Acquiring Software Design Schemas: A Machine Learning Perspective

Mehdi T. Harandi and Hing-Yan Lee
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Avenue, Urbana, IL 61801
harandi@cs.uiuc.edu *hingyan@cs.uiuc.edu*

Abstract

In this paper, we describe an approach based on machine learning that acquires software design schemas from design cases of existing applications. An overview of the technique, design representation, and acquisition system, is presented. The paper also addresses issues associated with generalizing common features such as biases. The generalization process is illustrated using an example.

1 Introduction

Research in software reuse has hitherto focused on how software, be it program code, design, or specification, can be applied in similar subsequent situations. Although the notion of reusing software abstractions (also referred to as schemas, templates, and clichés) is attractive for many reasons, including greater reuse potential, ease of understanding, and reduced complexity, it remains a dream unfulfilled. The main obstacle to the practical use of schema-based systems (e.g., [3,4]) lies in the acquisition of schemas. Currently schemas have to be laboriously handcrafted.

Our research seeks to address this problem in the context of a schema-based design system, IDeA [10,11]. Our computational model is centered upon a bottom-up case-based approach [17] to be used by domain analysts. Design cases from existing applications for a design family are examined and analyzed to identify similar features (i.e., objects and operations) across examples.

2 A Machine Learning Approach

Machine learning research is concerned with computational theories of learning and development of learning tools [12]. In particular, similarity-based learning (SBL) techniques [8] provide an attractive tool for supporting the acquisition of design schemas. SBL algorithms operate by comparing externally supplied training examples to find similarities and then produce general rules or procedures from these common features [15]. We adopt a learning technique that uses positive training examples with bounded generalization.

2.1 Acquiring and Improving Schemas

To derive a design schema from design cases, mappings between corresponding features must be established. The object correspondences are obtained using domain knowledge. Where this is unavailable, analogies between objects, detected using background knowledge, are used. Operation correspondences are established using domain knowledge, operation structure knowledge and a similarity measure. Both syntactic and semantic matchings are used. In matching objects, object type knowledge is used where available; otherwise a similarity score based on data types of the attributes of objects is computed to determine the degree of match. The mapping process produces a set of common features that is then abstracted. This abstraction process is the subject matter of this paper.

Automated schema acquisition provides a solution to the problem of initially populating a design schema library. A schema, however, can be improved by refining it with new design cases encountered subsequently, removing any arbitrariness. Depending on the number of available design cases, different strategies can be adopted. One may opt for an *eager* abstraction

0-8186-2605-4/91 $1.00 © 1991 IEEE

```
if (retrieve(NewDesignCase, designSchema) == OK) and (one is chosen)
    then improve-schema
    else if (retrieve(NewDesignCase, designCase) == OK) and (one is chosen)
            then if (case retention threshold exceeded)
                    then derive-schema
                    else store(NewDesignCase)
            else store(NewDesignCase)
```

Figure 1: Top-Level Schema Acquisition/Improvement Algorithm

strategy, deriving a schema whenever two cases are available. Or one may defer schema derivation for as long as possible (the *lazy* abstraction strategy), retaining cases as they are obtained and deriving a schema only when it is needed for design. Each strategy has its own merits and shortcomings. The former has a conservative storage policy, anxious to discard cases for a design family after its schema is derived. However, insufficient cases would have been examined to detect frequency of design pattern occurrences that are useful to schema derivation. The latter strategy is inherently a storage hog. Because schema derivation requires interaction with a domain analyst, attempting to perform derivation when the schema is needed is impractical. The trade-off is likely to lie somewhere in between, possibly near the eager abstraction end i.e., $1 \leq$ case retention threshold $\ll n$ where n is a large number. The primary implication of this for a pure schema-based design approach is that cases coexist with schemas in the design library, resulting in a hybrid approach to software design.

The acquisition/improvement algorithm is sketched in Figure 1. Since there are in general more cases than schemas, schema retrieval is attempted first. The retrieved schemas are ranked and presented to the user to choose an appropriate one, which is then refined using the new design case. When no schema is retrieved or is found suitable, case retrieval is attempted. The retrieved cases are ranked and the one belonging to the same design family as the new design case is then chosen. If the case retention threshold is exceeded, the new design case is used together with other cases for that design family to derive a schema; otherwise it is stored along with other cases for the same design family. If case retrieval fails or no case is deemed suitable, a domain dictionary is created for the new design case (this assumes its design family is not known to the system yet).

2.2 Design Representation

The input to the acquisition system consists of one or more design cases. Each design case consists of two components: a data component and a process component. The data component is represented by an entity-relationship (ER) data model [5], where the entity types correspond to objects. The process component is represented by a set of layered dataflow diagrams (DFDs) [7], in which the transformations correspond to operations. In DFDs, the data stores correspond to the entity types and the dataflows are associated with data types. The representation of cases is the same as schemas, different only in their level of abstraction. Semantic information is associated with ER models and DFDs. For example, entity types are either regular or weak, while attributes have properties such as attribute kind (basic, acquired, or derived), attribute type (key or non-key), and attribute structure (simple or composite).

2.3 System Overview

Figure 2 shows a schematic overview of a design environment incorporating the schema acquisition/improvement subsystem, a knowledge base, an analogy-based retriever and the IDeA design system. The knowledge base has two components: background knowledge in the form of an object type lattice and a data type lattice, and a design library containing domain-specific schemas and their associated refinement and specialization rules. The analogy-based retriever supports both design and schema acquisition/improvement. In the design mode, the user-specified objects are used to fetch schemas and cases using the object type lattice. If this fails, an object similarity-based approach is used whereby attributes of the objects are used to determine the degree of similarity between user-specified objects and those in the object type lattice. In the acquisition/improvement

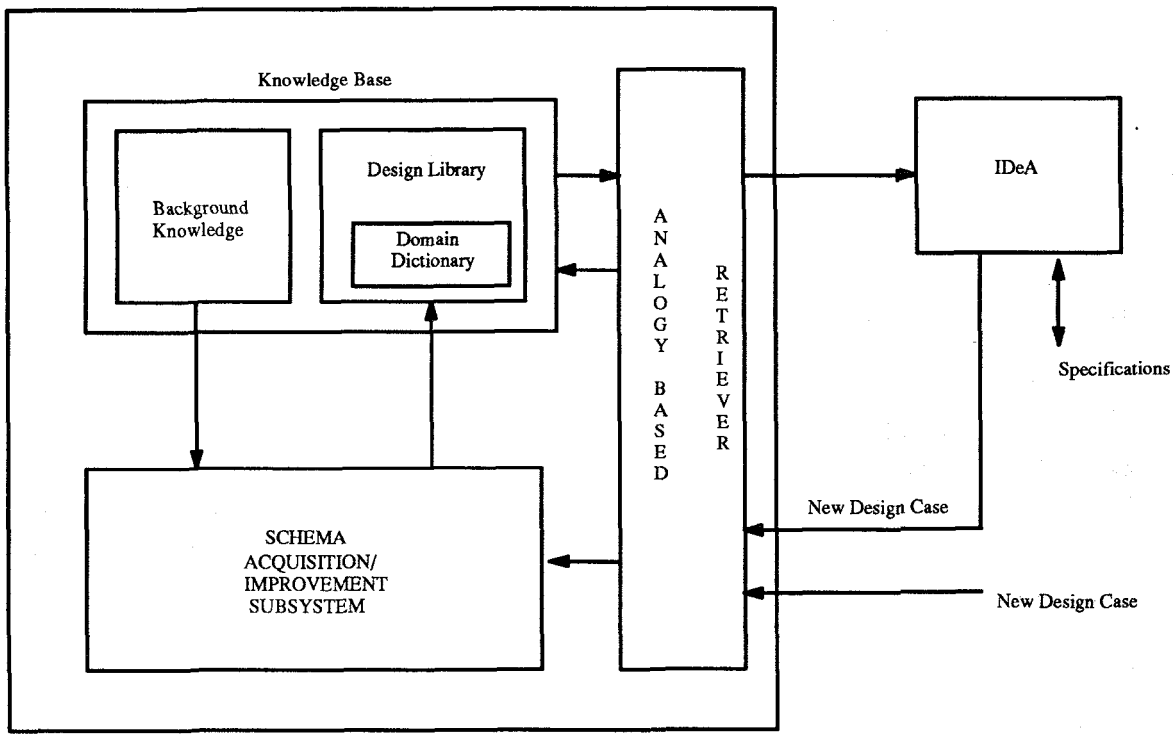

Figure 2: System Overview

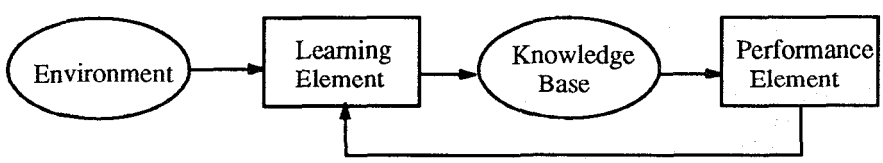

Figure 3: Machine Learning Model

mode, objects from the new case are used in a similar way.

The above system fits the machine learning model (reproduced in Figure 3) discussed in [6], where IDeA and the software designer correspond to the performance element and the schema acquisition/improvement subsystem to the learning element, and the domain analyst plays the role of the environment.

3 Example

Figures 4 and 5 show the process components of design cases for a book library and a video library, respectively. A schema derived from these two designs is depicted in Figure 6. The inset in each figure is a decomposition tree, showing the operations at various levels of abstraction; the root of each tree corresponds to the top-level DFD. The lack of space prevents the inclusion of data models associated with the cases. We will use this example to illustrate several aspects in operation abstraction of schema derivation. These include dropping of non common features, detection of missing abstraction level, detection of overlapping abstraction level, handling of dataflow orphans, and handling of dataflow discontinuities.

4 Abstraction

Abstraction involves generalization, aggregation, classification, and identification [14,16]. The desired result is a succinct description that hides details and focuses on the common features. Design abstractions are useful for helping to manage the intellectual complexity that software design entails. Classification is concerned with how the schemas for design families and subfamilies are organized. As design schemas are added or refined, the respective family and subfamily relationships are modified accordingly. Identification involves providing a system of indices to the objects and operations to facilitate the location of appropriate schemas in the design library. Aggregation refers to associating an aggregate feature with a relationship between features while generalization is concerned with regarding a set of similar features as a generic concept. We discuss only generalization using the library cases introduced earlier.

4.1 Non Common Features

Only the essential features of a design are embodied in a design schema. Essential features are taken to be those features common to the cases encountered so far. Although the frequency of feature occurrence is used as an indication of importance, this is a heuristic approximation. This measure is not perfect since some commonly perceived features may actually be spurious [2]. The ProduceCatalog operation from the video library case has no correspondence in the book library case and is excluded from the schema.

4.2 Adoption of Biases

For cases in a design family, there may be several possible corresponding design schemas. It is hence necessary to introduce *biases* that help determine which schema to prefer. Without the bias, there is no basis to favor one schema over another [13].

4.2.1 Missing Abstraction Level

The leaf nodes of a decomposition represent primitive operations (or P-ops) while the interior nodes represent the abstract operations (or A-ops). An A-op at a given level constitutes an abstraction level with respect to those operations it is decomposed into at the next lower level. A missing abstraction level hence refers to the absence of an A-op in one design case, for which mappings can be found for its constituent operations. The generalization process needs to recognize and detect missing abstraction levels that occur across cases. For example, there is no A-op in the video library case that maps to MaintainBook in the book library case, although the latter's constitutent P-ops, AddBook and RemoveBook, have mappings to AddTape and RemoveTape. Similarly, LoanTape in the video library case has no corresponding A-op in the book library case.

To reconcile this discrepency in the design schema, two alternatives exist; excluding the missing abstraction level or including it. The bias we use is to include the missing A-op. The application of this bias means that the resultant decomposition tree is no taller than the tallest of the input cases. In our example, the library schema incorporates both the unmapped (missing) A-ops, MaintainBook and LoanTape.

4.2.2 Overlapping Abstraction Level

When an operation in one case appears at an abstraction level that is different from the corresponding operation in another case, an overlapping abstraction level

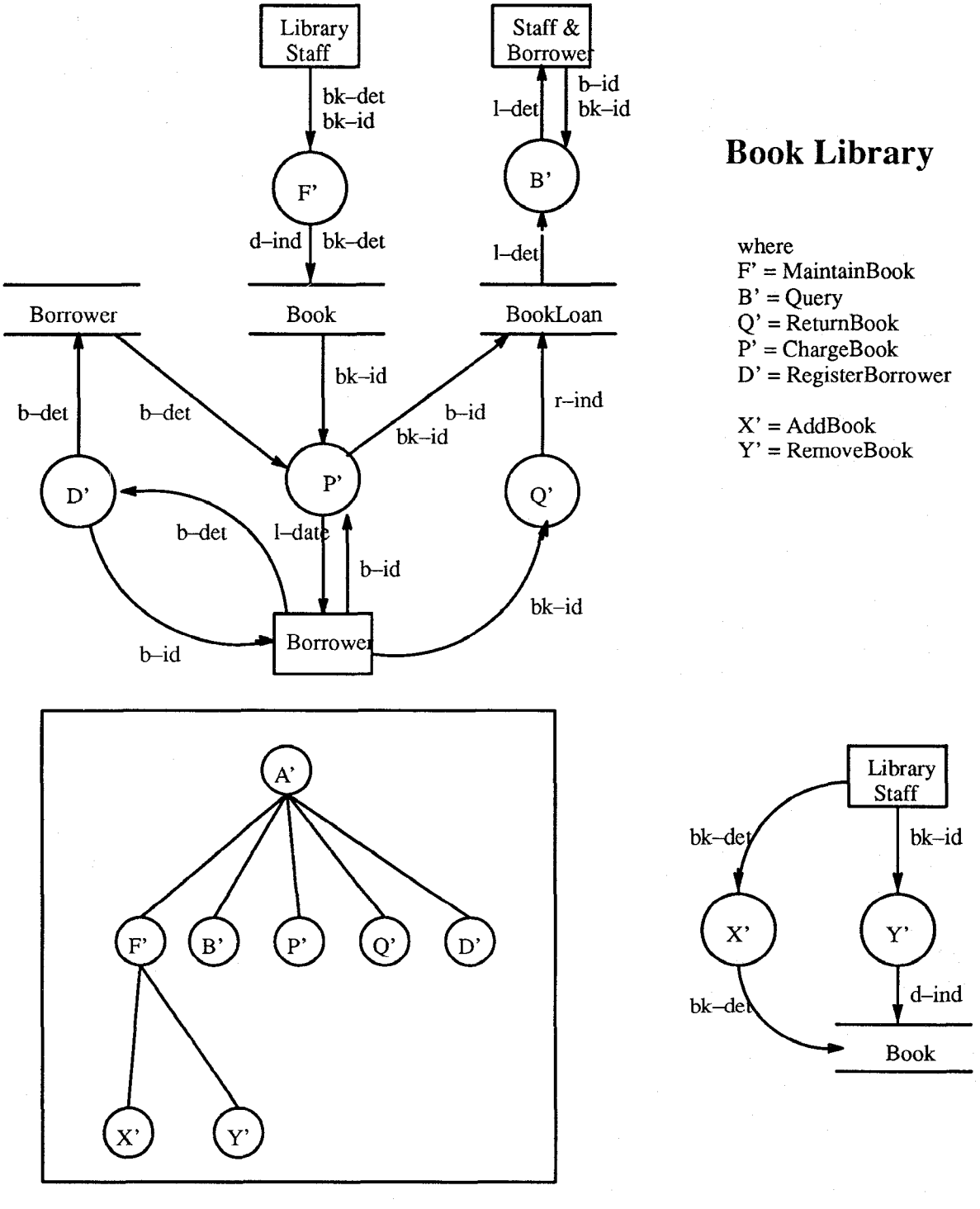

Figure 4: Book Library

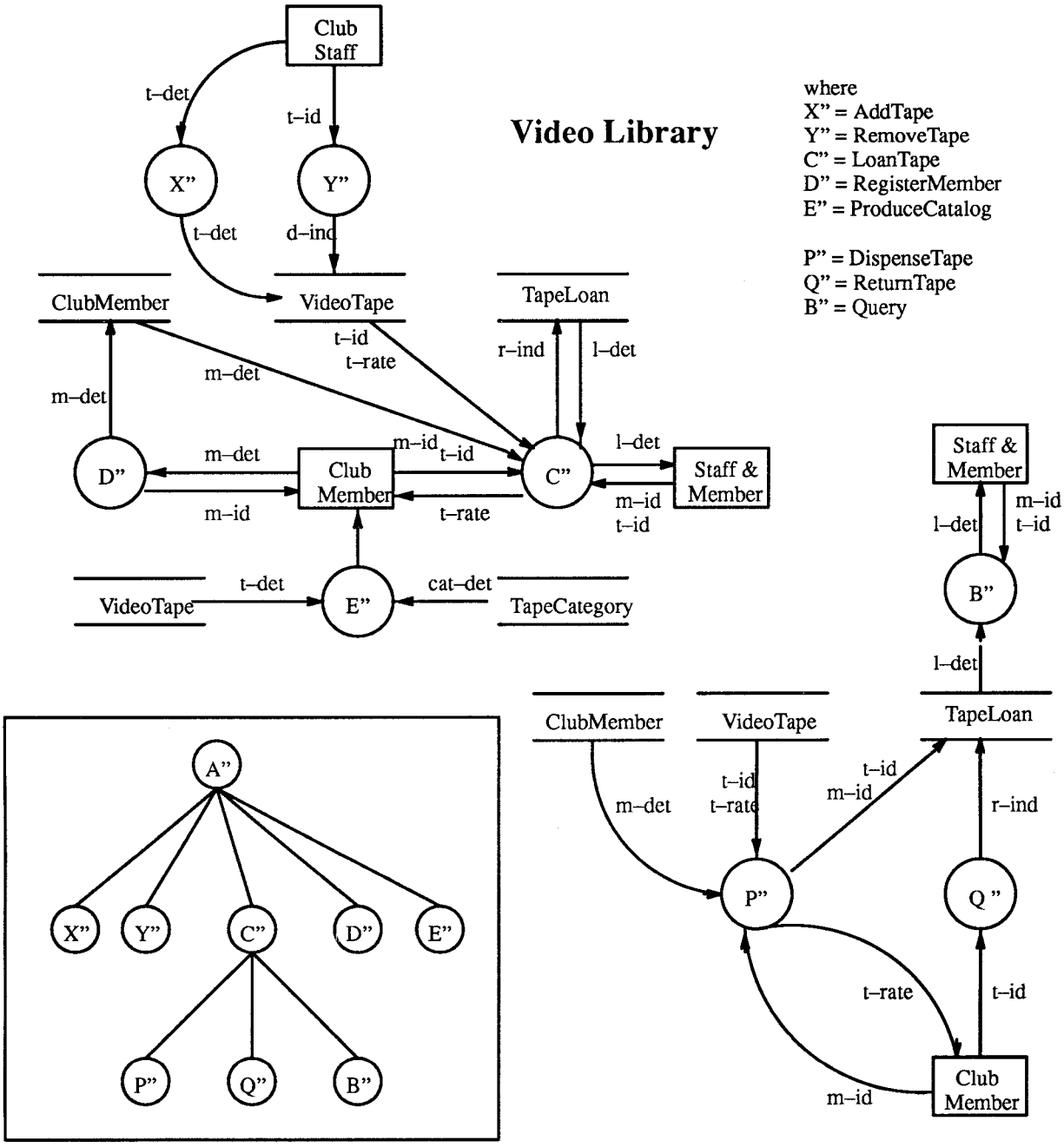

Figure 5: Video Library

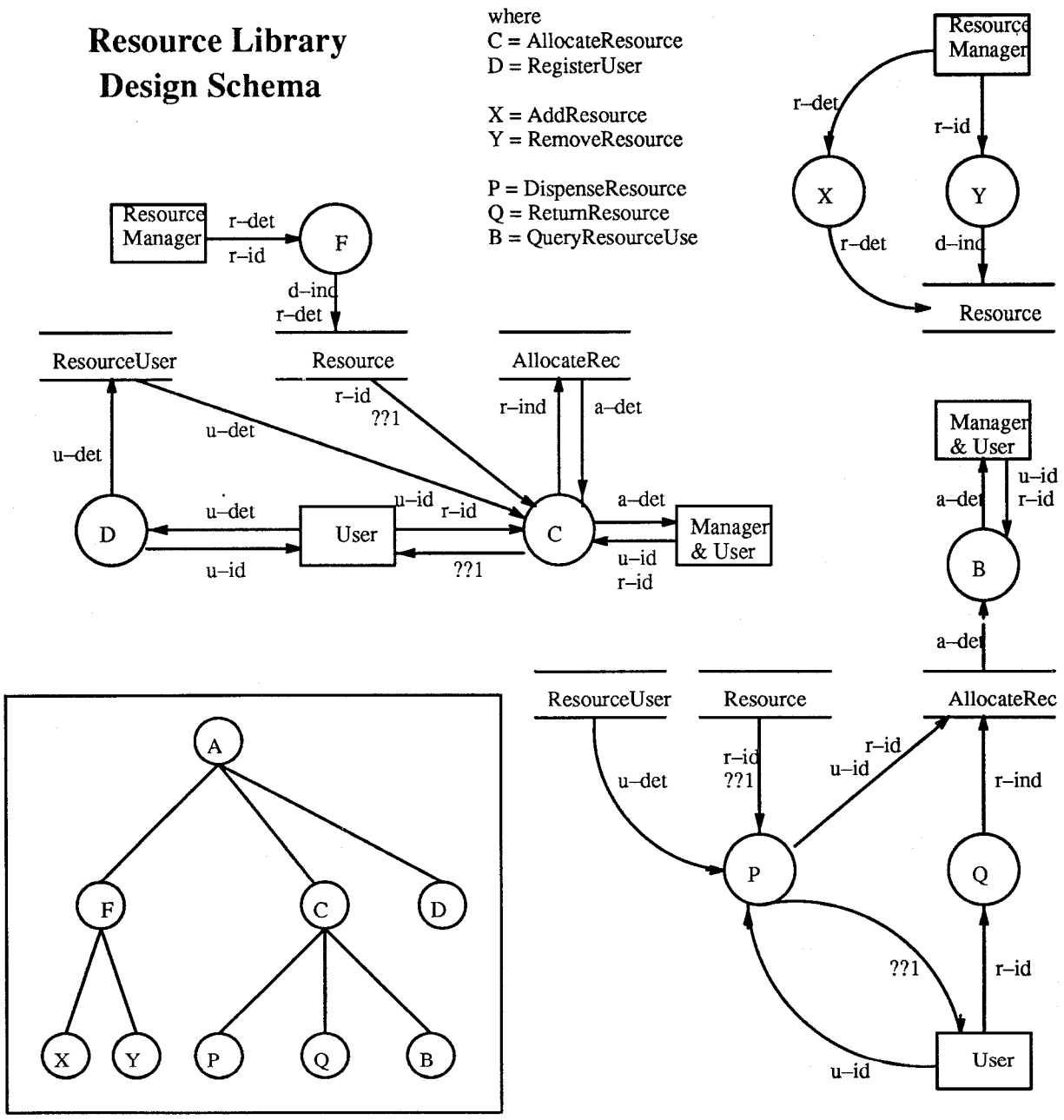

Figure 6: Resource Library Design Schema

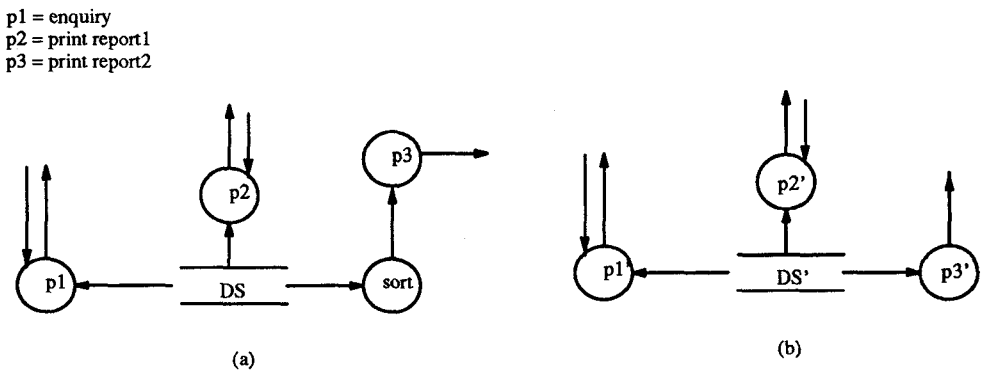

Figure 7: Dataflow Discontinuity

is found. This situation arises for Query (B') in the book library case. Although it has a correspondence in Query (B") in the video library case, they appear at different abstraction levels. Since there appears no real justification to prefer either in the derived schema, we adopt the bias that favors the lower (with respect to the level in the decomposition tree) of the two operations in the schema. In our example, QueryResourceUse (B) in the derived schema is placed in the same relative position as Query (B").

4.3 Abstracting Dataflows

At each abstraction level in a schema, the matching operations and dataflows are generalized. Dataflow generalization is an informed one in that it is guided by object and attribute mappings. It can be accomplished in a straightforward manner, when due attention is paid to dataflow orphans and discontinuities.

4.3.1 Dataflow Orphans

A dataflow orphan is a dataflow in a schema that is generalized from two corresponding dataflows with non-matching data types. In our example, a dataflow orphan (labelled ??1) in the library schema is created as a result of an outgoing dataflow from DispenseResource to User. This dataflow arises because the date_due dataflow from the book library and the rental_rate dataflow from the video library do not have matching data types.

4.3.2 Dataflow Discontinuities

Dataflows for an operation in a schema may suffer from discontinuity when its neighbor operations are not generalized and included. The abstract operation for PrintReport2 (p3 and p3') in Figure 7 has a dataflow discontinuity, because the non common SORT operation in (b) is excluded. The generalization process must recognize that the incoming dataflow is from DS.

5 Related Work

The problem of establishing feature correspondence between cases is similar to that encountered in view integration for database design. While previous methods use inter-view and intra-view assertions, our approach uses domain knowledge, heuristics and similarity measures [9].

The planning system of Anderson and Farley [1], PLANEREUS, creates abstract operators while organizing a given set of primitive ABSTRIPS-type operators into a taxonomic hierarchy and categories of abstract object types. While our goal is similar to theirs, the details differ. The ABSTRIPS-type operators do not have the layered structure DFDs have. PLANEREUS matches operators first, followed by objects. In our domain, knowledge of objects and operations is much richer. We map objects based on available object type knowledge, attributes and structural knowledge. These object mappings then facilitate operations mapping.

In SBL, syntactic rules of inference are used to ac-

complish generalization in predicate calculus. These inductive rules include turning constants to variables, dropping conditions, and adding options [6]. These rules have parallels in our generalization process. The exclusion of non common features from a schema constitutes the dropping conditions rule. The turning of constants to variables rule is analogous to a schema matching more cases than those from which it has been derived. Adding options occurs in the optional dataflows that are present in most but not all the input cases.

6 Summary

A solution to the problem of acquiring schemas for a schema-based software design system is presented. We described a bottom-up case-based schema derivation approach that is amenable to automation using AI techniques. This machine learning attempt at schema acquisition is but a first step toward overcoming the problem of library population in a reusable environment. We also showed how schemas should co-exist with cases in a schema-based system, when it is considered prudent to defer schema derivation until an adequate number of cases can be obtained.

Acknowledgements

This work is supported in part by NASA under Grant NAG-1-613 (ICLASS Project). The second author is supported in part by a National Computer Board (Singapore) postgraduate scholarship.

References

[1] John S. Anderson and Arthur M. Farley. Plan Abstraction Based on Operator Generalization. In *Proc. AAAI Natl. Conf. on Artificial Intelligence*, pages 100 – 104, St. Paul, MN, August 1988.

[2] Ray E. Bareiss. Protos: An Exemplar-Based Learning Apprentice. In *Proc. 4th Intl. Workshop on Machine Learning*, pages 12 – 23, Morgan Kaufmann Publishers, Irvine, CA, June 1987.

[3] Grady H. Campbell Jr. *Abstraction-based Reuse Repositories.* Technical Report 89041-N, Software Productivity Consortium, Herndon, VA, July 1989.

[4] Tom E. Cheatham, G.H. Holloway, and J.A. Tomley. Program Refinement. In *Proc. 5th Intl. Conf. on Software Engineering*, pages 430 – 437, San Diego, CA, 1979.

[5] Peter P.S. Chen. The Entity Relationship Model: Towards a Unified View of Data. *ACM Trans. on Database Systems*, 1(1):9 – 36, 1976.

[6] Paul R. Cohen and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence.* Volume 2, Addison-Wesley, 1982.

[7] Tom DeMarco. *Structured Analysis and System Specification.* Yourdon, Inc., New York, NY, 1979.

[8] Thomas G. Dietterich and Ryszard S. Michalski. A Comparative Review of Selected Methods of Learning from Examples. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann Publishers, Palo Alto, CA, 1983.

[9] Hing-Yan Lee and Mehdi T. Harandi. A Knowledge-based Approach to Correspondence Identification and Conflict Detection in View Integration. In *Proc. 4th Intl. Symposium on Artificial Intelligence*, Cancun, Mexico, November 1991. (To appear).

[10] Mitchell D. Lubars. The IDeA Design Environment. In *Proc. 11th Intl. Conf. on Software Engineering*, pages 23 – 32, Pittsburgh, PA, May 1989.

[11] Mitchell D. Lubars and Mehdi T. Harandi. Knowledge-Based Software Design Using Design Schemas. In *Proc. 9th Intl. Conf. on Software Engineering*, pages 253 – 262, Monterey, CA, March 1987.

[12] Ryszard S. Michalski. Understanding the Nature of Learning: Issues and Research Directions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning II*, pages 311 – 348, Morgan Kaufmann Publishers, Palo Alto, CA, 1987.

[13] Tom M. Mitchell. *The Need for Biases in Learning Generalizations.* Technical Report CBM-TR-117, Dept. of Computer Science, Rutgers University, New Brunswick, NJ, May 1980.

[14] John Mylopoulus. An Overview of Knowledge Representation. *Proc. Workshop on Data Abstraction, Databases and Conceptual Modeling, ACM SIGMOD Record*, 11(2):5 – 14, January 1981.

[15] Jude W. Shavlik and Thomas G. Dietterich, editors. *Readings in Machine Learning.* Morgan Kaufmann Publishers, 1990.

[16] John M. Smith and Diana C.P. Smith. Database Abstraction: Aggregation and Generalization. *ACM Trans. on Database Systems*, 2(2):105 – 133, June 1977.

[17] Will Tracz. RMISE Workshop on Software Reuse Meeting Summary. In Will Tracz, editor, *Tutorial: Software Reuse: Emerging Technology*, pages 41 – 53, IEEE Computer Society, 1988.