On Developing a Distributed CBR Framework through Semantic Web Services *

Belén Díaz-Agudo, Pedro A. González-Calero,
Pedro P. Gómez-Martín, Marco A. Gómez-Martín

Dep. Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
email: {belend, pedro, pedrop, marcoa}@sip.ucm.es

Abstract. jCOLIBRI is an object-oriented framework in Java that promotes software reuse for building CBR systems, integrating the application of well proven Software Engineering techniques with a knowledge level description that separates the problem solving methods, that define the reasoning process, from the domain model. In this paper we envision the evolution of this framework into an open distributed framework where contributions to the framework are published, searched and integrated through Semantic Web Services.

1 Introduction

Case-Based Reasoning (CBR) is one of most successful applied AI technologies of recent years. Commercial and industrial applications can be developed rapidly and existing corporate databases can be used as knowledge sources. CBR is based on the intuition that new problems are often similar to previously encountered problems, and therefore, that past solutions may be reused (directly or through adaptation) in the current situation. CBR systems typically apply retrieval and matching algorithms to a case base of past problem-solution pairs.

Developing a CBR system is a complex task where many decisions have to be taken. The system designer has to choose how the cases will be represented, the case organization structure, which methods will solve the CBR tasks and which knowledge, besides cases, will be used by these methods. This process would greatly benefit from the reuse of previously developed CBR systems.

Software reuse is a goal that the Software Engineering community has pursued from its very beginning. A number of technologies have appeared that directly or indirectly promotes software reuse. Unfortunately AI systems have remained for too long in the prototype arena and, in general, AI researchers do not worry too much about software engineering concerns. The most significant and long term effort within the AI community to attain effective software reuse is the KADS methodology [12] and its descendants. The KADS approach for building knowledge based systems proposes the reuse of abstract models consisting

* Supported by the Spanish Committee of Science & Technology (TIC2002-01961)

of reusable components, containing artificial Problem Solving Methods(PSMs), and ontologies of domain models.

During the last few years we have developed jCOLIBRI[1], a framework for developing CBR systems [6–8, 4]. jCOLIBRI promotes software reuse for building CBR systems, and tries to integrate the best of both worlds: the application of well proven Software Engineering techniques with the KADS key idea of separating the reasoning process (using PSMs) from the domain model.

In this paper we envision the evolution of this framework into an open distributed framework where contributions to the framework are published, searched and integrated through semantic web services; using component technologies the PSMs that were thought as internal methods of the framework become external components. Section 2 describes the main ideas lying behind jCOLIBRI and its current architecture pointing out some limitations we have encountered. We propose a solution to these problems based on Semantic Web Services in Section 3. Finally, Section 4 concludes.

2 jCOLIBRI

At the *knowledge level* jCOLIBRI is built around a task/method ontology that guides the framework design, determines possible extensions and supports the framework instantiation process. Task and methods are described in terms of domain-independent CBR terminology.

Every CBR system makes use of CBR terminology, the type of entities that the CBR processes manage. A CBR ontology elaborates and organizes the terminology found in, ideally, any CBR system to provide a domain independent basis for new CBR systems. On this way, CBROnto [8] elaborates an extensive ontology over CBR terminology, the idea beyond this ontology is to have a common language to define the elements that compose a CBR system and to be able to build generic CBR methods independent of the knowledge domain.

Within a knowledge level description, PSMs capture and describe problem-solving behavior in an implementation and domain-independent manner. PSMs are used to accomplish tasks by applying domain knowledge. Although various authors have applied knowledge level analysis to CBR systems, the most relevant work is the CBR task structure developed in [2]. At the highest level of generality, they describe the general CBR cycle in terms of four tasks (4 Rs): *Retrieve* the most similar case/s, *Reuse* its/their knowledge to solve the problem, *Revise* the proposed solution and *Retain* the experience. Each one of the four CBR tasks involves a number of more specific sub-tasks. There are methods to solve tasks either by decomposing a task in subtasks or by solving it directly.

Figure 1 depicts the task decomposition structure we use in our framework. The task structure indexes a number of alternative methods for solving each task, and each one of the methods sets up different subtasks in its turn. This kind of task-method-subtask analysis is carried on to a level of detail where

[1] jcolibri-cbr.sourceforge.net

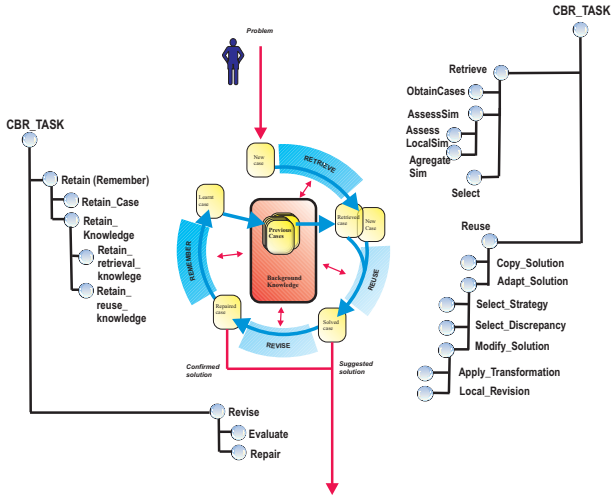

Fig. 1. CBR execution cycle [2] and CBROnto Task Structure

the tasks are primitive with respect to the available knowledge (i.e. there are resolution methods). Besides this task structure, jCOLIBRI includes a library of PSMs to solve these tasks. It describes CBR PSMs by relating them within CBROnto concepts representing the tasks and domain characteristics. PSMs in our library are organized around the tasks they resolve. We also need representing the method knowledge requirements (preconditions), and the input and output "types". These characteristics are described by using vocabulary (i.e. concepts) from the CBROnto ontology.

2.1 Framework Architecture

The jCOLIBRI framework is organized around the following elements and integrated through the architecture of Figure 2:

Tasks and Methods XML files describe the tasks supported by the framework along with the methods for solving those tasks.

Problem solving methods The actual code that supports the methods included in the framework.

Case Base Different connectors (XML, JDBC, RACER, ...) are defined to support several types of case persistency, from the file system to a data base [4].

Cases The framework includes a number of interfaces and classes to provide an abstract representation of cases.

Tasks are a key element of the system since they drive the CBR process execution and represent the method goals. Tasks can be added to the framework at any time, although including a new task is useless unless an associated method exists.

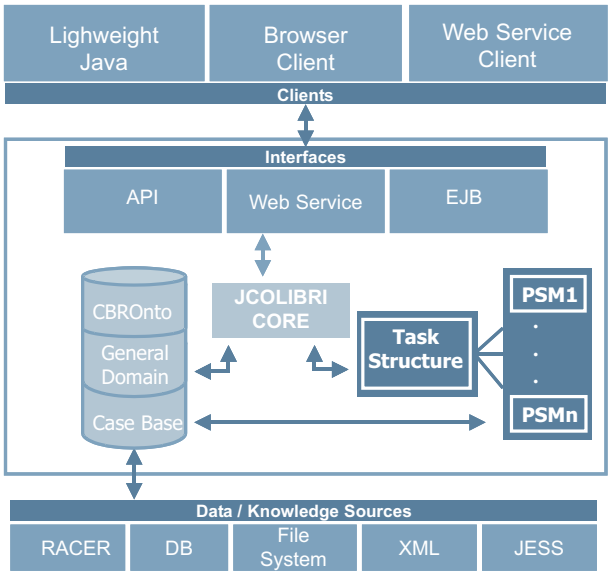

Fig. 2. jCOLIBRI architecture

Regarding methods, most approaches consider that a PSM consists of three related parts. The *competence* is a declarative description of *what* can be achieved. The *operational specification* describes the reasoning process. The *requirements* describe the knowledge needed by the PSM to achieve its competence [9].

Some approaches like CommonKADS [10] specify much of how the PSM achieves its goals, i.e. the reasoning steps, the data flows between them, and the control that guides their execution. As we focus on PSM applicability assessment we consider what the method does, i.e. the task it solves, and its knowledge requirements, and leave control-flow issues to informal documentation and method implementation code. This allow us to use a black box type of method reuse.

Our approach to the specification of PSMs competence and requirements makes use of ontologies and provides two main advantages. First, it allows formal specifications that add a precise meaning and enables reasoning support. Second, it provides us with important benefits regarding reuse because task and method ontologies can be shared by different systems.

Method descriptions follow an XML schema. This elaborated description has a counter-part description in a Description Logic(DL) syntax (we use OWL with RACER as the inference engine). It includes the following elements:

Name The fully qualified name of the class that implements the method. This class must implements the CBRMethod interface.

Description A textual description of the method.

ContextInputPrecondition A formal description of the applicability requirements for the method, including input requirements.

Type jCOLIBRI manages two types of methods: execution (or resolution) and decomposition. Execution methods solve the task, for which has been assigned to, while decomposition ones divide the task into other tasks.

Parameters Method configuration parameters (Inputs and outputs). These parameters are the variable hooks of the method implementation. For example, a retrieval method may be parameterized with the similarity function to apply. They are described by concepts that belong to the CBROnto ontology.

Competencies The list of tasks this method is able to solve.

Subtasks In decomposition methods this element provides the list of tasks that result from dividing the original task.

ContextOutputPostcondition Output data information obtained from this method execution. The information will be used to check which method can take as input the output of this one.

Building a CBR system consists on the instantiation of the jCOLIBRI framework. It is a configuration process where the system developer selects the tasks the system must fulfill and for every task assigns the method that will do the job. The execution of the resulting CBR system can be seen as a sequence of method applications where a method takes as input the output of the previous one. jCOLIBRI provides an user interface that allows the developer to choose the methods to be applied to perform every task.

Ideally, the system designer would find every task and method needed for the system at hand, so that she would program just the representation for cases. However, in a more realistic situation a number of new methods may be needed and, less probably, some new task. Since jCOLIBRI is designed as an extensible framework, new elements will smoothly integrate with the available infrastructure as long as they follow the framework design.

Obviously, not every method designed to solve a certain task can be applied once the method that solve the previous task has been fixed. For example, it makes no sense to apply a voting mechanism to obtain the result in the reuse process if the retrieval one returns just one case.

Apart from input/output constraints, method applicability can be also determined by more general constraints such as the requirement of a particular organization for the Case Base or the availability of a given type of similarity function defined on cases. These requirements are expressed as descriptions in a DL and correspond to the conditions to be satisfied by the *context* of the CBR system. The element *ContextInputPrecondition* in a method description describes the requirements that the application of the method impose on the input context, while the element *ContextOutputPostcondition* describes how the context is affected by the execution of this method. The method applicability checking is made using description logics and CBROnto.

2.2 Limitations that guide jCOLIBRI towards a distributed architecture

Nowadays, jCOLIBRI is managed using *sourceforge*, a software development website that provides a version control system (CVS). Users check out the source

code of the framework, and use its library of PSMs, or extend or create new methods. If the programmer wants to share her/his methods with all the community, (s)he has to commit the files to the central distribution.

Our goal with jCOLIBRI has been to provide with a reference framework for CBR development that would grow with contributions from the community. Even though it, we have found several difficulties within contributions due to the current monolithic architecture, namely:

1. Previously to the addition to the framework, all the contributions have to be processed, sometime too time consuming to the developer team.
2. It is not always easy to decide against incorporating some new method, because many of them, though useful for some other users, are too much specific for some kind of systems.
3. The contributions added to the framework are not incorporated to the local copy of the other users while they stay with the same version. As in the process of framework instantiation the system designer search for every task and method needed for the system using the local copy of the system, it is possible he is missing the opportunity of reuse other new methods.
4. CBR system designers usually find tedious (and a waste of time) to contribute to the method library of jCOLIBRI.

We have found that these difficulties can be tackled using a distributed architecture. This new approach is used both in the developing of a new CBR system and in its execution, using remote method calls and OWL-S [11] for the description of those methods.

3 Distributed Architecture

The distributed model is not meant to substitute the main core of the framework but help the publication of the new methods without having to contact with jCOLIBRI development team.

Users still checkout the last release of jCOLIBRI with the library of core PSMs, and they continue using the GUI in order to create new CBR systems. The difference arises when a jCOLIBRI user (CBR designer) has created (or modified) a method and (s)he finds it interesting enough for the rest of the community. Instead of sending it to be incorporated in the next release (increasing more and more the core of the framework) (s)he publicizes the method and allows that other (external) systems use it remotely.

With this approach, jCOLIBRI GUI should be able to find remote methods, i.e., it does not search only in the local copy of the framework but it uses the same techniques to search in the complete set of available remote PSMs.

3.1 Our proposal

Our proposal is using the jCOLIBRI GUI as a service discovery tool. Using the same techniques that we are using now (locally over the library of CBR

methods) [8], we aim to widen the scope of the search space to the semantic web, in particular to the set of CBR services (previously called methods, the PSMs) publicly available from the CBR community.

Our CBR ontology (CBROnto) defines CBR related terminology to describe CBR methods [6, 8]. So, the first step we are doing is exporting the methods in our library. OWL-S has some "hooks" where different kind of information can be added, even information outside OWL-S or outside OWL. These extensions could require some kind of specialized reasoner for them. We are integrating CBROnto in OWL-S using the hooks, using OWL itself as language to relate them.

The OWL-S Service Profile class does not dictate any representation of services: using OWL subclassing anyone can create specialized representations for them to be used as service profiles. We could design a new Service Profile subclass containing the information we consider important for our CBR methods.

Nevertheless, OWL-S provides the class *Profile* as a possible representation. An OWL-S Profile contains the functional description of the service specifying the inputs and preconditions required, the outputs generated, and the expected effects (postconditions). These four attributes are stored using OWL properties in the `Profile` class. It also contains more general information as the service name, a general description, a service category, etcetera. We have planned a mapping between the information currently contained in the XML Schema of our jCOLIBRI methods and the properties of OWL-S Profile.

1. **Name:** it is mapped to a string by means of the profile:ServiceName property.
2. **Description:** it also mapped to a string using the profile:textDescription property.
3. **Parameters:** they contain both input and outputs. OWL-S has a property called "`hasParameter`" which is subclassed in "`hasInput`" and "`hasOutput`". We will organize our previous parameter information to be correctly categorized as inputs or outputs using these properties. We will discuss about the range of `hasParameter` shortly.
4. **Competencies:** they are the list of tasks the method is able to solve. We are mapping them into the Profile serviceCategory property, using the ServiceCategory class (the range). In this way, ServiceCategory is used to describe categories of services on the bases of some classification that is outside OWL-S, but understood by our CBROnto specialized reasoner.
5. **ContextInputPrecondition:** we will use `hasPrecondition` property to store this information. We discuss its range (expr:Condition) below.
6. **ContextOutputPostcondition:** `hasResult` property will be used.

There are two elements of the method descriptions that are not mapped in OWL-S: the method type (execution or decomposition methods) and the subtasks. Both elements are relative to the task/method ontology. Currently we are limiting ourselves to use semantic web with the execution methods, so it is not important to incorporate into OWL-S information about decomposition.

Both preconditions and effects need information about the inputs and outputs of the methods respectively. They are expressed using OWL-S `Input` and `Output` classes, that are subclasses of `Parameter`. They must specify the parameter types, and, in some cases, their values. Type is store as an URI, and value as a plain text. Specialized reasoner using OWL-S as a "container" should be give sense to them when the discovery is taking place.

In our case, types are specified using the name of the classes in CBROnto that modelize them. Our reasoner will test if each class is a descendant class of CBRTerm.

OWL-S uses logical formulas to represent preconditions and effects. Instead of integrating them into RDF, OWL-S treats the formulas (expressions and conditions) as string literals or XML literals, which reference the inputs and outputs defined somewhere else. External reasoners are supposed to be able to analyse and understand these strings.

OWL-S has two basic classes concerning expressions. `Expression` class contains the string with the logical formula. It has the property `expressionLanguage` related to the `LogicLanguage` class. OWL-S includes three instances of this concept, referring to some concrete languages: SWRL, KIF and DRS. We have added the OWL language to describe description logic formulas that can be used to express the kind of conditions and effects that we need to model. Our reasoner used to service discovery employ the OWL expressions and RACER inference engine.

As said before, service profiles intention is to store information referring to "what the service does". OWL-S services also keep "how the service works" in the so-called service models. Concretely, OWL-S includes a Service Model class that, as Service Profile, is mainly empty, but concreted in the Process subclass. Its information is specially useful for composite services which store some kind of state between interactions.

The name "composite services" can suggest some kind of relationship with our decomposition methods. However, our decomposition methods cannot be modeled using the composite services supported by OWL-S because they have different semantic. Our decomposition methods follow a kind of "divide&conquer" philosophy being the method in charge of invoking the submethods. The OWL-S idea of composite services refers to the *user* calling to the different subservices. In other words, a composite process is not a behaviour a *service will* do, but a behaviour (or set of behaviours) the *client can* perform by sending and receiving a series of messages [11]. Consequently, our methods will be always atomic process from the OWL-S point of view, and we are not currently interested in making an advanced use of the Process subclass.

CBR services discovery

Converting jCOLIBRI framework to a distributed component-based system using OWL-S implies a change in the way the jCOLIBRI development GUI works. We need some kind of central registry where third-part components (also called methods or services) are published using OWL-S + CBROnto, and the

development GUI searches concrete methods using it, depending on the user requirements.

This central registry extracts the information referring to the query from the "OWL-S container", and obtains the specification on top of CBROnto in order to look for some existing method using our techniques based on description logics. Concretely, the main issue of verifying whether or not a service satisfies the set of restrictions is accomplish using Racer as inference engine.

As said before, currently we are not interested in method composition at this level. Tasks are decomposed using the task/method ontology in the *local* development tool, and all the queries are concerned to the "leaf" methods once all the decompositions have been decided.

4 Conclusions

We have presented jCOLIBRI, an object-oriented framework in Java to build CBR systems. This framework is built around a task/method ontology that facilitates the understanding of an intrinsically sophisticated software artifact. The current implementation of jCOLIBRI has been recently released as an open source effort to serve as development tool and profit from the input of the CBR community.

In the current framework-centered architecture of jCOLIBRI new CBR systems are developed through framework instantiation. In this process, users may extend available classes, developing new methods as needed. In our role as developers of the main core of the system, we expect those programmers to contribute to our method collection with the most relevant ones. We intend to process and filter all these third-party methods and add to the next release those potentially interesting to jCOLIBRI users.

In this paper we have proposed a new distributed architecture for jCOLIBRI profiting from the similarities between PSMs and component-based reuse. jCOLIBRI provides a battery of PSMs, and the programmer searches for the most useful for his purpose. Separating PSMs from the main core by using Web Services get us closer to the concept of software components technology, bringing its possibilities to jCOLIBRI. In that sense, both services and main core can evolve independently, so the version problem decrease. Each method developer is responsible for controlling his own versions of each method, and guarantees the backward compatibility, maybe using techniques used in other component technologies such as DCOM or Enterprise JavaBeans.

Web Services and remote invocation let the implementers choose a programming language different to that used in the jCOLIBRI implementation (Java). Any developer who respects the rules of the Semantic Web and creates correct descriptions in OWL-S of his services using CBROnto will be creating methods that will be available for the rest of the community.

The main drawback of distributed architecture is the performance due to the speed of remote calls, specially when the method granularity is high. The methods' implementer should create them with this problem in mind, trying to

provide suitable interfaces for them but minimizing the number of invocations. Another solution is to let the user of the method to get a copy of the source code to allow him to install the PSM as a local component reducing the call overload.

Our (ambitious) goal is to provide a reference framework for CBR development that would grow with contributions from the community. This reference would serve for pedagogical purposes and as bottom line implementation for prototyping CBR systems and comparing different CBR approaches to a given problem. This idea is so mature in the community that several efforts are pursuing it at time of writing: CAT-CBR [3], a component-based platform for developing CBR systems; JavaCREEK the Java implementation of the CREEK architecture for knowledge-intensive CBR systems [1]; IUCBRF [5] a Java framework developed at Indiana University, to mention just a few. The main contribution of the work presented in this paper is along the line of proposing a distributed architecture where different approaches to CBR system development would collaborate instead of compete.

References

1. A. Aamodt. Knowledge-intensive case-based reasoning in CREEK. In *Procs. of the (ECCBR 2004)*, pages 1–15. Springer-Verlag, 2004.
2. A. Aamodt and E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(i), 1994.
3. C. Abásolo, E. Plaza, and J.-L. Arcos. Components for case-based reasoning systems. *Lecture Notes in Computer Science*, 2504, 2002.
4. J. J. Bello-Tomás, P. A. González-Calero, and B. Díaz-Agudo. JColibri: An object-oriented framework for building cbr systems. In *Procs. of the (ECCBR 2004)*, pages 32–46. Springer-Verlag, 2004.
5. S. Bogaerts and D. Leake. *IUCBRF: A Framework For Rapid And Modular Case-Based Reasoning System Development.* http://www.cs.indiana.edu/ sbogaert/CBR/IUCBRF.pdf.
6. B. Díaz-Agudo and P. A. González-Calero. An architecture for knowledge intensive CBR systems. In E. Blanzieri and L. Portinale, editors, *Advances in Case-Based Reasoning – (EWCBR'00)*. Springer-Verlag, Berlin Heidelberg New York, 2000.
7. B. Díaz-Agudo and P. A. González-Calero. Classification based retrieval using formal concept analysis. In *Procs. of the (ICCBR 2001)*. Springer-Verlag, 2001.
8. B. Díaz-Agudo and P. A. González-Calero. CBROnto: a task/method ontology for CBR. In S. Haller and G. Simmons, editors, *Procs. of the 15th International FLAIRS'02 Conference (Special Track on CBR*, pages 101–106. AAAI Press, 2002.
9. A. Gómez and R. Benjamins. Overview of knowledge sharing and reuse components: Ontologies and problem-solving methods. In *IJCAI99 workshop on Ontologies and Problem-Solving Methods*. Sweden, 1999.
10. T. Schreiber, B. J. Wielinga, J. M. Akkermans, W. V. de Velde, and R. de Hoog. CommonKADS: A comprehensive methodology for KBS development. *IEEE Expert*, 9(6), 1994.
11. The OWL Services Coalition. *OWL-S: Semantic Markup for Web Services.* http://www.daml.org/services/owl-s/1.1/overview/.
12. B. Wielinga, A. Schreiber, and J. Breuker. Kads: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1), 1992.