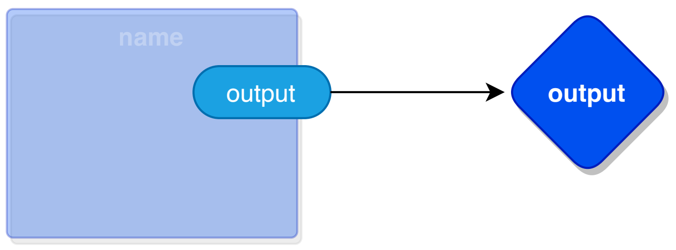
Visual Semantics

down



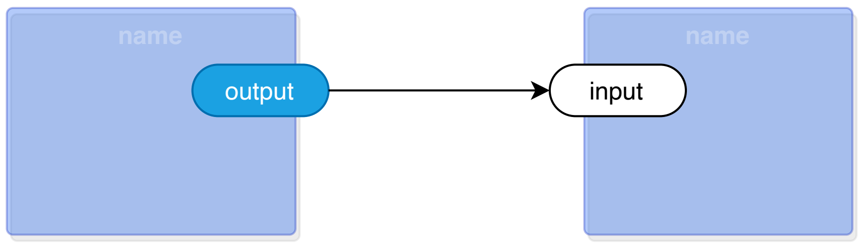
arrow from input gate of Container to input port of child

up



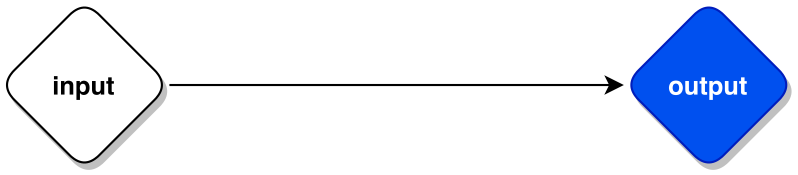
arrow from output port of child to output gate of Container

across



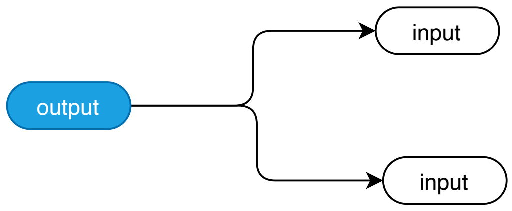
arrow from output port of child to input port of child

through



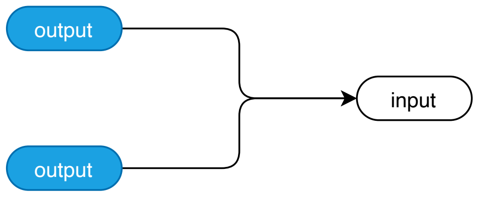
arrow from input gate of Container to output gate of the same Container

fan-out (split)



output arrows go to more than one port/gate

fan-in (join)

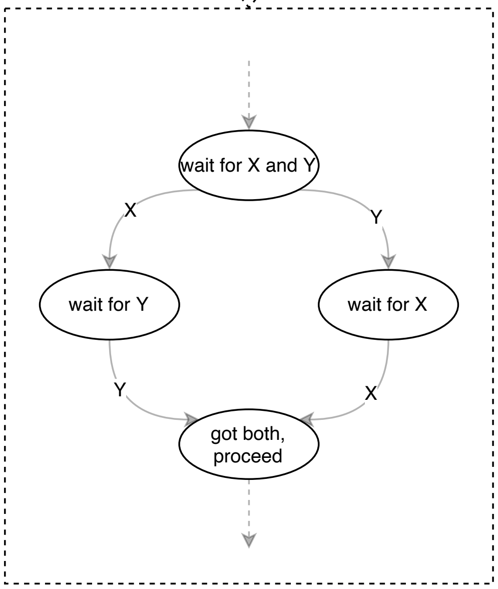


input comes from more than one place

N.B. Message Passing implies that inputs *NEVER* arrive at the same time - one input arrives before the other, or, vice versa

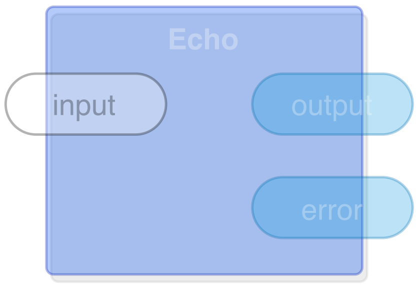
This is the *ONLY* "race condition" - is your hardware/software "fast enough" to differentiate between the arrival instant of 2 messages? If not, then there is ambiguity as to which message arrived first (i.e. a "race condition")

This kind of "race condition" is easy to handle with a state machine. (Harel's Fig. 2)

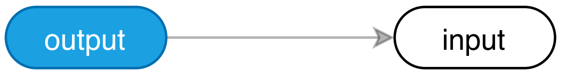


Style, Readability

opacity

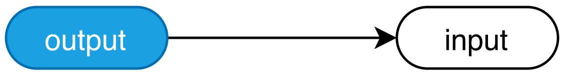


30% opacity 'style'
30% opacity 'text'

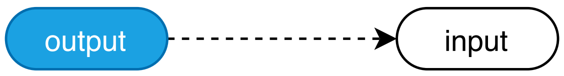


30% opacity 'style' for arrow

line style

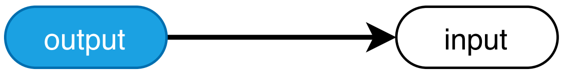


100% opacity 'style' for arrow
1pt line thickness 'style' for arrow
solid 'style' for arrow

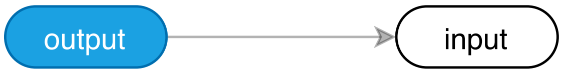


100% opacity 'style' for arrow
1pt line thickness 'style' for arrow
dashed (1) 'style' for arrow

line thickness



100% opacity 'style' for arrow
2pt line thickness 'style' for arrow



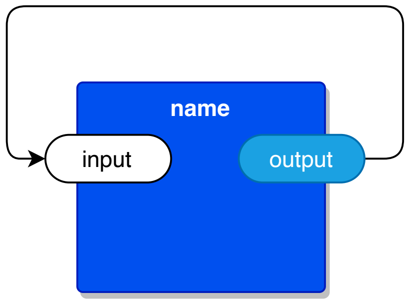
30% opacity 'style' for arrow
1pt line thickness 'style' for arrow

Idioms

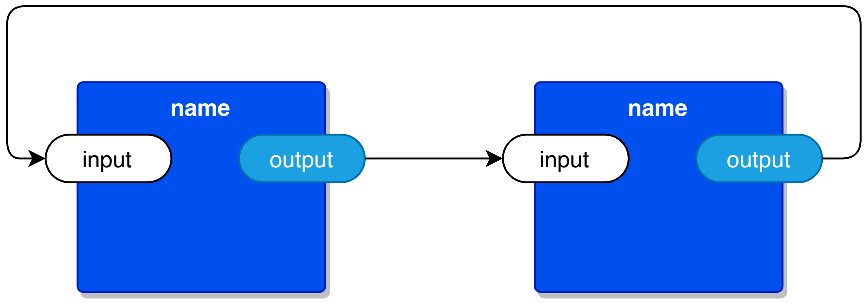
feedback

this is *NOT* recursion - there is a "propagation delay" due to message queing

ie. if a component A outputs X and feeds it back to its own input, then X will not arrive instantaneously, but will arrive "at the next step"



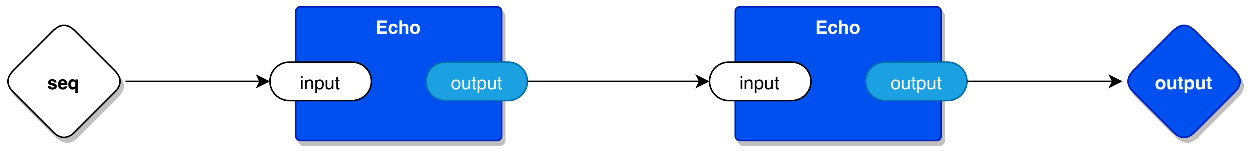
arrow back to component (direct)



arrow back to component (indirect)

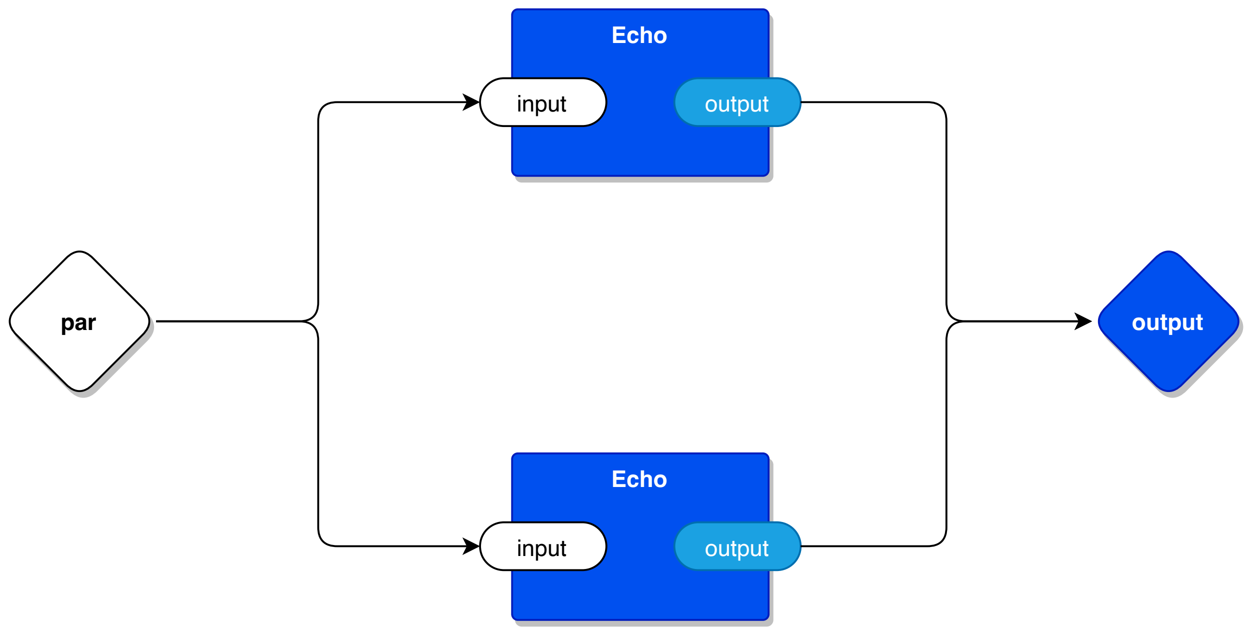
sequential

Sequential Routing



parallel, concurrency

Parallel Routing



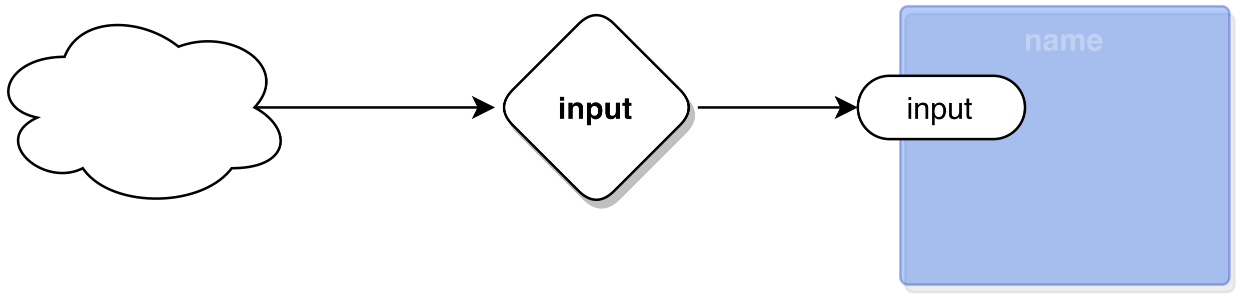
Failure, errors, exceptions

error

there is nothing exceptional about exceptions - exceptions are events (the Architect decides whether the event is special)

Low Level Technicalities

kickoff inject



driver code (in some other programming language) sends the first message into the top component

handle ()

```
echo_handler :: proc(eh: ^Eh, message: ^Message) {
    ...
}
```

A handler() accepts 2 parameters:
1. the Component's "self" (an ė (spelled "eh" in ASCII)),
2. a Message to be handled

Once activated (called), a handler can do anything it wants, but, typically, it process the incoming Message and sends 0, 1, or more Messages back out. Then, it terminates, allowing its Container to route its output Messages.

Typically, Leaf components are written as state machines, which keep and update their state in their own (private) instance data (accessed as a field in their ė structure (eh.instance_data)).

Simple Leaves, like *probe*, don't need to keep state, hence, they ignore their instance data. Such Leaves are initialized with *nil* instance data.

to see the definition of an ė, look in odin/0d/0d.odin (but, you really don't need to know this)

to see the definition of a Message, look in odin/0d/message.odin (which will cause you to look at odin/0d/datum.odin)

send ()

```
    send(eh=eh, port="output", datum=message.datum, causingMessage=nil)
```

`send()` is the way that a Leaf component creates a Message and places the Message on the Leaf's own output queue. The parent Container of the Leaf routes the output Messages created by a Leaf, when the Leaf is finished handling an event.

`send()` is supplied with 4 parameters -

1. the Component's "self" (an ė (spelled "eh" in ASCII)),

2. a port id (in this version, port ids are always strings),

3. a datum (a blob of data - see odin/0d/datum.odin), and,

4. the causing Message, i.e. the Message that the handler is currently responding to [this helps debugging, by essentially keeping a trail of breadcrumbs]

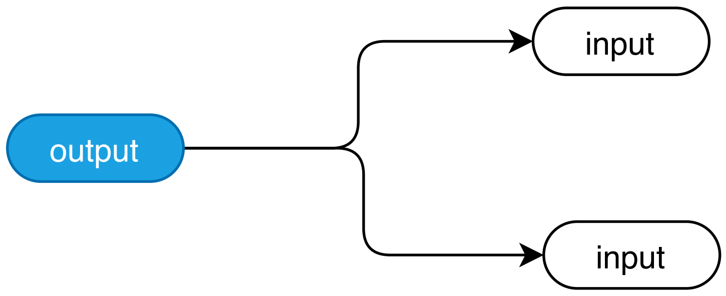
N.B. the unoptimized, debug, version of the code simply needs to keep all Messages in a heap, and, never delete any Messages, hence, making it possible to track message details backwards in time (kind of like a stack trace, but a tree of stack traces is needed for 0D apps), whereas the optimized version depends on the needs of a project - Messages can be deleted "soon" after use, or, kept in an arena, or ... . It is assumed that 0D apps are short-lived, and, that - on today's hardware - we have lots of memory, so, keeping messages is not a big deal. The Biblical Flood[1] method of garbage collection is "good enough" for development. Obviously, a 0D app that runs forever (e.g. a daemon) needs to worry about getting rid of stale Messages and reclaiming storage. See the odin0d repo (a separate repo, not this one) for ideas on implementing optimized versions of 0D. For Production Engineering, i.e. optimization, it helps to know what kind of data (datum) is being used, e.g. datum can be defined as a union of very specific types, allowing the Odin compiler and Production Engineer to produce better, more "efficient" code.

[1] UNIX style garbage collection - wait until a process dies, then reclaim all of its memory.

N.B. this protocol allows a Leaf to produce 0 or more output events in response to any input Message, which is very different from the 1-in 1-out protocol of Functional Programming[2], which hard-wires routing decisions, etc., whereas this Message Passing protocol leaves the routing decisions up to the Software Architect (aka programmer).

[2] FP uses 1-in, 2-out in reality

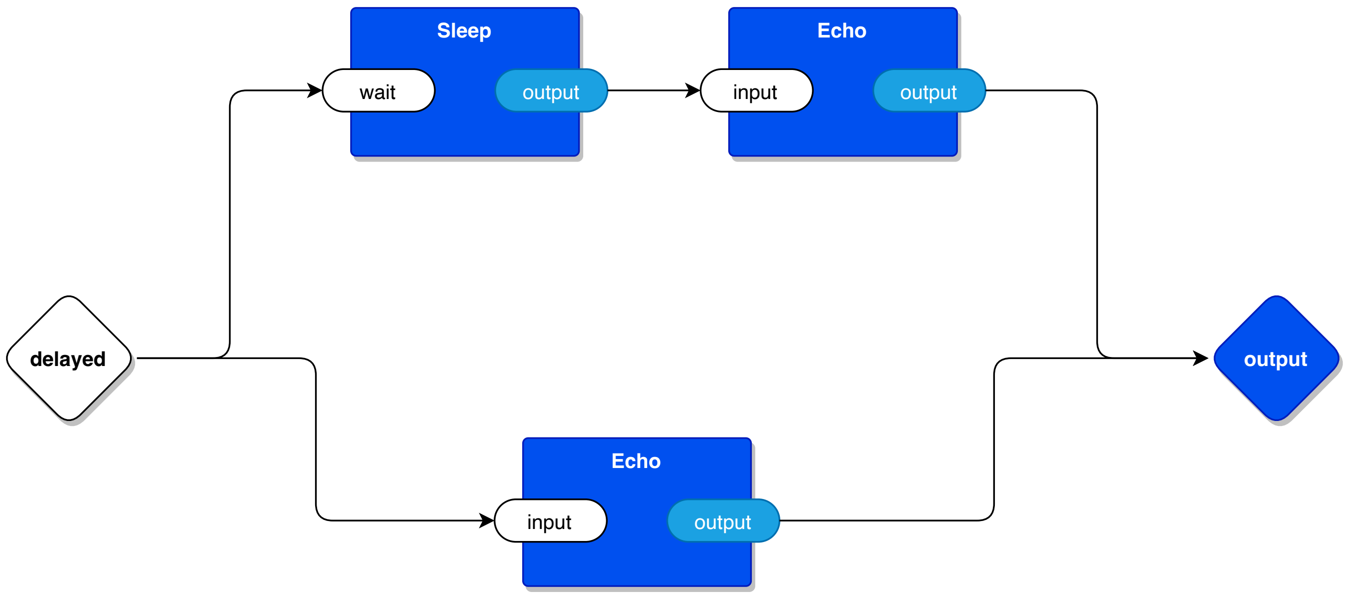
message copying



fan-out implies that runtime must implement Message copying and Message disposal (easy in GC'ed languages, less easy in non-GC'ed languages (but not impossible))

active and idle

Delay



Any Leaf that performs an operation that takes longer than one step (e.g. delay, or sell-out to another language) must set itself as 'active' then reset itself to 'idle' when finished

```
sleep_handler :: proc(eh: ^Eh, message: ^Message) {
    first_time :: proc (m: ^Message) -> bool {
        return ! zd.is_tick (m)
    }
    info := &eh.instance_data.(SleepInfo)
    if first_time (message) {
        info.saved_message = message
        zd.set_active (eh) // tell engine to keep running this component with 'ticks'
    }
    count := info.counter
    count += 1
    if count >= SLEEPDELAY {
        zd.set_idle (eh) // tell engine that we're finally done
        zd.forward (eh=eh, port="output", msg=info.saved_message)
        count = 0
    }
   info.counter = count
}
```

notes

a "gate" is a "port", but, is distinguished by the fact that it belongs to the enclosing Container, i.e. the input/output of a single diagram

"gates" are rendered differently (rhombus instead of pill-shaped) to distinguish them from the ports of children components, and, to help the compiler figure out what a specific diagram means

A single diagram contains components and wiring between the components (and itself, the parent Container). It is not possible to tell if a component is a Leaf or another Container. If a given component is another container, then its diagram *must* appear as another tab (with exactly the same name) in the same draw.io project.

Appendix - See Also

References

https://guitarvydas.github.io/2024/01/06/References.html

Blogs

https://guitarvydas.github.io/

https://publish.obsidian.md/programmingsimplicity (see blogs that begin with a date 202x-xx-xx-)

Videos

https://www.youtube.com/@programmingsimplicity2980

Books

leanpub'ed (disclaimer: leanpub encourages publishing books before they are finalized - these books are WIPs)

https://leanpub.com/u/paul-tarvydas

Discord

https://discord.gg/Jjx62ypR

all welcome, I invite more discussion of these topics, esp. regarding Drawware and 0D

Twitter

@paul_tarvydas

Mastodon

(tbd, advice needed re. most appropriate server(s))