Towards DPLs

Abstract

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 name “DPL” - Diagrammatic Programming Languages - when referring to this class of new programming syntax.

We assert that DPLs might be used *along side* of existing TPLs (Textual Programming Languages) and that DPLs are hybrids of diagram and textual notations.

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.

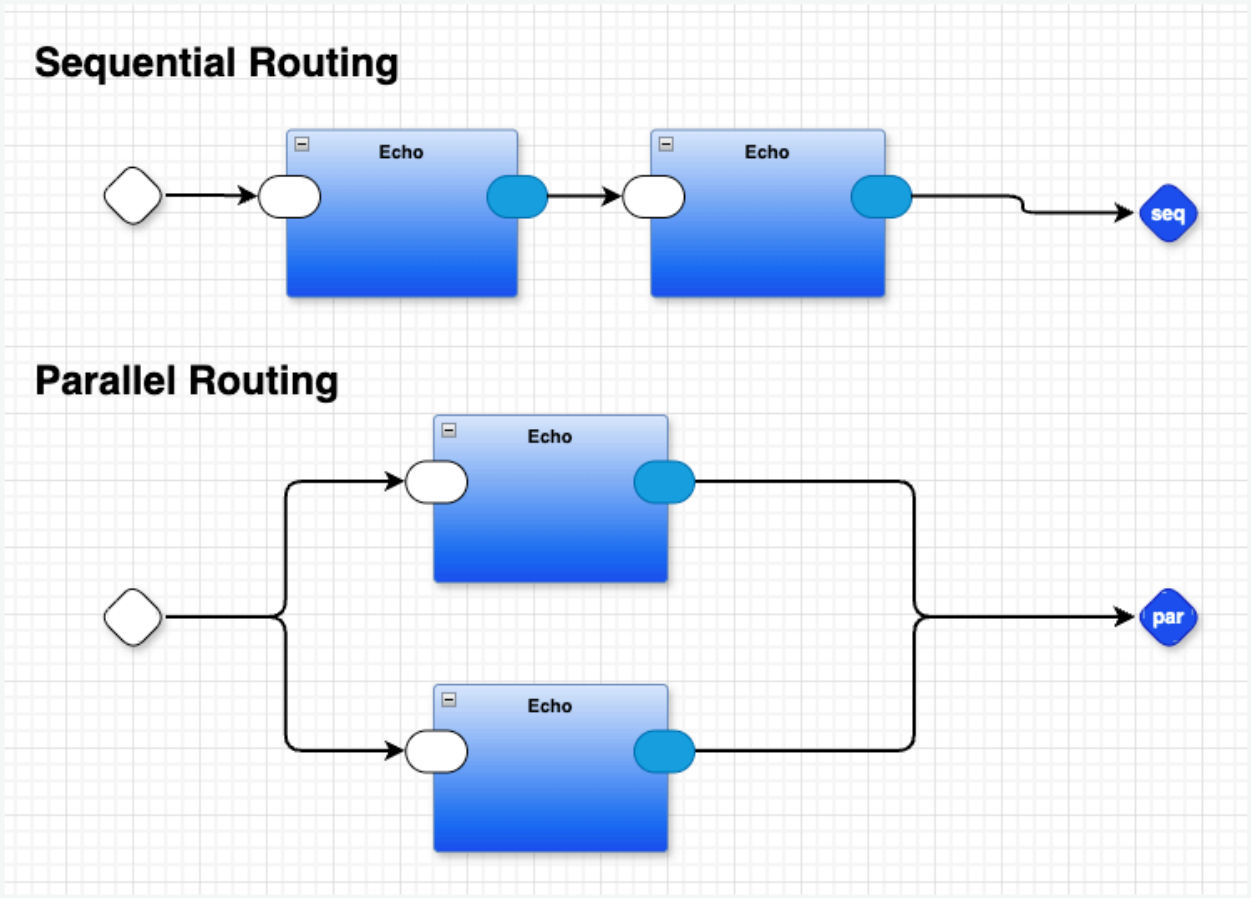
This paper touches on some of the key aspects of making compilable and understandable drawings of programs, e.g. 0D, syntactic simplicity, DI (Design Intent), etc.

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

[TBD] This paper is laid out in XXX (???) sections.

1. Motivating Example
2. Compilation and Execution
3. Other Examples and Extensions lists brief descriptions and code repositories for other POCs that employee this technique, along with possible future directions.
4. Principles lays out the basic principles for convenient development of programs

Example



The above diagram is a motivating example.

The above diagram shows but one sample of a practical DPL syntax. Variations and improvements on this syntax can be imagined. The above 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 belonging to 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*[2].

The graphML file can be processed using off-the-shelf tools, like an XML parser or a PEG-based parser[3][4].

In our case, we used the Odin programming language[5] and the XML parsing library that is included in the Odin library[6].

We choose not to reproduce the code for the actual diagram parser here, due to space limitations for this paper. The full source code for the functioning diagram parser is in an open-source repo[7].

The diagram parser produces internal data structures in the host programming language and/or JSON[8] 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:

```
      "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] ...
```

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 parent containing diagram).

Leaf Component “Echo”

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 this case). A sample of such an implementation is shown below. The implementation provides two (2) entry points

1. 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.
2. The code to handle incoming messages for each instance derived from this template. In this example, an “Echo” component does nothing more than forwarding the received message. The *handler* code is activated for *each* incoming message.

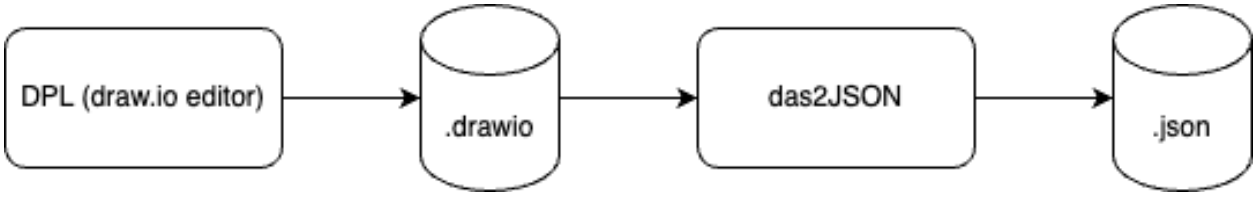
```
def Echo (reg, owner, name, template_data):
    name_with_id = gensym ("Echo")
    return make_leaf (name_with_id, owner, None, Echo_handler)

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

Compilation and Execution

Compilation and execution of this DPL consists of the following steps:

1. Convert DPL program diagrams to JSON

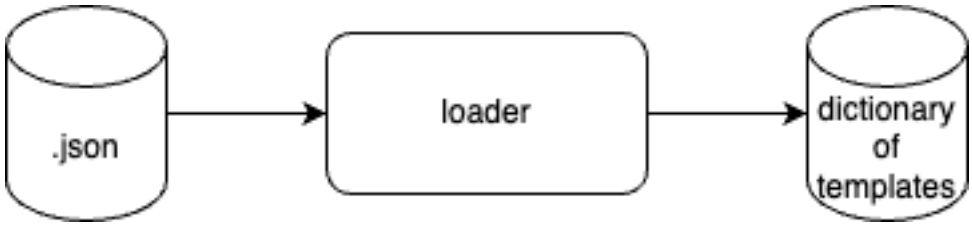


Transpilation of the diagrams (XML) into JSON (or internal data structures, if efficiency is at a premium). The diagrams represent *templates* for components.

In our implementation, das2json[1] is implemented[2] in the Odin programming language. The process begins with a straight-forward call to the XML parsing library. The XML data is then deconstructed into a convenient internal format (see 0d/ir/ir_odin)

```
    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* (which can be implemented as a *dictionary, map* or a *database*).

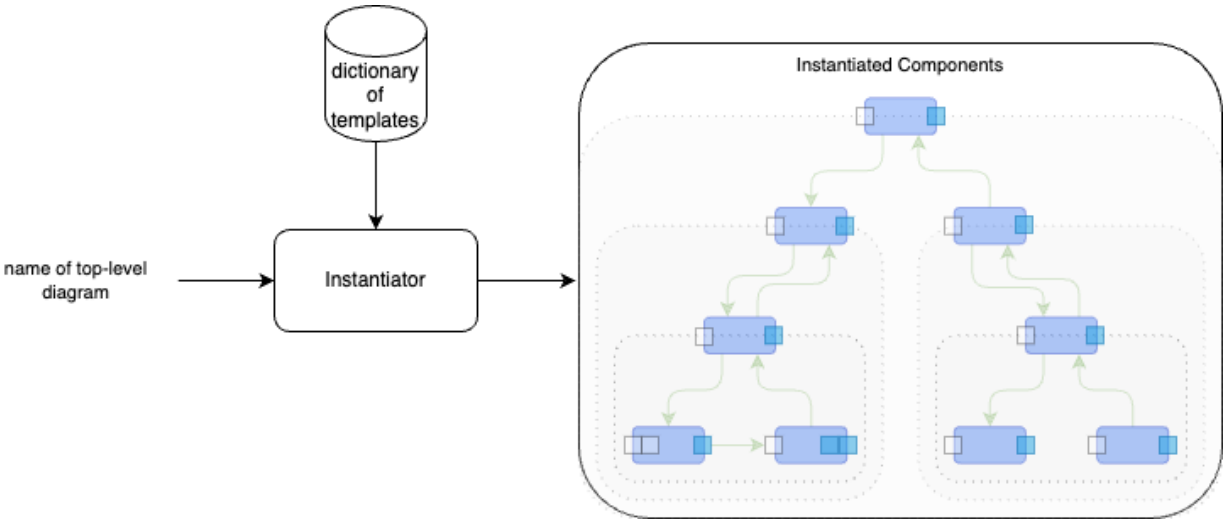
---

[1] DaS means Diagrams as Syntax.

[2] The implementation can be found in https://github.com/guitarvydas/onward/tree/main/helloworld0d/0D/das2json

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

3. Instantiate system

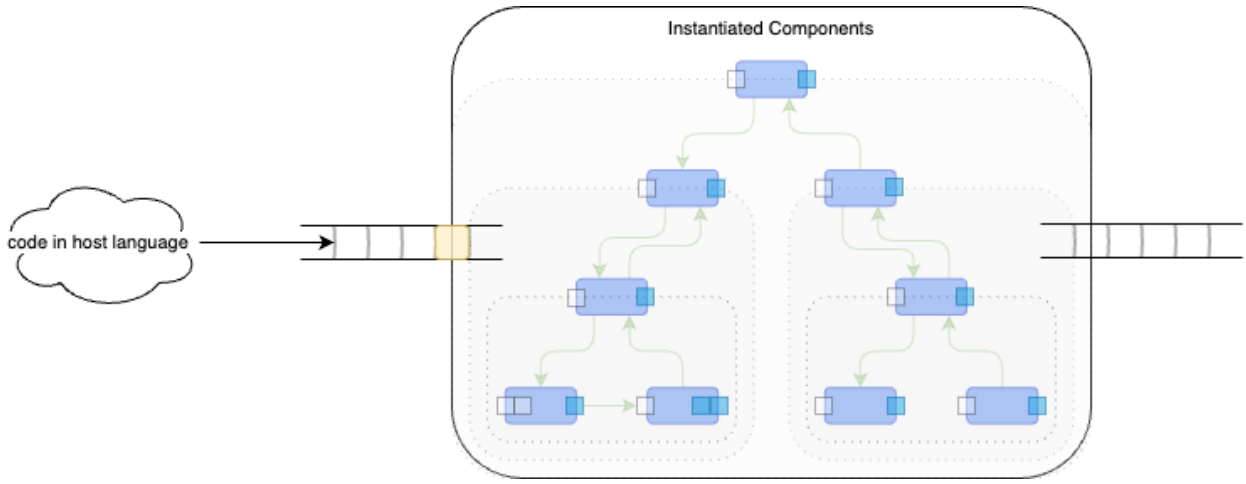


Instantiation[3] of an application beginning with a top-level diagram and proceeding downwards to instantiate all children components needed in the top-level diagram and, recursively, instantiating their children. Components are instantiated based on their DPL templates. Containers are instantiated by 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.

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

[3] see https://github.com/guitarvydas/onward/blob/main/helloworld0d/0D/python/std/run.py

4. Inject first message

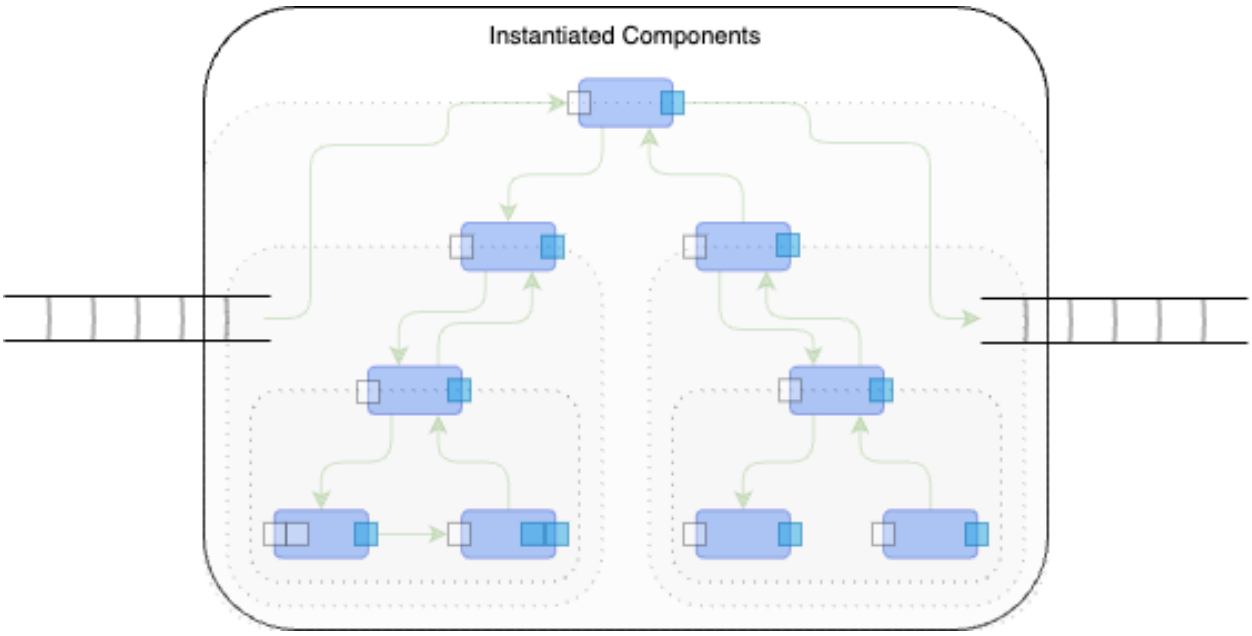


The first message is constructed using the host language. The message is injected[4] into the input queue of the top-most component.

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

[4] see https://github.com/guitarvydas/onward/blob/main/helloworld0d/main.py

5. Run



The application is executed.

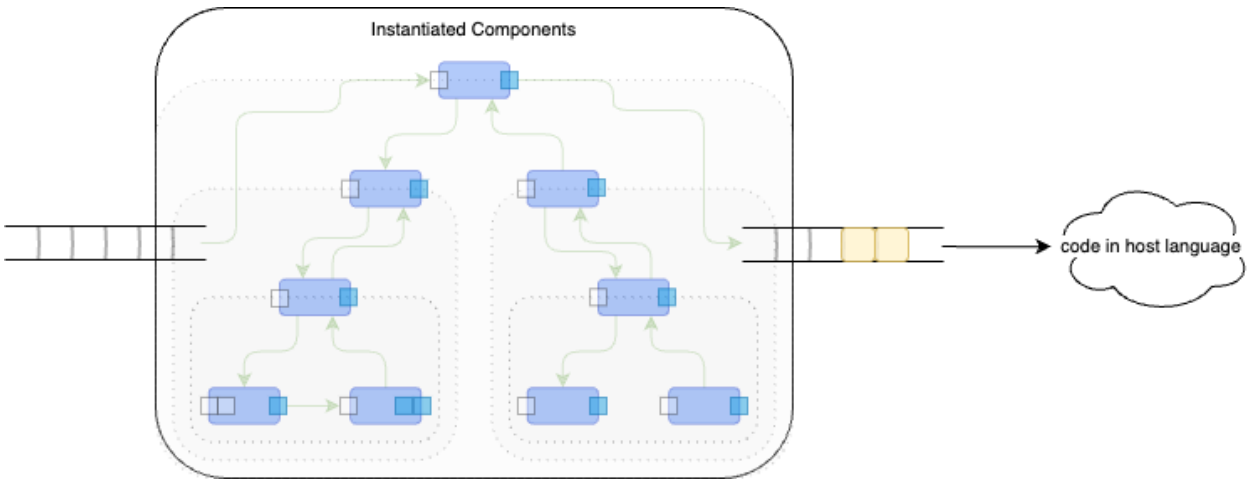
The application is interpreted by repeatedly delivering messages along connections, recursively, until no messages remain in any input queues.

It is possible to invoke already-written library code written in some other host language using the active flag - 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[ceptre].

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

6. 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.

Other Examples and Usage:

- VSH
- Arith0d,
- llm0d,
- transpiler
- kinopio2md
- Scheme to Javascript, Scheme to Python
- Prolog inferencing of diagram semantics when editor does not supply convenient information
- Generate components automatically
- delay0d
- 0D odin
- 0D cookbook

Principles

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
- No built-in paradigm. Async and ordered-in-time by default.
- 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.
- All components are asynchronous by default.
- CPUs are meant to be single-threaded
- There is no single happy path in many applications
- Subroutines are not functions
- Hidden dependencies
- Build-and-forget
- LEGO® block-like composition of software components (needs full isolation).
- Modern tooling for developers, breaking free of function-based paradigm.
- Building new paradigms using existing tools.
- T2t transpilation pipelines.

Future

- Compilation, instead of interpretation, to other languages.
- Parsing XML using OhmJS instead of calling XML parsing libraries.
- Testing, unit testing, coverage testing.

References

[1] https://app.diagrams.net/

[2] https://en.wikipedia.org/wiki/GraphML

[3] https://en.wikipedia.org/wiki/Parsing_expression_grammar

[4] ohmjs.org

[5] Odin

[6] Odin XML parser.

[7] das2json

[8] JSON

[pa] Parental Authority https://guitarvydas.github.io/2024/02/17/Parental-Authority.html

[Harel] Harel Statechart paper

[] Ceptre,