Ideas for UIMAJ v3

A place to collect ideas for the next version of UiMA Java core.

A nice way to see what's new on a page is to click "view change" on the line right underneath the title above. From the compare page you can progressively (with one click) compare previous versions too.

Here's a place to assemble a "spec" of what might actually be in version 3: UimaV3Spec

Defining UIMA's value proposition(s) from a Data perspective

The UIMA project's mission says in part it's related to the "spec" in OASIS which was more a spec about a wire-format (i.e., serialization format) for UIMA, based on XMI (XML Metadata Interchange) standard. XMI has not caught on very well. This topic is to flesh out from a data interchange point of view what the important things are.

Topic is here.

Framework interoperability

There are many big-data frameworks now. UIMA has a particular slant on things to encourage component development and reuse (I'm thinking of externalization of the Type System, merging of type systems). UIMA also has its scaleout approach, and the RUTA workbench facility. This topic is where we can think about UIMA components in other frameworks (e.g. Apache Spark), or vice-versa.

Interoperability could be facilitated by more standards around REST service packaging.

Complete JSON deserialization with an eye toward being "permissive" to receive data models from other frameworks?

Big changes

More dynamic type systems

Support annotators that have no type system, or that have just a piece of a type system This has two sub-ideas:

- Annotators which are Generic, perhaps taking a specification at run time, and producing the specified outputs
- The ability to deserialize data with type/feature metadata, either as separate type systems or embedded (e.g. JSON), and then have a way for annotators to work with selected parts (types/features) of that data, without knowing ahead of time the data's full type system.

UIMAv3DynamicTypes out this discussion.

More use of Java compiler (ecj) and decompiling

A portable Java compiler from Eclipse (ecj) and decompiling capabilities (e.g. Procyon) are appropriately licensed and could be part of the startup.

- JCasGen could be "automatic" for merged type systems, and merged instances of JCasGen'd user classes?
    - Users still would need a generated version for their code to compile against.
- Pear definitions for JCas cover classes could be merged?
- Could generate one kind of Java cover class for all types. (lazy, load on demand
    - eliminate / reduce use of TypeImpl in runtime.
    - generate for all merged types (except custom built ins)
        - (as opposed to current impl, where no JCas cover class is generated if it doesn't exist - the "standard" one is used instead)
- use class loader technology to support multiple type systems
    - Having same-named types, sharing the JCas cover types for those, but (after merging) having different sets of features.
    - This would only be used for UIMA (merged) Types that have same name but have different feature sets.
    - Current design uses the same JCas cover class for differing type systems (e.g., ones that have a different # of features for a type). In this case, the JCas cover type only is being used to set/read slots it knows about; other facilities might be used to read/set additional slots.

Feature Structure == an instance of its Java Cover class

One representation only of a FS; the static fields of the class have the typeImpl info..

Features represented directly as fields.

- To get around "reflection" slowness:
    - Support set/get by int <- class <- feature-name-string
    - Support set/get (bulk) ? <ordering among fields significant?>
    - possibly use something like ReflectASM which is like Java reflection but has a byte-code generator and is much faster (but probably not as fast as custom support code compiled into the Java Cover class).

Problem with name clash with existing non-JCas class

There are use cases where JCas cover classes are not being used for some classes, yet the users define a class named identically to a JCas cover class. This is permitted in UIMA v2.

For example, you could have a class x.y.z.ConceptType which was defined as a Java enum. You could also have a UIMA type, x.y.z.ConceptType, and work with it without using JCas APIs.

One possible approach is to map the uima type name to a special java class name for these use cases so there's no collision; of course, the user would need to use the non-JCas APIs for this type.

Problem with use-case of changing TypeSystems

This has one serious issue, not yet solved, illustrated by the use case:

1. make a pipeline,
2. deserialize some CAS's type system, and then deserialize that CAS
3. do some generic processing on that CAS
4. repeat 2 and 3 in a loop, with different type systems each time.

Setting up the merged type system and generating the Java class definitions means that those classes might need to be replaced, but they might be linked to the existing code.

Data Model (Types and Features) adapters

One of the values for UIMA is the facilitating interoperability among components. One difficulty in this is that different components may have somewhat different data models, with different names for similar things.

This topic looks at making this better.

Automatic JCasGen of merged type system

At startup-time, the Java classes for types could be generated from the "merge" of type information in all components (this merge is done in current UIMA, and is intended to let annotators "extend" each other's type model with additional features). Any component could run JCasGen on the types they were using, in order to get classes they could compile against. These would be ignored in favor of the generated-at-startup-time version.

This could be done using either ECL or via code generation.

User customization of Java cover classes, and PEAR classpath isolation issues

Currently users may customize their JCas cover classes. PEAR classpath isolation allows the use case where different customizations are present in one pipeline. The current implementation supports this, and switches the set of JCas cover classes as Pear boundaries are crossed. The idea of a Feature Structure being an instance of its cover class breaks down when multiple definitions of this exist. Some ideas for fixing this.

Alternatives: generating JCas definitions from merged type systems

There are two approaches - more dynamic and less dynamic.

- Have a separate step, run outside of the UIMA runtime environment, which generates the full set of JCas classes (except the built-ins), from the merged type system
    - Configure the JVM classpath to include these classes typically at the front of the classpath.
- Have an integrated approach, based on classloaders, that generate classes at type system merge (or lazily) and load them either all at once or via a special version of UIMAClassLoader, lazily.

More here.

Support parallel execution of components (if they don't depend on each other)

This would require parallel implementations of many of the internal data structures (e.g., indexes), which come at a cost, so this should be configurable, or better yet, automatically managed.

We could even consider implementing parallel capable versions of some internal UIMA Types (Lists, arrays, and Maps if we add that).

Consider ideas from other popular big-data frameworks: Hadoop, Spark

These typically have approaches to type systems that use user-defined Java types, and allow any kind of Java objects in the fields. There are new kinds of Serialization / Deserialization that work for all kinds of Java objects, but are more efficient than Java reflection-based approaches (e.g. Kryo used by Spark).

Add support for Collections and Maps

Users have wanted these kinds of objects; some implementations I've seen have tried to implement Sets using a combination of HashSet and UIMA FSLists, duplicating the data and keeping things in sync, which was very inefficient. More on this topic here.

More concurrency

Support parallel running of pipeline components.

Careful trade-off vs slower due to synchronization, cache-line interference. Key is to separate things being updated.

    Consider special index support for this

Supporting Java 8 streams

Iterating over FSs: alternative: have generator of FSs, process with stream APIs

- Possibly having a new kind of managed component? being either
    - The "functions" the standard operations on streams use
    - new standard operations on streams (unlikely I think)
    - I think this might be deferred until we have some more experience

(Unlikely) Making the element of the "stream" be a new CAS - replacement for CAS Multipliers. Seems like the wrong granularity... Maybe best to let Java evolve this for a few more releases.

New packaging support using component boundaries

Some new capabilities may benefit from specifying boundary actions. Some possible actions:

- If a PEAR defines an external resource for use within the pear, it could put the impl classes within the PEAR classpath boundaries. see **UIMA-4499** - Getting issue details... STATUS
- If a component defined some customization of JCas for some types, and we implement this via hooks, the hooks could be inserted/withdrawn at the boundaries. This is similar to switching JCas implementations at PEAR boundaries, but applies outside of PEARS and on finer grain size (e.g., just one component).
- A component uses some adapter(s) for type system differences, within the component boundaries

Additional capabilities

Integration with popular component reuse systems (e.g., Maven)

- for both Type Systems and Components
- additional maven like metadata store for
    - type system documentation
    - relationships - components, input / output, types
- additional tooling enabling composition and type adaptation

Integration with popular stores (Cloud, no-sql, etc.) for results and downstream processing

Improvements in debugging

Ability to specify "capture" of intermediate CAS results at specific points in the pipeline, integrated with JMX (Some of this has already been done as part of UIMA-AS, but should be put into the core)

CAS differencing

Custom UIMA JMX console

JSON support (deserialization)

Version 2.7.0 added JSON Serialization, but is missing deserialization - add that. Also not completed is whatever enhancements are needed to permit flexible interoperability with UIMA services that implement partially compatible type systems, and Delta CAS support (for sending back to a client just the changes made to a CAS that was sent from that Client).

Other changes

Integrate key ideas from uimaFIT

These include:

- Alternative, Java-centric way of specifying a type system - user write a Java class with annotations.
- Alternative, Java-centric way of specifying configuration information
- Convenience methods (e.g. selecting groups of feature structures using SQL-like specifications)
- Others?

Better support for "run-time" dynamic typing

Adding support "dynamic" typing - see paper: http://aclweb.org/anthology/W14-5209. An interesting thought is to add this without giving up the compile-time speed and checking advantages of statically strong typing. The result would be some kind of hybrid, with more performance available to fully specified static definitions.

Supporting "combining specifications" that map type systems

Different components should be easily combinable even if they have different type systems, if a mapping can be found and specified. For more complex mappings, custom adapters could be supported?

Using the Web to facilitate component combinations

User wanting to combine X with Y should be able to lookup on the web and download the adapter or 90% of the work predone. It should be easy for users to share this information on the Web.

Connecting to other initiatives in world-type-modelling

Google, Bing, and Yahoo have standardized on microformats for semantic HTML web markup, and have a big schema defined (see http://schema.org/); some kind of integration that lets users easily make use of this information would be nice. It would be nice to be able to use this without any download /copying, by referencing the (gradually evolving/changing) web site that specifies these things. For instance, see the entry for "place" : http://schema.org/Place

Judicious substitution of other packages for hand-built code

There's a plus and a minus for this - plus: we get better tested, better function (perhaps), better performance for some typical capabilities (e.g., parsing XML to/from Java Objects). Minus - it make the code depend on these other packages. Also, if it's working fine now, there's little motivation to invest in changing it.

Some areas to consider:

- XML parsing and writing for descriptors - use JAXB or Jackson (already used for JSON support)