# Introduction

I discuss a very simple system composed of two visual components.

I call this system Arrowgrams, or AG for short.

The components communicate through messages via a third component - their common parent.

As motivation, I provide the pseudo-html, and, I provide a diagram of the system.

The two main components are shepherded by a top-level component.

The two components can only communicate by using messages.[1]

The messages *cannot* be sent directly from one component to another. All message routing is performed by the parent, schematic[2] component. In this very simple example, that composite component is the top-level component.

The message delivery system is implemented by a *kernel* of code, in this case written as a JavaScript script. I show the details of a very straight-forward implementation of such a *kernel.* I consider the implementation details to be a distraction and, hence, I leave them to the end of this document[3].

I happen to use HTML + JavaScript. This system can be implemented in any language. In fact, I have implemented it in Common Lisp and various pre-cursors in C and assembler.

---

[1] E.G. The components cannot share memory.

[2] composite

[3] and github

# github

The code for this system can be found at https://github.com/guitarvydas/ag-js.

See, also, any follow-on or preceding versions of this system (I will try to call them ag-js-*).

# Pseudo-HTML

The system in this simple example consists of an HTML *input file* component and an HTML *paragraph* component that displays the contents of the file.

Obviously, such a system can be directly implemented - and easily understood - as HTML + JavaScript. The intention here, though, is to show the fundamentals of creating large systems using pluggable components.

In essence, what we want is:

```
<input type="file" id="part1" onchange="sendMessageToPart2(…)>…</input>
<p id="part2"></p>
```

Further, we want a top-level component that "owns" these two components.

```
<div id="topLevel">
    <input type="file" id="part1" onchange="sendMessageToPart2(…)>…</input>
    <p id="part2"></p>
</div>
```

We want the top-level component to route messages between its children.

The children cannot send messages directly to one another.

Routing of messages is done only by parent components.

The routing is represented as an Nx2 table in the parent part (in this case "topLevel").

I use the name "wire" to refer to each row in the table.

Each wire contains one sender and 0-or-more receivers.

Every reference to a part consists of two parts - the id of the part and a string[4] which I call a *pin*[5].

The *pin* is merely a selector which can be used to subdivide messages.

So, the above is embellished as:

```
<div id="topLevel">
  <input type="file" id="part1" onchange="sendMessageToPart2(…)>…</input>
  <p id="part2"></p>
  <wire>
    <sender ref="[part1,'fileObject']" />
    <receiver ref="[part2,'displayText']" />
  </wire>
</div>
```

Note that a wire can have zero (0) receivers. This is named NC[6] (for No Connection). An event that arrives at an NC is simply ignored (dropped)[7].

Note, also, that the wiring table (routing table) is stored in the *parent schematic*[8] (div) and not in the children. The children must not communicate directly with

---

[4] Detail: for this simple example, we will use single quotes for pins to avoid the need for JavaScript string escapes. We will use double-quotes only for calling JavaScript functions.

[5] A *pin* can be equated with the concept of a *topic* in pubsub.

[6] FYI: Most of this terminology comes from the field of digital electronics.

[7] Exception: when an event arrives at an NC of a top-level part, it is added to the results of the system, or, printed using console.log().

[8] I used the word "schematic" to mean a component that contains other components, i.e. a composite component.

one another.[9] This is an important design principle for maintaining architectural flexibility, e.g. an architecture can be *reused* by simply changing the wiring table.

The pseudo-code above shows "sendMessageToPart2" which directly names "part2". This, in my opinion, is bad practice[10]. The above example is only meant as a stepping-stone between current practices and better componentization. A further rewrite that avoids this problem is:

```
<div id="topLevel">
  <input type="file" id="part1" onchange="sendChanged()">…</input>
  <p id="part2" onreact="display()"></p>
  <wire>
    <sender ref="[part1,'changed']" />
    <receiver ref="[part2,'displayText']" />
  </wire>
</div>
```

where "sendChanged()" is a JavaScript routine, associated with the *input file* component, that sends the File object[11] to its own "changed" *pin*. Note that, in this case, the *input file* component doesn't know, nor need to know, where its output message will be routed, it simply *send()s* a message to one of its own *pins*. The data (the File object) will arrive at part2 with the *pin* tag 'displayText'. This is specified by the *wire* in the routing table of "topLevel". All messages to "part2" will invoke the JavaScript routine "display(…)" using events of the form {pin, data}, which in this case will be {'displayText', *fileObject*}.

My working example1.html has the form:

```
<div id="topLevel">
    <script>
        ... details ...
    </script>


    <input type="file" id="part1" onchange="sendChanged()"></input>
    <script>
```

[9] Most current PLs (Programming Languages) use *call* to name the receiver directly. This is merely an optimization and calcifies architecture.

[10] "do as I say, not as I do" :-)

[11] the result from the input operation

```
        ...  details ...
    </script>


    <p id="part2"></p>
    <script>
        ... details ...
    </script>

    <script>
       // initialization of topLevel part
       ... details ...
       /* map each output pin {part,pin} to an array of
          receivers [{part,pin},...] */
          topLevel.initializeWires ( [ ... details ... ] );
       </script>

       <script>
         var kernel = new Kernel (topLevel);
       </script>
    </script>
</div>
```

*Detail: [skip on first reading] - the* `onchange` *event is defined by HTML.* `Onreact`*, though, is a custom event. In fact, I defined my own events - AGevents - in raw JavaScript, instead of relying on CustomEvent()s. In this set of examples, I choose to use JavScript's* `{…}` *operator to define events on the spot, instead of defining a "class" for Events. This is a design choice and nothing more. The reader will note that* `Onreact` *does not even appear in my code.*

# Diagram

A first-cut diagram is shown in Fig. 1

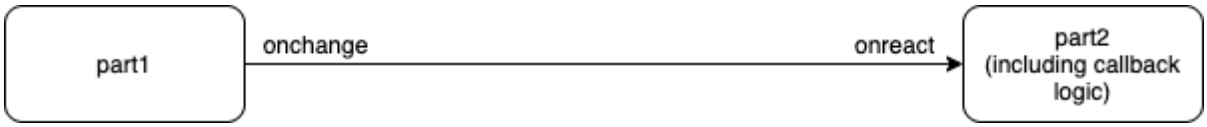


Fig. 1 First Version of Diagram

This diagram has the "problem" that all callback logic is implicit, probably buried in "part2".

We can pull the callback logic out into a separate component, as in Fig. 2:

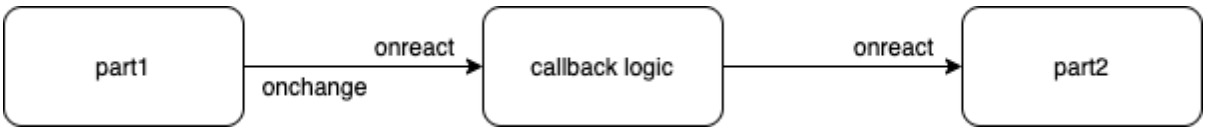


Fig. 2 Second Version of Diagram

The second diagram shows `onreact`[12] as the input *pin* of, both, the "callback logic" and the "part2" components. This is as intended, but is very JavaScript-y. In fact, every component reacts to the custom `onreact()` function in the same way - by calling its own `onreact()` function with two arguments - the *pin* and the *data*.

Let's refine the diagram to use better *pin* names in Fig. 3

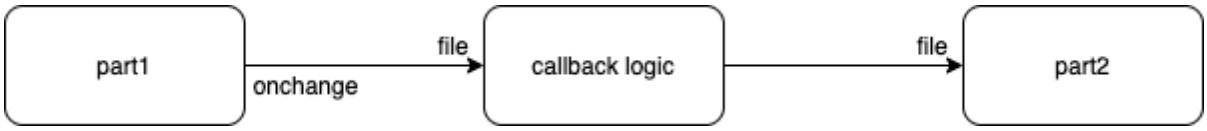


Fig. 3 Third Version of Diagram - Better Pin Names

This diagram still has a problem - it does not describe the entire situation. The callback logic has at least 3[13] possible outcomes -

1. success
2. error
3. no response

and, only (1) outputs a good File descriptor. Let's redraw the diagram with this detail, in Fig. 4

---

[12] I continue to talk about Onreact, since it might be more familiar to the reader, as a CustomEvent(). In reality, I implemented custom events directly in raw JavaScript.

[13] In fact, there are more than 3, but let's keep things simple (to read and understand).

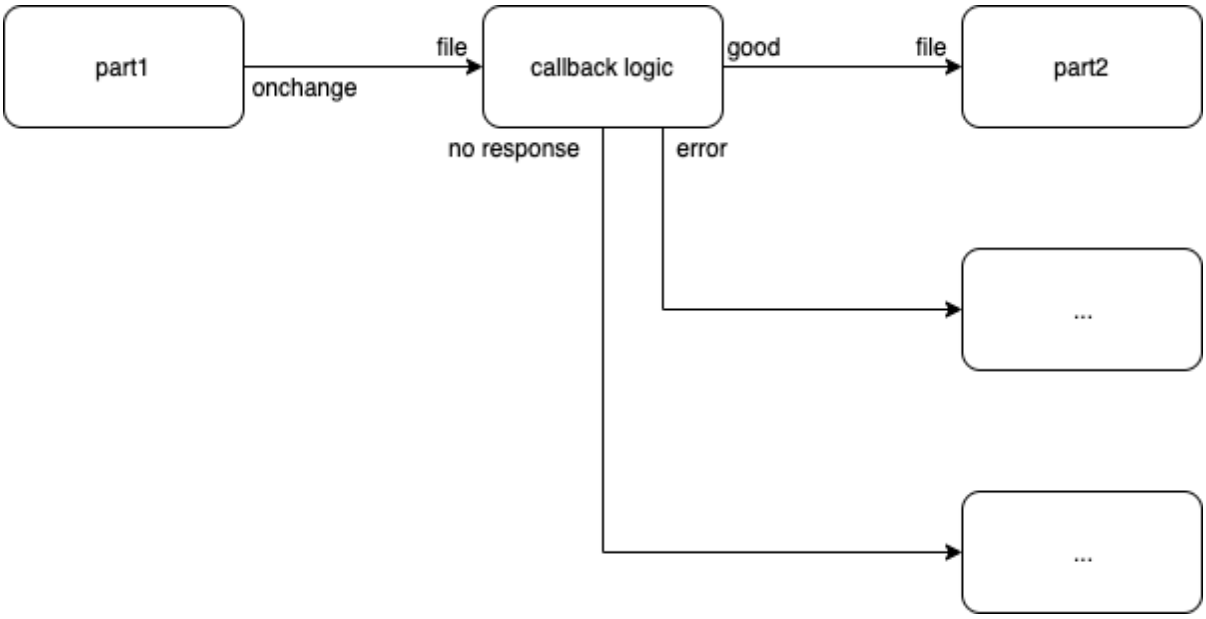


Fig. 4 Diagram That Shows 3 Callback Outcomes

but, how does "no response" get generated? Answer: a timer component, as in Fig. 5:

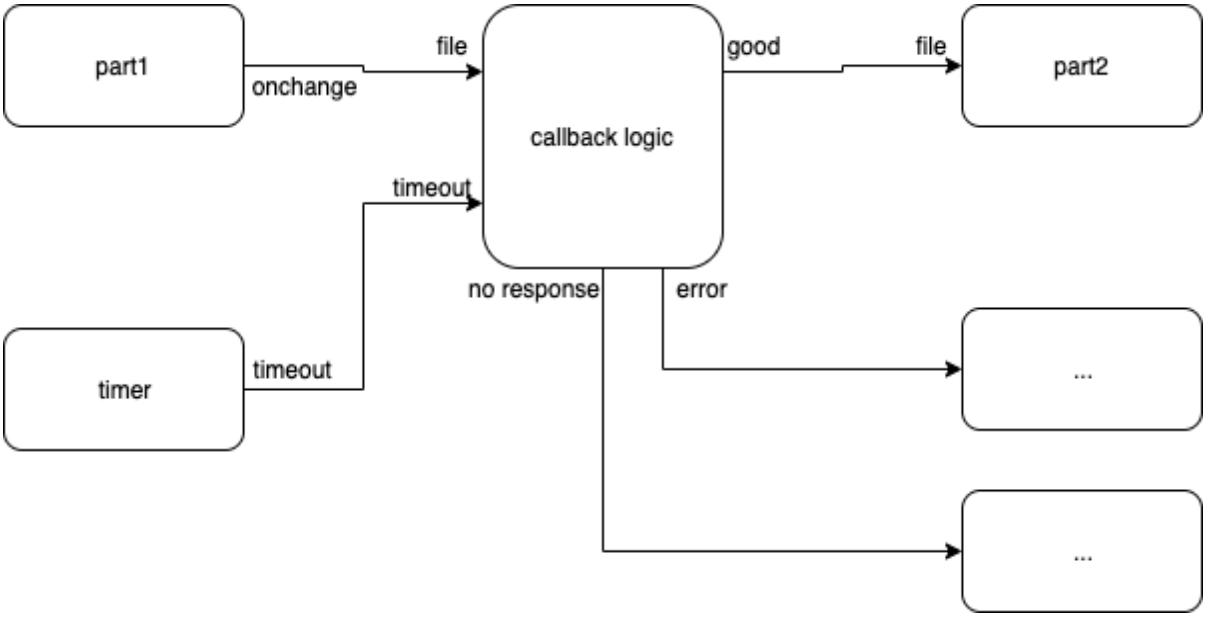


Fig. 5 Diagram That Shows a Timer Input

If this last diagram looks complicated, remember that this is what needs to be

done in JavaScript code, without a diagram, using only text (which, I claim, is even more complicated to understand).

If we wanted to simplify the diagram, we could show only the *happy path* as in Fig. 3, and lasso the rest of the components - the *unhappy path* - into various other hierarchical, composite, components.

Better yet, we could invent a programmers' diagram editor that showed layers - different views - of the diagram (e.g. the *happy path* layer, the *error* layer, the *timeout* layer, etc., etc.).

## Happy Path

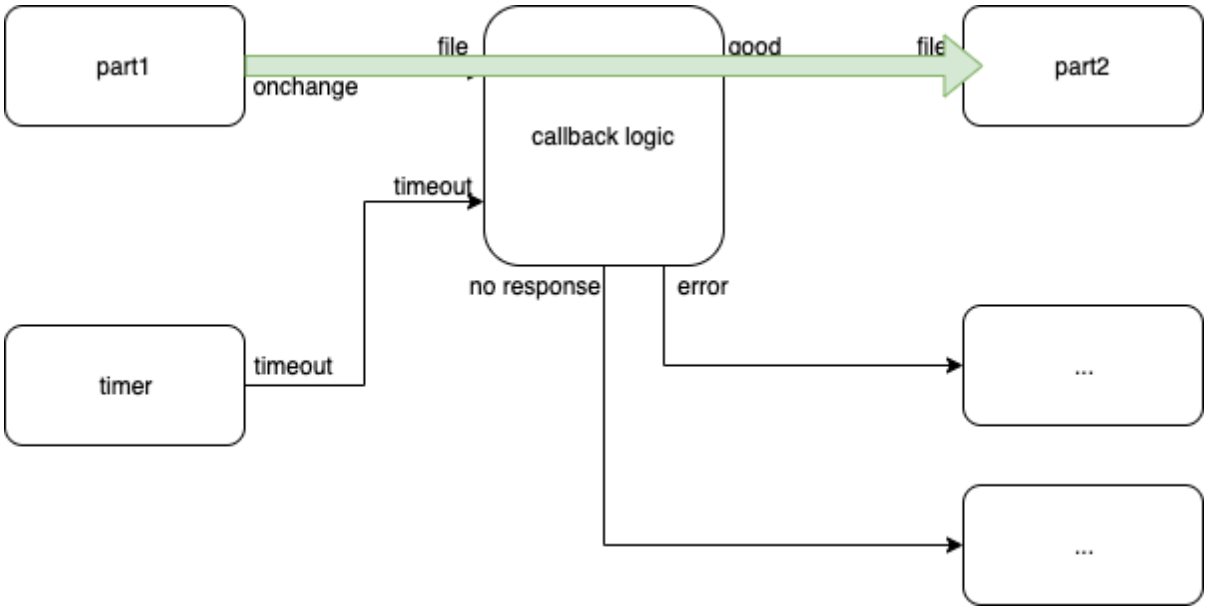


Fig. 6 Happy Path

The green arrow shows what happens when everything goes according to plan.

# Error Path

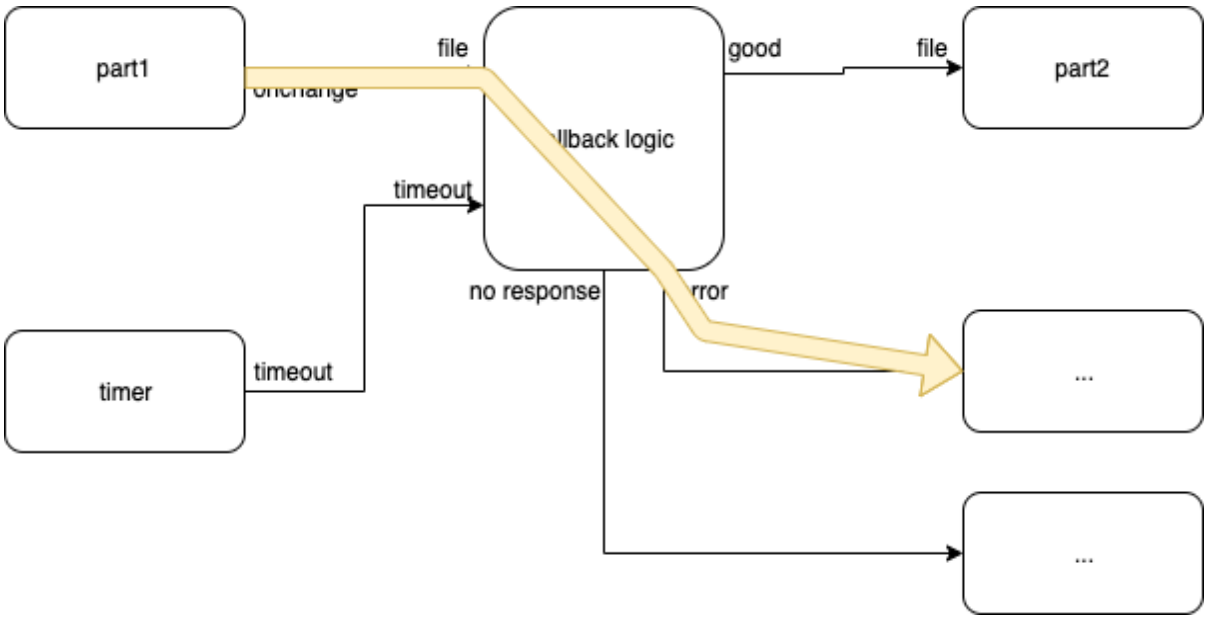


Fig. 7 Error Path

The yellow arrow shows what happens when the system (e.g. HTML) finds something wrong.

# Timeout Path

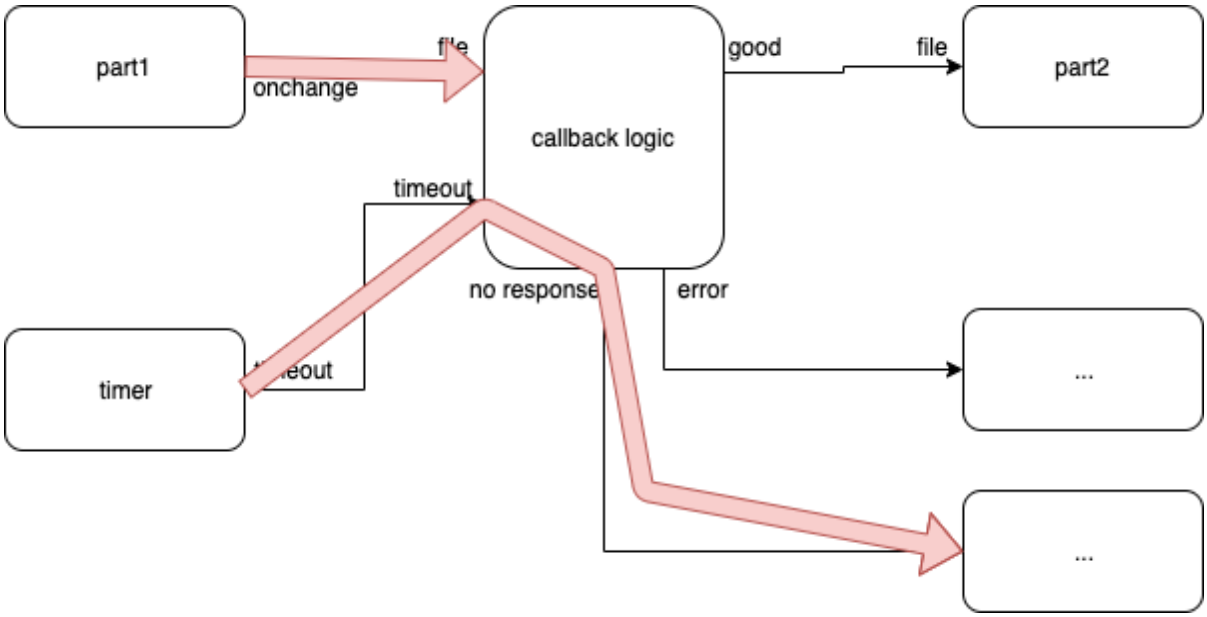

Fig. 8 Timeout Path

The red arrows show the control flow when things go horribly wrong, e.g. the FileReader can't read the file due to a server being down (or a break in the routing to the server, or, …).

# Diagram Layers

From the previous description, it becomes obvious that

1. We want to describe the system using diagrams, instead of using code - the diagrams could be automatically compiled to code.
2. We want to draw diagrams in layers, e.g. a layer for the *happy path,* a layer for the *error path* and a layer for the *no response* path.

I will show one way to do each of the above - how to compile diagrams and how draw and combine layers.

I will show this later,[14] after having described the code thus far.

For motivation, note that:

- I treat JavaScript as an assembly language and will be compiling diagrams into this kind of assembly language. This has similarities to the *no code* movement. I argue that compiling diagrams compiled to assembly language (e.g. JavaScript) is more flexible than most *no code* approaches.
- New-breed assembly languages don't complicate the transpiler (compiler) with details such as declaration-before-use and strong typing.[15] New-

[14] Maybe in other essays.

[15] Strong typing is for programmers, not transpilers. Strong typing for programmers will be provided by SCLs (Solution-Specific-Languages, a subset of DSLs). See my essay "New-breed HLLs".

breed assembly languages provide easy access to 1st-class functions and anonymous functions. See my essay "New-Breed HLLs" for further discussion.

- Note that the preceding diagrams used only 3 kinds of *glyphs*: boxes, text, and, lines (arrows). Unlike in programming editors, the glyphs are not arranged in a fixed 2D grid, and, unlike in state-of-the-art programming editors, the glyphs can overlap. If we consider the *glyphs* to be *tokens*, then there is almost nothing new here - we already know how to compile strings of tokens to code, e.g. most existing textual 3GL[16] compilers do this. The main aspects of these new-breed tokens is that they carry an (x,y) point with them and that they are hierarchical[17].
- Diff, git, github, etc., already know how to compare two code files and to merge them. Layers might be thought of as a git-style merge. In fact, we can stoop even lower by simply comparing items for equality - e.g. two layers are mergeable if the items[18] that they contain are exactly equal, and, missing items are simply unioned into the merged result. Readers who have studied EE (Electrical Engineering) will see similarities of this mentality to Kirchoff's Laws of Superposition.

---

[16] like JavaScript, Java, Python, C, etc.

[17] e.g. text glyphs, which contain all of the properties that we already associate with text.

[18] Compilers consists of various phases. The early phases (e.g. scanning and parsing) break a program down into separate items, e.g. types, functions, etc. In HTML, we would consider HTML elements to be items for comparison, for example  elements, <input …> elements, etc. HTML files (layers) could be merged if one file contained elements that did not appear at at in the second file. If both files contain, say, an <input …> element with the same id, then those elements would be considered to be equal only if (a) each attribute in each element was exactly the same, or, (b) if any specific attribute exists in only one file, but not the other. For example, '<input id="a" type="file" onchange="b">' could be merged with another attribute '<input id="a" type="file" accept="c">'.

# Details

# 50,000 Foot View

The file "example1.html" contains:

- script includes for the kernel
- script includes for 3 kinds of parts - constructor functions for schematics, file selectors and text output boxes
- a <body> that defines a top-level  called *topLevel*, and 2 components: (1) *part1* (a file selector input widget) and (2) *part2* (a text file output box (an HTML <p>).

This corresponds to Fig. 9.

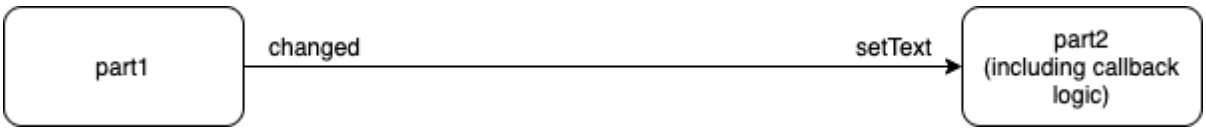


Fig. 9 Example1.html as a Diagram

Fig. 9 is almost the same as Fig. 1. The pin names have been changed.

# File Selector (part1)

Line 20 of example1.html defines a standard HTML file input element.

Lines 21-29 define the programmer-supplied JavaScript for this file selector element, namely the function *sendChanged()*.

When the user pushes the button and selects a file, the *sendChanged()* function is called.

This function grabs the appropriate data from the part1 HTML element and *send()*s it out the output pin of part1. The output pin is called 'changed'.

After the *send()*, we call *kernel.io()* to run the kernel dispatcher. This is a detail that will, later, be performed automatically by the transpiler. Only in this early example1.html, do we show the gory details of what is going on under the hood.

In fact, *kernel.io()* is an optimization.[19] The kernel's dispatcher could *loop* forever, running parts that are `ready` and obviate the need for calling *kernel.io()*. This[20] is a technique I learned by building production kernels on bare metal[21] and is suitable for asynchronous operation, like node.js calls.

# Text Output Box (part2)

The text output box is defined in lines 32-70 of example1.html as a `<p>` element plus some (a lot of) JavaScript supporting functionality.

The *happy path* is the call to `reader.readAsText(…)` followed by a successul call to the callback function for `onload()`. After the reader loads the file, the `onload()` function is called as a callback to display the text contents of the file[22].

Two (2) other outcomes might occur and are handled by the `onerror()` and `onabort()` callbacks. Later, I will show how I create these callbacks in a more "structured" manner[23]. For now, the gory details are on display. Note that the code for a simple text display component is "not so simple" when written out in JavaScript (or any textual) form.

In all, the code for our text component must handle 1 input and 3 outcomes

---

[19] Is this a premature optimization?
[20] using *kernel.io()*
[21] no operating system
[22] Note: this code works only for .txt files.
[23] For motvation, consider Drakon and StateCharts.

(good, error, abort). Current programming languages provide a way to express 1 input and 1 outcome easily. Syntax for handling 1 input and 2 outcomes exists (*throw*). There is no syntax for 1 input and 3 outcomes, hence, the JavaScript code is written at a very low level - an "assembly" level, in my opinion - and programmers can try to deal with such situations in a myriad of ways, e.g. by conflating the two *unhappy paths* into one by returning a non-scalar object[24]. The code, as written in this example1.html, is not production ready - it punts on the issue of errors and aborts.

Diagram notations, e.g. StateCharts and the notations I describe, can be used to express functionality with multiple outcomes and asynchronous behaviours.

# Part topLevel

The `topLevel` component is defined in lines 15-17 and is initialized in lines 72-77 of example1.html.

Further discussion regarding schematics, parts, events, etc., follow.

# Wires

A *wire* is a data structure that contains one *sender* and a list of *receivers.*

The *sender* is defined by a { part, pin } pair.

Each *receiver* is defined by a { part, pin } pair.

[24] e.g. {…, …}

# Kernel

The kernel provides 2 main functions:

- Send() - called by parts to deliver events to their outputs
- Dispatch() - the only "loop" in the system - it invokes parts that are `ready()`.

This version of the kernel, based on HTML and HTML events, also provides the `kernel.io()`[25] function. This allows asynchronous events to be converted from HTML (and JavaScript) into events supported by AG. In the least-optimized case, the `Dispatcher()` would be a loop that continually checks the `ready()`ness of parts and dispatches them. `Kernel.io()` allows us to spread this "loop" over every HTML event receiver and prevents wasting cycles until actual (asynchronous) IO occurs.

We don't need `kernel.io()` in dedicated systems, e.g. bare metal where the `Dispatcher()` can run an idle loop, without optimizing use of cycles. This kind of system might be an IoT[26] device, in which the code runs a simple (idle) loop checking for input and (maybe) running a task in the background.

# Key Points - Semantics

1. Children cannot (must not) send messages directly to one another, but must route messages through the parents' routing table.
2. A wire must lock all input queues of its receivers and deliver events to all receivers *atomically*.

(1) is a key aspect of treating components as pluggable LEGO® blocks.

[25] Influenced by Ric Holt, et al.
[26] Internet of Things

(2) ensures "sanity" of multi-tasking. Events must not supercede one another. Events cannot be tangled[27]. In hardware, this detail is addressed by using maskable interrupts (IRQs). Software can never run faster than the underlying hardware. Software techniques to regularize this problem of atomicity, can be invented (and have been), but, come with a performance penalty. The time saved in software development is paid back with slower hardware performance. Engineers (not programmers) should consider and make trade-offs depending on specific problems and specific solutions. They should find a local minimum for a specific solution, e.g. it might be acceptable to use slow, regularized, generalized locking in some cases, for example websites that must only respond in "human time", but such speed penalties might not be acceptable in missile systems and financial trading systems. The issue of creating a working design should be left to Architects, while the optimization of speed should be left to Production Engineers[28].

Dispatching should be performed in a breadth-first manner. Delivering events in a depth-first manner might result in the equivalent of the *priority inversion* problem encountered in traditional operating systems.

Fairness doesn't actually matter in this kind of system[29] , but, the actual dispatching algorithm can be tuned for specific problems and solutions.

Long-running operations, such as loop or a deep recursion, are considered to be "bugs"[30] Repetition can be had by sending events to oneself, e.g. "continue".

Feedback is allowed - parts can send events to themselves (directly or indirectly). Feedback is not quite the same as recursion, due to the fact that

---

[27] or, all bets are off

[28] see my essay about "Roles for Software Development"

[29] Parts finish a processing step in near-0 time. Long-running parts are "bugs".

[30] LOOP (and recursion) is the exception not the rule.

events are buffered by the input queues. Recursion tends to imply that a routine will run again immediately. Feedback events do not imply such a behaviour - only the `dispatcher()` determines when a part gets to run[31].

Events can be delivered to "nothing". NC - no connection. Such events are ignored and dropped. This, also, is a key aspect of LEGO® style component composition. It is not possible to know beforehand whether all functionality provided by a part will be used. Some parts might generate output events that are not needed for a certain application. It must be possible to ignore some events, i.e. to connect some pins as NC.

## Main Loop

It is possible to think of parts as each having a `mainloop`, analogous to windowed apps.

## Mutual Multitasking

This system employs mutual multitasking between components.

Mutual Multitasking was discredited as a way of building operating systems in early versions of Windows®.

Mutual Multitasking is, indeed, not a suitable paradigm for building time-sharing operating systems, where the operating system must guarantee isolation between components, but, mutual multitasking can be used to compose a single App - application.

[31] corollary: a part cannot name other parts, not even itself, in event send()s

In the case of an App, a "bug" is just a bug, and it is not reasonable[32] to expect a system to protect parts of the App from bugs in other parts of the App. Mutual Multitasking is suitable for this kind of design.

Mutual multitasking is inherently more efficient than time-sharing multitasking which uses full preemption[33] In fact, time-sharing brings a lot of baggage and bugs that are not needed by most Apps[34].

# Part

There are two kinds of parts:

- `schematics` - composite parts that contain other schematics and/or leaf parts
- `leaves` - parts that contain functionality, usually as code, and cannot be broken down further by the component system.

Every part, schematic and leaf, contains the following items:

- An `attribute` that tells the system whether the part is a schematic or a leaf
- An `input queue`
- A function that determines if the ready queue is not empty
- A function to determine whether the part is `busy`
- A function to determine when the part is `ready` to be dispatched (see discussion below)
- A function that is called, by the dispatcher, when the part is activated and sent an event. I call this function `react()`.

---

[32] Maybe this statement is not forward-looking enough? What if every routine/function could be protected by hardware MMUs? What if all pointer accesses were hardware-protected?

[33] Although full preemption is used in almost every threading library and O/S, full preemption is actually quite expensive. The literature calls the worst-case scenario "thrashing".

[34] This baggage is needed only by time-sharing applications, such as operating systems.

# Event

An event is of the form:

```
{ pin: "…", data: "…" }
```

i.e. an event - in the component system that I describe - is defined as consisting of two fields.

The first field is called `pin`. A `pin` is essentially a tag that identifies the kind of event. Receiving parts can case/switch on the `pin`.

The second field is called `data`. The data field can contain any arbitrary object, from a scalar number and a scalar string to a more complicated object, like `{ … }`.

The sender and receiver must "agree" on what the shape of the data is.

# Event Type Safety

The sender and receiver must "agree" on what the shape of the data is.

Checking type safety is punted to enclosing layers.

An analogy to this layered approach is: the layers in a network protocol. Each receiving layer strips off a part of the message[35] and checks it. Each sending layer wraps the data in micro-information that allows the receiving end to check the data, in layers.

Note that, if one takes composition of components to heart, then type

[35] like peeling an onion

checking becomes simply a layer (or filter) that is snapped into place in front of pins.

# Schematic

A *schematic* is a part that contains other parts (leaf and/or schematic parts).

A schematic contains two more pieces of information than the generalized *part* ancestor class:

- A list of part instances - its children.
- A list of wires. This is basically a routing table of messages between children. See below.

Children cannot (must not) send messages directly to one another, but must route messages through their parent's routing table.

This indirection is a fundamental principle of a component-based system that makes architectures flexible. One cannot treat components as composable LEGO® blocks if components can identify one another directly. Direct calls are *verboten* in the system that I describe[36]

The same `react()` function can be shared by all schematics. The job of a schematic's react() function is to pull an event from its own input queue and deliver it to all of its (direct) children that are waiting for the event[37].

For an example of schematic parts, see `schematic.js` and how it is used in `example1.html`.

[36] Call/Return can be used *within* a component, but cannot be used to communicate with other components.

[37] e.g. connected to the same wire

## Leaf Part

A leaf part is one that has a `react()` function that is specific to the part.

A leaf part contains no children[38].

Leaf parts can be implemented using code in some language, e.g. JavaScript.

For examples of leaf parts, see `fileselector.js` and `text.js`.

## Busy

A `part` is `busy` if it is doing work.

In a system that sits on top of an operating system, and, in which, parts cannot be interrupted, all leaf parts never appear to be busy and schematics are busy only if any of their children are busy.

Note that most simple applications of AG sit on top of an operating system, and, all parts run within the same user thread, i.e. all components run in the same app. This kind of system can ignore busy-ness of leaves and use a simple recursive algorithm to calculate schematic busy-ness.

In a system that allows preemption - e.g. a system using bare metal with IRQ[39]s, or a system using time-shared threads with parts spread across threads, busy-ness of leaves becomes an issue that needs to be dealt with. This would involve semaphores and locks and the standard bag of technologies used in present-day multi-tasking threads. The issues that need to be dealt with are: atomic enqueueing of events onto input queues and locking all input queues of all parts on a single wire, during event delivery. The main semantic that must be

[38] at least, no children known to the AG system
[39] hardware interrupts

guaranteed is that a single event arrives at *all* receivers in an atomic fashion. Variations on handling this problem might include STM technologies (which, by definition, have worse timing than finely tuned systems of locks and semaphores[40]).

The problem of keeping locks for as short a time as possible cannot be solved in general, and, depends on specific problems and their solutions. In hardware, and sufficiently modular technology, it is possible to calculate the *worst-case* lock time - if a system cannot lock a wire and finish depositing an event to all locked receivers within a given time, we just throw our hands up and use faster, more expensive, hardware.

Software designers should strive towards calculating throughput and worst-case throughput.

Note that classic operating systems, such as MacOS®, Windows®, Linux and most "real-time" RTOSs have throughputs that are so bad that no one bothers to mention the results.

The throughput calculation for classic operating systems is made harder by the use of full preemption - one must calculate the effect of an timer interrupt happening between any atomic machine instruction and the next, and, the cost of calling the system to perform context switching[41].

# Ready

A part is `ready()` only if:

---

[40] STM is basically a 2-phase Hoare Monitor which keeps locks longer than might be necessary, due to generality.

[41] including fiddling with the hardware MMU, etc.

1. it has at least one event in its input queue
2. it is not `busy()`.

Any part that is `ready()` can be `dispatch()`ed.

`Ready()` parts cannot (must not) rely on *when* they might be `dispatch()`ed. Ordering of part `dispatch()` is indeterminate.

Parts are concurrent[42].

[42] See my essays on concurrency vs. parallelism.