Brian Carroll
Sept. 25, 2006

Proposed Changes to ALF Event Format Considered Harmful

Recently, a number of changes have been proposed to the ALF Event Format that was originally documented in the ALF Architecture draft document last October. In once sense, that Event Structure has held up well and served us through two Proofs-of – concept (POCs). However, the reality is that both POCs were, by their nature, rather simple and neither exercised the full capabilities of the structure. So the recent examination of the ALF Event is healthy and timely as we are preparing for ALF's 1.0 release candidate.

API stability is important

Eclipse quality emphasis on stable APIs

The Eclipse website has a page (http://www.eclipse.org/projects/dev_process/eclipse-quality.php) and some articles (http://www.eclipse.org/articles/) on the importance of stable APIs. While this principle is obvious to developers, it is surprising how frequently it is violated. Sometimes the cause is haste and a lax attitude; other times the cause is not thinking through the use cases for an API and not anticipating likely uses.

For ALF Schemas and vocabularies are the APIs. So the importance of creating a Event definition that we can live with for a while is critical.

To add to the urgency for a stable Event schema is the current disarray in the web service world on how to version web services. The W3C Web Services Architecture group has taken a few stabs at the problem, but no satisfactory recommendations have emerged.

Architect "in the large" and Implement "in the small"

We could have an extended discussion on the purpose of architecture, but I view it as a way to match the requirements for a system with the appropriate materials (design approaches and technology), but in the process not unduly constrain the design to the initial requirements. The requirements for any architected system tend to change over time, and an overly constrained architecture tends to permeate the implementation and lock it into inflexibility. On the other hand, a flexible design that anticipates growth and changes experienced in similar system builds in long term adaptability and utility.
If for cost or schedule reasons, the implementation is constrained, as long as the architecture is flexible, the system can be expanded later with greatly reduced impact and cost.
Now, I recognize that the notion of "thinking through a system" is not in vogue. I've found that agile approaches that incrementally approach a solution work better on smaller projects or projects where an overall architecture has already laid out the "big decisions". I've seem cases where the incremental approach of starting with some basic goals and refactoring as you add in additional requirements can approach an implementation

equivalent to one architected from the beginning – but it took much longer. However, performing that exercise in the medium of thought during the design phases is considerably more efficient that evolving code through repeated refactoring in the medium of code. And constant evolution is not good for APIs, those interfaces upon which other teams depend and where stability is paramount.

The myth of Schema extensibility

One argument for defining a simpler Event format now is that "XML is extensible; we can add elements later without breaking existing programs." That is certainly true for XML Schema definition, especially if you take care to design in extension points. However, the notion of extensibility frequently completely breaks down as a Schema is translated into code to parse an incoming message. The way the DOM navigation or SAX event stream handling is coded can unwittingly make a program intolerant of any extensions, even if the Schema permits it. An experienced XML developer knows how to avoid such lock in, but how many developers have enough understanding to do that. And with the increasing reliance on toolkits that generate parsing code, the actual extensibility of the resulting implementation may vary widely.

ALF's process mechanism may evolve

Many of the proposed changes are designed to make the writing of BPEL processes simpler. BPEL 1.0 has been the most widely adopted and viable process standard, but it does have limitations and awkward aspects. While it has decision making capabilities, expressing even simple if-then-else logic in XML can be awkward. Mapping data among structures is certainly simpler if the structures themselves are simple, thereby reducing decisions.

Despite its current status as the de facto standard for automated process expression, do we really want to limit the capabilities of ALF's other capabilities to what is simple to express in BPEL? I would argue that the mechanism for expressing ALF processes will evolve. While the time was not right for ALF to initially embrace an Enterprise Service Bus (ESB), many provide their own process mechanisms. Some are based on BPEL, often translating BPEL to the native process and data mapping expressions of the particular ESB. ALF anticipates leveraging the Eclipse SOA Tools project in the future to offer alternative transport and data mapping mechanisms that may offer greater ease of use, performance, or quality of service. And while BPEL 1.0 is the process expression language of choice today, might we want to take advantage of the capabilities of native ESB process expressions?

In addition work at the Object Management Group (OMG) on the Business Process Modeling Notation (BPMN), is already claiming it will eliminate the need for BPEL in favor of higher level, more human-friendly expressions of process.
The point is that we should not unwisely cripple certain aspects of ALF, such as the Event Format, to make processes easier to write – there are better ways to accomplish that.

Proposed approach to resolving the discussion

The *mechanism* we should use is the Eclipse process to resolve the discussion, via transparent discussion open to the Eclipse community, followed by a vote of the ALF Committers.

Why Standards are broad and Profiles narrow

But I would like to suggest that the *approach* we should take is the same one employed by standards committees. Standards, like APIs, are intended to be useful for some time. Therefore, standards are written broadly to encompass a variety of situations, perhaps even use cases not anticipated at the time the standard is written.

However, being able to accommodate a variety of circumstances often does not lead to interoperability. So standards committees or interoperability groups develop *profiles* that define subsets of the specifications, often to ensure interoperability for smaller classes of use cases.

That combination successfully preserves the broadness and longevity of the specification while the profile limit the breadth to well known subsets of the use cases.

Proposed approach to architecting-in-the-large and implementing for today

I propose we take that approach with the ALF Event Format: Define the structure broadly, to accommodate a variety of use cases, and for stability, and define a Profile for ALF 1.0 which limits the options to those that are known and understood today and that make the writing of BPEL 1.0 processes easier.
That approach satisfies the Eclipse goal of stability of APIs.

Discussion of Proposed Changes

Some of the proposed changes are good ideas. They clean up aspects that were unclear or not fully specified. In this (and subsequent messages and), I will provide arguments and use cases that support not making some of the proposed changes. The first couple are:

Argument against the elimination of lightweight events

This is a subtle propose change that was not clearly described. By way of background, the original notion of ALF Event was as a relatively small lightweight structure that conveyed only the essential information that an identifiable event occurred. The motivation was that some tools may generate events that are of no interest and should be ignored. The ALF Event Manager would filter those events and discard them. But why should the tool expend time building a large event with all the attributes of the entity that changed if the event may be discarded. So the Event could contain just the Basic Event with no “payload”. Such lightweight events would be easier to build and less expensive to transmit, log, … The ideas was that if the tools invoked by a ServiceFlow ever got to process the event, they had enough information to call back to the tool that emitted the event to obtain the details. Clearly, such events were not suitable for all tools, such as batch tools that would not be around to be called back.

The reasons for keeping lightweight events are:

- Not all tools may allow the ALF administrator to "turn off" unneeded events, the ALF Event Manager will need to do that via its filtering mechanism.
- Some payloads may be large and expensive to transmit. So emitting an event with just sufficient identifying information such that the details can be obtained later, if really needed, can make the system more efficient.
- Some environments in which ALF will operate may involve hundreds of developer and a high volume of events. Having a mechanism for passing "just enough" identifying information in an event, can reduce network and processor loading. (Clearly, if we know a "payload" will always be needed and processed, it would make sense to include it routinely, unless that payload is large.)
- Passing only the identifying information in the event makes it easier to further reduce message traffic by packing the notification for many object changes into a single message. This is especially true if subsequent filtering in the ServiceFlow does not process all the objects.

Therefore we should retain the ability to send ALF Events that contain a payload or not. In the cases where a payload is not sent, if a tool subsequently needs the details, it can call back to the tool that originated the event.

The motivation for the <ProductCallbackURI> was to provide an endpoint where a tool could be called back to obtain the payload details. This was inadequately explained in the ALF Architecture document [mea culpa], but the notion was that there would be a standard, GetObject() or GetDetails() operation that all ALF conformant tools would expose that, given the identifying information from an ALF Event, would return the details (or payload) for that object instance.

Argument against the Proposed Change to Remove the <Object Array>

The original Event Schema defined an <Object Array> which had multiple intended purposes, though only one was documented. [again, mea culpa]. The use that was documented was to allow a relationship object within a tool to be referenced by the instances of the 2 or more entities that the relationship connected or associated. In hindsight, I believe that is the less useful use. The more interesting use is where the <Object Array> holds the identifying information for multiple objects of the same type. The use case is where many changes are made to a tool, but all can be processed efficiently at once by a single ServiceFlow, and that processing would be much more efficient than if a separate ServiceFlow were launched for each object.

The ALF project lead, Ali Kheirolomoom, has identified this use case in the Serena TeamTrack product (which is being ALF enabled), where multiple issues are transitioned to a new state. If each transition raised a separate event, the ensuing processing would be inefficient, but if all those transitions were packed in a single event, the processing could be much more efficient.

A key to making the use of the <ObjectArray> clearer is to add two element that indicates the purpose of the Object Array in a particular instance:

- An element that explicitly conveys the number of objects in the array, so that that count can be more easily checked by a BPEL process.

- Another element that conveys the intended use of the array: to convey the two or more object associated with a relationship instance that has changed, or simply as an array of objects. The role= attribute on the <Object> element was intended to perform that role, but a single element outside the array would have been a better design: more explicit and easier for BPEL to process.

I urge we retain the flexibility of the ALF Events capability of reporting changes to multiple objects of the same type in a single event, that is, retain the <Object Array> and add the elements to make its use easier and more explicit.

Sidenote: An appreciation for the thorough review

As the primary author of the EventManager and BasicEvent design, I would like to acknowledge appreciation the review and scrutiny. Many of the changes that were proposed truly improve the design. Some, I believe, catch oversights in the documentation and original explanations.