Extending Programming with Diagrammatic Programming Languages

ANONYMOUS AUTHOR(S)

We assert that DPLs - Diagrammatic Programming Languages - can be used as an adjunct syntax for creating programs.

We give an example of a simple DPL syntax and describe a method for creating executable code using diagrams drawn with off-the-shelf graphic editors.

Certain forms of expression are more easily expressed in DPL form rather than TPL - textual programming language - form. TPL-only expression of programs can lead to perceived complexity and other problems. The use of DPLs makes it possible to address these sorts of issues using fresh notations.

ACM Reference Format:

Anonymous Author(s). 2024. Extending Programming with Diagrammatic Programming Languages. 1, 1 (April 2024), 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION - DIAGRAMS AS ADDITIONAL SYNTAX FOR PROGRAMMING LANGUAGES

The goal of programming is to create sequences of instructions - programs - that cause electronic hardware to perform useful tasks.

To that end, programmers have been employing a single kind of syntax for writing programs - text.

Use of text for programs was originally based on the early limitations of electronic hardware and on the convention that the fruits of human thought need to be expressed as equations written on a two-dimensional medium - paper - using printing presses.

Early electronic hardware was only capable of supporting grids of non-overlapping bitmaps called "characters". Today, though, hardware can support vector graphics and overlapping, resizable windows and widgets of various sizes.

Early electronic hardware strongly encouraged the use of a small number of characters. Programming languages standardized on the use of ASCII, which limited the available number of printable characters to about 96 possibilites.

Early electronic hardware was made to be reprogrammable by the invention of the Central Processing Unit - CPU. CPUs were originally designed to be non-reentrant and single threaded. The high cost of early CPU-based computers prevented use of multiple CPUs in projects, thus, causing the insertion of extra software for simulating multiple CPUs on single hardware devices. Today, though, CPUs - and memory - are inexpensive and plentiful, a fact which enables the use of alternate forms of expressing programs. Additionally, the simulation of multiple CPUs via software allowed for direct memory sharing between simulated CPUs by default. Physical hardware CPUs do not share memory by default unless such features are specifically designed into the hardware. The addition of memory sharing by default in CPU simulations has been the cause of various kinds of accidental complexity.

The name "CPU" indicates the prevalent notion in early programming, that hardware designs needed to be controlled by a central authority. This belief is being strained as hardware becomes more distributed in the form of internet, robotics, etc. The notion of centrality and sequentiality was crystallized in early forms of TPL programming, beginning at least with the FORTRAN[5] programming language in 1954.

FORTRAN made the simplifying assumption that CPU subroutines could be used to express mathematical equations ("formulae"). This assumption was calcified and led to the invention of the callstack using shared memory. Preemption was invented when the use of shared memory between subroutines was found to conflict with the simulation of multiple CPUs.

These early kinds of expression continue to permeate most common programming languages even today.

In essence, this paper shows that these habits of thought can be easily re-cast using modern hardware and modern software.

This paper describes a technique for extending the syntax of programming languages.

The new syntax consists of drawings containing simple, closed, graphical figures and arrows.

We use the the term "DPL" - Diagrammatic Programming Languages - when referring to this class of new programming syntax.

We assert that DPLs might be used *alongside* of existing TPLs (Textual Programming Languages) and that DPLs are hybrids of diagram and textual notations. TPLs are useful for expressing a specific kind of program - computation - whereas other syntaxes make it more fruitful to express other kinds of programs that don't lend themselves to being expresses in equation form.

There are multiple possible DPL syntaxes. This paper suggests but a single DPL syntax. It is assumed that a variety of syntaxes might be invented, using the techniques described herein.

This paper touches on some of the key aspects of making compilable and understandable drawings of programs, e.g. 0D, syntactic simplicity, DI (Design Intent), etc.

We do not discuss fundamental DPL principles in this paper and refer the reader to other related writings on these subjects.

This paper is laid out in 8 sections.

(1) Introduction - Diagrams as Additional Syntax for Programming Languages

(2) Motivating Example
(3) Leaf Component "Echo"
(4) Compilation and Execution
(5) Other Examples and Usage - lists brief descriptions and code repositories for other POCs that employee this technique, along with possible future directions.
(6) Fundamental Principles - describes the basic principles for convenient development of programs
(7) Relevant Principles and Issues - lists various principles whose discussion is deferred.
(8) Future - lists possible avenues for further research.

2 EXAMPLE

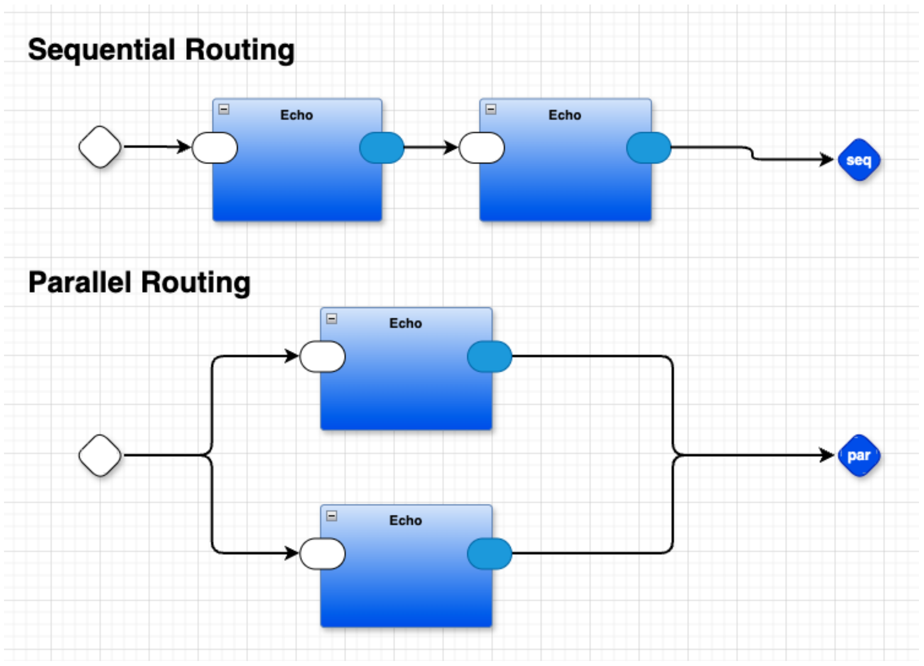

Fig. 1. Motivating Example

Figure 1 shows but one sample of a practical DPL syntax. Variations and improvements on this syntax can be imagined. This syntax is being used to produce actual applications like term-rewriting (*t2t* - text-to-text - rewriting) compilers, LLMs, DSLs for creating DSLs, Visual Shell prototypes, games, etc.

This DPL syntax consists of only a few kinds of closed figures plus arrows plus text associated with the closed figures. Everything else is considered to be a comment, and, is ignored. For example the bold text "Sequential Routing" is ignored. Colors are ignored. Line shapes and line widths are ignored, and so on.

This diagram was drawn using the off-the-shelf diagram editor draw.io[1]. The editor saves the diagram in a modified form of XML, called graphML[3].

The graphML file can be processed using off-the-shelf tools, like an XML parser or a PEG[4]-based parser[6].

In our case, we used the Odin programming language[7] and the XML parsing library that is included in the Odin library.

We choose not to reproduce the code for the actual diagram parser here, due to space limitations. The draw.io-specific parser is under 250 lines of Odin code (including comments and blank lines). The full source code for the functioning diagram parser is in an open-source repo[8].

The diagram parser produces internal data structures in the host programming language and/or JSON[9] files.

A draw.io file can contain several diagrams, each on a separate tab (window) in the editor.

The structure of the information needed by the DPL runtime is minimal, consisting of about four (4) fields for each diagram:

(1) The filename of the draw.io drawing
(2) The tab-name of the top-most diagram.
(3) A bag of Children components.
(4) A bag of Connections between children (including connections to/from the container itself).

and, is embodied in JSON as ...

```
"file": "helloworld0d.drawio",
"name": "main",
"children": [
    {
        "name": "Echo",
        "id": 6
    },
... (2 more children referring to the same template "Echo", but assigned
Different ids) ...
],
"connections": [
    {
        "dir": 0,
        "source": {
        "name": "",
        "id": 0
    },
    "source_port": "",
    "target": {
        "name": "Echo",
        "id": 6
    },
    "target_port": ""
},
... [7 more connections as per the diagram] ...
]
```

The .drawio file containing this information is verbose and contains a great deal of graphic layout information that is not relevant at the semantic level.

(In the following, we wave our hands in an attempt to simplify discussion of our approach...)

A Child derived from the Echo template is saved in the .drawio file as: (shown with artificial line breaks)

```
        <mxCell id="kChx-vSpJG1CHjQ2RL-N-43" value="Echo" style="rounded=1; \
                  whiteSpace=wrap;html=1;sketch=0;container=1;recursiveResize=0;\
                  verticalAlign=top;arcSize=6;fontStyle=1;autosize=0;points=[];\
                  absoluteArcSize=1;shadow=1;strokeColor=#6c8ebf;fillColor=#dae8fc;\
                  fontFamily=Helvetica;fontSize=11;gradientColor=#0050EF;\
                  fontColor=default;" parent="1" vertex="1">
          <mxGeometry x="172" y="190.5" width="138" height="100" as="geometry">
          <mxRectangle x="-98" y="-1230" width="99" height="26" as="alternateBounds" />
          </mxGeometry>
        </mxCell>
```

In this DPL, the *style* information is mostly noise from the perspective of semantics. We do need, though, to peer into the style information to determine if the mxCell is a node or a connector (vertex="1" versus edge="1").

When we remove the *style* noise, we get pseudo-XML something like:

```
        <mxCell id="kChx-vSpJG1CHjQ2RL-N-43" value="Echo" node=True
          <mxGeometry x="172" y="190.5" width="138" height="100" as="geometry">
          <mxRectangle x="-98" y="-1230" width="99" height="26" as="alternateBounds" />
          </mxGeometry>
        </mxCell>
```

Similarly, the *mxGeometry* information affects the visual display of the diagram, but has no semantic value for this DPL.

Culling that form of noise out, we get

```
        <mxCell id="kChx-vSpJG1CHjQ2RL-N-43" value="Echo" node=True>
        </mxCell>
```

Furthermore, the id can be internally hashed on a per-file manner, resulting in even less noise:

```
        <mxCell id=6 value="Echo" node=True>
        </mxCell>
```

And, from here we can derive the JSON form for a child component

```
...
    {
        "name": "Echo",
        "id": 6
    },
...
```

All of the above is currently achieved using an XML parser and by cherry-picking information out of the seemingly-complicated XML data. We argue, though, that this could, also, be replaced by a PEG parser which performs t2t (text-to-text) transformations.

In this example, the component id=6, derived from the Echo template, has one input port and one output port, seen in the graphml as

```
...
        <mxCell id="kChx-vSpJG1CHjQ2RL-N-44" value="" style="rounded=1;\
             whiteSpace=wrap;html=1;sketch=0;points=[[0,0.5,0,0,0],[1,0.5,0,0,0]];\
             arcSize=50;fontFamily=Helvetica;fontSize=11;fontColor=default;" \
             parent="kChx-vSpJG1CHjQ2RL-N-43" vertex="1">
          <mxGeometry x="-16.75" y="27.5" width="36.75" height="25" as="geometry" />
        </mxCell>
        <mxCell id="kChx-vSpJG1CHjQ2RL-N-45" value="" style="rounded=1;\
            whiteSpace=wrap;html=1;sketch=0;points=[[0,0.5,0,0,0],\
            [1,0.5,0,0,0]];fillColor=#1ba1e2;fontColor=#ffffff;strokeColor=#006EAF;a\
            rcSize=50;fontFamily=Helvetica;fontSize=11;" \
            parent="kChx-vSpJG1CHjQ2RL-N-43" vertex="1">
          <mxGeometry x="120" y="27.5" width="36" height="25" as="geometry" />
        </mxCell>
...
```

Here, the ports refer to parent="kChx-vSpJG1CHjQ2RL-N-43" and contain a great deal of graphic information that has no bearing on the semantics. Culling, we get:

```
...
```

```
        <mxCell id="kChx-vSpJG1CHjQ2RL-N-44" value="" parent=6 node=True>
        </mxCell>
        <mxCell id="kChx-vSpJG1CHjQ2RL-N-45" value="" parent=6 node=True>
        </mxCell>
...
```

Note that "" is a valid port name - the null string. The input port and the output port have the same name (the null string), but, the names do not conflict since they are in different namespaces (input namespace of template "Echo" vs. output namespace of template "Echo").

Grinding further through the XML, we see several connections, one of which contains a reference to "kChx-vSpJG1CHjQ2RL-N-45". In this case, the reference lists the id as a *source*. This implies that "kChx-vSpJG1CHjQ2RL-N-45" is an output port (there is more nuance to this, but, for simplicity, we'll skip over the nuance for now).

```
...
        <mxCell id="fJIQP-pnwGc3qDgAUfr9-5" style="edgeStyle=orthogonalEdgeStyle;\
         curved=0;rounded=1;orthogonalLoop=1;jettySize=auto;html=1;exitX=1;exitY=0.5;\
           exitDx=0;exitDy=0;exitPerimeter=0;entryX=0;entryY=0.5;entryDx=0;entryDy=0;\
            entryPerimeter=0;strokeColor=default;strokeWidth=2;align=center;\
            verticalAlign=middle;fontFamily=Helvetica;fontSize=11;fontColor=default;\
            labelBackgroundColor=default;endArrow=classic;" parent="1" \
           source="kChx-vSpJG1CHjQ2RL-N-45" target="kChx-vSpJG1CHjQ2RL-N-74" edge="1">
        </mxCell>
...
```

The connection can be culled down to semantically interesting information, somewhat like

```
...
        <mxCell id="fJIQP-pnwGc3qDgAUfr9-5" parent="1"
             source="kChx-vSpJG1CHjQ2RL-N-45"
             target="kChx-vSpJG1CHjQ2RL-N-74"
             connector=True>
        </mxCell>
...
```

Since the source and the targets are children of children of the drawing, the connection is deemed to be of type *across*. We see these relationships because "...-74" has parent "...-73" which has parent "1" and "...-45" has parent "...-43 which has parent="1". Aside: in this DPL, children components cannot SEND messages directly to other children - they must rely on the parent ("1") to route their messages. This kind of indirect routing encourages design flexibility, i.e. LEGO®-like composition of components. We strongly recommend this kind of routing in DPL designs.

3 CONTAINER AND LEAF COMPONENTS

The diagram in Figure 1 represents two (2) kinds of components.

The main diagram is, itself, a component. It is a Container which displays its implementation in diagrammatic form as a composition of four other components with connections between them.

Containers can contain other Containers or other Leaf components. In this very simple example, the child components happen to all be Leaf components and, happen to all be derived from the same template - “Echo”. In more practical drawings, this is not usually the case - Containers generally contain components derived from a variety of templates.

Leaf components contain executable code. Leaves do not contain other components.

Children components are drawn as rectangles and contain other, smaller rectangles representing input and output ports for each child component.

Connections are one-way and are drawn as arrows. Line width and style are ignored in this DPL. This allows software architects to choose graphic representations on a per-project basis, emphasizing certain aspects of the design and deemphasizing other aspects. In this example, we used 2pt line thickness to indicate “important” connections. Note that implementing bi-directional connections requires extra software. Bi-directional connections tend to obfuscate elements of the design.

Input and output ports of the diagram - “gates” - are drawn as rhombuses. T he kind of each port or gate is determined by how arrows are attached to it. A port that has an arrowhead attached to it is an “input port”, while “output ports” have arrowtails attached to them. Gates, though, are opposite in sense. Gates that have arrowheads attached to them are “output gates” while gates that have arrowtails attached to them are “input gates”.

All rectangles and rhombuses can have one text string associated with them. The text string is contained within the rectangles. Names represent template classes for components. For ports and gates, names are tags that are used by components to signify the kind of messages being received and sent. Note that empty strings are legal names and are reserved to mean UNIX®-like standard inputs and standard outputs.

The reader might notice a resemblance of this DPL with electronic components. In electronics, components are called “ICs”, and ports are called “pins”. Container diagrams are akin to “schematics”. ICs are black boxes encased in epoxy or plastic. One cannot peer “inside” of any IC to determine how it is implemented. Templates are called “connection diagrams” (found on IC datasheets). Components in this DPL and electronics ICs share the characteristic that they are completely asynchronous and completely isolated from one another. In contrast, software *functions* and *libraries* are tightly coupled and synchronous. Functions imply ad-hoc blocking - the caller must block indefinitely waiting for a response from the callee, and, the caller cannot rely on timing characteristics of callees due to the fact that callees can call other callees synchronously and block waiting for their response.

Each component instance - whether Container or Leaf - has exactly one input queue and exactly one output queue.

A key feature of any DPL is that components are truly isolated from one another due to the use of FIFO queues instead of reliance on LIFO callstacks. This conforms with common human understanding of the operation of drawings, objects and actors. Other aspects of DPLs contribute to component isolation, in particular the use of input and output queues and the restrictions on message routing (components cannot directly send messages to other components. Message routing is handled solely by their parent Containers).

One cannot determine “from the outside” whether a component is a Container or a Leaf, nor know anything about its implementation. One cannot know if a component contains state or is a pure function. In fact, knowledge of these kinds of details is unnecessary. Programmers can dig deeper into the innards of components if they want to know such low-level details.

Components process each incoming message. Processing is based on internal component state and the port tag contained in a message. A message is a 2-tuple {port X payload}.

Output message processing consists of placing tagged messages {tag X payload} onto the component’s (single) output queue. Output messages are sent in a deferred manner. Output messages sit in a component’s output queue until the component has finished processing a step of its potential actions and returns control back to its parent (which is always a Container). The parent container empties a child’s output queue and routes messages in first to last (oldest to newest) order of output message creation (note that “recursion” works in an opposite manner, in a last to first order - most recent first).

Output queues contain messages with port tags that are relative to the *sender*. Input queues contains messages with port tags that are relative to the *receiver*. Routers, i.e. Containers, must map from output tag of the sender to the appropriate input tags of the receivers.

Component instances need not be explicitly named. Their (x,y) position on a diagram uniquely identifies each instance. This agrees with normal human convention - visual placement on the diagram has semantic meaning.

Connections are triples {direction X sender X receiver}.

Fan-in and fan-out are supported. Fan-in means that a single input port/gate can receive messages from multiple sources. This is straight-forward to implement using the FIFO queue structure of components. Fan-out, though, means that a single port/gate can send messages to several receivers. This requires copying of message payloads or the application of copy-on-write semantics in the receivers. There is no semantic difference between these two techniques - the choice of implementation (copying vs. copy-on-write) is purely one of optimization and does not matter at the design stages. Efficiency of implementation matters only during Production Engineering stages, i.e. when a design is stable enough to be productized.

Figure 1 shows, both, fan-in and fan-out on the subnet labelled "Parallel Routing".

The output gate "par" shows fan-in, as it receives inputs from two different component instances. The outputs from the two "Echo" instances do not collide. The outputs are queued up on the diagram’s output queue in the order that they were generated. See Figure 2.

The corresponding input gate demonstrates fan out. Any message on the input gate is routed to both instances of the "Echo" component on the "Parallel Routing" subnet.See Figure 3.

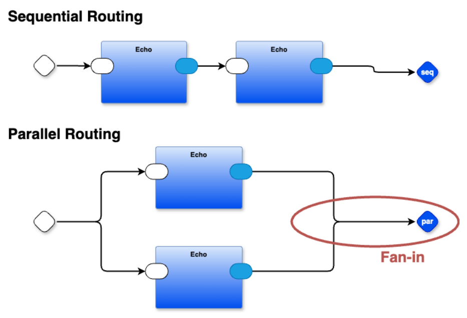

Fig. 2. Fan In

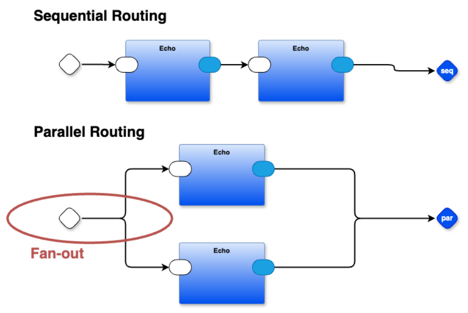

Fig. 3. Fan Out

Furthermore, fan-out requires atomic routing. A single output message must be delivered to all receivers in an atomic fashion without allowing interleaving messages to arrive from other sources. This requirement ensures that messages are queued in a consistent manner relative to one another. The actual, absolute timestamp of any message does not matter, and the final order of messages does not matter, but message relative-order does matter. If "A" arrives before "B" at one receiver, then, "A" must arrive before "B" at *all* receivers. Physics informs us that, ultimately, we cannot tell if "A" arrived before "B" or if "B" arrived before "A", and that that is the only real "race condition". Programmers can deal with that single kind of "race condition" as long as message-relativity is preserved. In general, though, message relativity cannot be preserved across large distances ("A" might arrive before "B" at one receiver, but, "B" might arrive before "A" at another receiver because "B" has less-far to travel). Preserving message relativity in-the-small, inside a single application, makes it easier to write such application programs. Note that this DPL is "yet another notation", like function-based programming, which can simplify programming in many cases, but, cannot be used in *all* cases.

Fan-out may be an inconvenience in the theoretical sense, but, is a vital feature for usability of DPLs. For example, fan-out makes it possible to abstract diagrams by grouping components together and pushing them over onto other diagrams, leaving only a single component with fewer ports on the originating diagram.

This kind of usage is fundamental to one’s being able to layer a design in meaningful ways. In contrast, function-based programming relies on a single expression for layering - the function (aka “lambda”). The paucity of expression in function-based-only-programming leads to obfuscation of design intent (“DI”) and makes designs appear more complicated than necessary to casual readers.

In this DPL, there are four (4) connection directions

- down
- across
- up
- through.

The sender information in connections must include a reference to a component instance and a port. The receiver information in connections must include a reference to a component instance and a port. As already mentioned, mapping from sender-ports to receiver-ports must be done by parent containers during the process of message routing.

3.1 Container Component

Each diagram is compiled into a container component containing the following information:

- template name - the name of the diagram
- bag of children 2-tuples {template X instance-id}
- bag of connection 3-tuples {direction X {sender X port} X {receiver X port}}

The example diagram is compiled into a container component containing the following information:

- name=“main”
- children=[{“Echo”, 1}, {“Echo”, 2}, {“Echo”, 3}, {“Echo”, 4}]
- connections=[

      {down,{self,“”},{1,“”}},{across,{1,“”},{2,“”}},{up,{2,“”},{self,“seq”}}
      {down,{self,“”},{3,“”}},{up,{3,“”},{self,“par”}},
      {down,{self,“”},{4,“”}},{up,{2,“”},{self,“par”}}

  ]

Further below, we will see that the compiler chose different values for ids. Specifically, one of the children is deemed to have id=6, instead of the of one of the idealized ids listed above.

3.2 Leaf Component "Echo"

In our implementation, we used the Odin programming language for creating the diagram parser, but, we use the Python programming language to load and execute the parsed results.

Figure 1 refers to a Leaf component template called "Echo". The "Echo" component is instantiated three (3) times. Leaf components are implemented with code in some host language (Python, in our case). A sample of such an implementation is shown below. Software components appear as rectangles on the drawing. The implementation provides two (2) entry points for each software component:

- The code to instantiate a component from the "Echo" template. In Figure 1, the "Echo" code creates three (3) instances, each being unique, but, derived from the same template.
- The code to handle incoming messages for each instance derived from the template. In this example, an "Echo" component does nothing more than forwarding the received message. The *handler* code is activated for *each* incoming message.

```
def Echo (reg, owner, name, template_data):
    name_with_id = gensym ("Echo")
    return make_leaf (name_with_id, owner, None, Echo_handler)

def Echo_handler (eh, msg):
    send_string (eh, "", msg.datum.srepr (), msg)
```

4 COMPILATION AND EXECUTION

Compilation and execution of this DPL consists of the steps listed below. Note that the diagrams are rough sketches intentionally simplified for overview purposes only.

(1) Convert DPL program diagrams to JSON.

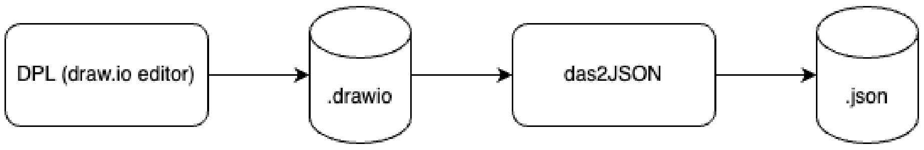

Fig. 4. Convert diagram to JSON

Transpilation of the diagrams (XML) into JSON (or internal data structures, if efficiency is at a premium). The diagrams represent *templates* for components.
In our implementation, `das2json` is implemented[10] in the Odin programming language. The process begins with a straightforward call to the XML parsing library. The XML data is then deconstructed into a convenient internal format (see `0d/ir/ir_odin/ir.odin` in the code repository).

```
    xml, xml_err := xml.parse(file)
...
```

(2) Load component templates from JSON.
Ingestion of the JSON, or internal data structures into an internal database that we call a *registry*. Registries can be implemented as a *dictionaries, maps* or *databases.*

```
    routings = json.loads(json_data)
```

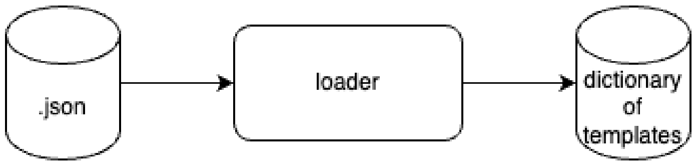

Fig. 5. Loader

...

(3) Instantiate system.

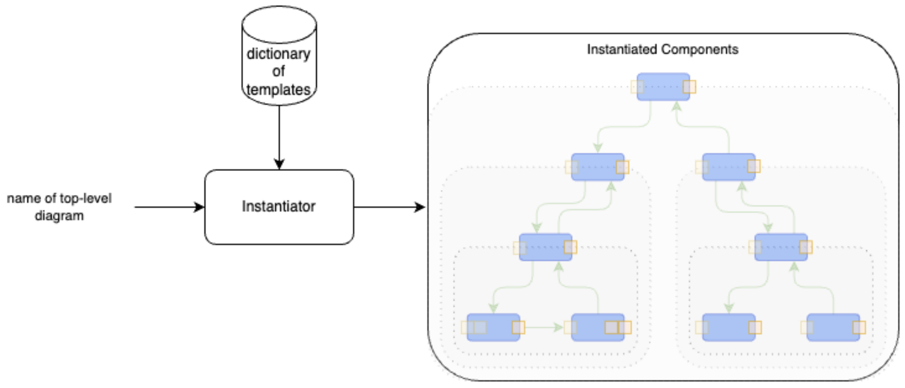

Fig. 6. Instantiate

Instantiation[11] of an application beginning with a top-level diagram and proceeds downwards to instantiate all children components needed in the top-level diagram and, recursively, instantiating the childrens' children. Components are instantiated based on their DPL templates. Containers are instantiated by instantiating all child components and all routings between components. Note that more than one child component can refer to the same DPL template. Components must be uniquely instantiated - typically their (x,y) position is enough to differentiate components, but, `draw.io` assigns unique id's to each template component, which we ended up using instead of relying on (x,y) graphical information. Assigning unique id's is a convenience, not a requirement, since the same information can be inferenced using multiple approaches.

```
...
    main_container = get_component_instance(pregistry,
        main_container_name,
        owner=None)

...
```

Software components are fully isolated from one another through the use of FIFO queues.

(4) Inject first message.

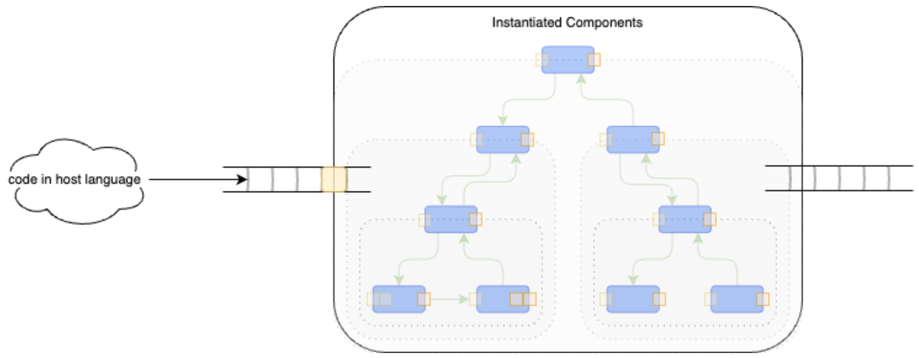

Fig. 7. Inject first message(s)

The first message is constructed using the host language. The message is injected[12] into the input queue of the top-most component.

```
...
def start_function (root_project, root_0D, arg, main_container):
    arg = new_datum_string (arg)
    msg = make_message("", arg)
    inject (main_container, msg)
...
```

(5) Run. The application is executed.

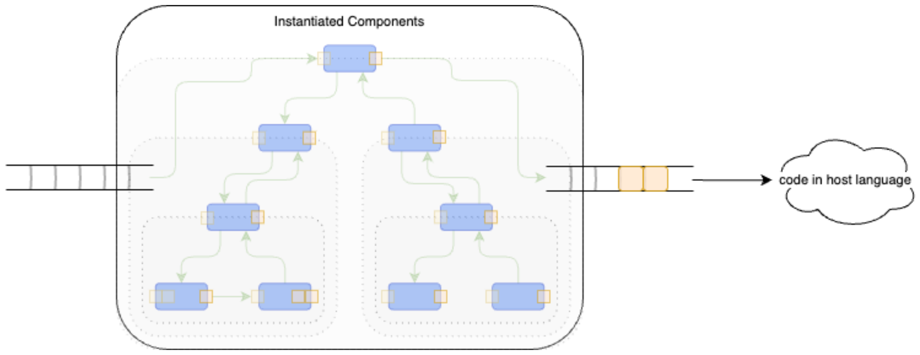

Fig. 8. Run

The application is interpreted by repeatedly delivering messages along connections, recursively, until no messages remain in any input queues.

It is possible to invoke already-written library code written in some other host language using an activity indicator - see the code repo for more details.
Containers must consume only one input message at a time while waiting for all children to reach quiescence.

```
...
    if not load_errors:
        injectfn (root_project, root_0D, arg, main_container)
...
```

(6) Display outputs.

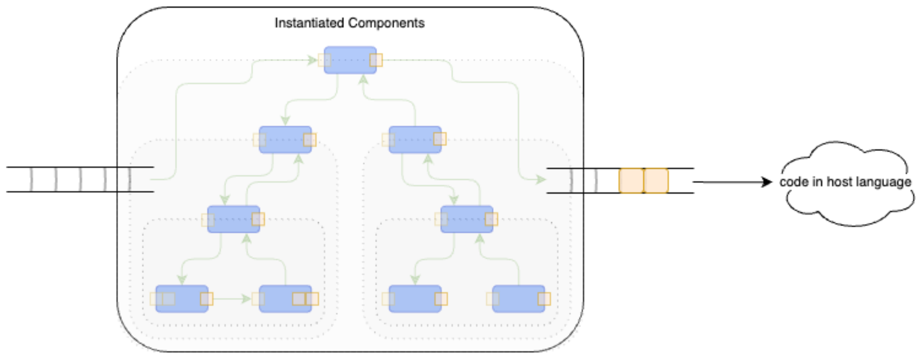

Fig. 9. Display Outputs

The outputs in the output queue of the top-level component are displayed using the host language.

```
...
    dump_outputs (main_container)
...
```

The Python version of the runtime system consists of roughly 9 files of code consisting of about 2,500 lines of code (including blank lines and comments) and is supported by a standard library consisting of another 5 files of code containing about 430 lines of code.

At the basic level, runtime component instances are asynchronous, but, use named and anonymous functions and closures, instead of operating system processes. As such, programs written this way "feel" like *programs* instead of as *processes* in terms of efficiency.

5 OTHER EXAMPLES AND USAGE

We have built working examples of various other technologies, using the DPL described in this paper, including,

- VSH[13] a Visual SHell that works like /bin/sh pipelines employing more input and output ports,

- arith0d[14] OhmJS' arithmetic grammar that concurrently emits four (4) target languages (Javascript, Python, Common Lisp, WASM),
- LLM0D[16] a single component diagram that uses the Go programming language as a host language to connect to openai's LLM for experimenting with LLMs,
- transpiler[15] a workbench for creating new textual syntaxes, uses OhmJS more than once in a pipeline
- kinopio2md[20] experiment for converting Kinopio "mindmaps" into markdown based on connections between ideas expressed in Kinopio
- Scheme to Javascript[17], Scheme to Python[18] - conversion of Nils Holm's Scheme code used in "Prolog Control Flow in 6 Slides" to produce a Prolog-like inferencing engine for Javascript, currently working on conversion to Python
- delay0d[19]
- dc0d[**?**] game engine based on Chris Marten's "dungeon crawler" example[2]; including gen0D generates components (in Odin) from the diagram (instead of interpreting the diagram, gen0D creates leaf components that are used in conjunction with the diagram)
- 0D odin[21] a DPL engine written in the Odin programming language (Odin is statically typed, and, does not provide garbage collection, similar to languages like C)
- Crystal 0D[22] a DPL engine written in the Crystal programming language, which is currently being used in a production system
- 0D cookbook[23] ongoing effort to document all standard DPL components as draw.io diagrams

6 FUNDAMENTAL PRINCIPLES

Software components are fully isolated from one another through the use of FIFO queues. Using the LIFO callstack for inter-component communication leads to hidden dependencies which leads to avoidance of DPLs. This is a requirement for enabling LEGO®-like composition of software components. FIFO-based callstacks only give the illusion of LEGO®-like composition of software libraries, but, actually result in dynamic, tight coupling of components.

There must be exactly one input queue and one output queue per component. We do not use, for semantic reasons, one queue per port. In general, single queues allow programs to track relative time-ordering of messages - an important feature when programming devices that perform sequencing instead of computation. In addition, multiple queues can lead to deadlock issues. These issues still exist in the large, but are mitigated through the use of single input and output queues and tend not to occur in hidden, unexpected ways.

Connections between components are triples that contain a routing direction. There are four (4) kinds of routings as discussed in Section 3. Routings are handled solely by parent Containers of components. Components, themselves, cannot route messages directly to other components, but, must leave output messages on their own output queues. Outputs are deferred, a parent Container of a component instance delivers the output messages, according to the routing connections, after the component has finished executing a step of its execution.

7 RELEVANT PRINCIPLES AND ISSUES

We do not discuss the following principles in detail in this paper, due to reasons of space.

- Structured Message Passing
- Rule of 7
- Parental Authority
- This technique implies no built-in paradigm. Components are asynchronous and ordered-in-time by default. The default behaviour can be explicitly altered by software architects.
- SEND in addition to CALL
- Locality of Reference
- Scoping of names
- Every diagram is stable, every diagram can be understood on its own. Children cannot change the behaviour of their parents.
- CPUs are meant to be single-threaded
- There is no single happy path in many applications. In general, applications can have multiple happy paths of equal importance.
- Subroutines are not functions
- Hidden dependencies
- Build-and-forget
- Enable breaking free of the function-based paradigm.
- Building new paradigms using existing tools.
- *t2t* text-to-text transpilation pipelines.
- Multiple syntaxes - notations - instead of general purpose programming languages.
- Using existing GPLs as *assemblers*.
- Writing code that writes code.
- Inferencing semantic information from diagrams using various techniques, when the graphical editor does not provide sufficient information. E.g. using Prolog to inference space-relationships, based on (x,y) coordinates and simple mathematical relationships like *intersection*, *larger/smaller*, *above/below/left-of/right-of*
- The low-level concepts of *loop* and *recursion* don't make sense in a distributed environment, like the internet. Modern programming languages can be used to program single *nodes* but are cumbersome notations for expressing designs of systems of networked nodes.
- if-then-else is too low level and ad-hoc for expressing control-flow

8 FUTURE

- Compilation, in addition to the interpretation, of DPLs.
- Parsing XML using the *Transpiler* component instead of calling XML parsing libraries.
- Testing, unit testing, coverage testing.

9 REFERENCES

REFERENCES

[1] Diagrams.net. https://app.diagrams.net
[2] Martens, Chris. Ceptre: A Language for Modeling Generative Interactive Systems. https://www.cs.cmu.edu/~cmartens/ceptre.pdf (Accessed: January 19, 2024).
[3] https://en.wikipedia.org/wiki/GraphML (Accessed: April 22, 2024)
[4] https://en.wikipedia.org/wiki/Parsing_expression_grammar (Accessed: April 22, 2024)
[5] https://en.wikipedia.org/wiki/Fortran (Accessed: April 22, 2024)
[6] https://ohmjs.org (Accessed: April 22, 2024)
[7] https://odin-lang.org (Accessed: April 22, 2024)
[8] *anonymous repository*
[9] https://www.json.org/json-en.html (Accessed: April 22, 2024)
[10] *anonymous repository*
[11] *anonymous repository*
[12] *anonymous repository*
[13] *anonymous repository*
[14] *anonymous repository*
[15] *anonymous repository*
[16] *anonymous repository*
[17] *anonymous repository*
[18] *anonymous repository*
[19] *anonymous repository*
[20] *anonymous repository*
[21] *anonymous repository*
[22] *anonymous repository*
[23] *anonymous repository*