Arrowgrams Overview

Arrowgrams is a way of constructing software using Parts[1] and Wires[2]. Parts can be Schematics[3], Code[4], or Parts[5].

The idea of Arrowgrams is to mimic Electronics Hardware design. Parts are like ICs (see <$n#figure:Integrated Circuit>). Schematics are collections of Parts (see 2. Wires look like lines on a diagram. The Arrowgrams editor is like a CAD system - it allows us to place parts on a Schematic and to wire them up.

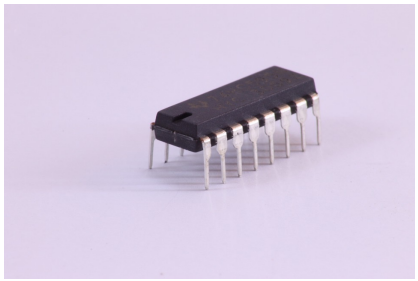

Figure 1

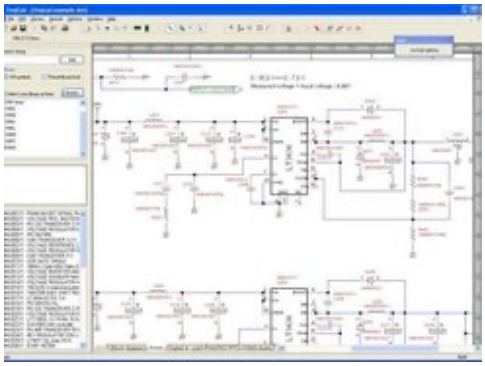

Figure 2

In Arrowgram, diagrams make intuitive sense. All pedantic issues are made

---

[1] Software components.

[2] Drawn on a diagram as a line.

[3] Hierarchical containers.

[4] In some base language, such as JavaScript. In this V2 version of the compiler, I used Common Lisp as the base language.

[5] Black boxes imported into the project from some other project.

to allow diagrams to make sense.[6]

A Part consists of 3 items:

1) A *Kind* - the *kind* determines how the part works, but each Part is a unique instance of a given kind[7].
2) A set of input pins.
3) A set of output pins.

Parts communicate with one another via *events*. Events are messages that can come in bursts, with time in between events.[8]

Parts cannot communicate directly with each other - they can only send events up to their enclosing[9] schematics. The enclosing schematics contain wiring lists that specify what each output pin is connected to. The designer can decide which output pins are wired to which input pins of parts on a schematic. This *encapsulates* the flow of events within a system. Events can come in "from the outside" if they arrive on an input pin of the schematic[10]. Events can flow out "to the outside" if they are wired/sent to the output pin(s) of a schematic.

Pedantic Issues (not required reading)

- Every Part can process exactly one input event at any one time. This includes Schematic parts, hence, a schematic is *busy* as long as *any* of it

---

[6] For example. OO uses *inheritance* as a way of constructing software. With *inheritance*, a *child* can override the operation of a *parent*. In Arrowgrams, this is not allowed - a *parent* always takes precedent, e.g. a *child* never overrides the operation of a *parent.* In this manner, *composition* always works - there are no hidden *gotchas* when parts are glued together.

[7] If you've used other programming languages, Types and Classes are much like Arrowgram Kinds.

[8] Another diagram-based technology, called Flow-Based Programming (FBP) uses *flows*, which implies continuous streams of information. Events are "shorter" than flows and are not sent continuously. Events are like pulses / edges in digital electronics.

[9] a.k.a. "Parent"

[10] And if that input pin is wired to some child's input pin.

children parts are processing input events.

- Output pins can send an event to a number of input pins on the same wire. The system *guarantees* that one event on one wire will reach *all* of its corresponding parts before any other event is delivered in the same schematic. This property is a form of *time ordering* and guarantees that there are no system-created race conditions.[11]
- Every part has a queue of input events and another queue for output events. Input events are 2-tuples consisting of {pin, data}, where the pin is a valid input pin of the given part. Output events are also 2-tuples {pin, data} where, in contrast to input events, the pin refers to a valid output pin of the part. The *dispatcher* converts output pins to input pins when delivering events[12].
- When a Part *sends an* output event, the event is *not* delivered to its destination(s) immediately, but is placed on the output queue of the part. The Dispatcher delivers the queued-up output events only after the part has completed processing of a single event.
- Parts must always run to completion. Parts must not employ infinite loops (nor infinite recursion)[13].
- Input pins of a schematic can be directly connected to the output pins of that same schematic.
- The astute reader will notice that our system resembles the ideas of mutual

---

[11] The following is a very pedantic point: race conditions can exist in a particular design, but, they will never be caused by interleaving event delivery. This is simple "physics" as intuited by most people. This pedantic issue is guaranteed only where efficiency permits, e.g. events sent on the same CPU or events sent on very fast wires. When event delivery can take a long time, e.g. events sent by CPUs that are distributed and on slow wires, this pedantic guarantee is lifted and needs to be addressed explicitly in the design. In fact, most opportunities for *race conditions* disappear with the semantics of Arrowgrams.

[12] The Dispatcher follows the wiring list in a schematic during event delivery.

[13] This is the reason that operating systems use full preemption strategies. Such strategies are expensive and are not needed by our language-level Arrowgrams system, but, the designer/programmer is entrusted to not create such infinite situations. Infinite looping is a bug in every known programming language. We tighten the definition to mean that long-lasting loops are also a bug.

multi-tasking and event loops. These ideas were tried only in operating systems before being discarded. The ideas, though, remain valid for programming-language level systems such as Arrowgrams.

The Problem

I have decided to rewrite the Arrowgrams compiler using Nils M. Holm's prolog[14].

The first version of the compiler, V1, consists of 29 passes. Each "pass" is extremely simple. Most of the passes are implemented in gprolog[15] and the passes are strung together with a *nix shell program. Prolog is used, since a lot of the work done by the compiler consists of searching a *factbase* using backtracking.

The new, V2, compiler will be built in Common Lisp and, initially will mimic the passes of the V1 compiler.

[14] http://www.t3x.org/bits/prolog6.html

[15] See the chapter on the V1 Compiler for more detail.

Problem #1

To set up a test bench where we can use the V1 compiler intermediate files to test parts of the V2 compiler as it evolves.

A Solution

A Solution to Problem #1

We know that we need a factbase.

To use the V1 intermediate files, we need a way to read Prolog facts, store them into a factbase, then write them back out as Prolog facts.

The above immediately points to 3 Parts[16], plus the Compiler (as it evolves)

1) A reader.
2) A factbase.
3) A writer.
4) The Compiler. The Compiler part is the test-bed for everything we will be building.

The following sections describe my design of these four parts. Note that there are many ways to solve this problem and that the following solution shows only one way to solve the problem.[17]

[16] A "Part" is a software component.

[17] How I choose to solve this problem.

Compiler

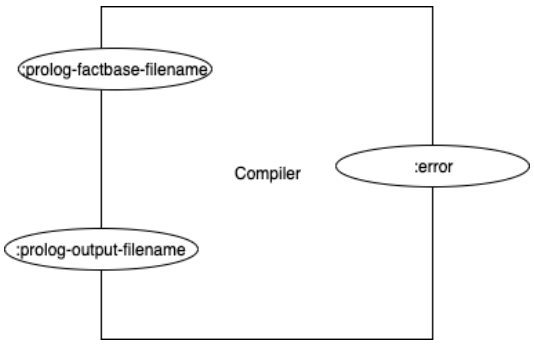

Figure 3

At the moment, the test version of the V2 compiler is a part with two inputs and one output.

The inputs are called `:prolog-factbase-filename` and `:prolog-output-filename`[18] respectively.

The output is called `:error`.

The intention of this (simple) design is that the V2 compiler test-bed accepts two filenames, as strings, on its inputs. If there is any problem, some kind of error event will come out of the `:error` pin.

The input pin `:prolog-factbase-filename` tells the test-bed to read the file with that name and convert it to some internal format.

The input pin `:prolog-output-filename` tells the test-bed to write out the factbase to the given file name, when the test has completed. I remind myself that the output format of the factbase is "prolog", since I want to tap into the V1 compiler using facts (prolog) that are compatible with the V1 internal format.

[18] See the section entitled "naming conventions" for the meaning of "-" and ":" in this context. "-" and ":" are just letters here, like a-z and A-Z.

Compiler Internals

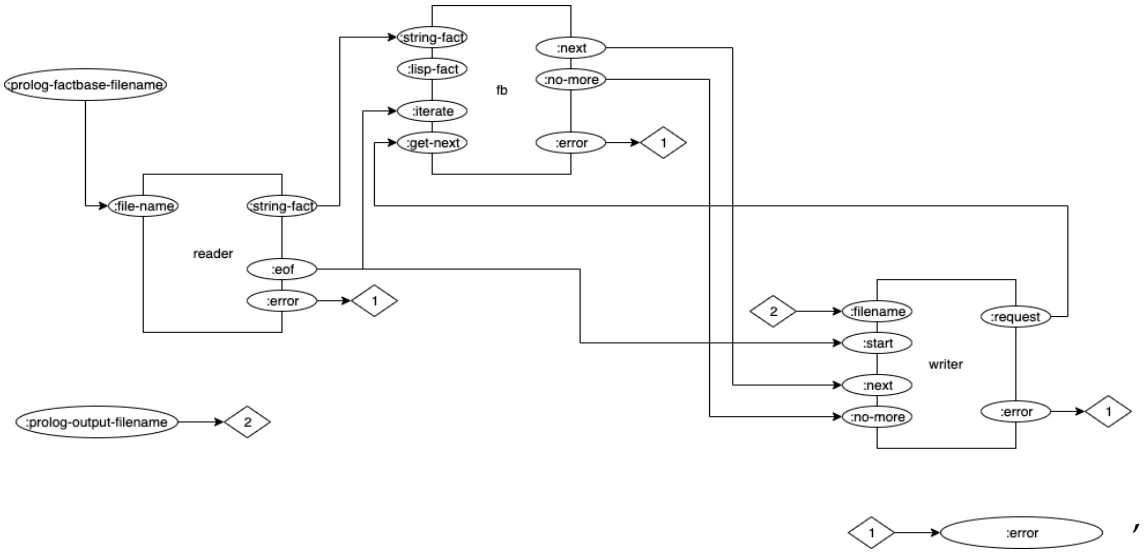

Figure 4

The V2 compiler test-bed is a schematic that holds 3 parts - one reader, one writer and one factbase part. The "Compiler" schematic is a parent that holds 3 part instances and wires them up. The designer[19] is responsible for placing the parts and wiring them up.

The wiring diagram of the V2 compiler test-bed is show in Figure 4.

Several aspects that are uncommon in state-of-the-art software can be seen in this diagram:

- A part (writer) sends and event "backwards" to another part (fb). Feedback loops are common in electronics design[20] and such feedback is allowed in Arrowgrams. The feedback in this case is quite "light". In the future, we will see cases where a part will output an event to one of its own input pins.
- One output pin is connected to two input pins. The Reader part output

[19] Me, in this case.

[20] Just about any design involving Op-Amps uses feedback loops.

pin called `:eof` is connected to the FB :iterate input pin and to the Writer :start input pin.[21]

- The Reader's `:eof` signal is just a signal, with no inherent data structure. I call it a *trigger*.[22] It is used to kick off iteration in the FB and to kick off writing in the Writer. After being started, the Writer immediately sends a `:request` *trigger* to the FB part, but this causes no problems due to the event-sending semantics discussed in the *Pedantic Issues* section of the overview chapter. In other words - the `:iterate` pin of FB and the :start pin of Writer get the *trigger* "at the same time". The Writer cannot send a `:request` that reaches the FB before the FB sees the *trigger* on its `:iterate` pin.

[21] Most software language allow for function calls that take many parameters, but very few allow multiple outputs and tend not to allow multiple outputs to be directed at multiple different routines.

[22] Again, the inspiration came from Triggers in digital circuits.

Reader

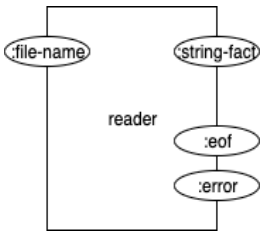

Figure 5

The reader Part has one input pin and three output pins.

The input pin is called `:file-name`. This is the string name[23] of the file that the reader will read. The reader knows that the file is formatted in Prolog format and will convert the facts to some internal format.

The reader output pin called `:string-fact` will emit a single fact as a string, as the fact is read from the file and converted.

The reader output pin called `:eof` will fire[24] when the reader has reached the end of the input file.

If any kind of error occurs, an error event will be output on the `:error` pin.

[23] Fully qualified.

[24] It sends a T value, but any value will do.

Writer

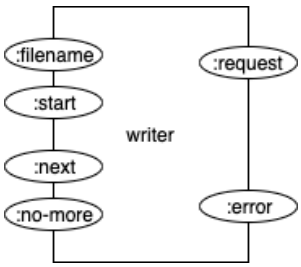

Figure 6

The Writer has 4 input pins and 2 output pins.

The input called `:filename` gets a string filename[25] that is the name of the file to be written.

The input pin called ":start" tells the writer to request facts from the factbase and to write them to the given output file in Prolog format.[26]

The input pin called `:next` gets a single fact[27] and causes the writer to convert it to Prolog format and write it out to the given output filename.

The input pin called `:no-more` is fired when there are no more facts to be had.[28]

The output pin called `:request` fires when the writer is ready to receive another fact for writing[29]. The intention is that the writer asks for each fact, one by one and receives responses on its input pin `:next`[30].

---

[25] Fully qualified

[26] Any input value will do. The fact that *anything* arrives on this input, starts the writer.

[27] In internal format

[28] Again, *any* input value will do here.

[29] Again, :request can output *any* value, but in this case outputs a T value.

[30] Or, it receives *anything* on its :no-more pin.

As usual, the output pin `:error` outputs an error event if anything goes wrong.

Factbase

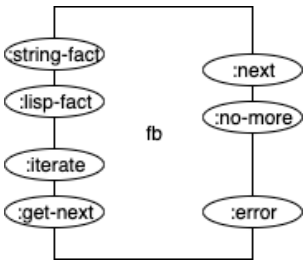

Figure 7

The factbase part holds the factbase in internal format.[31]

The FB part has 4 input pins and 3 output pins.

The input pin `:string-fact` accepts a string which contains one fact. The fact is converted to internal format and stored in the factbase.

The input pin `:lisp-fact` accepts a fact in Lisp format and stores the fact in the factbase.[32]

The input pin called `:iterate` is fired to request the FB part to set itself up for iteration through the factbase. The FB produces no output(s) when this pin is fired.

The input pin called `:get-next` is fired to request the FB to send the next fact out of its `:next` output pin. The FB, also, advances its iteration through the factbase when this pin fires.

The output pin called `:next` sends out one fact every time the `:get-next` input is fired, if there are more facts in the iteration.

---

[31] We don't need to know what the format is, as long as the FB and Writer parts agree on the format. For the record, though, the V2 test-bed is going to use Lisp lists as facts.

[32] Converting the fact to internal format, if necessary.

The output pin called `:no-more` signals that a single iteration has finished and that there are no more facts to be sent to the `:next` output pin.

If anything goes wrong, some kind of error event is sent to the output pin `:error`.

The logic of the FB component can be expressed as a state machine in Fig. 8.

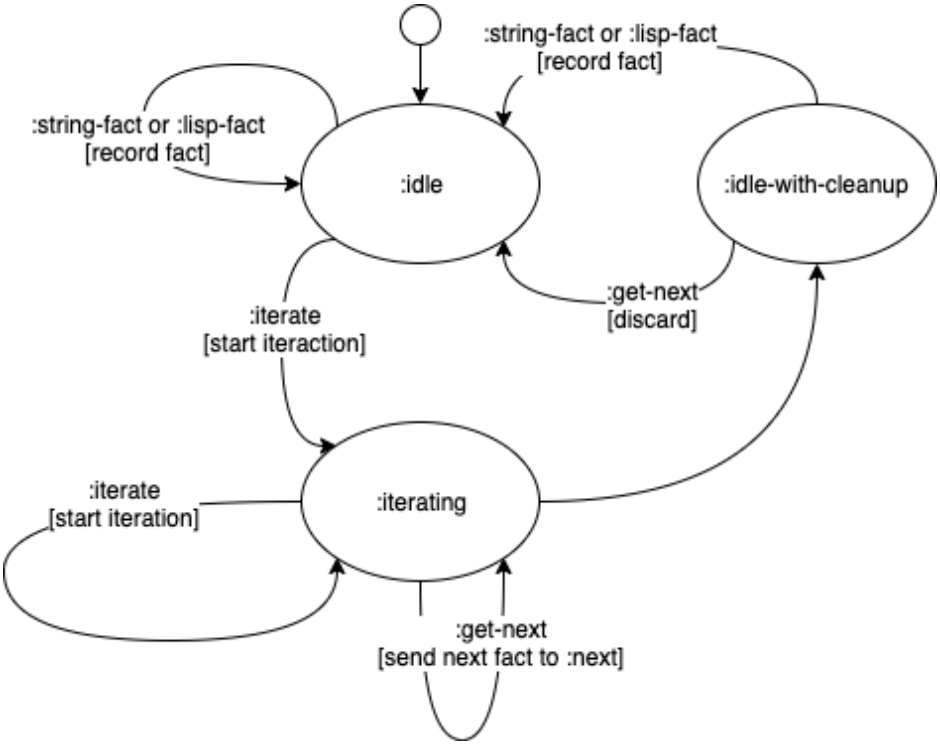

Fig. 8, FB State Machine

The FB must "clean up" its input, as exactly 0 or 1 `:get-next` *triggers* can be queued up at the point that it changes state.

Naming Conventions

I'm using Common Lisp naming conventions. A *dash* "-" is valid character in a name (it doesn't mean "subtract" in this context, as it would in other languages). And, a *colon, ":"* can be the first letter of a name (the meaning of a leading colon means something special in Common Lisp, but we don't need to discuss that here).