Quo Vadis UIMA?

Thilo Götz
IBM Germany R&D

Jörn Kottmann
Sandstone SA

Alexander Lang
IBM Germany R&D

Abstract

In this position paper, we will examine the current state of UIMA from the perspective of a text analytics practitioner, and propose an evolution of the architecture that overcomes some of the current shortcomings.

1 Introduction

UIMA (Unstructured Information Management Architecture, (UIMA, 2014)) became an Apache open source project in 2006. By that time, it had already seen about 5 years of development inside IBM. That is, it was already quite mature when it came to Apache. The core of UIMA has seen relatively little development since then, but the world around it has moved on. Thus, we should ask ourselves: how does UIMA currently fit into the software landscape, and what could be directions for future development to ensure that the fit remains good?

At its core, UIMA is a component framework that facilitates the combination of various text analytics components. Components are combined into analysis chains. Simple chains are relatively easy to build, more complex workflows are difficult.

UIMA components define and publish their own data models. The actual data is handled by the framework, and created and queried via APIs. Methods to serialize the data, e.g., for network transport, are also provided by the framework. The data access and component APIs exist for Java and C/C++, so that it is possible to combine Java and native components in a single analysis chain.

Over the years, the UIMA eco system has grown to include scale out frameworks that sit on top of the core and provide single or multinode scale out for UIMA analysis.

2 How does UIMA integrate with other frameworks?

The short answer is: it doesn't, and that is where we see the biggest chance of evolution for the future. UIMA is a framework itself, and does not integrate well with other frameworks. It has its own component architecture, its own idea of logging, its own idea of controlling file system access, etc.

This was fine at the time UIMA was built as we didn't have generic component frameworks as we have them today (actually they did exist, but they were not successful). It is still fine today if you just want to run a few files from disk, or simply don't need another framework.

However, nowadays general purpose (component) frameworks are in widespread use. On the one hand, we have enterprise frameworks such as J2EE (J2EE, 2014) and Spring (Spring, 2014) that host classic n-tier applications with some kind of http frontend. On the other, we have big data scale out frameworks such as Apache Hadoop (data at rest, (Hadoop, 2014)) or Apache Storm (data in motion, streaming, (Storm, 2014)). If you are building a non-trivial application, chances are good that you will make use of the many services such frameworks offer and base you application on one of them.

For example, imagine you want to scale out your analytics with Apache Hadoop. As long as you run your entire pipeline in a single mapper, everything is fine. There may be a few details that need to be ironed out, but in general, this kind of scale out scenario works pretty well. However, suppose you want

Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT, pages 77–82,
Dublin, Ireland, August 23rd 2014.

to run part of your analytics in a mapper, and another in a reducer, because you need to do aggregation of some kind. So you need to serialize the results of your first analysis step, and pass them on to the the second one. While this can be done with quite a bit of work and deep UIMA skills, it is much more complicated than it ought to be.

When we consider data-in-motion scenarios that are handled by Apache Storm or similar frameworks, this issue becomes even more pressing. There the individual units of processing are usually very small, and the framework handles the workflow. This is completely incompatible with the UIMA approach where UIMA needs to know about the entire processing pipeline to be able to instantiate the individual components. If every annotator works by itself and doesn't know where its next data package will come from, coordinating the data transfer between all those components becomes a huge effort.

It should be noted that even if you are willing to stay completely within the UIMA framework, there are things you might reasonably want to do that are difficult to impossible. For example, you can not take two analysis chains that somebody else has configured and combine them into one. Suppose you obtained a tokenizer and POS tagger from one source (so an aggregate analysis engine in UIMA speak), and some other chain that builds on that from somewhere else. You can't just combine the two, because it isn't possible to make an aggregate of aggregates. You need to take everything apart and then manually reassemble the whole thing to make one big analysis chain. Generally, this is possible, but it can be very difficult for something that should be really simple to do.

Over the years, UIMA has accumulated its own set of scale out frameworks. While this is ok if all you want to do happens in UIMA, that is rarely the case for most industrial applications. Instead, the framework is chosen for the kinds of data that are expected, or for how it fits into the rest of the software landscape. So a UIMA specific scale out framework extension is at best a niche solution.

Another area of business computing where UIMA plays no role at all today is mobile. UIMA is just too fat to run on a mobile device. That is not to say that there isn't a place for something like UIMA on mobile devices, but it would need to be much more lightweight than what we have today. We will make some suggestions in a later section how that could be achieved.

On a positive note, uimaFIT (Ogren and Bethard, 2009) is a move in the right direction. It allows you to configure your analysis engine from code, avoiding a lot of the top-level XML configuration that makes dealing with UIMA pipelines so cumbersome. However, UIMAFit is a layer on top of the base UIMA framework, and while it addresses some concerns, it does not do away with the underlying issues.

3 The data model

The UIMA data model is very strict in the sense that it is statically typed. That is, when an analysis chain is instantiated, all information about the kind of data that can be created in the chain needs to be known ahead of time. This approach has certain advantages, which is why it was chosen at the time. Specifically, it allows for a very compact encoding of the data in memory, and very fast access to the data.

On the other hand, such a strict data model is quite cumbersome to work with, particularly when you need to combine analysis components that may come from very different sources. For example, a common scenario is that you are using a tokenizer from some source, and want to add information to the tokens (e.g., POS tags or some kind of dictionary lookup information). You can't do this unless you modify the type system for the entire chain, even though it is only relevant from the time you actually add the information.

What we are facing is a classic trade-off between performance (memory and CPU) and developer convenience. While we still need efficient processing, given the size of the data sets we are looking at nowadays, we would still like to argue that developer time is a more expensive resource than machine power. Thus, it is time to move away from the static type system for data, and move towards a dynamic model that is a lot easier to handle from the perspective of the developer.

It should be noted that some of the constraints on data were realized by UIMA developers very early on, basically as soon as people started running analyses in more than one process (or on more than one node). Options were added on top of UIMA to make communication between loosely coupled compo-

nents easier. Again however, these were added on top the base UIMA framework and are essentially just used by UIMA's own scale out frameworks. That is, they are quite difficult to use by end users who want to use a different framework for scale out.

Another point that is made difficult by a static type system is the caching of analysis results in (relational or other) databases. This is a point that comes up again and again on the UIMA users' mailing list. Actually, it is not so much the writing out as the reading in of previous results when your data model has changed in some minor way that doesn't even affect what you want to do with the data. Here also, a dynamic approach to data modelling would go a long way towards simplifying this process.

Finally, the need for the JCas (a system where you can offline create Java classes for your data structures) would go away entirely.

4 Analysis component reuse

The main raison d'être for UIMA is analysis engine reuse. Why then do we often find it so difficult to reuse existing components?

The main obstacle for an implementor who wants to make an analysis engine (AE) reusable is that the target type system is not known to and varies between users. The best a reusable AE can do is to allow users to map the input and output types as part of the configuration. Sadly, UIMA doesn't support this use case explicitly. There are no types for configuration values which are intended to contain type system values such as UIMA type or feature names.

Certain type systems are too complex for a type mapping and sometimes an AE is programmed against a specific type system. To integrate an existing AE in such a scenario is only possible with an adapter in front and behind the AE in order to convert the input and output types dynamically. The adapters must be implemented as an AE and they increase the complexity of the UIMA pipeline.

An implementor who updates an AE can only update the jar file used to distribute it to the users. An AE consists of two parts, the implementing classes and a XML Descriptor file. The descriptor file contains both the schema for the configuration and the user defined configuration values. Users typically copy this file and change the configuration values. An implementor can't update the users' XML descriptor to a newer version. A user will have to manually merge the changes of the customized file and the updated descriptor.

Even though it is possible to build reusable UIMA AEs, users repeatedly prefer integrating common open source NLP components themselves rather than dealing with the existing UIMA integrations (e.g., Apache OpenNLP). This is quite sad given that UIMA set out to make this sort of manual integration effort redundant.

5 Generic analysis engines

Another advantage of having a dynamic type system that can be added to at runtime would be to make generic analysis engines easier to develop and more convenient to use. By a generic annotator, we mean an annotator that is configured by the end user to perform a certain task. This could be for example a generic dictionary matcher, or a regular expression matcher as they already exists on Apache as a downloadable components.

So what is the issue then? Today in UIMA, the type system is defined ahead of time. So it is not possible to write, e.g., a regex matcher that creates a new kind of annotation, unless that annotation is also added in the type system. So you can either require users of such generic annotators to always change things in two locations at the same time: the type system specification, and the regex configuration. We all know what kind of chaos that leads to. Or you can have the generic annotator produce a generic annotation, such as a `RegexAnnotation`. The results range from the awkward (because results need to be transferred to the real target annotation) to the unusable (because the fixed generic annotations are not flexible enough to represent the intended results).

If on the other hand the type system was dynamic, generic annotators could create new data types at runtime from an end user specification, with no extra effort.

6 Design point: a layered architecture

In this section, we will propose a design that should overcome the shortcomings of UIMA that we have identified. While there are many specifics to be worked out, there are sufficient details to form the basis of an ongoing discussion. We'll begin with an overview of the design we envision, and then drill down into some details.

1. Off-line, serialized data layer (JSON and/or XML)
2. In-memory data access and creation APIs
3. Data (de)serialization
4. Component instantiation (with data source and sink)
5. Workflow
6. Application layer (logging, disk access etc.)

This is to be read from top to bottom as layers that only depend on the previous layers and are fully functional by themselves (without lower layers). We'll look at some aspects of this design proposal below.

6.1 Serialized data definition

As the basic layer of UIMA, we propose a non-programmatic, off-line format for UIMA data. Independently of what that format actually is, it is the basic level at which software components can interoperate. This is what goes over the wire/air, what is written to storage. So it is important that there is one well-defined serial format that everybody understands.

We follow the OASIS UIMA standard (Ferrucci et al., 2009) in this approach. Of course the format chosen in the standard (XMI) is very complicated. The argument given at the time was that XMI is itself a standard and that it would thereby be easy to achieve integration with other tools using this standard. If any such integration has ever materialized, we don't know about it. It seems much more important to use a format that is just powerful enough to do what needs to be done, and which is at the same time as simple as possible.

A likely candidate for such a format is JSON (ECMA, 2013). It is a simple, human readable format with very widespread use and good tooling support. There is also built-in support for many of the constructs we are used to in UIMA. Special support would have to be defined for annotations, and for references to build true graph structures (JSON just defines trees, like XML). There is no reason not to have an XML format as well, if that is desired, and tools that translate between the two.[1]

6.2 In-memory data access

By this we essentially mean what is now the UIMA Common Analysis System (CAS, (Götz and Suhre, 2004)). Apart from the changes for a dynamic type system, which might not be so large, our layering approach requires that a CAS should be independent of the rest of the framework, which is not the case today. However, if we give up on the idea of a global static type system, we are most of the way there.

If we take the approach from the last section seriously (i.e., that the serialized data format is the basis of our architecture), then the in-memory APIs need to map pretty directly to the external data definition.

6.3 Data serialization and deserialization

There is not much to say about this point. Data that was defined in terms of the in-memory API needs to be serialized to the external format, and vice versa. A more relaxed approach to the data model should make things a lot easier on this front.

[1] Two reviewers have independently remarked that if that is the case, we might as well stick with XMI, and have a translation to/from JSON. At a certain level, that is correct. However, that would require anybody who wants to deliver a compliant service to read/write the verbose and complicated XMI format. That seems counterproductive as the whole idea is to make the use of UIMA at lot easier than it is today.

6.4 Type System mapping and adapters

A type system mapping specifies which input and output types an analysis engine should use. A dynamic type system allows an analysis engine to create or extend the specified output types. Analysis engines are typically chained together in a pipeline, where the outputs of the previous analysis engines are the inputs of the following analysis engines. Just by defining the type mappings, the analysis engines will be able to dynamically create the necessary types.

We see the type system mapping as a useful tool for analysis engines when there is a common understanding in the community on how to encode the information in types. If there is a disagreement on how to model the data, the analysis engine should use an adapter to translate between the two type system worlds. With a dynamic type system the analysis engine can create input and output types in the flavor the implementor prefers and adapters can create the output types the user prefers.

6.5 The upper layers

Our proposal de-emphasizes the upper layers of the architecture, simply because we expect that many, if not most, applications will run on some sort of component framework anyway. What we have defined up to now is sufficient to run the kinds of scenarios that we described in the earlier sections.

We also believe that the configuration information that currently must be defined in the UIMA XML component descriptors is much better left to specific code of the individual components (note that one of the goals of uimaFIT is just that). If a component has a few settings you can adjust, then let there be APIs for those settings. It is so much easier to do this in your programming IDE than to fiddle with some poorly documented options in XML.

Nonetheless, for analysis workflows where the entire flow is running in a single process, it is still useful to have standard APIs that facilitate component instantiation, definition of analysis chains or simple workflows, and finally an application layer that can do some sort of simple document I/O. This is still useful for debugging, tooling and to build demo applications.

7 Conclusion

As we hope to have illustrated, UIMA is faced with a number of challenges. The UIMA framework doesn't integrate well with other framworks. Although UIMA can be used together with other frameworks, the integration is much more complicated than it should be. The statically typed data model is troublesome to work with. The type system is specified ahead of time and can't be modified during runtime. Type systems are often defined on a per-project basis and are incompatible with each other. Due to these limitations the reuse of UIMA analysis components across projects is often avoided.

For similar reasons the development of generic annotators is very limited. A generic annotator can't create new types during runtime and must either put a heavy burden on the end user, or return limited results.

To overcome these challenges, we have proposed the following changes:

- Move to a dynamic data model
- Standardize on an easy-to-use data interchange format (replace XMI by JSON, for example)
- Create a stratified architecture that supports framework integration

We have spent relatively more time talking about the perceived UIMA shortcomings than about our proposed solution. Why is that? The entire approach hinges on the decision to move to a dynamic data model. Once this step is taken, only then do certain architectural options become available. It is the crucial step, and execution of the rest of the design would not be very difficult in our opinion.

Our point of departure was the fact that UIMA analysis is difficult to integrate with other frameworks. However, we have seen that many current issues are also addressed by this evolution. We hope this paper will spawn a discussion among UIMA users and developers both, so we can make sure UIMA stays abreast of technical developments and continues to be relevant to the text analytics and computational linguistics communities at large.

References

ECMA. 2013. The JSON data interchange format. ECMA-404 (RFC 4627). Technical report, ECMA International. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

David Ferrucci, Adam Lally, Karin Verspoor, and Eric Nyberg. 2009. Unstructured Information Management Architecture (UIMA) Version 1.0, OASIS Standard. Technical report, OASIS.

Thilo Götz and Oliver Suhre. 2004. Design and implementation of the UIMA Common Analysis System. *IBM Syst. J.*, 43(3):476–489, July.

Hadoop. 2014. Apache Hadoop. http://hadoop.apache.org/.

J2EE. 2014. Java Enterprise Edition. http://www.oracle.com/technetwork/java/javaee/overview/.

Philip Ogren and Steven Bethard. 2009. Building test suites for UIMA components. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP 2009)*, pages 1–4, Boulder, Colorado, June. Association for Computational Linguistics.

Spring. 2014. Spring Framework. http://projects.spring.io/spring-framework/.

Storm. 2014. Apache Storm. http://storm.incubator.apache.org/.

UIMA. 2014. Apache UIMA. http://uima.apache.org/.