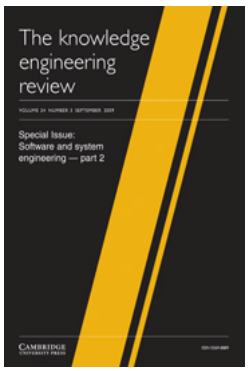

Semantic templates for case-based reasoning systems

Juan Antonio Recio-García, Belén Díaz-Agudo and Pedro Antonio González-Calero

The Knowledge Engineering Review / Volume 24 / Special Issue 03 / September 2009, pp 245 - 264
DOI: 10.1017/S0269888909990051, Published online: 18 September 2009

The Knowledge Engineering Review, Vol. 24:3, 245–264. © 2009, Cambridge University Press
doi:10.1017/S0269888909990051 Printed in the United Kingdom

Semantic templates for case-based reasoning systems

JUAN ANTONIO RECIO-GARCÍA, BELÉN DÍAZ-AGUDO and PEDRO ANTONIO GONZÁLEZ-CALERO

Department of Software Engineering and Artificial Intelligence, Universidad Complutense de Madrid, C/ Prof. José García Santesmases, s/n. 28040, Madrid, Spain;
e-mail: jareciog@fdi.ucm.es, belend@sip.ucm.es, pedro@sip.ucm.es

Abstract

In this paper, we present an approach to solve the drawbacks of manual composition of software components. Our approach is applied within the jCOLIBRI framework for building case-based reasoning (CBR) applications. We propose a system design process based on reusing *templates* obtained from previously designed CBR systems. Templates store the control flow of the CBR applications and include semantic annotations conceptualizing its behavior and expertise. We use CBR ontology to formalize syntactical, semantical and pragmatical aspects of the reusable components of the framework. The ontology vocabulary facilitates an annotation process of the components and allows to reason about their composition, facilitating the semi-automatic configuration of complex systems from their composing pieces.

1 Introduction

The key idea in software reuse is that most software systems are variants of systems that have already been built. Research within software reuse in the past 20 yrs includes reuse libraries, domain engineering methods and tools, reuse design, design patterns, domain-specific software architecture, componentry and applications generators (Frakes & Kang, 2005). Overlapping several of those research areas, application frameworks distill some key findings in software reuse ranging from low-level component reuse to high-level design and architecture reuse.

A framework is a reusable, semi-complete application that can be specialized to produce custom applications (Johnson & Foote, 1988). Application frameworks are targeted at a given application domain providing the design for a family of applications within that domain. In the last few years, we have developed several iterations of the jCOLIBRI (Díaz-Agudo *et al*., 2007) framework for building case-based reasoning (CBR) applications.

One of the biggest problems with frameworks is learning how to use them. A system developer needs to know the framework design to a certain extent, at least to determine what classes must be instantiated in order to obtain a given functionality. A deeper knowledge of the framework is needed when extending it. As the complexity of a framework rise, the manual composition of its composing software pieces became more difficult and tedious. In this paper, we present an approach to solve the drawbacks of manual software components composition applied to the jCOLIBRI framework.

Our approach is based on ontologies that formalize syntactical, semantical and pragmatical aspects of the framework components. The ontology vocabulary facilitates an annotation process of the components and allows to reason about their composition, facilitating the semi-automatic configuration of complex systems from their composing pieces. Thus, the knowledge that a

software engineer must acquire to compose an application using the framework is already included in the semantic description of the components. Moreover, we have designed several composition tools to guide the composition process using these annotations. Knowledge about CBR systems that abstracts the framework components is represented in CBROnto (Díaz-Agudo & González-Calero, 2002). CBROnto is an ontology incorporating general CBR terminology and problem-solving knowledge in terms of tasks and methods resulting from a knowledge-level analysis of CBR systems.

In order to facilitate the framework instantiation process, we propose a case-based approach where the designer retrieves a system from a library (case base) of previously designed CBR systems and, if needed, adapt it by adding, removing or substituting components in the selected system. Retrieval and adaptation of systems are possible through the use of *semantic templates* that have been previously abstracted from available systems. Each template is a generalization of several CBR systems that also includes semantic annotations from a human expert. Templates store the control flow of CBR systems, conceptualizing their behavior, and including the concepts and constraints required to model a number of related systems. The similarity between two given systems, or between a system and the designer query, is based on computing the similarity between the templates that abstract the systems. System adaptation is guided by the system level constraints represented in the template. Reasoning with templates is possible because they are formalized and annotated through the ontology.

jCOLIBRI offers graphical tools that guide the generation of CBR applications taking advantage of the knowledge represented in CBROnto: namely, a tool to create and formalize semantic templates; a tool to assist in the retrieval and adaptation of the system that best suits the user requirements; and a tool to check the coherence of the resulting composition using a subsumption-based approach.

The paper runs as follows. Section 2 introduces CBR system construction in jCOLIBRI and presents the case-based approach to software construction, introducing the domain of recommender systems that is used to exemplify the approach. Section 3 describes how to formalize the templates, using vocabulary from CBROnto, to support system retrieval and reuse. Section 4 describes how to measure similarity between systems and Section 5 describes how to adapt a selected system in a semiautomatic process based on the semantic descriptions and the subsumption mechanism. Finally, Section 7 reviews related work and concludes the paper.

2 Designing case-based reasoning systems with jCOLIBRI

Case-based reasoning (see Kolodner (1993) and Leake (1996)) is a well-known reasoning paradigm based on the value of the experience, and 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.

Now that CBR is a mature and established technology, two necessities have become critical: the availability of tools to build CBR systems, and the accumulated practical experience of applying CBR techniques to real-world problems. Many researchers in the field agree about the increasing necessity to formalize this kind of reasoning, define application analysis methodologies and provide a design and implementation assistance with software engineering tools. The best way for achieving this goal is the development of a framework. Frameworks are a well-known technology related with software reuse that leverages the prior efforts of developers in order to avoid recreating and revalidating common solutions. During the last few years, our research group has developed jCOLIBRI, an object-oriented framework in Java for building CBR systems, which greatly benefits from the reuse of previously developed CBR systems. The design process of CBR systems in jCOLIBRI (version 1) (Recio-García *et al.*, 2006) relies on decomposition methods that solve a certain task by breaking it into a linear sequence of subtasks. Each subtask is itself solved by either a method or another decomposition, depending on its complexity level. With this design process, the reusable components are tasks, methods and case-storage utilities.

2.1 The two-layer architecture

In Recio-García *et al*. (2006), we have reviewed the advantages and drawbacks of the jCOLIBRI framework (version 1) from the experience of 2 yrs of development. The new version of the framework (jCOLIBRI 2) tries to solve many of the problems identified in the previous version. jCOLIBRI 2 follows a new and clear architecture divided into two layers (Figure 1): one oriented to developers and the other oriented to designers (that is the focus of this paper).

The key idea in the design of the new version of the framework is to separate the core classes from the user interface. The bottom layer contains the basic components of the framework. It does not contain any kind of graphical tool for developing CBR applications; it is a white-box object-oriented framework that must be used by programmers. In this paper, we focus on the top layer, in which each component is annotated with semantic descriptions using technologies from the Semantic Web (Recio-García *et al*., 2007) and in which there are different tools that aid users in the development of CBR applications. Hence, the top layer offers a black-box framework with a visual builder. For the top layer, we propose a semi-automatic process to compose systems from semantically annotated components.

2.2 A case study: recommender systems

Our approach of designing CBR systems in case-based fashion, retrieving and adapting templates from a library of templates (i.e. a case base of CBR design experience) has one important bottleneck: the acquisition of an initial design experience template case base. To begin with, we have limited our system to work with a well-known and successful family of systems: recommender systems. A recommender system infers the goals and preferences of its user; it uses the inferred knowledge to select and/or rank products, services or information sources (generically called items); and it recommends to its user, items that may satisfy the inferred goals and meet the inferred preferences. Recommender systems combine ideas from information retrieval, information filtering, user modeling, machine learning and human–computer interaction.

Case-based reasoning has played a key role in the development of several classes of recommender system (Bridge *et al*., 2006; Smyth, 2007). As a case study, we have done an analysis of recommender systems that is based, in part, on the conceptual framework described in the review paper by Bridge *et al*. (2006). The framework distinguishes between collaborative and case-based,

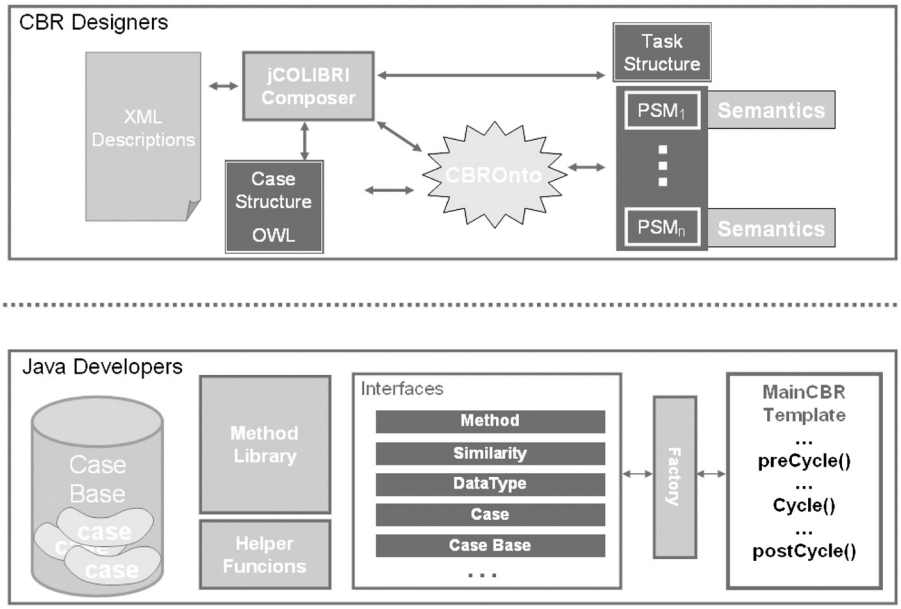

Figure 1 jCOLIBRI 2 architecture

reactive and proactive, single-shot and conversational, and asking and proposing. Within this framework, the authors review a selection of papers from the case-based recommender systems literature, covering the development of these systems over the last 10 yrs (we could cite Hammond *et al*. (1996), Wilke *et al*. (1998), Smyth and Cotter (1999), Bergmann (2002), Burke (2002), Shimazu (2002) and McSherry (2002) as illustrative examples).

Although we have used the domain of recommender systems as a case study to test our approach, our proposal is not limited to this type of CBR systems and could be applied to compose applications of other CBR families. For example, we are currently including templates to develop Textual CBR applications (where cases are represented as plain texts) that illustrate the plausibility of our thesis.

The composition schema presented here could not be applicable to compose generic software applications, but we believe that it is suitable for a wide spectrum of CBR systems as it is a well-known domain where the behavior and structure of the applications can be easily abstracted. In the case of the templates presented in this paper, they were designed in a few hours by an expert on the domain and they served, afterwards, to generate 14 different applications that exemplify a wide range of recommender systems.

2.3 Designing systems in a case-based reasoning way

We propose a semi-automatic way of designing CBR systems where the designer interacts with jCOLIBRI to configure the system. Once we have a library of templates obtained from a previously designed set of CBR systems. In case-based fashion, jCOLIBRI will retrieve templates from a library of templates (i.e. a case base of CBR design experience); the designer will choose one, and adapt it. For the present at least, we consider fully automatic design of CBR systems to be unachievable. It would require a very rich semantic mark-up of the reusable components. In our approach, jCOLIBRI suggests suitable substitutions from the semantic annotations in the components, but the designer manually chooses the component and adjusts the configuration for its input/output parameters.

In a general way, the templates of the jCOLIBRI library are composed by *tasks* that identify the steps of the CBR system and *methods* that solve each task with a particular implementation.

A *system* (or *executable template*) is a template where each task is solved by a method. Our design tool retrieves systems from the library and adapts them to create new CBR systems. To retrieve the system closer to the user preferences, we use similarity measures that take into account the structure of the template (i.e. the flow of tasks) and the methods that solve each task. Once a system is retrieved, it is adapted substituting its methods by another methods that solve the same task. When adaptation finishes, the tool generates most of the code required to execute the system.

Templates reflect the execution order of the methods without detailing the flow of parameters between the output of a method and the input of the following one. The connection between parameters must be performed manually by the developer. It means that the code generated by our tool is not fully complete. We have chosen this semi-automatic approach to simplify the representation of templates and to increase the flexibility when substituting methods. If we annotate the inputs and outputs for each task in a template, we could only use methods with the same signature to solve the task. If we do not specify this signature, we could adapt each task with other methods that may have additional parameters, and then, let the developer specify how to obtain that information.

Our library of templates is organized in the following way:

- At the bottom level there are *executable templates* (or *systems*). Each task of these templates is solved by a method. These tasks are called *simple tasks* because they can be directly solved by a method.
- Systems are organized according to the *final template* they implement. Final templates are also composed of simple tasks, but their representation does not include the methods that solve each task. Thus, we can say that a final template can be instantiated by several systems.

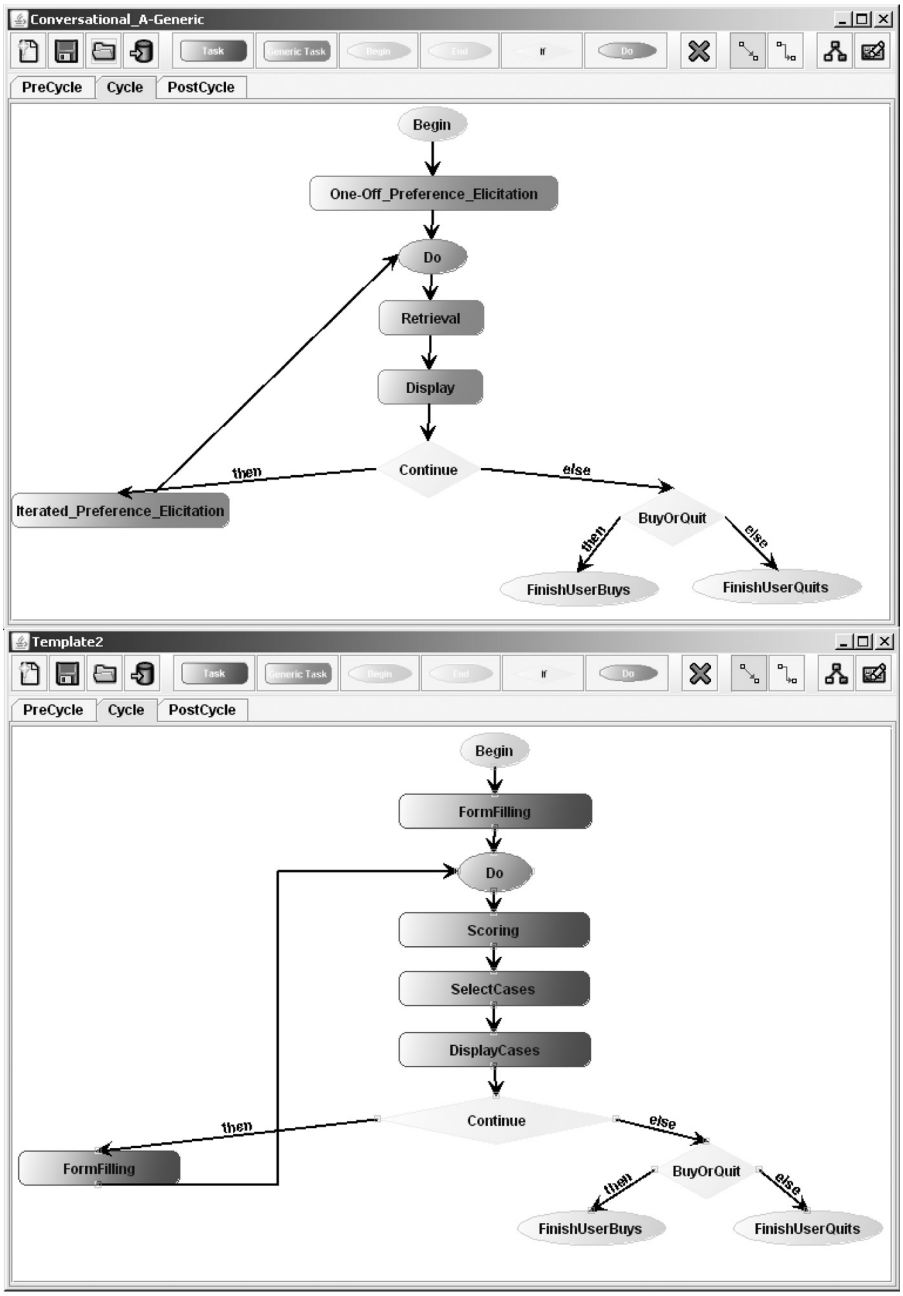

Figure 2 Template Graphical Editor

- At the top level, there are *generic templates* that provide a high-level view of a set of final templates. These templates are composed by *generic tasks* and *simple tasks*. Generic tasks encapsulate sequences of simple tasks. Depending on the decomposition of each generic task into sequences of simple tasks, we obtain several final templates. Therefore, generic templates summarize families of final templates that share a common global structure. These kind of templates are useful to obtain a high-level description of a system and also to compute the similarity between them.

In order to work with the templates we need to formalize and incorporate them into jCOLIBRI. We have developed a graphical template editor to create and save the templates of CBR systems. Figure 2 shows our template editor. The screenshot on the top shows the generic template of a conversational recommender (named Conversation A recommender). The bottom screenshot

shows a final template that extends the previous one. In this case, the 'retrieval' generic task is decomposed into the 'scoring' and 'select cases' simple tasks. Depending on the methods used to solve each simple task, we will obtain different systems.

Our approach poses a number of interesting research challenges:

- How to obtain a set of templates that are representative enough. The case base of templates represents design experience. Having a case base of templates requires thinking about granularity, level of generality, diversity and coverage of the template case base. Besides, we need to represent dependencies between the elements of a template, and annotate the templates and their components. Each template has different variability points, that is, tasks that can be solved in different ways using different methods.

  Section 3 (and Recio-García *et al.* (2007)) focuses on this challenge: obtaining and representing a set of templates. As the CBR community has produced many different families of systems, we begin by narrowing the scope of our template library to cover one of the most successful families: recommender systems, especially case-based recommender systems.
- How to define a similarity metric between the query and templates based on the semantic annotations of the components. Section 4 describes the similarity measures used to compare templates.
- How to adapt templates. The question that arises here is how easy it will be to (manually) adapt a template to fulfill the requirements of the user. Although it is a difficult process, the system can help by taking into account dependencies, and suggesting substitutions. In order to support these four processes, we are working on the semantic annotation of components and templates using vocabulary from CBROnto (Díaz-Agudo & González-Calero, 2002). The CBROnto ontology formalizes the syntactic, semantic and pragmatic aspects of the reusable components of the framework. Using the CBROnto vocabulary, we can annotate the components and reason about their composition. This will facilitate the automatic configuration of complex systems.

The following sections describe our contributions according to these challenges, beginning with obtaining and representing a set of templates for the family of case-based recommender systems.

3 Template representation

The template-based design process described in Section 2 requires a case base of templates of CBR systems represented in a formal language that allows us to retrieve and adapt the template according to the user requirements.

The top layer of the jCOLIBRI 2's architecture (Figure 1) is based on CBROnto. It is an ontology that stores the knowledge required to represent the templates, their components (mainly tasks) and, finally, the methods that will solve each task. The ontology is used during the process of template design, retrieval and adaptation. Although each step plays a very different role, CBROnto keeps all the knowledge required to implement them in a coherent way.

Figure 3 shows the main concepts of CBROnto. Dotted lines represent properties and solid ones represent subclassing. We can study the ontology through the three steps followed by our CBR application composer:

Templates representation. This part of the ontology contains the concepts required to represent the templates. In this way, there are concepts that define the steps of a CBR application (preCycle, Cycle, postCycle,…) and their flow of control (sequence, if-then-else,…). Tasks play a central role in the templates representation, as they are the basic building blocks of templates. The template representation concepts of CBROnto are adaptations of several Semantic Web Services standards as detailed next.

Templates retrieval. The process of retrieving templates also takes advantage of the knowledge in the ontology. We have defined several concepts that classify templates according to CBR systems features (by user information request, by selection methods,…). Using the capabilities

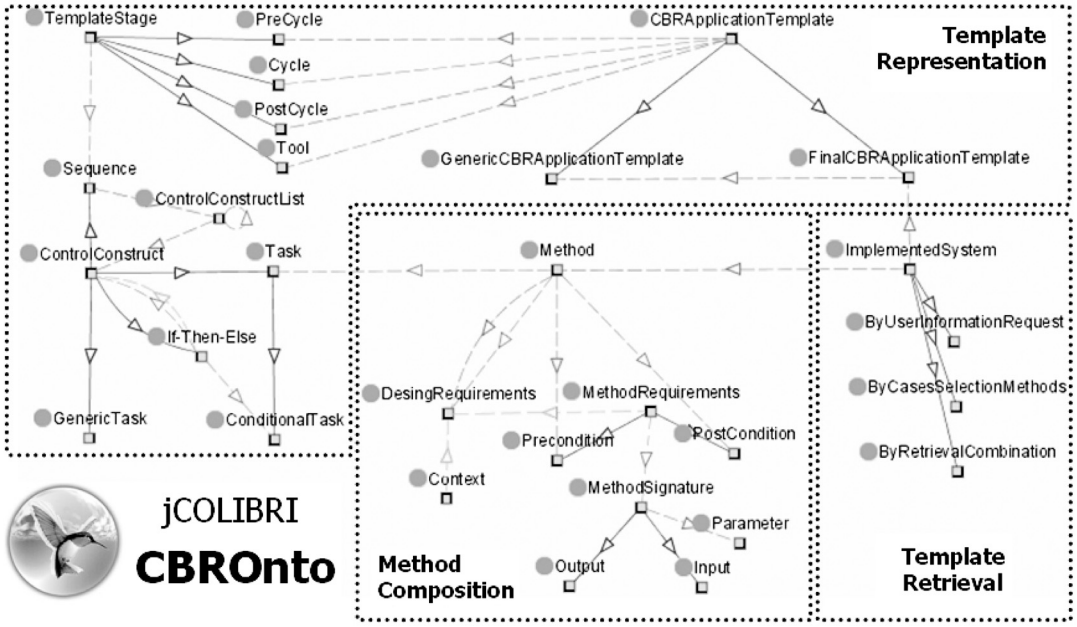

Figure 3 CBROnto overview

of the description logics (DL) reasoners, we can classify the templates and compute their similarity as explained in Section 4.

Templates adaptation. Adaptation of the retrieved template means assigning methods that solve each task. Each task requires some input data and can be executed only under certain conditions (e.g. that some other tasks were executed before). These data and condition requirements are represented in the ontology as the input/output and precondition/postcondition description of a task. Every method is also described with this information. This allows us to check if a method can be used to solve a task, using again the classification capabilities of the description logics. Section 5 details this process.

We use the OntoBridge library[1] to manage the ontology inside our application. This tool is open source licensed and based on the JENA library. It simplifies the managing of ontologies and the use of DL reasoners inside java applications.

From the template graphical editor (shown in Figure 2), we export the representation to reason with the templates. There are many formalisms for representing workflow templates. We use Semantic Web Services (SWS). In the SWS community, there are different standards to represent the behavior of software components and their composition. Examples are Ontology Web Language-Services (OWL-S) and Web Service Modelling eXecution environment (WSMX). Both formalisms have in common the use of ontologies to represent the information about components, a feature that fits perfectly in our system because we already use an ontology for CBR systems: CBROnto. As it stood, CBROnto provided the vocabulary to describe the methods (components) of the template framework, but it lacked a way to describe more complex control flow. We have now added the vocabulary needed to represent this aspect of the templates. For this, we use the approach of the OWL-S ontology, where several concepts are specially designed to represent the workflow of services. Our choice was pragmatic: CBROnto is already represented using the OWL ontology language, OWL-S also uses this language, and its representation of the workflows are much more suitable in our case than the WSMX approach.

The vocabulary that we have added is shown in Figure 3. A template (CBRApplicationTemplate) can be generic or final. Generic templates have at least one generic task and final templates

[1] `http://gaia.fdi.ucm.es/projects/ontobridge/`

are only composed of simple tasks. Each template is divided into three main stages: the preCycle (code to initialize the application), Cycle (main CBR cycle) and postCycle (post execution code). Moreover, a CBR application can have tools used for maintenance and other similar purposes. Each application stage contains a sequence of elements: a Begin Task, a Final Task, Generic Tasks, Conditions and other subsequences. A sequence is composed of a Control Construct List that contains Control Constructs. Conditionals (If-Then-Else) are defined with a Conditional Task (a task that returns a Boolean value) and two Control Construct branches that correspond with the then and else branches. All these concepts are direct adaptations of the OWL-S ontology. Our template graphical editor (Figure 2) allows us to create templates graphically and save them using the OWL formalism. Before creating a new template, the user must introduce some metadata: template name, authors, type, generic or final template, additional tools and so on. Then, each workflow can be created in a graphical way. When the user creates a new Task or Generic Task element, the tool reads the instances contained in CBROnto that belong to these concepts. Those instances represent the tasks solved by the methods of the framework and are included as new components. Once a template is ready, the editor generates the OWL representation using the concepts and properties defined in the extended CBROnto ontology (Figure 3). This generated representation is also shown graphically.

4 System retrieval

The process of selecting templates suitable for the user requirements is an important research challenge. If the user expresses her design requirements with the same vocabulary used to formalize the templates, the only aspect to solve is how to measure the similarity and differences between templates.

Our approach to compute the similarity between templates consists on comparing the structure of the templates looking for similar decompositions of complex tasks into simple ones. However, this approach does not take into account the behavior of the methods that solve the tasks. This feature is important because two templates with different structure can be instantiated with a set of methods that make the final systems behave in a similar fashion. Finally, there are other global features of the CBR systems that can be very useful to compute the similarity. These features must be defined by the expert that generates the templates and they are usually related to structural and behavioral characteristics of the template.

A good example of the latter classification is the recommenders domain. The templates and systems that are included in jCOLIBRI 2 can be also defined through the kind of information requested to the user: all the preferences (FormFilling), some preferences (Navigation-by-asking), preference profile (content-based profile) or using past recommendations. We can define the first group with the feature 'the template contains the FormFilling Task', but to define the recommenders that use past recommendations we use the feature 'the template contains the ReadProfile task and the CollaborativeRetrieval method'. In this way, the description of the templates mixes structural and behavioral characteristics that must be defined by the expert on the domain.

Our similarity measure combines these three approaches:

$$\mathrm{sim}(t_1, t_2) = w_1 \times \mathrm{ts}(t_1, t_2) + w_2 \times \mathrm{mc}(t_1, t_2) + w_3 * \mathrm{gf}(t_1, t_2)$$

where ts() is the task structure similarity, mc() the methods comparison measure and gf() the global features estimation. The weights of each metric (w_1, w_2, w_3) were obtained through empirical experiments. Results are detailed in Recio-García *et al.* (2008), where $w_1 = 0.1$, $w_2 = 0.4$, $w_3 = 0.5$ is reported as the best distribution. We use a slot-based representation for each template that allows us to compare them like normal cases of any CBR application. The details of the metrics and representations are:

Task structure similarity. Let G be the set of complex tasks $\{C_1, C_2, C_3, \ldots, C_n\}$ and Q the set of possible decompositions of G into sequences of simple tasks $Q=\{Q_1, Q_2, Q_3, \ldots, Q_n\}$. Each sequence Q_i is composed by several simple tasks $S=\{S_1, S_2, S_3, \ldots, S_n\}$. When comparing the

$$\text{cosine}(i_1, i_2) =$$

$$\text{sim}(\text{t}(i_1), \text{t}(i_2)) = \frac{\left| \left(\bigcup_{d_i \in \text{t}(i_1)} (\text{super}(d_i, CN)) \right) \cap \left(\bigcup_{d_i \in \text{t}(i_2)} (\text{super}(d_i, CN)) \right) \right|}{\sqrt{\left| \bigcup_{d_i \in \text{t}(i_1)} (\text{super}(d_i, CN)) \right|} \cdot \sqrt{\left| \bigcup_{d_i \in \text{t}(i_2)} (\text{super}(d_i, CN)) \right|}}$$

where: CN is the set of all the concepts in the current knowledge base; super(c, C), is the subset of concepts in C which are superconcepts of c; t(i) is the set of concepts the individual i is the instance of

Figure 4 Concept-based similarity function in jCOLIBRI 2 (Fernández-Chamizo *et al.*, 1996)

structure of the templates, the slots of the cases are defined by the complex tasks G and the possible values of these slots are the sequences of simple tasks Q. When comparing the attributes (slots) of a case, we use the equal function that returns 1 if they have the same value and 0 otherwise.

Methods comparison. In this approach, the slots of the case are the simple tasks S and the possible values of the slots are the methods that solve the tasks $M = \{M_1, M_2, M_3, \ldots, M_n\}$. To compute the similarity between methods we have organized them into subconcepts of the *Method* concept of CBROnto. These subconcepts conform a hierarchy that organizes the methods available in the framework according to their behavior. Then we apply the ontological similarity measures implemented in jCOLIBRI 2 to compare the methods. This family of ontological measures uses the structure of the ontology to compute the similarity between instances. Figure 4 shows the details of the applied similarity function. This function is described in Fernández-Chamizo *et al.* (1996) and is based on the vector space model. This measure represents each concept of the ontology as a vector of properties. The length of the vector is the number of concepts in the ontology and each position contains a 1 or 0 if the represented concept is a subconcept or not of the concept associated to that position. That work proposes a similarity function based on the cosine of the vectors that represent each concept and demonstrates that it can be calculated using the formula shown in Figure 4.

Global features. Here the ontology is used to store the expert's knowledge required to classify systems. Many times, systems must be classified according to heterogeneous features defined by a mixture of tasks and methods. These features can only be obtained from the experience of the expert in the development of such systems and they define several axes of classification for the templates. For example, in the recommenders domain, we can classify applications depending on the preference elicitation approach: Navigation-by-Asking (asking questions to the user) or Navigation-by-Proposing (proposing items to the user). Another classification could be the retrieval process: filtering or scoring. These two features (navigation type and retrieval process) define two different ways of classifying recommenders, and by extension, the templates that were extended to generate those systems.

Note that these features may overlap. For example, we can also classify templates according to the user information request: all preferences, some preferences, using a profile,… . Thus, a template classified according to its navigation type as a 'Navigation-by-Asking' template will be classified also as a template that requests some preferences to the user.

All these features (navigation type, retrieval process, user information request,…) serve to classify the templates from different points of view and are defined as concepts of the ontology. Figure 5 shows the concepts created to represent the global features of recommender systems. The first feature is 'ByCasesSelectionMethod' that classifies recommender systems into three groups: the ones that just select the most similar case (item), the systems that select similar and diverse cases and, finally, the group of systems that has lack of selection. The second feature distinguishes recommenders according to the type of preference elicitation: modifying the query with new requirements asked to the user, using the information given by a selected item or without this step. Following this idea, remaining

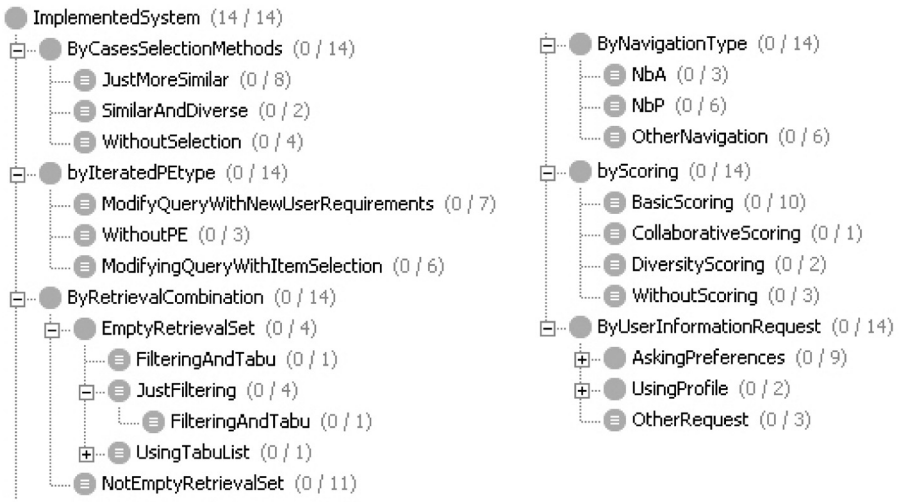

Figure 5 Global features classification

features also define several ways for classifying the recommenders: retrieval combination, navigation type, scoring strategy or the type of information requested to the user.

These features are very useful to compute the similarity of the templates because they classify systems into several partitions. To compute this similarity we coded into CBROnto the concepts shown in Figure 5. Each feature is described using DL predicates that represent a particular combination of tasks and methods (i.e. Navigation-by-Asking systems will contain the 'select attribute' and 'ask question' methods). Moreover, each implemented system is represented into the ontology as an instance with several properties that describe its tasks and methods. Then, using the capabilities of a DL reasoner, we can infer the concepts (features) that classify each instance (system). Thus, we get the features that describe each system and, by extension, the template that generalizes it. This process is also illustrated in Figure 5 through the Protege tool. After invoking the DL reasoner, Protege shows two numbers next to the concept name. They indicate the number of instances (systems) defined in each concept followed by the inferred instances obtained by the reasoner. The first number is always 0 because systems are defined as instances of the 'ImplementedSystems' concept (here, this first number is 14). The following number indicates the inferred instances (systems) that are classified automatically into the corresponding subconcept (feature).

To compare two systems, we classify the instances that represent them into CBROnto. These instances will be classified into one of the groups (subconcepts) defined under each feature. Then for each feature we measure the distance of both instances inside the subtree defined by that feature. For example, imagine that r_a and r_b were instances representing two recommenders. For each feature, both instances will be classified into some of the subconcepts defined under this feature. If we look to the 'ByNavigationType' feature, they can be classified into 'NbA', 'NbP' or 'OtherNavigation' subconcepts. In the case that both were classified under the same node (i.e. Navigation-by-Asking), their similarity will be 1. If they were classified into different nodes (one in Navigation-By-Asking and other in Navigation-by-Proposing) their similarity is obtained using the ontological similarity metric described in Figure 4. This process is repeated for each feature and finally we compute the average of all the similarity values given by each one.

4.1 Retrieval example

To exemplify the behavior of the retrieval and similarity process let us use a real case study. The designer wants to develop a single-shot system (make a recommendation and finish) that obtains the user preferences using a form, then computes a nearest neighbor retrieval and finally presents

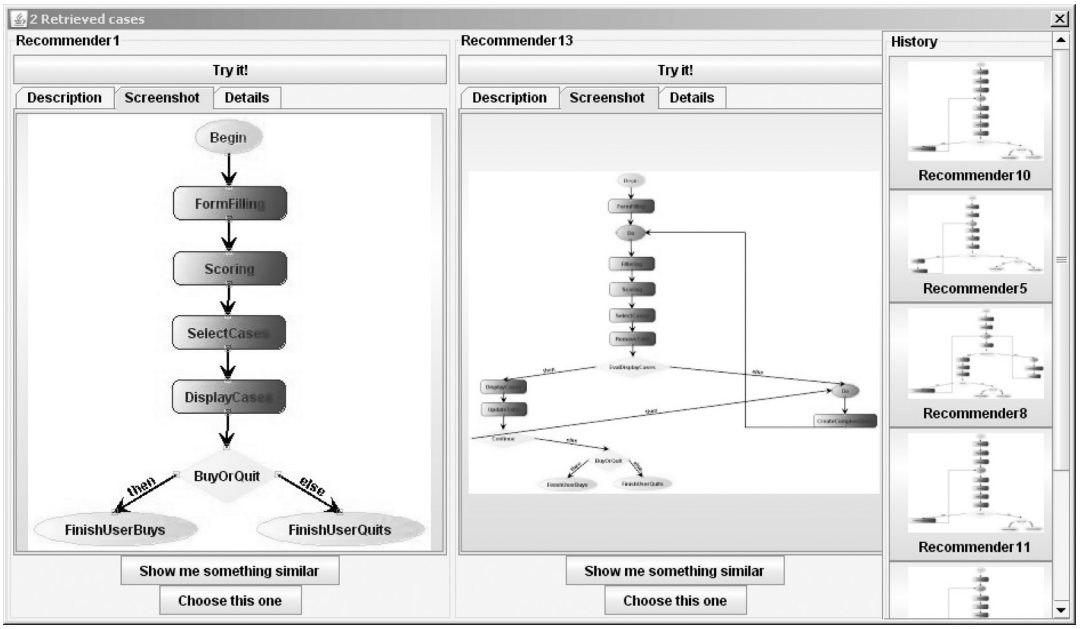

Figure 6 Screenshot of the templates retrieval tool

the most similar cases to the user. In our templates library, we have already included this system (labeled as system #1) but let us imagine we have not. In this case, the user will retrieve the single-shot system for collaborative recommendations (system #12). Both systems extend the same template although system #12 solves the Scoring task using a collaborative method instead the nearest neighbor one. In this way, the retrieved template (shown in Figure 6 (left)) will be selected by the user, as it is the closest to his/her requirements and then adapted as explained in Section 5.

As example, let us explain the similarity measure between system #12 and another system that extends the template depicted in Figure 2 (bottom). This system (labeled as system #2) implements a Conversational recommender using the form filling and NN retrieval methods. Figure 7 summarizes their tasks structure, assigned methods and inferred global features.

Using these system descriptions we compute the following similarity:

$$\mathrm{ts}(\#2, \#12) = 2/4$$
$$\mathrm{mc}(\#2, \#12) = 4/17$$
$$\mathrm{gf}(\#2, \#12) = 3/6$$
$$\mathrm{sim}(\#2, \#12) = 0.1 \times 2/4 + 0.4 \times 4/17 + 0.5 \times 3/6 = 0.39.$$

Note that CBROnto also includes a method classification hierarchy that is used to compute mc( ) through the cosine metric described in Figure 4. This hierarchy is not shown because of lack of space. Here, NNScoringMethod and CollaborativeRetrievalMethod are classified under the same concept ScoringMethods. DisplayCasesInTableEditQuery and DisplayCasesInTableBasic are also classified under the same concept DisplayCasesMethods. As the cosine measure returns 1 if two instances belong to the same concept, we obtain the 4/17 value. Finally, global features are obtained using the classification shown in Figure 5.

5 System adaptation

The final step of our design tool is the adaptation of systems (executable templates). Here, the tool guides the designer to substitute the methods that solve each task of the system to create a new one. Each method will require some data and conditions to be executed. After its execution, it will return some other data and modify the *state* of the CBR application with new conditions. This is

		System #2	System #12
Task Structure	One-Off P.E.	FormFilling	FormFilling
	Retrieval	Scoring+Selection	Collaborative
	Display	Display	Display
	Iterated P.E.	Form Filling	
Methods	FormFilling	FormFillingWithInitialValues	null
	SelectQuestion	null	null
	AskQuestion	null	null
	ReadProfile	null	ObtainQueryFromProfile
	Scoring	null	null
	SelectCases	null	null
	DisplayCases	null	null
	Filtering	null	null
	Scoring	NNScoringMethod	CollaborativeRetrievalMethod
	Selection	SelectTopK	SelectTopK
	RemoveTabu	null	null
	Display	DisplayCasesInTableEditQuery	DisplayCasesInTableBasic
	UpdateTabu	null	null
	FormFilling	FormFillingWithInitialValues	null
	SelectQuestion	null	null
	AskQuestion	null	null
	CreateComplexQuery	null	null
Global Features	CasesSelectionMethods	JustMoreSimilar	JustMoreSimilar
	IteratedPEtype	ModifyQueryWithNewUserRequirements	WithoutPE
	NavigationType	OtherNavigation	OtherNavigation
	RetrievalCombination	NotEmptyRetrievalSet	NotEmptyRetrievalSet
	Scoring	BasicScoring	CollaborativeScoring
	UserInformationRequest	AskingPreferences	UsingProfile

Figure 7 Systems description to compute their similarity

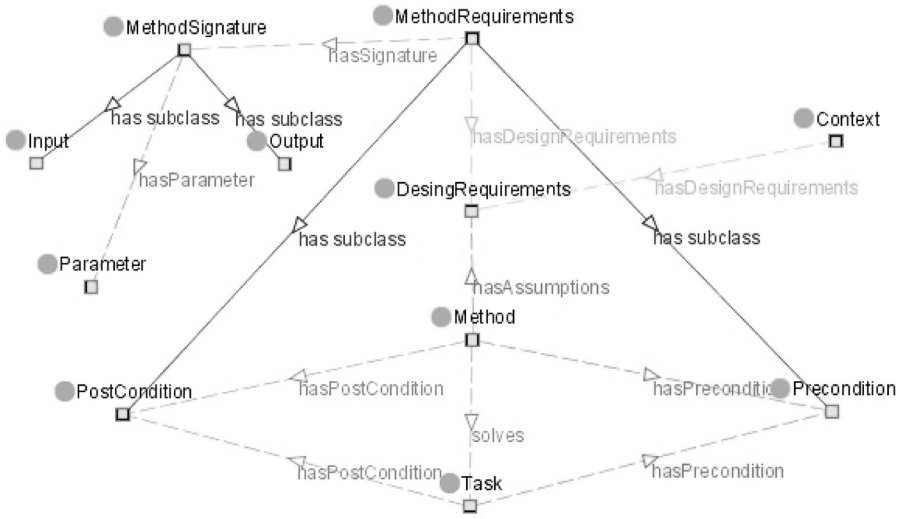

Figure 8 Methods composition ontology

represented as inputs/outputs and preconditions/postconditions. The inputs and outputs represent the data parameters received and generated by the method (integers, booleans, strings, etc.). The pre/postconditions are defined using instances of CBROnto that represent conditions over the data managed by the application. For example, these instances could represent the structure of the case base or any other information about the cases.

Figure 8 shows that each method has a precondition and postcondition instance that are composed by a signature and some design requirements. The signature is composed by input and output instances that contain the parameters received and generated by the method. The design

requirements represent some conditions that the CBR application must complain to execute the method successfully. Finally, each method is linked with the task that solves through the *solves* property of the ontology.

Two methods can be composed if the output of the previous method is more specific than the input of the following one. For example, if the output of m_i is an ordered list and the input of the following method m_{i+1} is just a list we can compose them because the flow of data is compatible. On the other hand, if the output of m_i is a list and the input of m_{i+1} is an ordered list we could not compose these methods because the requirements of m_{i+1} are not satisfied. This checking can be performed using the classification capabilities of a DL reasoner. If the postcondition of m_i is classified in the ontology under the precondition of the following task m_{i+1} it means that the postcondition is more specific than the precondition $m_{i+1}.\text{pre} \subseteq m_i.\text{post}$, and therefore both methods could be executed sequentially.

Although we followed the OWL-S approach in the design of the templates representation, we have chosen the WSMO (Web services modeling ontology) approach for describing the methods. The WSMO ontology is used to describe web services in the WSMX architecture, which is the most important alternative to OWL-S when developing SWS applications. Note, that CBROnto is coded using OWL, although its concepts and their relationships and properties are adaptations of concepts taken both from the OWL-S or WSMO ontologies. In this way, we take concepts from the OWL-S ontology to represent the structure of the template (as explained in Section 3) but we reuse a part of the WSMO ontology to represent method signatures. Thus, we integrate both approaches and keep in a neutral and compatible position with both standards.

In WSMO a service (or method in our case) is represented with the input, output, pre/post-conditions and some other not-mandatory characteristics of the parameters. These other characteristics are not required to execute the method but are a kind of recommendations. For example, in our CBR scenario, we have some retrieval methods that may take a long execution time when applied to large case bases, so we include a 'recommendation' in their description that suggests that these methods work better with small case bases. These recommendations are stored as design requirements using the *hasAssumptions* property (Figure 8).

Although we have explained how to check the compatibility of two consecutive methods, we must extend the algorithm because a method will require data generated by a previous method that was not executed immediately before. In some way, we need to represent a kind of blackboard where methods store their output. Therefore, the precondition of a method must be compatible with the information of this blackboard.

The blackboard is represented in CBROnto by means of the *Context* concept. Instances of this concept will contain the design requirements and parameters returned by previously assigned methods. Initially the context will contain some design requirements defined by the user: case structure, case base organization, etc. As new methods are assigned to solve a task, their output and postcondition will be added to the context.

Finally, our algorithm to choose the proper method to solve a task is the following one:

1. Select the methods that could solve a task (following the *solves* property).
2. For each selected method, check if the context is classified under its precondition. If it is, add the method to ApplicableList.
3. Rank methods of ApplicableList according to the optional design requirements. Top methods will complain more optional design requirements.
4. User selects a method from the ranked methods list and assigns it to the task.
5. Update context with the method postcondition.

5.1 System adaptation example

Now let us continue with the adaptation of the retrieved template in Section 4.1. In this example, the user had chosen a single-shot template shown in Figure 6 (left).

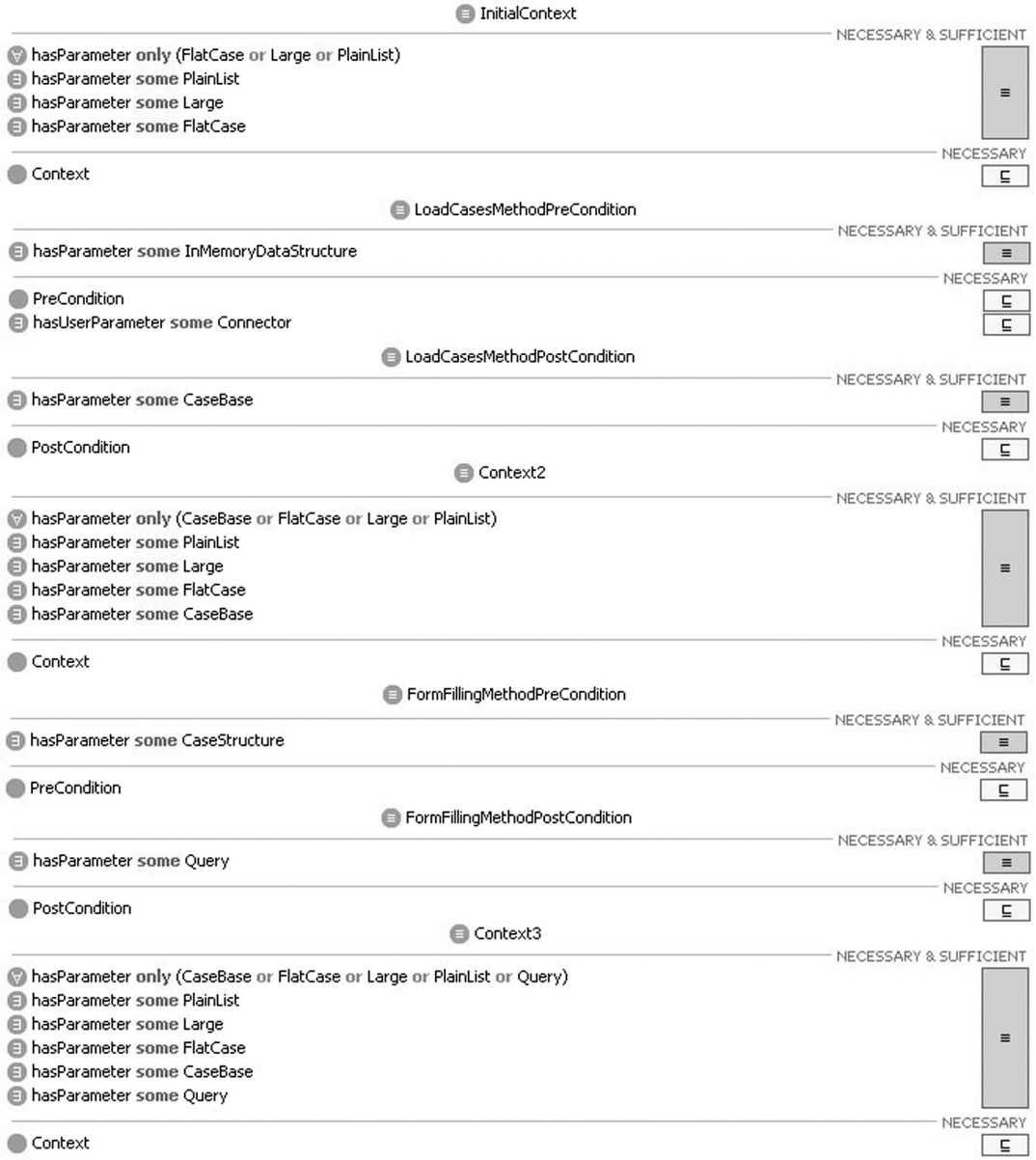

Figure 9 Compatibility checking example

In this step, the tool proposes which methods can be applied to solve each task of the system. Therefore, the user can substitute the method that currently solves a task by another one.

Before starting with the adaptation, the user has to define some features of the application *context*. In this case, the user configures the application to work with a small case base, organized using a plain list, and containing flat cases. The retrieved system was a collaborative one and its context contained a case base organized using a Pearson Matrix (common organization of the case base for collaborative recommendations) instead of a plain list. This modification of the context restricts the methods applicable to solve the tasks. The definition of the initial context is shown on the top of Figure 9.

At this point, the system begins recommending methods for the first task of the system. This task is LoadCases (it appears in the preCycle of the template) and is solved by the LoadCasesMethod. The tool obtains a list of methods able to solve this task by means of the *solves* property. In this case, there are no more methods than LoadCasesMethod. Its definition is shown in Figure 9 below the definition of the initial context. The precondition of this method contains

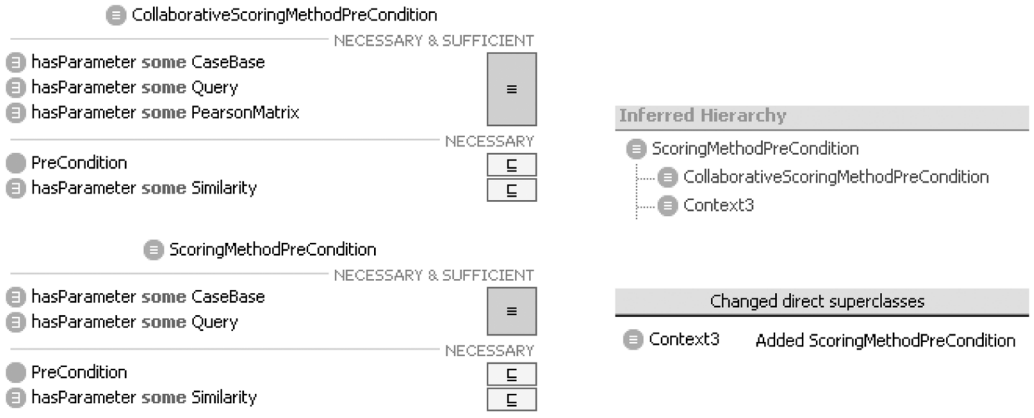

Figure 10 Compatibility checking example (continuation)

two conditions. The first one is in the necessary and sufficient block requiring a *InMemoryDataStructure* to be executed. As the initial context contains the property *hasParameter some PlainList* and PlainList is a subclass of InMemoryDataStructure the method can be assigned to the task. The other condition is *hasUserParameter some Connector* and indicates that the user must provide that parameter (a connector) to execute the method. To check that the method can be executed, we invoke a DL reasoner to classify the precondition of the method and the context. The method can be assigned if the context is classified under the precondition.

Once LoadCasesMethod is assigned to solve the LoadCases task, its postcondition is added to the context. Therefore, we obtain a new context (here named *Context2*) that also contains a *CaseBase* parameter (continue reading Figure 9 for details).

The following task is FormFilling that can be solved by two methods: FormFillingMethod and FormFillingWithInitialValuesMethod. Figure 9 only shows the description of the first one. In the precondition it requires a CaseStructure. As *Context2* contains *hasParameter some FlatCase* and FlatCase is a subclass of CaseStructure the method is compatible with the context and can be assigned to the task. Again, the postcondition is added to the context to obtain a new one (*Context3*).

The following task is *Scoring* and it is solved using the *CollaborativeScoring* method in system #2. The library also includes another method to solve this task: *NNScoring*. The user wants to create a system that uses this method instead of the collaborative one. CollaborativeScoring requires a Pearson Matrix case base organization to be executed. It is not present in the context because the user selected a plain list organization in the initial preferences. As Figure 10 shows, the NNScoring method is compatible with this organization but the CollaborativeScoring method is not. In the inferred hierarchy shown in this figure, Context3 is a subconcept of the NNScoring precondition but it is not a subconcept of the CollaborativeScoring one. Thus, the system only allows to assign the *NNScoring* method to solve the Scoring task.

This schema is repeated for each remaining task. The following is SelectCase, and there are many methods that can substitute the assigned one: *SelectAllCases*, *SelectTopK*, *GreedySelection*, *BoundedGreedySelection*, *DiversitySelection*… . All of them will be presented to the user as possible candidates to solve the task. However, the GreedySelection has an assumption to indicate that the method should be used only with small case bases. As our context contains *hasParameter some Large* and Large is a subclass of CaseBase, the assumption is failed. To reflect this, our tool recommends this method in the last position.

Finally, the system proposes methods to solve the Display task in a similar way than previous steps.

6 On-field evaluation

Before starting this work, our initial hypothesis was that the conceptualization of system behavior into templates could serve to ease the development process. Therefore, after developing the

recommender's templates we tested experimentally the viability of this approach. To achieve this goal we used a group of 50 students of an Artificial Intelligence and Knowledge Based Systems course at the Complutense University of Madrid, and proposed them to:

1. Design a recommender system.
2. Choose a development process: independent (without jCOLIBRI), composing manually the methods included in the framework (we will call this approach *method-based design*), or using the templates (*template-based design*).
3. Implement the recommender and provide feedback about its development.

As we were interested on measuring the design process quality instead of performance issues like accuracy or dialogue length, we used several surveys to obtain this information. Surveys served to compare the different approaches for composing software: independent, method-based and template-based. There were also other control variables about recommenders development that we measured using the surveys, mainly if the jCOLIBRI's templates could cover the huge range of designs proposed by the students and if the reuse of templates could ease the development process.

The experiment was split into several lessons where students were progressively introduced to the recommenders domain and its development. Firstly, the basis of recommenders development was exposed in a theoretical lesson showing several examples of real systems. During this lesson we did not use the templates to illustrate the behavior and structure of recommender systems. In a later lesson students had to define the requirements and design of their recommenders. Here students could freely choose the domain, the type of interaction, the type of similarity, if the system was collaborative or not, and any other issues. The following lesson introduced jCOLIBRI's templates and we conducted a survey to measure how close were these templates to the recommenders designed by the students. Then, we invited them to implement their design by themselves, composing manually the methods in jCOLIBRI, or reusing the templates. After the development process, a final survey served to confirm the development process followed by the students (independent, methods-based or templates-based), the general opinion about templates and several other features that are out of the scope of this paper.

In a general way, the experiment was very successful although we did not force them to use jCOLIBRI's templates. Students developed 25 recommenders for a wide variety of domains, and surveys showed that all of them chosen to use jCOLIBRI (62.5%) followed the template-based design and the remaining followed method-based approach (Figure 11, left).

After reviewing the recommenders implemented by the students we noticed that the election of the design process had not a clear explanation. Some students chose the template-based design and made several modifications to the examples, obtaining unusual recommenders. On the other

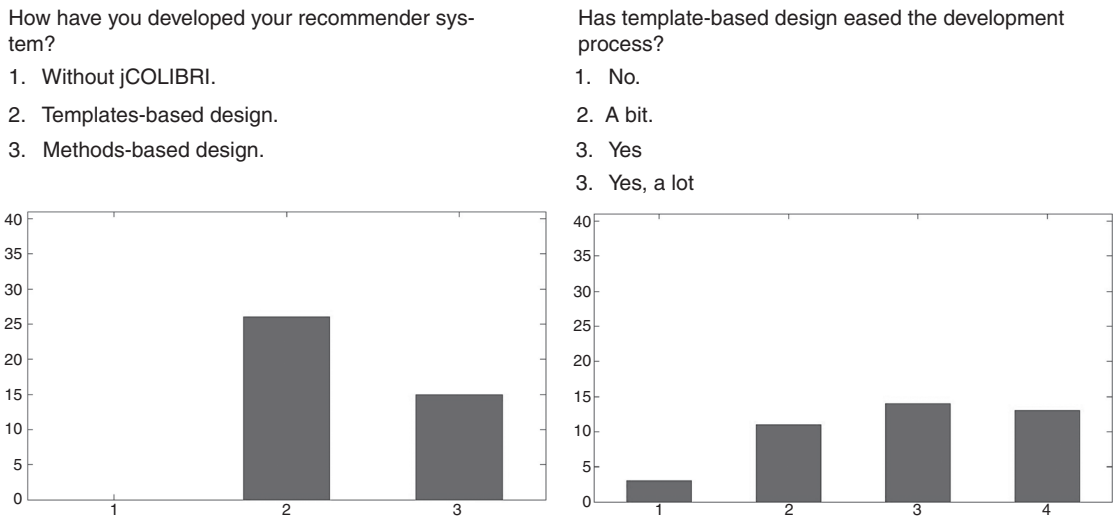

Figure 11 Student's opinion about template-based design

hand, students that followed the method-based design created systems that were also close to the examples included in the framework.

Although the selection of the design process was not clear, our final impression is that templates were always very close to the student's designs. Some of them used directly the templates-based approach and other chose the method-based design although implementing a system similar to one or several templates. This conclusion was confirmed by the survey where we measured the coverage of the templates. Here, students had to compare their designs to the templates before they received the introduction to jCOLIBRI in the second lesson. In average, the maximum similarity given to the closest template was 8.25. It means that at least one template was very similar to the student's original designs and that the coverage of templates was satisfactory.

Regarding the evaluation of the template-based design, the group of students that chose this approach reported a good (26%) or a very good opinion (25%). The remaining students thought that there was no improvement with respect to the manual composition. Note that these students could have chosen the methods-based design after testing the templates approach. These results are shown in Figure 11 (right).

7 Related work and conclusions

The work presented here relates to different areas of research. As an approach to let non-developers to build variations of previously developed software systems it connects to computational workflows as applied to scientific applications.

Workflows have emerged as an effective paradigm to manage large-scope scientific analyses. Computational workflows (Gil *et al*., 2007) are composed of software components that can be submitted for execution to several alternative execution resources (ranging from single-host to cluster platforms), process large-scale datasets and can be easily restructured to exploit parallel data processing. Existing workflow systems (Deelman *et al*., 2005; Oinn *et al*., 2006) have been demonstrated in a variety of scientific applications where workflow creation draws from catalogs of hundreds of distributed software components and data sources, where the generation of workflows of thousands of interrelated computing processes is largely automated, and where the execution of workflows takes place on high-end computing resources.

One of the goals of research on computational workflows is to close the gap between the end user, the scientist and the grid resources, by allowing the user to discover, reuse and repurpose previously designed workflows. Within the *my*Grid project[2], a workflow lifecycle has been proposed that extends beyond execution, to include discovery of previous relevant designs, reuse of those designs and subsequent publication (Wroe *et al*., 2007). They also propose algorithms for retrieving workflows based on requests specifying structural properties of the workflows. Given a partial workflow specification, find those workflows in the library that have the components included in the request connected in the same way. A first solution using role specialization in OWL Lite allowed only to consider sequential composition of components in the workflow (Goderis *et al*., 2005). More recently, Goderis and his colleagues have reported good results applying a technique for graph sub-isomorphism matching optimized to work over a repository of graphs (Goderis *et al*., 2006). Nevertheless, the solution to the problem of structure-based workflow retrieval is essentially hard since it is a problem that can be reduced to that of subgraph isomorphism, which is known to be an NP-complete problem (Gupta & Nishimura, 1996).

Comparing to computational workflow systems our approach for system representation does not explicitly represents data flow. Therefore, we look to provide a larger design space by not constraining possible adaptations to those preserving the given data flow. On the other hand, this extra flexibility requires the developer to write some 'glue-code' to fill the gaps after adapting a system. Regarding retrieval, we propose a similarity-based approach, typical in CBR systems, that ranks results according

[2] http://www.mygrid.org.uk/

to a computed similarity measure. This approach gives the user more freedom to explore the system library than the retrieval based on an exact match with the query used in workflow systems.

Through the explicit representation of variations of a family of related software systems and the connection of those variations with actual software components we also relate to work on domain and product line engineering. Domain engineering is a software reuse approach that focuses on a particular application domain. In domain engineering, we perform domain analysis and capture domain knowledge in the form of reusable software assets, which forms a basis for software product line practices (Kang *et al.*, 2002). Product line engineering (Sugumaran *et al.*, 2006) has been developed as a systematic approach to build product variants using product line assets including common architectures and components. These architectures and components are built with variation points based on the commonalities and variabilities of the products in the application domain.

Feature modeling (Czarnecki & Eisenecker, 2000) plays an important role in domain engineering. Features are prominent and distinctive user visible characteristic of a system. Systems in a domain share common features and also differ in certain features. In feature modeling, we identify the common and variant features and capture them as a graphical feature diagram. Although usually there is no reasoning support for feature models, Wang *et al.* (2007) present an approach to modeling and verifying feature diagrams using Semantic Web OWL ontologies that is related to the representation of template constraints in jCOLIBRI. They use OWL DL ontologies to precisely capture the inter-relationships among the features in a feature diagram, and they use FaCT++, as OWL reasoning engine, to check for the inconsistencies of feature configurations fully automatically.

The main difference between product line engineering and the semantic template approach to system construction presented here can be stated as the difference between deductive and inductive methods. Software product lines impose a deductive top-down approach where all possible feature combinations are identified and related to reusable software assets with the goal of describing every possible system that can be built with them. On the other hand, our templates are inductively created from implemented systems and only serve to generate variations of the available systems. Although more powerful in principle, the product line approach requires a much bigger knowledge acquisition effort which actually is the main drawback of this approach to software development.

Finally, the model-driven architecture (MDA)[3] as defined by the OMG consortium shares with the approach presented here the use of different models of a software system with different levels of abstraction. MDA proposes the use of different models for a software system along with a number of transformations going from more abstract models into more specific ones. In MDA terms, its goal is to separate business and application logic, which tends to be more stable, from the underlying platform technology, which may evolve more quickly due to technological evolution. Again, MDA is in principle a more powerful approach to the one proposed here but also requires a huge domain analysis effort that explains why it has been mainly applied to the more repetitive and best understood details of large scale distributed Web-based business applications.

In conclusion, we have presented an approach to build software systems from reusable components in a case-based way. This approach seeks a balance between the automation of computational workflow systems and the detailed but expensive models of software product lines and MDA, using descriptive instead of prescriptive models (Aßmann *et al.*, 2006). Software construction tools in jCOLIBRI have been warmly received in the community where some recent efforts have been dedicated to the construction of a family of applications and the modeling of the corresponding templates in the domain of textual CBR.

References

Aßmann, U., Zschaler, S. & Wagner, G. 2006. Ontologies, meta-models, and the model-driven paradigm. In *Ontologies for Software Engineering and Software Technology*, Calero, C., Ruiz, F. & Piattini, M. (eds). Springer, 249–273.

[3] http://www.omg.org/mda/

Bergmann, R. 2002. *Experience Management: Foundations, Development Methodology, and Internet-Based Applications*. Springer-Verlag.

Bridge, D., Göker, M., McGinty, L. & Smyth, B. 2006. Case-based recommender systems. *The Knowledge Engineering Review* **23**(3), 315–320.

Burke, R. 2002. Interactive critiquing for catalog navigation in e-commerce. *The Knowledge Engineering Review* **18**, 245–267.

Czarnecki, K. & Eisenecker, U. 2000. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley.

Deelman, E., Singh, G., Su, M., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C. & Katz, D. S. 2005. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal* **13**(3), 219–237.

Díaz-Agudo, B. & González-Calero, P. A. 2002. CBROnto: a task/method ontology for CBR. In *Proceedings of the 15th International FLAIRS'02 Conference (Special Track on CBR)*. AAAI Press, 101–106.

Díaz-Agudo, B., González-Calero, P. A., Recio-García, J. A. & Sánchez-Ruiz-Granados, A. 2007. Building CBR systems with jCOLIBRI. *Journal of Science of Computer Programming* **69**, 68–75.

Fernández-Chamizo, C., González-Calero, P. A., Gómez-Albarrán, M. & Hernández-Yáñez, L. 1996. Supporting object reuse through case-based reasoning. In *Advances in Case-Based-Reasoning (EWCBR96)*, Lecture Notes in Artificial Intelligence, **1168**, 135–149. Springer.

Frakes, W. B. & Kang, K. 2005. Software reuse research: status and future. *IEEE Transactions on Software Engineering* **31**(7), 529–536.

Gil, Y., González-Calero, P. A. & Deelman, E. 2007. On the black art of designing computational workflows. In *Proceedings of the 2nd Workshop on Workflows in Support of Large-Scale Science*. WORKS '07. ACMNew York, NY, 53–62.

Goderis, A., Sattler, U. & Goble, C. A. 2005. Applying description logics for workflow reuse and repurposing. In *Proceedings of Description Logics 2005*. Edinburgh, Scotland, UK.

Goderis, A., Li, P. & Goble, C. A. 2006. Workflow discovery: the problem, a case study from e-science and a graph-based solution. In *Proceedings of the International Conference on Web Services 2006*. IEEE Computer Society, 312–319.

Gupta, A. & Nishimura, N. 1996. The complexity of subgraph isomorphism for classes of partial k-trees. *Theoretical Computer Science* **164**(1–2), 287–298.

Hammond, K. J., Burke, R. & Schmitt, K. 1996. A case-based approach to knowledge navigation. In *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, Leake, D. B. (ed.). 125–136, AAAI Press.

Johnson, R. E. & Foote, B. 1988. Designing reusable classes. *Journal of Object-Oriented Programming* **1**(5), 22–35.

Kang, K. C., Lee, J. & Donohoe, P. 2002. Feature-oriented product line engineering. *IEEE Software* **9**, 58–65.

Kolodner, J. L. 1993. *Case-Based Reasoning*. Morgan Kaufmann.

Leake, D. B. 1996. *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. AAAI Press.

McSherry, D. 2002. Recommendation engineering. In *Proceedings of the 15th European Conference on Artificial Intelligence*, van Harmelen, F. (ed.). 86–90. IOS Press.

Oinn, T., Greenwood, M., Addis, M., Alpdemir, M. N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M. R., Senger, M., Stevens, R., Wipat, A. & Wroe, C. 2006. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience* **18**(10), 1067–1100.

Recio-García, J. A., Sánchez, A., Díaz-Agudo, B. & González-Calero, P. A. 2006. Lessons learnt in the development of a CBR framework. In *Proceedings of the 11th UK Workshop on Case-Based Reasoning*. CMS Press, University of Greenwich.

Recio-García, J. A., Díaz-Agudo, B., Bridge, D. & González-Calero, P. A. 2007. Semantic templates for designing recommender systems. In *Proceedings of the 12th UK Workshop on Case-Based Reasoning*. CMS Press, University of Greenwich.

Recio-García, J. A., Bridge, B., Díaz-Agudo, D. & González-Calero, P. A. 2008. CBR for CBR: a case-based template recommender system. In *Advances in Case-Based Reasoning, 9th European Conference, ECCBR 2008*, Althoff, K.-D. & Bergmann, R. (eds). Lecture Notes in Computer Science **5239**, 459–473. Springer.

Shimazu, H. 2002. ExpertClerk: a conversational case-based reasoning tool for developing salesclerk agents in e-commerce webshops. *Artificial Intelligence Review* **18**(3–4), 223–244.

Smyth, B. 2007. Case-based recommendation. In *The Adaptive Web*, Brusilovsky, P., Kobsa, A. & Nejdl, W. (eds). Lecture Notes in Computer Science **4321**, 342–376. Springer.

Smyth, B. & Cotter, P. 1999. Surfing the digital wave: generating personalised TV listings using collaborative, case-based recommendation. In *Proceedings of the 3rd International Conference on Case-Based Reasoning*, Althoff, K. D., Bergmann, R. & Branting, L. K. (eds). Springer, 561–571.

Sugumaran, V., Park, S. & Kang, K. C. 2006. Software product line engineering. *Communications of the ACM* **49**(12), 28–32.

Wang, H. H., Li, Y. F., Sunc, J., Zhang, H. & Pan, J. 2007. Verifying feature models using OWL. *Web Semantics: Science, Services and Agents on the World Wide Web* **5**, 117–129.

Wilke, W., Lenz, M. & Wess, S. 1998. *Intelligent Sales Support with CBR Case-Based Reasoning Technology, From Foundations to Applications*. Springer-Verlag, 91–114.

Wroe, C., Goble, C., Goderis, A., Lord, P., Miles, S., Papay, J., Alper, P. & Moreau, L. 2007. Recycling workflows and services through discovery and reuse. *Concurrency and Computation: Practice and Experience* **19**(2), 181–194.