# Asynchronous Active Values for Client-Side Interactive Service Coordination

Alan Dix

Talis,
43 Temple Row
Birmingham, B2 5LS, UK

School of Computer Science
University of Birmingham
Birmingham, UK

alan@hcibook.com

http://www.hcibook.com/alan/papers/avi2012-active-values/

## ABSTRACT

This paper describes Asynchronous Active Values (AAV), a framework for the production of reactive web interfaces that use API-based web service back-ends. Such interfaces are now becoming common due to API-oriented application development and more sophisticated post-Web2.0 mashups. A significant feature of such interfaces is the need for feedback when parts of the page display are in some way temporarily invalid, or in flux, while potentially slow API calls are responding to requests. AAV extends existing methods such as access-oriented programming and the observer pattern, by including a 'changing' event in addition to the normal 'onChange' to enable intermediate feedback.

## Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation (e.g., HCI)**]: User Interfaces – *graphical user interfaces, interaction styles;* H.5.4 [**Information Systems**]: Hypertext/Hypermedia – *user issues*.

## General Terms

Design, Human Factors.

## Keywords

AJAX, user-interface architecture, asynchronous update, web development

## 1. INTRODUCTION

A new breed of web-based interfaces are being developed using API-based web services. These incorporate rapid feedback for user interactions using JavaScript and DHTML, but with some operations giving rise to asynchronous API calls to web services. When these are sufficiently fast (less than a second), the calls may be perceived as sufficiently 'instant', however if there is any delay, whether due to network delays, or complex back-end processing, then some sort of intermediate feedback is usually required. For example Fig 1 shows the 'Sending..." indicator used in Gmail after the user has pressed the email 'Send' button, and Fig 2 shows the animated 'working' icon used by Facebook whilst loading new messages.

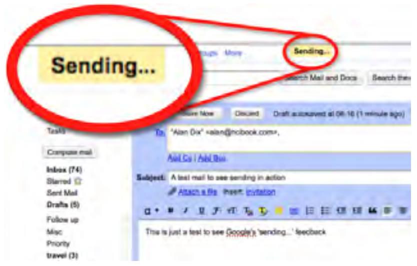


**Fig 1 in-flux indicators – "Sending..." in Gmail**

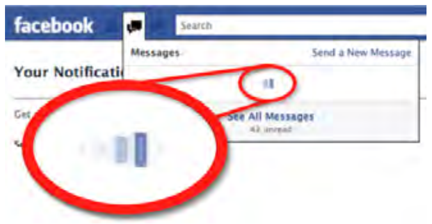


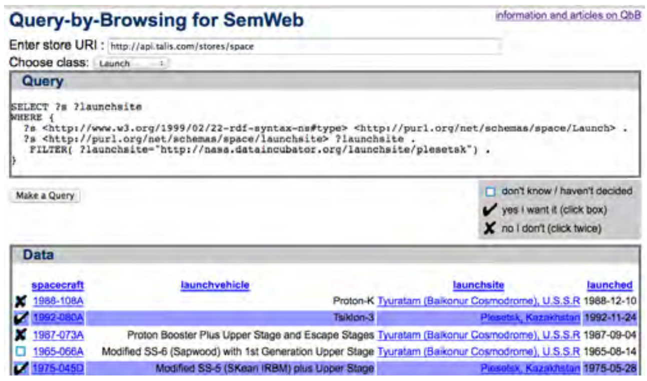
**Fig 2 in-flux indicators – messages loading in Facebook**

As Rosenberg notes in his analysis of the use of AJAX in Yahoo! mail [27], these progress indicators were a "nonissue in the "old Web"", as standard browser reload indicators were sufficient (the blank page!), but become essential when AJAX may update pages while the user continues with other activity. Other authors also emphasise the need for such 'changing' indicators, for example Tonkin [30] says "Without explicit visual clues to the contrary, users are unlikely to realise that the content of a page is being modified dynamically".

In both the examples above, there is a straightforward interaction sequence:

1. user clicks button or link
2. remote AJAX-based API call is initiated
3. display is updated to show 'waiting' value
4. API call returns
5. display is updated with results

However, this flow becomes more complex when a series of different API calls are required to build the interface. For example Fig. 3 shows the Query-by-Browsing for SemWeb interface. This is a variant of the Query-by-Browsing intelligent database interface [6] but modified for RDF/SPARQL data [25]. At the bottom is a listing of entities of a particular class together with their properties as a tabular listing (area labelled 'Data'). The user is able to select which entities are required by adding a tick (✓, wanted) or cross (✗, not wanted) against each. When ready the user clicks the 'Make a Query' button, at which point the application uses a machine-learning algorithm (a variant of ID3 [26]) to generate a SPARQL query that agrees with the users choices. This query is then shown in the 'Query' area and the entities selected by the query are highlighted in the listing. If the query is not as required the user can mark more entities with ticks or crosses and repeat the procedure.


Query-by-Browsing for SemWeb

information and articles on QbB

Enter store URI : http://api.talis.com/stores/space

Choose class: Launch

Query

```
SELECT ?s ?launchsite
WHERE {
  ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://purl.org/net/schemas/space/Launch> .
  ?s <http://purl.org/net/schemas/space/launchsite> ?launchsite .
   FILTER( ?launchsite="http://nasa.dataincubator.org/launchsite/plesetsk") .
}
```

Make a Query

☐ don't know / haven't decided
✓ yes I want it (click box)
✗ no I don't (click twice)

Data

| | spacecraft | launchvehicle | launchsite | launched |
|---|---|---|---|---|
| ✗ | 1988-108A | Proton-K | Tyuratam (Baikonur Cosmodrome), U.S.S.R | 1988-12-10 |
| ✓ | 1992-080A | Tsiklon-3 | Plesetsk, Kazakhstan | 1992-11-24 |
| ✗ | 1987-073A | Proton Booster Plus Upper Stage and Escape Stages | Tyuratam (Baikonur Cosmodrome), U.S.S.R | 1987-09-04 |
| ☐ | 1965-066A | Modified SS-6 (Sapwood) with 1st Generation Upper Stage | Tyuratam (Baikonur Cosmodrome), U.S.S.R | 1965-08-14 |
| ✓ | 1975-045D | Modified SS-5 (SKean IRBM) plus Upper Stage | Plesetsk, Kazakhstan | 1975-05-28 |

**Fig 3. Query-by-Browsing for SemWeb**

In this interface, updating the display actually requires three different coordinated API calls:

1. first the chosen entities are sent to the machine learning API, which returns a decision tree to distinguish wanted from unwanted entities
2. this decision tree is then passed to a second API to transform it into valid SPARQL
3. in parallel the decision tree and entities are passed to a third API, which filters the entities on the tree (equivalent to performing the SPARQL query).

Step 1 in particular can take some while, so it is important that the user can see that the query is in the midst of being updated. Furthermore steps 2 or 3 could return in any order.

If the application were more complex, for example drawing information from multiple Linking Open Data (LOD) sources [1], or from multiple APIs, then this picture can become more complex still.

It is possible to create such interfaces using a traditional architecture such as model–view–controller, or bespoke solutions, but the coding is relatively complex, and hence liable to be error prone and hard to maintain.

Asynchronous active values (AAV) were developed to deal with this, offering a coding paradigm tuned to deal with multi-step asynchronous actions. AAV is a framework offering concrete object classes and methods to create semi-declarative interface descriptions, but is still 'plain JavaScript', so allows the developer to modify detailed interactions if required. It therefore sits between raw code and more all embracing declarative notations such as Arrowlets [23] or Flapjax [20]. While AAVs could be used server-side, they are designed primarily for client-side interactions, unlike Go, Google's concurrent programming language targeted principally at backend processing [15].

AAV builds on long-standing use of active values dating back to access oriented programming in LOOPS [29], but in addition to standard '`onChange`' events, AAVs also have '`onInvalid`', and '`onChanging`' callbacks, which can be used for setting temporary display state, or other 'in progress' work. AAVs also provide ways for their value to be set synchronously (normal variable setting), asynchronously using RESTful JSON APIs, or asynchronously using bespoke code.

The AAV framework emerged from practical need, but, reflecting on its development, there were three principle design goals:

- *simplicity and parsimony* – aiming to hit 90% of problems with 10% of complexity, rather than be all embracing and top heavy.
- *flexibility* – does not assume the coder will work entirely within the framework/toolkit, but will use it as appropriate and alongside other techniques.
- *theoretically well founded* – not simply hacking or adding features, thus hoping to avoid 'gotchas' when unexpected future cases arise.

The remainder of this paper is in four parts. Section 2 gives some of the theoretical background, in terms of both interaction and architectural issues for networked interfaces. Section 3 gives an overview of the use of AAVs in real code. Section 4 explores the framework in more detail, looking particularly at race conditions due to asynchronous behaviour. Finally, section 5 discusses issues arising from practical use and potential future directions.

## 2. THEORETICAL BACKGROUND

### 2.1 The Evolution of Interactive Feedback

User interfaces have gone through various levels of development: the command-line in the 1970s, the GUI and direct manipulation in the 1980s, transactional web interfaces in the late 1990s, and now highly interactive client-side web applications. One of the distinguishing features of direct manipulation identified by Shneiderman was its rapid incremental feedback, and this has been a goal of interaction design ever since [28]. During the early years of web interfaces 'rapid' was not easily managed, however a more systolic interaction style became the norm with periods of rapid interaction with a web form, effectively setting up parameters for longer waits during back-end transactions; effectively a hybrid of GUI and command-response style interaction.

Web2.0 interfaces changed this picture using a combination of JavaScript, DHTML and AJAX to allow client-side web interactions that were similar to previous desktop applications. To some extent this makes interaction design in 2012 more similar to that of the early GUIs in 1983 rather than early web interfaces of 2000. In some ways Google Docs behaves just like a desktop word processor, with similar interaction and architectural properties to a desktop word processor except saving to cloud rather than local disk.

However, there are crucial differences as many local-device operations operate within known time bounds, whereas network-based interaction introduces potentially unbounded delays requiring subtle changes in interactive style [10]. These issues do not arise solely in networked systems, as even local operations, such as scanning a large disk, can take a long time, and it has been suggested, many years ago, that special rules are needed to deal with such delays, notably a form of *mediated interaction* where instant syntactic feedback is offered when semantic feedback is likely to be delayed [4]. This led to a number of delay-related design heuristics [5]:

- *"good enough now, perfect when there's time"*
  That is giving some sort of instant partial/approximate feedback,

but updating it when a longer processing or network activity is complete.

- "*don't stop the interface just because the system is busy*"
  That is, were possible, allow the user to continue to interact with the system even when there is some sort of long-term or slow computation/communication occurring.
- "*don't stop the system just because the user is busy*"
  That is, allow the system to continue with processing even when the user is on the midst of some sort of interaction.

All of these are about some level of asynchrony between user and system input rather than the immediate and completely user-controlled interactions normally assumed in direct manipulation. These delay principles were framed largely in the light of desktop interfaces where they were important, but occasional or rare, issues. However, they have now become the norm in API-based web interfaces.

## 2.2 User Interface Architecture

The dominant architectural style for interactive systems is usually some variant of the Model–View–Controller paradigm [18], originally developed for Smalltalk-based GUIs in the early 1980s and influential since, albeit with some shifts in details for transaction-based web systems. MVC is found in frameworks for desktop systems, for example Java Swing and also for the web, for example, Google's new Dart programming language is planned to include an MVC framework [3].

In MVC, user interactions are translated by the Control component directly into operations on the underlying system Model. Changes to the Model are then sensed by the View component (often using the observer pattern [14]), which re-renders the interface to reflect the current Model. Underlying MVC is the paradigm whereby the current display always reflects, as near instantly as possible, the underlying system state following Shneiderman's "continuous representation of the objects and actions of interest" for direct manipulation [28].

The very idea of the *current* system state, embodied in the MVC Model is always slightly problematic because interface state is something like a multi-headed hydra. At the base there is some core functional model corresponding to deep application state, but this is supplemented with various interaction state, some semi-permanent (such as the current font, or selection in a word processor), some more temporary (such as current interaction with a dialogue box). When we have asynchronous interaction, this effectively adds to the application state as, whether or not it is explicitly coded in normal variables of AAVs, the fact that an item is being updated is a real and important part of the underlying (distributed) system state. MVC or any other UI architecture needs to make all of this state available, in some way or other, to the user (see Fig 4).

As noted, MVC is usually implemented via some variant of the observer pattern. Effectively ordinary active values can be seen as a particular form of encapsulation of this pattern, and have been used as a primary interaction technique in a variety of user interface toolkits whether or not they are based on MVC.

The earliest use of some form of active value in the literature, of which we are aware, is the 'Access Oriented' programming paradigm in LOOPS [29]. In this it was possible to attach callbacks to any variable both just before it has been set and just after. The former allowed manipulation of the value being set, but

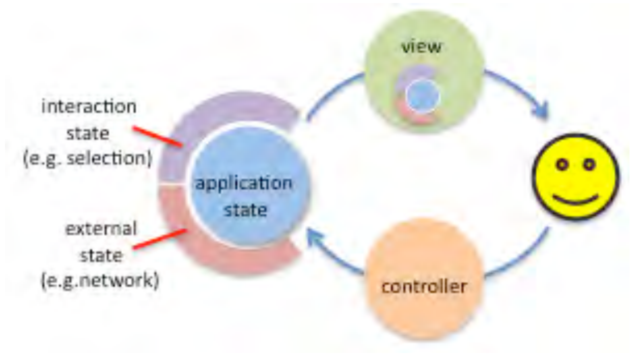

view
interaction state (e.g. selection)
application state
external state (e.g.network)
controller

**Fig 4. Interaction and external network state in MVC** (© A. Dix)

we have chosen not to implement this in the AAV framework, as it is not commonly used in user interface code.

There has also been a long history of data-flow-oriented interface toolkits, often termed 'one way constraints', notably the Garnet family [2]. More recently this has included web-based coding, for example, Arrowlets [23] or Flapjax [20]. Typically constraint-based UI toolkits use some form of special semi-declarative language, which allow the toolkit to calculate optimal update sequences. While this has advantages, it both adds another language to learn, and also limits the programmer to what is achievable within the special language. Instead we have taken the approach of providing active values as a lower level abstraction over which constraint propagation can be easily implemented.

## 2.3 Collaborative and Distributed Systems

Interfaces for collaborative systems have long had to deal with issues of asynchrony (e.g. see [16] for an early review). Solutions have either involved preventing problematic updates through various forms of locking (e.g. implicit locks in ShrEdit [19]), or allowing opportunistic concurrency for that is later solved through synchronisation (e.g. [12]) or operational transform algorithms (e.g. [13]).

These early groupware systems and algorithms, and more recent work building on them (e.g. [17] looking at groupware performance on modern browser technology) are specifically about groupware interfaces, and more application specific than AAV intends to be. However, they each, in various ways, deal with the core issues of any distributed system: liveness (allowing people to interact with minimal or no blocking) and consistency (dealing gracefully with race conditions). Solutions to these will inevitable have a level of application specific semantics. For example, when you post a status to Twitter through the web interface it temporarily 'disappears' and then later appears in your own stream – this used to lead to multiple posts when people thought their status had been lost, but Twitter now throws away repeated identical status updates. In contrast, the direct message stream for a particular user instantly adds the message to the conversation transcript (local feedback), but may later reorder the transcript if the other user has sent a message simultaneously but slightly ahead.

The AAV framework does not attempt to prescribe particular solutions to these global update issues (although the author is not without opinions!), but instead limits its scope to in-browser semantics and offers mechanisms to support the developer in managing local race conditions (see section 4.4)

## 2.4 Status–Event Analysis

AAV draw on the author's previous experience of software architectures inspired by Status–Event analysis [7], notably the aQtiveSpace framework presented in AVI 2000 [9]. Status–Event analysis recognises that many aspects of a user interface have a 'status' form; that is, they continually have a value, even if that value only changes during events. Furthermore, status–status mappings are common, for example, that the displayed value reflects an internal value or the Model–View relationship in MVC. The aQtiveSpace framework was designed precisely to deal with both status and event interface phenomena, and had a flexible model of components called 'Qbits' where data direction and control flow were orthogonal. The `ActiveVarProvider` interface (see section 4.4) is inspired by Qbits, which also managed a level of asynchrony. However, Qbits did not distinguish the intermediate 'changing' states effectively assuming that changes would happen 'fast enough'.

## 3. HOW IT WORKS – FOR THE CODER

It is possible to create an asynchronous interface using a standard model–view–controller paradigm, by adding additional state variables. For example, if there is a model variable for the current SPARQL query (call it `query`), then we would add a new variable `query_status` that can take values of 'loading' or 'loaded', and then the view component code to maintain the display of the query would look something like Fig. 5, where the view code is triggered when either of the model variables `query` or `query_status` is changed (code to add callbacks not shown).

```
if ( query_status == 'loading' ) {
  $('#query_content').html( "Loading ..." );
} else {
  $('#query_content').html( query );
}
```

**Fig. 5 code to update display based on an extra state variable**

Asynchronous Active Variables effectively wrap this into a single abstraction, so that each active variable can be set or read, but also has an intermediate 'changing' state during which reads still deliver the old value, but users can know that it is in some way incomplete, or in flux. The equivalent code to Fig. 5 is in Fig. 6.

```
if ( query_var.changing ) {
  $('#query_content').html( "Loading ..." );
} else {
  $('#query_content').html( query_var.get()
);
}
```

**Fig. 6 update display using active variable**

Although there is just one variable '`query_var`', this is not so different from the two-variable solution. However, in addition specific callbacks can be added for an '`onChanging`' event, leading to more event-driven code as shown in Fig. 7.

```
query_var = new ActiveVar();
query_var.onChanging.attach( function() {
    $('#query_content').html("loading ...");
  } );
query_var.onChange.attach( function(query) {
    $('#query_content').html(query);
  } );
```

**Fig. 7 AAV with callbacks for changing and changed states**

Note, this code is slightly longer as it includes the setting up of the callbacks, which would also be necessary for the code in Figs 4 and 5, and also the declaration of the active variable. That is, Fig. 6 is the complete code required.

If the variable '`query_var`', were set synchronously (using the method '`query_var.set(value)`',), then only the '`onChange`' event would be triggered, the callback invoked and the display set to the new query. However, it is the asynchronous use that is more interesting.

The programmer can take complete control of the active variable by calling the '`startChange`' method when initiating a long-running activity such as an AJAX call (or complex local calculation) and then updated using '`set`' when the activity is complete (see section 4 for more details). However, there are a number of convenience mechanisms to allow easier and more robust code.

The simplest level, and probably most common for standard web front-ends, is asynchronously setting the variable by an AJAX call, which is packaged in a single method '`setJSON( service_url, operation_name, args )`' (see example in Fig. 8). This method sets the '`changing`' flag, initiates the AJAX call, and also sets a handler so that when the AJAX calls returns successfully the variable is set with the returned value. If there is an AJAX call to set the variable already outstanding, this is aborted, so that only the latest value is set (see section 4.4).

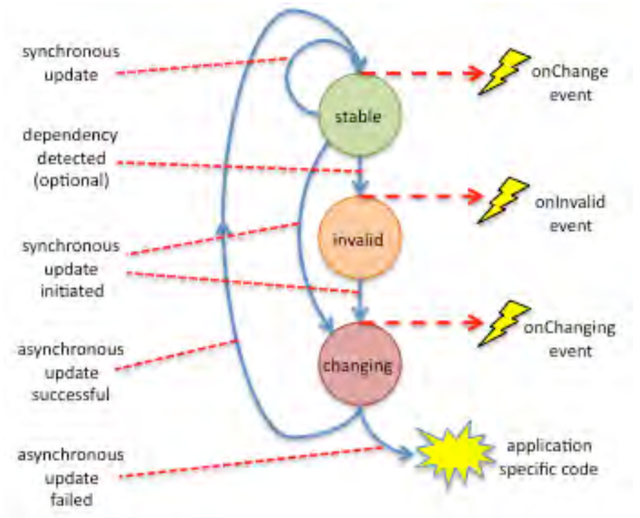
```
display_query_var.setJSON(
  tree_url,  'tree_to_sparql',
  {'type':table,'fields':fields,'tree':tree}
);
```

**Fig. 8 setting an active variable through an AJAX call**

Currently the AJAX call has to be of a particular form. However, the intention is to remove these restrictions on the service in the future by adding pluggable filters.

## 4. AAV – DETAILS AND INTERNALS

Figure 9 shows the main states of an asynchronous active variable (AAV). The 'stable' state is when there is no update in progress and the value represents a stable valid value.



synchronous update
dependency detected (optional)
synchronous update initiated
asynchronous update successful
asynchronous update failed
stable
invalid
changing
onChange event
onInvalid event
onChanging event
application specific code

**Fig 9. Main states of AAV (© Alan Dix)**

## 4.1 Basic Behaviours

The simplest use case is when the value is set synchronously using the `set` method. This both updates the current value, and also triggers a '`changed`' event. Application specific callbacks can be attached to this event and any such registered callbacks will be triggered at this point. Typically this will include updating the current display, but may also include updating other variables, either synchronously or by initiating asynchronous updates.

The second uses case is where a variable is set asynchronously. This can either be done by application-specific code, or more conveniently using the AJAX/JSON utility method `setJSON`. This changes the state of the active variable and also triggers a '`onChanging`' event. Callbacks for this event will typically add some sort of 'loading' or 'updating' message, maybe grey out or remove the old content of affected display areas. When the asynchronous activity is complete, the value is update and a '`onChange`' event triggered, as in the synchronous set.

A variant of this scenario is when the application detects that a variable's value is in some way invalid due to a data dependency even though an asynchronous update has not yet been initiated. This could happen, for example, in the QbB application when the query learning is initiated. The `query` variable is set asynchronously, so that it enters a 'changing' state. This allows a callback to update the display of the query. When the query is updated, this then means the set of currently selected entities in the list is changed. However, this list is effectively out of date as soon as one *starts* to change the query. One might want to emphasise this by removing or changing the highlight while the query is being updated. To do this a callback can be added to the `query` AAV's `onChanging` event, which then calls the invalidate method of the `selection` AAV, recording the data dependency. Display code can then attach itself to the `selection` AAVs `onInvalid` event and change the highlighting appropriately.

## 4.2 Dealing with Failure

Failures of the asynchronous update are currently left to the application to handle, maybe resetting the variable to some default value, or the previous value. With more experience we may add some further convenience methods to perform common recovery procedures. Note there are two forms of failure here. The most fundamental is actual failure of the AJAX code, and so represents a critical problem in the back-end service or network connection. In contrast, 'soft' errors are effectively 'successful' AJAX return values, which encode some form of error in their result. The former will almost always involve a fairly major application-specific response (although, of course, common in mobile applications). The softer forms of failure seem most suitable for 'default' actions.

## 4.3 Application Managed Asynchrony

As noted previously, the `setJSON` method provides the simplest means to update values asynchronously. However, if application developers require more control, for example a lengthy calculation is required, or a specialised network protocol used, AAV provides a number of alternative mechanisms.

The most basic mechanism is to simply call the `startChange` method directly, which simply triggers the `onChange` event. Once the data is ready (e.g. AJAX call completes) the application code can simply set the value using the synchronous `set` method. While this is provided for simple interactions the AAV does not 'know' what is updating it and therefore cannot help to deal with race conditions or conflicts if there are multiple attempts to update the same variable asynchronously (see section 4.4. below). This may happen if the user interacts before an asynchronous update has completed.

In the example of QbB for SemWeb, the user might select a number of entitles in the list and then press 'Make Query". This would then initiate an AJAX call to the backend to infer a query based on the examples, and also make the query window show some form of loading/working indication. However, if this takes some time the user might then select a few more entries and press "Make Query" a second time. In bespoke code or using the above startChange then set methods, the application code would need to keep track of the fact that there are several outstanding AJAX calls otherwise the completion of the first would update the query display and remove the loading/working indication.

To help in such situations, the AAV framework supports an `ActiveVarProvider` interface, which allows more active cooperation between the AAV and application code, particularly when there is the possibility of multiple updates. The application developer implements a provider class implementing the `ActiveVarProvider` interface or extends the generic base class. The must include three methods:

`setVar` – gives a reference to the variable, often redundant for very specific code, but useful for writing more generic cases.

`start` – called when you should start processing

`abort` – called if you should stop processing rather than continue to completion.

Assuming it has not been asked to stop by the abort method, the developer code can set the value of the variable when the processing is completed using the special AAV method `provideSet(provider,value)`. Note this includes the identity of the provider (usually, '`this`'); the reason for this is explained in section 4.3. In the case where the processing fails, the provider instead calls the `provideFail(provider)` method on the AAV.

Having implemented such a class, the application code can set the AAV value using the `setAsync(provider)` method giving an instance of the provider class. The AAV will then manage calling the provider's `setVar`, `start` and, if needed, `abort` methods when appropriate.

An alternative mechanism is also provided using the AAV's `getAsyncSetter` method. This returns an `ActiveVarSetter` instance (which is also a provider). This setter has its own `set` method that can be used when the application code has a value available, and also an `onAbort` event, to which callback can be added. The framework ensures that only the value set by the most recent setter is actually passed on to the AAV.

## 4.4 Multiple Overlapping Updates

When there are several synchronous updates, the behaviour is obvious: each `set` method updates the AAV value and triggers the `changed` event.

For asynchronous updates the situation is slightly more complex as a fresh asynchronous update may be initiated before the previous update has completed. The solution adopted for AAV is to cancel the earlier update and only retain the last asynchronous update, in a manner similar to multiple synchronous updates.

Where the `setJSON` method has been used for the asynchronous set, the AAV framework manages all this automatically.

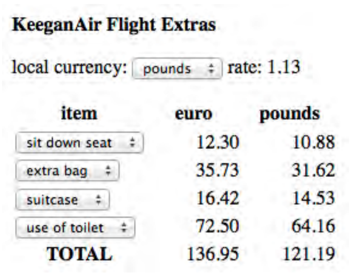
If the application has chosen to mange the update itself, this behaviour is enforced via the `ActiveVarProvider` interface. This is the reason for the `abort` method in the provider, which is invoked when a subsequent `set`, `setJSON`, `setAsync` or `getAsyncSetter` 'overwrites' the value being asynchronously updated. The AAV keeps track of the most recent provider and ignores others even if a provider still invokes `providerSet`, after being aborted. This is the reason why the provider is given as a parameter to the `providerSet` and `providerFail` methods.

Note that while this behaviour seems to be the right one for values representing current up-to-date state, it may be that need arises for 'stream' like variables, where each update matters (e.g. to animate in some way). If these use cases become apparent, then a stack-like mechanism of multiple providers may be added to the framework, or an alternative stream type added. However, for present the simpler mechanism seems sufficient.

## 4.5 Simultaneous Updates

The above mechanisms deal with simultaneous updates due to user interactions being faster than remote feedback. However, there are also potential race conditions that can occur within a single user interaction once one adopts a more data-driven approach to code.

As an example consider the extras calculator for a low-cost airline. The user selects extras from the drop down lists, then an API returns the cost of the item in Euro, which is then converted into a chosen local currency and the values added up to give an overall total.



**KeeganAir Flight Extras**

local currency: pounds rate: 1.13

| item | euro | pounds |
|---|---|---|
| sit down seat | 12.30 | 10.88 |
| extra bag | 35.73 | 31.62 |
| suitcase | 16.42 | 14.53 |
| use of toilet | 72.50 | 64.16 |
| TOTAL | 136.95 | 121.19 |

**Fig 10 flight extras calculator**

Assume there is an AAV '`rate`' for the current exchange rate, an array of AAVs `euro[0-3]` for the costs in euros (as returned by an AJAX call to the API), AAVs `local[0-3]` for the converted costs and an AAV `local_total` for the total cost. For the sake of the example, we assume the calculation of the total cost is done remotely using an AJAX API. If coded in a purely data flow manner we would register an `onChange` callback for each `euro[]` AAV which would update the corresponding `local[]` AAV, but also a callback on the `rate` AAV that would update all the `local[]` AAVs. Finally an `onChange` callback on each `local[]` AAV would initiates the calculation of the `local_total` using `setJSON`.

Now imagine what happens when the user changes the local currency. The `rate` changes which then updates each `local[]` AAV in turn. As each `local[]` is updated its `onChange` event is triggered and a `setJSON` call made on the `local_total` AAV; that is, `local_total` is set once for each extra.

Although each asynchronous set will cancel the previous one, this will only be after a request has been sent to the server using up bandwidth, causing local delays while the call is initiated, and wasting server time starting to service the API call.

While this is a little contrived to make a simple example, the general problem of multiple paths from the same initial event can occur in practice [8]. With AAV, the user never sees an inconsistent result, because previous asynchronous sets are aborted. However, as we see, it can lead to wasted effort.

If this is likely to be a problem, the AAVs can be set to wait a short period (typically 10ms) before actually starting their asynchronous setters. Because this is on a timer it is not actually initiated until all the updates directly due to the user interaction or current callback are complete. If during this period a new value is set the old setter is never started.

This timer-based mechanism does not remove every case of multipath dependencies, but does not require a full declarative notation that would be necessary for a complete runtime dependency analysis.

## 4.6 First Class Event Model

A number of the classes in the framework have one or more events to which callbacks can be added. This is common in user interface toolkits, for example, in Java one adds a callback for mouse events with:

```
component.addMouseListener(myListeneer)
```

or in jQuery, to monitor changes to a text component:

```
textComponent.change(myHandler)
```

Under the hood these are typically managed by some form of delegate object which has handlers added to it and then runs these when a relevant event is triggered.

In AAV the equivalent delegate object is made directly available as an object property. If `var` is an AAV `var.onChange` is an `EventPort` object. An `EventPort` has just three methods: `attach(handler)` and `remove(handler)` to add/remove a handler from the list of things to run when the event occurs, and `trigger(value)` which is called when the event occurs to run the handlers for the event.

This means that in the application code rather than `var.onChange(callback)`, as in other frameworks, instead one writes `var.onChange.attach(callback)`. This is slightly more verbose, but means that the event port is a first class object, allowing a level of decoupling between when things happen (events) and how values are updated (status).

## 5. DISCUSSION

## 5.1 Revisiting the Design Goals

In the introduction the broad design goals of AAV were described: flexibility and simplicity informed by theoretical understanding. These have worked together to create the current framework. The design was driven by practical needs, only adding features as they were needed and use cases arose; thus leading to simplicity/parsimony. However, this could easily end up with an ad hoc solution, which failed to generalise or broke in unexpected ways. It is the theoretical foundations that help avoid these problems, not least in dealing with potential race conditions in generic, yet flexible, ways. The flexibility goal is also crucial, again informed by theoretical considerations. Rather than creating a notation or language that 'solves' the problem of in-flux indicators, instead AAV provide a mechanism that *supports* the developer in solving these problems. Because the developer is still in control, unnecessary complexity can be avoided. Furthermore the developer is able to address new problems, not foreseen in the framework, and by so doing suggest new features.

## 5.2 Initialisation and Shadow Variables

One example of this is the use of 'shadow variables'. In the examples in section 3, the `query` variable was replaced by an AAV `query_var`, its value can be accessed as `query_var.get()`. However, in the actual code in most cases the original variable was 'left behind' in the code and its value set by an `onChange` handler on the corresponding AAV. This was partly due to incremental change of the code to use AAV; by having the shadow variable the rest of the code did not need to be altered and could still access the shadow variable.

However, these shadow variables tend to be particularly useful during initialisation. The AAV can be given an initial value when it is created, but sometimes the right initial value is not known when the variable is created, or sometimes variables are reinitialised, for example, when a new data store is selected. In some cases the AAV can simply be set using its `set()` method which will call the corresponding `onChange` handlers. However, in other cases this default behaviour is not wanted and the shadow variable is set directly without setting the corresponding AAV, hence avoiding running `onChange` handlers. These issues of initialisation were also encountered in the Garnet/Amulet family of toolkits [2].

This common use case suggest that some form of `setSilent` method might be useful and/or some means to automatically bind shadow variables to AAVs. However, in the spirit of parsimony this will be added only if it reoccurs sufficiently often. It is, of course, possible to put off this framework design decision because of the flexibility of the framework to allow such workarounds.

## 5.3 Callback vs. Procedural Coding

AAVs encourage a callback style of coding. This is common in many open-source platforms that allow plug-ins (e.g., WordPress filters and actions), as well as being normal in interface toolkits and UIDEs. One feature that is common in callback-based plug-in APIs is some form of priority, so that when there are multiple handlers for the same event/action the developer has some control over the order of execution. In contrast, most UI toolkits adopt a simpler last-in–first-served or first-in–first-served model, perhaps because multiple callbacks are relatively rare.. As AAV have been developed primarily for UI development, they so far adopt a default first-in–first-served model.

While event-based code is common in many practical applications, most programming courses start with a more procedural style of coding. Perhaps because of this, jQuery provides a `promise` interface and `deferred` objects that encapsulate asynchronous actions (including AJAX calls), and include a `then` method, which allows asynchronous actions to be chained together in a procedural manner.

In code developed using AAV both styles of coding have been seen. In some cases an AAV is declared and immediately an `onChange` handler is added with code for everything that must happen after a change; a more procedural "*after the variable is changed then do this*". In other places an AAV, say `x`, is declared, and then somewhere else in the code, typically where a user interface element or another AAV depends `x`, an `onChange` handler is added to `x`; a more data driven "*I depend on* `x`".

In current code these different uses of AAVs and also uses of jQuery sit side by side. It may be that slight variants of the existing AAV primitives would suit these styles better, or would enable close integration with jQuery promises.

## 5.4 Declarative Notation and Structured Data

One possibility would be to include a form of mini-language or more structured way of declaratively declaring variable interdependencies. This would be far closer to the form of older UI toolkits such as Garnet [2], or recent declarative notations such as Arrowlets [23]. There would be advantages to this approach, for example, in better detecting and dealing with multipath dependencies as discussed in section 4.5.

However, if these were added to AAVs it would be in the form of an optional mechanism in the spirit of supporting the developer, not an all-encompassing mechanism. For example, we have experimented with dependency declarations of the form:

```
new Dependency( in_bind, out_bind, engine )
```

Where `in_bind` and `out_bind` are bindings of names to AAVs and `engine` is an object that can initiate asynchronous activity when given a binding of names to (raw) values (matching the names in `in_bind`) and when it returns invokes a callback with a corresponding binding of out values. The dependency framework then registers appropriate `onChange` handlers for each input AAV to invoke the `engine`, and sets AAVs when it completes. This is promising, but as yet unclear whether useful enough to add.

Similarly, it may be useful to be able to add callbacks to parts of structured objects without making each part a separate AAV, for example, if `person` is an AAV to be able to refer to `person.part('address. number').onChange`.

## 6. CONCLUSION

Various authors have noted problems with AJAX-based interfaces and their potential to create unusable interfaces [11,21,22,24]. The AAV framework does not prevent UI disasters in AJAX-based web interfaces, but hopefully makes it easier to do certain things right. In the case of the QbB interface this meant that a mass of bespoke AJAX response handlers was replaced by a more declarative description of what needed to be updated when changes occurred. Whereas in the pre-AAV code only the query pane showed a "loading" indicator, after introducing AAV adding these became simple enough that any part of the interface that can potentially be waiting for update now shows an appropriate indicator.

In some ways the differences between AAVs and more traditional active variables are minor, in most part simply an extra `onChanging` event. However, as we have seen, allowing asynchronous updates creates the potential for various race conditions that need to be dealt with carefully to avoid inconsistencies in the interface. AAVs support the programmer in managing these, and so ultimately improving the user experience.

Finally, it is worth returning to the design goals and the way AAVs have been designed to *support* rather than take over the developer's job. This has allowed an incremental approach to development that is more likely to lead to a usable and useful framework.

## 7. REFERENCES

[1] Bizer, C., Heath, T. and Berners-Lee, T. 2009. Linked data – the story so far. *Int. J. Semantic Web Inf. Syst.*, 5, 3, 1–22.

[2] Bradley T. Vander Zanden, Richard Halterman, Brad A. Myers, Rich McDaniel, Rob Miller, Pedro Szekely, Dario A. Giuse, and David Kosbie. 2001. Lessons learned about one-way, dataflow constraints in the Garnet and Amulet graphical toolkits. *ACM Trans. Program. Lang. Syst.* 23, 6 (November 2001), 776-796. DOI=10.1145/506315.506318

[3] Clarke, G. (2011). Google plan to kill Javascript with Dart, fight off Apple. *The Register*, dated 14th September 2011, accessed December 2011. http://www.theregister.co.uk/ 2011/09/14/google_project_kill_javascript/

[4] A. Dix (1994). Que sera sera - The problem of the future perfect in open and cooperative systems. *Proc. HCI'94: People and Computers IX*, Eds. G. Cockton, S. W. Draper and G. R. S. Weir. Glasgow, Cambridge University Press. 397-408. http://www.hcibook.com/alan/papers/loop-HCI94/

[5] A. J. Dix (1994). *Seven Years on, the Myth Continues*. RR9405, University of Huddersfield. http://www.hcibook.com/alan /papers/myth95/

[6] A. Dix and A. Patrick (1994). Query By Browsing. *Proceedings of IDS'94: The 2nd International Workshop on User Interfaces to Databases*, Ed. P. Sawyer. Lancaster, UK, Springer Verlag. 236-248.

[7] Dix, A. and Abowd, G. (1996). Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, 11, 6, 334-346.

[8] A. Dix and G. Abowd (1996). Delays and Temporal Incoherence Due to Mediated Status-Status Mappings. *SIGCHI Bulletin*, 28, 2, 47-49.

[9] A. Dix, R. Beale and A. Wood (2000). Architectures to make Simple Visualisations using Simple Systems. In *Proc. AVI2000*, ACM, 51–60.

[10] A. Dix (2007). Network-Based Interaction. Chapter 14. *Human-Computer Interaction Handbook (2nd ed.)* , eds. A. Sears and J. Jacko. CRC Press, 2007.

[11] Dix, A. and Cowen L. (2007). 'HCI 2.0? Usability meets Web 2.0'. Panel position paper. *Proc. HCI2007*, Vol. 2 , British Computer Society.

[12] Paul Dourish. 1996. Consistency guarantees: exploiting application semantics for consistency management in a collaboration toolkit. In *Proc. CSCW '96*, ACM, 268-277. DOI=10.1145/240080.240300

[13] C. A. Ellis and S. J. Gibbs (1989). Concurrency control in groupware systems. Proceedings of 1989 ACM SIGMOD International Conference on Management of Data, *SIGMOD Record*, 18, 2, 399-407.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[15] *The Go programming language*. Google. retrieved 30 Sept 2011. http://golang.org/

[16] Greenberg, S, and Marwood, D. (1994). Real-Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface, Proc. CSCW'94, ACM, 207–217, DOI=10.1145/192844.193011

[17] Gutwin, C., Lippold, M. and Graham., N. (2011). Real-time groupware in the browser: testing the performance of web-based networking. In *Proc. CSCW '11*. ACM, 167-176. DOI=10.1145/1958824.1958850

[18] Krasner, G. and Pope, S. (1988). A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *JOOP*, 1(3).

[19] McGuffin, L. and Olson, G. (1992). *ShrEdit: A Shared Electronic Workspace*, CSMIL Technical Report, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, 1992.

[20] Meyerovich, L., Guha, A., Baskin, J., Cooper, G., Greenberg, M., Bromfield, M. and Krishnamurthi, S. (2009). Flapjax: A Programming Language for Ajax Applications, *OOPSLA '09*, October 25–29, 2009

[21] Nielsen, J. (2007). *Web 2.0 'Neglecting good design*. BBC News. 14 May 2007 Available at: http://news.bbc.co.uk/1/hi/technology/6653119.stm

[22] Nielsen', J. (2007). *Web 2.0 Can Be Dangerous...* , Jakob Nielsen's Alertbox, December 17, 2007. accessed Dec 2011, http://www.useit.com/alertbox/web-2.html

[23] Khoo Y P, Hicks, M., Foster , J. and Sazawal, V. (2008). *Directing JavaScript with Arrows (Functional Pearl)*, Technical Report CS-TR-4923, Dept. of Computer Science, University of Maryland, August 2008.

[24] Pilgrim, C. 2008. Improving the usability of web 2.0 applications. In Proceedings of the nineteenth ACM conference on Hypertext and hypermedia (HT '08). ACM, NY, USA, 239-240. DOI=10.1145/1379092.1379144

[25] Prud'hommeaux, E. and Seaborne, A. (2008). *SPARQL Query Language for RDF. W3C Recommendation 15*. January 2008. http://www.w3.org/TR/rdf-sparql-query/

[26] Quinlan, J. (1986). Induction of decision trees. *Machine Learning*, 1(1).

[27] Greg Rosenberg. 2007. A look into the interaction design of the new Yahoo! mail...: and the pros and cons of AJAX. *interactions* 14, 2 (March 2007), 33-34. DOI=10.1145/1229863.1229882

[28] B. Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer* 16, 8 (August 1983), 57-69. DOI=10.1109/MC.1983.1654471 http://dx.doi.org/10.1109/MC.1983.1654471

[29] M. J. Stefik, D. G. Bobrow, and K. M. Kahn. 1986. Integrating Access-Oriented Programming into a Multiparadigm Environment. *IEEE Softw*. 3, 1 (January 1986), 10-18. DOI=10.1109/MS.1986.232428

[30] Tonkin, E. (2006). *AJAX And Usability Issues*, UKOLN. Dated 4th March 2006, Accessed 14 Dec 2011. http://www.ukoln.ac.uk/qa-focus/documents/briefings/briefing-94/html/