Orchestration of Semantic Web Services for Large-Scale Document Annotation

Barry Norton, Sam Chapman, and Fabio Ciravegna

Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK
{B.Norton, S.Chapman, F.Ciravegna}@dcs.shef.ac.uk

Abstract. Armadillo is a tool that provides automatic annotation for the Semantic Web using unannotated resources like the existing Web for *information harvesting*, that is: combining a *crawling* mechanism with an extensible architecture for *ontology population*. The latter is achieved via largely unsupervised machine learning, boot-strapped from oracles, such as web-site wrappers. It is backed up by 'evidential reasoning', which allows evidence to be gained from the redundancy in the Web as well as inaccuracies in information, also characteristic of today's Web, to be circumvented. In this paper we sketch how the architecture of Armadillo has now been reinterpreted as workflow templates that compose semantic web services and show how the porting of Armadillo to new domains, and furthermore the application of new tools, has thus been simplified and benefits from semantic discovery and automatic orchestration.

1 Introduction

The vision of the Semantic Web (SW) is centred around sharing knowledge in order to acquire and reuse it [1]. Recently, it has become apparent that it is possible to share more than static knowledge, moving towards sharing operational and active knowledge, i.e. towards Semantic Web Services (*SWSs*). In the future SW it will be possible to compose large distributed systems by composing existing SW Services. One example of large scale services necessitating service reuse are automatic annotation systems, helping harvesting knowledge from existing unannotated documents [6], [11]. Harvesting is guided by an ontology. Ontologies can be overlapping or evolving from common roots, and thereofore - we claim - parts of existing harvesters should be reused also when ontologies are reused. As a simple example, many harvesters will share the need to recognise generic base types like people and organisations, and appropriate services are very likely to be existing on the Web. The more ontologies will be reused, the more the need of reusing harvesters will grow.

We claim also that a SWS-based architecture provides a clear place to separate (and apply) domain-dependent functionality as opposed to domain independent functionality. Also, such an architecture should provide a means to discover already existing implementations of functionalities for reuse.

A. Gómez-Pérez and J. Euzenat (Eds.): ESWC 2005, LNCS 3532, pp. 649–663, 2005.

Tailoring to a domain should therefore be realised by defining or reusing independent SWS's. All of the advantages of a distributed implementation will therefore hold, i.e.:

- the ability to build a new application, coordinating third-party services, without the need for major computing infrastructure;
- conversely, the ability to provide such services while keeping the implementation in-house for management and maintenance;
- the speed-up that can be achieved through parallelism.

This paper focuses on Armadillo [4], a system for the definition of ontology-specific harvesters for large repositories (e.g. the Web), and its organisation in terms of SWS's. Armadillo uses an ontology to define an annotation task. Annotation is defined as extraction and integration of information (harvesting). The system is based on the Information Food Chain metaphor [9], where information processing is defined as an ecosystem where basic search tools (herbivores) retrieve documents (raw matter), while information processors (carnivores) digest them to produce progressively more sophisticated information. The ecosystem in Armadillo is geared towards the production of knowledge and its implementation is based on integration of SWS. Practically, Armadillo annotates by extracting information from different sources (documents or repositories such as databases) and carrying out 'evidential reasoning' to validate the classifications of, and relations between, instances. This evidence is then integrated and the knowledge entered into a repository summarising the integrated knowledge.

The orchestration of SWS's is realised by providing workflows in BPEL [8], which allows us to express workflows where 'partners', *i.e.* workflow actors to be realised by web services, are parametrically typed but unbound to services. We therefore say that we define the Armadillo architecture as workflow templates where *service parameters* must be instantiated, *i.e.* 'filled in', for orchestration to effect a given task. We allow the process to be understood abstractly in terms of the OWL [15] concepts they deal with by describing the parts of this architecture as *semantic* web services, in OWL-S [7]. This furthermore allows *semantic* discovery to be engaged to help with finding the appropriate services to use to instantiate parameters; again we pragmatically decide to adapt the existing web services, *i.e.* UDDI[1], tools to semantic purposes in the spirit of [14]. Furthermore, BPEL allows us to represent graphically, rather than in code, the workflows and insofar as appropriate services can be found, a developer can apply Armadillo to a new annotation task with no coding at all. Even where coding is needed, the subtask needed is abstracted away from the Armadillo logic by service boundaries and the service produced is automatically described in OWL-S terms allowing it to be directly made available for future reuse.

In this paper we concentrate on the implementation via SWSs of the extraction and integration tasks, which are at the heart of the architecture. This paper is organised as follows: Section 2 describes the abstract harvesting strategy, Section 3 gives more details on the architecture and each subtask in detail, finally we make conclusions and sketch future work in Section 5.

[1] http://uddi.org/

2 Harvesting Strategy

Armadillo [4] annotates documents by harvesting knowledge from large repositories, i.e. by extracting information from different sources and finally integrating the retrieved knowledge into a repository. The repository can be used both to access the extracted information and to annotate the source(s) where the information was identified. Furthermore the information source(s) can be analysed to verify the correctness and the provenance of the information.

Unsupervised learning begins from seed data provided by a largely 'infallible oracle' (e.g. a list of relevant terms). Seed data are utilised by searching in the document repository for matching strings. If found, matching strings must be confirmed using some disambiguation or contextual strategies (e.g. local disambiguation as in SemTag [6] or multiple evidence for corroboration). Further annotations are identified by the process sketched in Figure 1, *e.g.* by adaptively learning from the context in which known entities were found. All new annotations must be confirmed, by the subtask called 'evidential reasoning', and the terms encapsulated by the annotations can be used to seed learning again. Finally the discovered knowledge is integrated (e.g. some of the new entities / facts are merged) and stored into a format for future use, typically a database.

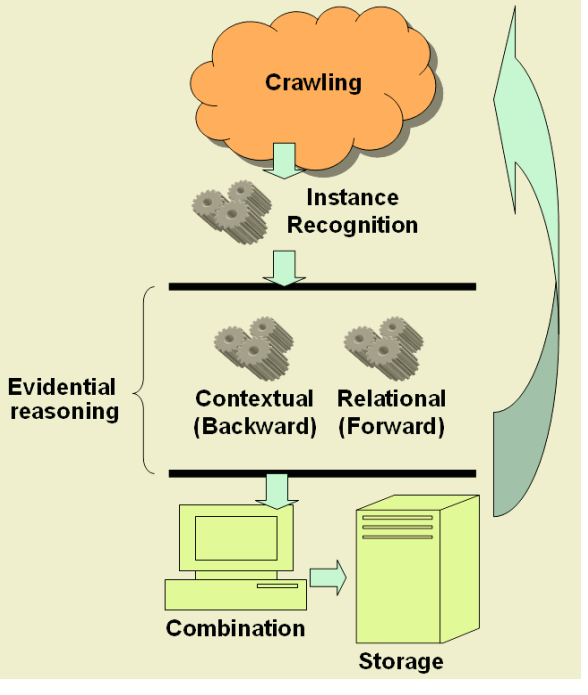

Fig. 1. Harvesting Strategy

3 Architecture

Armadillo employs the following techniques/technologies:

- *Crawling* to explore the repository in a efficient and exhaustive way.
- *Adaptive Information Extraction* from texts (IE): used for spotting information and to further learning new entities.
- Information Integration (II): used to (1) discover an initial set of annotations to be used to seed learning for IE and (2) to confirm the newly acquired (extracted) information, *e.g.* using multiple evidence from different sources and (3) to integrate knowledge *e.g.*, by merging entities.
- *Semantic Web Services*: the architecture is based on the concept of "services". Each service is associated to some part of the ontology (*e.g.* a set of concepts and/or relations) and works in an independent way. Each service can use other services (including external ones) for performing some subtasks. This paper details the organisation of the SWS architecture and their orchestration.
- *RDF Repository*: where the extracted information is stored and the link with the pages is maintained.

The act of porting Armadillo to a new ontology population task begins by providing a domain-specific ontology. Population is performed by starting focusing on one concept and looking for its occurrences for it or for occurrences of its instances. These concepts tend to be characterised by unique identifiers such as proper names or rather unique descriptions[2] When key concepts have been identified, other entities with a minor degree of uniqueness can be identified by exploiting the context given by relations with already identified entities. For example dates tend to be too generic for unique identification in isolation. If "26 October 2004" is looked for as the date of e.g. a seminar, there is no way that all relevant occurrences of that date can be distinguished by other occurrences of the same date in other contexts. The context is used by the seminar instances previously identified to separate the relevant occurrences of the date from the irrelevant ones. Previously identified entities and their relations with respect to the looked for entity are used as a context. This means that the plan for ontology population is implemented as a directed search on the ontology graph: base (unique) entities are identified and the relations in the ontology are assigned a direction. The plan details the order in which the concepts and relations will be explored and therefore the ontology populated.

For each relation to be followed, we identify a concept A for which we will already have found instances and a concept B that we wish to populate by following the relation. We might also identify concepts C which 'belong' transitively, *i.e.* via further relations, to B so that related instances of such may also provide evidence for the instance of B discovered.

[2] Of course names can be ambiguous and descriptions not completely unique; what we mean here is that the concept looked for must have some degree of simplicity in identification).

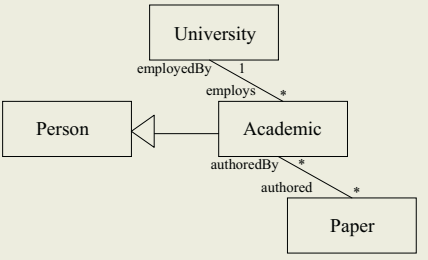

Fig. 2. Example Ontology

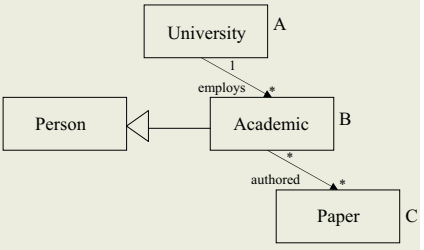

Fig. 3. Example with Ordering

Figures 2 and 3 illustrate the process in terms of a fragment of the example of populating an ontology for the UK academic domain used in [4] and [10]. We will use that example in the remainder of the paper. We first identify the 'University' concept as being one from which we can hope to obtain reliable instances from an oracle such as a list of universities and departments. Such list can be obtained for example by wrapping the Yahoo taxonomy or the RAE web pages[3]; university names tend to be unique terms.

We then decide that we should like to populate the 'Academic' concept, via the 'employs/employedBy' relationship. This concept is less unique than the university name (e.g. you can have many different people called "John Smith") For this task, then, 'University' becomes concept A and we take the recognisable super-concept to 'Academic', 'Person' as concept B. We cannot choose directly the concept Academic because in a university there are many people that are not academic but that still are working for the university. We therefore need to discriminate a second time which recognised people are academics and which are not. Therefore we choose 'Paper' as an appropriate concept able to help discriminating (concept C) and exploit the relation 'authored/authoredBy' as a discriminator (in this small example the definition of academic is a person who writes papers). Papers titles tend to be largely unique. Figure 3 shows how we have added this information to the ontology fragment.

In the rest of the section we describe the details of the recognition of concept B using the context of A. As shown in Figure 1 it involves the following subtasks:

- Crawling;
- Instance Recognition;
- Evidential Reasoning;
- Combination and Storage.

Crawling will systematically retrieve documents associated with an instance of concept A. Instance Recognition will find candidate instances of B in the context of that original instance. Evidential Reasoning will (1) find support for the

[3] http://www.hero.ac.uk/rae/

classification of the concept as B and (2) confirm the relation between A and B by looking for evidence, both within the document and outside, of the relation between the two entities. Finally, if the evidence support the initial hypothesis, the annotation is stored.

In the organisation of an SWS based architecture, A, B and C are variables ranging over concepts, as discussed above, as are Doc, $DocId$ to be tied to the documents to be analysed and $Evidence$, explained later. Moreover, additional *service parameters* must be similarly bound to create an executable process (dotted boxes in the figures below); their signatures may involve the concept variables so it is necessary to decide on this first. Having done so, the choice of services can be aided by *semantic discovery* [14]. We shall describe the choices in the remainder of the paper in terms of the following classes of service:

- *generic* services: wholly independent of domain and context;
- *context-dependent* services: where reuse depends on the context but not the domain (*e.g.* the type of documents or repository used);
- *domain-tailored* services: parameterised to be tailored to a given domain;
- *domain-trained* services: encapsulate machine-learning which can adapt semi-automatically to a given domain;
- *domain-specific* services: encapsulate techniques which are hard-coded for a given domain (but might still be re-used across applications in that domain).

We now go through each abstract subtask of the population task in the terms that these were set out in Figure 1.

3.1 Crawling

The general form of the 'crawling' task as a workflow template over its core services[4] and service parameters is shown in Figure 4:

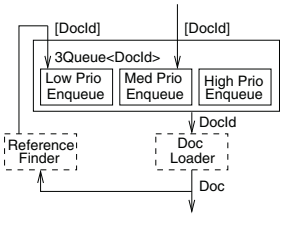

Fig. 4. *Crawling* Task

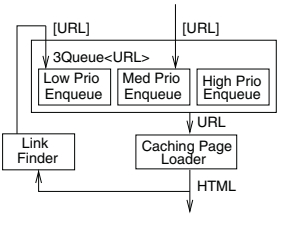

Fig. 5. Example *Crawling* Instantiation

[4] The containing box $3Queue$ means that the service, actually an instantiation in terms of the type of document identifier to be queued, consists of several operations. We omit the trigger (de-queueing) operation since it is not relevant at this level.

To reiterate, there are two levels at which this workflow is parameterised[5]: the $DocId$ and Doc labels [6] are type variables that must be instantiated at concrete types; the $DocLoader$ and $ReferenceFinder$ tasks are service variables to be instantiated with concrete services.

In these terms then, the crawling task becomes one of queueing up document references, in any form by which they can be given persistent identifiers, which are individually loaded and immediately inspected for references to related documents and also passed forward to the *instance recognition* task.

The reason that the original document list is enqueued at a medium priority level is that we will inspect first documents strongly related to the current one; as shown in Figure 14 these are fed back at a high priority level. Once a given document and strongly-related discoveries are exhausted we move on to the next from the original list. Only once this process has been completed for each member of the list do we move on to those only weakly related (there being only tentative reasons to presume those referred to in the document will relate strongly to the original instance of concept A). Note that the signature of the 'Reference Finder' service parameter (in functional terms $Doc -> [DocId]$) allows the possibility that some prioritisation may take place in the list returned, allowing for intelligent analysis.

Figure 5 shows how we could achieve a concrete instantiation of this subtask. First we choose instances for the type variables consistent with Web-oriented technology, i.e., in this example, $Doc = HTML$ and $DocId = URL$. We then choose *context-dependent* services that meet the resulting signatures, i.e. loading a page from its URL, with a 'caching page loader', and respectively finding list of URLs from a page with a 'link finder', itself an instantiation of a domain-tailored regular expression matching service but we do not show this decomposition.

3.2 Instance Recognition

As shown in Figure 6, the 'instance recognition' task is achieved foremost by the concurrent execution[7] of some number[8] of different '*B-Recogniser*'s, *i.e.* services each of which can extract from a document parameter potential occurrences of the concept B. We assume that the document parameter leading to the 'split' operation is broadcast to each service used in the instantiation of this template. Thereafter each of the lists of *candidate instances* returned by these services is

[5] Actually three since angle brackets, for instance $3Queue\langle DocId\rangle$, mean instantiation of some generic (parameterised) service, so that $3Queue$ represents some generic queue service that stores instances of some type notated $DocId$, at three different priority levels, and supplies them to the consequent workflow one at a time.

[6] We use square brackets to represent lists over the contained type, and so $[DocId]$ is the type of a list of elements of type $DocId$, and regular brackets to represent tuples.

[7] We represent concurrent execution, and re-synchronisation on completion, by the solid bars, 'split' and 'join' respectively, in the diagram in the style of UML Activity Diagrams [2].

[8] We represent multiple instantiation also in UML style with a multiple outline.

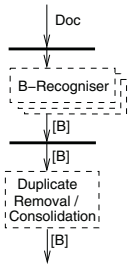

Fig. 6. *Instance Recognition* Task

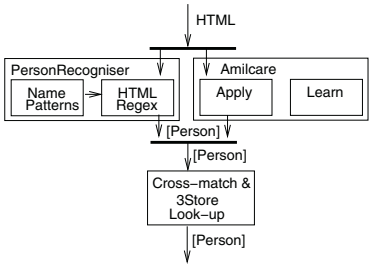

Fig. 7. Example *Recognition* Instantiation

concatenated to make the final list. This is then passed to a service parameter whose instance should be capable of removing internal duplicates, if multiple recognition strategies are employed, as well as carrying out a 'pre-integration' step. This latter step may, and usually will, involve checking in a repository to see whether the instance is already known. For simplicity we show a signature for this service that just involves some refinement of the list, $[B] -> [B]$. In fact we allow additional triples, based on the instance, to be introduced at this point so that, for instance where existing coincident repository instances are found that were introduced from some other source, we may continue to investigate them (rather than dropping them) but introducing a 'sameAs' relationship.

The role of B-Recogniser can be realised by both *domain-tailored*, and *domain-trained* services. This is illustrated in Figure 7 we see that both *domain-tailored* regular expression matching and an instance of the *domain-trained* IE system Amilcare [5] will be used side-by-side, Amilcare learning from the successfully validated instances produced by the former. Since the signature of this service parameter is so general, with the implementation details encapsulated behind the service boundaries, other Information Extraction tools can easily be employed.

The subsequent consolidation stage is typically domain-specific Information Integration but reuse can be made; for instance from the 'similarity metrics' library, *SimMetrics*, which we are developing as an open source project for string metrics and similarity-based integration, and recently released [3].

3.3 Evidential Reasoning

Having identified some consolidated list of candidate instances we then queued these to be validated both for proper classification and to verify and classify the implicit relation through which they were discovered. We call this task 'evidential reasoning' and its overall workflow is presented in Figure 8.

The candidate instances from the instance recognition task are first queued to be examined one by one. For each, some number of concurrent instances of reasoning tasks will be executed. Each will fit one of two general strategies described as follow.

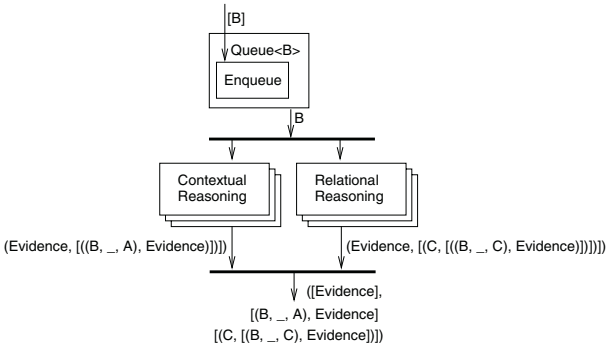

Fig. 8. *Evidential Reasoning* Task

Contextual Reasoning. considers each potential B instance in the context of the A instance via which it was discovered and attempts to be more specific about the relation between them than the implicit relation that instance recognition achieves. In the process more evidence for the classification of the instance being investigated may be found. As seen in Figure 9, two services will be used to find occurrences of the B instance in general, and co-located with the A instance, respectively. A third service then produces a list of potential triples relating these instances, with evidence supporting the relation.

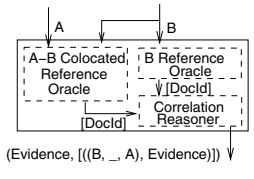

Fig. 9. *Contextual Reasoning* Task

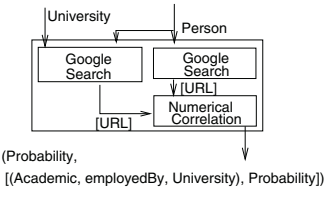

Fig. 10. Example *Contextual* Instantiation

Figure 10 shows a simple instance of this strategy where we 'promote' the candidate instance to being an academic employed by the university based on co-located references on the web, obtained by a Google wrapper which is domain-independent, where a simple probability is based on correlation.

Relational Reasoning. provides evidence for the candidate B instance being correctly classified as such, based on other relations an oracle may find. As such, for each instantiation within this class, two service parameters must be provided: as shown in Figure 11, one finds related new potential instances, another encodes this alongside some kind of evidence.

In this subtask, as shown in Figure 12, we may apply *domain-tailored* or *domain-specific* technologies such as gazetteers — we use the example of a gazetteer of people's forenames — and site wrappers — we use the example

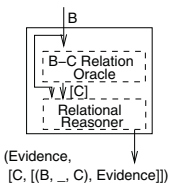

Fig. 11. *Relational Reasoning* Task

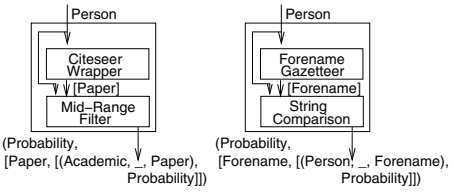

Fig. 12. Example *Relational* Instantiations

of the DBLP portal[9] wrapped as a semantic web service and providing candidate instances for the 'Paper' concept and the 'authored' relation. Such results will be cached and bootstrap a separate workflow on this relation.

As part of the relational reasoning subtask, We may also apply *domain-trained* relation extraction, as we are developing in the tool *T-Rex*. As in the instance recognition subtask we will bootstrap learning based on oracles, for instance in this case on the DBLP results.

3.4 Combination and Storage

The essence of the combination task is to remove duplicates from the candidate instance and its candidate relations and to combine the evidence for these. This logic is encapsulated and instantiates the 'Combination Logic' service parameter and is domain-specific, but may decompose into some reused generic services; in particular statistical functions.

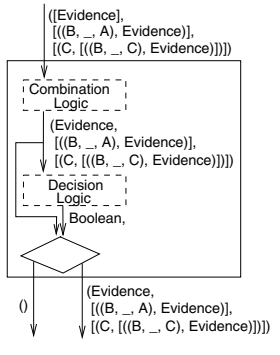

Fig. 13. *Combination* Task

On a basic level we must reduce: a list of evidence for the correct identification of an instance, $[Evidence]$, into a single combined value; all identified $(B, \_, A)$ triples, *i.e.* relations back to the original contextual instance, and the separate *Evidence* values that have been found for each relation, into a non-repeating list;

[9] http://dblp.uni-trier.de/

all identified $(B, \_, C)$ triples, *i.e.* different relations to different new candidate instances C, and the separate *Evidence* values that have been found for each, into another non-repeating list.

The 'Decision Logic' task then decides which of these candidates has sufficient evidence for storage. The diamond style of the final component service in the combination task implies non-determinism: either an untyped value[10] is returned (in case the evidence is insufficient) or the supported classification of the candidate and its relations are returned. Since the 'Repository' service parameter, implementing the storage task as seen in Figure 14, also returns an untyped value, the two possible threads of control are merged to pass such a value back to the 'Trigger' operation of the 'B-Queue' service.

4 Orchestration

The complete workflow template that orchestrates the tasks discussed in the previous section is shown in detail in Figure 14. As stated above, the orchestration is currently realised by encoding the workflow in BPEL [8]. As well as the pragmatic advantages put forward in [12], we state specifically the advantages of having an editor in the Eclipse environment[11] [13], where integration is possible with other plug-ins and tools described later, and having debugging and process monitoring tools[12].

Deficiencies, however, in the BPEL approach mean that it is not possible to directly encode the workflow shown in the simple dataflow manner represented in Figure 14. In particular there are two features that are incompatible with the 'Flow' construct: the looping behaviour shown on the far right hand side, and the non-determinism associated with choices we have illustrated via the diamond boxes. To explain what this is intended to represent, we pick up explaining the thread of control as it finished at the end of Section 3.4. Whichever service, *i.e.* instantiating 'Combination' or 'Repository', returns a value to the 'Trigger' operation of the 'Queue$\langle B \rangle$' service, there are two possibilities, decided between only at this point in the execution: if there are remaining candidate B-instances to be considered then the evidential reasoning task is re-entered; if not another empty value triggers the document queue until this is exhausted and the flow is complete.

Since we can neither 'cross' the lines of a choice process, and furthermore are explicitly are disallowed from having loops in a dataflow, we are forced into implementing subtasks discussed in Section 3 as different services and then mixing the imperative, explicit looping, and dataflow styles, as is typical of BPEL. Similarly we are forced into imperative style in order to mediate between tasks in the workflow, both in the one-to-one fashion normally discussed, and

[10] The *empty type* is represented by () as in languages like Haskell.

[11] http://www.eclipse.org

[12] Like those provided by the Collaxa tool, now part of Oracle Application Server: http://www.oracle.com/technology/products/ias/bpel/index.html

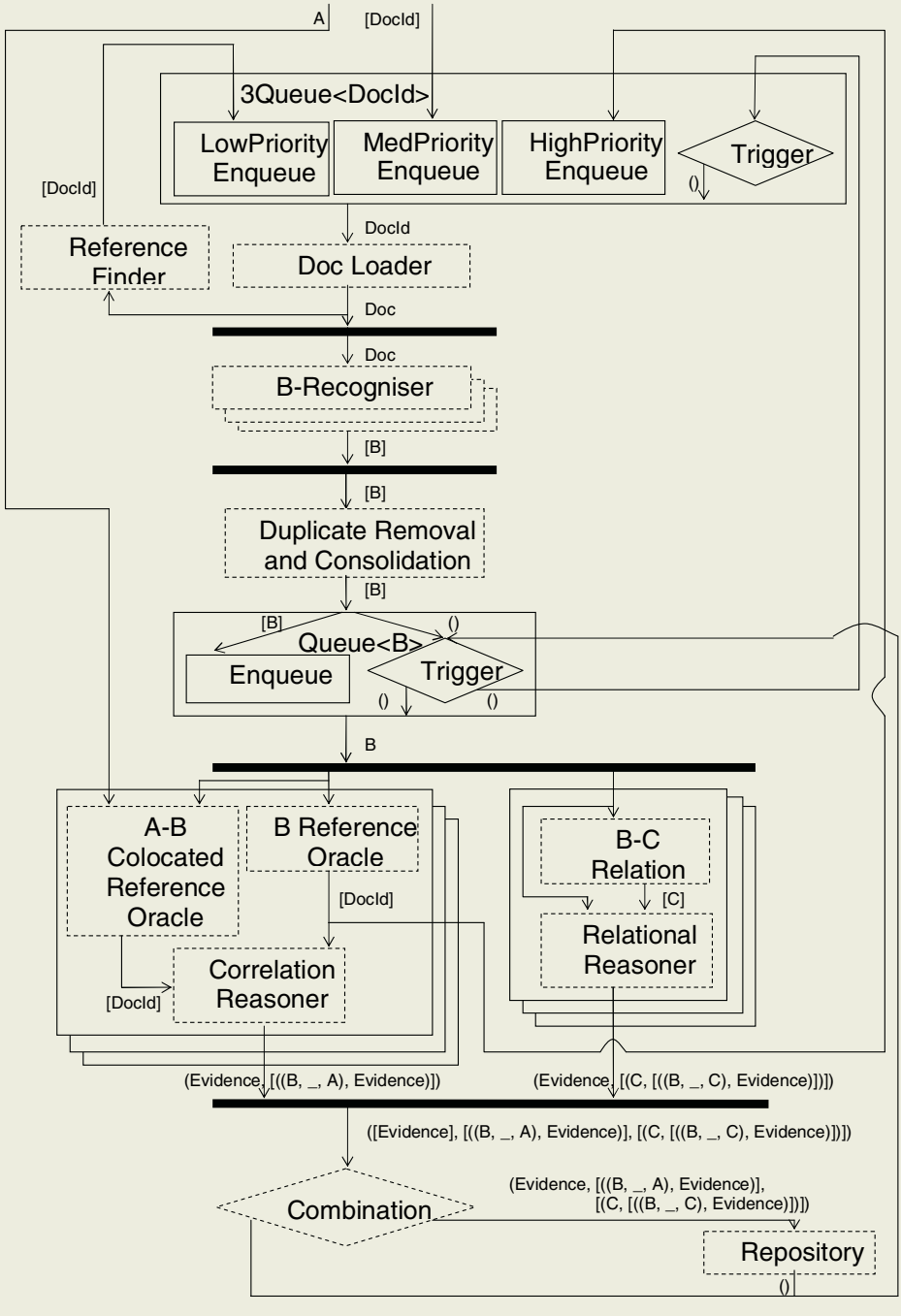

Fig. 14. Architecture in Detail

the many-to-one fashion associated with the 'join' operators that resynchronise concurrent activities.

We note that OWL-S [7] already allows, in the abstract, the binding of different languages to all these points in the workflows described therein. Furthermore more algebraic mixing of operators is specified as being provided. Since few implementations are currently available of OWL-S, however, our decision has been to use the Armadillo architecture as a motivation for, and a test of, a new OWL-S implementation where the language used at these points will be more declarative. For example this should allow a polymorphic functional-style 'list concatenation' function to be bound to the first join operator and thus not have to be tailored towards new instantiations for the parameter 'B'. Furthermore we could allow the user more choice at this stage, for instance allowing a polymorphic 'list cons' function to be bound so that a different consolidation service, which needs to know which candidates came from which B-Recogniser could receive $[[B]]$.

5 Conclusions

We have described the way SWS's are orchestrated to harvest information from the Web, and other corpora, to provide annotations for the Semantic Web. The architecture presented is based on workflow and follows an IE-oriented strategy. Initial approximations to both classification and implicit relation extraction are followed by evidential reasoning based on both context and further relations. In this way a wide variety of semantic web services may be accommodated and porting is eased since, in many cases, users can avoid coding altogether, merely using the workflow templates to guide semantic discovery and composition.

The architecture is currently implemented in BPEL in terms of services grounded in SOAP, but described both syntactically in WSDL and semantically in OWL-S; the generation of both is bootstrapped by an automatic translation. Our development process is wholly Eclipse-based with JDT support for Java coding alongside SWeDE[13] support for OWL(-S) editing, Oracle support for BPEL editing and Ant[14]-hosted tasks for translation, building and deployment.

In future we would like to integrate more 'semantic' forms of discovery to help the process of constructing a concrete workflow, reusing from tools such as [14] as these become available. In the longer term we have hopes for the use of a full OWL-S or WSMX based orchestration solution and our own work towards the former is being carried out as an open source project[15], as well as the accompanying editor[16].

[13] http://owl-eclipse.projects.semwebcentral.org/
[14] http://ant.apache.org/
[15] http://savannah.nongnu.org/projects/CaSheW-s-Engine
[16] http://savannah.nongnu.org/projects/CaSheW-s-Editor

Using a SWS-based architecture for Armadillo provides many benefits associated with service-oriented architectures, such as speed-up from concurrency and distribution, an automatic means for reuse of any code created during an instantiation, and the ability to provide services remotely to users with little infrastructure. A number of instantiations of this architecture have already been carried out and are currently being evaluated, in particular concentrating on efficiency and scalability compared to the existing Armadillo software.

Acknowledgements

This work was carried out within the AKT project (http://www.aktors.org), sponsored by the UK Engineering and Physical Sciences Research Council (grant GR/N15764/01), and the Dot.Kom project, sponsored by the EU IST asp part of Framework V (grant IST-2001-34038).

References

1. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, o284(5):35–43, 2001.
2. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman, 1999.
3. Sam Chapman. SimMetrics. http://sourceforge.net/projects/simmetrics/.
4. Fabio Ciravegna, Sam Chapman, Alexiei Dingli, and Yorick Wilks. Learning to harvest information for the semantic web. In *ESWS*, pages 312–326, 2004.
5. Fabio Ciravegna and Yorick Wilks. *Annotation for the Semantic Web*. Series Frontiers in Artificial Intelligence and Applications. IOS Press, 2003.
6. Stephen Dill, Nadav Eiron, David Gibson, Daniel Gruhl, R. Guha, Anant Jhingran, Tapas Kanungo, Sridhar Rajagopalan, Andrew Tomkins, John A. Tomlin, and Jason Y. Zien. Semtag and seeker: bootstrapping the semantic web via automated semantic annotation. In *WWW '03: Proceedings of the twelfth international conference on World Wide Web*, pages 178–186. ACM Press, 2003.
7. Anupriya Ankolekar et al. DAML-S: Web service description for the semantic web. In *Proc. 1st International Semantic Web Conference (ISWC)*, 2002.
8. IBM et al. Business process execution language for web services version 1.1. http://www-128.ibm.com/developerworks/library/ws-bpel, 2003.
9. Oren Etzioni. Moving up the information food chain: Deploying softbots on the world wide web. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1322–1326, Menlo Park, 4–8 1996. AAAI Press / MIT Press.
10. Hugh Glaser, Harith Alani, Les Carr, Sam Chapman, Fabio Ciravegna, Alexiei Dingli, Nicholas Gibbins, Stephen Harris, Monica M. C. Schraefel, and Nigel Shadbolt. CS AKTiveSpace: Building a semantic web application. In *Proc. 1st European Semantic Web Symposium*, pages 417–432, 2004.
11. P. Kogut and W. Holmes. AeroDAML: Applying information extraction to generate DAML annotations from web pages. In *First International Conference on Knowledge Capture (K-CAP 2001).*, 2001.

12. David J. Mandell and Sheila A. McIlraith. Adapting BPEL4WS for the Semantic Web: The bottom-up approach to web service interoperation. In *Proc. 2nd Intl. Semantic Web Conference (ISWC2003)*, 2003.
13. Barry Norton. Eclipse as a development platform for semantic web services. Eclipse Technology Exchange (eTX04), 18th European Conference on Object-Oriented Programming (ECOOP-2004), 2004. http://www.dcs.shef.ac.uk/∼barry/CASheW-s/Norton04.pdf.
14. Naveen Srinivasan, Massimo Paolucci, and Katia Sycara. Adding OWL-S to UDDI: implementation and throughput. In *Proc. 1st Intl. Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, pages 6–9, 2004.
15. W3C. OWL web ontology language overview. http://www.w3c.org/TR/owlfeatures, 2004.