Developing Specifications for the Accessible Rendering Item Format (ARI)

Daniel R Rehak
Daniel R Rehak Consulting, LLC
January 2015

There are a number of ways that the ARI Format can be represented in a specification or related documents. Notes about some of the potential options are presented, along with a few points to consider.

Standards and Specifications

Standards and Specifications are designed to provide conformance guidance, i.e., sufficient information to decide if an implementation conforms to the requirements of the standard or specification.

- They tend to be detailed, precise, well structured and boring.
- They are not terribly useful for content developers or system implementors.
- They should not be created without the active involvement of both content developers and system implementors; you risk creating something than cannot be built and worse, that does not meet the end user's needs.

Standards and specifications are similar in concept.

- Formally (as defined by the standards' organizations), a standard can only be produced by a nationally or internationally recognized standards development organization (SDO), e.g., ISO/IEC, IEEE Standards Association, ANSI, DIN.
- Anything else is a consensus specification, typically produced by an industry group/consortium, e.g., IETF, W3C, ECMA, IMS.
- In terms of practice, this distinction is immaterial; many specifications are more widely adopted and used than formal standards.
- Remember, the idea of a standard is to "standardize" an approach and resolve discrepancies between alternatives; a specification can be more "definitional".

The process to get to consensus will differ widely by organization; but starting with a submission based on an operational system created externally to the standardization or specification development process is a useful starting point. Developing a standard or specification through a consensus group process composed of participants with conflicting agendas results in many compromises.

Being conformance documents, standards and specifications need to differentiate the parts of the document that are relevant for conformance, e.g., *normative* content, from other *informative* content.

Organizations determine the actual document content and organization/structure of their standards or specifications. Common parts include (order may vary):

- Introductory Material (informative)
- Conformance Statements (normative)

    - Overview of what it means to be conformant; i.e., the parts of the document relevant to conformance.
    - The general criteria for evaluating conformance vs. non conformance.
    - Conformance for extension, e.g., are extensions permitted, if/how they can or cannot be conforming. Extension mechanisms need to be considered as part of the initial design.
    - Conformance is often specified by behavior, e.g.:
        - A conforming data producer.
        - A conforming data consumer.
        - A conforming data transformer.
        - A conforming data instance.
    - What is required vs. optional. This is often behavior specific; producing a particular data item may be optional but any consumer may be required to process the data.
- Technical Content (normative)
    - The detailed technical description, e.g., data models, APIs (multiple sections as needed).
    - Most technical content uses words like "SHALL", "MAY", "REQUIRED" to specify conformance details; RFC 2119 provides the common definitions of these and related conformance terms (use of RFC 2119 is often cited as part of a terminology or conformance section).
    - Within technical content, illustrations, diagrams, examples, code listing, notes, footnotes, etc., are usually considered ancillary and informative (and often noted as such).
    - When available, using formal representational notations is good.
    - How the technical content is presented may be described as part of a terminology or conformance section.
- Acronyms and abbreviations (normative)
- Glossary (normative)
    - Some organizations have a standard glossary for common domain-specific terms; the standard glossary is included by reference, either explicitly or implicitly.
    - All other special (domain-specific) terms should be defined; confusion often arises due to different people having preconceived notions of what specific (and sometime common) words mean.
    - Don't needlessly make up new terms or definitions.
- Terminology (normative)
    - Specific terminology or document conventions.
- Normative References (normative)
    - References to other standards and specifications essential to evaluate conformance.
    - Most referenced standards have versions and can be updated, e.g., XML 1.0 vs. XML 1.1.
        - Normative references need to specify what version of the referenced content to use if that content is updated, e.g., use the specific version cited or use a newer version if it is available (using newer versions is generally not recommended).
        - Addressing how to handle versioned references is usually done with a blanket statement that describes the default, e.g., for dated references, the cited version applies; for undated references, the most recent version applies.
- Informative References/Bibliography (informative)
    - References to other documents not essential to evaluate conformance.
- Examples (informative)
    - Complete, illustrative, conforming examples.
    - Sometimes a separate document.
- Supporting Material, e.g., Annexes or Appendices
    - These can be informative or normative, e.g., schemata and corresponding schemata locations might be either normative or informative.
    - Useful supporting material includes:
        - Conceptual Models; key ideas behind the approach.
        - Design decisions: capturing history of why things are the way they are so those who do follow on work can understand why things are the way they are (and know the implications of "breaking" something).

- Implementation guidance: suggested practices for an effective implementation.
- Security considerations: data and operational security, data privacy.
- Locations of sample data, schemata, code.

Design / Implementation / Profile Documents

An alternative to a formal standard or specification is a set of design documents or design specifications (conceptual, detailed) that describe the system but are targeted at implementation rather than conformance.

- They contain most of the information that is in a standard or specification, but in a less formal way.
- The target audience is anyone developing an implementation, focusing on capabilities, not conformance.

When a design builds upon and integrates other standardized components (not just using underlying standardized technologies), e.g., ARI using content packaging, a collection of design/implementation documents can be created that describes how to **profile** the existing pieces into the whole – subset, extend and combine the parts. Such a profile should only build upon other standardized components. Anything new to the ARI should be defined in its own specification or implementation documents independent of being incorporated into the overarching profile. The profile describes how all the pieces work together.

User Guides

End users (content or item developers, tool builders) don't care and shouldn't care about specifications and standards, conformance or interoperability. Most specifications are useless for this audience. User guides that show how to use the ARI are essential. These guides only include why something is being done if *essential* to understating how to do it. Guides should be organized around user roles, e.g., the item developer doesn't need the details of packaging, someone building a packaging tool doesn't need to know the details of how an item works.

Licenses

No matter what the documentation or outputs, licenses, copyright and patent issues are hidden gotchas that should be addressed up front. Things to address include:

- Determine who owns the copyright to all work, not only the finished documents but also all related content, including any software.
- Determine the licensing and permitted reuse strategy for the final work.
- Establish a policy for permitted use of all material submitted from anyone who is not doing "work for hire" or otherwise not formally/contractually/legally covered by some other agreement.
- Establish a patent disclosure and use policy for all submitted material.

It should be clear to all contributors how their contributions can and will be used; you don't want claims to pop up after the work is done. Everyone should understand any patent issues pertaining to use of the contributions or final work, including existing patents needed for implementation.

Versioning

Things change. Establish a versioning strategy and a version numbering/labeling scheme up front. Things that are versioned (and can be versioned *independently*) include:

- A document
    - There can be editorial changes that do not impact conformance, behavior or interoperability.
    - There can be new technical updates/additional features.

- A schema
    - Can be updated independently from any other documents.
- An API
    - Can be updated independently from any other documents.

To maintain independent updating, all major works should be independently versioned; parallel updates can be released when a change impacts multiple items.

Often version numbers are included in a data model instance, API or schema.

- Specifications should describe how to deal with different versions of data or different versions of APIs as part of interoperability, including both forward and backward compatibility and having different API versions processing different data model versions.

Implementations

A working implementation can be used as an alternative or an adjunct to a specification. Implementations come in two flavors; *reference* and *sample* (reference implementations are often confused with sample implementations).

A reference implementation is "definitive".

- If there are any discrepancies between *the* (one) reference implementation and the standard or specification; the reference implementation is correct and the document is wrong.
- Reference implementations need to be complete and cover the entire feature set.
- Many so called "reference" implementations are really sample implementations.

Sample implementations are "illustrative".

- If there are any discrepancies between the sample implementation and the standard or specification; the document is correct and the implementation is wrong.
- A sample implementation need not implement the entire feature set.
- A sample implementation may include non standard features and extensions.
- Some organization require multiple (2 or 3) independent, proven interoperable implementations of a feature before it is included in the specification; each implementation may provide only a subset of all the features, but the union of all implementations covers all the features.
- Multiple sample implementations can be used to demonstrate interoperability.

An alternative to an implementation is a set of core, public libraries, APIs, or schemata that can be used either as samples or as part of an official conforming implementation. Providing an official or semi-official implementation of core features or critical parts of can provide a jump start on interoperability and let others build value-added features on top of an established common approach.

No matter how many disclaimers are associated with a sample implementation, or how poor or inefficient it is, if the ARI achieves wide-spread support, expect that any sample implementation will become part of someone's commercial product offering.

All implementations (and any other software products) should be fully documented. If successful, the potential life cycle will be years, and most likely none of the original designers and developers will be involved in future revisions and maintenance.

Test Suites / Test Harnesses / Conformance Programs

Test suites and test harnesses provide sample data, a testing environment, etc., used to test data instances or implementations. Test suites can be used to demonstrate functionality or can be used to test a specific range

of behavior. Complex systems require large numbers (hundreds or thousands) of tests to cover all functionality and all of the potential interaction of discrete components.

A conformance program will test a designated subset of conformance requirements (testing all requirements may be onerous). Conformance may be done via self-test, or a formal conformance-testing or certification program (often best administered externally).

Conformance generally tests that things work or that different systems are interoperable. They do not stress test and they do not offer comprehensive tests to probe edge cases and to try to make things fail. They do not test efficiency or good implementation approaches.

Someone will try to game a conformance program to achieve a conformance certification with minimal work. Let's says the specification says something is "required", and something else is "optional" but optional means only an optional feature that may or may not be used. If the optional capability is needed, it should be done in the way specified as "optional". If the conformance program only tests for required features, someone may build an implementation that provides the optional feature in a proprietary or non-interoperable manner, not as described in the specification. Conformance certification needs to go beyond just a checklist of proper use of the specification to include use of the specification as envisioned.

Other Design Tips

Selecting the "latest and greatest" technology or version of an included technology may not be advantageous. Take as example picking XML 1.1 vs. XML 1.0.

- Anything in XML 1.0 is forward compatible with XML 1.1, so there is no need to select XML 1.1 if all you are using is features from XML 1.0.
- It takes several years for industry practices and support tools to fully support new versions such as XML 1.1, so using the new version may make implementation more difficult.
- Picking XML 1.1 is only advantageous if the new XML 1.1 features are essential.

Many new technologies never reach critical mass and sustainability. Lifetime of anything built upon other components needs to consider the entire life cycle of all the component parts. Pick stable, well established technologies if they do the job.

Having a core, underlying conceptual model is good. When questions or conflicts arise, the tenets of the core model can be used to address them. Simple, clean abstract models are good; if it looks messy, it's going to cause problems.

Common approaches, common data models, common data types etc., should be used whenever possible (e.g., use established data formats for dates, times, languages, best practices for XML, JSON, ...). Common models improve interoperability. Using established best practices leverages community expertise. The benefits of using a different or specialized approach needs to outweigh the cost of being unique.

Don't sub optimize (what are the life cycle costs). Don't optimize what you haven't measured. Understand how technology influences optimization (e.g., using 2^n sized bit strings makes perfect sense in hardware level protocols; using 2^n sized arrays make no sense as data fields where there is additional system overhead).

Unlimited, unbounded or unspecified data sizes exist only in theory. Real systems impose a (hard) limit somewhere. Better to pick and define a specific minimum size that an implementation *must* process rather than leave it to the implementer to pick something. A good guideline is to require support for ~ 99% of ALL potential data sizes, e.g., 2 standard deviations.

Don't use two value logic (T/F) to encode 3 or 4 value data (T/F/NA, T/F/Null, T/F/NA/Null). Corollary: Null is not blank or 0 or an empty string.