A Hybrid Model for Case Indexing and Retrieval in Building Design

Zeyno Aygen

Submitted to the School of Architecture of
Carnegie Mellon University in fulfillment of the requirements
for the degree of Doctor of Philosophy

School of Architecture and
Institute of Complex Engineered Systems (ICES)
Carnegie Mellon University

Abstract

Precedents are commonly used as a means of investigation and inspiration in architectural design. Designers often refer to past solutions when they are confronted with a similar problem context. This offers a promising application domain for recent research in AI that introduces the technique of case-base reasoning (CBR) to the design domain. The computational support in a case-based design (CBD) system involves the recall and re-use of past solutions in new problem situations. An efficient indexing of past solutions is crucial to computational design systems performing complex retrieval on large case-bases. This research suggests an hybrid approach to the indexing and retrieval of design precedents. The suggested approach accounts for the issues of classification manifested in architectural discussions on type and CBD literature. The indexing scheme integrates description-logic based representation for classification and an object-based representation for precedents. The hybrid scheme constitutes a basis for the implementation of a generic case indexing and retrieval mechanism for SEED: a Software Environment to support the Early phases of building Design. The suggested classification and case-base functionalities are supported by two distinct engines: SEED-KBC (SEED - Classification Knowledge-Base) and SEED-CBD (SEED - Design Case-Base), and accessed through engine specific APIs.

Acknowledgment

I wish to express my gratitude to my advisor and mentor Ulrich Flemming, for his support and contributions. I have learned a lot from his working style: promoting self confidence and autonomy. I am also grateful to my thesis committee Omer Akin and Steven Fenves for their patience, valuable suggestions and knowledge. I wish to extend my thanks to all the *seedlings* for the inspiring discussions and positive criticisms. I am also thankful to my friends in Pittsburgh for making this adventure a delightful one. Most of all, I would like to thank my parents, Sezar and Aynur, and my sister Artemis, for their unreserved love and support. Their encouragement has been the most important driving force for my work.

Table of Contents

List of Figures

List of Tables

Introduction

Designers often refer to a previous design solution when they find a fit between the existing problem situation and previously encountered problems. This use of precedents is common in traditional design practice, and hence constitutes a major motivation for a new generation of computational design systems. This study provides a computable model for the recall and re-use of precedents. In this introductory chapter, I outline the major characteristics of the re-use of precedents in traditional design practice and identify where and how the currently available computational support fails to conserve these characteristics. I set the scope for developing a computational model which addresses this shortcoming and conclude this section with an overview of the thesis contents.

I . 1 Overview

Recent developments in computational design have extended the case-based design (CBD) approach, a design specific application of the AI paradigm of case-based reasoning (CBR), to the context of architectural design. CBD can be conceived as a continuation of the use of *precedents* in design. The term *precedent*, introduced to the computational design literature by Oxman (1994), refers to a representation of the knowledge about a past design in a form that makes it “re-usable” in new, but similar problem situations. The use of the term in this study, however, does not inherit Oxman’s knowledge organization scheme.

CBD approaches differ from other design methods in the way they make use of specific knowledge about previously encountered problems instead of relying on generic knowledge represented by rules or grammars. The specific knowledge is structured in the form of *cases*, which - taken together - constitute a case-base or case library. CBD systems recall these cases to use in new problem situations. A CBD system uses a case-base with special retrieval capabilities instead of a generic database as a means of storing past problem solving episodes. These episodes are retrieved based on their similarity to the current problem situation, where the assessment of similarity often involves more than a purely syntax-driven matching between attributes.

I . 2 Motivation

SEED requires that the case-base indexing and retrieval capabilities make use of the information available in the computational representation of a design case as well as the thematic information which may have to reside outside the case-base scheme. Currently available case-base design systems do not offer an indexing and retrieval mechanism with the capability to issue and combine structural queries with queries based on classification. At the same time, SEED provides a rich context to address general issues that arise for CBD in building design.

The key characteristics of the reuse of design precedents can be identified as the *representational* and *recalling flexibility*. These two characteristics have a major impact on the representation and classification schemes of a computable memory model.

- **Representational flexibility:** The reuse of a design solution is not limited to a specific level of abstraction. A precedent could be as specific as a window detail or as abstract as a circulation diagram[1].
- **Recalling flexibility:** The recalling of a precedent may be based on a piece of information which may not be available at the time the precedent is registered as a solution. As the designer's memory acquires relevant information, it dynamically re-registers the existing problem episodes to reflect the changes[2].

The majority of CBD systems has already attempted to address the former characteristic, yet has remained indifferent to the latter. One of the major motivations for this study is the lack of an efficient, yet expressive modeling approach that is flexible in both representing and classifying precedents. For many CBD systems, indexing is considered within the context of case representation. The range of modifications that can be applied to classifications in this approach is, therefore, bounded by the scheme used to represent cases. This causes these indexing mechanisms to be inflexible. In the following chapters, I argue with respect to architectural typology and CBD (Aygen, Z. et.al., 1998) that

- classifications in a case-base must allow for modifications if the CBD system is expected to incorporate new information on cases. Therefore, case retrieval mechanisms may have to cope with partial index descriptions and multiple classifications, which appears to be true for architectural design.

1. Different levels of abstractions are typological levels which can be defined as *scales of planning in which the design decisions present a unified system of choices* (Leupen, B. et.al., 1997)
2. Dynamic memory (Kolodner, 1991).

- classifications may have to incorporate thematic features and features reflecting subjective judgements on cases. Often these features cannot be derived from the symbolic representation of a design precedent.

- classifications speed up the retrieval of cases by allowing the system to perform needed matches only on a subset of the case-base. Indexing and classifications are particularly important for CBD systems where the retrieval of complex design representations often introduces computational inefficiencies.

The second motivation is the parallel relation between the issues related to indexing in CBD and the notion of type and classification in the architectural literature. The study of type in architecture is beneficial for this study when the emphasis is given to the cognitive aspects of architectural type (i.e. how a group of persons would recognize the likeness between architectural precedents and conceptually subsume these as being of the same type[1]). The typological discussion hints at the complexities involved in dealing with the categorization of precedents and accounts for some of the issues that need to be addressed in order to build flexible indexing schemes. The study of architectural type provides insights to refine the case indexing and retrieval mechanisms.

I . 3 Research objective and approach

This research is an attempt to provide computational support for the reuse and recall of precedents as part of a case-base design system. In order to address the previously outlined flexibility issues, this approach decouples precedent representation and classification in its data modeling and retrieval techniques. More specifically, it suggests a hybrid memory consisting of

- an object model with object attributes and relations persistently stored in a object oriented database (OODB), and

- a knowledge-base of classifications supporting subsumption inference.

The Software Environment to Support Early Building Design, SEED, provides the first implementation environment and testing ground for the hybrid memory model (Flemming & Woodbury 1995). SEED requires the persistent storage of design precedents and their retrieval for re-use in a multi-functional, multi-user, and distributed design environment. The memory model outlined above has been implemented to realize a case-base design capability in SEED.

1. The account on the cognitive aspects of architectural type (Tezar, P. 1991).

I . 4 Scope

This research concentrates on the recalling of the precedents. Classification, as an inherent mechanism of the proposed model, is one of the major determinants of the thesis scope. Consequently, it has also been the focus for studying

- **Architectural typology:** The account of the typology literature has been limited by the view of classification found in AI's and Cognitive Science. The emphasis has been on the understanding of type as a classificatory device in architectural theory.

- **CBD:** In the CBD literature and research projects, the emphasis has been on the representation, indexing and retrieval of cases. Issues of case adaptation and the problem of creativity in CBD are of considerable significance, but have been ignored for the purposes of this study.

The thesis scope includes the implementation of the hybrid modeling scheme as part of SEED's case-based design engine. However, the choice of CBD techniques is not based solely on SEED's implementation requirements. CBD constitutes a natural implementation environment for demonstrating the suggested model's capabilities in indexing and retrieving precedents. This proposal assumes that the CBD paradigm serves as 'a partial model' for processes involved in design; it does not seek to provide a comprehensive model of the architectural design process. Moreover, the association between architectural typology and CBD does not imply any procedural similarity between case-based design and any typological methodology.

The implementation is in the form of specific APIs (Application Programming Interface) which access distinct databases to build and query classification knowledge-bases (SEED-KBC - SEED's classification knowledge base), and case-bases (SEED-CBD - SEED's case-base design engine). Therefore, the issues pertaining to the design of user interfaces for interacting with these engines are not within the scope of this research. The APIs provide the basic database functionality on which module-specific user interfaces can be build to create and maintain classification knowledge-bases and case-bases.

Chapter II provides a literature survey covering material from architectural typology, cognitive psychology and AI. Chapter III introduces the hybrid model based on the literature survey. Chapter IV elaborates the model and then outlines the implementation of the classification and the case-base engines for SEED. Chapter V describes the SEED-KBC engine, i.e. the classification knowledge-base. Chapter VI introduces SEED's case-base indexing and retrieval capabilities, i.e. SEED_CBD. Chapter VII provides an example of retrieval performed on a demo classification knowledge-base and a sample case-base. Finally, Chapter VIII provides a summary and outlines the contributions and future research areas that can be explored, based on this research.

Background

This section provides a reference framework for a comparative study of the architectural literature on type and the CBR literature on indexing and retrieval. The framework is largely based on Smith and Medin's (1981) work on concept acquisition and categorization and borrows from their terminology. The framework will be used to outline a hybrid model for representation and classification of precedents in the next chapter.

II . 1 Type and classification

Types, in the most generic sense, are categories of thought that can be organized in *generalization* hierarchies. In a generalization type hierarchy, the descriptive features of a type are inherited by its subtypes. The lower levels of the hierarchy contain tokens which denote specific instances of the type concepts[1]. Both types and tokens can be represented at the same hierarchy level, provided that the relationship between a type and its subtype (i.e. is-included-in association) is distinguished from the one between a type and a token (i.e. is-instance-of association).

1.1 Types

Jackendoff (1994), in his theory of types, defines a *type concept* as a finite set of conditions that can be used to categorize novel tokens. Since one can generate new types at will on the basis of encountered tokens, the total set of possible types is infinitely large. Jackendoff argues that the set of possible types can be characterized by a finite set of conceptual formation rules. This constitutes a conceptual formation scheme which is used to select or construct new type concepts. When various type hierarchies need to be integrated in a knowledge representation formalism, a *type lattice* is used instead of a hierarchy tree to organize the type con-

1. Jackendoff's definitions for type, token, and hierarchy (1994).

cepts. The two other important issues, often addressed when dealing with type hierarchies, are exceptions and multiple inheritance related problems. The first issue arises when a subtype fails to possess all the features of its supertype. The latter happens when a particular subtype, by having more than one supertype, inherits conflicting features. In the following section I introduce Prototype Theory (PT), which argues that incorporating prototypes (or exemplars) in a representation scheme may prove to be useful in handling the problems of exceptions and multiple inheritance. The hybrid model which I propose extends the prototype theory by introducing a layer of abstraction for the categorization of prototypes. Section 1.2 reviews some of the influences of prototype theory (PT). Section 1.3 introduces PT through Smith and Medin's framework. FInally Section 1.4 sketches the proposed hybrid approach.

1.2 Sources of PT

Before introducing PT through Smith and Medin's framework, I will consider motives behind the provision of distinct models for episodic (exemplar-based knowledge) and generic knowledge. Tulving's comparative analysis of generic and episodic memory (Tulving, E., 1972), along with various accounts of intensions and extensions in knowledge representation, is essential to the understanding of PT and related AI paradigms like CBR.

The majority of knowledge bases assisting CBR problem solvers follow Tulving's model of memory, where the semantic aspects of human memory are distinguished from the episodic ones. The distinction, however, does not deny the overlap between the semantic and episodic information processing systems. Episodic memory deals with personal experiences and their simple temporal relations, whereas semantic memory deals with language faculties that receive, retain and transmit information about meaning and classification of concepts.

The memory systems differ from each other in the following aspects:

- **Nature of stored information:** Episodic memory deals with the perceptual properties and temporal-spatial relations of the information. Semantic memory, on the other hand, is directly related to thought processes and not to perception. Perceptual features are encoded in the semantic memory only if they uniquely identify the semantic information.

- **Denotative reference of input events:** The reference in the episodic memory is autobiographical, i.e. it goes back to the rememberer's knowledge of accumulated episodes. Inputs to the semantic memory, on the other hand, have cognitive references, which are detached from the autobiographical references. The semantic information is contained in cognitive structures such as concepts (varying in their generality and complexity), relations, quantities and propositions. Consequently, the recording of information in the episodic memory is direct, whereas it is indirect and organized within cognitive structures in the semantic memory.

- **Conditions and consequences of retrieval**: The episodic memory necessitates the direct entry of particular episodes; it cannot infer or generalize. Inference, deduction, generalization, rule application and the use of algorithms are methods used by semantic memory. On the other hand, retrieval operations may not have any effect on the structure of semantic memory, whereas each retrieval operation is entered into the episodic memory as another episode. Retrieval, by providing feedback through these retrieval episodes, may lead to changes in the contents of the episodic memory.

- **Susceptibility to interference and erasure of stored information:** Forgetting is more typical of the episodic memory. The loss of information has been claimed to be caused by an interference in the temporal encoding: episodes, being encoded temporally, are accessible only if an accurate time reference is provided. Almost nothing has been said about the loss of information in the semantic memory.

When Sowa comments on Tulving's categorization (Sowa, J. F., 1984), he bases the distinction between the semantic and episodic memory not on the mechanisms of each memory but on the nature of what is stored. The episodic memory stores detailed facts about individual things and events in the form of episodes (e.g. historical and biographical knowledge). Whereas the information held in the semantic memory, or the *universal principles* in Sowa's account, is more abstract and generic (e.g. knowledge contained in a dictionary). The suggested distinction can be considered in connection with Quine's account of intentions and extensions with respect to the meaning of words (Quine, W. V., 1961). The intension of a word meaning follows from the general principles in semantic memory and the extension of a word is the set of all existing things to which the word applies (i.e. the intension of a word is its definition, and the extension is the set of things in the world to which it applies).

1.3 Smith and Medin's survey

When Smith and Medin attempted to provide a 'systematic' review of the psychology literature on concept acquisition and categorization, they were both stimulated and challenged by what they called the 'muddled' state of the literature. The authors cite a particular instance of such confusion in which no two researchers seem to mean the same thing by the term *prototype*. Their effort in trying to straighten out some of the issues in knowledge representation is a major contribution to the literature. Many researchers refer to their survey of existing views and the corresponding processing models in locating their approach with respect to a framework. In this section, I summarize Smith and Medin's systematic analysis of various approaches to represent concepts and categories.

1.3.1 The classical view

Smith and Medin collect the common assumptions of the philosophically oriented studies of language (e.g. Katza, 1972, 1977, Fodor, 1975), linguistic studies (Lyons, 1968, Bierwisch, 1970, Bolinger, 1975), psycholinguistics (Fodor, Bever, Garrett, 1974, Miller and Johnson-Laird, 1976, Anglin, 1977, Clark and Clark, 1977) and the psychological studies of concept attainment (Bruner, Goodnow and Austin, 1956, Bourne, 1966, Hunt, Marin, and Stone, 1966)[1] under the *classical view*. In this approach, all instances of a concept share common properties, and these properties are necessary and sufficient to define the concept.

Assumptions

- **Summary representation:** The representation of concepts is the result of an abstraction process; it does not need to correspond to specific instances and applies to all possible test instances.

- **Necessary and sufficient features**: The features of a concept are necessary and sufficient for its definition. Therefore, disjunctive features are not allowed to reside in a concept definition because if an object can either have A or B as a feature set, then none of the features in A and B are necessary.

- **Nesting features in subsets:** A concept A is subsumed by a more general concept B, if A's features are subsumed by B's.

Criticism

- **Exclusion of functional features:** A common criticism is that the classical view deals only with the structural features of concepts. There are, however, concepts that are defined by functional features which are necessary and sufficient. For instance, the classical view would describe a cup by its fixed property of *concavity* and would prohibit a property such as *being used to hold something*. Since the classical view cannot incorporate the functional features, it cannot handle all concepts. Smith and Medin object to this criticism by indicating that there is no assumption in the classical view to prohibit the use of functional properties in a concept definition.

- **Exclusion of disjunctive features:** This criticism responds to the second assumption of the classical view (i.e. the use of necessary and sufficient features to describe concepts). There are cases where people describe concepts by making use of disjunctive features, and the classical view cannot handle concepts described in this fashion. For instance, the material property for a cup can be enumerated by disjunctive features like *made-of-glass, made-of-ceramics, made-of-metal* and so on. None of these properties is necessary and sufficient alone to define a cup. In the classical view, material would not be part of concept description. Smith and Medin consider these criticisms controversial by arguing that there are not many instances of disjunctive concepts in the domain of natural concepts. However, this criticism is still valid for concepts representing artifacts.

1. For a complete listing of these sources refer to Smith, E. E. and Medin, D. L. (1981).

- **Unclear cases**: The classical view assumes that if a concept A is a subset of concept B, the defining features of B are nested in those of A. Given this, it is relatively easy to determine a subset relation. Nevertheless, people are unclear about particular subset relations, or even do not have the same answer to an is-a-subset question when asked at different occasions. The classical view cannot account for such unclear cases. One reason for this is the incompleteness of some concept definitions through missing features. For instance, the reason why many people are not sure about the particular subset relation entailed by the question ‘Is a tomato a fruit?’ is that they are missing some of the defining features of the concept of fruit. Another reason for unclear cases is the possibility of concepts with multiple definitions (e.g. technical vs. common definition examples). Consider the following quote from Smith and Medin’s example of an unclear case caused by multiple definitions:

  ... Thus one might be unsure about what concept a tomato belongs to because a tomato meets the technical definition of a fruit (for example, it has seeds) but <also> the common definition of a vegetable (it plays a particular role in meals).

- **Failure to specify defining features:** This criticism is based on an empirical argument (i.e. concepts may not be expressible in terms of necessary and sufficient features) which contradicts the assumption underlying the summary description. Consider the following quote from Smith and Medin’s example of the concept ‘game’:

  One of Wittgenstein’s (1953) most famous examples was that of the concept of games, and we can use it to illustrate the flavor of this argument. What is a necessary feature of the concept of games? It cannot be the competition between teams, or even the stipulation that there must be at least two individuals involved, for solitaire is a game that has neither feature. Similarly, a game cannot be defined as something that must have a winner, for the child’s game of ring-around-the-rosy has no such feature. Or let us try a more abstract feature - say that anything is a game if it provides amusement or diversion. Football is clearly a game, but it is doubtful that the professional football players consider their Sunday endeavors as amusing or diverting. And even if they do, and if amusement is a necessary feature of a game, that alone cannot be sufficient, for whistling can also be amusement and no one would consider it a game. This is the kind of analysis that led Wittgenstein to his disillusionment with the classical view.

1.3.2 The probabilistic view

Smith and Medin group the spreading activation model of Collins and Loftus, 1975, the property comparison model of McCloskey and Glucksberg, 1979, the simple distance model implicitly used by Hyman and Frost, 1975, and some other models in research on both artificial and natural concepts under the probabilistic view[1]. The common claim of these specific models is that the instances of a concept vary in the degree to which they share certain properties, and consequently vary in the degree to which they represent the concept.

The following two assumptions are accepted by the majority of these models, and characterize the probabilistic view:

- The representation of a concept is a summary description of an entire class.
- The representation of a concept cannot be restricted to a set of necessary and sufficient conditions (it is a measure of central tendency instead).

The authors identify various approaches under the probabilistic view: the featural, dimensional and holistic views. In the dimensional approach, each concept depicts the average or mean dimension values of a class. In the featural approach, each concept represents the modal features of its class. The holistic approach uses templates (an isomorphic and unanalyzable representation of a holistic property[1]) in representing concepts of concrete objects. This section provides a review of the featural approach, which is the most representative of the probabilistic view.

Assumptions

- **Summary representation:** The summary representation is an abstraction and may not be realizable as an instance. It is used to decide whether an instance is a member of a concept.
- **Non-necessary features:** The features that represent a concept are salient ones that have a substantial probability of occurring in the instances of a concept. The probability of a feature to be salient is updated with each encountered concept instance. The features that seem to appear in most of the instances are likely to be considered salient features. One important aspect of the featural representation is that continuous properties like size are represented discretely by either defining a set of possible sizes (e.g. {small, large, medium}) or by introducing nested features that provide preciseness to the roughly defined feature (e.g. a *small size* feature is nested in a *medium size* feature which is in turn nested in a *large size* feature). With this assumption, non-necessary features are now permissible in categorization.
- **General processing:** In order to determine whether an instance belongs to a particular concept or a concept is a subset of another concept, features are compared and the matched feature weights are added to a weight counter, which is checked against a membership threshold value.

1. For a complete listing of these sources refer to Smith, E. E. and Medin, D. L. (1981).
1. A detailed description of the concept *template* can be found in Smith, E. E. and Medin, D. L. (1981).

Based on the assumptions stated above, the authors reconsider the problematic aspects of the classical view:

- **Disjunctive concepts**: Since the category membership is based on a weighted sum of features, and not on sufficient and necessary features, in the general featural model the same sum can be obtained by different combinations of features and feature weights. For example, the probabilistic view allows the concept ‘furniture’ to be a disjunctive one, since different combinations of features for ‘rug’ and ‘table’ can match those of ‘furniture’. Yet, the degree of disjunctiveness is considerably small.

- **Unclear cases**: The classical view’s subsumption algorithm, which is used to test for concept membership, fails to account for unclear cases. The probabilistic view, on the other hand, offers two explanations for this situation: the accumulation of a membership value very close but less than the membership criteria threshold; and accumulating close or equal membership values for the same membership test.

- **Failure to specify defining features**: This problem is naturally avoided because necessary and sufficient features are not assumed to define a concept.

- **Simple typicality effect**: Typical members are categorized faster than atypical members. This effect doesn’t challenge the classical view, but it has been given a natural explanation through the probabilistic view by an additional assumption: The typicality of a concept can be measured by the weighted sum accumulated through a match with the parent concept. Consider, for instance, the concepts ‘robin’ and ‘chicken’ as candidates for being typical of the concept ‘bird’. The accumulated weight of non-necessary (e.g. *flies* and *sings*) and necessary features (e.g. *feathered* and *winged*) for ‘robin’ is higher than for ‘chicken’. ‘Robin’ is, therefore, more typical of ‘bird’ than ‘chicken’.

- **Determinants of typicality**: Following the typicality assumption stated in the previous item, the probabilistic model also gives an account for typicality (an item is a typical member of a concept to the extent that it contains features shared by many other members). The typical member inherits the largest set of features from the parent concept. Since all concept members inherit the parent concept’s features, the typical concept is likely to contain the largest number of features shared by other members.

- **Use of non-necessary features**: Non necessary features are allowed in the concept definition; therefore, the problem is avoided from the beginning.

- **Nested concepts**: The probabilistic model is more consistent with the data on the distinction between usual and exceptional concept members. For instance, a usual concept member ‘robin’ is categorized as ‘bird’ faster than it is categorized as ‘animal’, with ‘bird’ being nested in ‘animal’. The weighted membership value provides an explanation by suggesting that more features are matched between ‘robin’ and ‘bird’ than between ‘robin’ and ‘animal’. An exception concept member ‘chicken’ is matched faster to ‘animal’ although it is closer to ‘bird’, which is nested in ‘animal’. The authors propose to include a feature *found-on-farms* within the concept definitions of ‘animal’ and ‘chicken’ (and not in ‘bird’) to enforce a faster match between ‘chicken’ and ‘animal’. Nevertheless, it is not guaranteed to find discriminating features like *found-on-farms* in all such cases.

Criticism

- **Correlated features:** A listing of features may not be sufficient to define a concept. The featural approach doesn’t have any mechanism to represent relations between features like dependencies. For example, the features *sings* and *small* seem to be correlated for ‘bird’ in the sense that the small birds are more likely to sing. Smith and Medin cite more evidence in the domain of artificial concepts because categorizations are more efficient for instances that contain correlated features. The authors suggest the use of conjunctive features to represent correlated features as in *sings-and-small*. Yet this would not only violate the generality constraint on features, but also suggests a presumed decomposition. In addition to correlation, there may be other kinds of relation such as embedding of features. For example the feature *wings* may have in turn the feature *large* to form the feature *large wings*. For cases, where the conjunction may not be expressive enough, the authors suggest to differentiate between types and tokens of features and to introduce feature-to-feature links. Consequently, they suggest a feature network for representing concepts.

- **Lack of a constraining mechanism:** The featural approach, by relaxing the classical view’s constraint of necessity and sufficiency, offers too much of freedom. The authors argue that the approach should not allow any feature to be part of a concept definition. This can be achieved by imposing relaxed constraints such as necessary-but-not-sufficient or sufficient-but-not necessary on features. For instance, *being-animate* for a person seems to be a necessary-but-not-sufficient feature since it appears in most of the instances. Similarly, the feature set *feathered, animate*, *flies* seems to be sufficient to define the concept bird, where flies is clearly a non-necessary feature. The problem with the latter provision is that too many features can meet the sufficiency-but-non-necessity constraint.

1.3.3 The exemplar view

The exemplar view suggests that there is no single representation of an entire class, but only a set of specific representations of the class’s exemplars. The definition of exemplar in this approach is rather ambiguous. An exemplar can be an subset or an instance of a concept. In the first case, the definition allows for some level of abstraction. Most of the models adopting the exemplar view allow summary descriptions in the concept definitions, but use them less intensively than the exemplars during the categorization process. The basic premise of this approach is rooted in the results of experimental studies revealing that people make extensive use of examples when they categorize.

Concept representation and categorization

The representation of a concept consists of separate descriptions of some of its exemplars. An exemplar may be a subset, which in turn may be defined in terms of its own exemplars, or a summary description or both; it may also be an instance of the defined concept. The representation is explicitly disjunctive and therefore, is likely to be a better approach to represent artificial concepts. The approaches based on the exemplar view show less abstraction than representations based on the probabilistic and the classical view. The exemplar view challenges the following assumptions made by the previous views:

- **Summary description is the result of an abstraction process:** A concept definition in the exemplar view collects separate descriptions of its exemplars rather than providing an abstract description that would hold for all the instances.
- **Summary description does not need to correspond to a specific instance:** A concept definition may consist of multiple instances.
- **Summary description is used every time a category membership is determined:** This assumption is not violated by all the exemplar models. Some still use the summary description to determine concept membership; nevertheless, they rely more on exemplars.

Benefits

Exemplar models can deal with disjunctive concepts since their representation is explicitly disjunctive. They can also provide an account for unclear cases, which occur if the number of a concept’s exemplars that match with a particular instance is less than the membership threshold, or if an equal number of exemplars for two concepts match the same instance. For the exemplar view, there is no reason to specify defining features for a concept since it works with exemplars instead of a set of necessary and sufficient features. The simple typicality effect is explained by assessing a similarity between a typical test instance and a best-example since the typicality condition (i.e. a typical instance sharing more features with other concept members) is presupposed in this model. Finally, the exemplar view allows for the use of non-necessary features.

To illustrate the exemplar view’s account of similarity ratings of regular concepts and exceptions with respect to nesting, consider the following model: assuming that we represent ‘robin’ as an exemplar of the concept ‘bird’ and nest ‘bird’ under the concept ‘animal’, a usual concept member ‘robin’ is categorized as ‘bird’ faster than as ‘animal’. In the case of exceptions, assuming that ‘chicken’ is an exemplar of ‘animal’ instead of ‘bird’, ‘chicken’ has faster access to ‘animal’ than to ‘bird’ during categorization.

Weaknesses

- **Representing more knowledge in concepts:** The exemplar models do not provide any mechanism to relate exemplars of a concept since exemplars are represented separately.

- **Lack of constraining mechanisms:** The lack of constraints on exemplar properties results in a large degree of disjunctiveness and causes computational inefficiency in determining class membership. Therefore some of the properties of exemplars in a concept should be specified as necessary or sufficient.

- **Defining a relation between disjunctive exemplars:** A collection of exemplars may point to the same concept but may not meet any theoretical notion of concept; there should be some principled constraints on the relations between exemplars that can be joined in a representation (consider for instance, an artificial concept 'furds' exemplified by chair, table, robin and eagle). Those principled constraints may be represented as a set of necessary and sufficient conditions which would apply to all the exemplars of the concept. This set, in fact, would summarize all the exemplars. Another reason for having a summarized information is the need to deal with generic propositions such as 'all birds lay eggs' etc., without going into each instance and adding a new property.

1.4 Using the framework - A hybrid representation

Smith and Medin's survey identifies the following questions as the point of departure for the classical, probabilistic and exemplar views:

- Is there a single or unitary description for all the members of a concept?

- Are the properties specified in a unitary description true of all members of a class?

The classical view has its limitations in terms of defining a unified description for the perceptual features of its instances; however it offers a reliable inference mechanism to determine class membership since it deals with concept properties that are necessary and sufficient. The probabilistic and especially exemplar views, on the other hand, provide better models to represent artificial concepts by allowing the use of disjunctive features and non-necessary features in the categorization. Nevertheless, the mechanisms employed by these views do not yield an absolute true or false result of a class membership test; they provide a probabilistic inference or a degree of membership. A concept in the classical view is stabilized for its individuals, whereas in the probabilistic and the exemplar views, it is relative with respect to the encountered instances. This comparison hints at the possibility of using the best of both worlds in a hybrid representation. As Smith and Medin pointed out in relation to the criticism directed towards the exemplar view, the inclusion of a summary description can partially eliminate some weaknesses of an exemplar model. Consider the following quote from the authors' account of the mixed representations:

> *We cannot ignore the possibility that the representation of a single concept can contain both probabilistic and exemplar components, that is, both a summary representation and exemplars. Earlier we suggested that such a mixed representation might be needed for superordinate concepts such as furniture. Now we wish to point out that there is a good reason to think that mixed representations may be needed with other kinds of concepts as well.*

Tulving's model of memory, which consists of semantic and episodic components reflects, to a certain extent, the structure of the proposed hybrid model. In connection with the previously considered relation between the intensions and the type lattice, Sowa considers the use of concept primitives (Sowa, J. F., 1984). Yet, he admits that there is no evidence of a truly universal set of primitives that would generate all possible concepts through simple logical operations like conjunction. Moreover, since most of the everyday concepts can hardly be defined through the use of primitives, people make use of family resemblances to determine class membership. Sowa concludes that a realistic theory should not reduce every concept definition to a combination of primitives and could allow for the use of exemplars ('prototypes' in Sowa's terms) in determining class membership.

II . 2 Type and typology in architecture

The architectural discourse on type is one of the richest in design theory. It very often derives its effectiveness and power from a confused agreement or a cultural consensus on a vague definition of type (Bandini, M., 1989). Nevertheless, It is particularly important for this study to reduce the ambiguities inherent in the typological discussion since the study of type is beneficial only to the extent that it is congruent to a mathematically well-founded and applicable framework of representation.

2.1 Analytical vs. generative typologies

Vidler, in his study of the transformation of type in 18th and 19th centuries (1976), identified two traditions in which the notion of type influenced the production of architecture. The first was the justification of architectural designs through the rooting of architecture in types as first principles, e.g. principles derived from nature (Laugier's primitive hut) or industrial production (a typology of mass production objects). The second tradition associated the notion of type with other theories of classification (e.g. theories dealing with the classification of *natural kinds* in the 19th century) in order to develop a taxonomy of architectural artifacts and to suggest a basis for the creation of new types. Vidler's observation about the first group also applies to the Neo-rationalists, who propose an ontology of the city in order to justify their approach to architectural design. Neo-rationalists argue that the architectural product reveals its past and present through a type-form that resides in its physical structure. By incorporating the forms of the traditional city, their typology provided means to maintain the continuity of forms and history (Vidler, A., 1977).

The same distinction is interpreted by Leupen (Leupen, et.al. 1997) as one between analytical and generative typologies. The analytical typology is confined to naming various architectural elements and describing how these elements fit together in a composition. The generative typology, on the other hand, provides the designer with solutions, where type is *the bearer of design experiences pertaining to a similar issue*. Researchers making use of analytical typology are concerned with different classifications, whereas designers are concerned with the principles of classification. It is however, difficult to imagine a theory of classification which does not become involved in the principles of classification. Moreover, the process in which the designer discovers design experiences pertaining to a similar problem may in itself be of interest to researchers, even more so if we extend Vidler's second influence of type to include theories of classification dealing with *artificial kinds.* In Herbert Simon's terms (1969), artificial kinds are distinguished from the *natural kinds* in the following aspects:

- Artificial things are synthesized (though not always with full forethought) by man.
- Artificial things may imitate appearances in nature while lacking, in one or many respects, the reality of the latter.
- Artificial things can be characterized in terms of functions, goals, adaptation.
- Artificial things are often discussed, particularly when they are being designed, in terms of imperatives as well as descriptives.

Simon's conception of artificial kinds provides a basis for studying the classificatory use of type. For the purposes of the present study, the focus is on issues related to representation of concepts and categories for the classification of architectural precedents. The notion of type is considered independent of any typology. Typologies imply a particular view of the design process and are often associated with specific design methodologies[1]. The review, therefore, excludes the study on particular typologies from its scope, unless they adopt a fairly generic conception of type. There are two major areas of interest within our scope: conception of type within a linguistic analogy and the questions concerning the a priori vs. a posteriori nature of type with respect to representational issues.

1. Bandini (1989) associates the acceptance of a formal framework or any attempt to systematize knowledge in architecture with what is operational rather than theoretical and her distinction is often associated with the one between type and typology. Here, however, the distinction between type and typology is not between an operational level discussion dealing with *typology* and a theoretical discussion dealing with *type*.

2.2 Linguistic analogy

The motivation behind suggesting an analogy between architecture and language is to provide means to read and understand architecture. The linguistic approaches often rely on the following argument:

- Architecture seems to display some kind of syntax: there is a possibility to describe rules governing the combination of parts to form an architectural object.

- An object of architecture is similar to a sentence in its syntactic structure. Hence, the object has a meaning to be deciphered, and this meaning is composed of the meanings of its parts.

It is, however, difficult to infer the existence of a grammar, in the linguistic sense, from a syntactic structure alone. To claim that architecture has a grammar is to suggest that the meanings of parts of an architectural object determine the meaning of the whole. When the analogy is taken to this extent several questions have to be addressed: What constitutes the meaning for an architectural object and how does this meaning differ from the linguistic meaning? Is there a parallel between semantic and syntactic unity of a sentence structure and that of an architectural object?

To examine the validity of the suggested analogy, a comprehensive comparison of architectural and linguistic meanings is required. This, however, can easily turn into a circular argument, since the reason the suggested analogy is introduced in the first place, is to achieve a better understanding of the architectural meaning. I will, therefore look into the linguistic counterpart and to Frege's widely accepted account on linguistic meaning in particular (1892). Frege suggests that the meaning in language can be thought out in three levels: words, expressions and complete sentences. A word, a sign, a sign-compound, an expression designates or signifies its nominatum and expresses its sense. Hence it is possible to designate an object with a sign (word, expression) as well as its sense (connotation, meaning) in which the context is contained. A complete declarative sentence, on the other hand, has a proposition which Frege argues should be regarded as the sense of the sentence, and a truth value, which is its nominatum. As Frege rightly points out, the question of truth is irrelevant to the discussions on meaning in the context of art. Whether an object of architecture has a nominatum is insignificant as long as its conceived as a work of art.

> *In regard to the words we must note that, owing to the uncertain correlation of images with words, a difference may exist for one person that another does not discover... Among the differences possible in this connection we mention shadings and colorings which poetry seeks to impart to the senses. These shadings and colorings are not objective. Every listener or reader has to add them in accordance with the hints of the poet or speaker. Surely art would be impossible without some kinship among human imageries; but just how far the intentions of the poet are realized can never be exactly ascertained. We shall henceforth no longer refer to images and pictorizations; they*

were discussed only lest the image evoked by a word be confused with its sense and nominatum... In listening to an epic, for example, we are fascinated by the euphony of the language, and also by the sense of the sentences and by the images and emotions evoked.

Similarly, in the context of architecture, the sign (or the physical object) itself is essential to the understanding of architecture through the imagery and emotions it evokes, which is clearly distinguished from the sense and nominatum. Moreover, the definition of connotation (sense, meaning) and denotation (nominatum) for an architectural object is highly controversial. For instance, Eco suggests that the understanding of architecture is the reading of various meanings from an architectural sign through the ideas it connotes and the functions it denotes according to its use[1] (e.g. the gothic style connotes the idea of religiosity and may denote various functions). Colquhoun suggests that type has a communicative value based on the analogy between structural linguistics and art (1969). He proposes that '*intelligible forms of the past*' or '*typologically fixed entities*' (architectural sign), convey (connote) artistic meaning within a social context. The reduction of the communicative value to an iconic one and the type to an architectural image (e.g. Venturi and his followers) is caused by the forced analogy between structural linguistics and architecture. Yet, as Scruton (1979) points out, the use of the terms denotation and connotation does not necessarily allow a theory of linguistic meaning to apply to a non-linguistic context. Frege's account on denotation and connotation in natural language does not justify (and does not even apply) to various theories concerning meaning in architecture.

In relation to the second question, the truth-value of a complete sentence is decided based on the truth conditions derived from what its parts, i.e. words, signs, expressions, refer to. Therefore the syntax derives from this relation to truth. The existence of a stand-alone syntactic structure (independent of semantics) is not plausible in natural language. The vagueness of what an architectural sign denotes causes the syntactic structure of an architectural composition not to have the same strong semantic correspondence that a sentence structure has. Hence the analogy cannot support the argument that architectural meaning can be deduced from the meanings of parts of an architectural object.

The linguistic analogy therefore fails when it is pushed to the extent where the existence of semantics is denied. This is particularly true when it provides a basis for typological approaches where type is placed within this analogy for its role in justification. The use of type in classification, on the other hand, can provide a different account of denotation and connotation through the use of Quine's extension and intension: intension can be characterized by the concepts and ideas that make up a definition and extension by a set of objects to which the definition applies. For classification, type is placed within the domain of intension, hence architectural type becomes simply an artificial kind. As Moneo points out, the act of identification of an architectural element or of its parts is essential to represent and describe a particular artifact. This process of recognition (i.e. naming)

1. For an extended criticism of Eco's argument, on the relevance of associating the functions-use with the nominatum refer to (Scruton, R., 1979).

implies typification: establishment of common characteristics with a similar class of things (1982). Therefore the use of type is implicit in the natural language. Moneo's account of type is somehow closer to the one in cognitive psychology, where type as a fundamental conceptual structure is used in the categorization process, which is an essential aspect of cognition (Jackendoff, R. 1994). Here, the account of type does not imply any linguistic analogy as it does, for instance, in Colquhoun's case (Colquhoun, A., 1969). It simply conceives type as a manifestation of a generic capability of the human mind: generalization, similar to Tezar who recognizes this capability as a biological necessity (1991). Tezar argues that the current debate on architectural type favors a notion of type which is used to study the meaning assumed to be embodied in architectural objects and often neglects the more generic 'human side':

> *...the predominant focus on the architectural artifact has left a theoretical vacuum and considerable confusion on the other, human side of types: Why and on what basis does human mind classify experiences? Is any classification synonymous with a type?... Type simply seems to be a "natural" context of architectural experience, almost as natural as the actual setting of a building. Our perception of the world is phenomenally given to us in an already categorized manner and our memory is "typologically prefigured." As architects we are free to choose a theoretical position that opposes the notion of type, we may choose to design buildings that ignore the notion of type, but we have no control over "the other side of types."*

2.3 A priori vs. a posteriori

In the beginning of Section II . 2, I set the scope of the literature review to the classificatory use of type and in Section 2.2, I gave a linguistic account of architectural types. Another important aspect of typological discussion is the formation of types since it involves some of the representational issues that have been addressed in connection with classification. There are two major approaches in looking at the formation of types. The first suggests that architectural types are the extension of pre-existing categories (i.e. type is a priori), and the second suggests that they are defined by the comparison and grouping of the existing architectural artifacts (i.e. type is a posteriori).

One of the most frequently cited theoreticians, Quatremere de Quincy, is in the first group with his definition of type as an *ideal type, an elementary principle, a sort of nucleus about which are gathered, and to which are coordinated, in time, the developments and forms to which the object is susceptible (*1825*)*. The ideal type is neither visible, nor realizable and therefore cannot be copied, unlike the model that can be endlessly replicated. Vidler[1] points out that Quatremere de Quincy's definition is too abstract and is unlikely to be a working principle in design. Quatremere de Quincy, realizing the difficulties implied by his definition of ideal type, proposes another type concept: relative type, as in types of building, or designed objects. Relative type is based on the need, use and custom. When

1. More on Vidler's account on Quatremere de Quincy can be found in (Vidler, A. 1978).

Quatremere de Quincy writes '*Who does not believe that the shape of a man's back must provide the type of the back of a chair?*' he clearly refers to the latter and not to the *ideal type*. Other than the difficulty of laying out a priori categories which would constitute the elementary principles governing the design of objects, there is also the problem of explaining the creation of new types. If we assume that we could identify these elementary principles, ideas, categories that coordinate the developments and forms of existing architectural objects, then we should be able to, in a sense, predict the future of forms and developments. However, there are many outside factors that are involved in the creation of new types. For instance, according to Sullivan, the conditions and needs lead the architect to seek *a true normal type* as a solution to the design of tall office building (1947). From these conditions, the architect identifies a new design problem, which requires the use of a new type. Consequently, the tall office building took its place among the other architectural types as opposed to being derived from them.

In the second group we can cite Argan who avoids the problems of Quatremere De Quincy's definition by defining the type as being deduced from reality through an *a posteriori* operation which involves a comparison and overlapping of formal regularities (1963). Type therefore is a schema of form through which series of buildings are related to each other and not an to an *a priori* form. While Argan agrees with Quatremere de Quincy's on the vagueness or generality of type he opposes the idea of the a priori formulation of type:

> *It [type] is never formulated a priori, but always deduced from a series of instances... The birth of type is therefore dependent on the existence of a series of buildings having between them an obvious formal and functional analogy. In other words when a type is determined in the practice or theory of architecture, it already has an existence as an answer to a complex of ideological, religious or practical demands which arise in a given historical condition of whatever culture.*

The latter approach is more likely to provide a satisfactory explanation for the emergence of types and has much more to offer in terms of workability. However, it is important to realize that Argan's understanding of type and his argument of typology imply more than classification and get into the realm of justification by attributing the use of type to the creative process of design. Nevertheless, his emphasis on the a posteriori nature of type and his attempt to clarify the concept of the building series are fairly important for the purposes of this study. Before elaborating Argan's views on building series it is important to refer to his account on the distinction between model and type. Much as the type is identified as an outline object by Argan, the model is a specific example or mechanical reproduction of an object. In this sense type is similar to a type-concept, and a model can be conceived as a prototypical instance of the type-concept. Argan refers to a series of formal variants, more precisely, to a group of buildings exhibiting formal and functional analogy, by using both terms *building series* and *typological series*.

> *The type therefore, is formed through a process of reducing a complex of formal variants to a common root form. [...] It is not, in fact necessary to demonstrate that if the final form of a building is a variant deduced from a*

proceeding formal series, the addition of another variant to the series will necessarily determine a more or less considerable change of the whole type.

In this definition a formal variant is an instance that exhibits the characteristics of the type governing the series simply because the type itself is deduced from these instances, and hence the definition of type itself is dependent of the instances. In the hybrid representation scheme, these instances or variants are exemplars, and the dependency is maintained through the type-concept's encapsulation of both the necessary and sufficient conditions which apply to all the exemplars.

2.4 Multiplicity of groupings

If we consider type as a conceptual structure and assume that classification is an essential mechanism of human cognition, there is a use for type as a classificatory device in the understanding and production of architectural artifacts. We also suggested that types are a posteriori by nature. Types are defined through the comparative analysis of existing architectural objects. This involves the extraction of common characteristics to form types and the grouping of architectural objects based on these characteristics. For the natural kinds, the common characteristics can be organized by the use of a relatively small number of categories such as formal and organizational ones. On the other hand, type categories are more numerous for artificial kinds based on the fact that artificial kinds differ from the natural counterparts by being characterized in terms of functions and goals. Hence the classification of artifacts involves types characterizing the function, goal, behavior and structure of the artifact.

This point is supported by the architectural discussion on type, where a fairly large number of typologies are proposed to account for functional, institutional, formal, compositional, structural, historical aspects of architectural artifacts. Some of these classifications can be merged, whereas some remain orthogonal. In the literature, the multiplicity of classifications is often implied by an opposition between typologies favoring either one of the formal/geometric or use-related/functional groupings/classifications (e.g. Durand ‘s formal classification[1], Purves' organizational patterns (1982) vs. Pevsner's functional building classification (1976)). Aymonino[2] in his neo-rationalist attempt to describe an ontology of the city, identified two levels in which type finds its definition: formal and functional levels. The former suggests an independent typology which is used to classify architectural objects based on formal differences as in Rossi's analysis of city (1982). The latter is favored by Aymonino, who seeks to trace the persistence of certain types with respect to their use in the city. Aymonino's functional type has much in common with Tezar's use-related type; however by bringing the discussion into the city scale, Aymonino proposes more of an institutional classification. Tezar suggests that architectural types are primarily defined on the basis of use since these types

1. More on Durand's approach to classification appears in (Vidler, A., 1976).
2. An extended account on Aymonino's conception of type can be found in (Bandini, M., 1989).

constitute a shared framework of reference unlike some other classifications such as the ones based on a compositional principle or constructional system:

> *It is interesting to note that the common names of most buildings refer to them as functional types: house, school, grocery store, library, church, court-house. This seems to indicate that the use of buildings has primacy as a collective distinction and the buildings are socially predominantly remembered, anticipated, recognized and thought about on that level. In other words, architectural types, on the building level, are functional building types.*

Argan, on the other hand, argues that the fundamental type for architecture is formal and is not deduced from physical functions of the artifact. He limits the number of categories for formal classification to three and associates each with a particular stage of the design process. The three main categories are: the complete building configuration guiding the planning phase; major structural configuration for the design of the structural system and, finally, a typology of decorative elements guiding the design of the ornamental elements. It is important to recognize the change of the kind of classification with respect to the tasks encountered in various stages of design process; however, Argan would still need to justify:

- a design process model suggesting a particular decomposition in terms of design stages where each stage involves a set of design tasks.

- an association between a specified design task in a specified stage and a particular classification.

Argan's argument relies on strong assumptions about the design process in favor of his proposed typology and disregards the fact that the conceptualization and classification of an architectural artifact is closely related to the intention behind its production, the medium by which its produced in addition to its formal characteristics. This is why Moneo[1]'s understanding of type seems to be more comprehensive for the purpose of this study.

> *What then is type? It can most simply be defined as a concept which describes a group of objects characterized by the same **formal structure**. It's neither a spatial diagram nor the average of a serial list. It is fundamentally based on the possibility of grouping objects by certain inherent structural similarities. It might even be said that type means the act of thinking in groups [...] But what is precisely a formal structure? One could attempt a series of opposing definitions. First the aspects of Gestalt could be emphasized. This would mean speaking about centrality or linearity, clusters or grids, trying to characterize form in terms of a deeper geometry. [...] This however reduces the idea of type as inner structure to simple abstract geometry. But type as a formal structure is, in contrast, intimately connected with reality - with a vast hierarchy of concerns running from social activity to building construction. Ultimately, the group defining a type must be rooted in this reality as well as in an abstract geometry.*

1. More on Moneo's account on type can be found in (Moneo, R., 1982).

Moneo recognizes that the use of type in architecture cannot be reduced to a mere formal classification, nor to a functional one. It is necessary to provide a framework for type definition that would support multiplicity in the groupings or classifications of architectural artifacts. For instance, in the initial stages, a designer may be interested in using a courtyard layout for climatic reasons or for some other consideration concerning the formal characteristics of courtyard layouts. At a later stage, in specifying the room layouts he may use a functional classification. In short, the same artifact may be grouped over and over under different types depending on the design stage and the particular goals identified for the design problem at hand. The use of type provides a mechanism to group artifacts based on the similarity in terms of a set of characteristics. These characteristics form an open set which is subject to change in relation to the context of a particular design task. The context may be conceived in terms of the level of the design task as well as the goals to be achieved by solving the particular design problem.

Based on the above, the need to combine various concepts to form classifications is inevitable. An architectural object can be multiply classified by a mechanism referred to earlier as conceptual combination. For the suggested hybrid representation scheme, a conceptual combination is performed through multiple inheritance, where a classification concept inherits the summary descriptions of more than one classification concept.

II . 3 Indexing and retrieval in CBD

Falling under the more general category of reasoning by analogy, CBR suggests a computational model for the use of analogy in problem solving. CBR approaches are different from other approaches in AI in that they make use of specific knowledge of previously encountered problem situations instead of relying on generic knowledge of a problem domain. The specific knowledge is structured in the form of *cases* as part of a case-base. CBR systems recall these cases to use in new problem situations. Another difference of CBR approaches is that a system build on the premises of CBR evolves in time since it learns from each problem that has been encountered and solved[1]. CBR uses a case-base instead of a database as a means of storing data as past problem solving episodes and retrieving these episodes based on similarity and not solely on a direct syntactic value matching. CBR systems typically have to deal with the representation of the case content, the organization of the case memory, strategies for recalling cases, and mechanisms to modify cases to fit new problem situations. The following subsections elaborate on these issues of indexing, memory organization and retrieval in the context of case-based design (CBD).

1. For an introduction to the case-based reasoning paradigm refer to (Aadmodt, A. and Plaza, E., 1993).

3.1 CBR in design

CBD is the application of CBR technology to solve problems in the domain of design. In architecture, CBD is often considered in connection with precedent-based design (PBD), which has been introduced to the computational design literature by Oxman[1] as *the process of selecting relevant ideas from prior designs in current design situations.* CBD differs from PBD in its support for systematic storage and adaptation of cases. In CBD systems, the use of a case-base is not limited to a browsing activity; it involves the recalling of past designs, ideally in a form that immediately enables their adaptation to meet the requirements of a new problem.

Prototype-based design is an alternative to CBD where design prototypes encapsulate more generalized knowledge about design solutions[2]. The prototype-based approach is preferred in situations where design generation and refinement cannot directly benefit from specific design instances. The case-based approach, on the other hand, utilizes specific knowledge encapsulated in detailed instances that are retrieved when a sufficiently close problem situation arises. The difference, however, is rather vague and depends mainly on the level of abstraction of the representation envisioned for a design case and prototype. Similar issues arise for the indexing and retrieval of both cases and prototype.

Design in a CBR model is a description or a set of descriptions generated to satisfy requirements specified as part of a design problem. The processes involved in generating designs have to deal with relations between topological, physical and geometric properties. The aspects of design process affecting the use of CBR technology in problem solving are identified by Maher (1995) as follows:

- Real world design problems are large and complex.
- The design case representation is composed of various modes of representation such as text, graphics, equations, and drawings.
- In design, there is no predefined mapping between a set of requirements and a design solution, and in some cases an initial specification cannot be predefined either.
- Different types of knowledge may have to be integrated in the design process.
- Often the design solution is found by merging various parts of various old solutions.

1. More on precedent-based design in architecture can be found in (Oxman, R., 1994).
2. A detailed account on the prototype-based design appears in (Rosenman, M. A. et.al. 1992).

- CBR should take into account other computer-based representations and processes since the design practice already makes use of computer programs.

By implication, design cases often have complex representations that integrate various kinds of information (e.g. geometry, function, compositional aspects etc.) expressed in different levels of abstraction. The representational complexity constitutes a burden on the searching and matching mechanisms in terms of efficiency and effectiveness. Consequently, the organization of the case memory and indexing of cases for effective retrieval and reuse are vital for CBD systems. The first four items, because of their impact on the indexing and retrieval of design cases, are among the determinants of the classification scheme proposed in this study.

3.2 Memory organization and Indexing

In recalling a previous design for the generation of a new design solution, the appropriateness of the selections is one of the most important criteria in determining how useful a CBD approach can be. The computational support for finding the relevant cases in a CBD system hinges on an efficient indexing mechanism integrated with a case memory. For some case-base actions such as making a diagnosis, assessing a situation etc., the indexing may be insignificant for retrieval since case selection is supposed to rely primarily on surface and contextual features (Waltz, D., 1991). Nevertheless, in design, the case selection cannot be limited to an attribute-by-attribute matching of surface and contextual features. The retrieval is likely to involve lengthy comparisons of compositional and geometric properties of the cases. An efficient indexing makes the retrieval a tractable computational problem and speeds up the process by partitioning the memory so that the matching is performed only on a subset of the case-base. Moreover, in complex problem solving activities such as design, the retrieval may require the use of thematic features (e.g. goal, function, behavior etc.) which may not be inferred from the case structure. These thematic features, referred to as deeper features in the CBR literature, are obtained through an elaboration and interpretation of generalized models of the design domain. Indexing supports the organization of cases based on these deep features[1] without overloading the case content.

The process of indexing is often described as assigning labels to cases to ensure their retrieval in relevant situations. An indexing vocabulary is defined to be a subset of the vocabulary used for symbolic representation of cases (Kolodner, J., 1993). Kolodner lists the following aspects of a good indexing vocabulary:

- **Prediction:** Indices should be case aspects that tend to predict solutions and outcomes of cases.

1. Similar to the previously discussed type concepts, case indices may have feature-based or dimensional representation. In CBD literature, a case index is generally described as a collection of labels suggesting a feature-based representation.

- **Specificity:** Indices should be specific enough to allow for all the useful discriminations in the case memory.
- **Generality:** Indices should be general enough to capture relevant similarities among the cases.
- **Usefulness:** The use of indices during retrieval should produce useful results.

The solutions and outcomes of design problems are not always predictable. The aspects that are critical in generating a design solution may change with the encountered problem situation or with a designer's perspective. Nevertheless a small subset of features may be assumed to be more critical than others, and used as a starting point for retrieval. The specificity and generality aspects suggest a hierarchical structuring for indices where a CBD system can make use of different levels of specificity during retrieval. The 'useful' aspects, like the critical ones, are hard to predetermine and can only be apparent to the designer using the system. Hence it may be beneficial to adopt an indexing scheme that would allow the user to extend or modify the indexing vocabulary.

Another reason for adopting an extendable indexing vocabulary is related to the use of deep features in retrieval. When a subset of case features is used to define an indexing vocabulary, it becomes the basis for determining the relevance of cases to the current problem situation. However, as stated earlier, the selection of relevant design cases may require the use of deep features during retrieval. Very often, these deep features do not belong to the previously defined set of case features. CBD systems can allow for the use of deep features during retrieval by providing the functionality to add these features into the indexing vocabulary as a means to organize and re-organize the case memory.

An indexing scheme is a structured indexing vocabulary to organize cases. One of the major concerns in designing indexing schemes is flexibility. A flexible scheme provides:

- an extendable indexing vocabulary which is not limited to surface features,
- means to extend and modify the model of case memory and indexing scheme, along with mechanisms to ensure soundness and,
- multiple paths to cases.

Maher groups the computational indexing schemes as descriptive schemes and relational schemes (1995). Descriptive schemes often employ a fixed set of surface features to serve as pointers to various cases. Computationally, these features use a list or tree structure. In a list, each element indicates the set of cases labeled by it. In a tree hierarchy each node points at cases carrying the feature defining the node. Hence cases are represented in a generalization hierarchy. Descriptive schemes are fairly easy to implement. However they often fail to satisfy the flexibility requirement for the reasons stated earlier in connection with the use of deep features in retrieval. Design specifications are subject to change within the design problem solving process, and a fixed set of surface features may not provide room for these changes.

Relational schemes, on the other hand, capture deeper aspects of cases, i.e. abstract relationships which can be represented through features, objects or graphs. The use of objects-based or graph-based representations allows for the addition and instantiation of relationships within a dynamic model of case memory. Therefore the indexing scheme can be interactively extended provided that there are mechanisms to perform consistency checking. Relational schemes are considered to be more efficient since the retrieval doesn't require an exhaustive search of all cases. With the help of the scheme only the cases in the category of interest are accessed. Referring back to the definition of a flexible indexing scheme, relational schemes are potentially more flexible than descriptional ones.

3.3 Retrieval strategies

Retrieval is the act of selecting the most similar cases to a given problem description. In retrieval, the use of indices narrows down the search scope and provides a shortcut to relevant cases. Once the case-memory is partitioned based on relevance to the current problem situation, matching is performed on the candidate cases. Unlike some CBR systems that employ parallel search techniques, CBD systems are likely to make use of indices by build-in heuristics or user interaction during case-base partitioning.

A search strategy can be regarded as a collection of methods employed during the search of a case memory. Maher groups various retrieval strategies under the following categories:

- **List checking:** This strategy uses a feature list, where each item points to a number of relevant cases to be retrieved. A case can be retrieved as a whole or in parts using the indexed features.
- **Concept refinement:** This strategy uses a hierarchical tree where each node points to a number of relevant cases to be retrieved. Refinement begins at a more general concept and proceeds downward until a match is reached. If no match is found at a specific level, the problem description can be redefined to match a higher level in the hierarchy. Both whole and parts of cases can be stored organized in the index tree.
- **Associative recall:** This strategy is used in relational indexing schemes. Case indices are graphs incorporating concepts, relationships (deep and surface features) defining a generalized model of a particular domain. The problem description is converted to a graph, and graph-matching is performed to retrieve relevant cases.

Maher's grouping of case retrieval strategies should be considered in connection with the previously referred indexing schemes and case representation. For instance, a descriptive indexing scheme can hardly provide a base structure for the concept refinement and associative recall strategies. In addition to the strategies listed above, there are various techniques used to lead a case-base reasoner to previously inaccessible cases during retrieval. Index elaboration and revision are two techniques that Maher considers in connection with associative recall. Below are brief descriptions of these along with some other techniques mentioned in the CBR literature:

- **Index elaboration:** Index elaboration is an incremental process launched after the retrieval of a initial set of cases. Based on an analysis of these cases, critical or discriminative features which were not part of the initial problem specifications are identified. Index elaboration occurs by either adding more discriminative features to the problem specification (index expansion) or identifying the critical features (index reduction). Using the elaborated specifications, the case-base reasoner retrieves cases that are more relevant. The modification of problem specifications requires extensive use of domain knowledge, which should be made explicit in the memory organization. User interaction maybe a more flexible and viable alternative to the use of domain knowledge for the identification of discriminative or critical features.

- **Index revision:** By the influence of an initial set of retrieve cases, a problem specification is revised to access more appropriate cases. Index revision is characterized as a change in the index description. Unlike index elaboration, the change is not necessarily one of reduction or expansion. It provides a means to case-memory exploration by adding or dropping indices to create a new problem specification.

- **Relaxation:** When the problem description introduces too many constraints and the retriever fails to select a relevant case, some operators may be used to relax the boundary imposed on the solution space. Relaxation operators work in two ways: they either eliminate some of the constraints or weaken them in order to access a wider range of candidates. For instance, elimination of some index feature or widening the range indicated by an index dimension are means of relaxing the boundary around the solution space.

- **Index transformation and mutation:** This technique uses heuristics to activate domain specific mutation operators on indices during retrieval. Navichandra argues that mutation is particularly useful as an innovative design strategy, provided that the case-based reasoner is constrained to produce semantically correct alternatives (1990). He cites the CYCLOPS program as a case-base reasoner which uses a restricted representation scheme (CLP) to circumvent the semantic correctness issue. The program performs mutations on the constraints defining the solution space. For example, *house-on-stilts, stilts-on-house* are among the mutated alternatives for a constraint *house-on-site* in CYCLOPS.

The retrieval of design cases starts with a partitioning of the case-base in order to access potential matches for a given problem specification. The partitioning is done through a target index definition by adopting one or more retrieval strategies or techniques described above. The process of defining the target index can be automated in a design process model or can be performed through user interaction. Upon the retrieval of a set of potential matches (or their parts), the case-base reasoner selects and ranks the best cases. This additional assessment of the retrieved cases involves a comparison based on usefulness and similarity to the given problem specification.

Maher groups the existing CBD approaches to matching and ranking based on their choice of similarity metrics and ranking scheme. The first group defines the best match in terms of the maximum number of matched properties (features or attribute-value pairs). The second group uses the weighted sum of matched properties to rank the cases. In both of these approaches, the current problem specification provides the similarity metrics for ranking. The third group uses the context as the similarity metrics to select the case with most potential to satisfy the goal in the current problem situation. CBD systems may benefit from a combination of these approaches in defining their similarity metrics. For instance, SEED adopts a similarity metric which incorporates the context and the current problem specification within the target index. SEED ranks the cases based on the weighted sum of attribute-value pairs, and performs matching on both context and problem specification (Flemming, U., et.al. 1996).

Conceptual Model

This chapter introduces a hybrid memory scheme based on the salient issues discussed in the literature survey. The scheme constitutes the basis of the computable model which underlies SEED's case indexing and retrieval engines. The section also provides a requirement analysis for the suggested implementation environment.

III . 1 Memory organization

This section describes the implications of the literature survey on the design of the computable model and discusses the major design decisions and compromises.

1.1 Distinct schemes

The conceptual model behind this work traces the distinction between episodic and semantic components for a memory model (Figure 1) in:

- Cognitive Psychology,
- Knowledge Representation,
- AI, and
- Architectural Typology.

Tulving introduced the distinction to the Cognitive Psychology literature and focused on the distinct information processing mechanisms for episodic and semantic memories (Tulving, E., 1972). In Knowledge Representation, the separation is considered in the context of the information processed by these mechanisms. Episodic knowledge is expressed in terms of exemplars (or *prototypes)* and the semantic knowledge in terms of generic descriptions that summarize these exemplars. Various forms of analogy-based reasoning paradigms (e.g. CBR) borrow from the Prototype Theory[1] and focus on the use of episodic knowledge in problem solving. Computational design systems modeled after these paradigms (e.g. Case-Base Design systems) store exemplars/prototypes as solutions and recall them in similar problem contexts.

On a similar track, the literature on Architectural typology identifies the notions of *type* and *building series,* which reflect the separation between exemplars and summary descriptions. More importantly, the literature describes the formation of type as an a-posteriori process[2]. As an implication, any attempt to represent architectural type computationaly should take in to account that types cannot be modeled in a deterministic fashion. Their definitions are subject to change as long as there is a possibility of introducing new buildings or defining new ways of grouping. Another important finding of the typological discussion is that buildings can belong to multiple groupings accounting for different typologies (functional, compositional etc.).

The literature survey identifies the following issues as the major determinants of the conceptual model behind the hybrid approach described in Section III . 3:

- the separation of information captured in precedents and classifications in terms of representation and processing
- classifications incorporating multiple groupings
- an evolving classification vocabulary.

In the hybrid approach, a classification may be a primitive concept or a complex concept composed of a conjunction of other concepts. Each classification has a set of necessary and sufficient concepts which apply to all of the precedents that it describes. The scheme should ensure consistency within the set of necessary and sufficient concepts defining a classification. Using Smith and Medin’s terms, the hybrid approach incorporates the classical, the exemplar and the probabilistic view in one model. The precedents are treated as exemplars or specific instances of design solutions. These instances are grouped based on classifications capturing orthogonal and multiple classifications, similar in content to a summary description. The classifications, however, contain more information with respect to the relationships between the features defining the concept. The assessment of similarity for individual exemplars requires a probabilistic inference to determine the closeness in fit.

1. Refer to Section 1.2 for more on PT.
2. Note that there is no consensus on the a posteriori nature of type in the typological discussion. The reference here is to those who argue that the types are not extended from pre-existing categories.

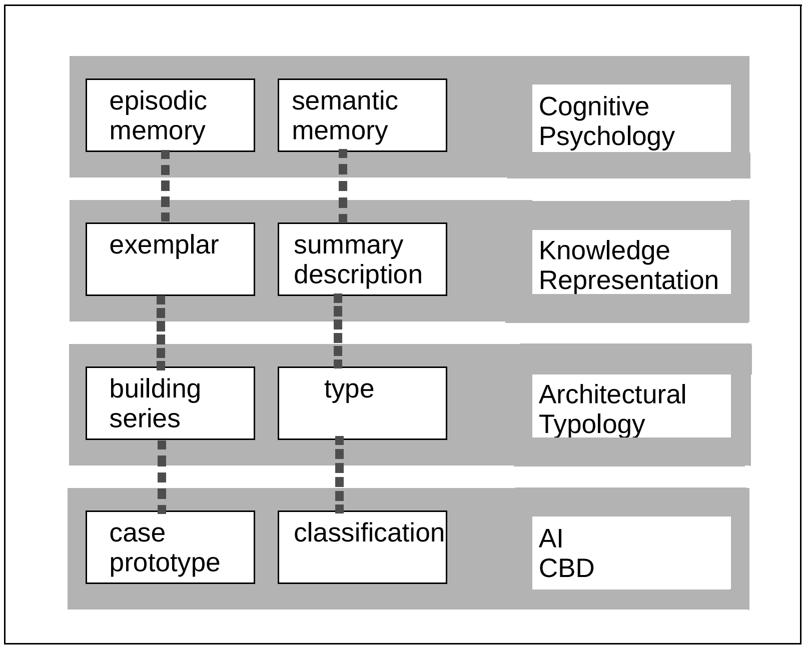


FIGURE 1. Tracing the separation between semantic and episodic memory

1.2 Trade-offs

In the suggested computable model, distinct representation schemes for precedents and classification are used in order to reflect the separation between semantic and episodic components of the memory model. Accordingly, distinct inference mechanisms are used for the retrieval of cases and comparison of classifications. This way, the scheme used to represent the precedents does not have to be modified every time new thematic information is introduced to the system. This information, on the other hand, can be represented in terms of classifications. The classification vocabulary can be augmented with new concepts, or the existing concepts can be dropped when they are no longer relevant to the design context. The grouping of precedents is a meta-level operation which does not necessarily require any change in the representation of these precedents. Having a separate engine for classifications and groupings brings high flexibility to the retrieval mechanism and more expressiveness to the representation scheme.

On the other hand, the hybrid approach has potential weaknesses in comparison to a unified system of representation and retrieval:

- **Redundancies:** Since there are two distinct schemes for representing design information, extra modeling effort is required to reduce redundancies. It is important to decide on the nature of the information before assigning it to the classification or precedent domain. The consistency between the two domains will become an issue, if information is replicated in both domains.

For instance, if the precedent representation comprises a constituent hierarchy, the classification does not have to introduce concepts which will be used to group precedents based on a *part-of/consists-of* relationship.

- **Ambiguities:** Efforts to reduce the redundancies may encounter concepts that can be equally represented as part of the classification or precedent scheme. Similarly, in some situations, classifications may have to combine the concepts that belong to the precedent scheme with classification concepts in their description. The modeling of the design information will have to consider such ambiguities.

- **Expensive maintenance:** Reducing redundancy and preserving consistency between the classification and precedent engines require additional mechanisms for data maintenance.

Consequently, the hybrid approach adds some level of complexity to the modeling process and causes the maintenance of the system to be relatively expensive. The design decisions for the suggested computable model have been finalized based on the issues identified in the literature survey as well as by the requirements for building a CBD system as part of the SEED project. The following sections identify these requirements.

III . 2 Requirements

The generic requirements are based on some of the prominent issues I addressed in the survey of architectural typology and case-base design. I also identify implementation-specific requirements within the context of the SEED system.

2.1 Generic requirements

Conforming with the discussion on classification vocabularies and architectural types, the generic requirements for indexing and retrieving precedents within a case-base design system can be stated as below:

- **Flexibility, extensibility: designing in an "open world"** (Hinrich, 1992)**:** When a CBD system performs tasks in an open world, it is likely to deal with incomplete knowledge in the form of incomplete knowledge of categories (1), incomplete domain theories(2) or under-specified problems (3). The indexing is affected by the first and third type of incompleteness. The design domain includes open categories or unbounded sets which are widely used in classifying design precedents. Their classifications in a case memory do not form a closed set. New classification concepts may be added to the system, and existing classification instances may be modified. These two form of incompleteness necessitate the use of a flexible/extendible scheme for case indexing and retrieval.

- **Use of deep features:** In complex problem solving activities such as design, the retrieval may require the use of thematic features (e.g. goal, function, behavior etc.) which may not be inferred from a case structure. These thematic features, or deep features in the CBR literature, are obtained through an elaboration and interpretation of generalized models of the design domain. Indexing should support the organization of cases based on these deep features without overloading the case content.

- **Allowing for multiple groupings of cases, multiple paths to cases:** The classification of a design precedent may incorporate orthogonal taxonomies representing functional, spatial, organizational concept hierarchies as in the description: *private-office-for-chief-executive*.

- **Computational efficiency:** In design, case selection cannot be limited to an attribute-by-attribute matching of surface and contextual features. The retrieval is likely to involve lengthy comparisons of compositional and geometric properties. An efficient indexing makes the retrieval a tractable computational problem and speeds up the process.

2.2 SEED specific requirements

The implementation context for the hybrid model is the indexing and retrieval capabilities for SEED’s CBD engine. SEED's architecture is based on a division of the preliminary design process into phases. SEED intends to support each phase by an individual support module based on a shared logic and architecture. Each module in SEED addresses a specific task within the overall preliminary design process. A module may use its own internal representation of design problems and solutions. This allows for the local use of various pieces of existing and possibly heterogeneous software, and the development efforts can be distributed among several teams and over time. On the other hand, each module should appear to the user as part of a unified whole. To facilitate this, the information exchange between SEED modules is centered on a handful of shared concepts such as (Woodbury, et.al. 1994):

- **Specification Unit (SU):** A SU is responsible for completely specifying all information needed to select or develop a spatial program (possibly in the form of a FU hierarchy as required by the layout module of SEED). At a minimum, a SU consists of the building type, capacity, and site-context. Information regarding the budget, names of other special codes/regulations which are applicable to the current project, and the client's preferences are also needed (Akin, et.al 1994).

- **Functional unit (FU):** A FU is an identifiable object intended to perform a specific function or combination of functions in a building (e.g. a living room, a load-bearing wall). A FU has associated constraints and criteria on its shape, size, placement, relations with other FUs etc. A FU can contain other functional units, which are called its constituents.

- **Design unit (DU):** A DU is a part of the spatial or physical structure of a building with an identifiable spatial boundary. In a complete design, each design unit has a FU associated with it. DUs can contain other DUs so that a hierarchical decomposition of design units reflects a hierarchical decomposition of the associated FUs and vice-versa.

The application of the hybrid approach to the design of SEED-CBD's case indexing and retrieval capabilities should take into account the following requirements (Flemming, 1994):

- Case representation in SEED should be unified to extend case storage and reuse across tasks or modules and across problem levels within a module.

- Case representation should be structured around the triad *problem, solution,* and *outcome* corresponding to the problem specification, generation and evaluation components of a SEED module.

These requirements imply that each member of the triad may vary in content depending on the module or task level. For instance, a solution description in one module can be conceived as a problem specification for an other. In SEED's architectural programming module SP, SUs are conceived as problems and FUs as solutions. On the other hand, in SEED's schematic layout design module SEED-Layout, FUs are part of problem specifications. Since the CBD indexing scheme should cater to all problem levels and modules, it must provide a common interface to represent a case index on which matching is performed. It is possible that the case index consists of parts of a problem or parts of a solution depending on how a case is conceived in a module. For instance, in its current configuration, a case index for SEED-Layout is a problem that includes the current context and FUs to be allocated; it is used to retrieve the associated solution when, at a later time, a similar problem is specified. It is possible that in a future version, SEED-Layout may decide to retrieve cases based on the geometric properties of solutions. In this case, the geometric representations of DUs, which constitute a solution in SEED-Layout, may be included in the case index in order to perform case retrieval based on the geometric properties of the solution.

The classification capabilities are essential to the suggested case indexing scheme and to the SEED project in general. SEED requires a classification engine able to:

- define a taxonomy which supports subsumption, multiple inheritance, disjoint partitioning

- use the classification to retrieve prototype objects with default properties

- to speed up the retrieval of cases in the SEED-CBD engine

The requirements on the indexing scheme guide the design of the retrieval mechanism. When the case index is treated as an aggregation of objects, the selection of the matching algorithm used in retrieval depends on the type of object specified in the case index. A retrieval based on geometric properties would require the use of a geometric matching algorithm. Therefore, matching in SEED-CBD may have to support, for instance, R-Trees[1] to index the geometries in *DU*s, in order to retrieve layouts that satisfy a specified spatial containment relationship (e.g. finding the layouts that *contains*, *is-contained-by* or *overlaps DU*s).

It should be possible in SEED to retrieve cases based on their classifications, on their attribute values, on their structure with respect to their containment hierarchies, and these categories can be used alone, or in a combination.

To summarize SEED's case indexing and retrieval requirements:

- The indexing scheme should provide a common interface to build a case index incorporating various types of objects which, in turn, have associated classifications.

- The retrieval mechanism should provide a common interface for specifying a target that incorporates various matching algorithms and their associated objects to be matched against the case index components.

III . 3 Hybrid model

The *generality* and the *separation of the classification from the matching inference* are the major criteria in defining the computable model for classifying and recalling precedents. These criteria distinguish the suggested approach from other approaches to case indexing and retrieval.

- **Generality:** Generality is manifested in terms of a simple and common interface for case-base operations which allows any module in SEED to use its own semantics to define case components. Each module provides the content for case index, solution, and outcome, which are merely generic containers. Accordingly, a module's account of how the retrieval is performed and what the result should be, is captured in the content of a generic *target*. The common interface also decouples the indexing and retrieval system from its clients so that the system does not have to go through a major change when a new sub-system is introduced to SEED.

- **Separation of the *classification* from the *matching inference*:** This criterion arises from the differences between the two inference engines. The classification inference yields a TRUE or FALSE to a *is-a?* query, whereas the matching yields a degree of similarity. Classifications are represented by relatively simple data structures (e.g. classification concepts do not have the notion of equality based on recursive component identity) which nevertheless allows the engine to make complex inferences. For instance, the system can

1. A R-tree is a self-maintaining data structure for quick searching large amounts of spatial data. R-trees work well for representations of multi-dimensional objects which span a range along one or more axes (Guttman, 1984).

infer subsumption relations from the representations of classifications instead of relying on the direct assertions of these relations. The simplicity of these representations also allows for a safer use of multiple inheritance. The matching inference, on the other hand, deals with fairly complex object structures. To assure polymorphism, SEED modules use single inheritance in their object-based representation. The matching inference, therefore, deals with single inheritance hierarchies but possibly complex part-of lattices. The mechanisms for classification and matching inference can be modeled separately; however, they need to cooperate during retrieval.

Table 1 summarizes the distinctions between the two engines with respect to the kind of data they operate on. Precedents and classifications are compared based on their choice of representation and typing schemes, and on their corresponding inference mechanisms.

Conceptual model	**Precedents**	**Classification**
Entities	*cases*: solutions generated by the system *prototypes*: object prototypes with standard or default properties	concepts, individual descriptions
Representation	object-based representation: complex object configurations with behavior	description-logic based representation: design descriptions incorporating thematic features
Typing scheme	explicit naming using a rigid type lattice	subsumption relations inferred from flexible design descriptions
Inheritance	single inheritance to assure polymorphism	multiple inheritance to support multiple classification
Strategy	structural matching yielding a degree of similarity	subsumption based inference yielding TRUE or FALSE

TABLE 1. Cases, prototypes and classifications

The precedents in an object model are persistently stored as part of cases or prototypes in a case-base. Precedents have object-based representations and reside in an object-oriented database. These data objects are accessed at runtime for indexing and retrieval applications. The classifications, on the other hand, are persistently stored in parallel knowledge-bases as descriptions. The knowledge-base supports subsumption inference and performs consistency checking. The classifications have a description-logic based representation which allows for multiple inheritance. The descriptions are interactively generated and queried by a runtime classification engine.

The objects that are used to represent precedents may be assigned classifications. During retrieval therefore, the two engines may have to work in coordination. The similarity between the object configurations are measured by an object-by-object, and attribute-by-attribute matching. The subsumption relations between their corresponding classifications are determined by querying a classification knowledge-base. When both of the engines are active, the classification engine reduces the number of candidates on which a lengthy comparison will be performed by limiting the search to the objects with compatible classifications.

Software Architecture

This chapter describes the implementation of the hybrid model introduced in Chapter III. The software requirements are identified within the context of the SEED development environment. The individual component architecture is provided along with the outline of the classification knowledge-base and case-base organization and functionality. The last section describes the case-base matching and retrieval engine and discusses various retrieval options with respect to the implementation. The classification knowledge-base and the case-base will be further elaborated in the chapters dedicated to SEED-KBC and SEED-CBD, respectively.

IV . 1 Software requirements

SEED's multi-team development encourages the use of as many commercial software as possible and produces software only when it is not commercially available. SEED's strategy towards the use of commercial software applies to the development of the SPROUT modeling environment, the database support envisioned for SEED[1]. The classification knowledge-base and the case-base engine are conceived as part of SPROUT functionality. In this section I identify the packages and programming environments selected for the development of SPROUT. The integration of SEED-CBD, SEED_KBC and the SPROUT modeling environment provides an implementation framework based on which the software requirements are identified.

1. SEED Project's Representation of Objects Utilizing Technologies (Snyder, J. et.al., 1995).

1.1 Object databases

The modules in SEED make use of object-based representations. Consequently, for the persistent storage of the information generated by the modules, SPROUT favors the use of a database which supports object-based representations. Moreover, the suggested object database system should not require the use of a specific programming language such as C++, for the reasons I introduce when I discuss the need for platform-independent programming languages. The UNISQL object/relational database system meets all the requirements specified above. In addition to the provision of full object implementations, UNISQL supports an extended version of SQL for complete object management and queries.

1.2 Description logic-based classification

In order to support multiple classification of the persistently stored objects, SPROUT requires the use of a representation technique which allows for the definition of orthogonal taxonomies. The CLASSIC knowledge representation system, developed by AT&T Bell Labs, constitutes a reference implementation model for SEED-KBC. CLASSIC (Borgida, A. et.al., 1992) uses a description-logic based representation technique and has algorithms that are known to be tractable. CLASSIC concentrates on the definition of structured concepts and their organization into taxonomies. Subsumption and classification, key inferences supported by SEED-KBC, are implemented based on CLASSIC’s definitions.

1.3 Platform-independent runtime systems

SEED is a heterogeneous software environment in which multiple hardware platforms can be accommodated. The use of compiled languages such as C/C++ produce programs which need to be ported to each hardware platform incorporated within SEED’s development environment. The Java virtual machine, as a platform independent runtime system, can be used to generate program executables for multiple hardware platforms. Consequently, through the use of Java, extensive hardware-specific re-developments can be avoided. Both SEED-KBC and SEED-CBD provide application programming interfaces written in Java in order to ease the integration with the Java based server architecture envisioned for SPROUT.

IV . 2 Overview

The complete SPROUT modeling environment incorporates a shared data model, the classification model and the case-base model (Snyder, J., 1998). The SEED-KBC implements the classification model as a distinct component in order to allow for multiple classification models. In this way, each SEED module or agent can create its own classification knowledge-base. Similarly, the separate implementation of SEED-CBD allows for the management of multiple case-bases.

Another important integration issue is the notion of workspace suggested by the SPROUT system architecture. A workspace can be defined as a collection of active and accessible objects that are defined in a particular representation[1]. The SPROUT software architecture suggests that shared object representations be included into a SPROUT workspace, which in turn can incorporate subsystems such as the classification and case-base software components. The SPROUT workspace is a client to the UNISQL server - database management system. Individual modules or agents can access the SPROUT facilities (including classification and case base retrieval queries) using the application programming interfaces of their own workspaces, referred to as host workspaces. The agents are required to provide an implementation of a workspace and maintain the links between their workspace and the SPROUT workspace.

SEED-KBC and SEED-CBD provide both a C and a Java application programming interface. The functionality provided by these APIs can be accessed by host applications or host workspaces either directly by using the C API, provided they can open client connections to the UNISQL database server, or through the Java API, which creates a client connection for each transaction. The APIs can also be accessed by the agent host workspaces through SPROUT once they are incorporated to the SPROUT workspace.

The SEED-KBC and SEED-CBD components are implemented as distinct engines. They reference the objects in the SPROUT data model through global object identifiers and type signatures. A global object identifier is used to access a unique SPROUT representation of a data object. A type signature is the name of the SPROUT class from which a data object is instantiated. In SPROUT specifications, both global object identifiers and type signatures are represented by strings. SEED-CBD's runtime retrieval capabilities require access to SEED-KBC, SEED-CBD and SPROUT data models during matching, and hence are implemented in another distinct component (Figure 2). The following section provides a more detailed description of these components.

IV . 3 Components

Based on the software requirements and SPROUT's system architecture, SEED-KBC and SEED-CBD implement classification and case-base capabilities as application programming interfaces in C and Java.

3.1 Component architecture overview

The conceived base architecture (Figure 3) is common to both engines; it consists of the following components:

- **Schema:** The schema contains the object-based representations of engine-specific concepts. The classes and objects reside in a UNISQL database file. The queries that address the SEED-KBC engine trigger specific inferences that are implemented as object or class methods in the schema.

1. The definition is borrowed from (Snyder, J. 1998)

- **C - API:** The C - API is a direct interface to the engine functionality built using UNISQL's C application interface and data structures. The applications or workspaces (SPROUT or host workspaces) that use the C - API should be able to do their own database transactions management.
- **UNISQL C - API:** The UNISQL API is an interface to the database functionality consisting of a library of C functions and data structures. The API is supported by UNISQL for complete object management and queries.
- **Java Native Interface (JNI):** Java comes with hooks for working with system libraries to make calling of native methods possible. Native methods are methods that are written in languages other than Java. The Java Native Interface is a language binding supported by all Java Virtual Machines.

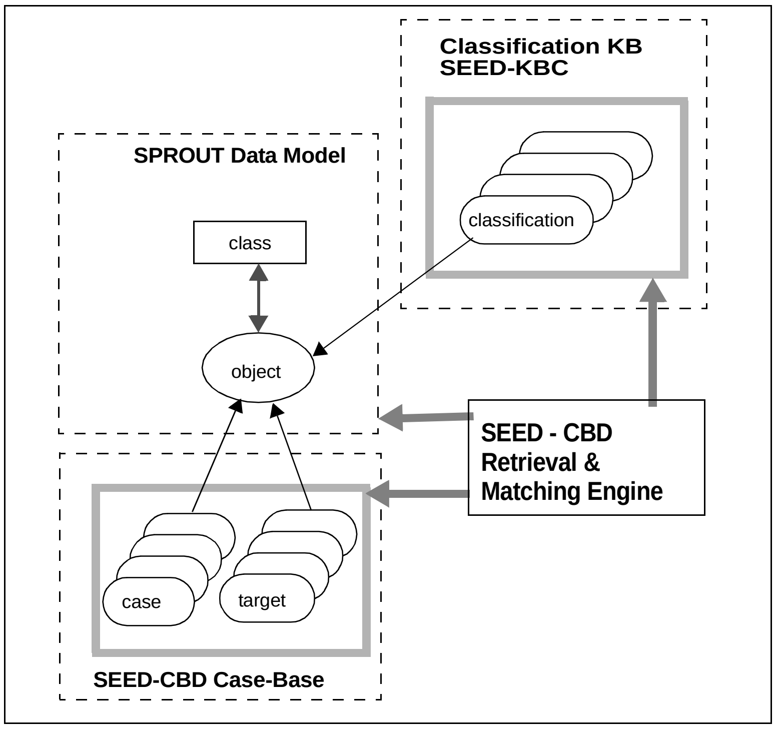


FIGURE 2. Data models for SPROUT database, SEED-CBD, SEED-KBC.

- **Java - API:** The Java API uses Java's Native Interface to connect to the functionality provided by the C -API. While calling a native method, it also opens a client connection to the target database and closes it upon the completion of the transaction. In this way, the database is locked to other client requests only within the duration of the transaction. The Java - API can be used simply by importing the java-api class into the host application[1].

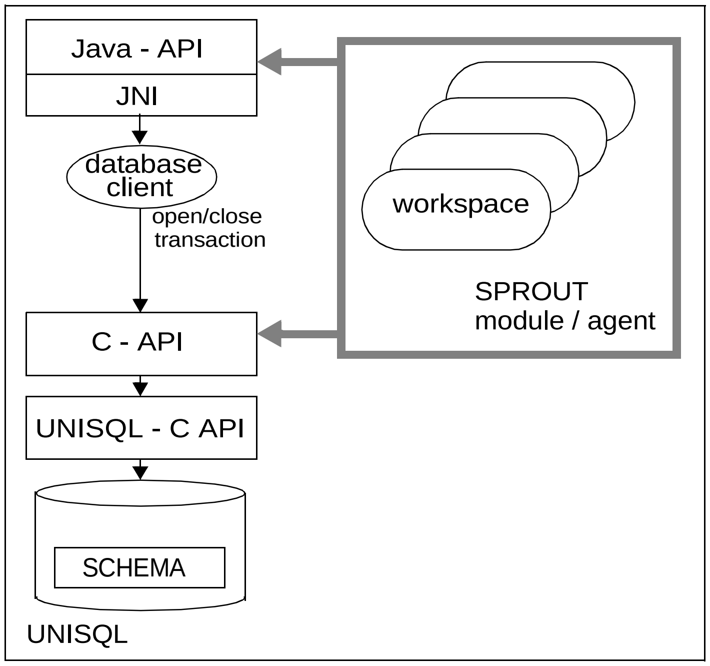


FIGURE 3. Component architecture

3.2 SEED-KBC

The classification knowledge-base is completely independent of the SPROUT data model in performing its key inferences and conflict checks. The only dependency is manifested in the object references. Objects that are persistently stored in the SPROUT database can be registered and assigned a classification in the knowledge-base. The classification knowledge-base schema requires a global object identifier and a type signature for the registry. Other SEED-KBC specific concepts that are defined in the knowledge-base schema are

- knowledge bases,
- classifications and other concepts used to define classifications (e.g. primitives),
- a dictionary of registered host objects (objects that are defined outside the classification knowledge-base) and the classification they are associated with.

1. Refer to Appendix A: Using SEED-KBC and SEED-CBD APIs for the use of these APIs.

The SEED-KBC engine functionality can be summarized under the following generic transaction types:

- **requests** to build knowledge-bases, to create and modify classification descriptions, to register and classify host objects,
- **queries** to find out the classification of a particular host object and to compare various forms of classifications in terms of their subsumption relationships.

3.3 SEED-CBD

Unlike SEED-KBC, the case-base engine contains a distinct retrieval matching engine, and hence, performs the majority of its inferences outside the case-base. The matching inference, which is triggered by the retrieval queries, is implemented outside the class and object methods that are specified within the case-base schema. The matching inference depends on the SPROUT data model as well as the classification knowledge-base. Access to various databases during matching is coordinated by an outside retrieval engine. Similar to the classification knowledge-base, SEED-CBD depends on the objects that are persistently stored in the SPROUT database. These objects can be registered as *proxies* and can be used to define case and target contents. The schema requires a global object identifier and a type signature for the registry. Other SEED-CBD specific concepts that are defined in the case-base schema are

- case bases,
- cases and targets,
- match operators.

The SEED-CBD engine functionality can be summarized under the following generic transaction types:

- **requests** to build case-bases, to create and modify case and target descriptions, to register proxies, to define match operators, to annotate cases (i.e. to allow the less structured case-specific textual information to be attached to cases in the form of annotations)
- **queries** to retrieve cases based on a target description, a classification or an annotation; and to browse case and target descriptions.

3.4 Retrieval and matching

SEED-CBD’s retrieval capabilities allow for the recall of persistently stored objects that are organized and indexed as part of cases. In a standard retrieval, recall is based on the assessment of similarity between a target description representing the problem situation and cases that reside in the case-base. The comparison involves matching between objects referenced within the case and target indices. The type of matching inference can be specified within the target by assigning a match_operator to the current matching task. The standard SEED-CBD matching inference requires the following queries to be available in the SPROUT application programming interface:

- is_subclass/is_superclass: Compare two type signatures to find out whether one is a subclass/superclass of the other.
- *is_instance_of*: Given a global object identifier and a type signature, determine whether the referred object is an instance of the class denoted by the type signature.
- *get_attributes*: Given a global object identifier, retrieve the attributes of the referred object.
- *get_attribute_value*: Given a global object identifier and an attribute path, return the specified attribute value for the referred object.

The retrieval capabilities make use of classification inferences through the following queries provided by the SEED-KBC API: *IsClassifiedSpobj()*, *IsRegisteredSpobj(), classificationCompare(), getAllClassified(), getClassification().* The method specifications for these queries can be found in Chapter V.

Another type of retrieval is classification-based retrieval where cases are recalled only if their indices reference objects with compatible classifications, where a compatible classification is either identical, equivalent or subsumed by the target classification. The last type of retrieval recalls cases solely based on annotations added by the user who defined the case. Table 2 identifies the engines that are coordinated in order to perform various types of retrieval in SEED-CBD. Chapter VII illustrates each retrieval type within a demo case-base and a sample classification knowledge-base.

retrieval	**matching inference**	**classification-based**	**annotation-based**
SEED-CBD	X	X	X
SEED-KBC	X	X	
SPROUT-DB	X		

TABLE 2. Retrieval types and engines involved

SEED's Classification Knowledge-Base

This chapter describes the implementation of SEED's classification engine in terms of its database schema and its key inferences: subsumption and classification. The specifications for the SEED-KBC Java Programming Application Interface can be found in Appendix B.

V . 1 Overview

SEED modules to capture design information by means of object-oriented representations of classes, subclasses and their instances as complex object configurations. A subclass inherits properties (attributes and behavior) from the class it is derived from. The SEED developers decided early on that the database would have to support only single inheritance because the anomalies and ambiguities inherent in multiple inheritance cannot be resolved consistently across different programming languages and object-based representations. Specifically, modules use single inheritance carefully in order to take advantage of polymorphism.

However, SEED modules require objects to be multiply classified through multiple, often orthogonal classification hierarchies. But the database's single inheritance representation scheme cannot be used for this type of classification. Therefore, SEED-KBC is set up as an independent classification engine that provides the following functionalities to overcome this shortcoming:

- means to build a taxonomy which supports subsumption, multiple inheritance among classes, partitioning with disjoint primitives for data objects.
- provision of permanent storage for the classifications along with the identifiers of the classified objects
- means to query subsumption relations between classifications.

- means to issue queries to identify objects classified by a certain classification and/or by its subsumees.
- means to maintain multiple classification knowledge-bases that allow SEED modules to operate on distinct taxonomies.

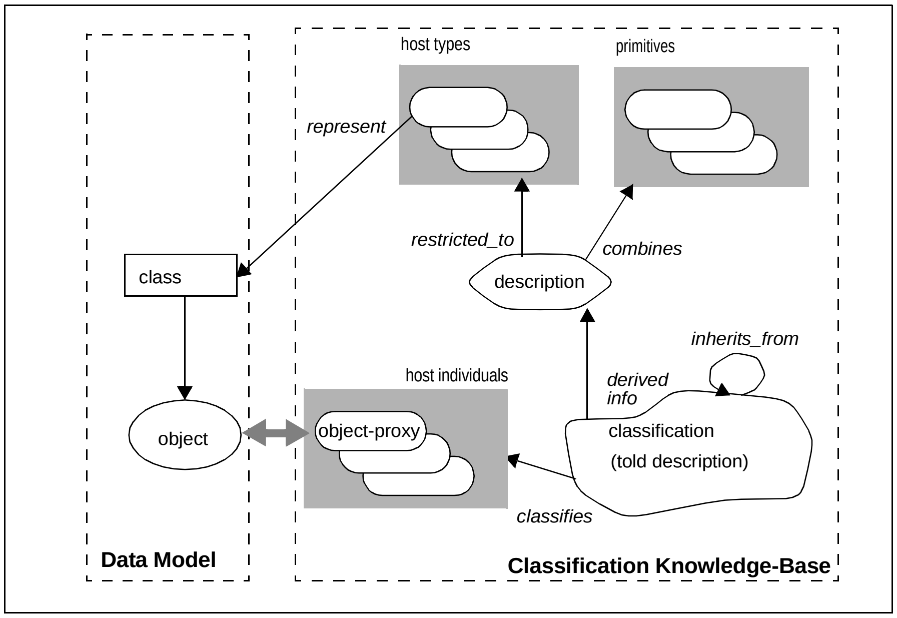


FIGURE 4. SEED-KBC

It is important to understand how classification is conceived in SEED-KBC for an efficient use of the engine. Classifications should not to be used to build complex data models incorporating geometry, tuples or series. Classifications have no notion of equivalence based on recursive component identity since they are not defined in terms of *has-a* / *part-of* relationship hierarchies. These structured partial descriptions are best used to provide thematic categorization support. The subsumption inference employed by the engine is not based on the structural properties or behavior of the classified objects. Such inferences would require the replication of the structural rationale and information inherent in the single inheritance representation of the classified objects on the classification knowledge-base side.

The classification of a data object requires the object to be registered in the knowledge base. Once the object has been registered, it can be associated with a previously defined classification. These classifications can be modified by means of adding or retracting information.

A classification knowledge-base schema resides in a database file along with a dynamically linked shared object file for the methods of the inference engine. SEED-KBC currently consists of Java and a C API incorporating the methods that access the classification knowledge-base.

V . 2 Classification

This sections describes what constitutes a knowledge-base in SEED-KBC. The basic structure of the knowledge-base schema, the concept definitions and the inference mechanisms liberally borrow from the Classic knowledge representation system (Borgida, A. et.al., 1993). Subsumption is best defined in the Classic context by Woods (1991) as follows:

> *In traditional semantic networks, the conceptual taxonomy is composed of directly asserted subsumption relations. In systems in which there are formally structured concepts, as in KL-ONE, subsumption of structured concepts can sometimes be inferred from the structures of the concepts (together with the subsumption relationships of their constituents.)*

The SEED-KBC engine can maintain multiple *kb* instances (knowledge-bases) that are specialized for various SEED modules.

2.1 KB instance

A kb instance maintains a domain of *primitives* (internally defined types), *host types* (class names or type signatures of the host objects) and *classifications*. It also maintains a dictionary of global object identifiers for host objects and the associated classifications. The domain specifications and the dictionary are specific to the kb instance and hence, cannot be shared between different kb instances.

2.2 Primitive

A *primitive* is an internal type or category residing in a single inheritance type hierarchy. Primitives are combined to form classifications. SEED-KBC recognizes two types of primitives:

- a *simple* primitive represents a categorization concept (e.g. types, residential in (Figure 5)).

- a *disjoint* primitive represents a disjoint grouping concept (e.g. composition in (Figure 5)).

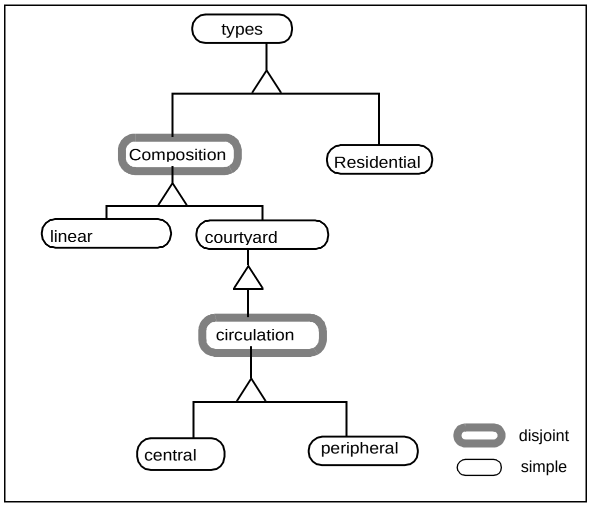


FIGURE 5. A sample primitive hierarchy.

A simple primitive is *disjunct* if it has a disjoint primitive ancestor (e.g. linear, central). A disjunct primitive conveys a concept together with the information that it DOES NOT convey any other concept represented by primitives in its disjoint grouping. Consequently, a disjunct primitive cannot be combined with another primitive in its disjoint grouping. Note that a primitive can belong to multiple disjoint groupings. For instance, the primitive peripheral in Figure 5 belongs to two disjoint groupings by having composition and circulation as its ancestors.

2.3 Host type

Host types (or host concepts) are type signatures of registered and classified objects *in the data model*. The classification knowledge-base maintains the host types to allow the user to restrict the target domain of classification assignments, if necessary, to subsets of data objects. The class inheritance relations between the host types, on the other hand, are not maintained in order to avoid replication of the information existing outside the classification knowledge-base.

2.4 Classification and description

A classification is a *told description,* which is composed of primitives and a set of restrictions. A restriction is an allowed host type for objects to be classified. A classification can in turn inherit from one or more classifications.

A told description may be modified by adding or retracting primitives, restrictions or inherited classifications. When a classification is modified, the changes are propagated to all the classifications that inherit from the altered told description. As a consequence of the updates, an altered classification may no longer classify a host individual due to changes in the restriction set.

A derived description is the information derived from a told description[1]. It contains new and inherited primitives and restrictions in normalized form. A description resides in a subsumption graph along with other derived descriptions.

2.5 Host individual

A host individual represents a database object through its unique identifier and type signature (a host type). A database object must be registered as a host individual in the knowledge base before it can be classified. Classifications are thus linked to data objects through host individuals.

2.6 KB organization

The following properties are true for the structure of the classification knowledge base:

- a host individual can be associated with at most one told description
- a told description is always associated with one (normalized & classified) description
- a classification may classify no or many host individuals
- more than one classification can be associated with the same description (they are called *synonyms*)
- A classification exists independently of host individuals
- Primitives exist independently of classifications
- When the user attempts to discard a told description, the associated description is discarded along with it only if the told description has no synonyms.

2.7 Subsumption inference

A classification C_1 subsumes another classification C_2, if C_1 is equivalent to C_2, or C_1 is more generic than C_2. More specifically, in order for C_1 to subsume C_2, for each primitive used in defining C_1 there should be an equivalent or more specific primitive in C_2. Similarly, for the host type restrictions, C_1's set of restrictions

1. See Section 2.8 for the definition of normalization process through which a derived description is generated.

should be a either an empty set (no restrictions, the most generic form) or a superset of C_2’s set of restrictions.

2.8 Normalization and classification

Normalization of a told description involves the instantiation of a derived description. The classification engine checks for possible conflicts when it combines the told primitives and restrictions with the ones derived from the inherited classifications (e.g. *disjoined primitive conflict*, *inheritance conflict*). It also eliminates primitive redundancies by keeping the most specific primitives. Once a derived description has been created it is inserted in to the subsumption graph after its subsumees and subsumers are identified.

2.9 Conflicts

A *disjoined primitive conflict* arises when there is an attempt to combine disjoined primitives (primitives in different branches of a disjoint grouping).

A *restriction conflict* arises when there is an attempt to associate a host individual with a classification which is restricted to host types other than the current host individual’s.

An *inheritance conflict* arises when a classification inherits from two disjoined classifications. Two classifications C_1 and C_2 are disjoined if a derived primitive (an inherited or a told primitive) p_1 of C_1 and a primitive p_2 of C_2 are disjoined.

V . 3 System architecture

This section describes the software architecture envisioned for the SEED-CBD engine in terms of its components and introduces the application programming interface for the classification functionality. The conceived system architecture for SEED-KBC (Figure 6) consists of the following components:

- **KB schema:** The kb schema is implemented using UNISQL’s object-based representation scheme (Appendix D: Database Representations). The KB class and instance methods that are used to maintain and query the knowledge-base are accessed by the database through a dynamically linked library file: dbmethods.so.
- **UNISQL C - API:** The UNISQL API is an interface to the database functionality consisting of a library of C functions and data structures. The API is supported by UNISQL for complete object management and queries.
- **KBAPI:** The KBAPI consists of wrapper functions implemented in C to interface the KB class and instance methods.

- **KB Java-API:** The Java KB API class provides methods to manage a classification knowledge-base session. A KBWorkspace uses JNI (Java Native interface) to call the KBAPI functions through a dynamically linked shared object file: kbapi.so. Java server agents (e.g. the case-base server) can import the java KB API class to access the SEED-KBC engine.

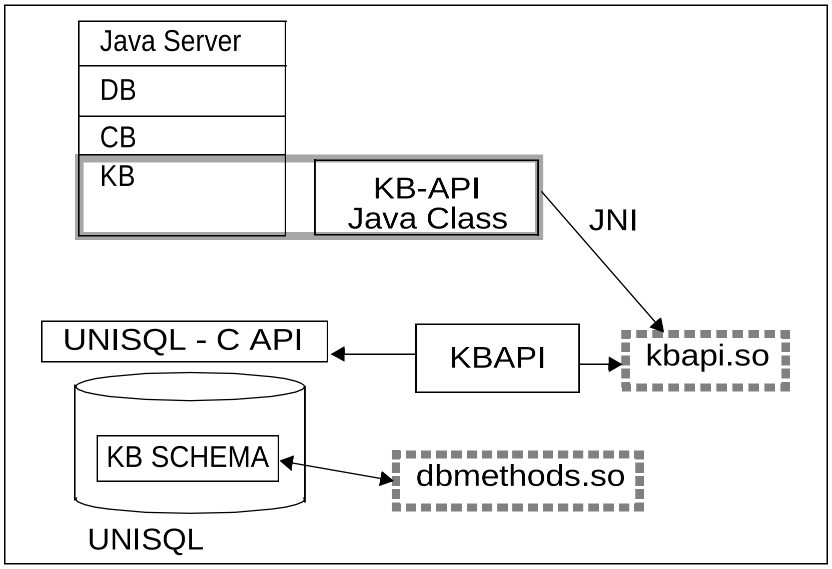


FIGURE 6. System architecture

SEED's Case-Based Design Engine

This chapter describes the implementation of SEED's case base design engine in terms of the database schema, retrieval mechanisms and matching inference. The specifications for the SEED-CBD Java Programming Application Interface can be found in Appendix C.

VI . 1 Case-base

SEED's case-base design engine provides more than a repository for precedents: it allows individual SEED modules to incorporate their module-specific reasoning within the case-base. A module's implementation for the case-base functionality customizes SEED-CBD's generic capabilities of case representation, indexing and retrieval based on the module's internal logic.

This specialization of the generic functionality happens at three different stages: the representation, indexing and retrieval. During the representation stage, the modules organize the objects they generate within a case composition. In the indexing phase, they decide which object(s) are indispensable in recalling a case. In the retrieval phase, they provide module/problem-specific matching inferences for the objects they define in the case index. The use of generic containers in the case definition gives the modules the flexibility to modify the organization, indexing and matching inference assignments at any point.

The first two layers of functionality are provided inside the case-base and use the object and class methods that are defined in the schema. The third functionality uses the same schema methods, yet is coordinated from outside the case-base. Figure 7 provides a more detailed breakdown of the case-base data model, the retrieval engine and the matching inference with respect to the other data models they interact with. From SEED-CBD's point of view, the SPROUT data model is as abstract as a repository of objects with type signatures (or classes). These objects are represented via proxies inside the case-base.

The interaction with the classification engine takes place during retrieval and matching. The SEED-KBC API is accessed whenever object classifications are compared in order to decrease the number of matching operations, or when the retrieval is solely based on a specified classification. The engine interacts more with the SPROUT data model during matching. The matching inference does not need to know about the case-base data model, since the comparison occurs between objects that are persistently stored in the SPROUT database. During retrieval, on the other hand, objects which belong to the case-base data model are compared (i.e. case and target).

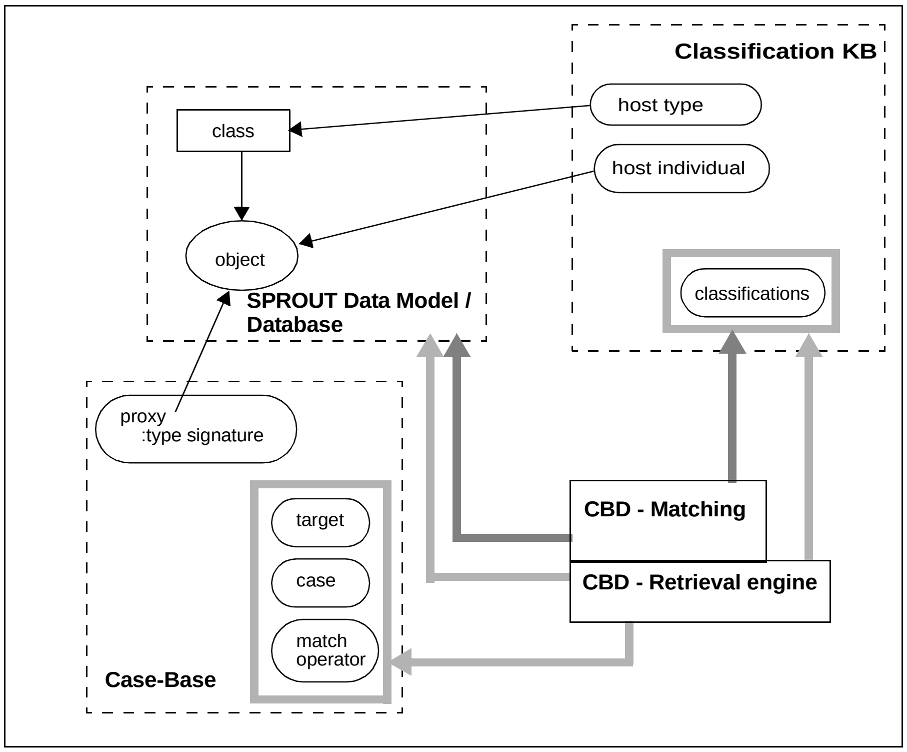


FIGURE 7. Data models and inference engines.

1.1 SEED-CBD concepts

The SEED-CBD engine can maintain multiple case-bases in which modules define and populate their *cases, targets, match operators* and register the objects from the SPROUT data model as *proxies*. This section defines and describes the concepts that are introduced as part of a case-base schema.

1.1.1 CB

The SEED-CKB engine stores multiple *cb* instances (case-bases), which are identified by their unique name. The concepts that are created within a case-base are associated with the cb instance through this uniquely identifying name. When a cb instance is discarded or cleaned up, subsequent discard calls are triggered on various case-base concepts (e.g. cases, targets) which will be described in the following sections.

1.1.2 Case

A case is composed of four containers corresponding to a problem, solution, outcome and a case descriptor. Each container implements a set of proxy object references (Figure 8). In SEED-Layout for example, a case problem is a set containing references to proxy objects that represent a Functional Unit and a context object from the SPROUT database. In the same example, the solution is another set which consists of a proxy object reference representing a layout.

A case descriptor is another container of object references which are considered to be significant in recalling a case. For example, the default descriptor for a case is its problem for SEED-Layout. However, if the module decides at one point that the retrieval of a case should consider the geometry of a layout, the case descriptor (or case index) can be augmented by a layout object reference. The proxy references contained in a case descriptor are called *case matchables*. In addition to the case index, a case can be recalled based on its *annotations*. This allows the cases to be distinguished based on unstructured information (substring match).

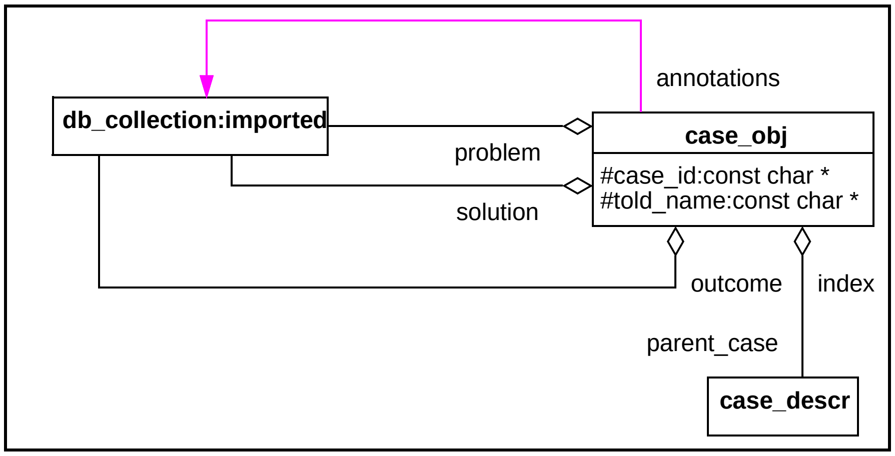


FIGURE 8. Case decomposition

1.1.3 Target

A target is similar to a case descriptor both in idea and structure. Targets are used to describe a particular problem situation; they contain information crucial to retrieve a case. This information is represented through a *target matchables* set (Figure 9). A target matchable differs from a case matchable because it contains information about how to perform a match in addition to what to perform the match on. A target matchable consists of a pair containing a proxy object and a match operator reference. If the retrieval is to be performed using the case-base's default matching mechanism, the match operator is set to default for a matchable pair.

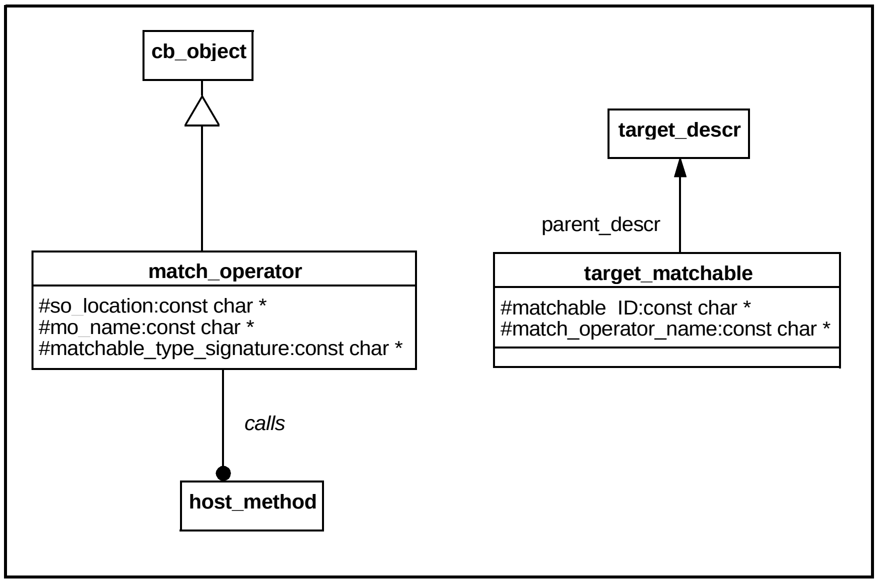

cb_object
target_descr
parent_descr
match_operator
#so_location:const char *
#mo_name:const char *
#matchable_type_signature:const char *
target_matchable
#matchable_ID:const char *
#match_operator_name:const char *
calls
host_method

FIGURE 9. Target description and match operator

1.1.4 Proxy

A proxy is a case-base representation of a data object which is persistently stored in the SPROUT database. The case-base engine requires a proxy to contain referred object's global identifier and type signature as information (Figure 7).

As an independent engine, the case-base does not perform any consistency check for the referred object in order not to replicate the inheritance information maintained in the SPROUT database. The host workspace or application which uses the API is responsible of registering the proxies and maintaining the consistency between the data objects in SPROUT and their case-base proxies.

1.1.5 Match operator

A match operator is a matching strategy which is selected for a particular retrieval session. More specifically, it is a database representation of a C function which is called to perform matching between a target and a case matchable at runtime. The function is associated with a match operator class method in the case-base schema and accessed by the UNISQL database at runtime. Since the creation of a match operator modifies the case-base schema (unlike the other transactions which solely operate on the case-base data), an invalid file reference would cause the match operator class definition to be inconsistent. Therefore, at the creation time, a match operator should be registered with a valid path to its implementation file (a shared object file). A match operator instance inherits information about its implementation function (i.e. c-function name and the shared object file location) from its parent class: *match_operator* (Figure 9).

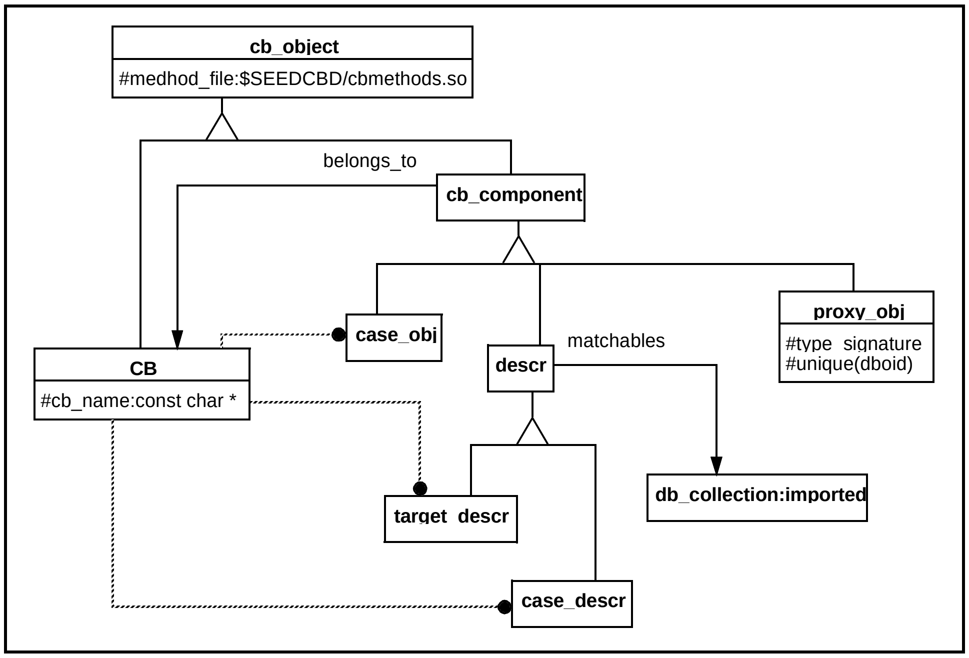


FIGURE 10. Case-base organization

1.2 Organization

The SEED-CBD database maintains multiple case-base instances for various modules based on the following requirements or facts:

- Modules may define different case contents.
- Modules may record different retrieval information.
- Modules may implement different retrieval strategies.
- Modules need to maintain different sets of proxies depending on their case-base retrieval scenarios.

Hence, each case-base instance is associated with multiple module-specific cases, targets, match operators, and proxy objects (Figure 10). These case-base components cannot be shared among different case-base instances.

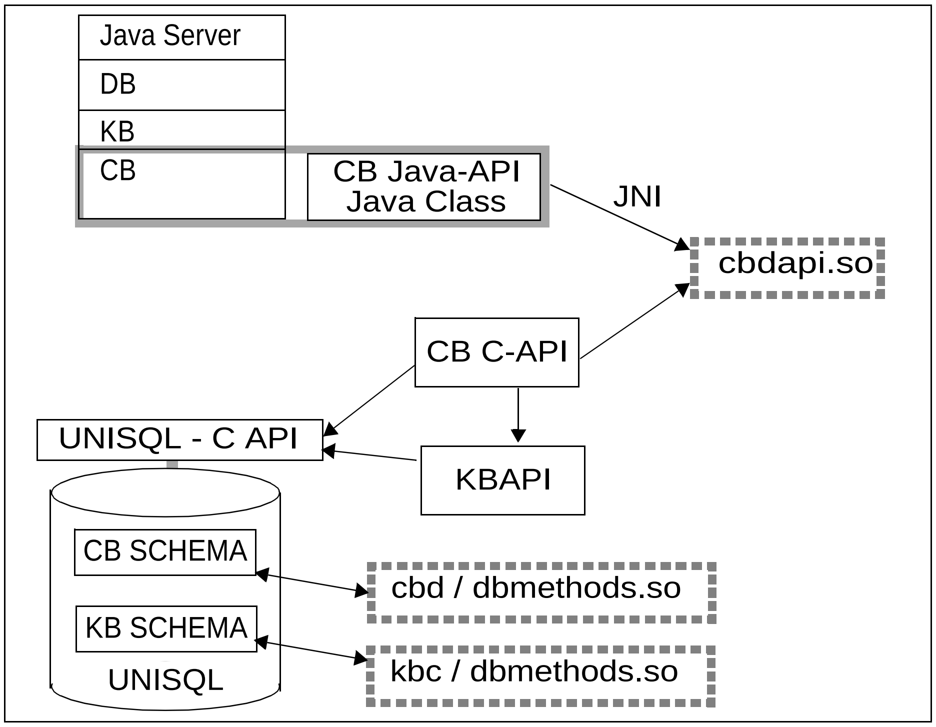


FIGURE 11. SEED-CBD system architecture

VI . 2 System architecture

The SEED-CBD system architecture (Figure 11) consists of the following components:

- **CB schema:** The cb schema is implemented using UNISQL's object-based representation scheme (Appendix D: Database Representations). The *case_base*, *case*, *proxy*, *target*, *cb_component* class and instance methods that are used to maintain and query the knowledge-base are accessed by the database through a dynamically linked dynamic library file: cbd / dbmethods.so.

- **UNISQL C - API:** The UNISQL API is an interface to the database functionality consisting of a library of C functions and data structures. The API is supported by UNISQL for complete object management and queries.

- **KBAPI:** The KBAPI consists of wrapper functions implemented in C to interface the KB class and instance methods.

- **CB C-API:** The CB C-API consists of wrapper functions which interface the *case_base*, *case*, *proxy*, *target*, *cb_component* class and instance methods.
- **CB Java-API:** The CB API is implemented via Java class methods that manage a case-base session. CB Java-API uses JNI (Java Native Interface) to call the CB C-API functions via a dynamically linked so file: cbdapi.so.

Retrieval

This chapter provides demo retrieval sessions. These retrievals are performed on a demo case-base and a classification knowledge-base in order to illustrate the coordination between two distinct inference mechanisms employed by the SEED-CBD's retrieval engine: subsumption inference and matching.

VII . 1 A demo classification knowledge-base

This section describes a knowledge-base for thematic descriptions which are used to classify a number of demo data objects. These data objects are referenced inside the knowledge-base through their unique object identifiers. In the suggested knowledge-base, a data object is represented by the concept *host individual*. A host individual is defined in terms of a unique identifier and a class reference. In the classification knowledge-base terminology, a class reference is a *host type*. Host type references the class of a data object through the class name (type signature). Host types and host individuals can be created independent of the other classification concepts. However, host types must be defined prior to the definition of host individuals.

In addition to host types and individuals, the demo classification knowledge-base consists of *classifications* and a knowledge-base instance they are associated with. Recall that in knowledge-base terminology, a classification is a *told description*. Told descriptions are defined in terms of *primitives* and restrictions targeting host types. Hence, before a told description can be created in a knowledge-base, the referenced primitives and host type restrictions must already exist. Along with these concepts, a knowledge-base instance maintains a subsumption graph for descriptions, primitive hierarchies and the records of associated host individuals and the classification pairs. The subsumption graph is derived from the descriptions, and the descriptions are derived from the told descriptions by the engine. The definitions for the knowledge-base concepts and the classification assignments, on the other hand, are provided by the designer-builder of the knowledge-base.

In order to satisfy the precedence constraints while building of a knowledge-base, the following steps must be performed in order:

1. Define primitives and register host types
2. Register host individuals
3. Define classifications
4. Assign classifications to host individuals

1.1 Primitives

When creating a primitive, the concepts that are declared in its definition must exist in the knowledge-base. Hence the building of a primitive hierarchy proceeds from top to bottom-- from the most generic primitive to the most specific primitives. A sample primitive hierarchy for the knowledge-base “SEED_Layout” is provided in Figure 12. This sample primitive hierarchy is created in order to define complex building classifications incorporating orthogonal type hierarchies such as *two-company-headquarter-army-firestation* and *one-company-satellite-army-firestation.* These building classifications combine functional and organizational concepts with concepts representing scale and centrality. Each concept category is defined as a primitive or as a disjoined primitive (e.g. *privacy*).

The overall context for this example is the design of firestations for Army-bases. The primitives in Figure 12 provide basis for classifying programmatic and spatial components in such buildings.

1.2 Host types

Host types are similar to primitives, but they do not reside in a generalization hierarchy[1]. Therefore, they can be registered in any order. The data objects which will be used in the retrieval session later in this chapter have the following types:

Building_FU
Story_FU
Massing_FU
Zone_FU
Room_FU
Building_LAYOUT
FU_Context

Each type signature listed above is represented by a host type in the suggested demo classification knowledge-base “SEED_Layout”. In the context of SEED-Lay-

1. For the reason explained in Section 2.3 of Chapter V

out, the particular types listed above represent spatial components of buildings and the requirements they must satisfy.

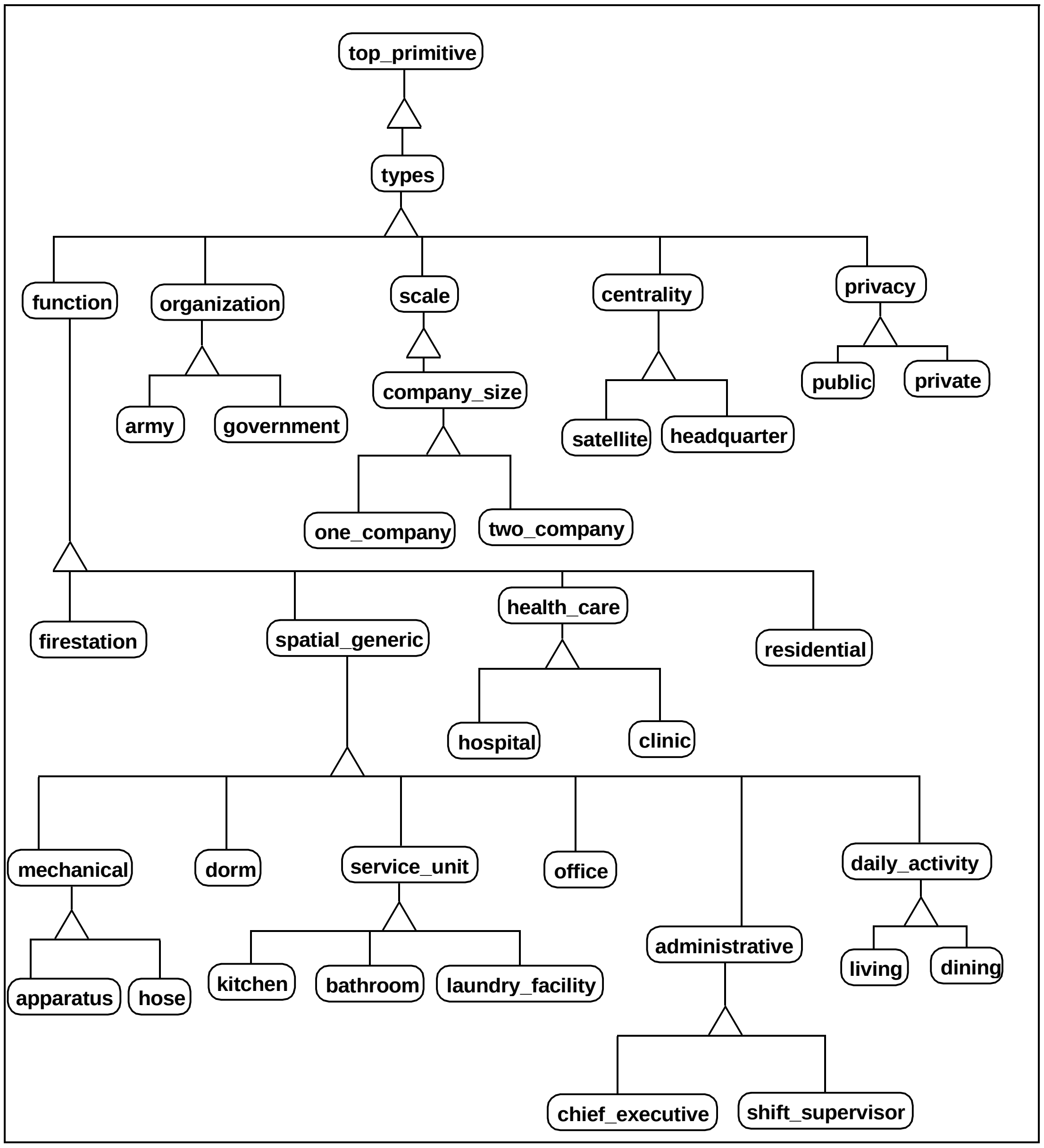


FIGURE 12. Sample primitive hierarchy

1.3 Host individuals

Host individuals must be registered in the knowledge-base before they are assigned classifications. The knowledge-base must have a definition of the host type in order to register the individual. The individuals that are registered in the test knowledge-base “SEED_Layout” are:

Unique Object Identifier	Type Signature
SPB_1	*Building_FU*
SPB_2	*Building_LAYOUT*
SPB_3	*Building_FU*
SPB_4	*Building_LAYOUT*
SPC_1	*FU_Context*
SPC_2	*FU_Context*
SPC_3	*FU_Context*
SPM_1	*Massing_FU*
SPM_2	*Massing_FU*
SPM_3	*Massing_FU*
SPM_4	*Massing_FU*
SPM_5	*Massing_FU*
SPZ_1	*Zone_FU*
SPZ_2	*Zone_FU*
SPZ_3	*Zone_FU*
SPZ_4	*Zone_FU*
SPZ_5	*Zone_FU*
SPR_1	*Room_FU*
SPR_2	*Room_FU*
SPR_3	*Room_FU*
SPR_4	*Room_FU*
SPR_5	*Room_FU*
SPR_6	*Room_FU*
SPR_7	*Room_FU*
SPR_8	*Room_FU*
SPR_9	*Room_FU*
SPR_10	*Room_FU*
SPR_11	*Room_FU*
SPR_12	*Room_FU*
SPR_13	*Room_FU*
SPR_14	*Room_FU*

Unique Object Identifier	Type Signature
`SPR_15`	*`Room_FU`*
`SPR_16`	*`Room_FU`*
`SPR_17`	*`Room_FU`*
`SPR_18`	*`Room_FU`*

Except for **`SPB_2`** and **`SPB_4`**, all objects identified in the above table represent elements of specific spatial programs for firestations. SEED_Layout is able to create layouts of these elements by assigning to each a Design Unit in a layout. **`SPB_2`** and **`SPB_4`** represent such layouts.

1.4 Classifications

Similar to primitives, the knowledge-base must have the definitions of the concepts which are used to define a classification (e.g. primitives, inherited classifications, allowed host types) at the time of its creation.

The following is a list of the classifications in the demo knowledge-base "SEED_Layout" accompanied by the classification information derived by the system.

1.4.1 Told information

The information required to define a told description consists of a unique told description name, a list of names for inherited told descriptions, a list of primitive names representing various concept categories, and a list of host type names for restricting the classification assignments to particular types of data objects. In the example below, *Basic_Building* constitutes a base description through which a set host type restrictions are identified and passed to all of the inheriting told descriptions. For instance, the classification *CL2* inherits from *Basic_Building,* and hence, it is restricted to classify objects of type *Building_FU* or *Building_Layout.* In addition to the restrictions inherited from *Basic_Building*, *CL2* contains the primitive *headquarter*. In a similar example, *CL17* inherits from the classification *CL13* which in turn is derived from another base classification *Spatial_function*. *CL17* inherits the restrictions *Zone_FU*, *Massing_FU* and *Room_FU* (from *Spatial_function* through *CL13*), and the primitive *mechanical* (from *CL13)* in addition to its own primitive *apparatus*.

Told description name	Inherits from	Primitives	Restrictions
Basic_Building			*Building_FU* *Building_Layout*
CL1		*firestation, one_company, satellite*	*Building_FU*

Told description name	Inherits from	Primitives	Restrictions
CL2	*Basic_Building*	*headquarter*	
CL3	*Basic_Building*	*satellite*	
CL4	*CL2*	*one_company*	
CL5	*CL3*	*one_company*	
CL6	*CL2*	*two_company*	
CL7	*CL3*	*two_company*	
CL8	*army, firestation*		
CL9	*government*		
CL10	*army, firestation, gov-ernment*		
Spatial_function			*Zone_FU, Massing_FU, Room_FU*
Spatial_unit_function			*Zone_FU, Room_FU*
CL11	*Spatial_function*	*dorm*	
CL12	*Spatial_function*	*administrative*	
CL13	*Spatial_function*	*mechanical*	
CL14	*Spatial_unit_function*	*daily_activity*	
CL15	*Spatial_unit_function*	*bathroom*	
CL16	*CL12*	*private*	
CL17	*CL13*	*apparatus*	
CL18	*Spatial_unit_function*	*kitchen*	
CL19	*Spatial_unit_function*	*dining*	
CL20	*Spatial_unit_function*	*chief_executive*	
CL21	*Spatial_unit_function*	*shift_supervisor*	
CL22	*CL20*	*private*	

1.4.2 Derived Information

For each told description, the KBC engine generates a derived description unless there is an equivalent derived description that already exists in the knowledge-base. The information in a derived description consists of normalized sets of (told and inherited) primitives and restrictions. Based on this information, the existing knowledge-base descriptions are re-classified with respect to the new derived description in order to identify new subsumption relationships. In the *Basic_Building* example, the derived description associated with *Basic_Building* has no subsumers since it is considered as a base description. The classification engine infers subsumption relationships that are not explicitly told. For example, *CL5* is told to inherit from the classification *CL3*, and hence it belongs to *CL3*'s set of subsumees by definition. On the other hand, the subsumee/subsumer relationship between *CL3* and *CL1* is not explicitly stated, yet inferred by the engine based on the told information. The engine also eliminates redundancies when it creates derived descriptions. For instance, the told information for *CL17* (as introduced in Section 1.4.1) contains the primitives *mechanical* and *apparatus*. The derived information, on the other hand, contains only the most specific primitive *apparatus*[1].

Classification name	Primitives	Restrictions	Subsumer	Subsumee
Basic_Building		*Building_FU Building_Layout*		*CL2, CL3, CL4, CL5, CL6, CL7*
CL1	*firestation one_company satellite*	*Building_FU*	*Basic_Building CL3, CL5*	
CL2	*headquarter*	*Building_FU Building_Layout*	*Basic_Building*	*CL4, CL6*
CL3	*satellite*	*Building_FU Building_Layout*	*Basic_Building*	*CL1, CL5, CL7*
CL4	*one_company, headquarter*	*Building_FU Building_Layout*	*Basic_Building CL2*	
CL5	*one_company satellite*	*Building_FU Building_Layout*	*Basic_Building CL3*	*CL1*
CL6	*two_company, headquarter*	*Building_FU Building_Layout*	*Basic_Building CL2*	
CL7	*two_company satellite*	*Building_FU Building_Layout*	*Basic_Building CL3*	
CL8	*army, firesta-tion*			*CL10*
CL9	*government*			*CL10*
CL10	*army firestation government*		*CL9, CL8*	
Spatial_function		*Zone_FU Massing_FU Room_FU*		*Spatial_unit_function, CL11, CL12, CL13, CL14, CL15, CL16, CL17, CL18, CL19, CL20, CL21, CL22*
Spatial_unit_function		*Zone_FU Room_FU*	*Spatial_function*	*CL11, CL12, CL13, CL14, CL15, CL16, CL17, CL18, CL19, CL20, CL21, CL22*

Classification name	Primitives	Restrictions	Subsumer	Subsumee
CL11	*dorm*	*Zone_FU Massing_FU Room_FU*	*Spatial_function*	
CL12	*administrative*	*Zone_FU Massing_FU Room_FU*	*Spatial_function*	*CL16, CL20, CL21, CL22*
CL13	*mechanical*	*Zone_FU Massing_FU Room_FU*	*Spatial_function*	*CL17*
CL14	*daily_activity*	*Zone_FU Room_FU*	*Spatial_function, Spatial_unit_fun ction*	*CL19*
CL15	*bathroom*	*Zone_FU Room_FU*	*Spatial_function, Spatial_unit_fun ction*	
CL16	*administrative private*	*Zone_FU Massing_FU Room_FU*	*Spatial_function, CL12*	*CL22*
CL17	*apparatus*	*Zone_FU Massing_FU Room_FU*	*Spatial_function, CL13*	
CL18	*kitchen*	*Zone_FU Room_FU*	*Spatial_function, Spatial_unit_fun ction*	
CL19	*dining*	*Zone_FU Room_FU*	*Spatial_function, Spatial_unit_fun ction, CL14*	
CL20	*chief_executive*	*Zone_FU Room_FU*	*Spatial_function, Spatial_unit_fun ction, CL12*	*CL22*
CL21	*shift_superviso r*	*Zone_FU Room_FU*	*Spatial_function, Spatial_unit_fun ction, CL12*	
CL22	*chief_executive private*	*Zone_FU Room_FU*	*Spatial_function, Spatial_unit_fun ction, CL16, CL20, CL12*	

1.5 Classification assignments

Once the data objects have been registered as host individuals, they can be associated with told descriptions through classification assignments. Below are the classification assignments for the previously listed host individuals. In the sug-

1. The primitive mechanical is more generic than apparatus. (See Figure 12)

gested knowledge-base, there are two objects of type *Building_FUs* (**`SPB_1`**, **`SPB_4`**) and one *Building_Layout* (**`SPB_2`**) with the same classification (*CL5--one_company, satellite*), since the *CL5*'s restriction set includes both type signatures.

Unique Object Identifier	Type Signature	Is Classified By
SPB_1	*Building_FU*	*CL5*
SPB_2	*Building_Layout*	*CL5*
SPB_3	*Building_FU*	*CL3*
SPB_4	*Building_Layout*	*CL5*
SPC_1	*FU_Context*	*CL10*
SPC_2	*FU_Context*	*CL9*
SPC_3	*FU_Context*	*CL8*
SPM_1	*Massing_FU*	*CL11 (dorm)*
SPM_2	*Massing_FU*	*CL12 (admin)*
SPM_3	*Massing_FU*	*CL13 (mechanical)*
SPM_4	*Massing_FU*	*CL12 (admin)*
SPM_5	*Massing_FU*	*CL11 (dorm)*
SPZ_1	*Zone_FU*	*CL14 (daily_activities)*
SPZ_2	*Zone_FU*	*CL12*
SPZ_3	*Zone_FU*	*CL13*
SPZ_4	*Zone_FU*	*CL11*
SPZ_5	*Zone_FU*	*CL15 (bathroom)*
SPR_1	*Room_FU*	*CL16 (private, admin)*
SPR_2	*Room_FU*	*CL17 (apparatus)*
SPR_3	*Room_FU*	*CL14*
SPR_4	*Room_FU*	*CL18 (kitchen)*
SPR_5	*Room_FU*	*CL19 (dining)*
SPR_6	*Room_FU*	*CL15*
SPR_7	*Room_FU*	*CL12*
SPR_8	*Room_FU*	*CL20 (chief_executive)*
SPR_9	*Room_FU*	*CL21 (shift_supervisor)*
SPR_10	*Room_FU*	*CL12*
SPR_11	*Room_FU*	*CL14*
SPR_12	*Room_FU*	*CL18*
SPR_13	*Room_FU*	*CL19*
SPR_14	*Room_FU*	*CL20*
SPR_15	*Room_FU*	*CL22 (private, chief_executive)*
SPR_16	*Room_FU*	*CL21*

Unique Object Identifier	Type Signature	Is Classified By
SPR_17	*Room_FU*	*CL12*
SPR_18	*Room_FU*	*CL12*

The classifications listed above can also be used to retrieve the associated database objects without having to activate the CBD engine in the process. For example, given the classification *CL12* (an administrative zone, massing or room unit), the KBC engine can be queried to retrieve the database objects associated with *CL12* (i.e. **SPR_17** and **SPR_18**). In addition to these directly classified database objects, KBC engine can also retrieve objects having classifications that are subsumed by *CL12* (i.e. **SPR_14**, **SPR_15**, **SPR_16**, and **SPR_1**).

VII . 2 A demo case-base

A case-base is a collection of cases, targets, match operators, proxies and a case-base instance they are associated with. The sample case-base described below consists of four cases, two target objects and one match operator, which constitute the minimum amount of information required to illustrate the three types of retrieval supported by the SEED-CBD engine.

In order to build a case-base, the system requires that:

1. The registration of proxies precedes the definition of cases and targets.
2. The creation of match operators precedes the definition of targets.

2.1 Proxies

Unlike the host individuals in the classification knowledge-base, proxy types do not have to be declared in the case-base prior to the creation of proxies, since the proxy types are not represented as case-base concepts. The proxies that are registered in the test case-base “SEED_Layout” are the following:

Unique Object Identifier	Type Signature
SPB_1	*Building_FU*
SPB_2	*Building_Layout*
SPB_3	*Building_FU*
SPB_4	*Building_Layout*
SPC_1	*FU_Context*
SPC_2	*FU_Context*
SPC_3	*FU_Context*

2.2 Cases

Following a case declaration, a case object with an empty content is instantiated and a unique case identifier is provided. The case descriptor, solution, problem, outcome and annotations can then be populated using this unique identifier. The case descriptor contains data object references on which matching is performed during retrieval. For example, the descriptor of *CASE_3* consists of one *Building_DU* and one *Building_FU* object reference classified as *CL5* (*one_company, satellite*) and one *FU_Context* object reference classified as *CL10* (*army, firestation, government*)[1]. The existing case definitions in the demo case-base "SEED_Layout" are the following:

Case Name	Descriptor	Problem	Solution	Outcome	Annotations
CASE_1	**SPB_1**, **SPB_2**, **SPC_1**	**SPB_1** **SPC_1**	**SPB_2**		
CASE_2	**SPB_3**	**SPB_3** **SPC_3**	**SPB_2**		
CASE_3	**SPB_1**, **SPC_2**	**SPB_1**, **SPC_2**	**SPB_4**		"odd context"
CASE_4	**SPB_4**, **SPC_3**	**SPB_4**, **SPC_3**	**SPB_4**		"bad match"

2.3 Match operators

The creation of a match operator requires that its implementation as a C procedure exists within a specified shared object file at the time of declaration. The sample match operator in the case-base "SEED_Layout" is **OP_1** is shown below:

Match Operator Name: **OP_1**
Implementation: *cbd_deep_match_retrieval*,
Shared Object File Location: "<db methods path>/rmethods.so"
Matchable Type Signature: *Building_FU*

Match operators are user-defined matching strategies that can be incorporated within the SEED-CBD's retrieval engine. *Implementation* is the name of the database method which calls the actual C implementation. In the demo case base, *cbd_deep_match_retrieval* is a stub representing an external function implementing an alternative retrieval strategy. Shared object file location indicates location of the method implementation. The UNISQL engine accesses the suggested C func-

1. In SEED-Layout, the context objects are not classified and they are always part of a problem. The context examples provided in the demo case-base have associated classifications in order to illustrate how the CBD engine deals with descriptors containing multiple data object references with classifications.

tion in runtime. The matchable type signature is used for a type checking before the match operator is launched on matchable candidates.

2.4 Targets

Similar to cases, targets can be declared and instantiated as empty target objects. The unique target identifier acquired upon the declaration can later be used to set the content for the matchables. The test case-base “SEED_Layout” contains the following targets:

Target Name	Matchables
TARGET_1	(**SPB_3**, *DEFAULT*), (**SPC_3**, *DEFAULT*)
TARGET_2	(**SPB_3**, **OP_1**), (**SPC_3**, *DEFAULT*)

The retrieval session described in Section 3.1 uses *TARGET_1* as its target description. *TARGET_1* contains a reference to a *Building_FU* object classified as *CL1* (*firestation, one_company, satellite*) and another reference to a *FU_Context* object classified as *CL8* (*army, firestation*). The *DEFAULT* keyword indicates that the default SEED-CBD strategies will be employed during retrieval instead of a user-defined match operator.

VII . 3 Sample retrieval sessions

This section provides the output of three retrieval sessions for the demo case-base “SEED_Layout”.

3.1 Retrieval by matching

The retrieval by matching uses a target to rank the cases in the case-base. The retrieval starts in the case-base. The cases with type signatures that do not match the type signatures of the target are filtered out. A case has matching type signatures with a target if for every type signature (t_t) in the target type signature set there is one type signature (t_c) in the case type signature set such that t_t denotes a class which is the same as or a superclass of the class denoted by t_c. Following the pre-selection, the case descriptors are matched to the target descriptor. In the following retrieval example, the pre-selection based on type signatures is not effective since all the cases defined in the demo case-base have matching type signatures with the target *TARGET_1*. Consequently the demo retrieval considers all cases for matching.

The matching between the target and case descriptor compares respective sets of matchables. If a target matchable is classified in a specified classification knowledge-base, only the case matchables containing compatible classifications are considered for comparison. A case matchable classification is compatible with a target classification if it is equal to, a synonym of, or subsumed by the target classification. The classification-based filtering takes place in the classification knowledge-base.

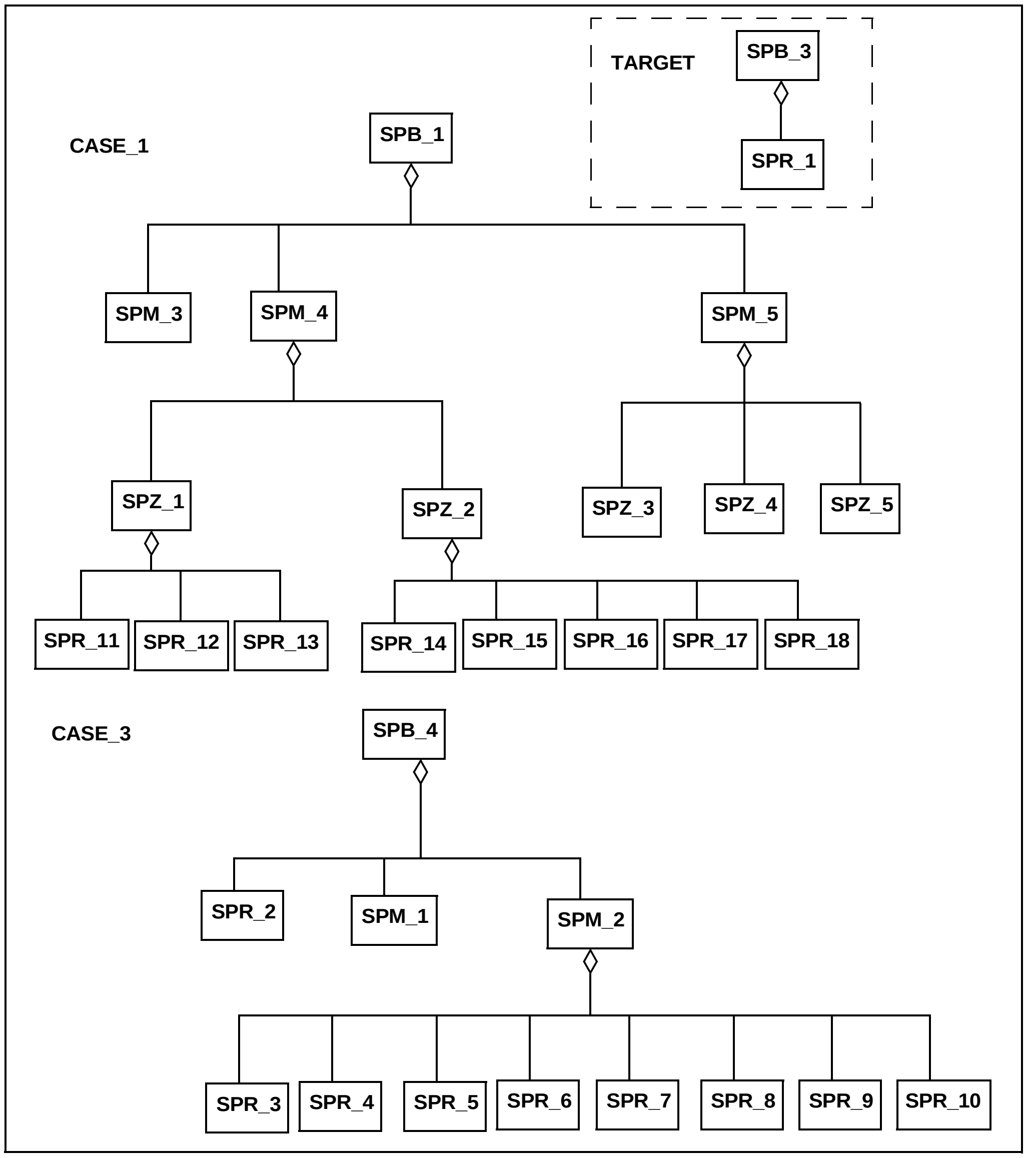


FIGURE 13. Proxy object configurations

Finally, the SPROUT object configurations (as represented in Figure 13) denoted by the target and case matchables are compared. This involves a structural matching, which proceeds object-by-object and attribute-by-attribute and returns a value between 0 and 1, where a 0 would indicate that there has been no match between the compared objects[1]. In this phase of matching, both the SPROUT database and the SEED-KBC are queried.

In the following example, *TARGET_1* is used to rank cases: *CASE_1, CASE_2, CASE_3* and *CASE_4*. *TARGET_1*'s descriptor contains a Building FU (**SPB_3**) classified as *CL3* (a basic_building with a satellite primitive) and a Context object classified as *CL8* (army, firestation). **SPB_1** (classified as *CL5*: a one company, satellite basic_building) and **SPB_4** (classified as *CL5*) are identified as matching candidates since they have classifications that are compatible with *CL3*. In the structural matching phase, the target matchable **SPB_1** with a constituent **SPR_1** (classified as *CL16*: a private and administrative spatial function) is compared to the constituent hierarchies **SPB_1** and **SPB_4**. As a result of the comparison, **SPB_1** is ranked as a better match by having **SPR_15** (classified as *CL22*: a private spatial function for a chief executive) as a constituent. In this example, the structural matching between objects **SPB_3 - SPB_1**, **SPB_3 - SPB_4** (Figure 13) and **SPC_3 - SPB_1**, **SPC_3 - SPC_2** is performed using stubs simulating a SPROUT-database connection. The output below represents the comparison between the target and case descriptors.

```
> cb retrieve TARGET_1 SEED_Layout

*** UniSQL/X Client Release 3.5.3 Patch Level 4 ***
   Generated Nov 18 1997 at 16:15:53
Cases unranked
target id: #TARGET_1#SEED_Layout
Building_FU
FU_Context
potential case ids:
#CASE_1#SEED_Layout
#CASE_2#SEED_Layout
#CASE_3#SEED_Layout
#CASE_4#SEED_Layout
```

At the first stage the potential matches are identified through type-signature filtering.

```
Case matchables for #CASE_1#SEED_Layout: SPB_1
SPB_2
SPC_1
Target matchable op # 0: DEFAULT
Target matchable id # 0: SPB_3
Target: SPB_3 with CL3 and case: SPB_1 with CL5
```

1. Refer to the matching algorithms deep_match and base_match defined in (Flemming, U. et.al. 1996).

```
TARGET SUBSUMES
PROCEEDING to structural MATCH...
Max = 1.000000, val = 1.000000
Target: SPB_3 with CL1 and case: SPB_2 with CL5
Max = 1.000000, val = 0.000000
Target: SPB_3 with CL3 and case: SPC_1 with CL10
Max = 1.000000, val = 0.000000
Target_sum = 1.000000
Target matchable op # 1: DEFAULT
Target matchable id # 1: SPC_3
Target: SPC_3 with CL8 and case: SPB_1 with CL5
Max = 0.000000, val = 0.000000
Target: SPC_3 with CL8 and case: SPB_2 with CL5
Max = 0.000000, val = 0.000000
Target: SPC_3 with CL8 and case: SPC_1 with CL10
TARGET SUBSUMES
PROCEEDING to structural MATCH...
Max = 1.000000, val = 1.000000
Target_sum = 2.000000
final_match_value = 1.000000
```

The target matchables **SPB_3** and **SPC_3** are compared with *CASE_1*'s matchables **SPB_1** and **SPC_1** (Structural match between **SPB_3** and **SPB_1** -- See Figure 12)

Final match value for *CASE_1*

```
Case matchables for #CASE_2#SEED_Layout: SPB_3
Target matchable op # 0: DEFAULT
Target matchable id # 0: SPB_3
Max = 1.000000, val = 1.000000
Target_sum = 1.000000
Target matchable op # 1: DEFAULT
Target matchable id # 1: SPC_3
Target: SPC_3 with CL8 and case: SPB_3 with CL3
Max = 0.000000, val = 0.000000
Target_sum = 1.000000
final_match_value = 0.500000
```

The target matchables **SPB_3** and **SPC_3** are compared with *CASE_2*'s matchable **SPB_3**

Final match value for *CASE_2*

```
Case matchables for #CASE_3#SEED_Layout: SPB_1
SPC_2
Target matchable op # 0: DEFAULT
Target matchable id # 0: SPB_3
Target: SPB_3 with CL3 and case: SPB_1 with CL5
TARGET SUBSUMES
PROCEEDING to structural MATCH...
```

```
Max = 1.000000, val = 1.000000
Target: SPB_3 with CL3 and case: SPC_2 with CL9
Max = 1.000000, val = 0.000000
Target_sum = 1.000000
Target matchable op # 1: DEFAULT
Target matchable id # 1: SPC_3
Target: SPC_3 with CL8 and case: SPB_1 with CL5
Max = 0.000000, val = 0.000000
Target: SPC_3 with CL8 and case: SPC_2 with CL9
Max = 0.000000, val = 0.000000
Target_sum = 1.000000
final_match_value = 0.500000
```

The target matchables **SPB_3** and **SPC_3** are compared with *CASE_3*'s matchables **SPB_1** and **SPC_2** (Structural match between **SPB_3** and **SPB_1** -- See Figure 12)

Final match value for *CASE_3*

```
Case matchables for #CASE_4#SEED_Layout: SPB_4
SPC_3
Target matchable op # 0: DEFAULT
Target matchable id # 0: SPB_3
Target: SPB_3 with CL3 and case: SPB_4 with CL5
TARGET SUBSUMES
PROCEEDING to structural MATCH...
Max = 0.000000, val = 0.000000
Target: SPB_3 with CL3 and case: SPC_3 with CL8
Max = 0.000000, val = 0.000000
Target_sum = 0.000000
Target matchable op # 1: DEFAULT
Target matchable id # 1: SPC_3
Target: SPC_3 with CL8 and case: SPB_4 with CL5
Max = 0.000000, val = 0.000000
Max = 1.000000, val = 1.000000
Target_sum = 1.000000
final_match_value = 0.500000
```

The target matchables **SPB_3** and **SPC_3** are compared with *CASE_4*'s matchables **SPB_4** and **SPC_3** (No structural match between **SPB_3** and **SPB_4** -- See Figure 12)

Final match value for *CASE_4*

```
RESULT SEQUENCE: {
'#CASE_1#SEED_Layout',
'#CASE_4#SEED_Layout',
'#CASE_3#SEED_Layout',
'#CASE_2#SEED_Layout'}
```

3.2 Retrieval by classification

The classification-based retrieval starts in the classification knowledge-base. Target in this retrieval case is the classification *CL8* (army, firestation) which subsumes the classification *CL10* and consequently classifies the SPROUT objects **SPC_1** and **SPC_3**. In the next phase, the SEED-CBD engine retrieves the cases which contain **SPC_1** and **SPC_3** in their descriptors. The following output is the resulting set of the retrieval based on *CL8*: *CASE_1* with **SPC_1** and *CASE_4* with **SPC_3**; that is, *CASE_1* and *CASE_4* are found.

```
> cb retrieve_by_classification SEED_Layout SEED_Layout CL8

*** UniSQL/X Client Release 3.5.3 Patch Level 4 ***

    Generated Nov 18 1997 at 16:15:53

set{'#CASE_1#SEED_Layout', '#CASE_4#SEED_Layout'}
```

3.3 Retrieval by annotation

The last example performs a substring match on case annotations and returns an unordered set of cases that have annotations containing “odd” as a substring.

```
> cb retrieve_by_annotation SEED_Layout “odd”

*** UniSQL/X Client Release 3.5.3 Patch Level 4 ***

    Generated Nov 18 1997 at 16:15:53

set{'#CASE_3#SEED_Layout'}
```

Conclusions

This chapter identifies the contributions of this research and outlines possible research directions and enhancements based on the work accomplished in designing and building a case-base engine for building design.

VIII . 1 Contributions

This research investigates classification of architectural precedents and introduces a classification scheme which is of potential use in computational design systems for a broad range of problems and domains. In generic terms, this study establishes common principles and patterns in seemingly different knowledge areas and describes their use in a particular problem domain to improve the known techniques. At the conceptual modeling level, the contributions can be summarized under the following categories:

- **Precedent classification:** the research proposes a general framework of memory organization borrowed from Tulving (1972), and Smith and Medin's review on approaches to the representation of concepts and categories (1981) and specializes them for the design context. The specialized framework is used to develop a classification and representation scheme for cases and prototypes as part of a case-base design system. In doing so, the relevance of the generic definitions and mechanisms have been tested in the implementation of a prototype case indexing and retrieval system.

- **Typology:** the development of the classification scheme also benefited from a review of architectural literature on types and typology. The general framework of memory organization provided a base of reference for a structured survey of known approaches to type and typology. The modeling of SEED-CBD and SEED-KBC engines depended heavily on the pertinent issues identified by the typology survey.

- **Case-base design:** The role of classification in a case indexing mechanism is identified in order to build a comprehensive model of case-memory. Based on the prominent issues identified in the context of indexing, a recall mechanism is developed for the suggested case-memory organization. The major components of this mechanism are the classification inference and matching engines.

The combination of the following features is novel in the suggested approach to case-base indexing and retrieval:

- **generality:** For the indexing mechanism, a case index is merely a container of objects with classifications. The semantics for a case index that resolves whether a case is retrieved based on its problem specification, outcome, or solution is left to individual applications using the CBD, which do not have to be SEED modules. Similarly, the retrieval mechanism allows the applications to specify their own matching operations when they need to employ domain specific reasoning. The suggested functionality will be accessed through a common interface. Consequently, the indexing and retrieval mechanisms will not be affected by the addition or removal of sub-systems as clients.

- **hybrid approach to model a case memory:** The classification is separated from the matching inference. This enables the applications to modify their classification knowledge-base without having to modify their domain knowledge for cases.

- **extensibility of the classification scheme:** As a follow-up to the previous feature, a common interface to support the functionality to add, remove or modify the classifications is provided. This way, notions that are new to the case-base's knowledge domain can be introduced to extend the existing classification scheme.

SEED-CBD is assisted by a distinct classification engine and offers numerous advantages when compared to the existing case-base indexing and retrieval approaches, which can be grouped under *heterogeneous* and *unified* representation systems.

Heterogeneous representation systems often work with loosely-structured representations in order to incorporate case information in various formats (drawings, multi-media files). This approach is typical of electronic libraries. The case indexing and retrieval are rarely structured around a problem/solution pair since the system itself is not a problem solver. These systems use feature lists instead (or list of attribute value pairs) to describe cases. These features are based on issues relevant to the design context. A case index is a set of selected attribute/value pairs, where attributes correspond to the design issues that are identified as the key issues. The case indices do not reside in a type hierarchy. Heterogeneous systems are often implemented using relational databases.

Unified representation systems use solutions generated by the system to solve similar problems, and hence the problem/solution pairing is critical for these systems. The case information is structured using a unified representation scheme such as an object-based language. These schemes can incorporate various types of inheritance (e.g. types and structural inheritance) using class-based and compositional hierarchies. Indexing is conceived within the same representation scheme and often implemented using class-based inheritance. Unified systems can be implemented using object-relational databases.

None of these implementations offers an indexing and retrieval mechanism with the capability to examine both structural properties and thematic descriptions for similarity assessment. Moreover, retrieval focuses on the problem alone, excluding the possibility to retrieve cases based on a specific structural pattern, which may only be represented at the solution level. SEED-CBD's indexing and retrieval capabilities make use of both types of information:

- the information available in the computational representation of a design case, and
- the thematic information which may have to reside outside the case-base scheme.

Both the object-based representation of cases and the thematic classification scheme are implemented using an object-relational database. The indexing is not limited to a set of features as in the first group of or to a class-based inheritance as in the second group. The retrieval coordinates two distinct inference mechanisms (subsumption and matching) in order to support as many retrieval scenarios as possible.

In Table 3, existing case-base design systems are compared to the suggested SEED-CBD engine in terms of case content, indexing, use of types and classifications, and the retrieval capabilities.

VIII . 2 Future research directions

In this section I outline possible enhancements that would benefit this research and identify future research directions. The enhancements can be conceived in both the classification and case-base design components of the suggested hybrid system.

2.1 Classifications with roles

The current implementation of the classification engine works with descriptions which can inherit from each other. The subsumption relations and the disjoined classifications are identified by a comparison of the descriptions that are derived from the user-defined classifications. The descriptions combine multiple primitive concepts in their definitions. These primitives, in turn, reside in a type hierarchy, and the inheritance relations between them are directly asserted by the user of system. In addition to the type-subtype relations, primitives can also form disjoined groupings.

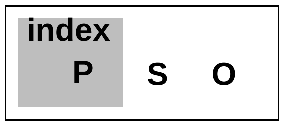
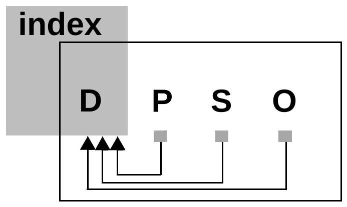
Function and feature	**Existing**	**Proposed**
case		
index	1- Feature list, collection of attribute-value pairs 2- Object-based	A collection of complex object configurations
types	1- None 2- Object inheritance hierarchy (limited to case vocabulary)	Object inheritance hierarchy & classification knowledge base
retrieval	1- Queries a relational database representation 2- Queries an object-relational database representation	Matching complex object configurations & subsumption query for the classifications

TABLE 3. Comparison between the suggested and existing systems

The descriptions, however, do not incorporate *roles* (in the CLASSIC sense) which would allow the users to define their own dependencies between classifications. Roles can be described as relations that are created outside classifications, and that can be used across classifications. A role can associate classifications with other classifications or with concepts belonging to a host domain. Using CLASSIC's terminology, a host domain, which is limited to the SPROUT type signatures in SEED-KBC, can be augmented to include basic types (e.g. integers, strings), and accordingly roles can be defined on these new domains. Currently, SEED-KBC supports only the built-in role: *restrict_to*. *Restrict_to* has the host types as its domain.

2.2 User Interface (UI) for case-base and classification knowledge-base

The current interaction with the SEED-KBC and SEED-CBD engines is accommodated at the application programming interface level. This is mainly an attempt to keep the case-base design and classification support at a generic level so that each SEED module can build its own user interfaces in order to customize the offered functionalities according to module-specific needs. There is, on the other hand, a considerable overlap in the use-cases for module-specific CBD and classification applications. A set of common use cases can be identified for various modules and can be implemented as part of a UI application framework. The modules can benefit from a standard set of classes or libraries for interacting with the case-base and classification knowledge-base at the UI level.

2.3 Matching strategies

The suggested retrieval engine, by default, operates using two matching strategies: *deep_match* and *base_match*[1]. The SEED-CBD architecture, on the other hand, allows for the creation and the use of match operators which can implement various other matching algorithms. The case-bases can be extended to incorporate a library of matching strategies which would cater to module-specific retrieval scenarios.

2.4 Combining match operators

SEED-CBD's current definition of a target description allows for a pairing of one match operator with one matchable object. The match engine can be enhanced to allow for association of conjoined and disjoined operators with each matchable in order to refine or broaden the search. Although there is no direct mechanism to apply multiple operators to one matchable, it is possible to simulate the disjoined operator behavior in the current implementation by creating a target containing multiple matchable-operator pairs with identical matchables. The SEED-CBD engine will perform the matching using all the specified operators and will only consider the pair with the highest match value.

1. Refer to the matching algorithms deep_match and base_match defined in (Flemming, U. et.al. 1996).

References

Aadmodt, A. and Plaza, E. (1993) “Case-based reasoning: foundational issues, methodological variations, and system approaches” in *AICom: Artificial Intelligence Communications*, 7(1), (url: http://www.iiia.csic.es/People/enric/AICom_ToC.html, 1997).

Akin, O., Donia, M., and Sen, R. (1994) “SEED-Pro: A framework for computer supported architectural programming”, (url: http://seed.edrc.cmu.edu/SP/carlsbad.html, 1999).

Akin, O., Donia, M., Sen, R. and Zhang, Y. (1995) “SEED:-Pro: computer assisted architectural programming in SEED” in *Journal of Architectural Engineering*, ASCE, 1(4), pages 153-161.

Argan, G. C. (1963) “On the typology of architecture” in *Architectural Design*, December, pages 564-5.

Aygen, Z. and Flemming, U. 1998. “Classification in SEED-CBD: A hybrid approach for case-indexing and retrieval.” *Proceedings of CAADRIA `98: 3rd Conf. on Computer-Aided Architectural Design Research in Asia*. Japan.

Bandini, M. (1984) 'Typology as a form of convention” in *AA Files*, vol. 6, pages 73-81.

Borgida, A., Brachman, R. J., McGuiness, D. L. and Resnick, L.A. (1992) Classic: A Structural Data Model for Objects, tech. rept., AT&T Bell Laboratories, Murray Hill, NJ.

Borgida, A. and Patel-Schneider, P. (1994) “A semantics and complete algorithm for subsumption in the CLASSIC description logic” in *Journal of Artificial Intelligence Research*, vol. 1, pages 277-308.

Colquhoun, A. (1969) “Typology and design method” in *Meaning in Architecture*, C. Jencks and G. Baird (eds.), The Cresset Press, London, pages 43-49.

Domeshek, E. and Kolodner, J. (1992) “A case-based design aid for architecture” in *Proceedings of the Second International Conference on Artificial Intelligence and Design*, J.S. Gero, (ed.), The Netherlands: Kluwer Academic Press, pages 497-516.

FABEL (1997) F. Gebhardt (ed.), (url: http://nathan.gmd.de/projects/fabel/prototype.html, 1999)

Flemming, U. (1994) “Case-based design In the SEED system” in *Knowledge-Based Computer-Aided Architectural Design,* G. Carrara and Y.E. Kalay (ed.s) Amsterdam, Netherlands, Elsevier.

Flemming, U., Aygen, Z., Coyne, R., Snyder, J. (1996) “Case-based design in a software environment that supports the early phases in building design” in *Issues and Applications of Case-Based Reasoning to Design*, Maher, M. L. and Pu, P. (eds) Lawrence Erlbaum Associates.

Freeston, M. (1995) “A general solution of the n-dimensional B-tree problem” in *Proc. of the 1995 ACM SIGMOD,* SIGMOD Record, New York, pages 80-91.

Frege, G. (1892) “On sense and nominatum” in *Readings in Philosophical Analysis*, H. Feigl and W. Sellars (eds.), Appleton-Century-Crofts, 1949, pages 85-102.

Fu, A. L. (1997) Content-Based Image Indexing (url: http://www.cs.cuhk.hk/~drsam/Index.html, 1999)

Gamma, E., Helm, R., Johnson, R. and Vlissides J. (1994) Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA.

Guttman, A.: (1984) “R-trees: a dynamic index structure for spatial searching” in *Proc. of the 1984 ACM SIGMOD*, SIGMOD Record, New York, pages 47-57.

Hinrich, T. R. (1992) Problem Solving In Open Worlds, Lawrence Erlbaum Associates, NJ.

Jackendoff,R. (1994) Consciousness and Computational Mind, Cambridge, Mass., The MIT Press.

Janetzko, A.D and G. Strube (1991) “Case-based reasoning and model-based knowledge acquisition“ in *Engineering and Cognition*, First Joint Workshop Proceedings, F. Schmalhofer,G. Strube and T. Wetter (ed.s), Berlin, Germany, Springer-Verlag, pages 99-114.

Kolodner, J., (1984) “Retrieval and Organizational Strategies” in *Conceptual Memory - A Computer Model*, Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey.

Kolodner, J., (1993) Case-Based Reasoning, Morgan Kauffman Publishers Inc., CA.

Kumar, B. and Raphael, B., (1996) “CADREM: A Case-based System for Conceptual Structural Design” in *International Journal of Engineering with Computers,* Springer-Verlag,.

Kumar, H. S. and C. S. Krishnamoorthy (1995), “A framework for case-based reasoning in engineering design” in *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, Cambridge University Press, vol. 9, pages 161-182.

Leupen, B., Grafe, C., Kornig, N., Lampe, M. and Zeeuw, P.D. (1997) “Design and typology” in *Design and Analysis*, Van Nostrand Reinhold, Rotterdam, Netherlands, pages 132-149.

Maher, M.L, (1994) “Using case-based reasoning for design media management” in *Computing in Civil Engineering*, pages 25-32.

Maher, M.L., and Zhang, D.M. (1993) “CADSYN: a case-based design process model” in *Artificial Intelligence in Engineering, Design, and Manufacturing*, 7(2) 97-110.

Maher, M. L., Balachandran, M. B. and Zhang, D. M. (1995) Case-Based Reasoning in Design, Lawrence Erlbaum Assoc., New Jersey.

Moneo, R. (1978) “On Typology.” in *Oppositions: A Journal for Ideas and Criticism in Architecture*, vol. 13, pages 22-45.

Navichandra, D. (1990) Innovative Design Systems: Where are we, and where do we go from here, Robotics Institute Technical Report, CMU, Pgh. PA.

Oechslin, W. (1986) “Premises for the resumption of the discussion on typology” in *Assemblage*, MIT Press, vol. 1, pages 37-53.

Oxman, R. (1994) “A computational model for the organization of case knowledge of a design precedent” in *Design Studies*, 15 (2).

Papadias, D., Sellis, T., Theodoridis, Y. and Egenhofer, M. J. (1995) “Topological relations in the world of minimum bounding rectangles: a study with R-trees” in *Proc. of the 1995 ACM SIGMOD*, SIGMOD Record, New York, pages 92-103.

Pevsner, N. (1976) A History of Building Types, Princeton University Press, Princeton, New Jersey.

Purves, A. (1982) “The persistence of formal patterns” in *Perspecta: The Yale Architectural Journal*, vol. 19, pages 138-163.

Quine, W. V. (1961) “Two dogmas of empiricism” in *From a Logical Point of View*, Harvard University Press, Cambridge.

Quatremere de Quincy, A. C., [1825] (1977) “Type with an introduction by A. Vidler” in *Oppositions 8* (Spring), pages 148-150.

Ramaswamy, S. and Kanellakis P. C. (1995) “OODB indexing by class-division” in *Proc. of the 1995 ACM SIGMOD*, SIGMOD Record, New York, pages 139-50.

Raphael, B. and B. Kumar (1996) “Indexing and retrieval of cases in a case-based design system” in *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, Cambridge University Press, pages 47-63.

Reich, Y. and Fenves, S. J. (1995) “A system that learns to design cable-stayed bridges” in *Journal of Structural Engineering*, ASCE, July.

Resnick, L. A., Borgida, A., Brachman, R. J., McGuiness, D. L., Patel-Schneider, P. and Zalondek, K.C. (1993) CLASSIC Description and Reference Manual For the COMMON LISP Implementation Version 2.1.

Rosenman, M. A., Gero, J. S. and Oxman, R. E. (1992) “What’s in a case: the use of case bases, and databases in design” in *Proc. CAAD Futures 91*, G. Schmitt (ed.), Wiesbaden.

Rossi, A. (1982) “Typological questions” in *The Architecture of the City*, MIT Press, pages 35-45.

Scruton, R. (1979) “The language of architecture” in *The Aesthetics of Architecture*, Princeton University Press, Princeton, New Jersey.

Simon, A. H. (1969) Sciences of The Artificial, MIT Press, Cambridge, MA.

Smith, I.F., D. Kurmann and G. Schmitt (1994) “Case combination and adaptation of buildingspaces” in *Computing in Civil Engineering*, pages 155-162.

Smith, E. E. and Medin, D. L. (1981) Categories and Concepts, Cambridge, Mass., Harvard University Press.

Snyder, J., Aygen, Z., Flemming, U. and Tsai, J. (1995) “SPROUT - a modeling language for SEED” in *Journal of Architectural Engineering*, ASCE, 1(4), pages 195-203.

Synder, J., (1998) Conceptual Modeling and Application Integration in CAD: The Essential Elements. Ph.D. Thesis, Department of Architecture, Carnegie Mellon University, Pittsburgh, PA.

Sowa, J. F. (1984) Conceptual Structures: Information Processing in Mind and Machine, Reading, Mass., Addison-Wesley.

Sullivan, L. H. (1947) “The tall office building artistically considered” in *Kindergarten Chats and other Writings*, NY, Shultz, pages 202-13.

Tezar, P. (1991) “The other side of types” in *Type and the (Im)Possibilities of Convention*, G. Rockcastle (ed.), Princeton Architectural Press, New York, NY, pages 165-175.

Tulving, E. (1972) “Episodic and semantic memory” in *Organization of Memory*, Tulving, E. and W. Donaldson (eds.), NY, Academic Press, pages 381-403.

Vidler, A. (1977) “The third typology” in *Oppositions*, MIT Press, vol. 7, pages 1-4.

Vidler, A. (1976): “Introduction: a note on the idea of type in architecture” in *The Building Of A Club: Social Institution and Architectural Type*, 1870 - 1905, Princeton University Press, Princeton, New Jersey.

Waltz D., (1991) “Is indexing Used for Retrieval?” in *Proceedings: Workshop on Case-Based Reasoning (DARPA), Washington, D.C. San Mateo, CA: Morgan Kaufmann.*

Way, E. C. (1991) Knowledge Representation and Metaphor, Kluwer Academic Publishers, Boston.

Woods (1991), “Understanding Subsumption and Taxonomy: A Framework for Progress” in *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, Sowa, J. F. (ed), Kauffman, San Mateo, pages 45-94.

Woodbury, R., Chang, T.W., Chiou, S.C., Coyne, R., Fenves, S., Flemming, U., and Gomez, N. (1994) SEED Config Requirements, (url: http://seed.edrc.cmu.edu/SC/requirements-new/SC-req-new.book.html, 1999).

Appendix A: Using SEED-KBC and SEED-CBD APIs

This section contains the sample code segments and requirements for incorporating the SEED-KBC and SEED-CBD (both Java and C-APIs) within a host application.

1. SEED-KBC

The classification functionality provided by the SEED-KBC engine requires several changes in terms of the creation and modification of various environment variables. This section identifies these changes and provides information about the file locations for the Java and C APIs.

1.1. Environment variable

The following environment variable should be set prior to using the classification engine. This path is necessary for the UNISQL database to access the schema specific method implementations.

```
KBCLITE:  /usr/users/zeyno/seed_kbc/dbmethods
```

1.2. Database

The SEED-KBC schema and methods are defined in the database named **KBCLite**. The classification schema can be browsed using any UNISQL interface. For instance, the following command-line can be used to access KBCLite via the isqlx (interactive-sql):

```
> isqlx KBCLite
```

1.3. C - API

The C-API (*.c and *.h files) for SEED-KBC is located at:

/usr/users/zeyno/seed_kbc/kblite_api.

The SEED-KBC static library *libkbliteapi.a* in the same directory, can be linked to other C/C++ applications.

1.4. Java API

Before using the SEED-KBC Java API, the following path needs to be added to the **CLASSPATH** environment variable.

/usr/users/zeyno/seed_kbc/javaimpl/classes

Additionally, to enable the Java-API's JNI connection (through the shared object file: libkbcapi.so) the following path needs to be added to the **LD_LIBRARY_PATH** environment variable.

/usr/users/zeyno/seed_kbc/middleware/jniimpl

The SEED-KBC API is implemented as the class: **DBWorkspace**. In order to use the Java-API as part of a Java application the following code segment can be used.

```
public class MyJavaApplication extends AnApplication {
            ...
            protected DBWorkspace workspace;
            ...

            public MyJavaApplication() {
            // workspace connection
            workspace = new DBWorkspace("KBCLite");
              ...
            }
            ...
}
```

In SEED-KBC's Java API, sets are represented by Java strings. Below is a sample code segment for processing set String s:

```
String s = null;
...
StringTokenizer t = new StringTokenizer(s, ":");
int nItems = t.countTokens();
for (int i=0; i < nItems; i++) {
            s = t.nextToken();
            ...

}
```

2. SEED-CBD

The case-base functionality provided by the SEED-CBD engine requires several changes in terms of the creation and modification of various environment variables. This section identifies these changes and provides information about the file locations for the Java and C APIs.

2.1. Environment variable

The following environment variable should be set prior to using the case-base engine. This path is necessary for the UNISQL database to access the schema specific method implementations.

SEEDCBD: /usr/users/zeyno/seed_cbd/dbmethods

2.2. Database

The SEED-CBD schema and methods are defined in the database named **CBD-Seed**. The case-base schema can be browsed using any UNISQL interface. For instance, the following command-line can be used to access CBDSeed via the isqlx (interactive-sql):

```
> isqlx CBDSeed
```

2.3. C - API

The C-API (*.c and *.h files) for SEED-CBD is located at:

/usr/users/zeyno/seed_cbd/cbdapi.

The SEED-CBD static library *libcbdapi.a* in the same directory, can be linked to other C/C++ applications.

2.4. Java API

Before using the SEED-CBD Java API, the following path needs to be added to the **CLASSPATH** environment variable.

/usr/users/zeyno/seed_cbd/javaimpl/classes

Additionally, to enable the Java-API's JNI connection (through the shared object file: libcbdapi.so) the following path needs to be added to the **LD_LIBRARY_PATH** environment variable.

/usr/users/zeyno/seed_cbd/javaimpl/jnimpl

The SEED-CBD API is implemented as the class: **CBWorkspace**. In order to use the Java-API as part of a Java application the following code segment can be used. Similar to SEED-KBC, sets are represented by Java strings in SEED-CBD's Java API[1].

```
public class MyJavaApplication extends AnApplication {
            ...
            protected CBWorkspace workspace;
            ...

            public MyJavaApplication() {
            // workspace connection
            workspace = new CBWorkspace("CBDSeed");
              ...
            }
            ...
}
```

1. Refer to Section 1.4 for a sample Java code for string-set manipulation.

Appendix B: SEED-KBC API specifications

Based on the software requirements and SPROUT's system architecture, SEED-KBC implements classification capabilities as application programming interfaces in C and Java. This section contains the specifications for the SEED-KBC Java API.

1. Constant values

The following are the type constants employed by the SEED-KBC API.

Primitive creation constants	
	PRIMITIVE
	DISJOINT_PRIMITIVES
Retrieval constants	
	SUBSUMEE
	SUBSUMER
	EQUIVALENT
Update constants	
	SUPERS
	PRIMITIVES
	RESTRICTIONS

2. Return values

The following are the constants representing values returned by the SEED-KBC API methods.

Error status	
	KBAPI_OK
	KBAPI_ERROR
Boolean constants	
	KBAPI_TRUE
	KBAPI_FALSE
Conflict constants	
	INHERITANCE_CONFLICT
	DISJOINED_PRIMITIVES
	RESTRICTION_CONFLICT
Comparison constants	
	EQUAL
	EQUIVALENT
	SUBSUMEE
	SUBSUMER
	DISJOINED
	DISTINCT
String	
	Return value *String* can denote a string or a set of strings concatenated in to one string with ':'. Java class StringTokenizer can be used to identify set members (refer to Appendix A: Using SEED-KBC and SEED-CBD APIs for a sample code segment processing a Java set string).

3. KB class methods

SEED-KBC maintains multiple classification knowledge-bases that are represented via distinct kb instances. These instances are all instantiated from a kb class which holds the generic definition of a classification knowledge-base. The following methods access this generic definition instead of a specific kb instance.

getKBNames()	**String getKBNames (void)**
Arguments	none

Return value	A non-null string if there is no error
Description	This function finds the kb instances that currently exist in the classification knowledge-base and returns their names.

createKB() **int createKB (String kbname)**

Arguments	a string denoting a name which uniquely identifies the new kb instance.
Return value	an error status
Description	This function instantiates a new kb instance.

4. KB instance methods

The following methods are called on a specific kb instance to maintain and query its components: primitives, host types, classifications and host objects (individuals).

discardKB() **int discardKB (String kbname)**

Arguments	a string denoting a name which uniquely identifies a kb instance
Return value	an error status
Description	This function discards the specified kb instance along with the primitives, host types, host individuals and classifications associated with it.

cleanupKB() **int cleanupKB (String kbName)**

Arguments	a string denoting a name which uniquely identifies a kb instance
Return value	an error status
Description	This function discards the primitives, host types, host individuals and classifications associated with the specified kb instance.

compare()

int compare (String kbName, String sSet, String primSet, String restrSet, String tdName)

Arguments name of a kb instance, a string denoting a set of classification names, a string denoting a set of primitive names, a string denoting a set of host type names, and a name of a classification.

Return value a comparison type a conflict type or KBAPI_ERROR

Description This function creates and normalizes a *temporary description* which inherits from the specified classifications and contains the specified primitives and restrictions. The temporary description is then compared to the specified classification.

The result of the comparison is DISJOINED if the temporary description and the derived description of the specified classification are disjoined.

The result of the comparison is EQUIVALENT if the temporary description matches in content the derived description of the specified classification.

The result of the comparison is SUBSUMEE if the temporary description is subsumed by the derived description of the specified classification.

The result of the comparison is SUBSUMER if the temporary description subsumes the derived description of the specified classification.

The result of the comparison is DISTINCT otherwise.

If a conflict is detected during normalization of the temporary classification, the function will be aborted and a conflict type will be returned as result.

retrieve()

String retrieve (String kbName, int retrievalType, String sSet, String primSet, String restrSet)

Arguments name of a kb instance, an int constant denoting the type of retrieval, a string denoting a set of classification names, a string denoting a set of primitive names, and a string denoting a set of host type names

Return value a string denoting the names of the requested classifications or null in case of error

Description This function creates and normalizes a temporary description which inherits from the specified classifications and contains the specified primitives and restrictions. The temporary description is then used to retrieve the requested classifications in the specified kb instance.

If the retrieval flag is SUBSUMEE, the result of the comparison is a set of names of classifications subsumed by the temporary description.

If the retrieval flag is SUBSUMER, the result of the comparison is a set of names of classifications that subsume the temporary description.

If the retrieval flag is EQUIVALENT, the result of the comparison is a set of names of classifications with derived descriptions matching the temporary description in content.

If a conflict is detected during normalization of the temporary classification, the function will be aborted and null will be returned as result.

getClassifiedSpobjs() **String getClassifiedSpobjs (String kbName, String sSet, String primSet, String restrSet)**

Arguments name of a kb instance, a string denoting a set of classification names, a string denoting a set of primitive names, and a string denoting a set of host type names

Return value a string denoting the unique identifiers of host individuals or null in case of error

Description This function creates and normalizes a temporary description which inherits from the specified classifications, and contains the specified primitives and restrictions. The temporary description is then used to retrieve the host individuals having associated classifications that are equivalent or subsumed by the temporary description.

If a conflict is detected during normalization of the temporary classification, the function will be aborted and null will be returned as result.

createPrimitive()	**int createPrimitive (String kbName, String primName, String superName, int primType)**
Arguments	name of a kb instance in which the primitive will be created, a string denoting a name which uniquely identifies the new primitive, name of the super primitive or null if the primitive is a top primitive, and an integer primitive type constant
Return value	an error status
Description	This function creates a simple or disjoint primitive and places it in the primitive hierarchy under the specified super primitive.
IsADisjointPrimitive()	**int IsADisjointPrimitive (String kbName, String primName)**
Arguments	name of a kb instance which contains the specified primitive and the primitive’s name
Return value	a boolean type or KBAPI_ERROR
Description	This function finds out whether the specified primitive is a disjoint primitive.
IsADisjunctPrimitive()	**int IsADisjunctPrimitive (String kbName, String primName)**
Arguments	name of a kb instance which contains the specified primitive and the primitive’s name
Return value	a boolean type or KBAPI_ERROR
Description	This function finds out whether the specified primitive has a disjoint primitive ancestor.
AreDisjoinedPrimitives()	**int AreDisjoinedPrimitives (String kbName, String primName1, String primName2)**
Arguments	name of a kb instance which contains the specified primitive and the names of the primitives to be compared
Return value	a boolean type or KBAPI_ERROR
Description	This function finds out whether the specified primitives are disjoined from each other.

getSuperPrimitive()	**String getSuperPrimitive (String kbName, String primName)**
Arguments	name of a kb instance which contains the specified primitive and the primitive's name
Return value	a string denoting the name of a primitive or null in case of error
Description	This function returns the name of the super primitive or an empty string if the specified primitive is a top primitive.
getSubPrimitives()	**String getSubPrimitives (String kbName, String primName)**
Arguments	name of a kb instance which contains the specified primitive and the primitive's name
Return value	a string denoting primitive names or null in case of error
Description	This function returns the names of the sub primitives of the specified primitive.
getPrimitives()	**String getSubPrimitives (String kbName)**
Arguments	name of a kb instance
Return value	a string denoting primitive names or null in case of error
Description	This function returns the names of all the primitives that belong to the specified kb instance.
createHostConcept()	**int createHostConcept (String kbName, String hcName)**
Arguments	name of a kb instance in which the host type will be created, and a string denoting a name which uniquely identifies the new host type
Return value	an error status
Description	This function creates a simple or disjoint primitive and places it in the primitive hierarchy under the specified super primitive.

getHostConcepts()	**String getHostConcepts (String kbName)**
Arguments	name of a kb instance
Return value	a string denoting host type names or null in case of error
Description	This function returns the names of all the host types that belong to the specified kb instance.
registerSpobj()	**int registerSpobj (String kbName, String hiName, String hcName)**
Arguments	name of a kb instance in which the host individual will be registered, a string denoting a name which uniquely identifies the new host individual, and name of a host type denoting its type
Return value	an error status
Description	This function registers a host individual of the given type in the specified kb instance.
unregisterSpobj()	**int unregisterSpobj (String kbName, String hiName)**
Arguments	name of a kb instance, and name of a host individual
Return value	an error status
Description	The host individual is removed from the specified kb instance.
classifySpobj()	**int classifySpobj (String kbName, String hiName, String tdName)**
Arguments	name of a kb instance, name of a host individual, and a name denoting an existing classification
Return value	an error status or RESTRICTION_CONFLICT
Description	This function associates a host individual with the specified classification.

IsClassifiedSpobj() **int IsClassifiedSpobj (String kbName, String hiName)**

Arguments	name of a kb instance, and name of a host individual
Return value	a boolean type or KBAPI_ERROR
Description	This function finds out whether a host individual is classified in the specified kb instance.

IsRegisteredSpobj() **int IsRegisteredSpobj (String kbName, String hiName)**

Arguments	name of a kb instance, and name of a host individual
Return value	a boolean type or KBAPI_ERROR
Description	This function finds out whether a host individual is registered in the specified kb instance.

getClassificationSpobj() **String getClassificationSpobj (String kbName, String hiName)**

Arguments	name of a kb instance, and name of a host individual
Return value	a string denoting a classification name or null in case of error
Description	This function returns the classification of the specified host individual.

getSpobjs() **String getSpobjs (String kbName)**

Arguments	name of a kb instance
Return value	a string denoting host individual names or null in case of error
Description	This function returns the names of all the host individuals that belong to the specified kb instance.

createClassification()	**int createClassification (String kbName, String tdName, String superSet, String primSet, String restrSet)**
Arguments	name of a kb instance in which the classification will be created, a string denoting a name which uniquely identifies the new classification, a string denoting a set of classification names, a string denoting a set of primitive names, and a string denoting a set of host type names
Return value	an error status or a conflict type
Description	This function creates and normalizes a told description which inherits from the specified classifications, and contains the specified primitives and restrictions. The told description is then classified to reflect the changes upon acquisition. If a conflict is detected during normalization of the new classification, the function will be aborted and a conflict type will be returned as result.
discardAllClassifications()	**int discardAllClassifications (String kbName)**
Arguments	name of a kb instance
Return value	an error status
Description	This function discards all the classifications in the specified kb instance. All the host individuals are unclassified as a consequence of this function.
addToClassification()	**int addToClassification (String kbName, String tdName, int type, String itemSet)**
Arguments	name of a kb instance, name of the classification to be modified, an constant int denoting the type of the set items, and a string denoting a set of item names
Return value	an error status or a conflict type

Description This function modifies the content of the given classification by adding the items of the specified type. The modified told description is then normalized and the change is propagated to the effected classifications. The told description and effected told descriptions are re-classified to reflect the changes.

If the item type is SUPER, the specified classification is added new inherited parents.

If the item type is PRIMITIVE, the specified classification is added new primitives.

If the item type is RESTRICTION, the specified classification is restricted to additional host types

The associated host individuals are unclassified if their effected by the modification.

If a conflict is detected during normalization of the modified classification, the function will be aborted and a conflict type will be returned as result.

retractFromClassification() **int retractFromClassification (String kbName, String tdName, int type, String itemSet)**

Arguments name of a kb instance, name of the classification to be modified, a constant int denoting the item type, and a string denoting a set of item names

Return value an error status or a conflict type

Description This function modifies the content of the given classification by retracting the items of the specified type. The modified told description is then normalized and the change is propagated to the effected classifications. The told description and effected told descriptions are re-classified to reflect the changes.

If the item type is SUPER, items will be retracted from the classification’s set of inherited parents.

If the item type is PRIMITIVE, items will be retracted from the classification’s set of primitives.

If the item type is RESTRICTION, items will be retracted from the classification’s set of restrictions.

The associated host individuals are unclassified if their effected by the modification.

If a conflict is detected during normalization of the modified classification, the function will be aborted and a conflict type will be returned as result.

discardClassification() **int discardClassification (String kbName, String tdName)**

Arguments	name of a kb instance, and name of the classification to be discarded
Return value	an error status
Description	This function discards the given classification and propagates the change the effected classifications. The effected told descriptions are re-classified to reflect the changes. The associated host individuals are unclassified.

printToldDescription() **int printToldDescription (String kbName, String tdName)**

Arguments	name of a kb instance, and a classification name
Return value	an error status
Description	This function prints the told information about the specified classification to the standard output device.

printDescription() **int printDescription (String kbName, String tdName)**

Arguments	name of a kb instance, and a classification name
Return value	an error status
Description	This function prints the derived information about the specified classification to the standard output device.

getToldSupers() **String getToldSupers (String kbName, String tdName)**

Arguments	name of a kb instance, and a classification name

Return value	a string denoting a set of names or null in case of error
Description	This function returns the names of the parent told descriptions for the specified classification.

getToldPrimitives() **String getToldPrimitives (String kbName, String tdName)**

Arguments	name of a kb instance, and a classification name
Return value	a string denoting a set of names or null in case of error
Description	This function returns the names of the told primitives for the specified classification.

getToldRestrictions() **String getToldRestrictions (String kbName, String tdName)**

Arguments	name of a kb instance, and a classification name
Return value	a string denoting a set of names or null in case of error
Description	This function returns the names of the host types to which the specified classification is told to be restricted.

getDerivedPrimitives() **String getDerivedPrimitives (String kbName, String tdName)**

Arguments	name of a kb instance, and a classification name
Return value	a string denoting a set of names or null in case of error
Description	This function returns the names of the derived primitives for the specified classification.

getDerivedRestrictions() **String getDerivedRestrictions (String kbName, String tdName)**

Arguments	name of a kb instance, and a classification name
Return value	a string denoting a set of names or null in case of error

Description	This function returns the names of the host types to which the specified classification is restricted.

getSubsumers() **String getSubsumers (String kbName, String tdName)**

Arguments	name of a kb instance, and a classification name
Return value	a string denoting a set of names or null in case of error
Description	This function returns the names of the classifications that subsume the specified classification.

getSubsumees() **String getSubsumees (String kbName, String tdName)**

Arguments	name of a kb instance, and a classification name
Return value	a string denoting a set of names or null in case of error
Description	This function returns the names of the classifications that are subsumed by the specified classification.

getSynonyms() **String getSynonyms (String kbName, String tdName)**

Arguments	name of a kb instance, and a classification name
Return value	a string denoting a set of names or null in case of error
Description	This function returns the names of the classifications that are equivalent to the specified classification.

getClassifications() **String getClassifications (String kbName)**

Arguments	name of a kb instance
Return value	a string denoting classification names or null in case of error
Description	This function returns the names of all the classifications that belong to the specified kb instance.

classificationCompare()	**int classificationCompare (String kbName, String tdName1, String tdName)**
Arguments	name of a kb instance, and two strings denoting the names of the classifications to be compared
Return value	a comparison type or KBAPI_ERROR
Description	This function compares the specified classifications: tdName1 and tdName2. The result of the comparison is EQUAL if tdName1 and tdName2 have the same identifier. The result of the comparison is DISJOINED if tdName1 and tdName2 are disjoined. The result of the comparison is EQUIVALENT if tdName1 and tdName2 have the same derived description. The result of the comparison is SUBSUMEE if tdName1 is subsumed by tdName2. The result of the comparison is SUBSUMER if tdName1 is subsumes tdName2. The result of the comparison is DISTINCT otherwise.
getToldClassified()	**String getToldClassified (String kbName, String tdName)**
Arguments	name of a kb instance, and a classification name
Return value	a string denoting a set of host individual names or null in case of error
Description	This function returns the names of the host individual classifications that are told to be classified by the specified classification.
getAllClassified()	**String getAllClassified (String kbName, String tdName)**
Arguments	name of a kb instance, and a classification name
Return value	a string denoting a set of host individual names or null in case of error
Description	This function returns the names of the host individuals that are classified by the specified classification.

5. Database transactions

The following methods are used to handle some of the generic database transactions.

commit()	**void commit (void)**
Arguments	none
Return value	none
Description	This function is called if the changes to the classification knowledge base are needed to be made permanent.

connect()	**void connect (void)**
Arguments	none
Return value	none
Description	This function is called to connect to the classification knowledge base.

disconnect()	**void disconnect (void)**
Arguments	none
Return value	none
Description	This function is called to disconnect from the classification knowledge base.

Appendix C: SEED-CBD API specifications

Based on the software requirements and SPROUT's system architecture, SEED-CBD implements case-base capabilities as application programming interfaces in C and Java. This section contains the specifications for the SEED-CBD API.

1. Constant values

The following are the type constants employed by the SEED-CBD API.

Case-base concepts	
	CBAPI_CASE
	CBAPI_CASE_DESCRIPTOR
	CBAPI_TARGET
	CBAPI_CASE_BASE
	CBAPI_MATCH_OPERATOR
	CBAPI_PROXY_OBJ
Case info container types	
	CASE_PROBLEM
	CASE_SOLUTION
	CASE_CASE_OUTCOME
Boolean types	
	CBAPI_TRUE
	CBAPI_FALSE

2. Return values

The following are the constants representing values returned by the SEED-CBD API methods.

Error status	
	CBAPI_ERROR_VALUE
	CBAPI_OK
String	
	Return value *String* can denote a string or a set of strings concatenated in to one string with ‘:’. Java class StringTokenizer can be used to identify set members (refer to Appendix A: Using SEED-KBC and SEED-CBD APIs for a sample code segment processing a Java set string).

3. CB object

The following method is called to query cb_object: the base class. If the specified case-base name is not valid, the method returns CBAPI_ERROR_VALUE.

existsObject()	**int existsObject (int type, String objname, String cbname)**
Arguments	an integer denoting a case-base concept type, a string denoting the name of the object, and the name of the case-base to which the object belongs.
Return value	a boolean type or CBAPI_ERROR_VALUE.
Description	This function looks at the specified case-base and returns CBAPI_TRUE if it finds the object.

4. CB

The following methods are called to query the cb class and instances. If the specified case-base name is not valid, the methods return CBAPI_ERROR_VALUE.

getCBs()	**String getCBs ()**
Arguments	none

Return value	a string denoting case-base instance names or null in case of error
Description	This function retrieves names of the existing case-base instances.

createCB() **String createCB (String cbname)**

Arguments	a string for the new case-base name
Return value	an error status
Description	This function creates a new case-base instance and assigns it a unique name.

discardCB() **String discardCB (String cbname)**

Arguments	a string denoting an existing case-base instance
Return value	an error status
Description	This function discards the specified case-base instance and all the associated case-base components.

cleanupCB() **String discardCB (String cbname)**

Arguments	a string denoting an existing case-base instance
Return value	an error status
Description	This function discards all the cases, target objects, match operators and proxies that are associated with the specified case-base instance.

5. Case

The following methods are called to query the case class and instances. If the specified case-base name, case object identifiers, or container types are not valid, the methods return CBAPI_ERROR_VALUE.

caseCreate()	**String caseCreate (String cbname, String casename)**
Arguments	a string for the new case name and a string denoting an existing case-base instance
Return value	a string denoting a unique identifier or null in case of error
Description	This function creates an empty case in the specified case-base and assigns it a unique identifier.
getCases()	**String getCases (String cbname)**
Arguments	a string denoting an existing case-base instance
Return value	a string denoting a set of case names or null
Description	This function retrieves the names of the cases associated with the specified case-base instance.
getCaseIDs()	**String getCaseIDs (String cbname)**
Arguments	a string denoting an existing case-base instance
Return value	a string denoting a set of case identifiers or null
Description	This function retrieves the unique identifiers of the cases associated with the specified case-base instance.
caseDiscardAll()	**int caseDiscardAll (String cbname)**
Arguments	a string denoting an existing case-base instance
Return value	an error status
Description	This function discards all the cases associated with the specified case-base instance.

caseDiscard()	**int caseDiscard (String caseid)**
Arguments	a string denoting a unique case identifier
Return value	an error status
Description	This function discards the specified case.
caseGetRank()	**double caseGetRank (String caseid)**
Arguments	a string denoting a unique case identifier
Return value	a double positive rank value between 1 and 0 or a negative value in case of error
Description	This function finds out the rank of the specified case with respect to the most recent retrieval by matching request.
caseSetToldName()	**String caseSetToldName (String caseid, String casename)**
Arguments	a string denoting a unique case identifier, and a string denoting a new case name
Return value	a string denoting a new case identifier or null in case of error
Description	This function renames the specified case and returns the new unique identifier.
caseGetToldName()	**String caseGetToldName (String caseid)**
Arguments	a string denoting a unique case identifier
Return value	a string denoting a case name or null in case of error
Description	This function returns the name of the specified case.
caseAnnotate()	**int caseAnnotate (String caseid, String annotation)**
Arguments	a string denoting a unique case identifier, and a new annotation string

Return value	an error status
Description	This function adds the new annotation to the annotations set of the specified case.

caseDropAnnotation() **int caseDropAnnotation (String caseid, String annotation)**

Arguments	a string denoting a unique case identifier, and an annotation string
Return value	an error status
Description	This function performs a substring match on the specified case annotations, and drops the ones that match the given annotation.

caseFindAnnotation() **String caseFindAnnotation (String caseid, String annotation)**

Arguments	a string denoting a unique case identifier, and an annotation string
Return value	a string denoting a set of annotations or null in case of error
Description	This function performs a substring match on the specified case annotations, and returns the ones that match the given annotation.

caseAddTo() **int caseAddTo (String caseid, int containerType, String proxyid)**

Arguments	a string denoting a unique case identifier, an integer denoting a case info container type, and a string denoting a unique proxy object identifier
Return value	an error status
Description	This function adds a new proxy object reference to the specified case. The new object reference can be added to the problem, solution or the outcome of the case.

caseDropFrom() **int caseDropFrom (String caseid, int containerType, String proxyid)**

Arguments	a string denoting a unique case identifier, an integer denoting a case info container type, and a string denoting a unique proxy object identifier
Return value	an error status
Description	This function removes the specified proxy object reference from the case. The new object reference can be removed from the problem, solution or the outcome of the case.

caseGet() **String caseGet (String caseid, int containerType)**

Arguments	a string denoting a unique case identifier, and an integer denoting a case info container type
Return value	a string denoting a set of proxy object identifiers
Description	This function returns the content of the specified case container i.e. the problem, solution or the outcome of the case.

caseAddMatchable() **int caseAddMatchable (String caseid, String matchableid)**

Arguments	a string denoting a unique case identifier, and a string denoting a unique proxy object identifier
Return value	an error status
Description	This function adds a new proxy object reference to the matchables set of the specified case.

caseDropMatchable() **int caseDropMatchable (String caseid, String matchableid)**

Arguments	a string denoting a unique case identifier, and a string denoting a unique proxy object identifier
Return value	an error status

Description	This function removes the proxy object reference from the matchables set of the specified case.

caseGetMatchables() **String caseGetMatchables (String caseid, String matchabletype)**

Arguments	a string denoting a unique case identifier, and a string denoting a proxy object type
Return value	a string denoting a set of proxy object identifiers
Description	This function returns the matchable set for the specified case. If the matchabletype string is the keyword **all**, proxy object identifiers of all types are returned; if it is a proxy object type signature, only the matchables of the specified type are returned.

caseGetTypeSignatures() **String caseGetTypeSignatures (String caseid)**

Arguments	a string denoting a unique case identifier
Return value	a string denoting a set of proxy object type signatures
Description	This function returns a set of type signatures for the matchables that belong to the specified case.

6. Proxy

The following methods are called to query the proxy class and instances. If the specified case-base name or proxy object identifiers are not valid, the methods return CBAPI_ERROR_VALUE.

proxyRegister() **String proxyRegister (String cbname, String proxyname, String proxytype)**

Arguments	a SPROUT object identifier and type signature, and a string denoting an existing case-base instance
Return value	a string denoting a unique identifier or null in case of error

Description	This function creates a proxy of a SPROUT object in the specified case-base and assigns it a type signature and a unique identifier for the case-base it belongs to.

getProxies() **String getProxies (String cbname)**

Arguments	a string denoting an existing case-base instance
Return value	a string denoting a set of SPROUT object identifiers or null
Description	This function returns the SPROUT object identifiers from the proxy objects that are associated with the specified case-base instance.

getProxyIDs() **String getProxyIDs (String cbname)**

Arguments	a string denoting an existing case-base instance
Return value	a string denoting a set of proxy object identifiers or null
Description	This function returns the unique proxy object identifiers that are associated with the specified case-base instance.

proxyDiscardAll() **int proxyDiscardAll (String cbname)**

Arguments	a string denoting an existing case-base instance
Return value	**Return value**: an error status
Description	This function discards all the proxy objects associated with the specified case-base instance. As a consequence, all the case and target matchables that reference the discarded proxy objects are updated to reflect the changes.

proxyIsRegistered() **int proxyIsRegistered (String cbname, String proxyname)**

Arguments	a string denoting an existing case-base instance and a SPROUT object identifier.

Return value	a boolean type or CBAPI_ERROR_VALUE
Description	This function looks at the specified case-base and returns CBAPI_TRUE if it finds a proxy object for the specified SPROUT object.

proxyUnregister() **int proxyUnregister (String proxyid)**

Arguments	a string denoting a proxy object identifier
Return value	an error status
Description	This function unregisters the proxy objects associated with the specified case-base instance. As a consequence, all the case and target matchables that reference the unregistered proxy object are updated to reflect the changes.

getProxyTypeSignature() **String getProxyTypeSignatures (String pId)**

Arguments	a string denoting an existing case-base instance
Return value	a string denoting a set of proxy object type signatures or null
Description	This function returns the proxy object type signatures that are associated with the specified case-base instance.

getProxyDBOID() **String getProxyTypeSignatures (String pId)**

Arguments	a string denoting an existing case-base instance
Return value	a string denoting a set of proxy object identifiers or null
Description	This function returns the unique proxy object identifiers that are associated with the specified case-base instance.

7. Match operator

The following methods are called to query the match operator class and instances. If the provided case-base name, match operator identifiers, shared object file location, or the implementation function name are not valid, the methods return CBAPI_ERROR_VALUE.

matchOperatorCreate() **String matchOperatorCreate (String moname, String cbname, String cfunction, String solocation, String matchtypesign)**

Arguments	a match operator name, a C implementation function name, a shared object file location, a type signature, and a string denoting an existing case-base instance
Return value	a string denoting a unique identifier or null in case of error
Description	This function creates a match operator instance and adds a class method to the match operator schema. Since the changes are made at the schema level, the implementation function and the shared object file should exist at the time of creation, otherwise the match operator class will be considered undefined. Upon successful completion the match operator instance is assigned a unique identifier for the case-base it belongs to.

getMatchOperators() **String getMatchOperators (String cbname)**

Arguments	a string denoting an existing case-base instance
Return value	a string denoting a set of match operator names or null
Description	This function returns the names of the match operators that are associated with the specified case-base instance.

getMatchOperatorIDs() **String getMatchOperatorIDs (String cbname)**

Arguments	a string denoting an existing case-base instance
Return value	a string denoting a set of match operator identifiers or null

Description This function returns the unique identifiers of the match operators that are associated with the specified case-base instance.

matchOperatorDiscardAll() **int matchOperatorDiscardAll (String cbname)**

Arguments a string denoting an existing case-base instance

Return value an error status

Description This function discards all the match operators associated with the specified case-base instance. As a consequence, all the target matchables that reference the discarded match operators are updated to reflect the changes.

matchOperatorSetName() **String matchOperatorSetName (String moid, String moname)**

Arguments a string denoting a unique match operator identifier, and a string denoting a new match operator name

Return value a string denoting a new match operator identifier or null in case of error

Description This function renames the specified match operator and returns the new unique identifier.

matchOperatorGetName() **String matchOperatorGetName (String moid)**

Arguments a string denoting a unique match operator identifier

Return value a string denoting a match operator name or null in case of error

Description This function returns the name of the specified match operator.

matchOperatorDiscard()	**int matchOperatorDiscard (String moid)**
Arguments	a string denoting a unique match operator identifier
Return value	an error status
Description	This function discards the specified match operator. As a consequence, all the target matchables that reference the discarded match operator are updated to reflect the changes.
matchOperatorSetSoLocation()	**int matchOperatorSetSoLocation (String moid, String solocation)**
Arguments	a string denoting a unique match operator identifier, and a string denoting a new shared object file location
Return value	an error status
Description	This function sets the specified match operator's shared object file location to the provided path string.
matchOperatorGetSoLocation()	**String matchOperatorGetSoLocation (String moid)**
Arguments	a string denoting a unique match operator identifier
Return value	a string denoting a path for a shared object file or null in case of error
Description	This function returns the location of the shared object location file for the specified match operator.
matchOperatorSetMatchableTypeSign()	**int matchOperatorSetMatchableTypeSign (String moid, String matchtypesign)**
Arguments	a string denoting a unique match operator identifier, and a string denoting a type signature
Return value	an error status

Description	This function sets the type signature for the specified match operator to the provided type signature. The match operator compares objects of the specified type.
matchOperatorGetMatchable-TypeSign()	**String matchOperatorGetMatchableType-Sign (String moid)**
Arguments	a string denoting a unique match operator identifier
Return value	a string denoting a type signature or null in case of error
Description	This function returns the required type signa-ture for the specified match operator.
matchOperatorSetImpl()	**int matchOperatorSetImpl (String moid, String moimpl)**
Arguments	a string denoting a unique match operator identifier, and a string denoting an implemen-tation function name
Return value	an error status
Description	This function sets the implementation for the specified match operator to the provided C function name. The function must be accessi-ble from the match operator's shared object file.
matchOperatorGetImpl()	**String matchOperatorGetImpl (String moid)**
Arguments	a string denoting a unique match operator identifier
Return value	a string denoting an implementation function name or null in case of error
Description	This function returns the C function name which implements the match method for the specified match operator.

8. Target

The following methods are called to query the target class and instances. If the provided case-base name, target object identifiers, match operator or proxy object names are not valid, the methods return CBAPI_ERROR_VALUE.

targetCreate()	**String targetCreate (String cbname, String targetname)**
Arguments	a string for the new target name and a string denoting an existing case-base instance
Return value	a string denoting a unique identifier or null in case of error
Description	This function creates an empty target object in the specified case-base and assigns it a unique identifier.
getTargets()	**String getTargets (String cbname)**
Arguments	a string denoting an existing case-base instance
Return value	a string denoting a set of target object names or null
Description	This function retrieves the names of the target objects associated with the specified case-base instance.
getTargetIDs()	**String getTargetIDs (String cbname)**
Arguments	a string denoting an existing case-base instance
Return value	a string denoting a set of target object identifiers or null
Description	This function retrieves the unique target object identifiers associated with the specified case-base instance.
targetDiscardAll()	**int caseDiscardAll (String cbname)**
Arguments	a string denoting an existing case-base instance
Return value	an error status

Description This function discards all the target objects associated with the specified case-base instance.

targetDiscard() **int targetDiscard (String targetid)**

Arguments a string denoting a unique target object identifier

Return value an error status

Description This function discards the specified target object.

targetSetName() **String targetSetName (String targetid, String targetname)**

Arguments a string denoting a unique target object identifier, and a string denoting a new case name

Return value a string denoting a new target identifier or null in case of error

Description This function renames the specified target object and returns the new unique identifier.

targetGetName() **String targetGetName (String targetid)**

Arguments a string denoting a unique target object identifier

Return value a string denoting a target object name or null in case of error

Description This function returns the name of the specified target object.

targetAddMatchable() **int targetAddMatchable (String targetid, String matchablename, String matchopname)**

Arguments a string denoting a unique target object identifier, a match operator name and a proxy object name

Return value an error status

Description	This function adds a new proxy object, match operator reference pair to the matchables set of the specified target object.
targetDropMatchable()	**int targetDropMatchable (String targetid, String matchablename, String matchop-name)**
Arguments	a string denoting a unique target object identifier, a match operator name and a proxy object name
Return value	an error status
Description	This function removes the specified proxy object, match operator reference pair from the target object matchable set.
targetNMatchables()	**int targetNMatchables (String targetid)**
Arguments	a string denoting a unique target object identifier
Return value	an integer count or CBAPI_ERROR_VALUE
Description	This function returns the count of matchables for the specified target object.
targetGetMatchOperatorAt()	**String targetGetMatchOperatorAt (String targetid, int matchableindex)**
Arguments	a string denoting a unique target object identifier, and an integer index value
Return value	a string denoting a match operator identifier or null in case of error
Description	This function retrieves the match operator identifier of a matchable pair located at the specified index. If the index is out of bounds of the target object matchable set, the function returns null.
targetGetMatchableIDAt()	**String targetGetMatchableIDAt (String targetid, int matchableindex)**
Arguments	a string denoting a unique target object identifier, and an integer index value

Return value	a string denoting a proxy object identifier or null in case of error
Description	This function retrieves the proxy object identifier of a matchable pair located at the specified index. If the index is out of bounds of the target object matchable set, the function returns null.

targetGetTypeSignatures()	**String targetGetTypeSignatures (String targetid)**
Arguments	a string denoting a unique target object identifier
Return value	a string denoting a set of proxy object type signatures
Description	This function returns a set of type signatures for the matchables that belong to the specified target object.

9. Retrieval

The following methods are called to retrieve cases in the specified case-base. If the provided case-base name, classification knowledge-base name, classification name, or the target object identifiers are not valid, retrieval methods return CBAPI_ERROR_VALUE.

retrieve()	**String targetGetMatchableIDAt (String targetname, String cbname, String kbname, String relattrname)**
Arguments	a string denoting a target name, name of a existing case-base instance, a classification knowledge-base name and a string denoting a SPROUT object relation attribute name
Return value	a string denoting a sequence of case identifiers or null in case of error

Description This function ranks the cases associated with the specified case-base. The ranking is based on the matchables and match operators specified in the provided target. The knowledge-base name is used to issue subsumption queries on a SEED-KBC knowledge-base. The relation attribute name argument allows the matching engine to perform a transitive subgraph match on the specified relational hierarchy, if not specified a base match is performed on SPROUT objects (for more about *transitive match* and *base match* refer to Flemming et. al. (1996)). Upon the successful completion of the retrieval, cases are ranked based on a value between 0 and 1, and the identifiers of the ranked are returned in descending order.

retrieveByClassification() **String retrieveByClassification (String cbname, String kbname, String clname)**

Arguments a string denoting the name of a existing case-base instance, a classification knowledge-base name and a string denoting a SEED-KBC classification name

Return value a string denoting a set of case identifiers or null in case of error

Description This function retrieves cases which contain descriptor objects having classifications compatible with the target classification. A descriptor object classification is compatible if it is equal, equivalent to, or is subsumed by the target classification. The knowledge-base name is used to issue the necessary subsumption query on a SEED-KBC knowledge-base. Upon successful completion of retrieval, the function returns a set of case identifiers associated with the specified case-base instance.

retrieveByAnnotation() **String retrieveByClassification (String cbname, String matchstring)**

Arguments a string denoting the name of a existing case-base instance, and a match string

Return value	a string denoting a set of case identifiers or null in case of error
Description	This function retrieves cases which contain annotations matching the target annotation. Upon successful completion of a series of substring matches, the function returns a set of case identifiers associated with the specified case-base instance.

10. Database transactions

The following methods are used to handle some of the generic database transactions.

commit() **void commit ( )**

Arguments	none
Return value	none
Description	This function is called to commit the transactions made in the case-base.

connect() **void connect ( )**

Arguments	none
Return value	none
Description	This function is called to connect to the case-base.

disconnect() **void disconnect ( )**

Arguments	none
Return value	none
Description	This function is called to disconnect from the case-base.

Appendix D: Database Representations

This section contains the schema specifications for the SEED-CBD and SEED_KBC engines.

1. SEED-KBC schema

CREATE CLASS kb_object;

CREATE CLASS kb;

CREATE CLASS role_restriction;

CREATE CLASS told_description;

CREATE CLASS kb_role;

CREATE CLASS individual;

CREATE CLASS concept;

CREATE CLASS kb_individual;

CREATE CLASS host_individual;

CREATE CLASS kb_concept;

CREATE CLASS host_concept;

CREATE CLASS description;

CREATE CLASS primitive;

CREATE CLASS tmp_description;

ALTER CLASS kb ADD SUPERCLASS kb_object;

ALTER CLASS role_restriction ADD SUPERCLASS kb_object;

ALTER CLASS told_description ADD SUPERCLASS kb_object;

ALTER CLASS kb_role ADD SUPERCLASS kb_object;

ALTER CLASS individual ADD SUPERCLASS kb_object;

```
ALTER CLASS concept ADD SUPERCLASS kb_object;
ALTER CLASS kb_individual ADD SUPERCLASS individual;
ALTER CLASS host_individual ADD SUPERCLASS individual;
ALTER CLASS kb_concept ADD SUPERCLASS concept;
ALTER CLASS host_concept ADD SUPERCLASS concept;
ALTER CLASS description ADD SUPERCLASS kb_concept;
ALTER CLASS primitive ADD SUPERCLASS kb_concept;
ALTER CLASS classifier ADD SUPERCLASS kb_object;
ALTER CLASS tmp_description ADD SUPERCLASS told_description;
ALTER CLASS kb ADD ATTRIBUTE
    status character(4) DEFAULT 'off ',
    kb_name character varying(1073741823),
    CONSTRAINT "u_kb(kb_name)" UNIQUE(kb_name);
ALTER CLASS kb ADD METHOD
    discard() FUNCTION cl_discard_kb,
    cleanup() FUNCTION cl_cleanup_kb,
    create_primitive() FUNCTION cl_create_primitive_kb,
    create_classification() FUNCTION cl_create_classification_kb,
    discard_classification() FUNCTION cl_discard_classification_kb,
    is_classified() FUNCTION cl_is_classified_kb,   is_registered() FUNCTION
cl_is_registered_kb,
    get_classification() FUNCTION cl_get_classification_kb,
    discard_primitive() FUNCTION cl_discard_primitive_kb,
    is_a_disjoint_primitive() FUNCTION cl_is_a_disjoint_primitive_kb,
    is_a_disjunct() FUNCTION cl_is_a_disjunct_kb,
    get_primitives() FUNCTION cl_get_primitives_kb,
    get_super_primitive() FUNCTION cl_get_super_primitive_kb,
    get_sub_primitives() FUNCTION cl_get_sub_primitives_kb,
    create_host_concept() FUNCTION cl_create_host_concept_kb,
    get_host_concepts() FUNCTION cl_get_host_concepts_kb,
    are_disjoined() FUNCTION cl_are_disjoined_kb,
    register_spobj() FUNCTION cl_register_spobj_kb,
    unregister_spobj() FUNCTION cl_unregister_spobj_kb,
    classify_spobj() FUNCTION cl_classify_spobj_kb,
    print_description() FUNCTION cl_print_description_kb,
    print_told_description() FUNCTION cl_print_told_description_kb,
    get_spobjs() FUNCTION cl_get_spobjs_kb,
    get_classifications() FUNCTION cl_get_classifications_kb,
    discard_all_classifications() FUNCTION cl_discard_all_classifications_kb,
    add_to_classification() FUNCTION cl_add_to_classification_kb,
    retract_from_classification() FUNCTION cl_retract_from_classification_kb,
    get_told_supers() FUNCTION cl_get_told_supers_kb,
    get_told_primitives() FUNCTION cl_get_told_primitives_kb,
    get_told_restrictions() FUNCTION cl_get_told_restrictions_kb,
    get_derived_primitives() FUNCTION cl_get_derived_primitives_kb,
    get_derived_restrictions() FUNCTION cl_get_derived_restrictions_kb,
    get_synonyms() FUNCTION cl_get_synonyms_kb,
    get_all_classified() FUNCTION cl_get_all_classified_kb,
    get_told_classified() FUNCTION cl_get_told_classified_kb,
    get_classification_subsumees() FUNCTION cl_get_classification_subsumees_kb,
    get_classification_subsumers() FUNCTION cl_get_classification_subsumers_kb,
    retrieve() FUNCTION cl_retrieve_kb,
    compare() FUNCTION cl_compare_kb,
    classification_compare() FUNCTION cl_classification_compare_kb,
    get_classified_spobjs() FUNCTION cl_get_classified_spobjs_kb

FILE    '$KBCLITE/kbmethods.so'
;
ALTER CLASS kb ADD METHOD
CLASS  new() FUNCTION cl_new_kb,
CLASS  find_active() FUNCTION cl_active_kb,
CLASS  activate() FUNCTION cl_activate_kb,
CLASS  deactivate() FUNCTION cl_deactivate_kb,
CLASS  find() FUNCTION cl_find_kb,
CLASS  get_kb_names() FUNCTION cl_get_kb_names_kb
;
```

```
ALTER CLASS told_description ADD ATTRIBUTE
    primitives set(character varying(1073741823)),
    restricted_to set(character varying(1073741823)),
    belongs_to character varying(1073741823),
    inherits_from set(character varying(1073741823));
ALTER CLASS told_description ADD METHOD
    add_primitive() FUNCTION cl_add_primitive_told_description,
    drop_primitive() FUNCTION cl_drop_primitive_told_description,
    restrict_to() FUNCTION cl_restrict_to_told_description,
    drop_restriction() FUNCTION cl_drop_restriction_told_description,
    print() FUNCTION cl_print_told_description,
    discard() FUNCTION cl_discard_told_description,
    add_super() FUNCTION cl_add_super_told_description,
    drop_super() FUNCTION cl_drop_super_told_description,
    subsumes_classification() FUNCTION cl_subsumes_classification_told_description,
    is_in_conflict_with() FUNCTION cl_is_in_conflict_with_told_description,
    clone() FUNCTION cl_clone_told_description,
    get_subsumers() FUNCTION cl_get_subsumers_told_description,
    get_subsumees() FUNCTION cl_get_subsumees_told_description,
    is_canonical_owner() FUNCTION cl_is_canonical_owner_told_description

FILE      '$KBCLITE/kbmethods.so'
;
ALTER CLASS told_description ADD METHOD
CLASS  new() FUNCTION cl_new_told_description
;

ALTER CLASS host_individual ADD ATTRIBUTE
    hi_type character varying(1073741823),
    belongs_to character varying(1073741823),
    is_classified_by character varying(1073741823);
ALTER CLASS host_individual ADD METHOD
    discard() FUNCTION cl_discard_host_individual
```

```
FILE      '$KBCLITE/kbmethods.so'
;
ALTER CLASS host_individual ADD METHOD
CLASS  new() FUNCTION cl_new_host_individual
;

ALTER CLASS host_concept ADD ATTRIBUTE
    belongs_to character varying(1073741823);
ALTER CLASS host_concept ADD METHOD
CLASS  new() FUNCTION cl_new_host_concept

FILE      '$KBCLITE/kbmethods.so'
;

ALTER CLASS description ADD ATTRIBUTE
    belongs_to character varying(1073741823),
    primitives set(character varying(1073741823)),
    restrict_to set(character varying(1073741823)),
    subsumees set(description),
    subsumers set(description);
ALTER CLASS description ADD METHOD
    subsumes() FUNCTION cl_subsumes_description,
    add_subsumer() FUNCTION cl_add_subsumer_description,
    add_subsumee() FUNCTION cl_add_subsumee_description,
    add_subsumer_set() FUNCTION cl_add_subsumer_set_description,
    add_subsumee_set() FUNCTION cl_add_subsumee_set_description,
    are_disjoined() FUNCTION cl_are_disjoined_description,
    print() FUNCTION cl_print_description,
    is_disjoined_from() FUNCTION cl_is_disjoined_from_description

FILE      '$KBCLITE/kbmethods.so'
;
```

```
ALTER CLASS description ADD METHOD
CLASS  new() FUNCTION cl_new_description
;

ALTER CLASS primitive ADD METHOD
CLASS  new() FUNCTION cl_new_primitive,
CLASS  discard() FUNCTION cl_discard_primitive,
CLASS  told_name() FUNCTION cl_told_name_primitive,
CLASS  is_a_disjoint_primitive() FUNCTION cl_is_a_disjoint_primitive_primitive,
CLASS  get_incompetibles() FUNCTION cl_get_incompetibles_primitive,
CLASS  belongs_to() FUNCTION cl_belongs_to_primitive,
CLASS  is_a_disjunct() FUNCTION cl_is_a_disjunct_primitive,
CLASS  get_super() FUNCTION cl_get_super_primitive,
CLASS  get_subs() FUNCTION cl_get_subs_primitive,
CLASS  get_n_descendants() FUNCTION cl_get_n_descendants_primitive,
CLASS  exists_in_kb() FUNCTION cl_exists_in_kb_primitive,
CLASS  are_disjoined() FUNCTION cl_are_disjoined_primitive,
CLASS  is_subprimitive() FUNCTION cl_is_subprimitive_primitive,
CLASS  is_superprimitive() FUNCTION cl_is_superprimitive_primitive

FILE      '$KBCLITE/kbmethods.so'
;

ALTER CLASS tmp_description ADD ATTRIBUTE
    classifies set(character varying(1073741823)),
    tmp_name character varying(1073741823),
    CONSTRAINT "u_tmp_description(tmp_name)" UNIQUE(tmp_name);
```

2. SEED-CBD schema

```
CREATE CLASS cb_object;
CREATE CLASS cb;
CREATE CLASS cb_component;
CREATE CLASS proxy_obj;
CREATE CLASS descr;
CREATE CLASS match_operator;
CREATE CLASS target_descr;
CREATE CLASS case_descr;
CREATE CLASS case_obj;
CREATE CLASS target_matchable;

ALTER CLASS cb ADD SUPERCLASS cb_object;
ALTER CLASS cb_component ADD SUPERCLASS cb_object;
ALTER CLASS proxy_obj ADD SUPERCLASS cb_component;
ALTER CLASS descr ADD SUPERCLASS cb_component;
ALTER CLASS match_operator ADD SUPERCLASS cb_component;
ALTER CLASS target_descr ADD SUPERCLASS descr;
ALTER CLASS case_descr ADD SUPERCLASS descr;
ALTER CLASS case_obj ADD SUPERCLASS cb_component;
ALTER CLASS target_matchable ADD SUPERCLASS cb_component;

ALTER CLASS cb_object ADD METHOD
CLASS  obj_exists() FUNCTION cbd_obj_exists_cb_object
FILE      '$SEEDCBD/cbmethods.so'
;

ALTER CLASS cb ADD ATTRIBUTE
    cb_name character varying(1073741823),
    ranking sequence(character varying(1073741823)),
    CONSTRAINT "u_cb(cb_name)" UNIQUE(cb_name);
ALTER CLASS cb ADD METHOD
    discard() FUNCTION cbd_discard_cb,
    cleanup() FUNCTION cbd_cleanup_cb,
    component_id() FUNCTION cbd_component_id_cb,
    unrank() FUNCTION cbd_unrank_cb,
    rank() FUNCTION cbd_rank_cb
FILE      '$SEEDCBD/cbmethods.so'
;
```

```
ALTER CLASS cb ADD METHOD
CLASS  new() FUNCTION cbd_new_cb,
CLASS  get_cbs() FUNCTION cbd_get_cbs_cb,
CLASS  discard_all() FUNCTION cbd_discard_all_cb,
CLASS  find_unique() FUNCTION cbd_find_unique_cb
;

ALTER CLASS cb_component ADD ATTRIBUTE
    belongs_to character varying(1073741823);

ALTER CLASS proxy_obj ADD ATTRIBUTE
    type_signature character varying(1073741823),
    cb_dboid character varying(1073741823),
    dboid character varying(1073741823),
    CONSTRAINT "u_proxy_obj(cb_dboid)" UNIQUE(cb_dboid);
ALTER CLASS proxy_obj ADD METHOD
    unregister_spobj() FUNCTION cbd_unregister_spobj_proxy_obj,
    get_dboid() FUNCTION cbd_get_dboid_proxy_obj,
    get_type_signature() FUNCTION cbd_get_type_signature_proxy_obj
FILE      '$SEEDCBD/cbmethods.so'
;
ALTER CLASS proxy_obj ADD METHOD
CLASS  register_spobj() FUNCTION cbd_register_spobj_proxy_obj,
CLASS  get_proxy_objs() FUNCTION cbd_get_proxy_objs_proxy_obj,
CLASS  discard_all() FUNCTION cbd_discard_all_proxy_obj,
CLASS  find_unique() FUNCTION cbd_find_unique_proxy_obj,
CLASS  get_proxy_obj_ids() FUNCTION cbd_get_proxy_obj_ids_proxy_obj,
CLASS  is_registered() FUNCTION cbd_is_registered_proxy_obj
;

ALTER CLASS match_operator ADD ATTRIBUTE
    so_location character varying(1073741823),
    matchable_type_sign character varying(1073741823),
    match_operator_name character varying(1073741823),
    match_operator_id character varying(1073741823),
    c_function character varying(1073741823),
    CONSTRAINT "u_match_operator(match_operator_id)" UNIQUE(match_operator_id);
ALTER CLASS match_operator ADD METHOD
    discard() FUNCTION cbd_discard_match_operator,
    set_so_location() FUNCTION cbd_set_so_location_match_operator,
    get_so_location() FUNCTION cbd_get_so_location_match_operator,
    set_matchable_type_sign() FUNCTION
cbd_set_matchable_type_sign_match_operator,
    get_matchable_type_sign() FUNCTION
cbd_get_matchable_type_sign_match_operator,
    set_match_operator_name() FUNCTION
cbd_set_match_operator_name_match_operator,
    get_match_operator_name() FUNCTION
cbd_get_match_operator_name_match_operator,
    set_c_function() FUNCTION cbd_set_c_function_match_operator,
    call_operator() FUNCTION cbd_call_operator_match_operator,
    get_c_function() FUNCTION cbd_get_c_function_match_operator
FILE      '$SEEDCBD/cbmethods.so'
;
ALTER CLASS match_operator ADD METHOD
CLASS  new() FUNCTION cbd_new_match_operator,
CLASS  discard_all() FUNCTION cbd_discard_all_match_operator,
CLASS  find_unique() FUNCTION cbd_find_unique_match_operator,
CLASS  get_match_operators() FUNCTION cbd_get_match_operators_match_operator,
CLASS  get_match_operator_ids() FUNCTION
cbd_get_match_operator_ids_match_operator
;

ALTER CLASS target_descr ADD ATTRIBUTE
    matchables set(target_matchable),
    target_name character varying(1073741823),
    target_id character varying(1073741823),
    CONSTRAINT "u_target_descr(target_id)" UNIQUE(target_id);
```

```
ALTER CLASS target_descr ADD METHOD
    discard() FUNCTION cbd_discard_target_descr,
    set_target_name() FUNCTION cbd_set_target_name_target_descr,
    get_target_name() FUNCTION cbd_get_target_name_target_descr,
    add_matchable() FUNCTION cbd_add_matchable_target_descr,
    drop_matchable() FUNCTION cbd_drop_matchable_target_descr,
    get_n_matchables() FUNCTION cbd_get_n_matchables_target_descr,
    get_matchable_id() FUNCTION cbd_get_matchable_id_target_descr,
    get_matchable_operator() FUNCTION cbd_get_matchable_operator_target_descr,
    get_type_signatures() FUNCTION cbd_get_type_signatures_target_descr
FILE    '$SEEDCBD/cbmethods.so'
;
ALTER CLASS target_descr ADD METHOD
CLASS  new() FUNCTION cbd_new_target_descr,
CLASS  discard_all() FUNCTION cbd_discard_all_target_descr,
CLASS  find_unique() FUNCTION cbd_find_unique_target_descr,
CLASS  get_targets() FUNCTION cbd_get_targets_target_descr,
CLASS  get_target_ids() FUNCTION cbd_get_target_ids_target_descr
;

ALTER CLASS case_descr ADD ATTRIBUTE
    parent_case case_obj,
    matchables set(character varying(1073741823));
ALTER CLASS case_descr ADD METHOD
    add_matchable() FUNCTION cbd_add_matchable_case_descr,
    drop_matchable() FUNCTION cbd_drop_matchable_case_descr,
    type_signatures() FUNCTION cbd_type_signatures_case_descr
FILE    '$SEEDCBD/cbmethods.so'
;
ALTER CLASS case_descr ADD METHOD
CLASS  new() FUNCTION cbd_new_case_descr
;
```

```
ALTER CLASS case_obj ADD ATTRIBUTE
    annotations set(character varying(1073741823)),
    case_id character varying(1073741823),
    told_name character varying(1073741823),
    case_index case_descr,
    problem_set set(character varying(1073741823)),
    solution_set set(character varying(1073741823)),
    outcome_set set(character varying(1073741823)),
    ranking double DEFAULT 0,
    CONSTRAINT "u_case_obj(case_id)" UNIQUE(case_id);
ALTER CLASS case_obj ADD METHOD
    add_index_matchable() FUNCTION cbd_add_index_matchable_case_obj,
    drop_index_matchable() FUNCTION cbd_drop_index_matchable_case_obj,
    get_index_matchables() FUNCTION cbd_get_index_matchables_case_obj,
    discard() FUNCTION cbd_discard_case_obj,
    set_told_name() FUNCTION cbd_set_told_name_case_obj,
    get_told_name() FUNCTION cbd_get_told_name_case_obj,
    add_annotation() FUNCTION cbd_add_annotation_case_obj,
    drop_annotation() FUNCTION cbd_drop_annotation_case_obj,
    add_to() FUNCTION cbd_add_to_case_obj,
    drop_from() FUNCTION cbd_drop_from_case_obj,
    get_set_of() FUNCTION cbd_get_set_of_case_obj,
    find_annotations() FUNCTION cbd_find_annotations_case_obj,
    get_type_signatures() FUNCTION cbd_get_type_signatures_case_obj,
    set_rank() FUNCTION cbd_set_rank_case_obj,
    get_rank() FUNCTION cbd_get_rank_case_obj
FILE    '$SEEDCBD/cbmethods.so'
;
ALTER CLASS case_obj ADD METHOD
CLASS  new() FUNCTION cbd_new_case_obj,
CLASS  get_cases() FUNCTION cbd_get_cases_case_obj,
CLASS  get_case_ids() FUNCTION cbd_get_case_ids_case_obj,
CLASS  find_unique() FUNCTION cbd_find_unique_case_obj,
CLASS  discard_all() FUNCTION cbd_discard_all_case_obj,
CLASS  quick_retrieval() FUNCTION cbd_quick_retrieval_case_obj
;

ALTER CLASS target_matchable ADD ATTRIBUTE
      matchable_id character varying(1073741823),
      parent_descriptor target_descr,
      match_operator_id character varying(1073741823);
ALTER CLASS target_matchable ADD METHOD
CLASS  new() FUNCTION cbd_new_target_matchable
FILE      '$SEEDCBD/cbmethods.so'
;

CREATE TRIGGER clear_case_content
  STATUS ACTIVE
  PRIORITY 0.000000
  BEFORE DELETE ON case_obj
  EXECUTE  delete  from case_descr where parent_case.case_id=obj.case_id;

CREATE TRIGGER clear_target_content
  STATUS ACTIVE
  PRIORITY 0.000000
  BEFORE DELETE ON target_descr
  EXECUTE  delete  from target_matchable where
parent_descriptor.target_id=obj.target_id;

CREATE TRIGGER clear_operator_dependant
  STATUS ACTIVE
  PRIORITY 0.000000
  BEFORE DELETE ON match_operator
  EXECUTE  delete  from target_matchable where
match_operator_id=obj.match_operator_id;

CREATE TRIGGER cb_delete_cases
```

```
  STATUS ACTIVE
  PRIORITY 0.000000
  BEFORE DELETE ON cb
  EXECUTE  delete  from case_obj where belongs_to=obj.cb_name;

CREATE TRIGGER cb_delete_targets
  STATUS ACTIVE
  PRIORITY 0.000000
  BEFORE DELETE ON cb
  EXECUTE  delete  from target_descr where belongs_to=obj.cb_name;

CREATE TRIGGER cb_delete_match_operators
  STATUS ACTIVE
  PRIORITY 0.000000
  BEFORE DELETE ON cb
  EXECUTE  delete  from match_operator where belongs_to=obj.cb_name;

CREATE TRIGGER cb_delete_proxies
  STATUS ACTIVE
  PRIORITY 0.000000
  BEFORE DELETE ON cb
  EXECUTE  delete  from proxy_obj where belongs_to=obj.cb_name;

CREATE TRIGGER clear_proxy_dependants
  STATUS ACTIVE
  PRIORITY 0.000000
  BEFORE DELETE ON proxy_obj
  EXECUTE  delete  from target_matchable where matchable_id=obj.dboid and
belongs_to=obj.belongs_to;
```