The Life Cycle of an Event

The topic here is the behavior of event-driven GUI applications built on the *flow object* model[1]. This paper describes the behavior of these applications in response to user-interface events.

What's important to keep in mind is that an event-driven GUI application is either sitting still doing nothing (that is, every event-aware user-interface projector component is listening for an event) or else the application is busy responding to one user-interface event. The application alternates between these quiescent/busy states, spending almost all of its time in the quiescent state.

This paper describes the entire behavior of these applications in terms of the response to one event by one projector component. It combines material from two preceding papers.[2,3] I am treating it as a topic unto itself for these reasons:

1. I find it interesting that the flow object model partitions the application so that its whole activity can be described simply in terms of the event response of a single user-interface component.
2. This paper presents a subsystem communication model that appears to be distinct from Model-View-Controller and could therefore be of general interest.
3. The algorithms described here look more like the solution of a physical constraint network[4] in the presence of a disturbance than object-oriented software.
4. This paper describes the overall event-processing design of every event-aware user-interface projector component in terms of a simple state machine.

---

1 http://melconway.com/Home/pdf/pattern.pdf

2 ibid, Update Protocol, p. 13.

3 http://melconway.com/Working/WP_9.pdf component computation scheduling algorithm, p. 13-14.

4 I am indebted to the work of Alan Borning on ThingLab. See https://en.wikipedia.org/wiki/ThingLab

© 2018 Melvin E. Conway
Twitter: @conways_law

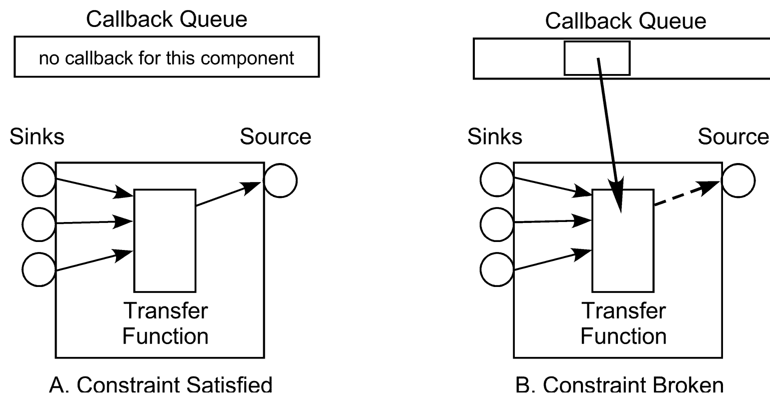
conway.mel@gmail.com

The Constraint-resolution View of the Network

The figure below shows that every component can be in one of two conditions with respect to whether its output is consistent with its inputs. If the output is consistent with its inputs the component's constraint is said to be *satisfied*; otherwise it is said to be *broken*.

The environment of the executing application contains the network of connected components; it also contains a callback queue. In this queue is one callback for each component with a broken constraint, and no callback for any component with a satisfied constraint.[5]

When all constraints are satisfied, that is, when the output of every component is consistent with its transfer function applied to its inputs, the application is quiescent and the queue is empty. (Part A on the left shows a component with a satisfied constraint and no entry in the queue.) Every component with a broken constraint is matched by one callback in the queue that can cause execution of the component's transfer function. (Part B on the right shows the broken constraint as a dotted arrow and the callback for this component pointing to the transfer function.)



Receipt of an event from the User Interface Management System (UIMS) will introduce at least one broken constraint into the quiescent network. This can happen in the projector component that receives the event, or it can happen as the result of executing the Update Protocol, which is triggered by the component receiving the event (see the next section).

Upon termination of the Update Protocol the environment will work through the callback queue(s)

---

[5] There are actually three queues; this is an optimization to minimize redundant display refreshes. See http://melconway.com/Working/WP_9.pdf page 14.

from the front, applying each callback (that is, computing its component's transfer functions), and then removing the callback from the queue.

After computing its transfer function each component will typically have a new output value. The consequences of this value must immediately be reflected downstream.[6] (I know of two cases that require a push at this point: adding a wire in the tool and changing the selection in a Selector component.[7] Otherwise, the network of dependent endpoints is already in place and all that is required is the **`notify dependents`** part of the Update Protocol.)

When a downstream component's sink connector receives notice that its value has changed, the component's constraint is then by definition broken, so a new callback might have to be added to the queue.

- If the component's constraint was previously satisfied[8] its callback is added to the rear of the queue. Its transfer function is not computed at this time.
- If the component already has a broken constraint, it already has a callback in the queue and nothing is added to the queue. (In one approach the callback is moved to the rear of the queue.)

[6] The exception is if there has been no change to the output; whether there is actually a test for that is a component-specific optimization.

[7] Whether any other cases exist is an open question.

[8] This can be determined either by searching the queue or by keeping a satisfied / broken flag in each component.

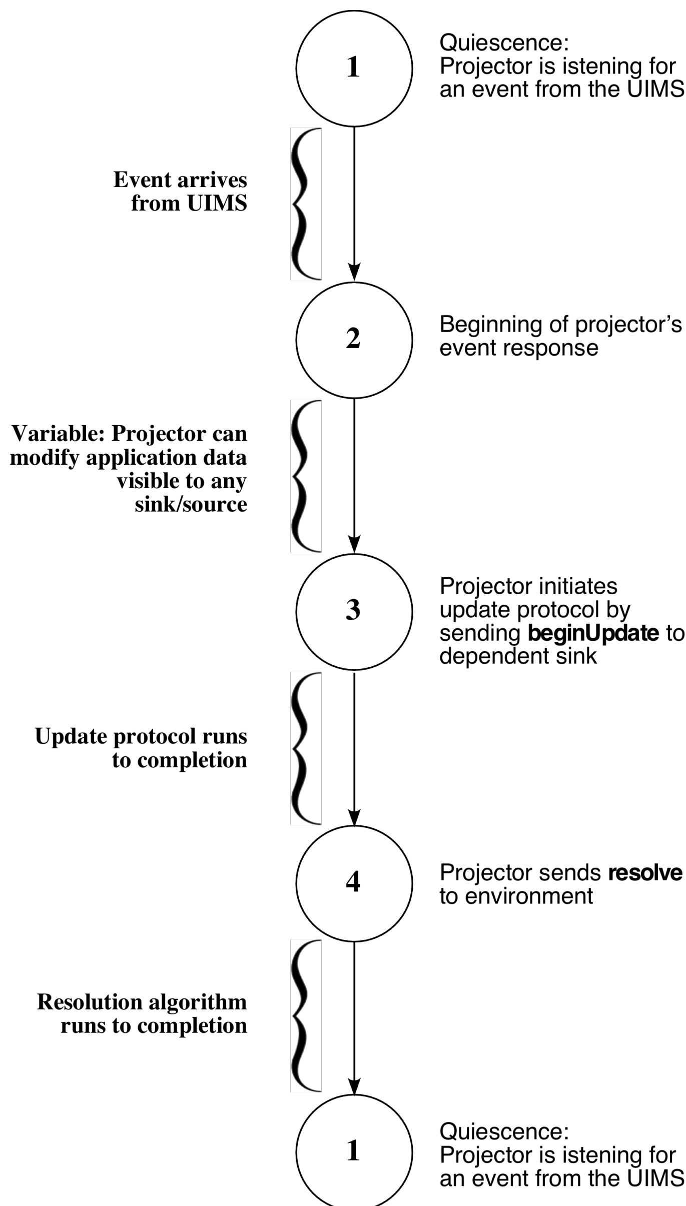
© 2018 Melvin E. Conway

Twitter: @conways_law
conway.mel@gmail.com

The State-machine Projector Model

The life cycle of an event begins and ends with the user-interface projector component that receives the event.

The accompanying figure describes the behavior of every event-aware user-interface projector component as a state machine. When it is not busy (this is **State 1**) the projector is listening for a particular event from the UIMS. Note that **Transition 2 → 3** is the only place where the component's activity is specific to that component's functional definition.



State 1. The projector is quietly listening for its particular event from the UIMS.

Transition 1 → 2. The event arrives, moving the projector to state 2.

State 2. The beginning of event processing.

Transition 2 → 3. What happens here is specific to the particular projector component. At most, the component will modify data visible to it via one of its connectors. For example, a click in a list box changes its selection index. This causes the projector to obtain the index from the UIMS and assign a corresponding value to the `selection` instance variable of the `SelectedOrderedCollection` object at the projector's sink. Or, the `enter` key in an editable text line sends an event signaling the completion of editing; the projector obtains the new string value from the UIMS and assigns it to the string object at its sink. Note that at the end of this step the user-interface appearance is consistent with the input to the projector.

State 3. Standardized behavior begins. The projector inititates the Update Protocol by sending the `beginUpdate` message to the sink connector carrying the object being projected (call it the *seed* object).

Transition 3 → 4. This is the Update Protocol; see reference [9]. From the constraint-network perspective, the equilibrium has been disturbed because a constraint is now broken: the component that owns the seed object has an output that doesn't conform to applying its transfer function to its input because its output was directly modified in transition **2 → 3**. Transition **4 → 1** will restore the equilibrium.

State 4. The projector initiates the Resolution Protocol by sending the **`resolve`** message to the environment.

Transition 4 → 1. This is the Resolution Protocol; see reference[10]. Each component in the network with a broken constraint has a callback in a queue whose execution will compute the correct outputs and push them down any connected wires. This might break the constraints of one or more other components, causing one or more callbacks to be queued. Eventually[11] the process will terminate with all queues empty, and the program returns to a quiescent state.

---

[9] http://melconway.com/Home/pdf/pattern.pdf page 13.

[10] http://melconway.com/Working/WP_9.pdf page 13.

[11] Absent an illegal cycle in the wiring, which must be checked, probably at the time each wire is added.