

Projecte de Tesi
Towards a Component-based Platform
for Developing Case-based Reasoning
Systems

presentat per
Jose María Abásolo

Programa de Doctorat en Intel·ligència Artificial
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

Directors: Enric Plaza i Josep-Lluís Arcos
Tutor: Ton Sales

Realitzat a l'Institut d'Investigació en Intel·ligència Artificial
Bellaterra, de 2004

Contents

1 Objectives and Motivation	2
2 State of the art	3
2.1 Case Based Reasoning	3
2.1.1 General notions	3
2.1.2 Methodologies for developing CBR applications	6
2.1.3 Applications for developing CBR applications	8
2.2 Knowledge Modelling	9
3 Framework for CBR components	14
3.1 A Component Description Language	15
3.2 The CAT-CBR platform	19
3.2.1 CAT-CBR processes	19
3.2.2 CAT-CBR Repositories	20
3.3 A general overview of the CAT-CBR platform development process	22
3.3.1 Configuring process	23
3.3.2 Enabling process	24
3.3.3 Enacting process	25
4 CAT-CBR Library	25
4.1 CBR Ontology	26
4.2 Tasks and PSMs in the CBR Library	28
4.2.1 Retrieve Task	29
4.2.2 Reuse Task	33
4.2.3 Retain Task	35
4.2.4 Tasks and methods for acquiring knowledge	36
5 An example of using CAT-CBR in a musical domain	38
5.1 Presenting and characterizing the domain problem	38
5.2 Configuring CBR systems with CAT-CBR	40
5.3 Enabling CBR systems	40
5.4 Enacting and Testing CBR systems	41
6 Conclusions and future work	43
A Publications	45
B Configuring process in detail	45
B.1 Elements for the configuring process	46
B.2 A general overview of configuring process	47
B.3 Three configuring strategies	48
B.4 An example of configuring a CBR application	51

1 Objectives and Motivation

There are two main goals in our Ph.D. work. First, to define a framework of components that allows to specify CBR methods at an abstract level of description. Second, to present a platform or environment that uses the previous described methods and helps the engineer in developing CBR systems. To achieve these goals we have followed a Knowledge Modelling approach.

Knowledge Systems are computer systems that deal with complex problems using knowledge. Knowledge may be acquired from humans or automatically derived using abductive, deductive, and inductive techniques. Knowledge Modelling community has mainly focused on research in knowledge engineering and has led to the development of Application Development Frameworks, specification languages for Knowledge Systems, libraries of reusable components and key modelling technologies, such as ontologies (reusable terminologies) and problem solving methods (reusable reasoning strategies).

Case Based Reasoning (CBR) is an AI approach for learning and solving problems from past experiences. This approach emerges for solving the acquiring knowledge problem associated with the traditional Knowledge Systems. These Knowledge Systems deal with a problem by acquiring some models about the domain of the problem; this acquiring knowledge process has a high cost due to it needs a lot of information and time dedication by the experts. CBR does not try to acquire directly this knowledge, but it uses past situations of the domain problem in order to solve new ones.

As it was presented in [2] the CBR process can be seen as a cycle of four steps: *Retrieve* (that deals with the problem of finding similar past cases to the new situation), *Reuse* (that deals with adapting the solution of past cases for solving the new situation), *Revise* (that determines whether the solution given to the new situation is correct), and *Retain* (that decides whether the situation and solution must be learned in order to be used in the future).

The work that we will present has two motivations that come up from two different kinds of engineers: application engineers and scientific engineers. On the one hand, Application engineers use the CBR techniques and want to apply them in some specific domain; this kind of engineers do not know in depth all the characteristics of the different methods for CBR and have the problem of selecting them without knowing which ones better fit to their domain. On the other hand, scientific engineers develop CBR techniques or methods and their problem consists on being able to reuse them in different domains and testing their methods against others to understand the differences of the different methods. All the problems of these kind of users can be summarized in: first, to know which techniques are going to be used in each one of the CBR cycle phases; and second, once the CBR system has been developed, to reuse it easily in another domain.

Solving these two problems is the main motivation of our work. What will be presented next is a Components Based Platform devoted to develop CBR systems. This platform will give the application engineer the tools to easy develop a CBR system for his domain; the platform also provides a framework for the scientific engineer in order to be able to reuse and specify his techniques. Within this platform there will be a Library of CBR components where some of the techniques used in the CBR cycle are specified. A Component Description Language (CDL) has been developed in order to specify the Library of CBR

components; this (CDL) allows the different components from the library to be described in a domain independent way, what gives these components the characteristic of being domain independent. In order to work with this library and develop effectively an application, the platform will be presented in three stages: Configuring (in which the components to use in the desired application will be chosen); Enabling (in which an application will be connected to a specific domain); and Enacting (in which the chosen techniques together with the domain, are operationalized obtaining an executable application). The Enabling stage will allow reusing a CBR system in a simple way, in the sense that the same configuration can be used in different domains only changing the Enabling stage.

The rest of the document is structured as follows. Section 2 describes the state of the art in CBR and Knowledge Modelling; concerning knowledge modelling, we will present some traditional approaches as well as UPML a framework that follows the Task-Method-Decomposition approach in order to model the knowledge. Then, in section 3 first, the goals and motivation of our work will be presented; at the same time, it will also be presented a Component Description Language and a general overview of the platform (CAT-CBR) we have developed in order to achieve our goals. In section 4 we present the methods that constitute the CAT-CBR Library. In section 5 we present an example of domain and how the CAT-CBR platform can be used in order to develop CBR systems in this domain; the chosen domain is a musical domain where the goal is to classify discrete percussion sounds, this domain belongs to the *Tabasco* project where this research is currently being done. Finally, in section 6 we present some conclusions of this document and future work. We also include three appendixes: appendix A presenting the publications done during this project, appendix B where we present the configuring process in detail.

2 State of the art

In this section we briefly present the basic concepts of the two principal elements developed in our work. Firstly some basic notions about Case Base Reasoning; secondly we present the Knowledge modelling framework and different traditional approaches; from all the traditional approaches we will specially focus on Component of Expertise [22] and UPML that are our starting point in order to develop a Component Description Language.

2.1 Case Based Reasoning

We will present Case Based Reasoning approach from three different points; first we will present the general notions of CBR, second we will describe some methodologies for developing a CBR system, and finally we will present some commercial tools for developing CBR systems.

2.1.1 General notions

Case Based Reasoning (CBR) is an approach for solving problems and learning in AI. CBR approach is different from other approaches in AI, instead of general domain models that represent the problem domain, as Knowledge Systems, or generalizing the relations between problem and solutions, as induction

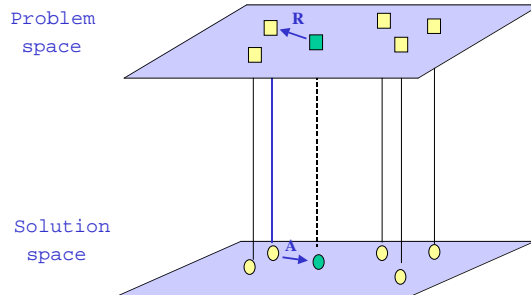


Figure 1: Relation between similarity in problem space and solution space. When a case (yellow boxes) in Problem Space is near (R distance or similarity) to the new problem (green box); then the solution of the previous case (yellow balls) will be near (A distance or similarity) to the new solution (green ball).

techniques; CBR uses the specific knowledge obtained from situations previously solved (cases) and domain models if it is necessary. With this approach CBR tries to solve the traditional problem of knowledge acquisition, because CBR system only needs a set of solved problems to start up. Some CBR systems, Knowledge-Intensive CBR, uses cases and acquired knowledge to solve problems.

A CBR system needs a collection of experiences, called cases, stored in a case base, where each case is usually described by the problem description and the solution applied to this problem. CBR takes two hypothesis concerning world's nature [17]:

- Regularity. It means that, from similar situations, we can get similar conclusions. As a consequence, the conclusions or solutions associated with a situation can be the base for a new situation.
- Recurrence. It is highly probable that the future situations are variations of the present ones.

Accepting these basic hypotheses of “similar problems have similar solutions” and “new problems are similar to problems previously solved”, case based reasoning focuses on two kinds of similarities. These two types of similarities are applied in different spaces, the description problem space and the solution space (see figure 1).

The process model of CBR (also called the “CBR cycle” fig. 2) is the following:

1. *Retrieve*. The goal of Retrieve step is, given a new case to recover from the precedent cases those cases that are similar to the new one.
2. *Reuse*. The goal of Reuse step is, given a set of cases similar to the new

Case-based reasoning

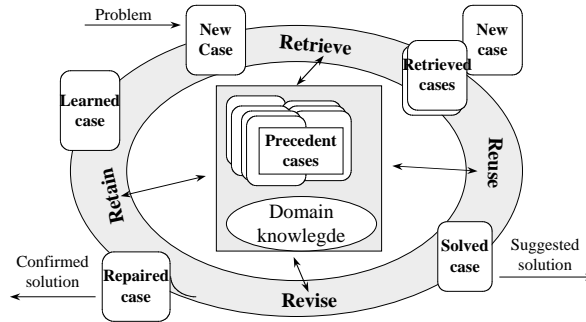


Figure 2: The CBR cycle [2].

case, to reuse their solutions in order to obtain a new solution for the new case.

3. *Revise*. The goal of Revise step is to determine whether the solution obtained is correct and, if it is incorrect to repair it into a correct solution.
4. *Retain*. The goal of the Retain step is to decide if the new case and the solution obtained must be included in the precedent cases in order to be used for solving new cases in the future.

The way in which cases are represented, how the similarity between cases is defined, how the solution is adapted, when the cases have to be retained, are questions that are interrelated.

Kolodner in [17] differentiates between two groups of CBR applications: analytic task and synthesis task. In analytic tasks the fundamental aspect is to decide whether a new situation can or cannot be treated as previous situations, based on the similarities between them. In synthetic tasks, the objective is to build a solution for the new problem adapting the solutions of the previous cases.

After these two groups of applications, Althoff presented a classification hierarchy of CBR applications (see figure 3). We can differentiate five subclasses inside analytic (classification) tasks: *Diagnosis* (e.g. Medical diagnosis or equipment failure diagnosis), *Prediction* (e.g. the forecasting of equipment failure or stock market performance), *Assessment* (e.g. risk analysis for banking, insurance or the estimation of project costs), *Process control* (e.g. the control of manufacturing equipment) and *Planning* (e.g. the reuse of travel plans or work schedules). In synthesis tasks, we can differentiate the following subclasses: *Design* (i.e. the creation of a new artifact by adapting elements of previous ones), *Planning* (i.e. the creation of new plans from elements of old ones) and *Scheduling* (i.e. the creation of new schedules from old schedules).

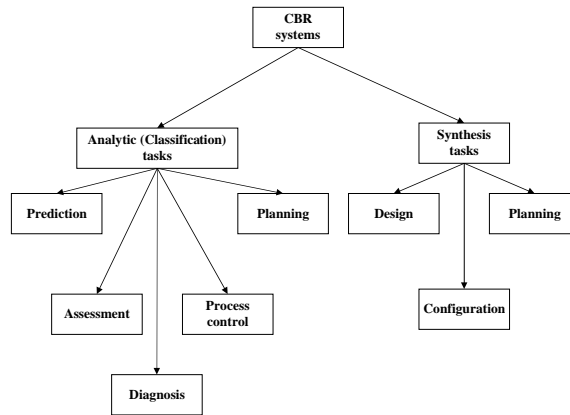


Figure 3: A classification hierarchy of CBR applications.

2.1.2 Methodologies for developing CBR applications

As developing CBR applications is a complex task, there have been some attempts to help in this developing process. Here we present two approaches for helping in this process; first we present the INRECA methodology [8], and then the CBR-PEB project [3].

INRECA methodology The INRECA methodology combines a number of methods for CBR and inductive learning through a philosophy that addresses a number of phases in the software development life cycle. The INRECA methodology provides the guidelines for the activities that need to be performed in order to successfully develop CBR software systems. These guidelines must be expressed using a well-defined terminology. In order to be suitable, a methodology like INRECA should provide significant benefits in terms of productivity, quality, communication and management decision making.

When developing a CBR application, the developer must consider a large variety of different kinds of processes. In the INRECA methodology a six step cycle is presented as a guideline for developing and improving applications. These six steps are:

- **Characterize.** The aim of this step is to characterize the project and its environment based on the available information. The factors used to characterize a project may be application domain, susceptibility to changes, problem constraints, techniques, tools, programming language and so on. This step provides a context for goal definition and for selecting reusable experiences.
- **Set Goals.** The goals of the project must be defined. There are a variety of viewpoints for defining goals, like user's viewpoints, customer, corporation and so on.
- **Choose Process.** On the basis of the characterization, the goals, and the previous experience, appropriate processes for implementing the project must be chosen. This results in the overall project plan.

- **Execute.** The project plan is enacted, causing the development project to be carried out.
- **Analyze.** At the end of each specific project, the data collected must be analyzed to evaluate the current practices.
- **Package.** All the information in the previous steps must be packaged in models in order to be reused in future.

The basic philosophy behind the INRECA methodology is the experience-based construction of CBR applications. This experience base is organized on three levels of abstraction: common generic level, cookbook level, and specific project level.

Common Generic level. At this level, processes, products and methods are collected in groups that are common for a very large spectrum of different CBR applications.

Cookbook level. At this level, processes, products, and methods are tailored for a particular class of application (e.g., help desk, planning, diagnosis). For each application class, the cookbook level contains a so-called *recipe*.

Specific project level. The specific project level describes experience in the context of a single, particular project that has already been carried out.

This experience base is used to develop new projects in the following way:

1. **Characterize the New CBR project:** identify the application area of the new CBR project. The goal is to decide whether an existing recipe from the cookbook level of the methodology covers this application.
2. **Recipe Available:** identify whether an appropriate recipe is available. If this is the case, then continue with step 3, otherwise continue with step 7.
3. **Analyze Processes from Recipe:** the process model contained in this recipe must now be analyzed in order to see whether it can be mapped and tailored to the application at hand.
4. **Select Similar Specific Project:** Analyze the project descriptions on the specific project level. If a similar project is available then it should be identified as if some of the specific experience can be reused within the scope of the current project. This might be a specific way to implement a certain process or software component. It can help to identify existing software components from previous projects that can be reused immediately on the code level.
5. **Develop New Project Plan by Reuse.** Develop a project plan for the new project. This project plan is based mainly on a process model from the selected recipe. However, application-specific tailoring and pragmatic modifications are typically required.

6. **Analyze Processes from Common Generic Level.** The set of processes described at the common generic level should be analyzed in the context of the new project. The goal is to identify those processes that are important for the new application.
7. **Develop the Project Plan by Combining Processes.** Based on the selected processes, a new project plan must be assembled. For this purpose, the processes must be made more precise and operational.
8. **Enact the Project and Record Project Trace.** Execute the project by enacting the project plan. To document the expert during the enactment of this project. Particularly, note all deviations from the developed project plan. This is important to ensure that the development is performed according to the necessary quality standards, and to feedback the new expertise into the experience base for reuse.

CBR-PEB The CBR-Product Experience Base CBR-PEB is a decision support system for Case-Based Reasoning (CBR) system development (i.e., CBR applications and CBR tools).

The CBR-PEB uses the same idea of Expertise Base as the INRECA methodology. There are many successful CBR projects from which we can take experience and reuse components for new applications. For this purpose CBR-PEB store cases of these CBR systems in order to provide the engineer the information needed to develop a new CBR system.

CBR-PEB offers three different kinds of support:

- Support for feasibility study. It analyzes if there is a similar system already implemented.
- Decision support. It provides information about who has developed a system and the techniques used in the implementation.
- Support for a state-of-the-art study.

One of the problems of the CBR-PEB is the granularity of the CBR systems stored in the Expertise Base. Many of these systems are big applications with low modularity that unable the user to reuse these solutions in order to develop a new system.

2.1.3 Applications for developing CBR applications

Next, we are going to present briefly some of the commercial applications for developing CBR systems:

CBR-Works CBR-Works was developed by tecInno as a software framework for developing CBR applications. It uses an object oriented approach for representing the domain ontology and the cases. CBR-Works allows to represent structured cases, it also offers predefined similarity measures to the user and includes editors for defining new structured similarity measures. For adaptation, CBR-Works uses simple adaptation methods based on adaptation rules of the type “if condition then action”. CBR-Works has been used for developing different CBR applications, mainly e-commerce. CBR-Works gives support for

developing classification systems and manages efficiently big case bases although it uses an exhaustive algorithm for retrieve.

K-commerce K-commerce, developed by Inference Corporation, is a family of products used mainly for customer support help-desks. K-commerce is unable to handle structured cases, although it allows to link additional information to the cases, for example, sounds, pictures and videos. K-commerce does not manage well cases with many unknown values and does not allow to represent general knowledge about the domain. The system provides very quick retrieve methods (nearest neighbor with indexing structures) and allows to manage big case bases efficiently. K-commerce has a maintenance mode that allows to modify the similarity measures and the weight vectors. The system also incorporates learning methods (retain) to add new cases, but it does not use methods for reuse and revise.

KATE The KATE system (developed by AcknoSoft) uses a nearest neighbor algorithm for the retrieval step. During retrieval, the similarity measures and the weight vectors about the attributes can be personalized to better fit to specific needs of the application. KATE combines the nearest neighbor with dynamic induction; in this process each consult generates questions to the user that allow to discriminate between the cases of the case base. KATE has also a data mining module that acquires hidden knowledge about the cases, generating automatically decision trees from the cases.

ReMind ReMind was developed by Cognitive Systems Inc. It has two versions, first as a C library that can be added in other application, and second as a complete development environment. ReMind is unable to handle structured cases and the induction method used does not work correctly with cases with many unknown values. About retrieve step, ReMind offers different alternative methods: nearest neighbor, two kinds of induction (simple and knowledge driven) for generating decision trees, and traditional data base queries. ReMind adapts cases using formula and allows learning cases (retain). ReMind also explains why the cases have been retrieved.

ReCall ReCall was developed by Isoft (developed in C++) and combines nearest neighbor and inductive methods. ReCall uses an object-oriented description that allows a structured representation of domain knowledge, incomplete cases and uncertain knowledge. It allows communication with external applications and data bases. ReCall includes graphic editors to define objects, relation between them, taxonomies and adaptation rules.

2.2 Knowledge Modelling

The essential aspect of knowledge modelling, which distinguishes it from the broader area of knowledge representation, is to focus on knowledge - i.e. competence, at a level that abstracts from implementation-level structures. For instance, a knowledge model of an engineering design application will focus on the problem solving behavior and the knowledge required to solve the application - e.g., how design extensions and modifications are carried out, rather than

on implementation-level mechanisms for speeding up problem solving - e.g. how to optimize data encoding and retrieval. A knowledge modelling framework defines the basic organization of an approach to knowledge modelling. Knowledge modelling distinguishes the basic types of modelling components, their relations, and proposes a model development methodology. A modelling framework can be described in terms of three main aspects:

- **Types of Components.** A description of the different types of generic model structures proposed by framework.
- **Relations between Components.** A description of the main structuring relations between generic modelling components.
- **A Model Development Methodology.** A description, drawn either from empirical evidence or published guidelines, of the model development methodology associated with a modelling approach.

Below we will present briefly different knowledge modelling frameworks: *KADS/Common Kads*[25], *Components of Expertise*[22], *Generic Tasks*[9], *Protégé*[20] and UPML [11].

KADS/Common KADS The Common KADS framework distinguishes among three categories of knowledge which are “necessary and sufficient for the description of application-related knowledge”[25]: *tasks*, *inference*, and *domain*. Task knowledge describes both, what problem solving needs to be carried out in the domain and how. A task definition includes input, output, goal of the task and its task-subtask decomposition and control. Inference knowledge describes primitive reasoning steps, which specify the *leaves* of a task-subtask hierarchy. Domain knowledge “specifies form, structure, and contents of domain specific knowledge that is relevant for an application”[24]; it specifies both the application domain knowledge and the domain ontology.

The relations between components are defined by means of knowledge roles. A knowledge role is defined as “an abstract label that indicates the role that domain knowledge which the label is attached to, plays in an inference process” [21].

Components of Expertise The Components of Expertise (CoE) approach was proposed by Luc Steels (1990) with the aim of providing a comprehensive modelling framework. CoE approach distinguishes three basic components of expertise: tasks, methods, and domains¹. Tasks specify what needs to be solved; methods specify how to solve them; domain models specify viewpoints imposed by tasks models over an application domain.

For a given task there are, in general, a number of methods which can be used to solve it. In the CoE approach we can differentiate between conceptual and pragmatic features of a method. The former specify what a method can do; the latter make it possible to distinguish among different methods applicable to the same task.

¹This kind of approach having three types of components (tasks, methods and domains) is also used in other approaches like UPML and we refer to it as Task-Method-Domain (TMD) approach. We use the same approach in the definition of our Component Description Language described in detail in section 3.1

A model, in CoE, is developed through a task analysis process, in which a task model of the application is built by starting with a task defining a problem type, and then a task-method decomposition tree to select the best method for a particular task, until a complete task model for the application has been developed.

Generic Tasks The GT approach [9] is an example of the use-oriented view; a use-oriented view states that both the nature and the representation of the domain knowledge are determined by the particular task (or method) which is carried out. Therefore, only tasks and methods are taken into consideration in GT approach. Methods solve tasks either directly or by introducing new subtasks which are then in turn to be solved.

This approach does not consider task-independent domain models, thus limiting the possibility for reuse.

Protégé The Protégé approach differentiates tasks, methods, and domains. The tasks, in Protégé, are only described in terms of their inputs and outputs. Protégé also distinguishes between methods and mechanisms; the former decompose tasks into subtasks, and the latter are direct methods that solve a task. Methods are described in terms of their ontological requirements, input-output relation, control and data flow.

In order to maximize reuse, both method-independent domain models and domain-independent methods can be specified independently and integrated by means of an *application ontology* [13]. This integrates methods and domain ontology by introducing the required mapping relations.

UPML approach UPML (Uniform Problem-solving Methods description Language) was developed inside the IBROW project [11]. Problem-solving methods provide reusable architectures and components for implementing the reasoning part of knowledge systems. UPML aimed at describing and implementing such architectures and components to facilitate their semiautomatic reuse and adaptation easier.

UPML is not, strictly speaking, a language; UPML is a software architecture. Software architecture is the study of the large-scale structure and performance of software systems. Important aspects of a system's architecture include the division of functions among system modules, the means of communication between modules, and the representation of shared information. Additionally, design guidelines provide ways to develop a system constructed from the components and adapters that satisfy the constraints.

The basic structure of UPML is illustrated in figure 4. UPML defines four main knowledge components: Task, PSM, Domain Model, and Ontology. Ontology components are used by the other three components to define their universe of discourse that is represented by the bold arrows from the Ontology box in the figure. Each component can be modified to work in a certain context, through refiner adapters. The bridge adapters control the interactions between the Task, PSM and Domain Model components. Three types of bridges are present in UPML: Task-Domain Bridge, PSM-Domain Bridge and PSM-Task Bridge. In the figure the interactions are presented with arrows and the bridges are located on these arrows.

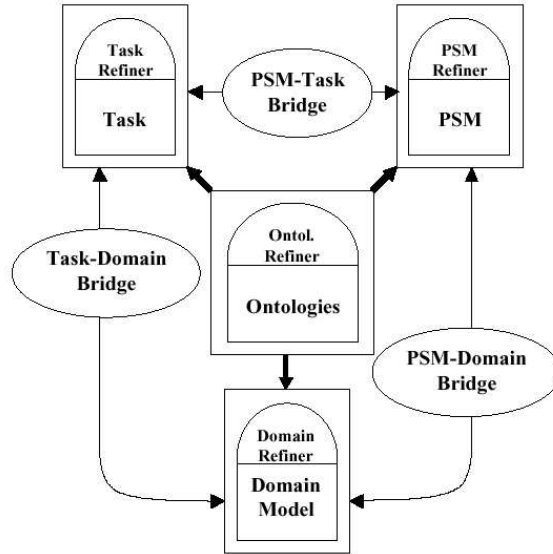


Figure 4: UPML architecture

UPML provides both a framework and a language to describe libraries of knowledge components and their relationships to form knowledge systems. The language relies on a meta-ontology that defines the concepts and relations that are then instantiated for specific knowledge components. This ontology is based on a clear top-level ontology consisting of concepts and binary relationships. The hierarchy of classes used to define UPML is presented in Figure 5.

There are four main concepts in UPML: *Ontology*, *Domain Model*, *Task*, and *PSM*. They all are subclasses of the concept *Knowledge Component*. Each component has a pragmatics description and relies on one or more ontologies that define its universe of discourse.

An *Ontology* defines a terminology and its properties, used by tasks, problem-solving methods, and domain models. The core of an ontology specification is its *signature* definition, which defines signature elements that hold terms. An ontology also provides the axioms that characterize logical properties of the signature elements. Additional theorems may list useful statements which are implied by the axioms. An Ontology may import other ontologies via the ontology attribute inherited from the Knowledge Component concept, and refine them with additional terminology and axioms.

A *Task* specifies the goal to be achieved by the PSMs of the library. The *input roles* and *output roles* together with the *competence* property define the input/output specification of the task. Input roles specify input of case data and output roles specify output of case data. Its *assumptions* property defines requirements on the knowledge that is used to define the goal. A Task can import and refine other tasks via its *uses* attribute.

A *Domain Model* introduces domain knowledge, merely the formulas that are then used by problem-solving methods and tasks. A Domain Model consists of three elements: *properties*, *meta-knowledge*, and *domain knowledge*. The Meta-knowledge captures the implicit and explicit assumptions made while building

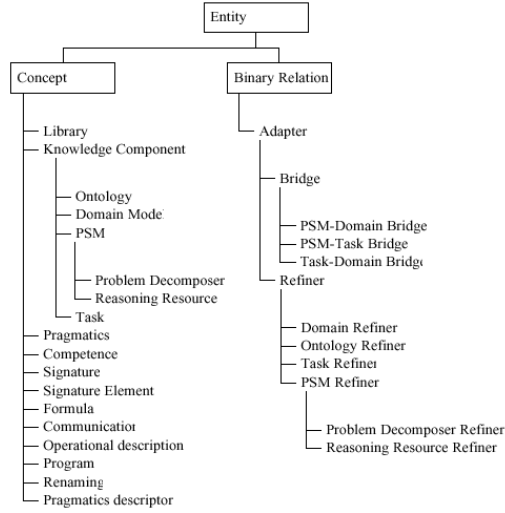


Figure 5: UPML hierarchy

a domain model of the real world. Meta-knowledge is assumed to be true, i.e. it has not been proved or cannot be proved. The domain knowledge is the knowledge base of the domain that is necessary to define the task in the given application domain and to carry out the inference steps of the chosen problem-solving method. The domain knowledge is built under the assumption that the meta-knowledge is true. Properties (the synonym for theorems) can be derived from the domain knowledge, they are visible to, and can be used directly to reason about the Domain Model.

A *PSM* represents a problem-solving method, merely defined by its *competence* and *communication* properties. The *input roles* and *output roles* of a PSM specify its inputs and outputs. Input roles specify input of case data and output roles specify output of case data. Its communication property describes the protocol of communications with the environment of the PSM, in particular other (PSM) components.

The Problem Solving Method (PSM) concept has the following two subclasses: *Problem Decomposer* and *Reasoning Resource*. A Problem Decomposer decomposes a task that has to be solved, into subtasks; its *operational description* specifies the control structure over the subtasks. A Reasoning Resource solves a subtask provided by a problem decomposer: It specifies the *assumptions* on the domain knowledge to perform a primitive reasoning step. It does not describe its internal structure, which is regarded as a simple implementational aspect of no interest for the architectural specification of the knowledge system. The *knowledge roles* attribute specifies the input of (domain) knowledge for the reasoning resource.

The interactions between UPML knowledge components are defined by Binary Relation. The root binary relation of UPML is *Adapter*. Like UPML specification components, an Adapter has pragmatics information and refers to specific ontologies. It also holds renamings correspondence between the terms

of both arguments. There are two subclasses of Adapter: *Bridge* and *Refiner*

A *Bridge* connects two Knowledge Components of different kinds. A Bridge defines mapping axioms and additional assumptions about the components that it relates. We introduce three subrelations of Bridge: *PSM-Domain Bridge* that connects a PSM with a Domain Model, *PSM-Task Bridge* that connects a PSM and a Task, and a *Task-Domain Bridge* that connects a Task and a Domain Model. Note that there are not defined bridges between the Ontology component and the other knowledge components, because this relationship is already part of the description of Task, PSM and Domain Model: Each has an inherited ontologies attribute that directly defines their universe of discourse.

A *Refiner* connects two knowledge components of the same type and is used to express the stepwise adaptation of components, e.g. the refinement of a task or a problem-solving method. Very generic problem-solving methods and tasks can be refined to more specific ones by applying a sequence of refiners to them. A Refiner assumes that its two attributes in and out are of the same type. This guarantees that a refiner modifies a given component, as opposed to mapping it to a different component via a Bridge relation. Each main UPML component has its own associated type of refiner. Thus, the Refiner relation has four component-specific subrelations: *Domain Refiner*, *Ontology Refiner*, *Task Refiner*, and *PSM Refiner*. The definition of a refiner holds in the attributes that are specific to each kind of component. Each refiner has its own restrictions on input and output: Domain Refiner contains redefined properties, meta-knowledge and knowledge in the refined component; similarly, Ontology Refiner contains refined signature, theorems, and axioms; Task Refiner has refined competence and assumptions, and PSM Refiner has refined communication and competence.

3 Framework for CBR components

As we have presented before, there are two main goals in this work. First, to define a framework of components that allows to specify CBR methods at an abstract level of description. Second, to present a platform or environment that uses the previous described methods and helps the engineer in developing CBR systems.

To deal with the first objective we will present a Component Description Language (CDL). This CDL follows the Task-Method-Domain (TMD) approach, where the main concepts are Tasks that specify the problem to solve; Problem-Solving Methods (PSMs) that specify the way of solving the Tasks; and Domain Models, that specify the knowledge used by the PSMs to solve the Tasks. Our approach takes some ideas from UPML and Components of Expertise (CoE), but adding some changes that solve some limitations presented in these two approaches.

In order to solve the second goal we are developing the CAT-CBR platform. This platform divides the development process of a CBR system into three steps: Configuring, Enabling and Enacting. The Configuring step selects the CBR components that can form a complete configuration that satisfies the requirements for a CBR system. The Enabling step connects the components configuration (obtained in the previous step) to the specific domain of application. Finally the Enacting step operationalizes the configured system into an

executable application.

In this section, first we present the Component Description Language used in the CAT-CBR platform; next, we present the different parts of the CAT-CBR platform; and finally, we describe briefly the process of developing a CBR system using the CAT-CBR platform.

3.1 A Component Description Language

The Component Description Language (CDL) allows the specification of the different components that the CAT-CBR platform uses in order to develop CBR systems. The CDL we present follows the Task-Method-Domain (TMD) approach, learning from the experience in using CoE and UPML, and introducing specific new definitions for Tasks, Problem-Solving Methods, and Domain Models that allow us to model the dynamic nature of CBR systems in an adequate way.

Following the TMD approach, the CAT-CBR Component Description Language is organized around three main concepts: Tasks, Problem-Solving Methods (PSM) and Domain Models (DM). However, we also use the notion of *case-model*, presented in CoE, that specifies the problem-solving situation. In this section we will present a specific characterization of each concept as well as the relationships among them.

A Library of components L will be specified as a tuple

$$L = \langle \mathcal{T}, \mathcal{P}, \mathcal{O} \rangle$$

where \mathcal{T} is a set of tasks, \mathcal{P} is a set of Problem-Solving Methods and \mathcal{O} is a set of ontologies used to describe tasks and problem-solving methods. An Ontology o specifies the allowed terms to specify the components; in our case we have specified a CBR Ontology used to specify the CBR Components. This CBR-Ontology is explained in section 4.1.

A Task t describes which problems can be solved when the task is achieved, and is defined by a tuple

$$t = \langle R_i, R_o, P, G \rangle$$

where R_i specifies the input roles, R_o specifies the outputs roles, P specifies the preconditions, and G specifies the goals using the ontology. The input and output roles are specified by roles r ; a role r is defined as tuple

$$r = \langle \sigma, s \rangle$$

where σ specifies the name of the role and s the type of the *case-model* required by the task; the output roles specify the type of the *case-model* obtained when the task is solved. The *preconditions* describe the requirements over the input needed to carry out the task, and the *goals* specify the properties over the output roles that are obtained after solving the task.

Notice that our definition of Task does not include *assumptions* (characterizing requirements on Domain Models) as UPML does. Thus, there is no direct link between Task and Domain Model; in our approach (as well as in CoE) a Task is matched to a PSM that is linked to a Domain Model. The problem of having a direct link between Tasks and Domain Models (as in UPML) is that it causes a multiplicity of task description in the library without any benefit.

Moreover, CAT-CBR also allows Tasks with the goal of constructing a Domain Model and, therefore, the tasks have a Domain Model as output. Neither CoE nor UPML allow this kind of tasks, but they have been proposed [6] to model learning systems where knowledge is dynamically generated. We have enlarged the notion of task in this way because CBR systems need the dynamic generation of domain knowledge models.

A Problem-Solving Method (PSM) describes a specific way to solve a Task. Problem solving in CAT-CBR approach is considered as the construction of a *case-model*. We follow the “problem solving as modelling” view, that is to say, solving a problem consists of building a model specific to the problem that satisfies the task goals (case-model). In this view, PSMs use domain models to enlarge the input case-models until a complete and correct case-specific model is built—where “complete and correct” are with respect to the goals of the task. Moreover, in our approach, the case-specific model resulting from a CBR system after solving a specific problem is identified with a *case* of that problem.

A PSM is defined as a tuple

$$P = \langle R_i, R_o, P, C, \{ \langle R_k, A \rangle \} \rangle$$

where R_i specifies the input roles, R_o specifies the output roles, P specifies the preconditions over the input, C specifies the competence that describes the properties holding after the application of the PSM, and the set of pairs $\langle R_k, A \rangle$ specify the knowledge role R_k used by the PSM and the assumptions A required to be satisfied by this model.

Each input role is specified by a role whose type characterizes the type of the case-model required by the PSM; each output role is specified by a role whose characterizes the type of the case-model obtained. The PSM *preconditions* specify the requirements to be satisfied by valid input case-models; while PSM *competence* characterizes the output case-models. *Preconditions* and *competence* are described using the *CBR-Ontology*. Each knowledge role is specified by a role which characterizes the type of the Domain Model needed by the PSM; finally the assumptions associated with this model characterize properties needed to be satisfied by the Domain Models used by the PSM.

The Problem Solving Method (PSM) concept has the following two subtypes: *Problem Decomposer* and *Reasoning Resource*. A Problem Decomposer solves a task dividing it into subtasks; its *operational description* specifies the control and data flow over the subtasks. A Reasoning Resource directly solves a task using some domain models. The fact that task decomposition is specified by a PSM moves us closer to UPML than to CoE (where a task description specifies the decomposition into subtasks). However, we allow Domain Models and assumptions to be specified in both PD and RR—while UPML allows them only on RR. In our experience, PDs also need to use domain knowledge (in the form of Domain Models) in the process of decomposing a task into subtasks; for example a PD may know some information about the knowledge in order to select the following subtasks.

Moreover, CAT-CBR allows PSMs whose competence is to generate a Domain Model and, therefore, dynamically creates a new Domain Model. We call these PSMs *Enablers* since they are used in the Enabling process (see section 3.3.2). These PSMs allow us to generate new Domain Models, while CoE and UPML assume that all Domain Models to be used in a particular application are to

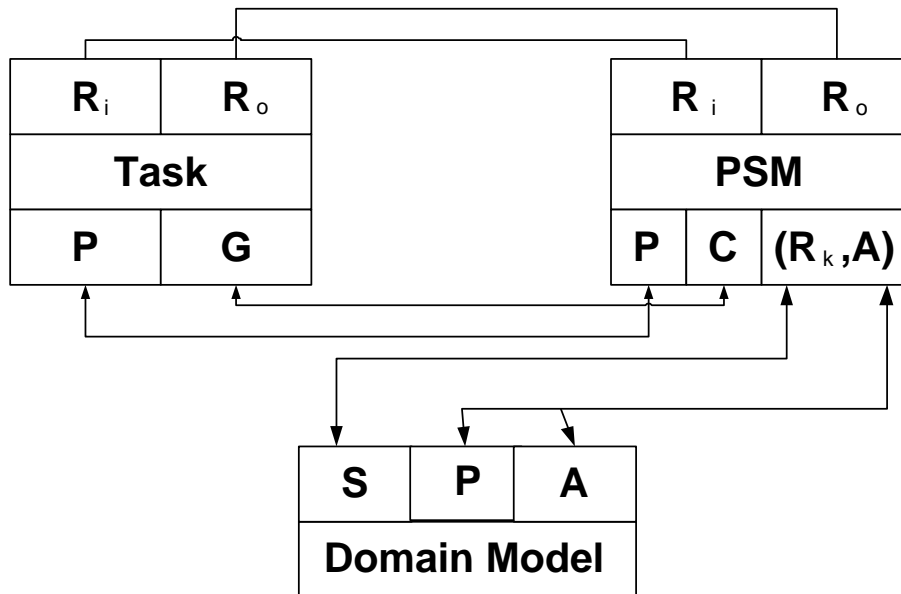


Figure 6: Component matching between Tasks, PSMs and Domain Models.

be given from the beginning of the component-based knowledge engineering process.

Domain Models introduce domain knowledge as required by the PSMs and Task definitions. In our framework a Domain Model is a tuple of three elements

$$M = \langle S, P, A \rangle$$

where s specifies the type of the domain knowledge content, P specifies a characterization of properties of the domain knowledge and A specifies assumptions of the domain model.

Properties and *Assumptions* are both used to characterized the content of the domain model. In our approach, properties and assumptions are expressed in the ontology of Tasks and PSMs, while knowledge content is expressed in the ontology of the domain. The knowledge content of the Domain Model is characterized by the properties, while assumptions deal with external requirements that relate the domain knowledge with the actual domain (e. g. some features of the environment of the system). Notice that, although the three elements of a Domain Model in our approach reassemble those of UPML, we distinguish between the knowledge content (expressed in the domain ontology) and the characterization of that content (expressed in the ontology of Tasks and PSMs).

After presenting the three concepts used in CAT-CBR (Tasks, PSMs and Domain Models), we will describe the relationships among them. First, as we have seen, there is no direct relationship between Tasks and Domain Models. The relationship between Tasks and PSMs is made through *component matching*. *Component matching* is a technique that comes from the area of software component engineering [26]. Specifically, the CAT-CBR Component Description Language includes two types of binary relations between components, namely

Task-PSM matching and PSM-Domain matching.

- A *Task-PSM matching* relation is defined between a task and a PSM. Intuitively, a task-PSM matching denotes a *suitability* relation: a task-PSM relation is verified (is evaluated as true) when the PSM is suitable for the task. In other words, a task “matches” a PSM if the PSM is able to solve the type of problems defined by the task. This relation compares the inputs, outputs and competence of a task against the homonym features of a PSM to determine whether the application of the PSM is able to achieve the postconditions of the task, whenever the preconditions of the task hold.
- A *PSM-domain matching* relation is defined between a PSM and a collection of domain models characterizing the knowledge required by the capability. Since a PSM may include many knowledge-roles, then a domain-model would be required to fill in each knowledge-role. Intuitively, a PSM-domain matching denotes a relation of *satisfiability*: a PSM “matches” a set of domain-models when the knowledge characterized by those domain-models satisfies the knowledge requirements (the *assumptions*) of the PSM for each knowledge-role. This relation is defined in terms of knowledge-roles and capability assumptions that are satisfied by the properties and meta-knowledge of the set of domain-models.

In our approach, we assume that Tasks and PSMs are described using the same ontology: this fact allows us to directly use the notion component matching (fig.6). In this respect CAT-CBR is similar to CoE, while UPML assumes a task ontology different from the PSM ontology (requiring the construction of “bridges” for translating expressions from one ontology into the other). The experience of using UPML in the IBROW project to develop component libraries showed that the development of tasks libraries and PSMs libraries was very interrelated. Essentially, the selection of the tasks descriptions for a Task library, directly implies the PDs and RRs that can be specified in a PSM Library. Thus, from a practical point of view, it is necessary to build both libraries conjointly and, therefore it is reasonable to use a common ontology for both².

Finally, we turn to the relationship between PSMs and Domain Models. In the CAT-CBR approach, we have an ontology for describing Tasks and PSMs that is independent of, and different from, any ontology of a particular domain. In this respect, our approach resembles UPML, but not CoE (that assumes a common ontology for Tasks, PSMs and Domain Models). The PSM-Domain Model relationship in CAT-CBR has two aspects: the adequacy aspect and the ontology aspect. We say that a Domain Model is *adequate* for a PSM when: 1) the type of the Domain Model satisfies the type of a model role, and 2) the assumptions in the PSM associated with the model are satisfied by the properties or assumptions of the Domain Model. The process of determining the adequacy of a Domain Model to a PSM is direct, since (as we have said when defining the Domain Model) properties and assumptions are expressed in the ontology of Tasks and PSMs. The *ontology aspect* is concerned with the mapping between the terms used by a PSM and the corresponding terms in the content of

²However, notice that we keep (as a useful notion) the distinction between the domain ontology and the task-method ontology.

a Domain Model. This mapping in CAT-CBR is embodied in a specific Domain Model that provides the required correspondence relations between terms in both ontologies. The most distinctive of these Domain Models for CBR systems is the *Case-Langue-Model* that provides a PSM with a description of the case representation in a specific domain (see an example of the Domain Models in section 5). Our approach is similar to CoE but differs from UPML in distinguishing two aspects in the PSM-Domain Model relationship. UPML focuses on the adequacy aspect (where it assumes two different ontologies for PSMs and for domain knowledge characterization) and does not provide any particular solution for the ontology aspect (relegated to the component operationalization phase).

3.2 The CAT-CBR platform

Once we have presented our goals and desired features for a language to describe components, we will present the CAT-CBR platform as a component based platform that tries to achieve these goals and to solve some of the problems presented in other approaches. Next, we will enumerate and describe the different parts and processes that form the CAT-CBR platform (see fig.7). These parts can be classified into two types: *CAT-CBR Processes* and *CAT-CBR Repositories*.

3.2.1 CAT-CBR processes

The CAT-CBR platform has three reasoning modules: Configuring, Enabling and Enacting. These three processes carry out the three steps of the developing process.

Configuring The first process is the Configuring, that carries out the configure step of the CAT-CBR. This process takes as input the user requirements of the desired CBR system. These requirements constitute a *Consult* or *Application Requirements*. The *application requirements* are specified by a tuple

$$ar = \langle G, A, R_i, M \rangle$$

where G represents the goals of the desired application, A represents the assumptions that the user can ensure to be true before the application is run, and R_i specifies the input roles of the application, and M specifies the Domain Models that are available for the application to be run. All these elements of the *application requirements* are expressed using the CBR ontology.

The outcome of the configuring process is a CBR system that achieves the user requirements. This *system* is specified by the following tuple

$$s = \langle d, G, A, M \rangle$$

where d represents a task-method decomposition tree, G the general goals that the system achieves, A the assumptions of the system, and M the Domain Models that the system needs to be run.

The Configuring process uses the following repositories from the CAT-CBR platform: the CBR library and the Configuration Case Base. It uses the CBR ontology from the CBR Library to understand and reason about the possible

configurations and to decide when a configuration is final. The configuring process uses the CBR Library in the process of selecting which are the appropriate methods to be applied to solve a task, and to update the partial states during the configuration. And finally, it uses the Configuration Case Base when the process is done by using strategies that need information of past configurations in order to decide which is the most appropriate method to be applied to solve a task.

As future work, we plan to add new strategies to perform this configuring process. These new strategies will follow the constructive adaptation paradigm [19].

Enabling The goal of the *Enabling* process is to link a configured CBR system with the explicit domain knowledge. This process takes as input the configured system s resultant from the Configuring process, and has as output the configured system specification within the explicit domain knowledge, what we call a *enabled system*. A system s is enabled when all the domain models needed by the system are linked to explicit domain knowledge.

To carry out this linkage, the enabling process builds *enablers*. An enabler links a method with the file or resource that contains the needed data. Sometimes this knowledge can not be accessed directly. For example an application that needs a decision tree to be executed and it is not available, but it is available a case base and description of the cases for extracting the decision tree; in these cases the Enabling process advises the user of this situation and finds out using the Configuring process a configuration that achieves this process of extracting or learning this new knowledge. Then, an enabler represents the linkage of the method either with a concrete knowledge or with a configuration that extracts this knowledge.

All this Enabling process could be seen in more detail in next section and in the example of section 5.

Enacting Finally the CAT-CBR platform has the Enacting process. The Enacting process has as goal to transform a *enabled system* s into a real executable application. This executable application will be used by the user to solve new problems in his domain. To carry out this task the Enacting process joins the implementation of the different PSMs that appear in the configuration and defines the order and data flow between these implementations in order to carry out the configuration.

This Enacting process allows the user to test different configurations for the same requirements in order to determine which is the better solution. This a posteriori evaluation is needed because although we have a formal specification of all the components and the domain where they will be applied, the platform cannot assure which will be the component that better fits with this domain. For this reason the CAT-CBR platform gives the opportunity of testing different configurations.

3.2.2 CAT-CBR Repositories

Once the CAT-CBR processes have been presented, we describe the CAT-CBR Repositories. The CAT-CBR platform uses two Repositories: the *CBR Component Library* and *Configuration Case Base*.

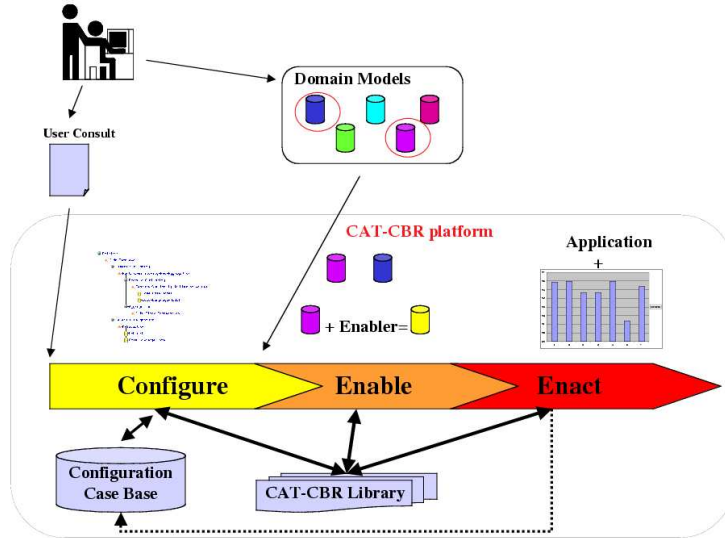


Figure 7: Overview of the CBR system development process using CAT-CBR as well as the different parts and processes that conform the CAT-CBR platform

CBR Component Library The first repository of the CAT-CBR platform is the library of methods described using the Component Description Language and the terminology of the CBR Ontology. The CAT-CBR platform uses these descriptions in order to configure a CBR system for the user. These configuring process will be described in detail below.

In the CBR library we include methods for retrieve, reuse and retain processes of the CBR cycle; we do not describe components for the Revise step (usually done manually by the expert). It is not our purpose to have a complete library of CBR methods, our goal is to present a way for describing these methods and enable their reusability. We introduce a formal specification of these methods that allows the CAT-CBR platform to reason about these methods in order to find a configuration of a CBR system that fits better to the user requirements.

In section 4 we summarize the methods that are actually included in the CBR Library, as well as the Ontology used to describe them. As future work we plan to add an editor of methods that allow the user to enlarge the Ontology and the CBR Components and reuse their own methods.

Configuration Case Base The last repository of the CAT-CBR platform is the Configuration Case Base. In this case base the CAT-CBR platform stores past configurations in order to use them to find new ones. This case base will guide the configuring process in two of the three strategies implemented in the CAT-CBR platform that we will present below.

The Configuration Case Base stores cases with the following structure: on the first place the *Consult* that represents the application requirements, these re-

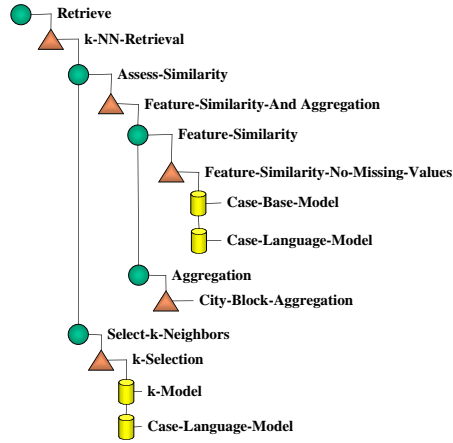


Figure 8: A final configuration that satisfies the consult requirements. Circles stands for tasks, triangles for PSMs and cylinders for domain models.

quirements include the goals of the desired CBR system, the knowledge available and the assumptions needed to use the CBR system; secondly the *Configuration* that has been configured by the CAT-CBR platform and that achieves the previous consult, this configuration is a task-method decomposition tree that represents the different methods, tasks and knowledge forming the CBR system.

3.3 A general overview of the CAT-CBR platform development process

Once we have presented the different modules that form the CAT-CBR platform, we will describe a general overview of how the Component Architecture Technology for a CBR platform (CAT-CBR platform) works. The CAT-CBR platform uses specifications of the components described on the Library in section 4, and allows the components to be reused in order to configure new CBR systems. This platform enables us to fulfil the component reuse objectives and the independence from the domain, and guides us in the development process of CBR systems.

As we have presented in the previous subsection, the way the platform helps us in the development process of CBR systems is following a cycle of three processes: *Configuring*, *Enabling* and *Enacting*.

In the *Configuring* process the necessary methods for developing an application are selected, the ones which will fulfil our needs. The outcome of this process is a configuration; a configuration is a complete task-method decomposition tree (see fig.8); we say a configuration is complete when for each task there is an associated method that solves it.

Once we have obtained the specification of a final configuration, the goal of the *Enabling* process is to connect this configuration with the specific domain application. This domain is represented by Domain Models that a user can reference directly or can acquire from other Domain Models by using *Enablers*. Finally, in the *Enacting* process the configuration together with the domain is operationalized, i.e. produces a CBR system that can be executed. Moreover,

in this last process we also allow to test the obtained system, so we can analyze its accuracy, and even compare two or more systems (alternative configurations) in order to determine which is most suitable.

Next, we will describe each one of the development processes in detail. The explanation of each one of the stages will be illustrated by an example in section 5. The examples we will use are taken out from a musical domain in the Tabasco project. In this project we work with discrete sounds, each one characterized by 49 numerical attributes, and the objective is to be able to distinguish which are relevant ones when doing a classification of these sounds into different hierarchically organized categories, for instance family or instrument.

3.3.1 Configuring process

The configuring process starts with an engineer input, that is what we call a *Application Requirements*. These Problem Requirements contain the needs for the target application, specifying its inputs, preconditions and postconditions, plus the Domain Models available for configuring an application. As preconditions the engineer describes all the properties that he can assure that are true, while the postconditions represents the properties that he wants to be satisfied by the application.

A target application will be a configuration of components of the library that satisfies the consult. A configuration specifies a PSM for each Task. A PSM can be decomposed in new subtasks (problem decomposer) or be elementary (reasoning resource). A configuration also specifies which Domain Models available will be used by each reasoning resource. Figure 8 shows an example of configuration.

The CAT-CBR platform performs this configuring process as search in the space of possible configurations. The configuring process starts up with the Application Requirements and transforms it into an *initial state*. This initial state specifies the preconditions and postconditions of the desired application and the task to be solved. Each *state*, that represents a partial configuration, during the configuring process is defined as a tuple

$$\tau = \langle G, A, \Gamma, \Delta, K, \kappa, B, \beta \rangle$$

where G represents the goals of the desired system not yet satisfied in the state τ ; A specifies the preconditions of the partial configuration represented by the state τ and are not satisfied yet; Γ represents the goals achieved by the partial configuration; Δ specifies the assumptions satisfied by the partial configuration; K represents the knowledge needed in the partial configuration but is not available yet; κ specifies the domain models used in the partial configuration; B represents the bindings (a Task bound with a PSM) in the partial configuration and β specifies the open bindings (tasks with not PSM bound).

The search process starts assigning PSM for all the open task and updates the postconditions achieved. This process of assigning PSMs is done until the CAT-CBR arrives to a final state. A state is final when all the tasks have a PSM that will solve the task, and when all the requirements are fulfilled. This final represents a final configuration.

More details about how CAT-CBR selects a PSM to be bound to a Task, and the different strategies that CAT-CBR implements to perform this configuring

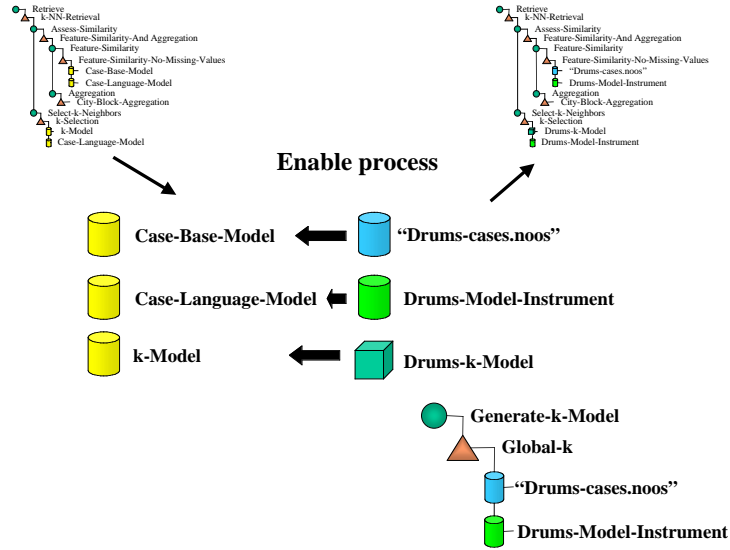


Figure 9: Scheme of the *Enable Process*.

process are presented in the appendix B. A detailed Configuring process in a concrete domain is presented in section 5.

3.3.2 Enabling process

The goal of the *Enabling process* is to connect the Configuration, obtained in the Configuring process, with the specific domain where it will be applied. A configuration is enabled when all the domain models required by the configuration are specified. In order to specify the domain models, the user has two options: first, to reference a concrete domain model (file, URL or structure), and second, to construct a domain model using the knowledge acquisition techniques specified in the CAT-CBR Library. This second option gives the user the possibility of defining a domain model in runtime, what can not be done in UPML approach.

The *Enabling process* is done in two steps. First the user has to determine the domain models that are used in the configuration directly, that means domain models that are available and which the configuration can use directly. In this first step the CAT-CBR platform load this domain models in order to use in the *Enacting process*.

In the second step, the CAT-CBR has to acquire the domain models that are not available. The way how the CAT-CBR platform acquires these domain models are specified through *Enablers* that appear in the *Configuration*. During this process the CAT-CBR platform selects the domain models needed by the *Enabler* and *enacts* the enabler in order to acquire the required domain model. The Enabler is enacted using the *Enacting module* of the CAT-CBR platform.

Finally, the CAT-CBR platform join the Configuration with the domain models selected by the user and the ones obtained from the Enablers; all these information constitutes a *Enabled System* what is the information needed by the CAT-CBR platform to start the *Enacting process*. In short, the *Enable process*

takes a configuration and connects it with the concrete domain models, either referencing concrete domain models or constructing them through *enablers* (see figure 9). The example presented in section 5 shows in more detail the enabling process.

3.3.3 Enacting process

The configuring process yields a configuration of components (see fig.8), the enabling process connects the models with the configuration, and finally the *enacting process* deals with how an *Enabled system* must be operationalized, i.e. obtaining an executable application. To operationalize a *Enabled system* the CAT-CBR platform uses the implementation of the different methods and substitutes the parameters of the different methods in order to carry out the configuration specified in the *Enabled system*.

To explain this process, let us suppose that a task (named *Task-i*) has two inputs: *var-1* and *var-2* with types *Typevar-1* and *Typevar-2*. This task is bound with a PSM in the configuration, suppose that this method is called *PSM-i* and it has the same number of inputs and the type of these inputs subsume the types of the task inputs³. This *PSM-i* uses also some Domain Models; in this specific case, let us suppose that the PSM uses three Domain Models: *dm-1*, *dm-2* and *dm-3* which types are *TypeDm-1*, *TypeDm-2* and *TypeDm-3* respectively.

The *PSM-i* is implemented in the CAT-CBR platform as follows:

```
(define-method PSM-i-Method
 :inputs ((var-1 Typevar-1) (var-2 Typevar-2))
 :domains ((dm-1 TypeDm-1) (dm-2 TypeDm-2) (dm-3 TypeDm-3))
 :subtasks (Task-1 Task-2)
 method-code)
```

During the Enacting process, a task has to be carried out when it is called through the special function *use-subtask*. In our case, it will be in the following way: (*use-subtask Task-i var-1 var-2*). At this point, the CAT-CBR platform searches in the configured system the PSM bound with the task, and substitutes the general call to the task with the method that implements the PSM. In our case, to implement the *Task-i*, the CAT-CBR platform will call the *PSM-i-Method* in the following way:

```
(PSM-i-Method :inputs (var-1 var-2) :domains (DM1 DM2 DM3))
```

Where *DM1*, *DM2* and *DM3* are the domain models selected in the *Enabling process* to perform the *PSM-i*. This process is shown in more detail through a concrete example in section 5.

4 CAT-CBR Library

One of the main parts of the platform is the Library of components. In this Library we will specify the characteristics for each one of the tasks and methods. These specifications will allow us to reason about the components with the goal of configuring a CBR system. As we have previously presented, the specifications will be done by using a Component Description Language.

³We can assume this because the *PSM-i* is bound with the *Task-i* and therefore the PSM has to satisfy the matching criteria

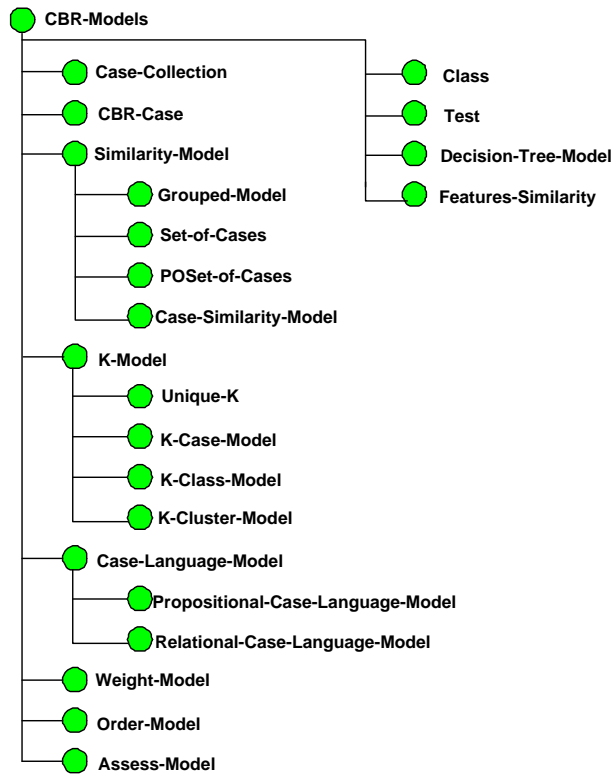


Figure 10: Hierarchy of sorts defining CBR Models.

The Library consists of, besides the ontology, CBR tasks and methods. These constitute the components that will be used to configure a CBR application. The Library undertakes three of the four CBR processes: Retrieve, Reuse and Retain; we will not include Revise methods because this process is usually carried out by an expert, and the existing automatic methods are always tied to the working domain, so they cannot be specified independently from the domain, which is one of our main objectives.

4.1 CBR Ontology

The CBR ontology defines the vocabulary used to describe all the reasoning components in the platform: the components in the CBR Library (Problem Decomposers, Reasoning Resources, Tasks), the user requirements and the domain models contributed by the user. This ontology is included in the Library.

This ontology is defined as a hierarchy of feature terms. These terms can be grouped in two families: a) CBR Models, that are used to represent concrete data (i.e. inputs and outputs of a PSM) , and b) CBR concepts that are used to specify properties of tasks and PSMs (i.e. postconditions of a Task). Next we present part of the CBR Models and CBR Concepts that constitutes the CBR Ontology:

- *CBR-Case*: It is the structure that contains one case used in the CBR

process.

- *Case-Collection*: It represents a collection of CBR-Cases. This collection of cases can be part of the input or output of a component, as well as they can form the knowledge of a Case Base.
- *Similarity-Model*: It holds the similarity information obtained from the Retrieve Task. There are four subsorts of similarity model: 1) *Set-of-cases* that stores cases more similar to a given case; 2) *Grouped-Model* that stores the cases, grouped by classes or clusters, that are more similar to a given case; 3) *Poset-of-cases* that represents a set of similar cases with a partial order between them; and 4) *Case-Similarity-Model* that holds the similar cases and their similarity to the given case.
- *k-Model*: it holds the information about the number of more similar cases k that have to be retrieved. There are four subsorts of k-model (see fig.10) depending of the information they hold: 1) *Unique-k* that stores only one value of k ; 2) *K-Case-Model* that stores, for each case in a case-collection, the k -values that classify correctly the case in a leave-one-out strategy; 3) *k-Class-Model* that holds, for each class, the best k -value that classifies it correctly; and 4) *k-Cluster-Model* that holds, for each cluster, the best k -value that classify it correctly.
- *Case-Language-Model*: it holds the information about the representation of the cases. There are two subsorts: 1) *Propositional-Case-Language-Model* that holds the set of attributes that describe a propositional case; and 2) the *Relational-Case-Language-Model* that holds the sorts used to describe a relational case.
- *Weight-Model*: it holds a number (weight) that represents the relevance of the attributes.
- *Order-Model*: it holds a function used for ordering the attributes.
- *Assess-Model*: it holds, for each case from a case collection, the similarity to a given problem case.
- *Class*: is the solution of Classify Task.
- *Decision-Tree-Model*: it holds a decision tree, where the nodes are tests and leaves are collections of cases. This decision tree can be used either for Retrieve Task or Classify Task.
- *Test*: it holds the information of the intermediate nodes of a decision tree.
- *Features-Similarity*: for each attribute, it holds the similarity between two cases.

Next we present some of the CBR Concepts defined in the CBR Ontology:

- *Noise-Tolerance*: It represents the resistance of a component to work with noise in the data. We differentiate two subsorts: *High-Noise-Tolerant* and *Noise-Tolerant*.

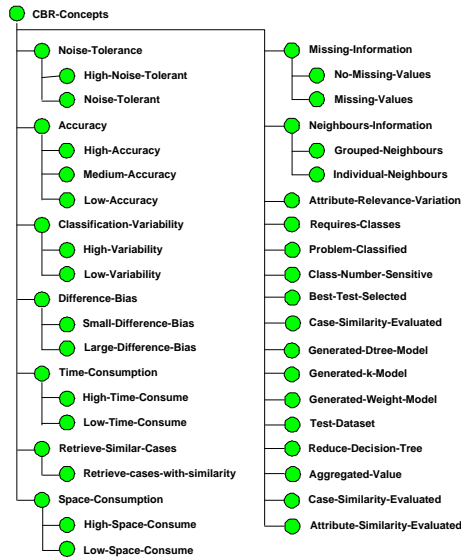


Figure 11: Hierarchy of sorts defining CBR Concepts.

- *Accuracy*: represents the accuracy of a method. We distinguish three subsorts: *High-Accuracy*, *Medium-Accuracy* and *Low-Accuracy*.
- *Classification-Variability*: It represents the variability of a method in classification. We distinguish *Low-Variability* and *High-Variability*.
- *Difference-Bias*: It represents how to compute the similarity between two cases. There are two subsorts: 1) *Small-Difference-Bias* means to punish big differences; and 2) *Large-Difference-Bias* means to punish low differences.
- *Time-Consumption*: It represents the information about time cost of a method. There are two subsorts: *High-Time-Consume* and *Low-Time-Consume*.
- *Space-Consumption*: It represents the information about memory cost of a method. There are two subsorts: *High-Space-Consume* and *Low-Space-Consume*.
- *Missing-Information*: it represents the tolerance of a method to work with missing values.

The remaining Concepts (see fig.11) characterize specific aspects of the components from the Library. Actually this CBR ontology is fixed, but in future work we will add an editor that will allow the user to enrich the CBR Ontology as well as to define new reasoning components for the CBR Library.

4.2 Tasks and PSMs in the CBR Library

In this section we will describe the different tasks and PSMs that compose this library. To develop this library we take the approach presented in [12]. This

approach shows the problem solving method development as a process taking place in a three-dimensional space, defined by problem-solving paradigms, domain assumptions, and task commitments. These three dimensions are the following:

- *Problem-Solving paradigm* fixes some basic data structures, provides an initial task-subtask decomposition and a generic control regime. This generic control regime is meant to be shared by all problem-solving methods which subscribe to the same problem solving paradigm.
- *Domain knowledge assumptions* are assumptions on the domain knowledge that is required to instantiate a problem solving method in a particular application. These assumptions specify the types and the properties of the knowledge structures which need to be provided by a domain model, in addition to those required to fulfil task-specific commitments.
- *Problem commitments* specify the commitments of the type of problem that is solved by the problem-solving method.

In CBR there are different techniques for each one of the CBR cycle stages. The CAT-CBR Library tries to group these different techniques, with the objective of being able to use them in different applications. So, all these techniques are described using our Components Description Language independently from the domain.

In this way, the library undertakes Retrieve methods for propositional cases as well as for relational cases. At the same time, in this retrieve stage, exhaustive methods (Nearest Neighbor) and inductive methods (Decision Trees) are incorporated. In the Reuse stage we are introducing either classification methods (analytic task) alike case based configuration methods (synthesis task). To what Retain stage concerns, there are introduced in the Library different methods for retention policies and active learning.

Finally, due to CAT-CBR platform allows the introduction of new models for a configuration during the Enabling stage, the library includes methods in order to acquire these models. Within this group (Methods for model acquisition) we will find methods for extracting weight models for the attributes, inferring decision trees, extracting k models for the k-Nearest neighbor family, discretizing numeric values and so on. We do not include methods for the Revise stage in the library. Below we will briefly present the different tasks and methods the library includes.

4.2.1 Retrieve Task

Given a new problem P the objective of the Retrieve task is to retrieve cases similar to the problem P. The output of this task is a similarity model (see ontology). These similarity models can be of different kinds, a collection of cases, a partially ordered set of cases, a set of cases together with its similitude to the problem P, or also a grouped similarity model.

Considering the kind of cases which a method can work with, two families of methods can be differentiated: methods that deal with propositional cases and methods that deal with relational cases. Propositional cases are those in which the case is represented by a vector of attribute values, meanwhile relational cases are described by more complex structures of characteristics.

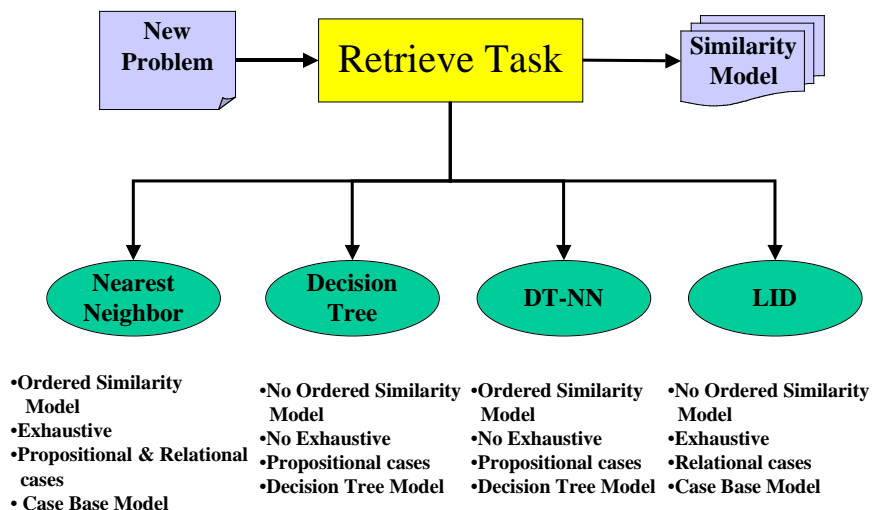


Figure 12: Classification of the different families of methods depending on the *Problem-Solving paradigm*.

At the same time we can also differentiate several kinds of methods depending on the knowledge they use, so we specify methods that work over the Case Base and methods that work using Decision Trees. Within the dimension problems commitments two families of methods can be differentiated: methods that can deal with unknown values in the cases, and methods that cannot do that.

Taking into account the domain knowledge assumption dimension we have included two families of methods according to the knowledge they use for achieving the Retrieve Task: methods that use decision tree and methods that use a case base. Concerning the problem-solving paradigm dimension there are two families of methods: 1) methods that handle problems with no missing values; and 2) methods that handle problems with missing values. In the second family the methods can follow different strategies for solving the missing value problem.

Below we will go on analyzing the different methods incorporated in the library, taking into account the previously presented aspects.

k-Nearest-Neighbor Family Firstly, we have the family of k Nearest Neighbor retrieve methods. These nearest neighbor methods have the following general working scheme: given a new problem P, they evaluate the distance between all the cases stored in the case base and the problem P, and then they select all the nearest or most representative k cases.

Inside the library, this family of methods is represented as a problem decomposer *k-NN-Retrieval*. This problem decomposer has two subtasks: the former in which it is evaluated the distance from the problem to the database cases, Asses-Similarity; and the latter, Select-k-Neighbors in which the most relevant

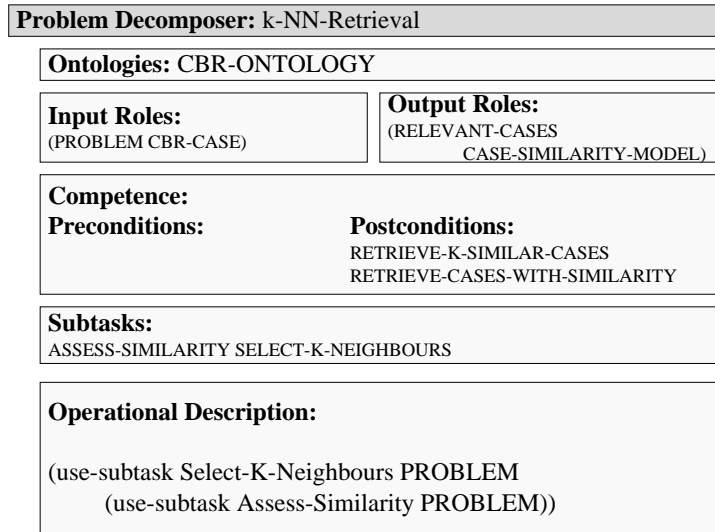


Figure 13: Specification of the *k-NN-Retrieval* problem decomposer

or nearest k cases are selected.

Depending on the methods used in order to solve these two subtasks, the PSM k-NN-Retrieval will be able to deal with known or unknown values, with propositional or relational cases, at the same time we will have different kinds of outputs. Before going on with subtasks, it is necessary to emphasize that this family of k nearest neighbors methods are exhaustive methods, this means that for each new problem, the retrieve process works with all the cases from the Case Base.

Figure 13 shows the specification of this problem decomposer. The input of *k-NN-Retrieval* is a Problem of sort *CBR-Case*. The output is a *Case-Similarity-Model* containing the more similar cases and their similarity to the Problem. The operational description shows that this method first solves the *Assess-Similarity* Task and then the output of this task is used for solving the *Select-K-Neighbors* Task.

The goal of the *Assess-Similarity* subtask is to evaluate the similarity between the cases from the case base and the Problem. The input of the *Assess-Similarity* task is Problem, and the output is *Assess-Model*. This model contains the cases and their similarity to Problem. If we focus on the domain knowledge assumptions dimension, we can differentiate two families of methods depending on the representation of the cases: methods for propositional cases and methods for relational cases. In the first group we have defined the following decomposer:

Within the first group we have specified it as a problem decomposer, *Feature-Similarity-And-Aggregation*. This method will firstly evaluate the similarity of the attributes of the cases and next it will aggregate this feature similarity into a unique value. So, this problem decomposer has two subtasks: *Feature-Similarity* and *Aggregation*. In the second family we have described the *SHAUD*[7], that is a reasoning resource that achieves the *Assess-Similarity* task when the cases are represented as relational cases.

The *Feature-Similarity* task has as a goal to evaluate the difference between two cases regarding a feature. Taking into account the problem-solving paradigm dimension we have two methods: *Feature-Similarity-No-Missing-Values* and *Feature-Similarity-Missing-Values*.

The goal of the *Aggregation* task is to aggregate the differences in all the features. Depending on the domain knowledge assumption dimension there are methods that need a Weight Model (*Weighted-Mean* and *WOWA*), methods that need an order model (*OWA* and *WOWA*), and methods that do not need any models (*City-Block-Aggregation*, *Euclidean-Aggregation*, *Txebitxeve-Aggregation*).

It is left to analyze the second subtask of the K-NN-Retrieval, this subtask is Select-k-Neighbors. This task objective is, given an Assess-Model, to return a Similarity-Model. Depending on the k model used and on the method selected for solving his task, the task output will be a Grouped-Model or a Case-Similarity-Model. So the methods that can be applied are:

- *k-selection*: It is a reasoning resource that uses a unique k as K-Model to select the cases. It returns the k nearest neighbors of the problem in Case-Similarity-Model.
- *k-selection-case*: It is a reasoning resource that uses a K-Case as K-Model. The M nearest neighbors of the problem are selected and that k which classifies correctly most of these M neighbors is used to return the k nearest neighbors of the problem. As previous method it also returns a Case-Similarity-Model.
- *Group-selection-class* that uses a K-Class as K-Model and returns the neighbors grouped by classes.
- *Group-selection-cluster* that uses a K-Cluster as K-Model and returns the neighbors grouped by clusters.

Decision Tree Retrieval Family This family of retrieve methods, unlike the nearest Neighbor, works with decision trees. A decision tree is a model with tree structure, in which the internal nodes contain a test about an attribute, and sets of cases are stored in the leaves. In order to work with these models, we have two groups of methods, one that does not allow unknown values and another one that allows them. Within the first group it is the *DT-Retrieval*.

The *DT-Retrieval* is a reasoning resource that solves the *Retrieve* task. The goal of *DT-Retrieval* is to retrieve a set of cases using a decision tree. This reasoning resource has as input a *Problem* of sort *CBR-Case*. The output is a *Set-of-Cases* similar to *Problem*. The DT-Retrieval starts at the top of the decision tree and descends through it making the tests stored in the nodes until it reaches a leaf; then it returns the set of cases stored in the leaf.

In the second group we have three methods, that implement three different strategies in order to deal with the problem of the unknown values. These three methods are: *DT-Retrieval-Missing-Values*, *DT-Retrieval-MostFV* and *DT-Retrieval-AllBranch*.

The *DT-Retrieval-Missing-Values* is a reasoning resource that has as Inputs and outputs the same than those of *DT-Retrieval*. The difference between both reasoning resources is that *DT-Retrieval-Missing-Values* can handle cases with

missing values. When a missing value is found the method returns all the cases found under the node where the missing value has been found.

The *DT-Retrieval-MostFV* is a reasoning resource that solves *Retrieve* task. Inputs and outputs of *DT-Retrieval-MostFV* are the same than those of *DT-Retrieval-Missing-Values*. The difference between both reasoning resources is how they handle the missing values. When a missing value is found, *DT-Retrieval-MostFV* evaluates which is the most common value and continues the retrieve through this branch of the decision tree.

The *DT-Retrieval-AllBranch* is a reasoning resource that solves *Retrieve* task. Inputs and outputs of *DT-Retrieval-AllBranch* are the same than those of *DT-Retrieval-Missing-Values*. The difference between both reasoning resources is how they handle the missing values. When a missing value is found, *DT-Retrieval-AllBranch* continues the retrieve process through all the branches of the decision tree.

Decision Tree and Nearest Neighbor The *DT-NN-Retrieval* is another problem decomposer incorporated for solving the *Retrieve* task. Inputs and outputs of *DT-NN-Retrieval* are the same than those of *k-NN-Retrieval* (i.e. Problem and Case-Similarity-Model respectively). As a problem decomposer it has two subtasks: 1) *Retrieve-for-Sets* that retrieves a subset of cases using a decision tree; and 2) *Retrieve-for-K-NN* that retrieves cases with similarities from the subset of cases retrieved by *Retrieve-for-Sets*. This method is very fitting for big case bases, so it allows to annotate the group of cases about which the nearest neighbor is applied through the decision tree.

LID *LID* is reasoning resource that deals the *Retrieve Task* over *relational cases*[5]. This PSM returns as similarity model a set of cases and works with a *Case-Base-Model*. This method starts with a general description of the problem, and specializes this description in the case base; this process of specializing is done until the set of cases that are specialized belongs to the same class or it can not be specialized. The method returns a set of cases with no order between them.

4.2.2 Reuse Task

The reuse process has as an objective, given the problem P and a similarity model obtained in the Retrieve process, to Reuse the solutions of the similarity model to apply it as a solution of the problem P. Depending on the kind of solution that has to be given there are different methods to obtain them. In this library we will present methods to carry out classification tasks (analytic tasks) and configuration tasks (synthesis tasks), in which it is found to take advantage of previous cases and its solutions to find a new configuration.

Classification Task The goal of classification task is given a new problem, to classify it into a collection of predefined classes (determined a priori by the user). These classes are described in the Case-Language-Model, that also stores which attributes are to be used, which are the objective classes and how to have access to each one of them inside the cases. Depending on the similarity model we have at our disposal, obtained in the Retrieve process, we will have

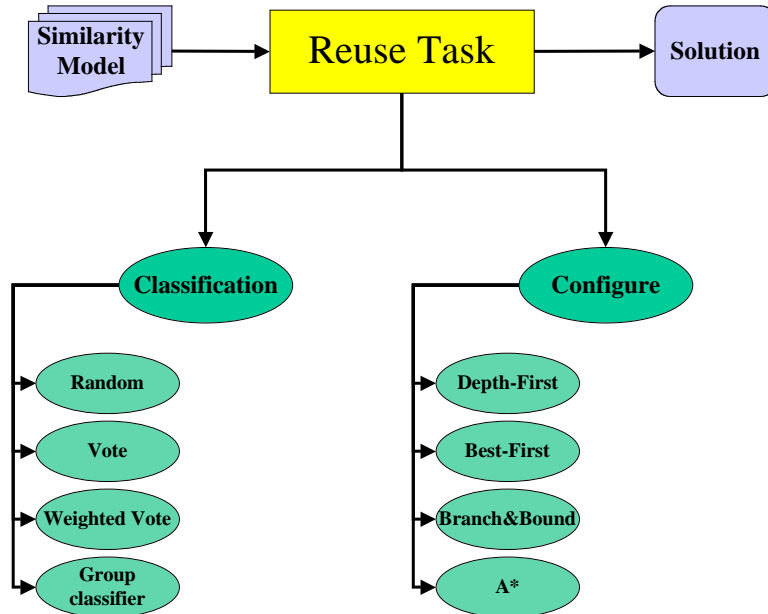


Figure 14: Classification of the different families of methods for reuse depending on the task they solve.

different alternative methods to be applied. When we have set of cases or Case-Similarity-Model we can apply the following methods: *Majority*, *Weighted-Voting*, *Random-Selection*.

Majority is a reasoning resource that takes a set of cases as an input, and its result is the majority class. This method works with a set of cases as with Case-Similarity-Model without distinction, the only thing is that in the second case it does not use the information of the similarity between the cases with the problem P.

Weighted-Voting is a reasoning resource that carries out a vote in order to select which is the majority class, but it does not take into account the similarity of each one of the cases to execute this vote. This method can only work with Case-Similarity-Model.

Finally, *Random-Selection* is a reasoning resource that selects a case randomly among a set of cases, and takes its class as the class for the new problem. This method allows to select classes although there are only a few classes that give support to these classes.

Apart from the methods that work with a set of cases and Case-Similarity-Model, we have methods that work with a Grouped-Model. This method is called *Grouped-Model-Classification*. *Grouped-Model-Classification* is a reasoning resource that takes a Grouped-Model as an input, and returns a class or cluster as a result. Let us remember that a Grouped-Model keeps a set of cases for each class or cluster. The method works as follows: for each group it is calculated the relation of cases for the group that really belongs to this class, the class that has more relation between cases in this class over classes from the

group, is the one considered as result class.

Configuration Task The goal in the Configuration task is: given a new problem, to construct a new solution for this problem by reusing the solution of the most similar cases. There are different approaches to solve this task, but in the Library only methods based in a search process are incorporated. In future other approaches will be included.

This family of methods deals with the problem by searching in the solutions space, a configuration (solution) which satisfies the new problem. To guide the search process the methods use the Similarity Model obtained in the Retrieve task. Depending on the kind of Similarity Model they use and the characteristics of the solution, we distinguish the following methods:

- *Depth-First* is a reasoning resource that uses a set of cases as similarity model. It searches the solutions space in depth finding a solution that satisfies the problem.
- *Best-First* is a reasoning resource that uses a Case-Similarity-Model to find the solution. It selects the components for the final solution by the heuristic of choosing the components that belong to the solution of the nearest cases until the configuration satisfies the problem.
- *Branch-And-Bound* is a reasoning resource that uses a Case-Similarity-Model and needs a cost information about the components. It uses the Case-Similarity-Model in the same way that *Best-First* to guide the search process and the cost of the components to discard some possible configuration in order to get the solution.
- *A** is a reasoning resource that works in the same way than the *Branch-And-Bound* but it also assures that the found solution is the one that has the minimum cost and satisfies the problem.

4.2.3 Retain Task

A main issue in learning is to select which are the examples of the target problem to learn from. The goal of the *Retain* task is just this, to determine when a case has to be added in the case base to improve the application when solving new problems. We can differentiate two different strategies: 1) *active learning*, that uses an strategy to determine when to select a case to be stored without knowing its solution; and 2) *retain policies*, that decides when solved cases have to be added in the case base.

In the first family we distinguish two reasoning resources: *Informative-Disagreement* that uses an Information measure to decide if it is useful to add the new case to the case base, and *Frontier Selection* that uses a distance measure to decide if the new case is in a frontier and then adds it to the case base.

We can differentiate in the last family, three different strategies for determining when a case must be added to the case base: *On-Failure Retain* that adds those cases that are bad classified; *Never-Retain* that never retains a case in the case base, this strategy can be useful in order to increase noise in the case base; and finally, *Always-Retain* that corresponds to passive learning strategy.

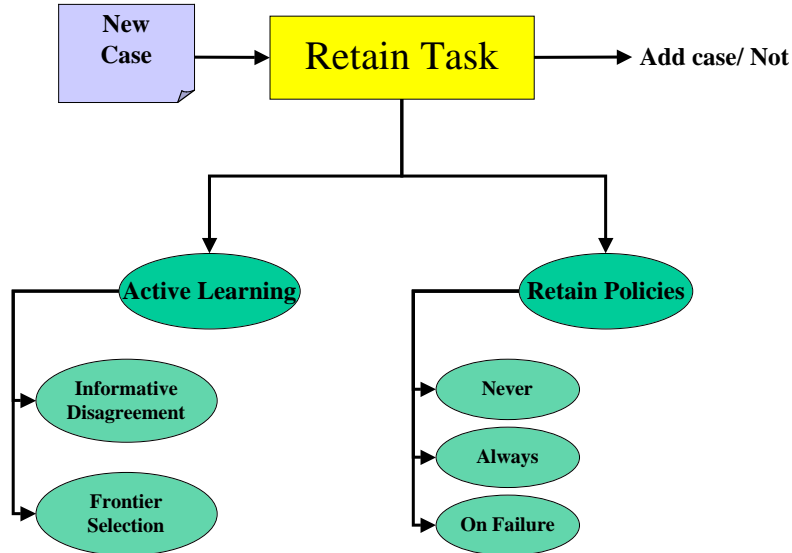


Figure 15: Classification of the different families of methods depending on the *Problem-Solving paradigm*.

These strategies and methods have been recently included in the library and must be studied in more detail, in order to better specify them. This is part of our future work.

4.2.4 Tasks and methods for acquiring knowledge

In addition to the tasks and methods described previously, the library also contains methods and tasks for acquiring knowledge (Domain Models) from other knowledge. These methods are mainly used in the Enabling step of the CAT-CBR-Platform. In the next section we introduce the *Construct-DT* task that induces a decision tree from a case base, and the methods that solve it. Then, we present the *Generate-k-Model* task that learns a *k-Model* from a case base. Finally, we present methods that solve *Learn-Weight-Model* task and allow us to extract a Weight Model for a propositional case from a Case Base and a Case-Language-Model.

Construct-DT Given a collection of cases, the goal of *Construct-DT* task is to construct a decision tree that indexes them. There are two methods that solve the *Construct-DT* task: *DT-Construction* and *DT-Construction-Pruning*.

DT-Construction is a problem decomposer that solves the *Construct-DT* task decomposing it into three subtasks: *Stop-Criteria*, *Select-Test* and *Branching*. *DT-Construction-Pruning* is a problem decomposer that solves the *Construct-DT* task decomposing it into four subtasks: *Stop-Criteria*, *Select-Test*, *Branching* and *Pruning*. Both methods evaluate whether the collection of cases satisfies

the *Stop-Criteria*. When it is not satisfied, then *Select-Test* determines the best criteria to split the collection into subsets. After this, *Branching* generates all the branches for each subset. This process is recursively done until all the final branches satisfy the *Stop-Criteria*. The result of this process is a decision tree.

Let us analyze in detail the task decomposition of *DT-Construction* and *DT-Construction-Pruning*. The goal of the *Stop-Criteria* is to determine whether a collection of cases must be split or not. There are two methods for solving this task: 1) *Homogeneous-Partition* is a reasoning resource that has the following criterion: a collection must not be split if all the cases belong to the same class; and 2) *Fixed-Number-Partition* is a reasoning resource that decides not to split a collection if the number of cases is less than a predetermined number.

The goal of the *Select-Test* task is to choose a test useful for dividing the collection (input) cases. There are a lot of heuristic methods for solving the *Select-Test* task: *Gain* [15], *Chi-Square* [14], *G-Statistic* [14], *GINI-Index* [10], *Gain-Ratio* [15] and *RLM-Distance* [18].

The goal of the *Branching* task is to generate new branches of the decision tree. Each branch contains the subsets obtained from *Select-Test* task.

In addition to *Stop-Criteria*, *Select-Test* and *Branching* tasks, the *DT-Construction-Pruning* method has the *Pruning* task. The goal of *Pruning* task is to reduce the size of the decision tree without reducing its accuracy. There are several methods that can be used for solving the *Pruning* task:

- *Error-Complexity*: It is a reasoning resource that takes into account both the number of errors and the complexity of the tree. This method is fully developed in [10].
- *Critical-Value*: It is a reasoning resource that relies on estimating the importance or strength of a node from calculations done in the tree creation stage [14].
- *Minimum-Error*: It is a reasoning resource that finds the single tree which should theoretically give the minimum error rate when classifying independent sets of data [23].
- *Reduce-Error*: It is a reasoning resource which produces a series of pruned trees by using the test data directly, rather than using it only for selection of the best tree [16].
- *Pessimistic-Error*: It is a reasoning resource which aims to avoid the necessity of a separate test data set [15].

Generate-k-Model Given a case base, the *Generate-k-Model* task has as goal to generate a *k-Model*. In the CAT-CBR Ontology we have explained in detail the different kinds of *k-Models*. We have included in the library a method for generating each *k-Model*. Some of this methods are the following:

- *Global-k*: It is a reasoning resource that returns the best *k* for all the Case Base using leave-one-out.
- *Local-k-Unrestricted*: It is a reasoning resource that returns, for each case, the set of *k* that classify correctly each case.

- *Local-k-Pruned*: It is a reasoning resource that returns, for each case, the set of k that classify correctly each case. Only those k that appear more than a number of times given by the user are kept.
- *Local-k-Class*: It is a reasoning resource that returns a k for each class. This k is selected by leave-one-out.
- *Local-k-Cluster*: It is a reasoning resource that returns a k for each cluster. This k is selected by leave-one-out.

Learn-Weight-Model Given a case base, the *Learn-Weight-Model* task has as goal to generate a *Weight-Model* for the attributes specified in a Case-Language-Model regarding a classification specified in the same Case-Language-Model. This Weight-Model represents the relevance of the attributes in order to classify cases. Methods that solve this task are the same heuristics used in the *Select-Test* task. These methods are the following: *Gain* [15], *Chi-Square* [14], *G-Statistic* [14], *GINI-Index* [10], *Gain-Ratio* [15] and *RLM-Distance* [18].

5 An example of using CAT-CBR in a musical domain

In this section we will present an example of application domain, and how an application engineer can use CAT-CBR in this domain. The application domain is a musical domain, more specifically is a domain of discrete unpitched percussion sounds. This application domain is part of the *Tabasco* project where the CAT-CBR research is been carried out.

First we will describe and characterize the domain; then we will configure a number of CBR systems for this domain; later we will present how to enable these CBR systems; and finally we will show some experimental results after the enacting process.

5.1 Presenting and characterizing the domain problem

Classification is one of the processes involved in audio content description. Audio signals can be classified according to miscellaneous criteria. A broad partition into speech, music, sound effects (or noises), and their binary and ternary combinations is used for video soundtracks. Automatic labelling of instrument sounds has some obvious applications for enhancing the operating systems of sampling and synthesis devices, in order to help sound designers to categorize new patches and samples.

In our case will work with unpitched percussion sounds and try to develop a CBR classification system. In this case we face with three problems: a) there are a lot of descriptors that can be extracted from a sound (either temporal or spectral), but the user does not know which are the more relevant in order to classify them; b) as the source recordings are different, the classification is more difficult and relevance of the descriptors can change depending on the source; and 3) also the user can use different criteria to classify the sounds (family, instrument, etc).

The sounds are characterized by 207 numerical descriptors and classified in more than thirty classes. We will work with a case base of 1800 cases. First we have to define the case structure that characterizes this domain. The cases are propositional cases made of 207 attributes plus the class they belong to; all these attributes are numerical attributes. The structure of these cases is defined as follows:

```
(define-sort Mostpercussionclasses
  (Class Instrument)
  (Specflat Number)
  (Speccent Number)
  (Speccentn Number)
  (Strpeak Number)
  (Kurtosis Number)
  (Scr Number)
  (Zcr Number)
  (Strdecay Number)
  (Varsc Number)
  (Varzcr Number)
  (Skew Number)
  (B1 Number)
  (B1bis Number)
  (B2 Number)
  ....)
```

The first attribute “Class” is the solution attribute, but it is defined as the other ones; the reason is that the solution attribute and the features that we will consider to classify sounds are defined in the *Case-Language-Model*. This *Case-Language-Model* is defined as follows:

```
(define (Case-Language-Model :id Drums-Case-Language)
  (case-spec (define-spec (Mostpercussionclasses)))
  (case-attributes Specflat Speccent Speccentn Strpeak Kurtosis
    Scr Zcr Strdecay Varsc Varzcr Skew B1 B1bis
    B2 B2bis B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13
    ....)
  (solution-attribute Class)
  (solution-accessor 'Class-Accessor))
```

The *Case-Language-Model* defines the solution attribute as the class feature and all the 207 sound descriptors as relevant for classification.

Once we have defined the structure of the cases we have to define the type of each attribute. In CAT-CBR we can define four types of features: Numeric, Symbol, Ordered Symbol and Structured. In our case the features are Numeric, except the “class” feature that is a symbol. These features are defined as follows:

```
(define (Symbol-Attribute :id Class)
  (name 'Class)
  (accessor 'Class-Accessor)
  (sim-function 'Symbol-Similarity)
  (range Bongo BongoElec ClapAcous ClapElec Clave-WoodB1 Conga CongaElec Cowbell Crash
    CrashElec Cymbal808 HHAcousCl HHAcousOp HHElecCl HHElecOp Kick808 Kick909 KickAcous
    Ride RideElec Shaker ShakerElec SideStick SnareAcous SnareElec Tabla Tambourine TambourineEl
    Timbale TomHi TomLow TomMed Triangle))

(define (Numeric-Attribute :id Specflat)
  (name 'Specflat)
  (sim-function 'Specflat-Similarity)
  (range (define (interval)
    (inf-lim -61)
    (sup-lim -1))))
```

In this specification we define the range of the attribute and the similarity function. The similarity function can be defined either by using a similarity function available at the CAT-CBR platform or defining a domain-specific similarity (the option chosen in the example). The attribute definitions can be generated automatically by extracting the information from XML or ARFF files (ARRF files are used to define domain and cases in the WEKA⁴ library). Once we have characterized the application domain, the engineer can start the configuring process.

5.2 Configuring CBR systems with CAT-CBR

Once the engineer has characterized his domain, he has to define the goals and requirements of the desired CBR system. In our case, the problem domain is to classify sounds using CBR system; this domain has no missing values in the Case Base, and the knowledge available is a Case-Language-Model and a Case-Base-Model. All this information is represented in the *Application Requirements*:

Application Requirements

Assumptions → Propositional-Case

Goals → Classify-Problem

Input-roles → (Problem CBR-Case)

Models → (Case-Base-Model (properties Propositional-Case No-Missing-Values)),
(Case-Language-Model)

Where **input-roles** is the input of the desired CBR system; **Assumptions** and **Goals** represent the preconditions and postconditions (or goals) of the desired CBR system; **Models** specifies the Domain Models available for the desired CBR system. The CAT-CBR platform translates this requirements into an initial state that is the starting point of the search process (Configuring process). The CAT-CBR platform, with the help of the engineer, reasons about the components in the library until it finds a set of possible configurations.

Figure 16 shows one of the configurations generated by CAT-CBR given the application requirements. Note that in this configuration the classify top task is performed using a k-NN method and reusing the cases through a Majority strategy. For performing the Assess-similarity task inside the k-NN method, the Weighted-Mean aggregation has been selected. We have also to note that there are domain models which are not available in the Problem Requirements; these domain models will be introduced in the Enabling process. As the engineer does not know if Weighted-Mean is the best option to this domain, he can generate other configurations using Euclidean, Txebitxev and City-Block aggregation. After this configuring process, the user obtains four configurations that he can enable, enact and test.

5.3 Enabling CBR systems

Once the engineer has configured a CBR system he has to Enable it. As we have seen this process consists in linking the configuration with the specific

⁴WEKA is a collection of machine learning algorithms for solving real-world data mining problems. <http://www.cs.waikato.ac.nz/ml/weka/>

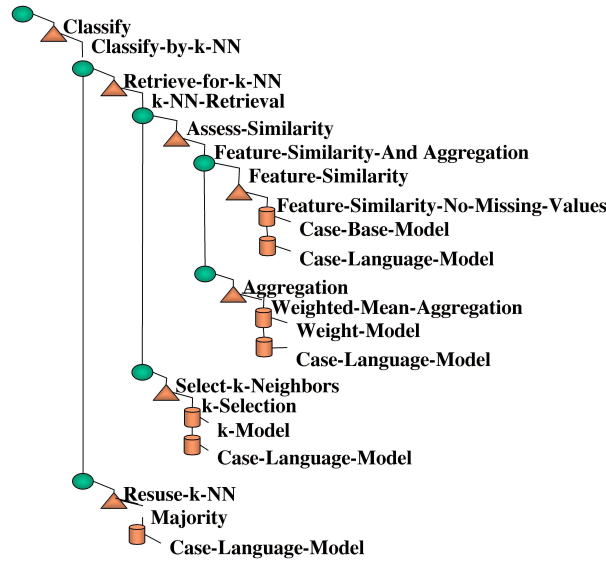


Figure 16: A CBR configuration to perform classify task using Weighted-Mean and Majority Strategy.

Domain Models to be used. In our system we have three DMs: a *Drums-Case-Language* with all the attributes that describe the cases, the *Drums-Case-Base* that stores all the cases and a *Drums-k-Model* with a fixed k value of 5. In the first step of the Enabling process the user links the Domain Models needed by the configuration with the concrete Domain models available; the links are the following: 1) the *Case-Language-Model* is linked to *Drums-Case-Language*, 2) the *Case-Base-Model* is linked to *Drums-Case-Base*, and 3) the *K-Model* is linked to *Drums-k-Model*.

The engineer needs also a *Weight-Model* for the configuration presented above. Let us suppose that the CAT-CBR platform has configured this *enabler*, during the Configuring process, using the RLM-Distance[18] method to acquire the model. This method also needs two Domain Models: a *Case-Language-Model* and a *Case-Base-Model*. At this point, the CAT-CBR platform asks the user for these two domain models and link them to the configured *enabler* and enact the enabler in the same way that we will show in the next section. The outcome of this enacting process is the *Weight-Model* needed by the configuration. This new model can be saved by the user in order to use it in new applications.

Finally the CAT-CBR platform loads all the domain models linked during this enabling process. These loaded domain models jointly with the configuration constitutes the *Enabled system*. This *Enabled system* is to be enacted into the final application.

5.4 Enacting and Testing CBR systems

The goal of the Enacting process is to convert an Enabled system into an executable CBR system. In this process the CAT-CBR platform has to join the

specification of the system represented in the Enabled system with the implementation of the methods.

As we have presented previously an implemented method has the following parameters: inputs, domains and subtasks (when the method is a *Problem Decomposer*).

As it has been presented in section 3.3.3, the *enacting process* is done by applying the implementation of the different methods with the correct parameters in order to carry out the specified application. Let us see this process in some of the tasks and methods of the enabled system presented before.

The operationalization of the application described in figure 16 will start by solving the task *Classify*. To solve this task the CAT-CBR platform will search in the configuration which PSM solves the task; in our case, the PSM that solves the Task *Classify* is *Classify-by-k-NN*. To enact this PSM the platform will apply its implemented method by adding the inputs and parameters in the following way:

```
(Classify-by-K-NN-Method :inputs ((Problem CBR-Case)) :domains nil)
```

This PSM has two subtasks: *Retrieve-for-k-NN* and *Reuse-k-NN*. While the method is applying, it will need to solve these new subtasks, that appear in the implemented code in the following way: (*use-subtask Retrieve-for-k-NN Problem*) and (*use-subtask Reuse-k-NN Problem Retrieved-Cases*). To solve these two subtasks, the CAT-CBR platform will perform in the same way like with the task *Classify*; so the platform will search the PSMs that have to be enacted to solve the tasks, and it will replace the inputs and parameters correctly.

When the PSM to be operationalized needs Domain-Models, for example in the PSM *Majority* of the previous configuration, the CAT-CBR platform will search also which Domain Models are needed and it will add them as parameters. In our case:

```
(Majority-Method :inputs ((Problem CBR-Case) (Retrieved-Cases CBR-Case))
:domains (Drums-Case-Language Case-Language-Model))
```

The CAT-CBR platform performs this process through all the configuration so it is finally enacted.

CAT-CBR gives the user the options of evaluating an application in two ways: first, the accuracy of an application, and second to compare two or more applications with the same general objectives and applied in the same domain. For the first option, CAT-CBR-platform provides two techniques: *Leave-One-Out* and *n-Fold-Cross-Validation*. To compare applications, CAT-CBR-platform provides two techniques: *Sign-Test* and *Signed-Ranked-Test*.

Here we will present some results of the tests done with the previous configurations. Figure 17 shows the results of a collection of configurations with different number of relevant attributes. We notice that the best accuracy is achieved using 75 attributes. We also notice that with all the features we only are introducing noise to the system. This can help the engineer in order to extract the relevant information of the sounds and allow him to determine which is the best configuration. These results are obtained performing five 10-fold cross-validation over the case base.

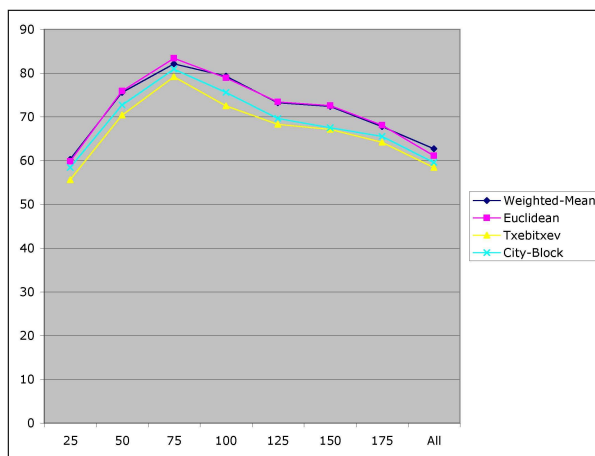


Figure 17: Results of applying the different configurations with different sets of relevant attributes.

6 Conclusions and future work

We can differentiate two different kinds of users that work with and apply CBR: application engineers and scientific engineers. Each of them face different problems. 1) Application engineers use CBR techniques to apply them in some specific domain; these users don't have deep knowledge of the specific characteristics of the different CBR methods and have to face the challenge of selecting methods without knowing which ones will better apply to their domain of interest. 2) Scientific engineers develop CBR techniques, and their challenge is being able to apply them to different domains; they also need to test different methods in different domains in order to assess the characteristics of each method.

In this work we have presented a component based platform in order to develop CBR systems and face the above-mentioned problems. Thus, this platform deals with problems in the following way: concerning the first problem, the platform provides the user with the necessary tools that will allow him to apply CBR techniques to his domain with no need to have deep knowledge of these techniques. Therefore, in order to configure an application, the user just needs to specify a few general characteristics of the desired application and the working domain in particular, i.e. the knowledge he has about it and the attributes characterizing the cases. Taking this into consideration, the platform will guide the user to provide him with a CBR system that suits his needs.

To deal with the second problem, the platform introduces a Component Description Language (CDL) that allows the scientific engineer to specify the different methods. The specifications are independent from the application domain, which allows for the reusability of these methods and techniques. The CDL developed presents some characteristics common to other approaches like UPML and Components of Expertise, but also eliminates some of the weaknesses present in these approaches, such as, for example, the lack of validity of UPML in domains generated dynamically.

There are commercial platforms that develop CBR systems, as seen in section 2.1.3. These applications are limited to certain number of techniques. In CBR-

Works the reuse rules are limited to rules of kind "if condition then action". Others just use nearest-neighbor techniques (KATE), or don't admit structured cases (ReMind). The goal of the CAT-CBR platform is not solving all these weaknesses in order to get a general platform for developing CBR systems, but providing the user with a simple way to experiment with CBR techniques within his domain of interest and to define new CBR techniques.

Concerning the development process of a CBR system, the CAT-CBR platform uses a three-step process. As we have presented these three steps are: Configuring, Enabling and Enacting. These approach is compatible to the INRECA methodology. In INRECA (eight steps) the first step is to characterize the new project; in our platform the engineer has to specify the application domain as well as the requirements of the desired CBR system, requirements which constitute the Problem Requirements of the Configuring process. The next six steps of INRECA are done in our platform during the Configuring process. Once the engineer has configured a CBR system, he has to enable this configuration and link it to the application domain; this step, which does not appear in the INRECA methodology, allows the engineer to directly reuse a configuration in different domains. Finally, like the INRECA methodology, the CAT-CBR platform yields an Enacting process that directly transforms the enabled configuration into a real executable CBR application which the engineer can test, store and reuse.

The platform is based on a components library that covers three of the four states in the CBR cycle: Retrieve, Reuse and Retain. Classical methods from the CBR literature, such as k-nearest neighbor or decision trees, can be found in this library. Other techniques developed at the IIIA for structured cases (similarity measures, retrieve, and reuse) and retain distribute techniques have also been included. We do not intend to specify and include all the CBR techniques in this library, but only representative techniques to serve as an example of a CBR component specifications library. Future work will aim to include an editor for the user to specify components more friendly than it is done through the existing hard coding process.

We have also presented an example of CAT-CBR work on an application domain. This application domain is a musical domain of discrete sounds which is also part of the *Tabasco* project where the present work is being carried out.

As future work we have two working lines. First, we want to incorporate knowledge intensive components to the CBR Library. The main techniques that we want to incorporate in the CBR Library are:

- First, we want to specify the different tasks and methods used in the CREEK platform [1]. The CREEK platform is based on causal models and proposes a collection of methods and models to perform explanations using causal relationships and past cases. Due to my stay at NTNU (Norwegian University of Science and Technology) during last year, we have started this specification and we have to finish with its implementation and experimentation with different domains.
- Second, we are incorporating the collection of retrieval components, developed previously at the IIIA, based on the notion of *perspectives* [4]. Perspectives is a mechanism developed to describe declarative biases for case retrieval in structured and complex representations of cases.

- Third, as traditionally the CBR community has confronted the Reuse step with developing techniques depending of the domain, we want to deepen in techniques more independent from the domain to solve problems of configuration. For this work, we have started the specification of the Constructive Adaptation approach, remaining its implementation and test with different domains.

The second working line is to study the domain problem of device configuration and develop, as we have done for the musical domain, CBR systems using the CAT-CBR platform. This domain will need structured information techniques, as well as, an incremental construction of the solution.

A Publications

Following we enumerate the publications written during this research:

M. Gomez, C. Abasolo, E. Plaza (2003), Open, Reusable and Configurable Multi-Agent Systems. Third price in the AgentCities Technology ATC03, in the Infrastructures category.

M. Gomez, C. Abasolo (2003), A general framework for meta-search based on query weighting and numerical aggregation operators. In Bouchon-Meuner, B. Foulloy, L and Yager, R.R. (eds.). Intelligent Systems for Information Processing: From Representation to Applications, pp. 129-140, Elsevier Science.

C. Abasolo, E. Plaza, J. Arcos (2002), Components for Case-Based Reasoning. In Proc.Congrés Catalá d'Intel.ligència Artificial, CCIA 2002.

M. Gomez, C. Abasolo, E. Plaza (2002), Problem-solving Methods and Cooperative Information Agents. In Proc. Special Issue of the International Journal on Cooperative Information Systems vol. 11(3), 2002.

M. Gomez, C. Abasolo (2002), Improving meta-search by using query-weighting and numerical aggregation operators. In Proc. Information Processing and Management of Uncertainty conference, IPMU 2002.

C. Abasolo, M. Gomez (2002), A framework for meta-search based on numerical aggregation operators. In Proc.Congrés Catalá d'Intel.ligència Artificial, CCIA 2002.

M. Gomez, C. Abasolo, E. Plaza (2001), Domain-independent ontologies for cooperative information agents. In Proc. Fifth International Workshop Cooperative Information Agents CIA-2001. Lecture Notes in Artificial Intelligence, LNAI 2128, p. 118-129. Springer-Verlag.

C. Abasolo, M. Gomez E. Plaza (2001), Agents d'informació independents del domini. In Proc.Congrés Catalá d'Intel.ligència Artificial, CCIA 2001.

J. M. Abasolo, M. Gomez (2000). MELISA: An ontology-based agent for information retrieval in medicine. Proceedings of the First International Workshop on the Semantic Web (SemWeb2000). Lisbon, Portugal, pp 73-82.

B Configuring process in detail

The goal of the *Configuring process* is to determine which components will be applied to solve each task; this goal reduces our problem to a component selection. After studying different approaches, we decided to develop a CBR system to configure an application, which we call *configuring process*.

PROBLEM REQUIREMENTS	
TASK	TASK
PRECONDITIONS	FORMULA
POSTCONDITIONS	FORMULA
INPUT	SIGNATURE-ELEMENT
KNOWLEDGE-ROLES	DOMAIN-MODEL

Figure 18: Features of the *Problem Requirements* sort.

Given an *Application Requirements*, specifying the user requirements, the configuring process uses the components described in the CAT-CBR Library, to elaborate a configuration of an application that satisfies the requirements.

To present this approach we go through the following sections. In section B.1 we will define the elements needed in the configuring process. Then, in section B.2 we will see the general process of configuring. Section B.3 explains the three different strategies implemented in the CBR system. Finally, in section B.4 we present an example of the configuring process in the domain of the Tabasco project.

B.1 Elements for the configuring process

The three elements needed in a configuring process are: the *Application Requirements*, the *Configuration* and the *State*. The configuring process starts with a user input, that is what we called *Application Requirements*. These Problem Requirements have all users requirements for the target application. Thus, it has to represent the application inputs, the preconditions and postconditions, the knowledge that is available and the top task to be achieved by the application. All this information is sent to the CAT-CBR platform as an instance of the sort *Problem-Requirements*, shown in figure 18.

The result of the CAT-CBR configuring process is what we called a *Configuration*. A Configuration is a task-method decomposition structure that represents an application. Essentially, given a task, the configuring process has to select a PSM able to achieve the goals of this task. A PSM can be either a Problem Decomposer or a Reasoning Resource. When a Problem Decomposer (PD) is bound to a Task (T), it means that PD achieves T by decomposing it into subtasks. Then the goal of the configuring process is to find PSMs achieving each one of these subtasks. When the PSM bound to a Task is a Reasoning Resource (RR), it means that the Task is directly achieved by the RR. In some situations a RR may need some *Domain Models* to achieved the Task. Domain Models represent some knowledge given by the user and the properties of this knowledge.

In addition to the Problem Requirements and the Configuration, a third element needed during the configuring process is the notion of *State* (or *partial configuration*). The main issue to be represented in a State is the set of task-method bindings used in a partial configuration. That is, a state only represents the subset of tasks included in a configuration and the tasks that have some

STATE	
GOALS	FORMULA
ASSUMPTIONS	FORMULA
INPUTS	SIGNATURE-ELEMENT
MET-GOALS	FORMULA
MET-ASSUMPTIONS	FORMULA
MET-KNOWLEDGE	USED-KNOWLEDGE
OPEN-BINDINGS	TP-BINDING
OPEN-KNOWLEDGE	SIGNATURE-ELEMENT
TOP-TASK	TASK
TP-BINDINGS	TP-BINDING
CURRENT-TP-BINDINGS	TP-BINDING

Figure 19: Features of a *State*.

binding with a specific PSM at a given point in time of the configuring process.

The second important issue within a State is to determine which pre- and post-conditions of the Problem Requirements are satisfied by the components involved in a partial configuration—and which have not yet been satisfied.

Figure 19 shows the general representation of *state*. *Goals* and *assumptions* are those postconditions and preconditions not yet satisfied in the current state; the *met-goals* and *met-assumptions* are the preconditions and postconditions satisfied by the partial configuration. The *open-bindings* hold the tasks that have no PSM bound to them. The *open-knowledge* are domain models needed by some *Reasoning Resource* that are not available in the *Problem Requirements*, while *Met-Knowledge* represents the domain models used by a Reasoning Resource. The *tp-bindings* are pairs of Task and PSM that are bound.

B.2 A general overview of configuring process

The goal of the configuring Process is, given some requirements of a goal application, to choose some components (Tasks, PSMs and Domain Models) which satisfy this requirements. The approach we have chosen to deal with this problem is to take the configuring process as a search process in a space of states.

The process starts with an *Application Requirements*, then it generates an initial state. From the initial state, the configuring process generates new states until one of these new states is considered as final state. A state is a *final state* when there are no *open-bindings* neither *open-knowledge* and all the *preconditions* and *postconditions* of the Problem Requirements are satisfied by the *met-goals* and *met-assumptions* of the state. At the end of the configuring process, the resultant configuration is presented to the user. Figure 20 shows an example of a configuration. The left part shows the task-method decomposition tree; the center part shows the preconditions required by the configuration; and the right

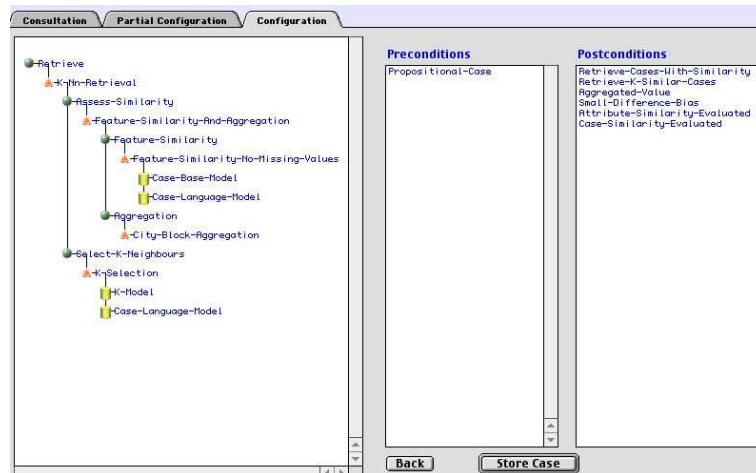


Figure 20: Interface that shows a final configuration.

part shows the postconditions satisfied by the output of the configuration.

Now, we will see how new states are generated from a given state. The open bindings are tasks with no PSM bound to them. So, given a task without bound PSM, we will retrieve from case base those PSMs that “match” with the task.

For all the PSMs that match with the task a new state is generated, updating the bindings, the *met-goals* and *met-assumptions*.

B.3 Three configuring strategies

We have seen the general idea of configuring as a search process. The basic process is, from an initial state, to explore successor states in some order until reaching a final state. We have implemented three different strategies for the search process, depending on the kind of user (expert or not) and the knowledge the system has (past configurations). First, the Search and Subsume strategy is appropriate for non expert users and even without using past configurations of the system. The Case Based strategy is appropriate when the (expert or non expert) user wants to reuse past configurations. Finally the Interactive strategy is oriented to expert users and is allowed to use past configurations. Following these three strategies are presented in detail.

Search and Subsume Strategy

The Search and Subsume Strategy implements a depth first strategy for searching. This strategy uses subsumption for retrieving PSM components from the Library to achieve an open task. The retrieved PSMs are those that satisfy the matching criterion explained before. For each retrieved PSM, a new state (called successor state) is generated. This new state represents a new partial configuration where all relevant information is updated.

The newly generated states are added to a stack of open states. The next state to be explored is always the head of this stack, thus performing depth-first search in the state space.

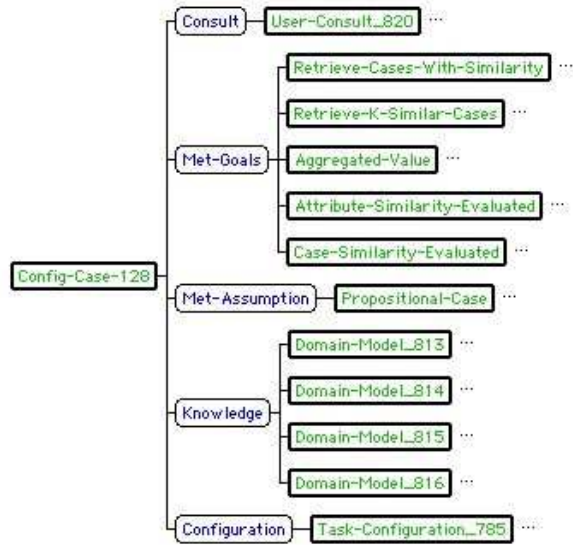


Figure 21: Partial browse of a Configuration Case for CAT-CBR Library.

Case-Based Strategy

The Case-Based Strategy implements a best-first search process in the state space. There is a heuristic function, that assesses which state is the “best”, based on previous solved problems—i.e. previous configurations.

The Case-Based Strategy retrieves PSM components from the Library for achieving an open task, as the Search and Subsume Strategy. From the set of retrieved PSMs, the Strategy generates a set of successor states. The Case Based Strategy uses the past configurations for ordering the successor states according to some measure of similarity to the current state.

This requires that the past configurations have to be stored into a case base. These past configurations are represented as feature terms. Figure 21 shows a Configuration Case specified as a feature term. Configuration features are the following: *Consult* that contains the Problem Requirements; *Met-goals* that contains the postconditions of the configuration; *Met-assumptions* that contains the preconditions of the configuration; *Knowledge* that contains the Domain Models used in the configuration; and finally, the *Configuration* that contains the Task-PSM decomposition that solves the consult.

First the Case-Based Strategy compares the requirements of the current problem with the requirements of each Configuration Case. This comparison is done using a similarity measure, called SHAUD [7]. SHAUD evaluates the similarity between two structures within feature terms. We had to develop this similarity measure for this Strategy, because the measures used in CBR are defined by vectors and what we need is a similarity between more complex

structures with sorts⁵. The result is a ranking of Configuration Cases that will be used to order the states.

For each open state the newly added hypothesis (the value of the feature *Current-Tp-Binding*) has to be considered. Then it must be checked in which configuration case the new hypothesis appears as a part of the configuration, and taken the one that appears in the most similar (to Problem Requirements) configuration case. The similarity value of this configuration case is the heuristic value assigned to the state.

Then, once we have reordered the states, using the heuristic value, the search process selects the state to expand the one with maximum heuristic value. What it is got with this strategy is a best-first search where the “best ordering” is given by the similarity degree that has just been presented.

The process followed by the Case-Based Strategy is the *constructive adaptation*. *Constructive adaptation* is a technique for case reuse that can be applied to synthetic tasks in CBR. The main idea is to address synthetic tasks as a search process, as classically done, and guide the search process using cases (embodying the problems solved in the past). For a particular domain task, the main design issue is to decide how to build the heuristic measure of best-first search from the similarity measure that compares the current problem with cases (past solved problems).

Interactive Strategy

This strategy is thought for expert users, since the user can choose which component has to be added up to the partial configuration. In order to carry out this selection, the user uses the intermediate features of a component described in the ontology domain.

The Interactive Strategy has a user interface that shows the partial configuration, the available components, some information about the components, etc (see fig 24). The left part of the interface shows the open Tasks. Then, the CAT-CBR platform presents to the user possible PSMs that can be applied for the selected open task. These PSMs are those which satisfy the component matching criterion, as defined in section B.2. The PSMs are ordered by the similarity measure used in the Case-Based Strategy. Then the user selects the PSM to be incorporated to the configuration (Available Components part in fig.23). To make this decision easier the Interactive Strategy presents some extra information about the PSMs (Component Info in fig.23) and the state information (postconditions to be achieved, needed knowledge...).

The CAT-CBR platform gives the user the possibility of taking back his decision (with the *back* button in Fig. 23). This allows the user to try different PSMs for a task.

⁵Most of the CBR applications the IIIA has developed have used retrieval techniques based on “symbolic similitude”, and not in similarity metrics. The reason for this was twofold: first, similarity metrics in CBR basically focus on attribute-value representations, and only a few contributions focus on similarity metrics for complex structures in frame-based representations, second, the notion of antiunification and the use of domain knowledge allowed us to research on the idea of a symbolic representation of similitude.

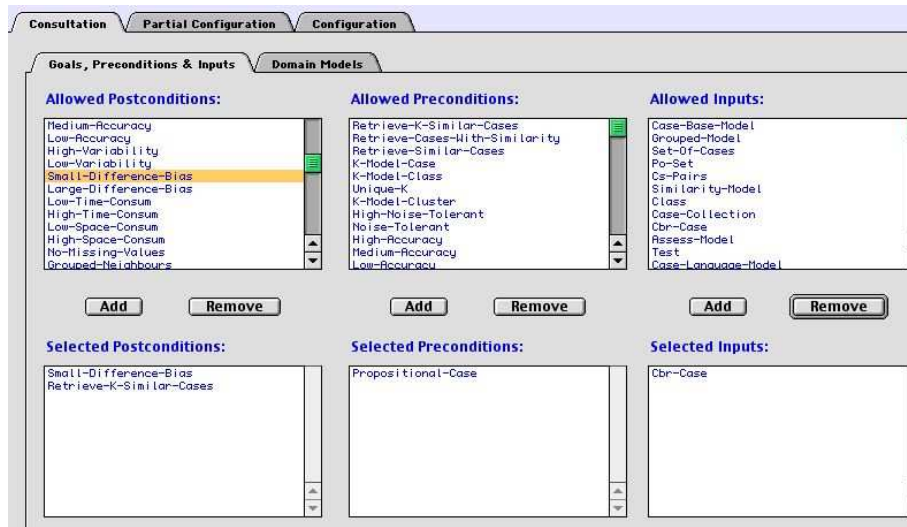


Figure 22: Interface for specifying *Application Requirements*. On top boxes we can see all the *Signature Elements* and *Formula* that the user can select to specify *Application Requirements*. On bottom boxes the user selection that specifies the Assumptions, Goals and inputs of the *Application Requirements*.

B.4 An example of configuring a CBR application

In this section we will develop an example of configuration. In particular, we detail how to configure a CBR application using some of the components described in the CAT-CBR Library. In order to configure this CBR application we will use the *Interactive strategy*. Let us suppose that the user wants to develop a CBR system that handles cases with propositional representation. The goal of the CBR system is to retrieve k cases such that small difference with input problem are preferred. The knowledge of this system is a case base and a k -Model. In terms of the *Tabasco* project, the user needs an application to retrieve sounds similar to a new sound; these retrieved cases will include a similarity measure to the new sound and we want to stress small differences. The specification of these requirements is the following:

Application Requirements

Assumptions \rightarrow Propositional-Case

Goals \rightarrow Retrieve-Cases-with-Similarity, Small-Difference-Bias

Input-roles \rightarrow (Problem CBR-Case)

Models \rightarrow (Case-Base-Model (properties Propositional-Case)),
(k -Model (properties Unique- k)), (Case-Language-Model)

tt \rightarrow Retrieve

Notice that domain models (feature **Models** of the specification) contain a *Case-Language-Model*. This model holds aspects about the case representation. This *Problem Requirements* are done through an interface that can be seen in figure 22.

From the *Application Requirements* the CAT-CBR platform generates the

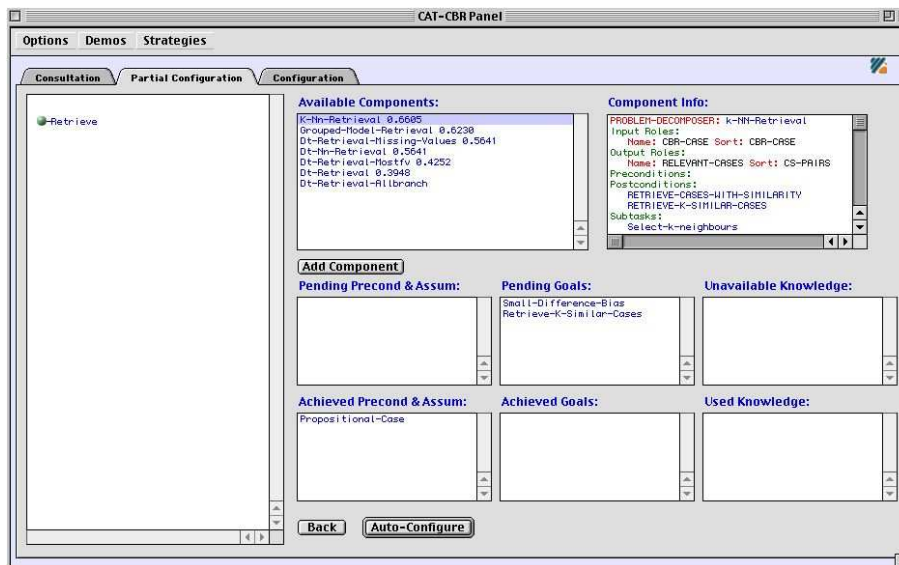


Figure 23: Interactive strategy's Interface showing the current state of a configuration. The current task to configure is the *Retrieve* task, the available components for solving the *Retrieve* task are ordered by similarity and some details about these components. Bottom boxes hold the information about the current state.

following initial state:

State
cpre → Propositional-Case
cpost → <i>Empty-Set</i>
opre → <i>Empty-Set</i>
opost → Retrieve-k-Similar-Cases, Small-Difference-Bias
ot → Retrieve
tp → <i>Empty-Set</i>

The CAT-CBR must choose a PSM to solve the *Retrieve* task. In the Interactive Strategy, the list of PSMs matching with the Task is presented to the user. This list has been ordered taking as a reference the similarity of the consult with previous cases. Figure 23 shows the PSMs that can solve the *Retrieve* task. Notice that this list is ordered according to the similarity of past configurations where the method appears, with the current *Problem Requirements*.

The non interactive strategies would generate a state for each one of the PSMs that can be bound. The difference between these strategies is the order in which the search of successor states will be continued. Meanwhile, the *Case-Based strategy* will evaluate the states in the same order in which they have been presented, the *Search and Subsume strategy* will evaluate the first state generated with no order.

Let us suppose that in the *Interactive strategy*, the user chooses the *K-NN-Retrieval* as the PSM for solving the *Retrieval* task. *K-NN-Retrieval* is a prob-

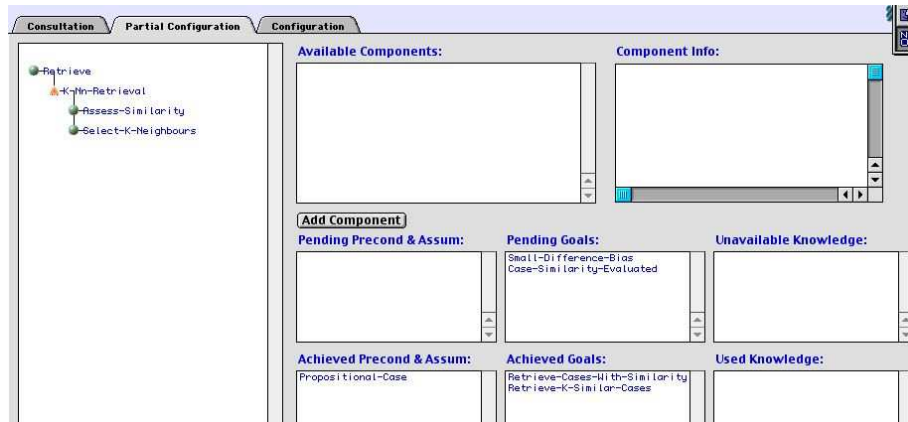


Figure 24: Interactive strategy's Interface showing the partial configuration for the *Assess-Similarity* task. The unique PSM available in the library for solving this task with propositional cases is *Feature-Similarity-And-Aggregation*.

lem decomposer with two subtasks: *Assess-Similarity* and *Select-K-Neighbours*. When this problem decomposer is introduced, the next state is the following:

State

- cpre** → Propositional-Case
 - cpost** → Retrieve-k-Similar-Cases, Retrieve-Similar-Cases-with-similarity
 - opre** → *Empty-Set*
 - opost** → Small-Difference-Bias
 - ot** → Assess-Similarity, Select-k-Neighbours
 - tp** → (Retrieve, k-NN-Retrieval)
-

This new state is presented to the user through the informative panel shown in figure 24. The achieved goals (*Retrieve-Cases-With-Similarity* and *Retrieve-k-Similar-Cases*) are represented in this panel. In the pending goals box there are the *Small-Difference-Bias* (from *Problem Requirements*) and *Case-Similarity-Evaluated* that is the new goal generated from the postcondition of the *Assess-Similarity* task.

The configuring process continues by selecting PSMs for each open tasks (i.e. *Assess-Similarity* and *Select-k-Neighbours*) until a final state is reached. In our example the final state is the following:

State

cpre, **cpost**, **opre**, **opost**, **ot**, **tp** **cpre** → Propositional-Case
cpost → Retrieve-k-Similar-Cases, Retrieve-Similar-Cases-with-similarity
Aggregated-Value, Small-Difference-Bias
Attribute-Similarity-Evaluated, Case-Similarity-Evaluated
opre → *Empty-Set*
opost → *Empty-Set*
ot → *Empty-Set*
tp → (Retrieve, k-NN-Retrieval),
(Assess-Similarity, Feature-similarity-and-aggregation),
(Select-k-Neighbours, k-selection),
(Aggregation, City-Block-Aggregation)
(Feature-Similarity, Feature-Similarity-No-Missing-Values)

The final state represents a complete configuration that satisfies the requirements of this *Problem Requirements*. This final configuration was shown in figure 8. In this configuration we see that *Assess-Similarity* task has been bound to the *Feature-Similarity-And-Aggregation* method. This method is a problem decomposer PSM with two subtasks: 1) *Feature-Similarity* task that is bound to the *Feature-Similarity-No-Missing-Values* reasoning resource that needs two models: *Case-Base-Model* and *Case-Language-Model*; and 2) *Aggregation* task that is bound to the *City-Block-Aggregation* reasoning resource. On the other hand, the *Select-k-Neighbours* task is bound to *k-Selection* reasoning resource that needs two models: *k-Model* and *Case-Language-Model*.

References

- [1] Agnar Aamodt. Explanation-driven case based reasoning. In S. Wess, K.D. Althoff, and M. Richter, editors, *Topics in Case-Based Reasoning*, Lecture Notes in Artificial Intelligence, pages 274–288. Springer-Verlag, 1994.
- [2] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications*, 7(1):39–59, 1994. online at <url:http://www.iiia.csic.es/People/enric/AICom_ToC.html>.
- [3] K. Althoff, M. Nick, and C. Tautz. Cbr-peb: An application implementing reuse concepts of the experience factory for the transfer of cbr system know-how.
- [4] J. L. Arcos and R. López de Mántaras. Perspectives: a declarative bias mechanism for case retrieval. In David Leake and Enric Plaza, editors, *Case-Based Reasoning. Research and Development*, number 1266 in Lecture Notes in Artificial Intelligence, pages 279–290. Springer-Verlag, 1997.
- [5] E. Armengol and E. Plaza. Lazy induction of descriptions for relational case-based learning. In *Submitted*, 2001.
- [6] Eva Armengol. *A Framework for Integrating Learning and Problem Solving*. PhD thesis, Universitat Politècnica de Catalunya, 1997.

- [7] Eva Armengol and Enric Plaza. Similarity assessment for relational cbr. In *Proceedings ICCBR 2001*, LNAI. Springer Verlag, 2001.
- [8] E. Auriol, S. Wess, M. Manago, K.-D. Althoff, and R. Traphöner. Inreca: A seamless integrated system based on inductive inference and case-based reasoning. In *Case-Based Reasoning Research and Development*, number 1010 in Lecture Notes in Artificial Intelligence, pages 371–380. Springer-Verlag, 1995.
- [9] B. Chandrasekaran. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design. *IIIE expert*, 1:23–30, 1986.
- [10] Breinman L. et al. Classification and regression trees. *Wadsworth International*, 1984.
- [11] D. Fensel, V. R. Benjamins, M. Gaspari S. Decker, R. Groenboom, W. Grosso, M. Musen, E. Motta, E. Plaza, G. Schreiber, R. Studer, and B. Wielinga. The component model of upml in a nutshell. In *Proceedings of the International Workshop on Knowledge Acquisition KAW'98*, 1998.
- [12] Dieter Fensel and Enrico Motta. Structured development of problem solving methods. *Knowledge and Data Engineering*, 13(6):913–932, 2001.
- [13] J. H. Gennari, S. W. Tu, T. E. Rothenfluh, and M. A. Musen. Mapping domains to methods in support of reuse. *International Journal of Human-Computer Studies*, 41:399–424, 1994.
- [14] Mingers J. Expert systems-rule induction with statistical data. *Journal of the Operational Research Society*, 1987.
- [15] Quinlan J.R. Inducing of decision trees. *Machine Learning*, 1, 81-106, 1986.
- [16] Quinlan J.R. Simplifying decision trees. *International Journal of Man-Machine Studies*, 1987.
- [17] Janet Kolodner. *Case-based reasoning*. Morgan Kaufmann, 1993.
- [18] Ramon López de Mántaras. A distance-based attribute selection measure for decision tree induction. *Machine Learning*, 6:81–92, 1991.
- [19] E. Plaza and J. L. Arcos. Construtive adaptation. In S. Craw and A. Preece, editors, *Advances in Case-Based Reasoning. Proc. 6th ECCBR 2002*, Lecture Notes in Artificial Intelligence, pages 306–320. Springer-Verlag, 2002.
- [20] A. Puerta, J. Egar, S. W. Tu, and M. A. Musen. A multiple-method knowledge acquisition shell for the automatic generation of knowledge acquisition tools. *Knowledge Acquisition*, 4:171–196, 1992.
- [21] Fidel Ruiz, Frank van Harmelen, Manfred Aben, and Joke van de Plassche. Evaluating a formal modelling language. In *Knowledge Acquisition, Modeling and Management*, pages 26–45, 1994.
- [22] Luc Steels. Components of expertise. *AI Magazine*, 11(2):28–49, 1990.

- [23] Niblett T. Construction decision trees in noisy domains. *Progress in machine Learning*, 1986.
- [24] Walter Van de Velde. An overview of commonKADS. In J. Breuker and W. Van de Velde, editors, *CommonKADS library for expertise modelling*. IOS Press, 1994.
- [25] Bob Wielinga, Guss Schreiber, and Joost Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5–54, 1992. Special Issue ‘The KADS approach to knowledge engineering’.
- [26] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.