Qanda and the Catalyst Architecture

Pranav Anand David Anderson John Burger[*]
John Griffith Marc Light Scott Mardis Alex Morgan
The MITRE Corporation
Bedford, MA 01730 USA

Introduction to Qanda and Catalyst

Qanda is MITRE's entry into the question-answering (QA) track of the TREC conference(Voorhees & Harman 2002). This year, Qanda was re-engineered to use a new architecture for human language technology called *Catalyst*, developed at MITRE for the DARPA TIDES program.

The Catalyst architecture was chosen because it was specifically designed for fast processing and for combining the strengths of Information Retrieval (IR) and Natural Language Processing (NLP) into a single framework. These technology fields are critical to the development of QA systems.

The current Qanda implementation serves as a prototype for developing QA systems in the Catalyst architecture. This paper serves as an introduction to Catalyst and the Qanda implementation.

What is Catalyst?

Catalyst is a framework for creating and experimenting with Human Language Technology (HLT) systems. It attempts to address several problems typical of current approaches to component-based HLT systems. The principal problems that Catalyst is designed to ameliorate are these:

- Systems do not scale easily to handle today's information processing needs. Systems are needed to process human language very quickly or in very large amounts.
- Experimenting with a variety of potential system configurations is difficult because each pair-wise component interaction typically requires specialized integration code for smooth operation.

The approach that we are using in the Catalyst framework to address these problems is to combine *standoff annotation* and *dataflow*.

Standoff Annotation

The Catalyst data model, like those of both the TIPSTER and GATE architectures(Cunningham, Wilks, & Gaizauskas 1996; Grishman 1996), is annotation based. A signal (text, audio, etc.) is augmented with annotations that mark up portions of the signal with supplemental or derived information.

[*]john@mitre.org

Catalyst annotations are standoff (versus inline) which means that the underlying signal is unmodified and annotations are maintained and communicated separate from the signal. By separating the signal from the annotations and annotations of different types from each other, Catalyst is able to automatically construct customized streams of annotation for each component in a system. The set of annotations, attributes and their names can all be transparently modified between each language processing component without modifying any component code or inserting additional scripts.

Every standoff annotation has an annotation type identifier, a start position, an end position, and zero or more attributes. The attributes are named fields that provide information derived from or associated with the annotated text. For example, a tokenizer might emit **word** annotations, with **text**, **stem** and **part-of-speech** attributes. The start and end of each such annotation would indicate where in the text the tokenizer found the words.

Dataflow Processing

In order to support distributed, scalable systems, Catalyst is based on a dataflow model of language processing components. We refer to these components as language processors (LPs). The dataflow model allows us to describe an HLT system directly in terms of the data dependencies between LPs. Furthermore, we are able to use the natural ordering properties of the standoff annotation indices to synchronize the operation of the various components.

Each LP in a Catalyst system is connected to others by annotation streams consisting of a flow of standoff annotations serialized according to the following predicate.

$$A_1 < A_2 \text{ if } \begin{cases} A_1.\text{start} < A_2.\text{start} \\ \lor\, ((A_1.\text{start} = A_2.\text{start}) \land (A_1.\text{end} > A_2.\text{end})) \\ \lor\, ((A_1.\text{start} = A_2.\text{start}) \land (A_1.\text{end} = A_2.\text{end}) \\ \quad \land\, (A_1.\text{annotation-type} < A_2.\text{annotation-type})) \end{cases}$$

As a node in a dataflow network, each LP has a declaration that defines its annotation input requirements and annotation outputs. A system declaration identifies the required language processors and the desired annotation stream connections between them (connections that satisfy each LP's input requirements). From these declarations Catalyst can arrange to deliver to each component only the annotations that are expected. Thus, components do not need to forward annotations unrelated to their specific function. For example, a sentence tagger may consume the bf word annotations produced by the tokenizer described above, and emit annotations indicating the boundaries of sentences. The sentence tagger need not copy the words to its output—if a third component requires both sentences and words, Catalyst will arrange to deliver the outputs of the tokenizer and the sentence tagger, suitably merged.

Catalyst's dataflow approach to building HLT system has a number of advantages.

- Error dependencies between components are limited to the precisely specified data dependencies.
- By using dedicated peer-to-peer channels Catalyst eliminates the cost of parsing and generating generic annotation interchange formats (such as XML) between independently constructed components.
- Component developers may work directly with an annotation model, rather that with particular data interchange formats. (Catalyst will support the exchange of data in XML for system I/O and for use with components not prepared for direct use in a Catalyst system.)
- A system can be run on a single machine or distributed across many. Individual components can be replicated to increase throughput.
- Component code can be simplified because the data presented is always consistent with the LP specification.

Distributability

Catalyst annotation streams are transported over sockets and can be connected between processes on many machines, permitting a wide range of processing strategies for optimizing system performance without having to rewrite component code. Once properly working on a single host, distributing a system across many machines requires only starting a server on each machine and editing a few lines in the system configuration file.

Control

A network of Catalyst servers exchange information for the purpose of creating and maintaining Catalyst-based systems. Connections are negotiated by servers and then handed-off to component processes. A single script, compiled from a static dataflow description of the system, works in concert with the servers to create each system. Servers also route and deliver control commands to each language processor.

Logging and Monitoring

A multi-process, distributed system can be difficult to debug and maintain. To assist component and system developers in this regard, Catalyst has both distributed logging and distributed monitoring capabilities.

The Catalyst log capability allows logger processes to collect information from some or all of the processes in a Catalyst system. Logging information includes events such as when language processors are stopped or started, user log messages, command events, errors, etc. Logs may be created at the same time the system is instantiated or may be added later as needed. Multiple loggers can be created simultaneously to record several views of the log at different levels of detail and can be used to create logs at multiple destinations.

The Catalyst monitor is used to examine the configuration and state of a Catalyst system. Using the monitor, a component or system developer can obtain snapshots of the current system configuration and track the flow of data through a system. The monitor provides information such as the list of current running components, the connecting annotation streams, the amount of information buffered with the system, the current indices for the various streams, etc. Debugging multicomponent systems such as Qanda requires a facility to examine the global system state easily. The monitor is an important tool for quickly diagnosing component interaction problems and identifying performance bottlenecks.

Information Retrieval in Catalyst

In addition to addressing some of the general problems of HLT system construction, Catalyst is also an experiment in developing a framework for combining NLP and IR in a single system. Standoff token annotations, grouped by term, form the basis of an inverted index for terms in a large corpus, similar to those used by traditional IR engines. By extending this usage to all other types of annotation, Catalyst permits the development of fast information retrieval techniques that query over NLP-generated products (see examples below).

Catalyst's dataflow model, combined with flexible inverted index streams, makes it possible to develop systems that can utilize both pre- and post-index NLP to improve the speed of query responses. Also, retrieval engines can be built that directly answer complex queries as needed for question answering (e.g., retrieve all paragraphs containing a person and one of terms A or B).

Implementing Qanda using Catalyst

Our previous TREC efforts have used inline-XML pipeline architectures, where all components monotonically added XML markup to retrieved documents. This approach had a number of problems:

- Components downstream had to understand (or at least parse) all upstream annotations, in order to ensure that these earlier annotations were properly replicated on output.
- This led to an inflation of the markup on documents: Often the final documents comprised 99% markup and 1% underlying character data.
- Some components' only purpose was to rewrite markup to make it more palatable to downstream components.
- It is difficult to parallelize such an architecture.

Our current Catalyst-based architecture suffers from none of these problems. Every component is delivered only the annotations that it requires to do its job. If a component is producing annotations that no other component currently

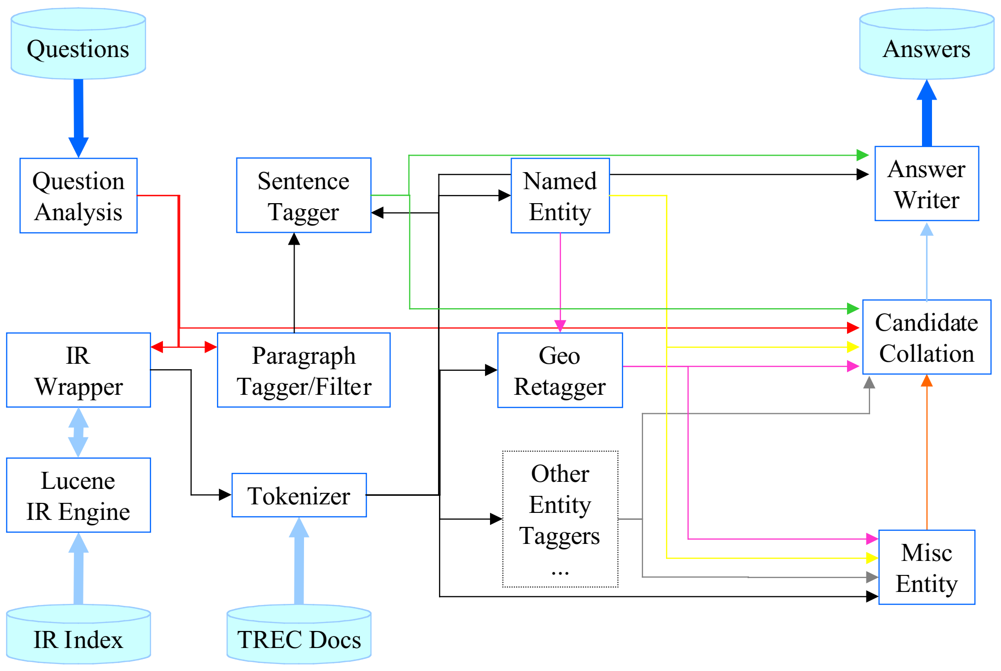

Figure 1: Qanda as a Catalyst System—Wide arcs indicate file system IO, narrow arcs are lightweight Catalyst annotation streams.

requires, Catalyst quietly drops them on the floor.[1] If necessary, we can instruct Catalyst to map between different annotation types in order to accommodate differences in the natural representations of different components.

Catalyst also allows us to lay the system out in a more natural manner than a single pipeline. Figure 1 shows our TREC system, which is naturally expressed as a directed graph. Note that many components do not need to communicate with each other, even indirectly, and can thus run in parallel, e.g., most of the entity taggers. Although we have not yet taken advantage of it, Catalyst will allow us to run language processors on different machines, even replicating slower components in order to increase throughput.

Using Catalyst

Working with Catalyst entails developing Catalyst-enabled components and assembling them into a system. Catalyst is designed to simplify this second task. First we describe the general way in which Catalyst systems are constructed; next we show two possible paths for integrating existing technology with Catalyst.

[1]Of course, the preferred behavior would be for the component to neglect computing such annotations in the first place, but at least they are not further processed.

Building a System in Catalyst

Figure 2 shows how the major components of Catalyst (the library, the server and the configuration compilers) are used in creating a running system. A configuration file is written for each language processing component (LP). It specifies which annotation streams it is able to process and which it generates. A component compiler transforms the configuration file into header files and other static information that are used to create each Catalyst-enabled executable. A system configuration file, referring to LP configuration files, defines which LPs are needed in a system and the stream connections that are required between the LPs. The system compiler transforms the system configuration file into a start-up script that is used to instantiate the system. The script (presently a PERL 5.0 script) communicates only with Catalyst servers, whose function is to create the operating system processes that will contain the LPs, establish connections between them, and forward configuration and control information from the script to each process.

The Catalyst library (linked into each component process) handles annotation communication between components and passes control information to and from the servers. The library can merge annotations from many different components and produce a single annotation stream specialized

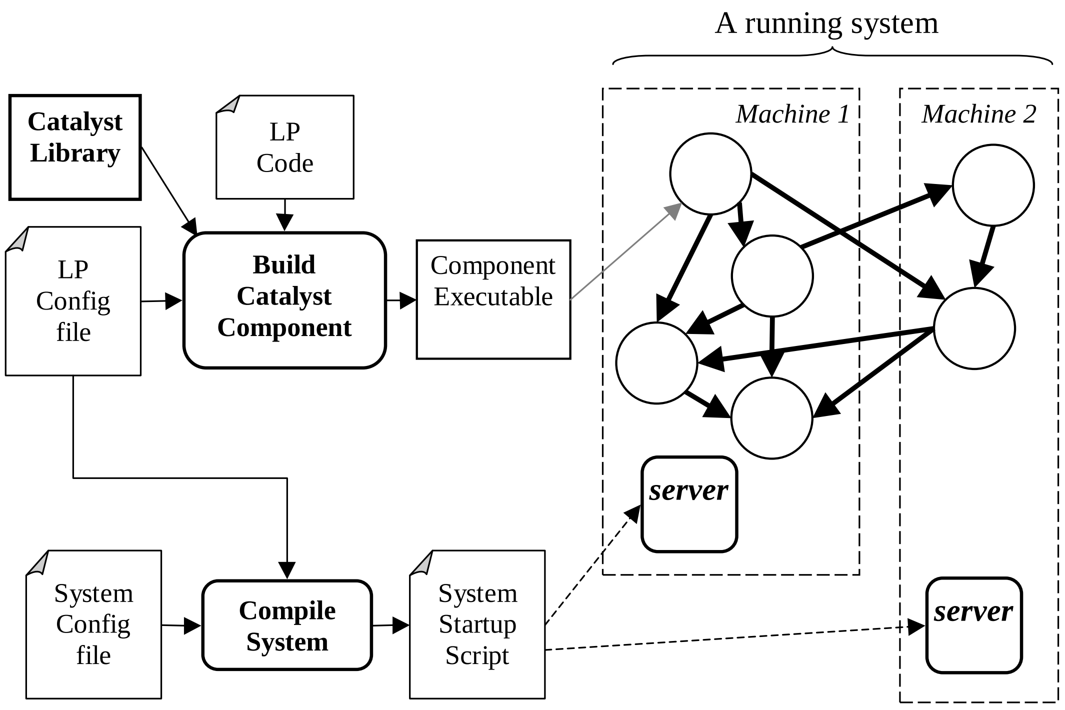

Figure 2: How Catalyst Builds a System—Configuration compilers transform configuration files into header, data, and script files that are used to create language processors[LPs] and systems. A compiled start-up script uses the network of Catalyst servers to create a running system. The system, once started, passes annotations via peer-to-peer communication managed by the Catalyst library.

to each component's input declaration. Similarly, a single stream of output from a component can be broken down into its constituent annotations and attributes and delivered piece-meal to many destinations. In this way, Catalyst delivers to each component, the precise set of annotations required by the component's declaration.

Integrating with Catalyst

There are two basic ways of integrating existing components with Catalyst: writing a Catalyst wrapper process and using the Catalyst API. The purpose of a Catalyst wrapper process is to convert the Catalyst annotations streams to and from a data format that an existing black box component uses. The Catalyst API, of course, provides direct access to all of Catalyst's features and provides maximum benefit.

A wrapper process allows one to connect existing technology into Catalyst without having to modify the component code. This is the only choice for components for which the code is unavailable. It would provide the advantage of delivering a precise set of annotations to the wrapped component but it would suffer from the cost of transformation to and from the appropriate interchange format. Also, transforming annotations from standoff to inline formats and back again (as most component technologies would require) can be difficult. The Catalyst project is, however, planning direct support for inline annotations in XML to facilitate integration via wrapper processes.

The Catalyst API defines a standoff annotation model and provides methods for sharing data via annotation streams. Standoff annotations allow for overlap in ways that cannot be constructed in an inline format such as XML, permitting, for example, components to output many possibly overlapping noun phrase bracketings or answer candidates. Additionally, a component can receive an annotation stream that contains the combined outputs of several components that all share the same task (e.g., it is simple to develop a component that looks at N different tagger outputs and selects the best tags by combining the results).

Future Directions for Qanda within Catalyst

Currently, work is proceeding within the Catalyst project on two important technologies: Persistent annotation archives and annotation indexes. The goal of archiving annotations is to store and then later reuse a stream of annotations. For instance, the tokenization and entity tagging in Qanda could be done on the entire TREC corpus ahead of time, then pulled from an archive at question-answering time. Consumer language processors will not be aware that their input annotations are being read from disk rather than being created by a "live" producer.

The goal of annotation indexing is to invert arbitrary text "containers", not just documents or paragraphs. Thus, one might query for archived **Location** entities containing the term *Berlin*, to answer a question such as *When did the Berlin wall come down?*. In addition, we want to index all annotations not just on the terms they contain, but on all of their other contained annotations as well. This is similar to the work of (Prager *et al.* 2000; Kim *et al.* 2001) but intended to be more general and comprehensive. With both of these capabilities in place, we imagine that complex queries might be formulated, such as:

> Retrieve **Sentence**s containing **Date**s and also containing the term *wall* and also containing **Location** annotations containing the term *Berlin*.

We believe that such targeted queries will allow for very fast and accurate question answering systems. Optimizing such queries is admittedly complex, however, as is determining appropriate scoring mechanisms. Deciding how best to use archived coreference information is also an issue. Nonetheless, we believe that Catalyst provides a valuable framework for such sophisticated language processing systems.

References

Cunningham, H.; Wilks, Y.; and Gaizauskas, R. 1996. GATE – a general architecture for text engineering. In *Proceedings of the 16th Conference on Computational Linguistics (COLING96)*. http://citeseer.nj.nec.com/43097.html.

Grishman, R. 1996. Tipster architecture design document version 2.2. Technical report, DARPA TIPSTER.

Kim, H.; Kim, K.; Lee, G. G.; and Seo, J. 2001. MAYA: A fast question-answering system based on a predictive answer indexer. In *Proceedings of the Workshop on Open-Domain Question Answering*.

Prager, J.; Brown, E.; Coden, A.; and Radev, D. 2000. Question-answering by predictive annotation. In *Proceedings of ACM SIGIR*. Athens.

Voorhees, E., and Harman, D., eds. 2002. *Proceedings of the Tenth Text Retrieval Conference (TREC-10)*. http://trec.nist.gov/pubs.html.