

Infrastructure Components for Large-Scale Information Extraction Systems

William W. Cohen

Center for Automated Learning and Discovery
Carnegie-Mellon University
Pittsburgh PA 15213
william@wcohen.com

Abstract

Large-scale systems for information extraction include many different classifiers and extractors. Experience in building such systems shows that finding an appropriate architecture is both difficult and important: in particular, in systems containing many learned components, it is important to cleanly share information between the components, and to flexibly sequence the actions of the components. In this paper, an architecture for large-scale information extraction systems is described, based a light-weight blackboard system for communication between components, and a declarative control system for automatically sequencing component-level tasks like classification, extraction, and feature computation.

Introduction

In this paper, I will describe what I believe to be a crucial piece of infrastructure for complex, real-world information extraction systems. Specifically, I will describe a *light-weight, flexible blackboard system* for communicating between extraction and classification components, and show how this information-sharing scheme can be closely integrated with a *declarative control system* that automatically sequences lower-level components, which perform tasks like classification, extraction, and feature computation.

The claim that such a system is “crucial” is to some degree at odds with the academic literature on learning and information extraction. Most published papers in these areas address a small set of well-defined tasks, such as text classification, named entity extraction, entity-name matching (e.g., (Cohen 1998; McCallum, Nigam, & Ungar 2000)) and entity association (e.g., (Jensen & Cohen 2001; Ray & Craven 2001)). In almost all published papers, performance on these problems is considered in isolation, independent of the larger context in which the subtask is performed, and independent of the architecture used to coordinate these subtasks.

However, realistic complete systems for information extraction and question-answering (e.g., (Knoblock *et al.* 1998; Craven *et al.* 2000; Murphy, Velliste, & Porreca 2001; Buckley *et al.* 1999; Cohen *et al.* 1998)) typically include many components, often including many types of classifiers

and extractors. In many cases, good overall system performance requires these components to interact closely. My experience in building such systems, in industry (Cohen, McCallum, & Quass 2000) and elsewhere (Cohen 1999), has been that architecting large-scale information extraction systems of this sort is quite difficult.

In particular, experience has shown that in systems containing many learned components, it can be surprisingly difficult to cleanly share information between the components, and to properly sequence the actions of the components. There are a number of reasons why this is true.

- *Multiple representations.* Large-scale extraction systems often use a diverse, heterogeneous set of techniques for classification and extraction. For instance, to extract information about university professors, it might be appropriate to use a character trigram model to determine if a page is in English; to use a hidden Markov model to extract personal names; to use hand-coded regular expressions to extract phone numbers and email addresses; to use an XPath (Clark & DeRose 1999) query to extract PDF and postscript papers; and to use a rote list to extract university names. In many cases, this means that the system will require *multiple representations* of what is conceptually the *same object*—for instance, a single web page might have a character-string representation, a token sequence representation, and a DOM representation.
- *Expanding sets of objects.* Information is shared between components about many different things. While some of these things are known to exist at the start of the computation (e.g., the page at “http://wcohen.com”) it is very common for new things to be “discovered”, or at least hypothesized to exist, during the course of a computation (e.g., “the common referent of the strings ‘William W. Cohen’, and ‘Dr. Cohen’”). Allowing a decentralized, heterogeneous set of components to create things makes sharing information more difficult—for instance, it is not enough to simply add labels to some fixed set of objects.
- *Relationships between objects.* In addition to stating properties about things (e.g., “page P is in the category ‘home page’”, or “entity name N is a ‘person name’”) it is also necessary for components to exchange information about the relationships between things (e.g. “entity names U and P appear in the same paragraph”, “univer-

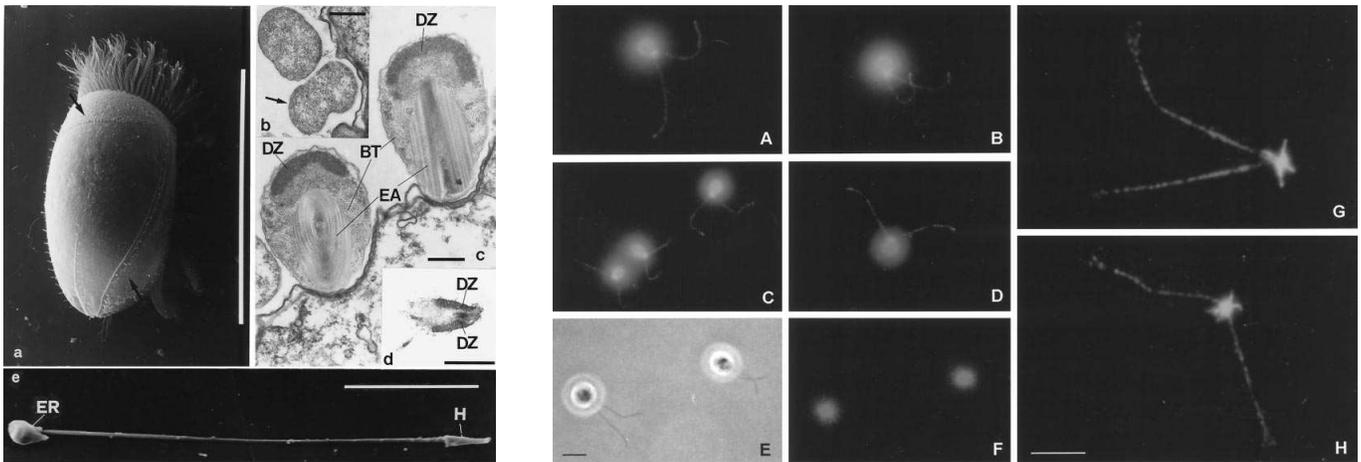


Figure 1: Sample scale-finding problems for SLIF (Murphy, Velliste, & Porreca 2001). Left: a group of panels, each with its own scale bar. Right: Panels A-F share one scale bar, and panels G-H share a second.

sity name U is an affiliation for person name P ”, or “the protein named P is dyed green in image I ”). Some of these relationships are ones of interest to the end user of the extracted information. Many more relations are used by the system itself to make classification and/or extraction decisions, and in general to describe the intermediate states of the extraction process.

- *Changing dependencies among components.* In complex domains, there are many interdependencies among the things involved in an extraction. Thus, in designing new and better classifiers and extractors, it is often useful to use as features the output of other classifiers, candidates proposed by other extractors, or various aggregations and counts based on other extractors and other classifiers.

For instance, in deciding what university-affiliated person P is associated with a particular home page, it might be useful to compute which name P is associated with the largest number of citations made in technical papers found near that home page. Note that computing this feature requires finding technical papers, and extracting and segmenting citations from them, so it imposes an ordering on how the classifiers must be run.

Our experience is that ordering constraints of this sort change substantially over the course of a project, as features are created, evaluated, and (occasionally) discarded, and as new ways of decomposing a problem are explored.

An extended example

Of course, none of the difficulties discussed above are insurmountable, if encountered on a small scale—the actions of a small and relatively static collection of extractors and classifiers can be combined with (for instance) a set of simple shell scripts. As systems become more complex and fluid, however, control and communication between components becomes more important.

To illustrate this, I will present a concrete real-world example of a subtask of a real-world information extraction

system. In current work, I and colleagues are extending a system called SLIF which extracts biologically important properties from certain images in biomedical publications (Murphy, Velliste, & Porreca 2001; Cohen, Wang, & Murphy 2003). Specifically, SLIF analyzes fluorescence microscope images of cells, and extracts features relating to the subcellular localization of proteins depicted in these microscope images. The extended system associates these *localization features* with cell names and protein names from the caption accompanying the image.

To compute biologically meaningful localization features, SLIF needs to know the scale of an image. Usually the scale is indicated by a bar in the image, and a comment in the caption (like “scale bar, 10mm”). One can thus find the scale of an image by locating the *scale bar graphic* in the image and separately finding the *scale bar measurement* (e.g., “10mm”) in the caption. Since knowing the scale is a prerequisite of the later, more complex image processing, this scale-finding task must be performed *before* the localization features are computed.

So far, the control issues can be easily addressed, but complexities arise when we consider other cases. A single figure can contain multiple subimages. These subimages may each have their own scale bar, as on the left of Figure 1; or, scale bars can be shared, as on the right of Figure 1. In this case, we need to split the full image into *panels* (subimages containing a single microscope image); find *all* scale bar graphics in the figure; and determine how scale bar graphics are shared, based on some understanding of the relative position of the panels. All this processing must again occur before localization features are computed.

It is also possible that there are different scale bar measurements associated with different scale bars in the same figure. For instance, the caption for the right-hand image of Figure 1 might read “Bars, A-F, 10 mm; G and H, 5 mm”. To handle this case, we need not only to split the full image into “panels”, but also associated each panel with its *panel label* (here “A”, “B”, . . . , “H”); further, we must associate

the different scale bar measurements from the caption with the appropriate panel labels. This requires using OCR on the individual panels to find their labels, as well as performing some non-trivial text-processing on the captions (Cohen, Wang, & Murphy 2003). Again, this must occur before localization features are computed.

As it turns out, performing OCR to find panel labels is relatively slow. To reduce processing time, one might elect not to run the scale-finding process unless some panel of the figure actually contains an image for which localization features can be computed. It turns out that the first step of finding localization features for a panel is to run a *panel classifier* that determines if a panel is a fluorescence microscope image. This classifier is relatively fast, so one can improve run-time performance if one imposes a new control constraint—that the *panel classifier* be run on each panel *after* partitioning the figure, but *before* running OCR.

On the other hand, it might be that having a scale-bar (or a scale in a certain range) is a good indicator of whether or not an image is a fluorescence microscope image. In that case, one might want to perform some of the less expensive steps of scale-finding (e.g., finding scale bar measurements in the caption) in the process of computing features for the panel classifier. This imposes an additional constraint, which now forces the conceptually separate steps of scale-finding and localization-feature computation to be closely intermixed. Some scale-finding must happen relatively early (before classifying panels); and some of must happen late (after performing OCR and parsing captions.)

For the example, it's hard to say how many of the control "constraints" discussed above need to be enforced. It might be that some of the more complex scale-finding cases are rare, and can be safely ignored; it might be that OCR is not substantially more expensive than the running a panel classifier; or it might be that finding scale bar measurements does not make the panel-classifier more accurate. Only detailed experimentation and benchmarking can determine which control strategies give the best tradeoff in terms of speed and accuracy. However, it is clear from the discussion that in a complex real-world extraction domain, it is important for a system to allow designers to *flexibly explore* different ways of sequencing the various extraction, classification, and feature-computation components.

Proposed Solution

Outline, requirements, and desiderata

In outline, the proposed solution to these difficulties consists of two pieces. The first piece is a *blackboard system* that can perform the key tasks of *creating identifiers* for new "things", storing and retrieving *alternative representations* of things, and storing and retrieving assertions about the *relationships between things* used by the extraction system.

The blackboard system is closely integrated with a *control system*. The *control system* takes as input a *declarative description* of what inputs are needed and what outputs are produced by each lower-level component, together with a list of *goal outputs* that must be produced by the system. The control system then finds and executes a sequence of calls to

the lower-level components that produce the goal outputs.

Reflection suggests a number of important requirements.

The blackboard must be able to store arbitrarily complex subsystem-specific representations of the objects being manipulated: for instance, one should be able to create and/or store multiple representations of a web page easily, as well as storing multiple representations of a web site (say, as a graph of hyperlinks, or as a set of pages). The blackboard must also have clear semantics, be easily implemented, and be accessible from multiple host languages.

It is desirable for the blackboard system to use accepted communication and representation standards, when they are appropriate. Further, since being able to work with many different types of components is important, it is desirable if the blackboard is (at least in part) *file-based*—by which I mean that files constructed by (or for) off-the-shelf software components can be viewed as part of the blackboard itself. In the event that a computation unexpectedly fails, it is also desirable to be able for programmers to easily browse the state of the blackboard. Finally, it is desirable for the blackboard system to be distributed. Information extraction is often computationally expensive, and it is advantageous to be able to distribute a computation over different machines on the same network.

The control system must be efficient, modular, and restartable. By *modular* we mean that all control information can be (easily!) segregated into a single subsystem, where it can be monitored, debugged, and changed. By *restartable* we mean that a failed computation can be restarted from the point at which it failed (up to the level of granularity provided by the lower-level components). Even more importantly, it must be possible to efficiently perform any computation done by a single lower-level component, so that each component can be effectively tuned and tested.

It is also desirable for the control system to be comprehensible to non-logicists, and for it to be feasible to merge declarative and imperative control, for those cases in which declarative control is not appropriate or convenient. For debugging, it should also be possible to determine which component is responsible for adding what blackboard data.

The underlying semantic model

The world modeled by the blackboard consists of three different varieties of entities: *external entities*, *decompositions* of entities, and *views* (or representations) of entities. These entities can also be associated together by *relations*.

An *external entity* corresponds to something that has (or is hypothesized to have) existence in the world being modeled. In the extended version of SLIF, these are objects like papers, captions, sentences, figures, and panels. External entities are not directly represented by the blackboard system, but *names* or identifiers of external entities do exist, and assertions about external entities can be stored.

Every external entity can be associated with any number of *views* or *representations* of itself. A *view* of an entity is a computer-readable description of that entity, at some level of abstraction. For instance, one might have both JPEG and color-histogram views of the same image, or token-sequence

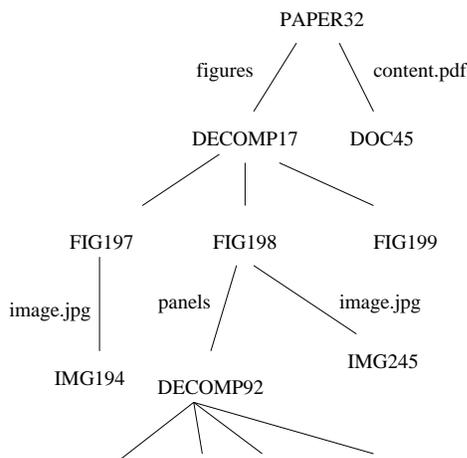


Figure 2: Data from a simple ontology for papers and figures. `image` and `content` are the view names, and `figures` and `panels` are decomposition names. All other strings identify external or internal entities.

and parse-tree views of the same document. Every view has a *label*, which consists of a name and an extension.

An external entity can also be associated with one or more *decompositions* of itself. A *decomposition* describes a particular way in which an external entity can be broken up into parts, and also specifies the resulting parts (which are external entities). The parts need not be an exhaustive partition of the original entity, but they should be of the same type. In figure processing, for instance, a paper is decomposed into figures, a figure is decomposed into panels, and a microscope panel might be decomposed into individual images of cells. A *decomposition* also has a *label*. Decomposition labels allow objects to be split into subparts in different, potentially overlapping ways; for instance, a PDF document might be decomposed into pages and also into sentences.

Views and decompositions are also considered to be entities, but of a different sort than external entities. While external entities correspond to things in the “real” world that the extraction system is trying to model¹, views and decompositions correspond to things inside the blackboard system itself.

The network of external entities, views, and decompositions can be visualized as a forest, as shown in Figure 2. By adding an additional artificial “root” entity to the forest, one can force it to be a tree. In this tree, called the *entity tree*, external entities are parents of associated views and decompositions, and decompositions are parents of their “part” entities.

In information extraction applications, most entities are naturally subparts of something else. The entity tree defines a part-subpart relationship, as follows: entity e_1 is a *subpart* of entity e_2 iff there is e_1 is a descendant of e_2 in the entity tree, and if the path from e_2 to e_1 passes through at least one

¹I use scare quotes around “real” since the world being modeled might well consist only of on-line documents, which are in some sense no more “real” than the blackboard system.

Entity	ExternalEntity	View	Decomp
ExternalEntity	→ ENAME	NamedDecomp	ENAME
View	→ ExternalEntity-VNAME	.EXT	
Decomp	→ NamedDecomp		
NamedDecomp	→ ExternalEntity-DNAME	/	
VNAME	→ [A-Za-z0-9]+	// the name of a view	
EXT	→ [A-Za-z0-9]+	// the MIME type of a view	
ENAME	→ [A-Za-z0-9]+	// the name of an entity	
DNAME	→ [A-Za-z0-9]+	// the name of a decomposition	

Table 1: BNF grammar for the naming scheme for entities.

decomposition entity. Entity e_1 is a *direct subpart* of e_2 iff the path from e_2 to e_1 passes through exactly one decomposition entity.

In addition to the part/subpart relationship, entities are associated together by extensionally defined *relations*. A *relation* is a named set of k -tuples of entities. Following convention, we will use $r(e_1, \dots, e_k)$ for the tuple $\langle e_1, \dots, e_k \rangle$ in the relation named r .

The naming scheme and blackboard

In order to manipulate these entities, they must be identified. A *naming scheme* is a one-to-one function f_{name} mapping entities to strings. It is possible to simply assign unique identifiers to every entity, as in the figure. However, in our system, the range of f_{name} is defined by the BNF grammar of Table 1, and f_{name} is defined compositionally according to the grammar, in the obvious way: for instance, for a view entity e_v with label “image.jpg” and parent e_x , $f_{name}(e_v) = f_{name}(e_x) + \text{“-image.jpg”}$ (here the plus sign denotes string composition).

As further examples (relating to the figure), if $f_{name}(\text{PAPER32}) = \text{“p32”}$ then we might have

$$\begin{aligned}
 f_{name}(\text{DECOMP17}) &= \text{“p32-figures/”} \\
 f_{name}(\text{FIG198}) &= \text{“p32-figures/fig3”} \\
 f_{name}(\text{IMG245}) &= \text{“p32-figures/fig3-image.jpg”} \\
 f_{name}(\text{DECOMP92}) &= \text{“p32-figures/fig3-panels/”}
 \end{aligned}$$

Notice that as new entities are constructed, it is fairly easy to assign them names, especially if they are direct subparts of existing entities. Notice also that the names of views look much like file names, and all entity names look much like URIs (Berners-Lee, Fielding, & Masinter 1998).

The blackboard stores *assertions* about entities: that is, k -tuples of entities, tagged with the name of a relation. The primitive operations on the blackboard are to *store* specific (ground) assertions, and *retrieve* all assertions that match a pattern. A *pattern* is a relation name, plus a k -tuple of strings, which might either be entity names, or the special value “*”, which matches every entity name. No more general notion of unification or logical variable is supported.

The blackboard also controls *access* to views. The primitive operations are reading and writing to a view, or equivalently, mapping views to file pointer/descriptor objects.

Since the names of view entities are so close to file names, this can be trivial; however, making the mapping an explicit process allows other nontrivial mappings to be used.

The blackboard is itself stored in a set of views. Specifically, all assertions $r(e_1, \dots, e_k)$ are stored in a view of entity e'_1 labeled r .FACTS, where e'_1 is the nearest ancestor of e_1 that is an external entity. For instance, an assertion about the dimensions of IMG245 in Figure 2 might be stored in the view “p32-figures/fig3-dimension.FACTS”. Notice that this means that assertions can be retrieved relatively easily if the value of their first argument is known, providing a simple indexing mechanism.

The view-based storage of the blackboard also means that the contents of the blackboard are contained in many separate subunits. It is convenient if the software components that use these subunits can control access to them. To implement this, define a *key* for an assertion $r(e_1, \dots, e_k)$ to be the pair (r, e'_1) from the discussion above. Any component can *claim* any unclaimed key, giving it “ownership” of the corresponding subunit. Once a key has been claimed, the owning component can restrict write access to this key by other components.

By default, the component that first claims a key is the only component with write access, and all other components have read access only. This encourages having one information “producer” and many “consumers” for each key, which simplifies execution planning and concurrency issues. A natural extension would be to restrict read access as well, which could be used for certain types of information hiding (e.g., to limit visibility of part of the blackboard to a certain set of components).

A consequence of this scheme is that components must identify themselves whenever they access the blackboard. This is desirable for other reasons: for instance, one could find redundant blackboard views that are written but never read. Another possible extension is to allow owning components to *close* a key, forbidding any further writes to it. This would allow a reasoning system to make a local closed world assumption (Etzioni, Golden, & Weld 1994).

Properties of the naming scheme

In this section, we will make some more precise claims about the nature of the proposed solution. We begin with some definitions.

Definition 1 (Properties of naming schemes) Let f_{name} be a naming scheme, S be an operating system, P be a unary predicate on entities, and R be a binary function on entities.

- f_{name} is RDF-compatible iff there is a function g_{URI} such that for every view or decomposition entity e , $(f_{name} \circ g_{URI})(e)$ is a valid uniform resource identifier (URI).
- f_{name} is lightweight on S iff there is a function g_{file} such that (a) for every view entity e_v , $(f_{name} \circ g_{file})(e_v)$ is a valid file name and (b) for every decomposition entity e_d , $(f_{name} \circ g_{file})(e_d)$ is a valid directory name in S .
- f_{name} is accountable iff it is lightweight and if $(f_{name} \circ g_{file})$ is efficiently invertible.

- f_{name} is faithful for P (respectively, for R) if there is a unary function on strings $g_P(s)$ (respectively, a binary function $g_R(s_1, s_2)$) such that $g_P(f_{name}(e)) = 1$ iff $P(e)$ is true (respectively, $g_R(f_{name}(e_1), f_{name}(e_2)) = 1$ iff $R(e_1, e_2)$ is true).

The desirability of these properties are clear. If f_{name} is RDF-compatible, then any blackboard assertions can be easily translated into Resource Description Framework (RDF) (Lassila & Swick 1999), which allows a wide variety of operation for transferring, storing, and reasoning with the data. If it is “lightweight”, non-external entities (views and decompositions) can be implemented using the host operating systems’ file system². If it is “accountable”, then files and directories created by the system can be easily mapped back to the entities they implement. If it is “faithful” for a predicate, then the predicate can be tested simply by looking at the entity names, without recourse to the blackboard.

We omit any proof of the following theorem, which is straightforward.

Theorem 1 Let f_{name} be the naming scheme of Figure 1. Let $P_{isView}(e)$ be true iff e is a view entity, $P_{isDecomp}(e)$ be true iff e is a decomposition entity, $R_{part}(e_1, e_2)$ be true iff e_1 is a direct subpart of e_2 , and $R_{part} * (e_1, e_2)$ be true iff e_1 is a subpart of e_2 .

Then f_{name} is RDF-compatible, lightweight on POSIX-compliant systems, accountable, and faithful for P_{isView} , $P_{isDecomp}$, R_{part} , and $R_{part}*$.

Extensions to the naming scheme

Some simple extensions of the naming scheme make it more practical; for instance, we also allow relations to contain “constant entities” like “number://98.6” and “string://false”, as long as (for RDF-compatibility) these entities are not the first argument of a relation.

Another useful extension is the notion of *computed subparts*. If the description of how to compute a subpart is compact, then it may be more convenient to include that description in the name of the subpart entity. As an example, consider the application of labeling substrings of a document as training data for a learned extraction system. One would like to generate assertions about substrings (for instance, “this substring is/isn’t a protein name”) but it is not necessary to actually create any views of the substring. Of course, later processing might require creating views of the substring (for instance, one might wish to save a parsed or tokenized version of a substring) so one would like views of computed entities to still be mappable to file names.

Computed subparts are supported by the following additional grammar rules:

```
ExternalEntity → ComDecomp NUMBER,(NUMBER)*
ComDecomp → ExternalEntity-VNAME.EXT,FNAME/
FNAME → [A-Za-z0-9]+ // a “part-of” function
```

For example, the substring e_s extending from character position 22 to 33 of the caption of FIG199 might be named by the string “p32-figures/fig4-caption.txt,substring/22,33”, and a

²Of course, other implementations are possible.

act:	<i>find scale bar graphic in panel Y of figure X</i>	R1
pre:	X-panels/Y-subimage.jpg	
post:	scaleBarGraphic(Y-subimage.jpg, UL,UR,LL,LR)	
act:	<i>extract scale bar from caption of figure X</i>	R2
pre:	X-caption.txt	
post:	scaleBarMeasure(X, SizeInMM)	
act:	<i>compute features for panel classifier, including the “scaleBarMeasure” feature from caption</i>	R3
pre:	X-panels/Y-subimage.jpg, scaleBarMeasure(X, SizeInNM)	
post:	X-panels/Y-panelClassifierFeatures.data	
act:	<i>run the panel classifier</i>	R4
pre:	X-panels/Y-panelClassifierFeatures.data	
post:	isFluorescenceMicroImage(Y-subimage.jpg)	
act:	<i>assign scale of panel—this is only done for panels that are fluorescence microscope images</i>	R5
pre:	X-panels/Z-subimage.jpg, isFluorescenceMicroImage(Y-subimage.jpg), scaleBarMeasure(X, SizeInMM), scaleBarGraphic(Z-subimage.jpg, UL,UR,LL,LR),	
post:	panelScale(Y-subimage.jpg, SizeInPixels)	

Figure 3: Sample control rules from the figure-processing domain.

tokenized version of e_s might be stored in “p32-figures/fig4-caption.txt,substring/22,33-tokens.txt”

Computed subparts are treated the same as any other external entity, exception in storing assertions: if e_1 is a computed subpart, then the *key* for $r(e_1, \dots)$ is (r, e'_1) where e'_1 is the nearest ancestor of e_1 that is a *non-computed* external entity. For instance, the assertion “protein(e_s)” would be stored in the view “p32-figures/fig4-protein.FACTS”.

The declarative control subsystem

The *control subsystem* consists of a set of STRIPS-like rules called *view builders*, together with a simple *planner*. Each view builder consists *preconditions*, *postconditions*, and an *action function*. The *postconditions* specify views that will exist after the actions are successfully executed, and the *preconditions* specify views that must exist before the actions can be executed. The *action function* is an imperative function in the host language that takes as an argument two lists of entity names, corresponding to the precondition and postcondition views, and creates the postcondition views.

The pattern language for preconditions and postconditions uses a special path-like syntax based on entity names. Excepting views of the root entity, every view name can be written as “*d/s-label.ext*” where “*d*” is a decomposition entity and “*s*”, “*label*”, and “*ext*” are strings; similarly, any decomposition can be written “*d/s-label*”. We call “*s*” here a *subpart stem*, and “*d*” the *subpart container*.

Each *pre-* or *postcondition pattern* is written like an entity name, except that subpart stems can be replaced with variables (uppercase letters), and subpart containers can be omitted. As an example, the following are all valid pre- and post- condition patterns:

1. Y-caption.txt
2. X-figures/Y-image.jpg
3. X-figures/Y-caption.txt

4. X-figures/Y-panels/Z-scaleBarSize.txt

These patterns are *not* matched by binding subpart stems to variables; instead, they are matching by binding the external entity names corresponding to these subpart stems to variables. For instance a pair of entities X, Y match pattern 2 if X has a decomposition-entity child labeled “figures”, Y has a view-entity child labeled “image.jpg”, and Y is a direct subpart of X . Each pattern thus compactly specifies a set of constraints on external entities in terms of the entity tree.

As a very simple example, the following is a legal control rule:

action:	<i>cc -c X-source.c</i>
pre:	X-source.c
post:	X-object.o

The similarity of control rules and rules for the UNIX “make” utility is deliberate: recall that one design goal was comprehensibility to persons without training in planning and logic.

To simplify pattern matching, all variables in the preconditions must occur in the preconditions, except for the special variable “*”, which is implicitly universally quantified.

For convenience, some syntactic shortcuts are allowed in the pattern language. A pattern for a view containing blackboard assertions for relation r can be written “ $r(e_1, e_2, \dots, e_k)$ ”. For instance, the pattern “X-dimension.FACTS” could be written as “dimension(X-image.jpg, Height, Width)” —the extra arguments “Height, Width” are not needed to build the view name, and will be ignored by the planner. In addition to better readability, this convention relieves the user from understanding the details of how keys are computed.

Figure 3 gives some control rules from the extended example of the introduction (with action functions replaced with comments). These rules specify the dependencies involved in a certain scale-finding process. To find the “panelScale” of a panel Y in a figure X , the planner might propose this sequence of actions based on these rules: (1) find all scale bar measurements (e.g. “5 mm”) in the caption for X (2) compute features for the panel classifier, including features based on the stated scale bar size (3) run the panel classifier (4) find the scale bar graphic in the image for Y (5) if Y is a fluorescence microscope image, associate a scale (in mm/pixel) with the panel.

Having the planner find this sequence “on the fly” means that it is not necessary to rewrite control code if ordering constraints change. For instance, if it is indeed necessary to find scale bar graphics that are shared between multiple panels, one might replace rule R1 in Figure 3 with

action:	<i>find scale bar graphics, perhaps in nearby panels</i>
pre:	X-panels/Z-subimage.jpg, X-panels/*-subimage.jpg, panelPosition(X,Z1,UL,UR,LL,LR),
post:	scaleBarGraphic(Z-subimage.jpg, UL,UR,LL,LR)

and the planner will automatically choose a new sequence of actions that satisfies these constraints.

Planning technology of this sort is fairly well understood. Except in very trivial cases, planning systems can not be guaranteed to be simultaneously sound, complete and efficient (Bylander 1994; Erol, Nau, & Subrahmanian 1995). In

our limited experience with this approach, however, planning is not a computational bottleneck: for SLIF, the time spent in sequencing actions of the components is dwarfed by the time needed for image processing and text analysis. This is true even though the current planner is a fairly simple one, based on backward chaining with a fixed depth bound to avoid looping.

Implementation Status

The system described above has been completely implemented (including a number of minor extensions not discussed above) and is used as the sole intra-component communication and control scheme for the extended version of SLIF, a non-trivial information extraction system. (The extended version of SLIF is currently based on about 48,000 lines of code in several languages, including Perl, C, Matlab, and Java.) Of these 48,000 lines of code, 580 non-comment source lines of code define the control strategy.

The control system and blackboard are written in Perl5, and consist of about 850 lines of non-comment source code. About 400 lines of these implement the planner. An additional 550 lines of code implement an HTML-based tool allowing developers to interactively browse or query a blackboard, or monitor progress of an extraction process.

An advantage of the extremely lightweight implementation is that it is feasible to re-implement key components in different host languages; for instance, one could easily re-implement the blackboard in C, Matlab, and Java, thus allowing closer co-ordination of components written in multiple languages. It would also be relatively simple to replace the current file-based implementation, for instance with one that keeps more information in memory for faster access.

A final advantage of the architecture is that, since view and decomposition names can be easily mapped to URI's, it would be straightforward to have the blackboard be distributed over multiple machines; in fact, the entire extraction system could be distributed across the web. This might be useful for an information extraction system developed by a geographically distributed group of researchers.

Concluding Remarks and Related Work

To summarize, I have described and motivated an important piece of infrastructure for complex, real-world information extraction systems. The infrastructure consists of two tightly-coupled subsystems: a RDF-compatible blackboard subsystem, and a declarative control subsystem. This scheme allows extraction, classification, and feature computation components to share information and co-ordinate their actions, even if they use completely different representations for entities—e.g., in the scientific figure processing task used as an extended example of the introduction, coordination is possible between image-processing algorithms and text-processing algorithms that operate on the caption and image associated with a figure, respectively. The scheme is also lightweight and conceptually simple.

A number of architectures for information extraction systems have been proposed in the past. Typically these have

grown out of work in natural language processing, and assume a more sequential control model, based on relatively well-understood steps like tokenization, part-of-speech tagging, etc. The general-purpose architecture most similar to ours is probably GATE (Cunningham *et al.* 2002), which is based on TIPSTER architecture (Grishman 1997). In GATE, intra-component communication is based on document annotations, and control is based around pipeline-like “processing components”. GATE includes tools allowing a user to interactively explore possible pipelines using a precondition/postcondition model of components. Compared to GATE, the architecture proposed here allows for more heterogeneity in representations and more flexibility in control.

Although applying this combination of subsystems to information extraction tasks is (at least to my knowledge) novel, it is acknowledged that few of the subsystems are themselves novel; in fact, most of them are much simpler than have been explored in recent research. As noted above, STRIPS-style planning is very well-understood (Bylander 1994; Erol, Nau, & Subrahmanian 1995), and very sophisticated methods for it exist (e.g., (Weld 1999; Kautz & Selman 1999)); however, simple techniques appear to be adequate for this task. Blackboard architectures for storing and retrieving tuples are also widely used, often in conjunction with distributed models of computation (for a survey, see (Papadopoulos & Arbab 1998)). The blackboard system adopted here is very simple, in part because distributed computation is typically not an issue in information extraction, as the task is highly data-parallel. (However, the “key” mechanism is necessary to ensure that intra-component communications are not be blocked when multiple extractions are run in parallel.)

I am not aware of other “naming systems” which formally and explicitly encode relationships between parts and subparts, or between external entities and internal (views and decompositions) of these entities. However, the idea of a naming system is a natural one, and one often used informally in software systems. The syntax for naming proposed here is heavily influenced by existing W3C standards for path-based query languages like XPath and representation schemes like RDF (Lassila & Swick 1999).

Finally, there is a large body of previous work on control architectures of this general sort in the multi-agent literature (for a survey, see (Stone & Veloso 2000)), but apparently little consensus as to what sorts of architectures are best for what sorts of tasks. To my knowledge, the most similar tasks considered in the multi-agent literature are information gathering tasks (e.g., (Menczer 1997; Srinivasan *et al.* 2002; Chen & Sycara 1998), in which agents conspire to rank and distribute existing documents. From a technical point of view, this is a rather different task—notably, since subparts are not constructed during a computation, information sharing and control issues are much simpler. In the terminology of this community, this paper is a specific proposal for (and experimental validation of) an multi-agent architecture that is appropriate for information extraction tasks based on many learned components.

References

- Berners-Lee, T.; Fielding, R.; and Masinter, L. 1998. Uniform resource identifiers (URI): Generic syntax. Technical Report RFC 2396, IETF. Available from <http://www.ietf.org/rfc/rfc2396.txt>.
- Buckley, C.; Cardie, C.; Mardis, S.; Mitra, M.; Pierce, D.; Wagstaff, K.; ; and Walz, J. 1999. The smart/empire TIPSTER IR system. In *TIPSTER Phase III Proceedings*, 107–121. Morgan Kaufmann.
- Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1-2):165–204.
- Chen, L., and Sycara, K. 1998. WebMate: A personal agent for browsing and searching. In Sycara, K. P., and Wooldridge, M., eds., *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, 132–139. New York: ACM Press.
- Clark, J., and DeRose, S. 1999. XML path language (XPath) version 1.0. Available from <http://www.w3.org/TR/1999/REC-xpath-19991116>.
- Cohen, P. R.; Schrag, R.; Jones, E. K.; Pease, A.; Lin, A.; Starr, B.; Gunning, D.; and Burke, M. 1998. The DARPA high-performance knowledge bases project. *AI Magazine* 19(4):25–49.
- Cohen, W.; McCallum, A.; and Quass, D. 2000. Learning to understand the web. *IEEE Data Engineering Bulletin* 23:17–24.
- Cohen, W. W.; Wang, R.; and Murphy, R. F. 2003. Understanding captions in biomedical publications. Submitted for publication.
- Cohen, W. W. 1998. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of ACM SIGMOD-98*.
- Cohen, W. W. 1999. Reasoning about textual similarity in information access. *Autonomous Agents and Multi-Agent Systems* 65–86.
- Craven, M.; DiPasquo, D.; Freitag, D.; McCallum, A.; Mitchell, T.; Nigam, K.; and Slattery, S. 2000. Learning to construct knowledge bases from the world wide web. *Artificial Intelligence* 118((1-2)):69–113.
- Cunningham, H.; Maynard, D.; Bontcheva, K.; and Tablan, V. 2002. GATE: A framework and graphical development environment for robust nlp tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics (ACL'02)*.
- Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence* 76(1-2):75–88.
- Etzioni, O.; Golden, K.; and Weld, D. 1994. Tractable closed world reasoning with updates. In Doyle, J.; Sandewall, E.; and Torasso, P., eds., *KR'94: Principles of Knowledge Representation and Reasoning*. San Francisco, California: Morgan Kaufmann. 178–189.
- Grishman, R. 1997. Tipster architecture design document version 2.3. Technical report, DARPA. Available from <http://www.itl.nist.gov/div894/894.02>.
- Jensen, L. S., and Cohen, W. W. 2001. Grouping extracted fields. In *Proceedings of the IJCAI-2001 Workshop on Adaptive Text Extraction and Mining*.
- Kautz, H., and Selman, B. 1999. Unifying SAT-based and graph-based planning. In Minker, J., ed., *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*. College Park, Maryland: Computer Science Department, University of Maryland.
- Knoblock, C. A.; Minton, S.; Ambite, J. L.; Ashish, N.; Modi, P. J.; Muslea, I.; Philpot, A. G.; and Tejada, S. 1998. Modeling web sources for information integration. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*.
- Lassila, O., and Swick, R. R. 1999. Resource description framework (RDF) model and syntax specification: W3c recommendation 22,. Available from <http://www.w3.org/TR/1999/REC-rdf-syntax-19990222>.
- McCallum, A.; Nigam, K.; and Ungar, L. H. 2000. Efficient clustering of high-dimensional data sets with application to reference matching. In *Knowledge Discovery and Data Mining*, 169–178.
- Menczer, F. 1997. ARACHNID: Adaptive retrieval agents choosing heuristic neighborhoods for information discovery. In *Machine Learning: Proceedings of the Fourteenth International Conference*, 227–235.
- Murphy, R. F.; Velliste, M.; and Porreca, G. 2001. Searching online journals for fluorescence microscope images depicting protein subcellular location patterns. In *Proceedings of the 2nd IEEE International Symposium on Bio-informatics and Biomedical Engineering (BIBE-2001)*, 119–128.
- Papadopoulos, G. A., and Arbab, F. 1998. Coordination models and languages. In *761*. ISSN 1386-369X: Centrum voor Wiskunde en Informatica (CWI). 55.
- Ray, S., and Craven, M. 2001. Representing sentence structure in hidden markov models for information extraction. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 584–589.
- Srinivasan, P.; Mitchell, J.; Bodenreider, O.; Pant, G.; and Menczer, F. 2002. Web crawling agents for retrieving biomedical information. In *In Proc. Int. Workshop on Agents in Bioinformatics (NETTAB-02)*.
- Stone, P., and Veloso, M. M. 2000. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots* 8(3):345–383.
- Weld, D. S. 1999. Recent advances in AI planning. *AI Magazine* 20(2):93–123.