Extending Programming with Diagrammatic Programming Languages

ZAC NOWICKI, Kagi Inc., USA
PAUL TARVYDAS, Retired, Canada

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 form. TPL-only expression of programs can lead to perceived complexity and increased program sizes. The use of DPLs makes it possible to address these sorts of issues using fresh notations.

ACM Reference Format:
Zac Nowicki and Paul Tarvydas. 2024. Extending Programming with Diagrammatic Programming Languages. 1, 1 (April 2024), 11 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 - text.

Use of text for programs was based on the early limitations of electronic hardware and on the convention that human thought needs 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.

Authors' addresses: Zac Nowicki, Kagi Inc., Palo Alto, CA, USA, znowicki@kagi.com; Paul Tarvydas, Retired, Toronto, Ontario, Canada, paultarvydas@gmail.com.

Manuscript submitted to ACM

Early electronic hardware was made 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 should allow the use of alternate forms of expressing programs.

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 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.

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 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. 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.

(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 - lays out 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

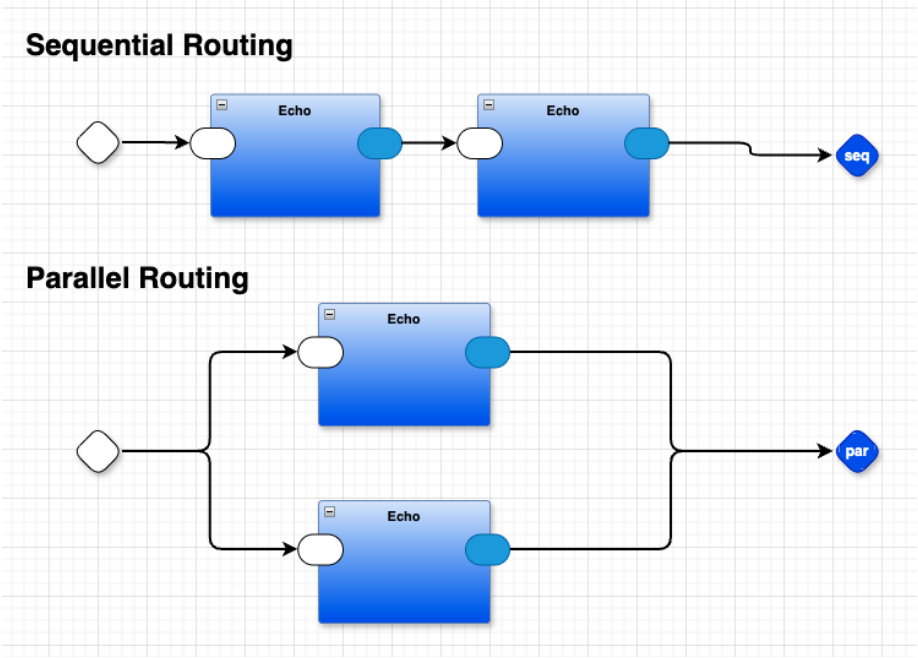

Fig. 1. Towards DPLs-1. Motivating Example

Fig. 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 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*[3].

The graphML file can be processed using off-the-shelf tools, like an XML parser or a PEG[4]-based parser[5].

In our case, we used the Odin programming language[6] 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 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] ...
]
```

The filename of the draw.io drawing

The tab-name of the top-most diagram.

A bag of Children components.

A bag of Connections between children (including connections to/from the parent containing diagram).

3 LEAF COMPONENT "ECHO"

In our implementation, we used the Odin programming language for creating the diagram parser, but, we used 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 following steps:

(1) Convert DPL program diagrams to JSON

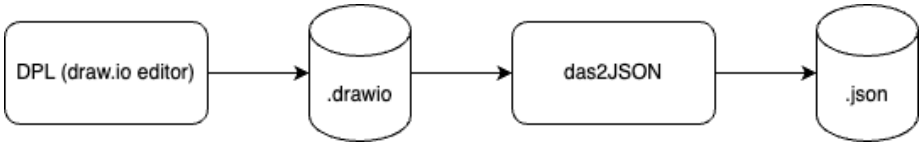

Fig. 2. Towards DPLs-1. Convert diagram to JSON

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[9] 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

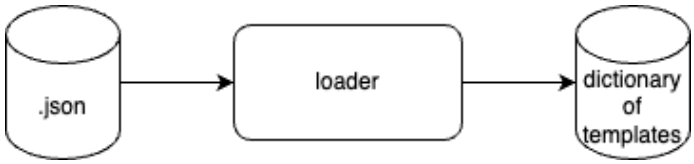

Fig. 3. Towards DPLs-2. Loader

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*).

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

(3) Instantiate system

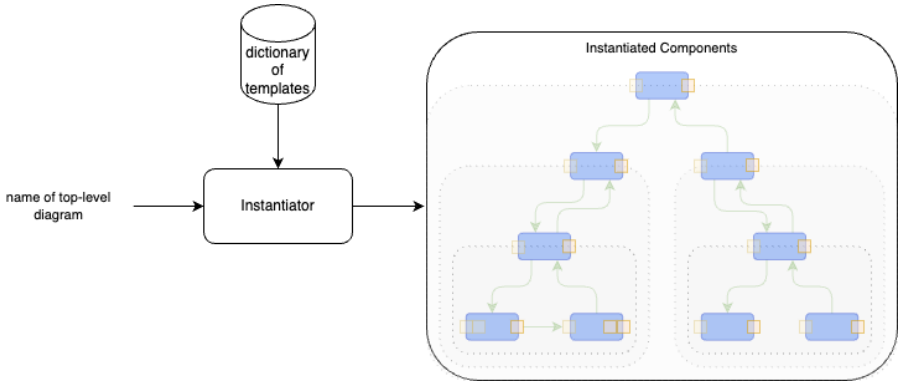

Fig. 4. Towards DPLs-3. Instantiate

Instantiation[10] 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.

```
...
    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

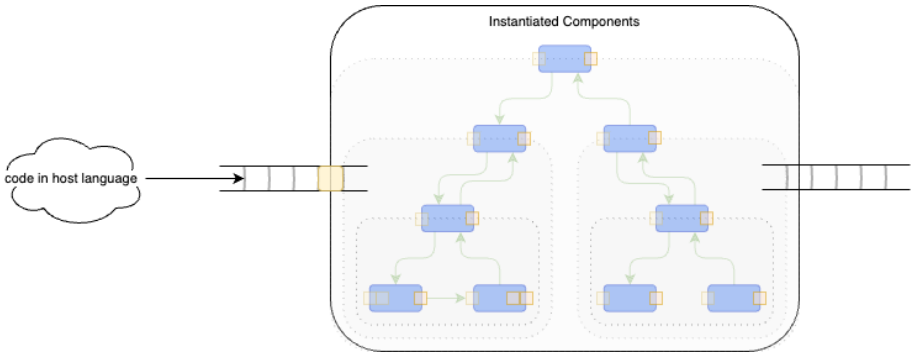

Fig. 5. Towards DPLs-4. Inject first message(s)

The first message is constructed using the host language. The message is injected[11] 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.

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.

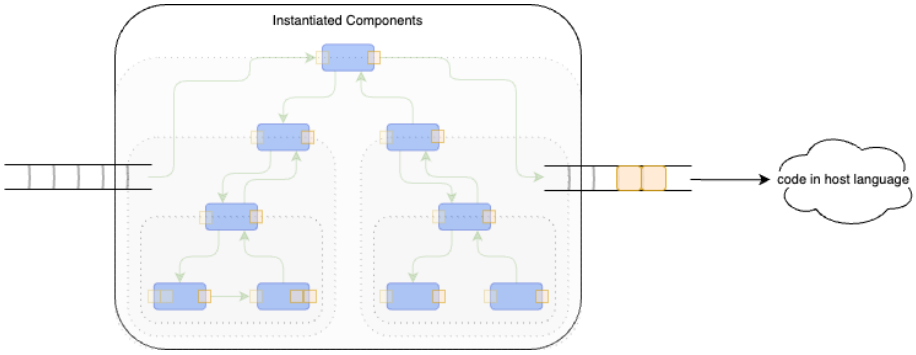

Fig. 6. Towards DPLs-5. Run

```
...
    if not load\_errors:
        injectfn (root_project, root_0D, arg, main_container)
...
```

(6) Display outputs

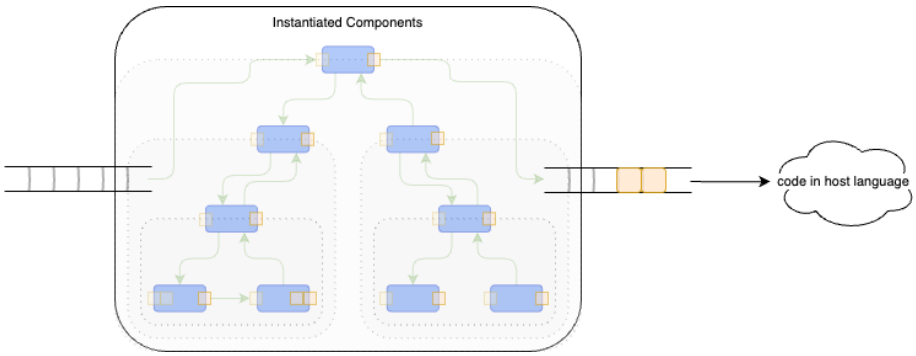

Fig. 7. Towards DPLs-6. Display Outputs

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.

5 OTHER EXAMPLES AND USAGE

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

- VSH a Visual SHell that works like /bin/sh pipelines employing more input and output ports,
- arith0d OhmJS' arithmetic grammar that concurrently emits four (4) target languages (Javascript, Python, Common Lisp, WASM),
- LLM0D 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 a workbench for creating new textual syntaxes, uses OhmJS more than once in a pipeline
- kinopio2md experiment for converting Kinopio "mindmaps" into markdown based on connections between ideas expressed in Kinopio
- Scheme to Javascript, Scheme to Python - 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
- 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 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 a DPL engine written in the Crystal programming language, which is currently being used in a production system
- 0D cookbook 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.

There is 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.

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
- 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 the function-based paradigm.
- Building new paradigms using existing tools.
- *t2t* text-to-text transpilation pipelines.

8 FUTURE

- Compilation, instead of interpretation, to other languages.
- Parsing XML using the *Transpiler* component instead of calling XML parsing libraries.
- Testing, unit testing, coverage testing.
- Depending on the graphical editor used, Prolog (among other languages) can be used to inference semantic information.

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
[4] https://en.wikipedia.org/wiki/Parsing_expression_grammar
[5] https://ohmjs.org
[6] https://odin-lang.org
[7] https://github.com/guitarvydas/onward/tree/main/helloworld0d/0D/das2json
[8] https://www.json.org/json-en.html
[9] https://github.com/guitarvydas/onward/tree/main/helloworld0d/0D/das2json
[10] https://github.com/guitarvydas/onward/blob/main/helloworld0d/0D/python/std/run.py
[11] https://github.com/guitarvydas/onward/blob/main/helloworld0d/main.py