VLDB Journal, 4, 1-43 (1995), Arne Solvberg, Editor
©VLDB

The Software Information Base: A Server for Reuse

Panos Constantopoulos, Matthias Jarke, John Mylopoulos, and Yannis Vassiliou

Received February 11, 1992; revised version received, April 2, 1993; accepted December 9, 1993.

Abstract. We present an experimental software repository system that provides organization, storage, management, and access facilities for reusable software components. The system, intended as part of an applications development environment, supports the representation of information about requirements, designs and implementations of software, and offers facilities for visual presentation of the software objects. This article details the features and architecture of the repository system, the technical challenges and the choices made for the system development along with a usage scenario that illustrates its functionality. The system has been developed and evaluated within the context of the ITHACA project, a technology integration/software engineering project sponsored by the European Communities through the ESPRIT program, aimed at developing an integrated reuse-centered application development and support environment based on object-oriented techniques.

Key Words. Information storage and retrieval, conceptual modeling, software engineering, object-oriented databases, reuse.

1. Introduction

Software reuse has grabbed center-stage in international software engineering research, promising to deliver the productivity increase that will eliminate, or at least alleviate, the software crisis. Unfortunately, the path that leads to reuse is not as clearly defined as its promised results. Software libraries, properly populated, are certainly a step in the right direction. So is organizational support and encourage-

---

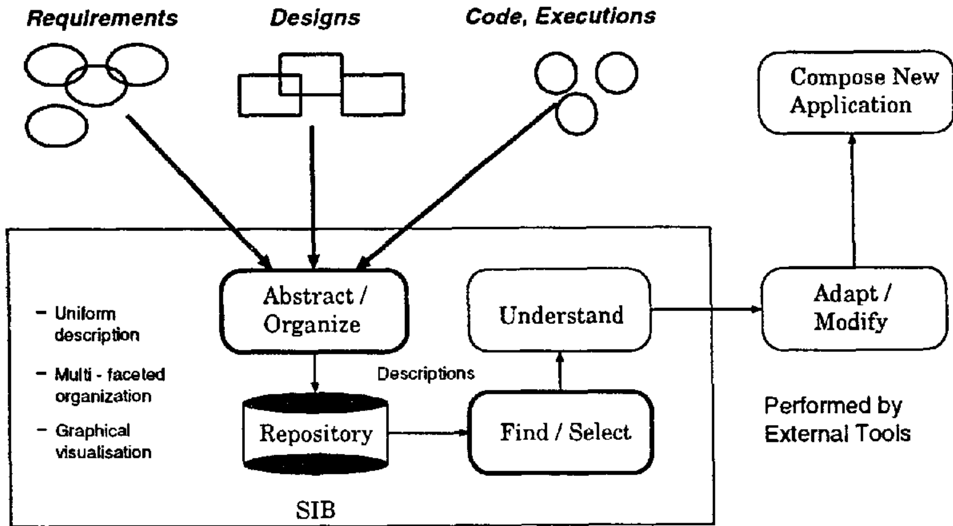
Panos Constantopoulos, Sc.D., is Associate Professor, and Yannis Vassiliou, Ph.D., is Professor, Institute of Computer Science, Foundation of Research and Technology—Hellas, P.O. Box 1385, Heraklion, Crete, 71110 Greece, *panos@ics.forth.gr*, *yannis@ics.forth.gr*; Matthias Jarke, Dr.rer.pol., is Professor, Lehrstuhl Informatik V, RWTH Aachen, Ahronstr. 55, W-51000, Aachen, Germany, *jarke@informatik.rwth-aachen.de*; and John Mylopoulos, Ph.D., is Professor, Department of Computer Science, University of Toronto 6, King's College Road, Toronto, Ontario, Canada M5S 1A4, *jm@cs.toronto.edu*.

ment of reuse, aided by rewards for experience-sharing among software development teams. Object-oriented computing constitutes yet another touted path to the reuse silver bullet. Better understanding of the process of software reuse, supported by appropriate tools is another. So is research that advocates linking software reuse to design reuse in general (as in hardware or architectural design) and developing general AI-based methods such as case-based reasoning and case-based knowledge organization to address it.

Despite the wealth of diverse approaches to reuse, some themes are common. Fundamental among them is the thesis that reuse concerns more than software code. Designs, requirements specifications, and development processes are also reusable and can contribute as much to the legendary productivity increase as the reuse of existing programs. Indeed, software reuse concerns all aspects of the software development experience. Consequently, one can characterize the degree of reuse in terms of a channel of communication between the original developers and the re-users. The broader and better defined the channel, the greater the potential for reuse and, therefore, for productivity improvements. For program libraries, for example, the channel is well defined but narrow, since all development experience other than coding is missing. For experience-sharing meetings between original developer and reuser the channel is broad but ill-defined because it relies on human memory. A major concern of reuse research is the development of methods and tools that broaden and sharpen this channel, by facilitating the recording of the software development experience in all its breadth and richness, and by assisting in its selection and adaptation to new software development tasks. A key component for this is repository technology.

Broad and comprehensive surveys of reuse and the technical challenges it poses have been published (Biggerstaff and Perlis, 1989; Biggerstaff and Richter, 1989; Krueger, 1992). This research addressed problems such as: designing-with-reuse, designing-for-reuse, software artifact classification (characterization), selection/comprehension of reusable objects, and adaptation. Krueger (1992) presented a taxonomy of reuse methods in terms of their ability to abstract, select, specialize (or adapt), and integrate the software artifacts (by composition or combination into a new system). Although repository technology is still immature, there are some major commercial efforts and platform standards, notably the IBM Repository Manager/MVS, Digital's CDD Cohesion, PCTE+ OMS, CAIS, and IRDS (Jobes, 1990; Jones, 1992). There are also a host of less ambitious products, for example ADW/MVS, CASE*Dictionary, DB Excel, or Brownstone (Plotkin, 1992). In research, there are a number of projects of narrower scope that experiment with applications of traditional or object-oriented database technology (Dittrich et al., 1987; Hudson and King, 1989), with hypertext environments (Garg and Scacchi, 1987, 1989; Bigelow, 1988), and with artificial intelligence techniques (Devanbu et al., 1991; Meyer, 1985; Ostertag et al., 1992; Jarke, 1993). Regarding repositories for source code fragments only, much progress has been made with object-oriented class libraries that provide powerful browsers (e.g., Objectworks, SPARCworks Pro-

Figure 1. The reuse process



fessional C++, CodeCenter, Classix, Eiffel, and C++ SoftBench Toolset).

This article presents a software repository system designed for reuse, and for broadening and supporting the communication channel between developer and reuser. The system is intended to store and manage *information about* requirements, designs, and implementations of software and offers facilities for locating and selecting software components. The repository system has been developed within the context of the ITHACA project, a software engineering project sponsored by the European Communities through the ESPRIT program, whose aim is to develop a complete integrated application development and support environment based on object-oriented techniques. The ITHACA environment includes an object-oriented programming language and database service, as well as application development and application support tools. In all aspects of the project, ITHACA adopts a reuse-oriented methodology. Therefore, much of the ITHACA work on languages and tools is focusing on how to make reuse a practical technology. This article gives an account of the structure and implementation of the software repository system, or Software Information Base (SIB) as a means towards reuse.

Assuming a repository-based reuse methodology, key technical challenges (directly related to repository system development) are: providing the right abstraction concepts/mechanisms, carefully organizing, effectively managing, and efficiently selecting and understanding the software artifacts. Figure 1 offers a simplified view of the repository-based reuse process; important functions (e.g., composition of the software artifacts into a new system) are the responsibility of application development tools acting as clients of the repository system.

A number of considerations had to be kept in mind in designing the SIB and its functionality. First, storing information other than code about a software system (e.g., requirements, designs, design decisions, and justifications) means that higher-level software specifications may be reused directly, and that they can serve as indexes to lower-level software artifacts (i.e., code). Second, the issues of representation and presentation of information about reusable artifacts do not have simplistic solutions and need to be addressed separately. Software artifacts should be treated as multimedia objects, which are created and used by distinct application development tools, and must be presented in a format compatible with those tools. A data flow or an entity relationship diagram, for instance, should be suitable to the tool that uses it. Yet, all these drastically different artifacts need to be abstracted and represented in a uniform way within the repository to facilitate the user's conception of the contents of the repository, and understanding of the supporting tools. This calls for a common SIB representation, extensible to accommodate new types of artifacts (e.g., SADT diagrams), and to support multiple presentation forms, depending on the tool using a particular artifact.

The representation language chosen for the SIB is Telos (Mylopoulos et al., 1990), a conceptual modeling language in the family of entity-relationship (ER) models (Chen, 1976), designed specifically for information system development applications. The main reason for the adoption of Telos over other extensions of the ER model (e.g., those used by the IBM Repository Manager/MVS or PCTE+ OMS) is its treatment of attributes as first-class objects, and the treatment of metaclasses, which together lead to a notation that is both expressive and readily extensible. Moreover, Telos has been shown to have a simple and elegant formal semantics in which the data structures and abstraction mechanisms (including extensibility) are specified in terms of a deductive relational database using only a few basic system facts, deduction rules, and integrity constraints (Jeusfeld, 1992). This simplicity offers advantages over existing object-oriented DBMSs, especially when designing multiple related query interfaces—a main feature of the SIB. Specifically, the Telos semantics enables the SIB to offer three integrated query interfaces that can be used by a user at different times, depending on the task the user is working on: a graphical network interface, a form-based interface, and a linear query language (e.g., SQL- or QBE-like) like the ones found in relational databases. The disadvantages of Telos with respect to object-oriented DBMSs (i.e., limited integration of procedural methods for complex update operations) play a lesser role in the SIB context, which is very much search-intensive. From a practical viewpoint, the multi-paradigm interface support offered by Telos is made possible through the use of a powerful graphical editor (Katevenis et al., 1990) and a hypertext engine. Navigation along links, searches through keywords, and structured presentations of the SIB contents are all supported as aids for the SIB user. The adoption of both hypertext and conceptual modeling facilities and their integration into a single user interface is intended to alleviate inherent problems of hypertext systems, such as user disorientation and cognitive overhead.

The SIB organization constitutes one of the keys to its usefulness. All querying facilities (e.g., browsing, filtering, navigating) are supported and facilitated by organization principles, which include the usual general principles of conceptual modeling (i.e., classification, generalization, and aggregation) and are enriched with principles of modularization and semantic similarity. These principles are intended to address both user and methodological needs. For example, there are similarity links because users find it natural to ask for "similar" components. In addition, they support software construction by "scripting" together existing classes (Tsichritzis, 1991) that are shared by ITHACA and other object-oriented viewpoints, where software composition replaces problem decomposition associated with structured programming.

Considering its size, complexity, and required investment, effective management of the SIB is obviously critical. The SIB has been implemented as an efficient and stable prototype. Persistent storage, together with much of the functionality found in relational or object-oriented database systems are realized with an object management component specialized for software repository systems.

A major goal of reuse is, of course, to "... find the software artifacts faster than the time it takes to develop them ..." (Krueger, 1992). Therefore, in addition to providing basic retrieval optimization mechanisms, our approach adopts a multi-paradigm selection strategy, which includes query processing, browsing, filtering, navigational facilities, and approximate retrieval based on similarities among software artifacts. Furthermore, the user interface has been carefully designed to meet the challenge of offering a combination of several retrieval modes in a user-friendly manner. Finally, the SIB system includes built-in facilities which make it possible to associate with any software object annotations and/or animations.

Apart from general considerations concerning the maintenance of any large repository, the context of reuse-based software development introduces a number of additional complications. In particular, there must be a provision for information acquisition, integrity enforcement strategies, version management, and schema evolution. We are only beginning to address these and other issues concerning support for SIB users, starting with the prototype implementation reported here.

Finally, while not addressing issues concerning software artifact understanding (Fichman and Kemerer, 1992), we do claim that the SIB offers valuable assistance to software artifact understanding efforts through the representation and organization of software descriptions.

The remainder of the article is organized as follows. Section 2 elaborates on the SIB structure in terms of descriptions which serve as basic building blocks, along with a number of link types used to organize the contents of the SIB. In Section 3, the SIB system is detailed, with emphasis on the rationale behind the choices made for its development. Section 4 presents an empirical evaluation of the approach in the context of a specific reuse-oriented methodology, that of ITHACA, including a sample usage scenario. Conclusions are drawn and future research directions are outlined in Section 5.

2. Structure of the Software Information Base: The SIB model

In a nutshell, the SIB is structured as a directed attributed graph, with nodes describing software artifacts (objects) and edges representing semantic relationships that hold among them. The software objects themselves are assumed to have their own representation, external to the SIB (e.g. in terms of a UNIX file storing a C program or an *SADT*™ [1] diagram; Ross, 1977), which are accessible from the corresponding SIB description.

The basic building block for the SIB is a *description* that provides information about a software system. This information may concern a requirements, design, or implementation specification for a particular software system. Descriptions may also be used to represent design decisions or run-time performance information about a software object. In addition, descriptions may be atomic, built up from primitives such as programming language expressions, or composite, having other descriptions as parts.

The modeling constructs (types of links) used in the SIB can be divided into four categories:

1. *General structural/semantic relationships,* including attribution, classification, and generalization. These are the basic structuring mechanisms offered by Telos.
2. *Special structural/semantic relationships,* including aggregation, correspondence, genericity, and similarity. These have been identified as a minimal set of system-supplied special software descriptors.
3. *User-defined and informal links,* including versioning and hypertext. The attribute definition facility supported by Telos can be used to extend this set of links.
4. *Associations* are groupings of software artifact descriptions into larger functional units, and are orthogonal to the above binary linking mechanisms. Associations are sets of descriptors along with private symbol tables, which allow for the partition of the SIB into coherent subspaces through the creation (in terms of queries) of materialized views (or snapshots) and of contexts (or workspaces).

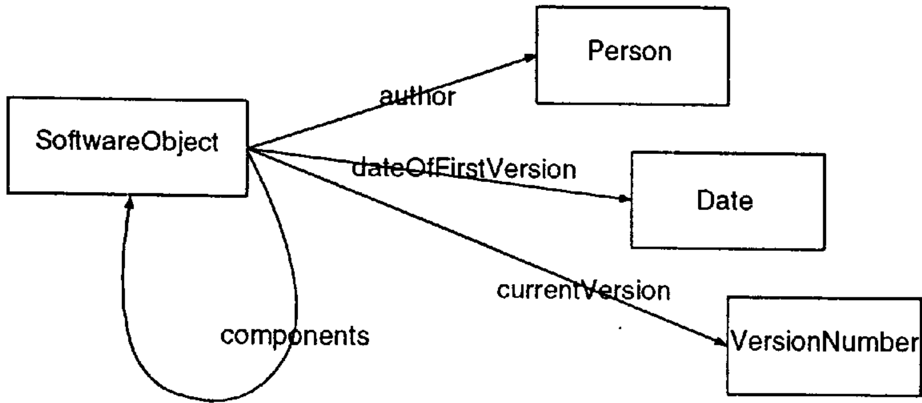
2.1 Attribution and Aggregation

One description can be related to another through a number of built-in semantic relationships. These include attribution and aggregation (part-whole) as in

```
Description SoftwareObject with
    attributes
        author: Person
```

1. SADT is a registered trademark of Softec, Inc.

Figure 2. Attribution and aggregation




```
        dateOfFirstVersion: Date
        currentVersion: VersionNumber
    hasParts
        components: SoftwareObject
end SoftwareObject
```

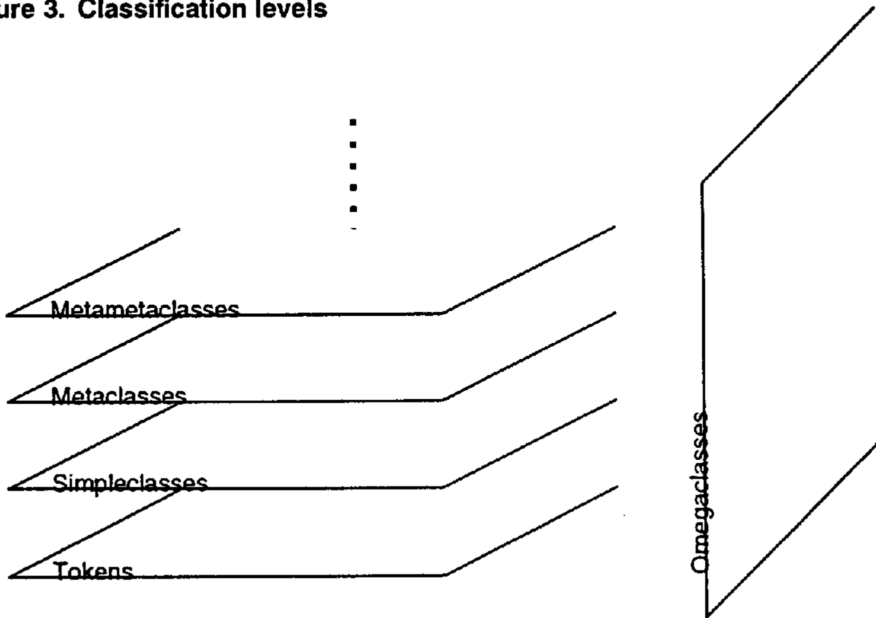
Here *SoftwareObject* is defined as a description having three attributes: `author`, `dateOfFirstVersion` and `currentVersion`, whose 0 or more values have to be instances of the descriptions `Person`, `Date` and `VersionNumber`, respectively. `SoftwareObject` also has 0 or more parts of type `SoftwareObject`. Schematically, this description can be represented in terms of the diagram shown in Figure 2, with boxes representing SIB nodes and arrows representing SIB edges.

Attribution (Mylopoulos et al., 1990), similar to the notion of attribution in Omega (Attardi and Simi, 1981), provides a general and rather unconstrained representation mechanism that describes an object in terms of attribute-value pairs. *Aggregation,* on the other hand, relates an object to its components (Winston and Chaffin, 1987). The components of an object are assumed to be of the same general kind (in our case, software object descriptions). Changes to a component (e.g., through the creation of a new version) imply changes of the aggregate object as well. This property is, in fact, not shared by attributes. It is noted that aggregation axioms imply referential integrity.

2.2 Classification

From a modeling perspective, *classification* (converse: *instantiation*) is perhaps the most important semantic relationship represented in the SIB. Every atomic SIB object

Figure 3. Classification levels

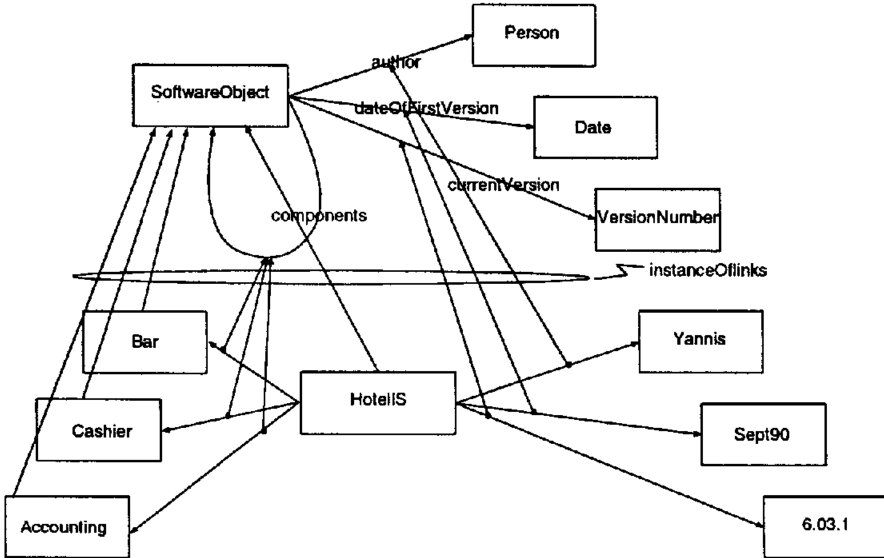


must be an instance of one or more other, generic SIB objects, referred to as classes. Classes are themselves instances of yet more generic classes (called metaclasses) and so on. Most SIB objects are assumed to lie on a unique classification level ranging from 0 for tokens with no instances of their own, to level 1 for simple classes with instances from level 0, to level 2 for metaclasses (or M1 classes) with instances from level 1, and so on. Certain SIB objects that take instances from several levels are known as omega classes and belong to the omega-level. Omega classes are needed to avoid infinite regress in the semantics of the language. From a pragmatic point of view, omega classes help define built-in classes such as Proposition (having all SIB objects as instances) and Class (having all classes as instances), hardcoded into the SIB system, and responsible for elements of the system's operational semantics. Figure 3 shows the structure of the classification dimension.

As with other classification mechanisms described in the literature, instantiation of a class involves instantiation of all associated semantic relationships. For example, instantiation of the `SoftwareObject` class involves defining 0 or more instances of its attributes and parts, as shown in Figure 4. Note that in the adopted representation, semantic relationships are treated as objects in their own right, and are instantiations of relationship classes. Moreover, interrelated objects need not lie on the same classification level. For instance, *HotelIS* may be a simple class (with particular executions of this information system as instances), while `6.03.1`, `Sept90` and `Yannis` are tokens.

Syntactically, the above instantiation is specified in terms of the following:

Figure 4. Classification example




```
Description HotelIS in SoftwareObject with
    author
        : Yannis
    dateOfFirstVersion
        : Sept90
    currentVersion
        : 6.03.1
    components
        : Accounting, Cashier, Bar
        ...
end HotelIS
```

Note that the edges associated with *HotelIS* may have their own labels, as in

```
Description HotelIS in SoftwareObject with
        ...
    components
        acc: Accounting
        cash: Cashier
        bar: Bar
        ...
end HotelIS
```

In the latter definition of *HotelIS,* its components can be accessed by traversing

the edges that are instances of the components link of SoftwareObject, or by traversing the edges labeled acc, cash and bar, respectively.

2.3 Generalization, Genericity and Correspondence

The three link types, generalization, genericity, and correspondence, have been grouped together because they all have definitions that include some kind of inheritance. Thus, a class lower down in one of these hierarchies has fewer possible instances, and its instances are inherited from their more general ancestors. Their differences will become apparent as we discuss them in turn.

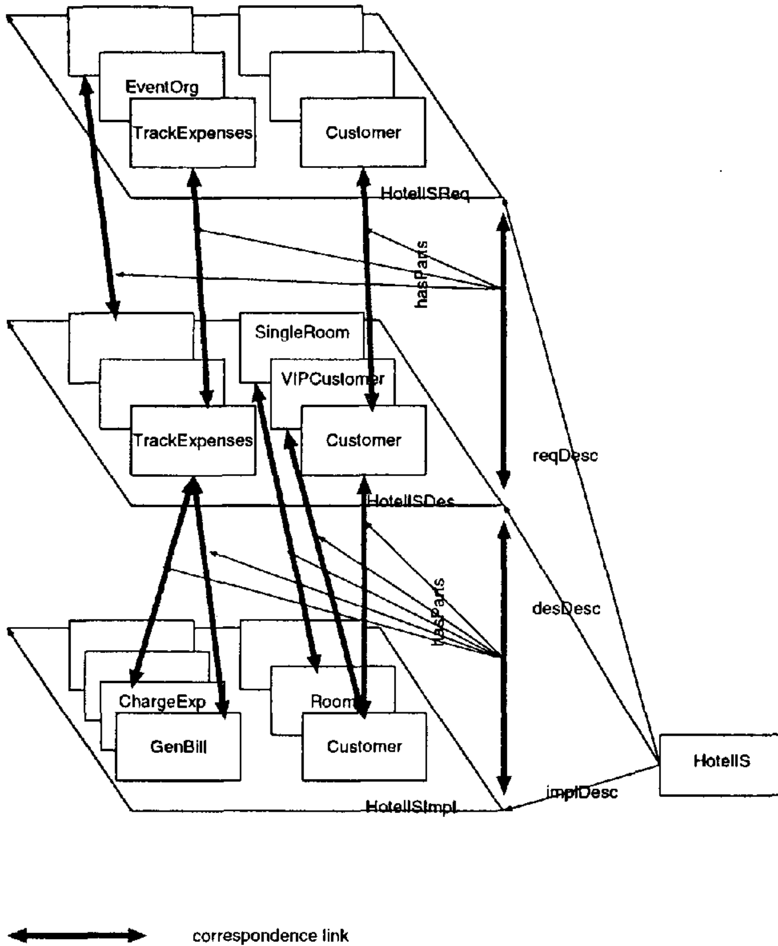
Generalization (converse: *specialization*) has traditionally been supported by semantic and object-oriented data models as well as knowledge representation schemes. The notion of generalization adopted here allows multiple, strict inheritance. For example, the data class StudentEmployee can be declared as a specialization of both classes Student and Employee, thereby inheriting attributes and parts from both classes (multiple inheritance). However, the definition of StudentEmployee cannot override any of the inherited information, it can only further constrain it. For instance, if age has been declared to be an attribute of Student, which has an integer range of 3–60, the age attribute of StudentEmployee can be constrained further to fall in the range 15–45, but cannot be redefined to have a range of, say, 15–70. In programming languages, the term *subtyping* has been used for generalization. Note, however, that when an object is a subtype of another, this does not imply that they share implementation.

Genericity (converse: *specificity*) links related software descriptions, and is intended to convey the sense that one class is an incremental modification of another. A good example of genericity can be found in programming languages where *implementation inheritance* has been used to represent a situation where a software object is more parameterized, and hence has greater genericity, than another with the implication that at code level the two share (some of) their implementation methods. Like generalization, genericity is assumed to be acyclic, transitive, and non-reflexive.

In general, the SIB will include several associated descriptions for a single software object. These may include zero or more versions of a requirements, design, and implementation description. A *correspondence* link represents information concerning the identity of the software system described by two descriptions. In addition, correspondence links can have parts that represent structural correspondences among the components of corresponding descriptions.

In Figure 5, for example, HotelISReq, HotelISDes, and HotelISImpl represent the requirements, design, and implementation descriptions of a hotel information system, described through the *HotelIS* description. Correspondence links indicate that the three descriptions respresent the same system at different levels of abstraction. These links have as parts other correspondence links, which relate the constituent descriptions of HotelISReq, HotelISDes, and HotelISImpl. Note

Figure 5. Correspondence relationships



that the correspondences of the constituents need not be one-to-one. The requirements description, for instance, may contain descriptions of activities taking place in the hotel environment within which the information system will function, such as the organization of an event (e.g., a wedding reception) for which there are no corresponding descriptions in the system design. Likewise, several implementation descriptions, say procedures `ChargeExpense` and `GenBill`, may correspond to a particular design description, say `TrackExpenses`, which is intended to keep track of all expenses associated with a particular client. Conversely, several design descriptions may correspond to a single implementation description. In the example, the design descriptions `Customer` and `VIPCustomer`, which may be part of a generalization hierarchy for customers, correspond to a single implementation description `Customer` (because, for instance, all customer records are stored on a

single database relation). In addition to their parts, correspondence links may have additional associated links that justify and otherwise annotate the correspondence relationship, as in the DAIDA environment (Jarke et al., 1992).

The SIB model emphasizes the use of carefully controlled correspondence hierarchies through the notion of *application frames*. Each application frame includes at least one implementation, and optional design and requirements descriptions:

```
Description ApplicationFrame in Metaclass with
   hasParts
       reqDesc : RequirementsDescription
       desDesc : DesignDescription
   hasParts, atLeastOne
       implDesc : ImplementationDescription
end ApplicationFrame
```

An application frame can be either a *generic application frame* (GAF) or a *specific application frame* (SAF). A SAF describes a particular, complete software system, and includes exactly one implementation (and optional design and requirements descriptions). A GAF, on the other hand, is an abstraction of a collection of applications pertinent to a particular application domain and includes one requirements description (describing the application), one or more designs, and one or more implementations for each of these designs.

```
Description SAF in Metaclass isA ApplicationFrame with
   hasParts, exactlyOne
       implDesc : ImplementationDescription
end SAF
```

```
Description GAF in Metaclass isA ApplicationFrame with
   hasParts, exactlyOne
       reqDesc : RequirementsDescription
   hasParts, atLeastOne
       desDesc : DesignDescription
end GAF
```

GAFs can be viewed as idealized design histories of SAFs, their evolution reflecting the accumulation of experience in deriving SAFs from GAFs and developing them further. Application frames can also be specialized according to the application for which a software system is intended, such as Text Processing or Public Administration.

In summary, generalization/specialization hierarchies (subtyping) "specialize towards a particular domain," (e.g., from administration in general to public administration). The upper classes contain less information than the lower ones, which are typically modeled by adding new attributes to subclasses, or by stronger constraints. These hierarchies concern the "depth" of knowledge covered. Genericity/specificity

hierarchies (incremental modification or parameterization) tend to promote a narrower class of solutions, for example, from generic aggregations of requirements, design, and code to specific ones in which parameters are instantiated—or from sorting procedures where the base to be sorted is a parameter type to ones where the base is an integer type. Finally, correspondence hierarchies tend to promote implementations. Thus, from the point of view of correspondence, an implementation is below a design which, in turn, is below a requirements specification. Vistas along the correspondence hierarchy (application frames) play a major role in the ITHACA methodology.

2.4 Similarity

Similarity links represent similarity relationships with software objects, and provide a foundation for approximate retrieval from the SIB. Similarity has been studied in psychology (Tversky, 1977) and artificial intelligence, most relevantly to this work within the context of case-based reasoning (Barletta, 1991). Similarity has also been offered, within the context of object-oriented systems, as a generalized version of generalization (Wegner, 1987). Its applications include the support of *approximate* retrieval with respect to a software repository as well as the re-engineering of software systems (Schwanke, 1991).

In general, similarity is a relation determined by a flexible comparison between distinct constituents of two entities. Quantitatively, the result of the comparison can be interpreted either as a measure of closeness in some abstract space (Tversky, 1977; Michalski, 1986), or as a probability of the entities resembling each other, even when possibly missing constituents are taken into consideration (Russel, 1988; Esposito et al., 1992). In the present work we adopt the first interpretation. Thus, similarity links are *derived* links, computed with respect to some abstractions, either explicit (represented in the SIB) or implicit (in the user's mind).

Within the SIB, we are primarily interested in similarity with respect to the abstractions (kinds of links) explicitly defined in it, as only such similarity links can be computed automatically. On the other hand, user-defined similarity links are also allowed for flexibility reasons. Similarity is computed with respect to *similarity criteria*, and expressed in terms of corresponding *similarity measures*, which are numbers in the range [0,1]. An aggregate similarity measure with respect to a set of criteria can be obtained as a weighted aggregate function of single-criterion similarity measures, the weights expressing the relative importance of the individual criteria in the set. This measure may be symmetric or directed. For example, similarity with respect to generalization may be defined as symmetric, whereas similarity with respect to type compatibility of the parameters of two C routines may be defined as directed.

Similarity can be used to define task-specific partial orders on the SIB, thus facilitating the search and evaluation of reusable software objects. Moreover, subsets of the SIB can be treated as equivalence classes with respect to a particular symmetric similarity measure, provided all pairs of the class are more similar than a given

threshold. Such similarity-equivalent classes may span different application domains, thus supporting inter-domain reuse.

2.5 User-Defined and Informal Links

Another important link type is that of *derivedFrom* links for version management. As in other version models (Katz, 1990), the version space of a description would be structured as a tree with *derivedFrom* links pointing away from the leaves (the latest versions) and towards the root (the initial version).

A key issue in any version model is the propagation of changes from a description to other related ones through correspondence or *hasParts* links. Configuration management tools such as those described by Rose et al. (1991) will be adopted to address this issue. Another planned extension is the inclusion of links denoting procedure calls within implementation descriptions. Such links can be particularly useful in debugging, software modification and reverse engineering.

In general, if users have foreseeable demands for other link types, they can define them through the mechanisms provided by Telos. For unforeseen representational needs, there is a hypertext link type which makes it possible to attach informal annotations or animations to any SIB object.

2.6 Associations

The SIB described so far can be viewed as a global information base where everything is visible and accessible through a symbol table that contains external identifiers for particular SIB objects. For example, the simple SIB of Figure 4 includes external identifiers `SoftwareObject`, `Person`, `Date`, `VersionNumber`, `HotelIS`, `Accounting`, `Cashier`, `Bar`, `6.03.1`, `Sept90` and `Yannis`. In general, the SIB will also contain descriptions with no external identifiers or with several.

Association is intended to allow the grouping of descriptions that play together a functional role (Brodie and Ridjanovic, 1984). For example, we may define an association as the descriptions that constitute a design specification for a hotel information system, or all the classes that define an implementation of that same system. Note that an association partly addresses the need for encapsulation facilities in conceptual modeling. The contents of an association can only be accessed through the entry points defined in its symbol table. Thus, an association is, actually, a tuple:

```
Association = (setOfDescriptions, symbolTable)
```

The SIB itself is a global association containing all objects. Its symbol table contains all the external identifiers of every object. Name conflicts are resolved by a precedence rule.

Associations can be combined to define new associations. Assuming that the functions `space` and `symTable` access the set of descriptions and the symbol table of an association, respectively, and that the components of entries of the symbol table can be accessed through `identifier` and `range`, we can define:

```
association1 = ( {X|X ∈ space(SIB) and instanceOf(X,
   DesignDescription)}, {{Y|Y ∈ symTable(SIB) and  range(Y) ∈
space(association1)})
```

or,

```
association2 = ( space(SIB) - space(association1),
                 {X| X ∈ symTable(SIB) and
                  identifier(X) ∉ identifier(association1)} )
```

Associations can be represented as special descriptions having the structure:

```
Description Association in OmegaClass with
   attributes
      author: Person;
      importFormula , exportFormula: DerivationFormula
   hasParts
      space: Description;
      symTable: SymbolTable
end Association
```

The *importFormula, exportFormula* components of an association are assumed to be maintained automatically, and keep track of interdependencies in the definitions of associations. In the earlier examples, `association1` imports from the SIB, and exports to `association2`, while `association2` imports from SIB and `association1`.

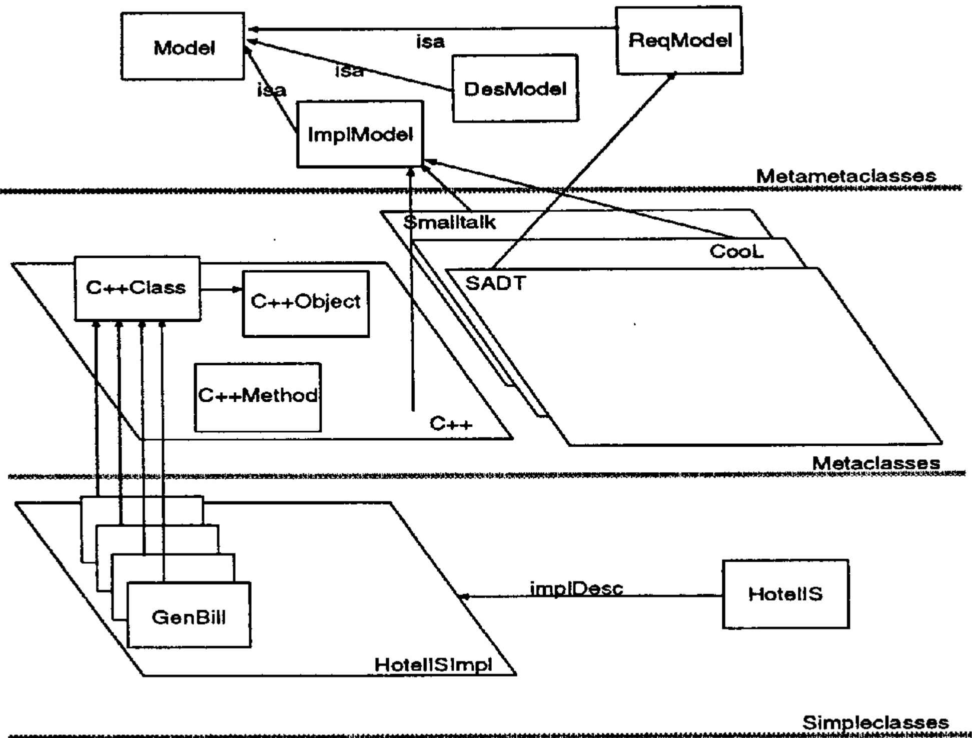
Associations can be considered as materialized views, defined through queries, or through set-theoretic operations from other associations. For pragmatic reasons, the SIB offers another form of modularization, called *views,* where the defined groupings are not materialized. Like their database cousins, and unlike associations, views cannot be updated directly, but only through updates of their `importsFrom` associations.

2.7 The Global SIB Structure

The structure and meaning of each requirement, design, or implementation description depends, of course, on the notation (linear, graphic, or other) used for that description. To accommodate different notations (e.g., SADT or ORM; Pernici, 1990) for requirements; E-R diagrams or some object-oriented notation for design; and C++ or COBOL for implementation requires facilities for modeling the nature of the symbolic structures accommodated by that notation. This is achieved within the SIB through extensive use of the classification dimension.

Figure 6 shows a number of C++ class descriptions, including the class GenBill, which defines the implementation of a hotel information system. For example, `HotelISImpl` is an association, and is part of an application frame named `HotelIS`.

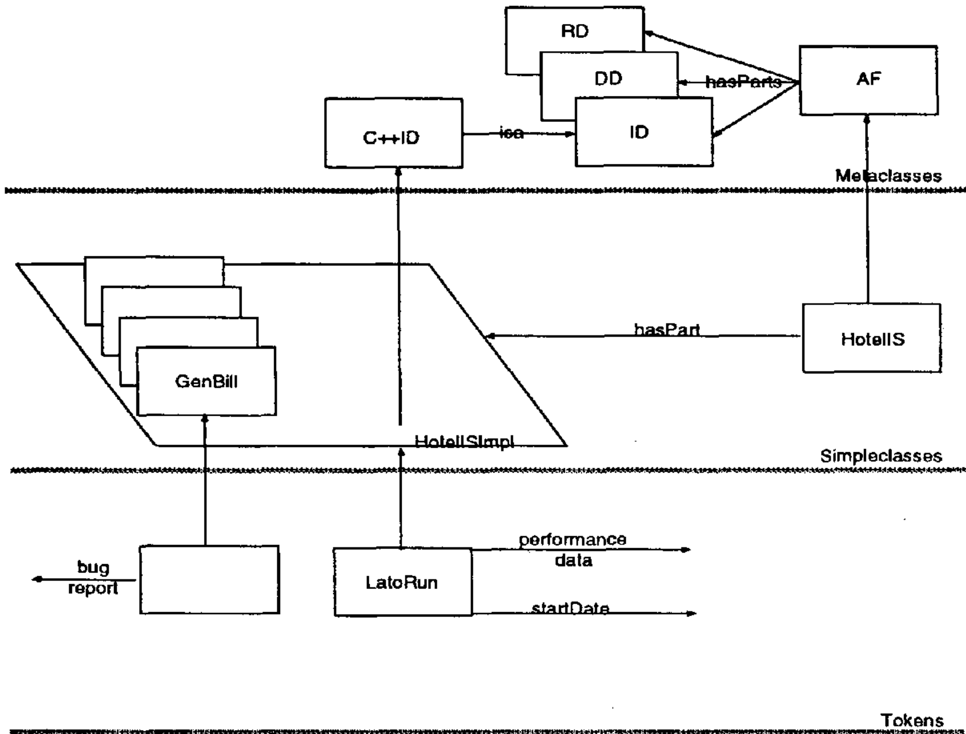
Figure 6. Global SIB structure



These are simple classes within the SIB. The figure also shows some of the metaclasses that might be instantiated during the process of inserting such application frames in the SIB: `C++Class`, `C++Method`, and `C++Object`, whose instantiations populate C++ implementation descriptions such as `HotelISImpl`. All C++ objects are included in the `C++association`, which is an instance of the metametaclass `ImplModel`, along with Smalltalk and CooL (CooL, 1990). SADT, on the other hand, is listed under `ReqModel`. Some models (e.g., the Entity-Relationship model), may be listed under more than one metametaclass model.

The picture of the SIB structure suggested by Figure 6 can now be augmented with that of Figure 7. The token level of the SIB is reserved for information concerning run-time experiences with software described in the SIB. For example, the `HotelISImpl` association is shown in Figure 7 to have been run for `Lato` (presumably, a hotel) with attached information on performance characteristics and bug reports for classes included in the association, such as `GenBill`. The simple class level of the SIB includes program descriptions (mostly declarations), along with descriptions of the designs and requirements from which they were derived. In addition, the simple class level includes more macroscopic units such as associations

Figure 7. SIB structure and application frames



representing implementation, design, or requirements descriptions, and application frames. Finally, the metaclass level includes generic descriptions of application frames as well as requirements, design, and implementation descriptions.

As mentioned earlier, all SIB objects (not just tokens) are subject to modification. However, pragmatic reasons dictate the adoption of different operational rules for objects at different classification levels. In particular, modifications at meta-levels are relatively rare and are under the authority of designated engineers, while at the simple class level application developers actually change the schema by populating the SIB with software descriptions (either manually or through development tools).

3. The SIB Prototype System

This section describes an industrial strength prototype SIB system that has been implemented at the Institute of Computer Science, FORTH. This robust prototype has been functional for some time and has been made available to other sites for experimentation. After an overview, the section presents the system's architecture and user interface, followed by a discussion of its implementation and integration

with other tools within the ITHACA application development environment. A usage scenario is detailed in Section 4 as part of the SIB evaluation.

3.1 System Functionality

The SIB system offers a number of maintenance, selection, and workspace management functions. Maintenance functions include insertion, deletion, and update of information in the SIB, and are supported by appropriate access language and interactive form-based data entry tools. The latter offer a form based on the type of object to be edited by reading out schema information from the SIB itself. Selection functions include retrieval and browsing, while workspace management involves the dynamic definition and modification of workspaces to provide easier and more efficient interaction with the SIB.

3.1.1 General Description. Maintenance and selection operations are performed on *workspaces,* which are application-specific and/or user-specific subsets of the SIB, and are represented as associations. When there are several overlapping workspaces, the identity of shared objects is maintained through reference to the identifiers used within the SIB itself. Classification and generalization links are also shared between workspaces, being an integral part of an object's identity. The use of workspaces offers several benefits, such as focusing attention on selected parts of the SIB by hiding irrelevant information; creating convenient views of the same objects either by renaming them or by hiding particular parts of their descriptions; improving search performance by effectively restricting the search space; and supporting privacy. In terms of the structure introduced in the previous section, workspaces are special cases of associations. The default workspace is the entire SIB (which can be thought of as a global workspace).

Queries to the SIB can be classified from a user's point of view as *explicit* or *implicit.* An explicit query involves an arbitrary predicate explicitly formulated in a query language or through an appropriate form interface. An implicit query, on the other hand, is generated through navigational commands in the browsing mode, or through a button or menu option, for frequently used, predefined queries. Browsing commands and explicit queries can also be issued through appropriate interfaces from external tools.

The selection of software descriptions from the SIB is accomplished through the *Selection Tool* (ST) in terms of an iterative process consisting of retrieval and browsing steps. Browsing is usually the final and sometimes the only step required for selection. The functional difference between the retrieval and the browsing mode is that the former supports the retrieval of an arbitrary subset of the SIB, and presumes some knowledge of the SIB contents, while the latter supports local exploratory searches within a given subset of the SIB without any prior knowledge. Operationally, both selection modes evaluate queries against the SIB.

Browsing in a software development environment is a powerful and required facility, as evidenced from the emphasis on good browsers in almost all available

software class libraries (Korson and McGregor, 1992). Of course, when offered as the only access mechanism in large libraries, browsing has its limitations. On the other hand, when augmented with filters and orientation facilities, and coupled with additional retrieval modes, browsing becomes a very effective access mode.

A few examples of non-Boolean predicates concerning software are: the relevance of a software description to some application, the similarity of two descriptions with respect to some criterion, and the coupling of two pieces of code in a running system. The SIB system is intended to support such non-Boolean queries through tools that order their response by relevance, similarity, affinity, and the like.

3.1.2 Basic Functions. The basic functions of the SIB are as follows:

Maintenance functions:

Insert: Descriptions × Associations → Associations
Delete: Identifiers × Associations → Associations
Update: Descriptions × Associations → Associations
NewVersion: Descriptions × Associations → Associations

The *Insert* function takes as input a description and an association, and inserts that description to the association as well as the global association (SIB). If the description is that of an association, insertion includes materialization of the association. *Delete* takes as argument the identifier of a description and an association, and deletes the description from the association. *Update* modifies a particular version of a description, while *NewVersion* turns the updated description into a new version.

Selection functions:

Retrieve: Queries × Associations → SetOf (Descriptions × Weights)
Browse: Identifiers × SetOf (Links × Depths) × Associations → Views

The *Retrieve* function takes as input an association and a (compound, in general non-Boolean) query, and returns a subset of the association with weights attached indicating the degree to which each description in the answer set matches the query. The prototype implementation only handles Boolean queries, but extensions to handle non-Boolean queries are already under way. Queries are formulated in terms of the query primitives offered by the Programmatic Query Interface. A set of queries of particular significance can be pre-formulated and offered as menu options, thus providing maximum ease-of-use and efficiency for frequent retrieval operations.

Browsing is a special retrieval operation that begins with a particular SIB description, which is the current focus of attention (the *current object*), and produces a view of a *neighborhood* of the current object within a given association. Since the SIB has a network structure, the neighborhood of the current object is defined in terms of incoming and outgoing links of interest. The size of the neighborhood can also be controlled. Thus, the *Browse* function takes as input the identifier (name) of the current object, a list of names of link classes paired with depth control

parameter values, and an association, and then determines a local view centered around the current object.

When the depth control parameters are all equal to 1, a *star view* results, showing the current object at the center surrounded by objects directly connected to it through links of the selected types. In topological terms, this is the simplest and smallest neighborhood of an object, with a controllable population. *Browse* can be called iteratively with argument one of the objects contained in the browser's view, resulting in a new current object and an updated view. Effectively, the *Browse* function provides a moving window with controllable filters and size, which allows navigational search over subsets of the SIB network.

When the depth control parameters are assigned values greater than 1, *Browse* displays all objects connected to the current object via paths consisting of links of the selected types (possibly mixed), where each type of link appears in a path, at most the number of times specified by the corresponding depth parameter. This results in a directed graph that is rooted at the current object. Finally, when the depth parameters are assigned the value ALL (infinite), the transitive closure of one or more link types is displayed with respect to the current object. Such a browse operation can display, for example, the call graph (forward or backward) of a given routine.

The multimedia nature of SIB descriptions calls for the development of a hypermedia annotation mechanism that would gracefully complement the SIB semantic network. This is accomplished by establishing referential links between descriptions, treated as a special category of attribute links, thereby integrating them within the SIB network model. Hypermedia annotations include text, graphics, raster images, and algorithm animations.

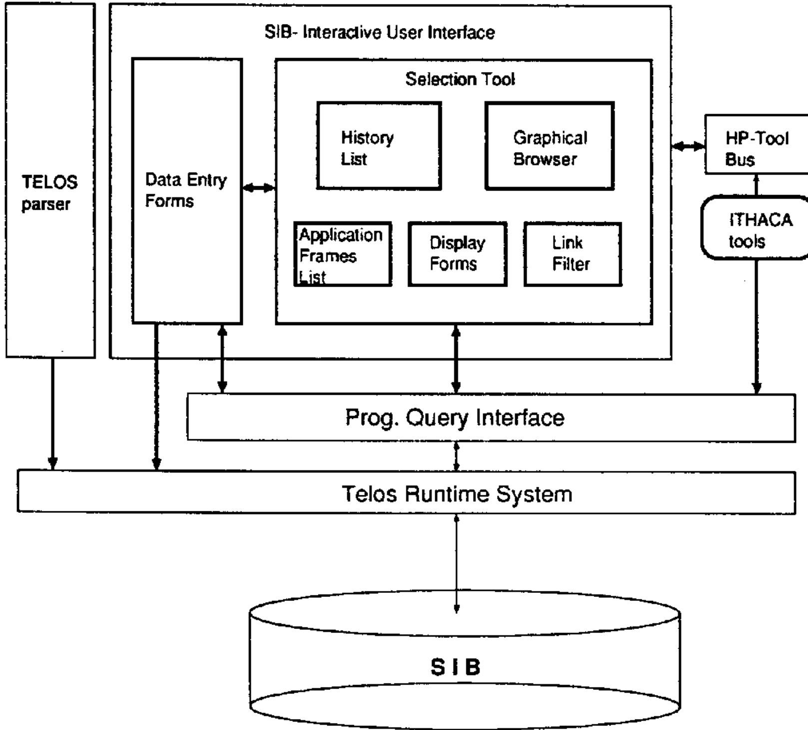
3.2 The SIB Architecture

The SIB system consists of the following modules (Figure 8):

- The *Interactive User Interface* generates and coordinates the other parts, including the interface tools of the Data Entry Forms and the Selection Tool. It is implemented using the OSF/Motif toolkit.
- As a component of the *Selection Tool,* the *Graphical Browser* presents parts of the SIB network graphically, and allows the user to browse through it by sending messages to the Interactive User Interface in response to user actions. The Graphical Browser is a LABY[2] graphical editor with only the working area present.
- The *Data Entry and Display Forms* offer form interfaces for entering and presenting data. These display forms are used to provide information about the current object or another selected node in a form layout. The forms

2. LABY is a general purpose graphical editor developed in part within the ITHACA project at the Institute of Computer Science, FORTH (Katevenis et al., 1990).

Figure 8. The SIB architecture



have been designed to support multimedia information (e.g., text, graphics, images, and animation).

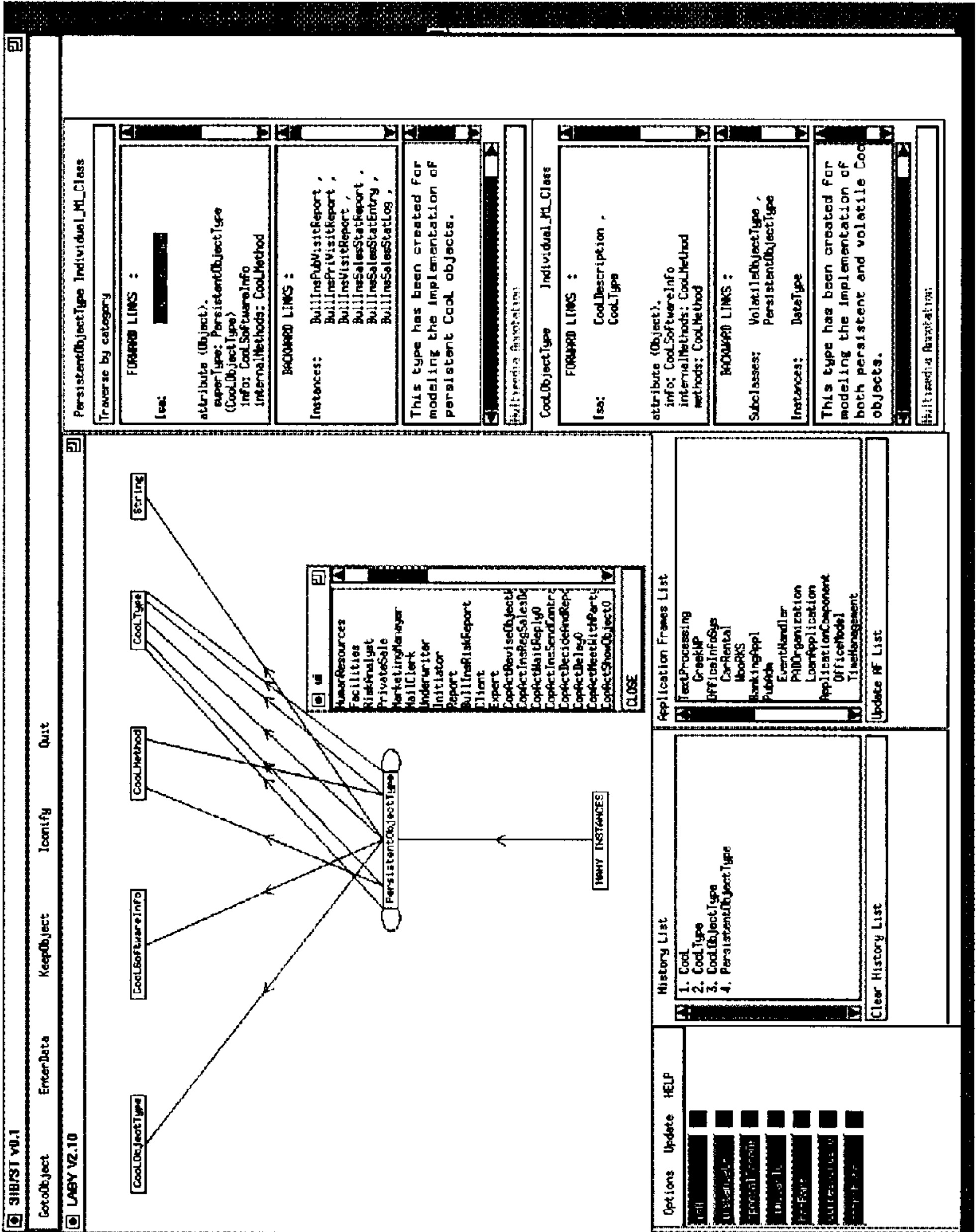
- The *Programmatic Query Interface* handles queries issued by various components of the Selection Tool, the Data Entry Form, or external tools. Processing of a query results in the construction of a file suitable for display by the Graphical Browser or the Display Form. In the current implementation, a separate query interface (based on the client-server model) is used for programmatic queries. However, the two query interfaces will be integrated in the future.
- The *ITHACA Tools,* integrated with the SIB at command level, use explicitly the Selection Tool for retrieval or access directly the programmatic query interface for query and data entry operations.
- The *Telos Runtime System,* described in Section 3.4, constitutes the kernel of the SIB system.

3.3 User Interface

The user interface of the system consists of the following windows (Figure 9):

- The *Graphical Browser*, built using the LABY graphical editor, displays a part of the SIB network in the neighborhood of the current object. The window of the Graphical Browser is topologically divided in two parts. At least one link emanates from each node appearing in the lower part, pointing to the current node. Likewise, there is at least one link emanating from the current node and pointing to each node appearing in the upper part. The types of links are represented by a color code, which is shown in the Link Filter. Nodes appearing in the graph of the browser are selectable with the mouse. The links displayed at any one time include direct links from the current object to/from other objects and computed isA and instanceOf links.
- The population of the display is controlled by means of the *Link Filter* (see below) and the *Instance Box*. The Instance Box appears in the display of the Graphical Browser when the instances of a certain active link type and adjacent to the current object are too many to be shown on the display. On selecting the Instance Box of a link class with the mouse, a list of objects related to the current one by that type of links appears. The objects on this list are selectable just like those displayed graphically.
- The *Link Filter* provides buttons corresponding to link types and is used for activating/deactivating links, thus controlling the information displayed in the Graphical Browser. The isA and instanceOf buttons further offer the option of displaying inherited as well as direct isA and instanceOf links. All buttons show the color code of the link classes and come with a help facility.
- The *History List* is a navigation aid intended to prevent users from getting lost, a common problem in hypertext systems (Conklin, 1987). The History List is scrollable and contains the names of the objects selected as current during a session in chronological order (the most recent one shown at the bottom, as with the history command of Unix). All entries of the list are selectable. A selection made on the History List is functionally equivalent to one made on the Graphical Browser.
- The *Application Frames List* contains the names of all application frames, thus presenting a bird's-eye-view of the SIB. This facility is provided as compensation for the limited scope of the Graphical Browser. The application frames are displayed in an indented list representing their hierarchical structure. Each item on the list is selectable, which effectively allows big jumps within the SIB network while in the browsing mode. Moreover, the Application Frames List serves as the initial entry point to the Selection Tool.
- The *Main Form* is the top right-most window of the Selection Tool and invariably displays information about the current object. This information includes the abstracted definition of the object in a form layout, generated

Figure 9. The SIB user interface.



by unparsing the SIB description of the object. The form may also contain multimedia annotations for the object, each one displayed in a separate window. At present, this annotation is textual and graphical, but can be of any other type with no additional effort, provided that appropriate tools exist within the SIB's operating environment.

- The *Auxiliary Form* is used to display information about an object other than the current one, without changing the actual view in the Graphical Browser (Figure 9, bottom right window). To minimize user distraction, only one auxiliary form can be open at a time and objects are not selectable on auxiliary forms. The auxiliary form is actually a preview mechanism and is offered as an orientation aid.
- Through the *Button Panel,* several other windows for performing a variety of useful functions can appear on demand. Currently these functions are:

  GotoObject: Allows direct access to invisible objects by name.

  EnterData: A Data Entry Form is offered for entering data into the SIB.

  KeepObject: Keeps a retrieved object in a local workspace.

  Iconify and *Quit*: Closes and iconifies a window; quits the Selection Tool respectively.

3.4 Implementation Aspects

As indicated earlier, the SIB model is based on Telos without its temporal reasoning mechanism and assertional sublanguage. Early experimentation with traditional and object-oriented database systems proved Telos to be superior in modeling flexibility. Moreover, its SIB implementation is based on C++ and clearly outperforms the earlier Prolog implementation of the language.

Before proceeding with an overview of the system implementation and the choices we have made, we list some of the object management and modeling requirements that actually shaped the implementation.

3.4.1 Object Management and Modeling Requirements. Foremost among these requirements is the need for a powerful and expressively rich data model. This requirement effectively ruled out traditional DBMSs based on the classical data models. However, there have been noteworthy investigations of the modeling requirements of software applications, including the studies of versioning models (Katz, 1990), the mechanisms supported in CACTIS for derived data (Hudson and King, 1989), and the configuration management model of Rose et al. (1991). Some work has been done in Software Engineering on the identification of relationship types for describing software objects (Meyer, 1985). Along a different direction, terminological languages such as CLASSIC (Borgida et al., 1989) offer facilities for defining terms that include a *subsumption* operation to determine automatically whether one

term description is more specialized than another. LaSSIE (Devanbu et al., 1991) illustrates graphically how such a facility can be used for a software repository.

A second requirement is the need for effective and efficient support for concurrent usage of the SIB, in addition to adequate query processing facilities. Unlike databases, software information bases contain mostly schema descriptions. Moreover, these schema descriptions include cycles and evolve dynamically. These considerations rule out the direct use of standard database implementation techniques, such as two-phase locking or directed acyclic graph methods for concurrency control. Current research indicates that the features of database transactions (Atomicity, Concurrency, Isolation, Durability) will have to be separated into multiple services, where each of these services might look quite different from the one used in traditional DBMSs. Software designers do not want to work in isolation but in overlapping workspaces with explicit communication. Atomicity and recovery have to be separated, because no one wants to reset a whole design transaction when a conflict occurrs. Likewise, the rich structure of software information bases such as the SIB render traditional query optimizations for DBMSs ineffective.

Of course, there are useful results in the area of databases, which can readily be adopted to improve the efficiency and safety of software information bases. For example, extending traditional database types with long fields and complex object structures eliminates the need for costly file-opening and closing operations when scanning software information. It also means that common parts can be shared among complex objects in a controlled manner. The DAMOKLES system (Dittrich et al., 1987) is an excellent example of this kind of extension.

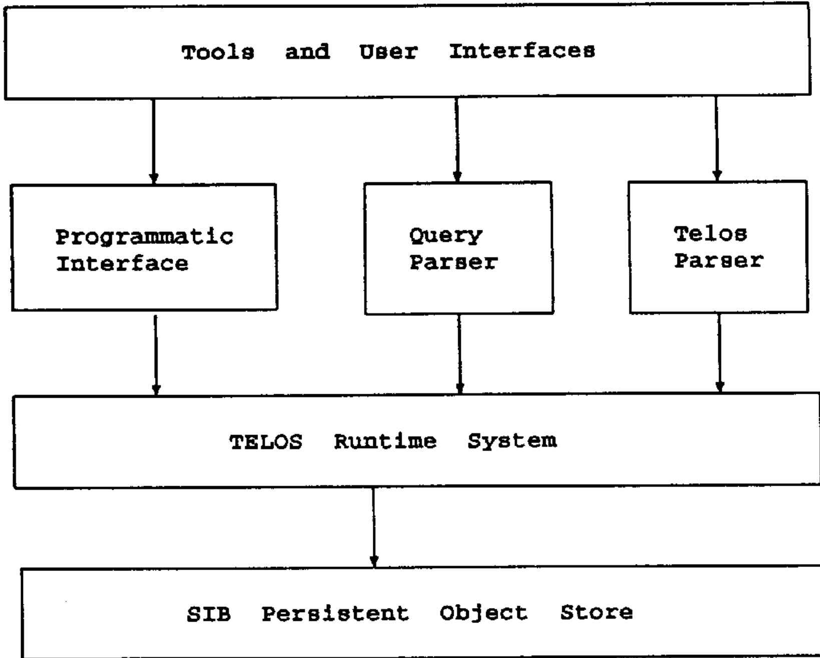
A final requirement of the SIB calls for a host of functions not available in DBMSs, including configuration managers that support the efficient and consistent re-configuration of complex objects when components have changed (e.g., CACTIS; Hudson and King, 1989), and similar measuring tools (e.g., code descriptions; Schwanke, 1991).

In summary, current database technology leaves much to be desired in terms of supporting software information bases intended for reuse. Accordingly, our work has been founded on a richer semantic data model that can be thought of as a layer on top of emerging object-oriented database systems. Nevertheless, the implementation technology for DBMSs in general, and object-oriented database systems in particular, has served as source of ideas and inspiration throughout. Admittedly, it is still an open question whether such a layered approach will be feasible for very large software information bases, or whether database technology will advance to satisfy some or all of the requirements discussed here.

3.4.2 System Implementation. As we indicated, the SIB has been developed as part of a complete application development environment containing several tools. Their interconnections are shown in Figure 10.

Architecturally, the prototype SIB system contains all the main components of a typical object-oriented DBMS implementation, but low-level optimizations and data

Figure 10. SIB tools interfaces and object store



structures are rather different (Dadouris et al., 1992; Constantopoulos et al., 1993). All data fields are set-valued and there has been much emphasis on efficient handling of network-like structures and fast retrieval of transitive closures for particular link types. All links can be traversed bi-directionally and there is direct support for all atomic retrieval operations (e.g., find all classes an object is an instance of, find all related objects). No optimization is done for range queries on primitive values (integers, floats, etc.). The schema is maintained completely at data-level, allowing for fast schema extensions at run-time. The data of a software information system are in general rather static, with infrequent and bulk changes, suggesting a strong preference for query optimization over update optimization. Telos objects are presented in terms of their (hopefully meaningful) external identifiers. Like object-oriented databases, objects have associated unique and system-supported internal identifiers (Khosafian and Copeland, 1986; Kim et al., 1989), which are invisible to the user.

Five C++ classes are used to represent all Telos objects. The contents of the five classes are sets of system identifiers for the *instanceOf, isA, attribute,* and *trigger* relations, respectively. All of these relations are kept in both directions to allow fast query processing. Triggers are either built-in or user-supplied, in which case they must be linked to the runtime system. Triggers can be used to provide access to data outside the database, thereby offering a powerful interface mechanism.

The Telos objects are kept persistent on disk and copied to memory in a cache-like manner. The system implemention employs demand loading in the current prototype, to be replaced later with a prefetch mechanism (Low, 1988). During the execution of a transaction, all modifications done on existing and newly created objects are kept in memory, and only become persistent at the end of the transaction. Queries may be allowed on these objects before the end of a transaction under certain restrictions. All dynamic memory allocations are done on a fixed granularity base to reduce allocation time and, more importantly, to avoid cluttering the virtual address space after longer execution times. Similar implementation issues have been discussed (Khosafian and Frank, 1988; Merrow and Laursen, 1987; Skarra and Zdonik, 1986). The association of system identifiers with object locations on persistent store and/or in memory is done by the system catalogue. The system identifiers are kept dense in the sense that the numerically lowest free identifier is allocated first. Moreover, since system identifiers are only internal, there are no identity conflicts. Identifier density allows a virtual memory-like indexing of the system catalog with use of page tables. Since it requires an additional indirection step after a growth of the database by a factor of 1,000, this design leads to nearly size-independent performance. Symbol tables translate from external identifiers to system identifiers and vice versa. The symbol table tuples are organized as the system catalog, giving good performance for translations from system identifiers to external identifiers translation. The inverse translation, however, is supported by B-trees and is the only component whose performance degrades with growing database size. Fortunately, the frequency of the inverse translation operation is several orders of magnitude lower than forward translation, since queries usually return larger answer sets than their argument set and all internal query processing is done in terms of system identifiers. Symbol tables as well as the system catalog are cached.

Tests for measuring the performance of the SIB with a population of 12,000 objects (links and nodes) yield query response times between 1 and 4 seconds (the maximum occuring for recursive queries following over 1,000 links). With a population ten times larger (120,000 objects), the response times for the same queries (now following up to 10,000 links) range from 1.2 to 4.5 seconds.

Concurrency control in network-like structures is still an open problem. Database parts to be locked can hardly be determined. Based on the assumption that updates are not too frequent, we have built only read and write locks into the implementation for the whole database. This implies that locks may be held only for short time intervals and that interactive data entry form operations must check consistency before they commit their data to the database. Cache invalidation is done at the lock grant on demand of each server instance, thereby minimizing degradation of performance with an increasing number of clients. The lock mechanism works reliably on local area networks.

Finally, the implementation has emphasized *portability* of the platform across all UNIX systems and a variety of hardware settings, including PC-based ones. The

system currently runs on Sun3, Sun4 series, SparcStations and 386 machines under UNIX. The X window system is required, preferably with a color monitor. The system may run in a local area network being based on a client-server architecture. Query processing using caches of controllable size is mainly done on the server side. Usage of shared caches for read-only access is currently under investigation.

4. Empirical Evaluation of the SIB Concept

The SIB model and system have been evaluated in the context of a specific reuse-oriented methodology developed in the ITHACA project. Moreover, we have begun the evaluation of the model with other object-oriented analysis and design methodologies (Fichman and Kemerer, 1992) starting with the Booch design methodology (Booch, 1991). An obvious advantage of doing the evaluation using the ITHACA methodology is that there already exist substantial amounts of data, generated by companies participating in the project. After all, it is difficult to establish the strengths of the SIB unless it is first properly and heavily populated.

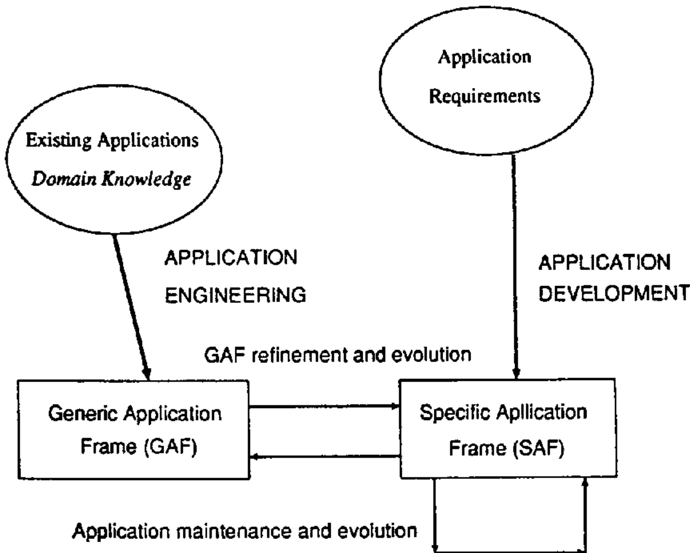
This section reviews the reuse-oriented development methodology, the test application domain and applications used to validate the SIB concept, and an extended example from these test applications intended to demonstrate the interaction with the system.

4.1 The ITHACA Object-Oriented Methodology

Consider a concrete scenario for software reuse, adopted from Ader et al. (1990), De Antonellis et al. (1991), and Fugini et al. (1992). The scenario assumes that all software information is organized in terms of Application Frames (AFs) which comprise descriptions of requirements, designs, implementations, and their interdependencies. Thus, an AF provides three views of a software system, plus some process information. As indicated earlier, several different models can be supported for each of the three views. For example, in the Object-Role-Model (ORM; Pernici, 1990), the requirements view is a network of application objects connected by roles. Under certain applicability conditions, design-level software object specification can be related to the set of roles the object is intended to support. Similarly, implementation objects are related to their design-level counterparts under certain applicability conditions. In general, design objects may be associated with multiple requirements, and implementation objects may be associated with multiple designs, and vice versa.

Development proceeds from a library of Generic Application Frames (GAFs) and the result of a development process is a Specific Application Frame (SAF). From the viewpoint of an application domain, a GAF is an abstraction of a collection of applications pertinent to this domain, while a SAF is a specific running application in the same domain. It is assumed that the initial class libraries and an AF library have been generated by *application engineers* and are clustered by application domains.

Figure 11. Application development scenario in ITHACA



The test application domain is Public Administration, including information systems for office applications. The *application developers* follow the steps below to produce a new application (Ader et al., 1990; see also Figure 11).

1. *Select an application frame* from the repository, meeting the requirements for the application being developed.
2. *Select useful classes.* In employing an application frame, the developer is guided to select reusable classes from the repository in that the AF drives requirements collection and specification according to pre-existing generic specifications and designs.
3. *Tailor classes.* The selected classes are adapted (incrementally modified), using previous design experiences, by supplying parameters or by modifying class behavior through inheritance.
4. *Use script application.* A new application is composed by linking design classes together by means of a "script" (Tsichritzis, 1991). The script artifacts are to be entered in the repository for future reuse.
5. *Monitor behavior and continuously develop.* Through testing and validation, or because of changes in the requirements, the application is adapted.

This methodology has been tested with an application development environment that includes the SIB, a requirements collection and specification tool supporting the ORM (RECAST; Fugini, 1992), a visual scripting tool (VISTA; de Mey et al., 1991), and the runtime environments of the object-oriented languages CooL (CooL, 1990) and C++.

Among the models that the SIB stores and supports, the ORM, C++, and CooL models are particularly relevant. A comprehensive account of how these models have been stored in the SIB is presented by Charalabidis et al. (1992). It is noted that since the SIB descriptions for the object-oriented programming language models (CooL and C++) are rather straightforward representations of the code, it was possible to construct tools that generate Telos descriptions from CooL and C++ code automatically and classify them within the SIB at the same time.

Public administration has been adopted as the test application area for the SIB. Professional application developers from three commercial organizations generated the code and the description of a generic office work flow system (WooRKS; Ader et al., 1991). The system offers assistance to a group of users collaborating to achieve a sequence of tasks to accomplish an office procedure. WooRKS consists of models for organizations, information handling, time, operation, and coordination of activities.

The current population of the SIB is about 20,000 separate objects (logical identifiers) for the above application. Most were introduced manually by the application developers, since the automatic population tools were not available at the time. The positive usage experiences of the developers were presented by Proefrock et al. (1992) and Charalabidis et al. (1992). Most of the interaction of the application developers is through the form-based SIB interface which provides full transparency from Telos. Application engineers, who also use Telos directly, have found the E-R nature of the language and the graphical visualization very effective.

4.2 An SIB Usage Example

Suppose we are assigned the task of creating an application dealing with letter processing to assist secretaries, managers, and others in writing, checking, and mailing professional letters. Looking at the AF List, we observe that an office information system, WooRKS, already exists. Actors, roles, and procedures are the basic concepts in WooRKS.

Our starting point in exploring the SIB will then be WooRKS, selected through the AF List. In the natural language description in the Main Form we read that WooRKS is an office work flow system that handles various activities (Figure 12). Therefore, it is a reasonable candidate to search for a letter processing application. The description of WooRKS includes three attributes which are further explained in the Main Form. One of them, *reqDescr,* describes WooRKS requirements, providing further information about what WooRKS actually does. It is convenient to preview the contents of this description before deciding to make it the current object. This is done using the Auxiliary Form, by selecting WooRKS1_RD_FORM from the Main Form.

In Figure 13, WooRKS1_RD_FORM is current in the Graphical Browser window. Among the classes of activities that it handles, OrderProcessing appears to be the most relevant, so we inspect it. After we conclude that this is not useful, we return

to WooRKS1_RD_FORM through the History List and try WarehouseProcessing and AccountProcessing, which also turn out to be irrelevant to our task. However, we notice that all three are instances of FormProcessClass.

To inspect FormProcessClass, we move to it using the GotoObject facility (Figure 14). FormProcessClass inherits the attributes of its superclass FormClass. By previewing the latter we see that it has two attributes, *roles* and *baseRole* (Figure 14). We decide to create a new instance of FormProcessClass, called LetterProcessing, whose respective attributes correspond to the requirements of our letter processing application. In particular, the *baseRole* of LetterProcessing will be LP_base_role, and the *roles* will be LP_letterCompose, LP_letterCheck, LP_letterApprove, LP_letterSend, LP_letterReceive, and LP_letterArchive.

The naming convention for *roles* is chosen by analogy to existing form roles. To see those we inspect FormRole (value domain of the *roles* attribute) by making it current with the GotoObject facility (Figure 15). Because FormRole has too many instances, these are not directly displayed on the Graphical Browser, but are shown instead on a scrollable list after clicking on the MANY INSTANCES box appearing on the Browser.

We now create the *roles* and *baseRole* of LetterProcessing. As an exploratory step related to the creation of LP_letterArchive, we inspect OP_orderArchive by selecting it from the list of instances of FormRole (Figure 15). This step turns out to be useful because a *correspondsTo* link from OP_orderArchive to ArchiveAct is found, the latter being an instance of ADMActivity (see the Main Form in Figure 16). Since ADMActivity is a class of design descriptions, we can define the design description of the new LP_letterArchive by analogy to ArchiveAct or simply reuse the latter. Similarly, we may use CompileRefAct and/or EvaluationOrderAct which *correspondTo* OP_orderCheck as the instance(s) of ADMActivity (describing e.g., LP_letterCheck).

Finally, to actually define the new LetterProcessing description, we move to FormProcessClass using the GotoObject option, and then invoke the EnterData facility, which does not presume any knowledge of the Telos syntax. Figure 17 shows the Data Entry Form for LetterProcessing along with the result of entering the information.

Figure 12. The selection tool of the SIB. `WooRKS` **is the current object.**

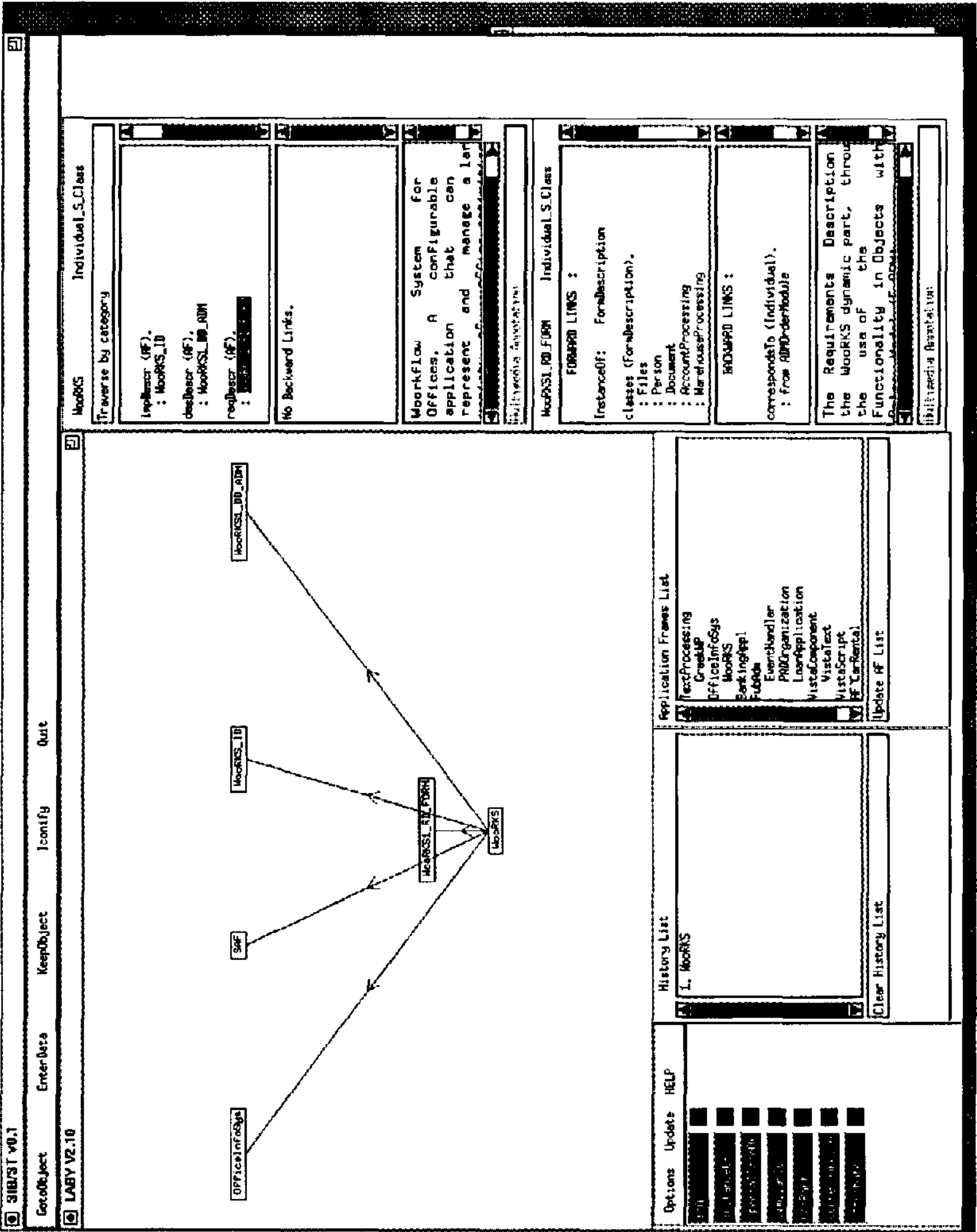


Figure 13. Requirements of `WooRKS` and preview of `OrderProcessing`

Figure 14. Inspecting `FormProcessClass` **and previewing** `FormClass`.

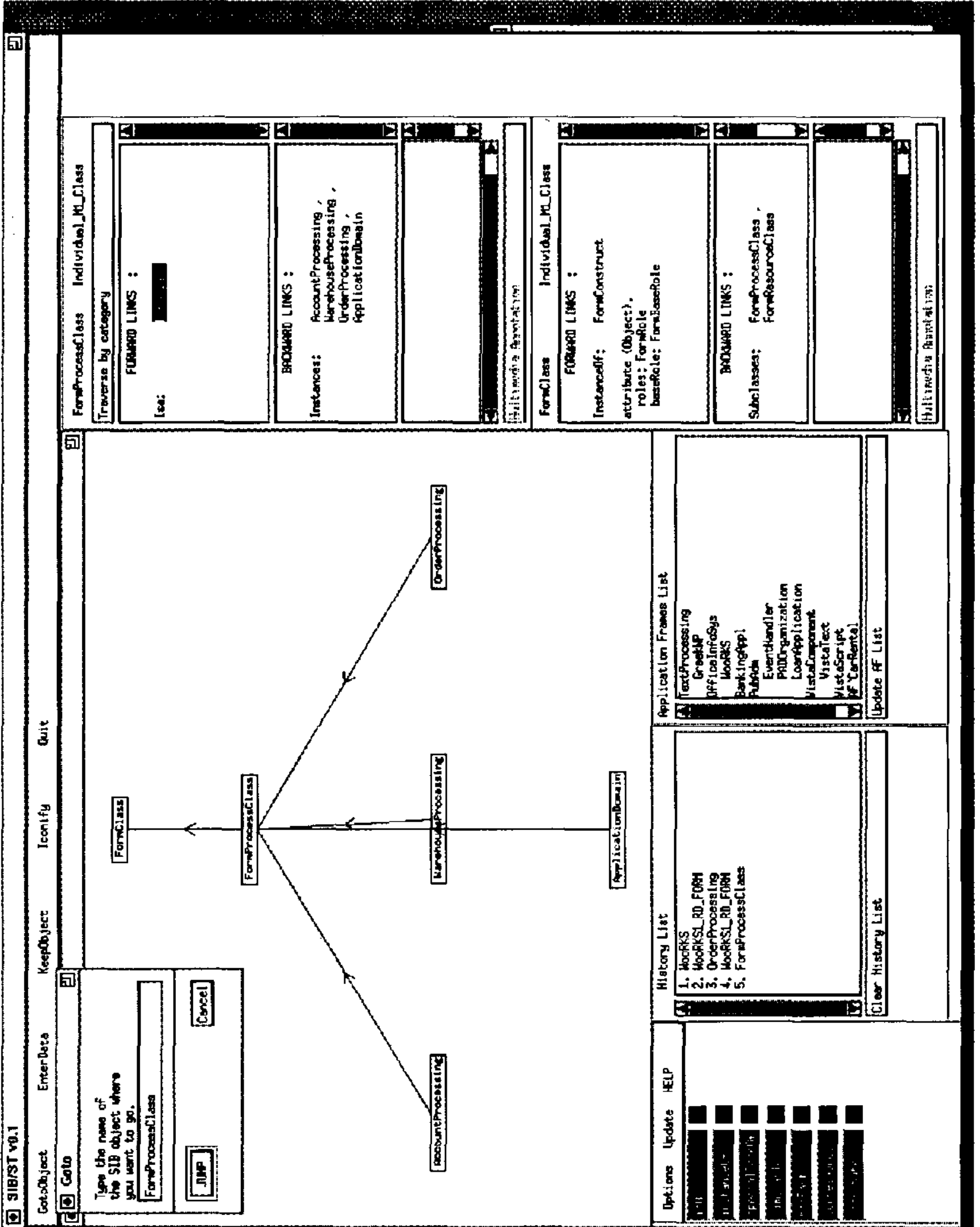


Figure 15. Instance Box display: Numerous instances of `FormRole`

Figure 16. Textual annotation to `ArchiveAct`

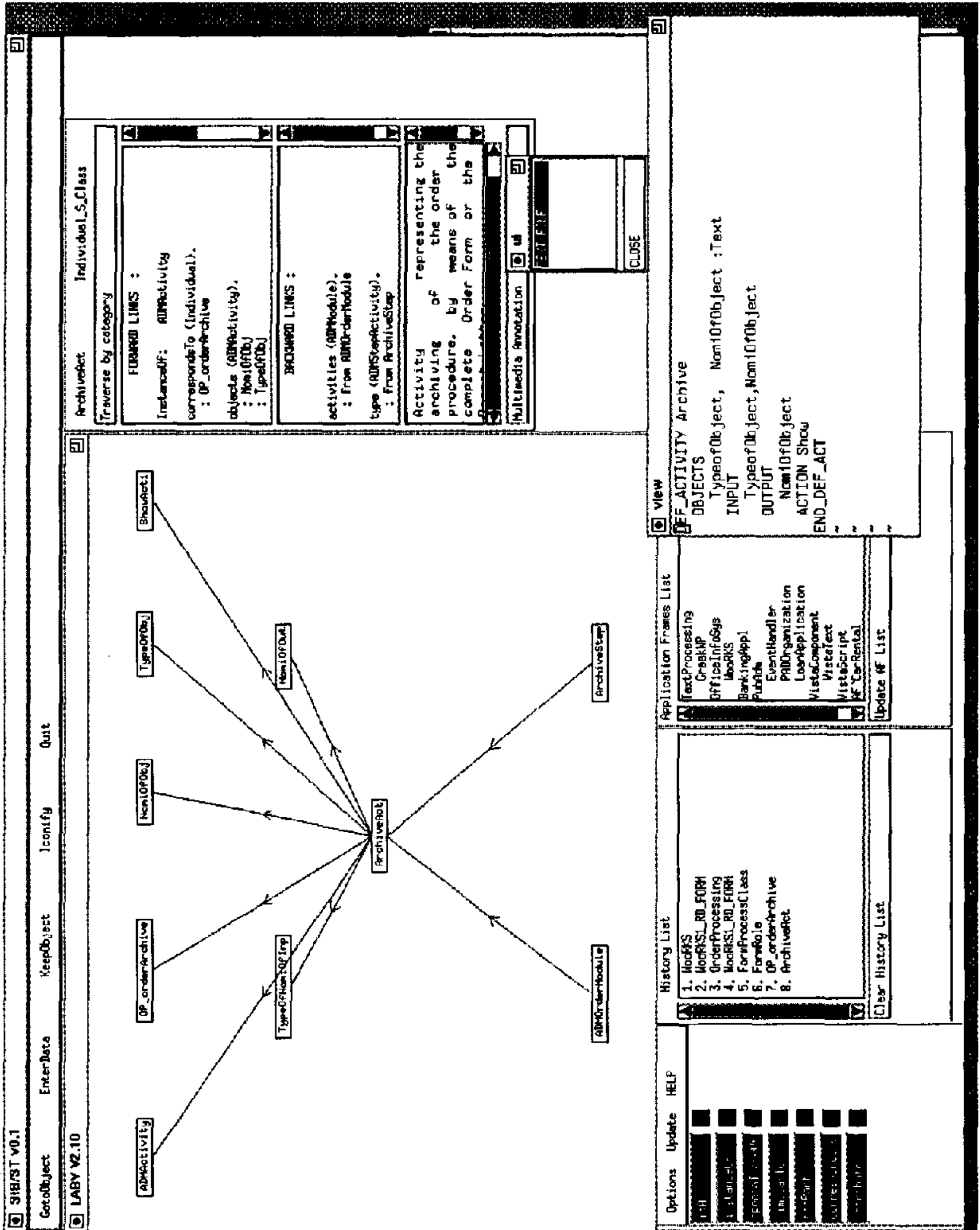


Figure 17. `LetterProcessing` **inserted in the SIB through the Data Entry Form**

5. Conclusion

The design of the SIB integrates ideas and techniques from knowledge representation, graphics, databases, and software engineering to offer a basis for supporting software reuse. To make the integration possible, these techniques had to be enhanced and extended. Among the enhancements we note the efficient handling of large conceptual schemata in Telos; the rapid change of viewpoints and flexible presentation of large information bases supported by the constraint-based presentation mechanisms of LABY (Katevenis et al., 1990); the tailor-made optimization of the SIB data management functions (Dadouris et al., 1992); and the extension of the DAIDA framework for information systems representation (Jarke et al., 1992) by object and link types specifically dedicated to reuse. Scalability, portability, size-independent performance, and support of a multi-paradigm access with a carefully designed interface have all served as guiding principles of the prototype design and implementation.

The population of the SIB with software information about a real office application, a first test of the SIB concept, has demonstrated the breadth of reuse viewpoints handled effectively and the practical usefulness of the specific graphical support. Furthermore, performance results confirm the technical choices made for the SIB. Ongoing experiments in different application domains and with different application development methodologies seem to indicate that this success is, in fact, generalizable.

However, the construction of application-specific SIBs requires more facilities than those offered by the current prototype. In particular, additional research is needed to address the problem of computer-assisted acquisition of similarity links. Ongoing work in this direction was reported by Spanoudakis and Constantopoulos (1993). To guide the SIB developer (or to partially automate the job), reference models for different classes of applications would be of great value, analogous to the call for "shared ontologies" in the Knowledge Sharing Project (Patil et al., 1992) of the Defense Advanced Research Projects Agency (DARPA). The extensibility offered by Telos is a big asset in the definition of such reference models. Moreover, besides defining the "right" classes of software objects and their relationships, there are problems with the description of individual objects to be classified. We need to know how to describe them, where to place them, and how to do most of this automatically. A useful scheme of classification facets was presented by Prieto-Diaz (1991). Based on library science classification mechanisms, this scheme distinguishes facets such as the actions a system performs, its characteristics as an object that has been created and stored, the main data structure it supports, or its intended usage. Based on this work and on semantic networks, a classification model (the AIRS model) was proposed by Ostertag et al. (1992) and has been used for Ada and C libraries. The goal of this work is to provide for similarity-based retrieval and to automate the classification process. We have adapted this model to deal with the idiosyncrancies of the object-oriented nature of the languages used within

the ITHACA setting.

Another potential extension of the SIB system could involve special-purpose tools for more "intelligent" support of the classification and retrieval process. One such tool has been developed in prototype form (Katalagarianos and Vassiliou, 1992). It employs case-based reasoning to adopt past experiences in searching for "analogous" software objects.

In the longer term, we want to experiment with the SIB model and system to include other, more general, software-related information such as business plans, organizational strategies, and the like. Such external information provides non-functional requirements on the systems to be developed, which can be exploited to steer the search process for reusable components. Non-functional requirements such as system performance, robustness, cost, and security can also play an important role in selecting software components and in understanding the rationale behind software system structure, thereby facilitating their adaptation.

Acknowledgements

The SIB system is the result of a large collective effort. We wish to acknowledge the influence of our colleagues within our respective research groups, particularly the SIB team at the Institute of Computer Science, FORTH, through long discussions and implementation work. In particular, we acknowledge the invaluable contributions of the technical manager of the SIB team, Dr. Martin Doerr, and those of Maria Theodoridou and Eleni Petra. We are grateful to Dr. Doerr and to Elena Pataki for help with the sections on implementation and the usage example, respectively. Finally, we acknowledge the fruitful cooperation with members from other ITHACA partners, namely, Siemens-Nixdorf (Germany), Universite de Geneve (Switzerland), TAO (Spain), National Technical University of Athens (Greece), Bull (France), and Politecnico di Milano (Italy), and the constructive and insightful comments of Prof. François Bodart.

References

Ader, M., Nierstrasz, O., McMahon, S., Mueller, G., and Proefrock, A-K. The Ithaca technology: A landscape for object-oriented application development. *Proceedings of the ESPRIT 1990 Conference,* Dordrecht, The Netherlands, 1990.

Ader, M., Murthy, S., Murthy, N., Bergnes, C., and Monguio, J. Organization model reference manual, ITHACA Report, ITHACA.Bull.91.D.1.4.#1.2, 1991.

Attardi, G. and Simi, M. Completeness and consistency of OMEGA: A logic for knowledge representation. *Proceedings of the International Joint Conference on Artificial Intelligence,* Vancouver, British Columbia, 1981.

Barletta, R. An introduction to case-based reasoning, *AI Expert,* August, 1991, pp. 43-49.

Bigelow J. Hypertext and CASE. *IEEE Software,* 5(2):23-29, 1988.

Biggerstaff T.J. and Perlis, A.J. *Software Reusability, Volume I: Concepts and Models, Volume 2: Applications and Experience,* ACM Press Frontier Series, Reading, MA: Addison-Wesley, 1989.

Biggerstaff, T.J. and Richter, C. Reusability framework, assessment, and directions. In: Biggerstaff, T.J. and Perlis, A.J., eds, *Software Reusability, Volume I: Concepts and Models,* ACM Press Frontier Series, Reading, MA: Addison-Wesley, 1989, pp.1-17.

Brodie, M. and Ridjanovic, D. On the design and specification of database transactions. In: Brodie, M., Mylopoulos, J., and Schmidt, J., eds., *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages,* New York: Springer-Verlag, 1984.

Booch, G. *Object Oriented Design with Applications,* Redwood City, CA: Benjamin/ Cummings Publishing Company, 1991.

Borgida, A., Brachman, R., McGuiness, D., and Resnick, L. CLASSIC: A structural data model for objects. *Proceedings of the ACM SIGMOD Conference,* Portland, OR, 1989.

Charalabidis, Y., Petra, E., and Vlodakis, G. Populating software repositories: The SIB-WooRKS case. *Proceedings of the ERCIM Workshop on Methods and Tools for Software Reuse,* Heraklion, Crete, 1992.

Chen, P.P. The entity-relationship model: Towards a unified view of data. *ACM Transactions on Database Systems,* 1(1):9-36, 1976.

Conklin J. Hypertext: An introduction and survey. *IEEE Computer,* 20(9):17-42, 1987.

CooL 0.2 Language Description, Report ITHACA.SNI.90.L2.#4, Siemens-Nixdorf, 1990.

Constantopoulos, P., Doerr, M., and Vassiliou, Y. Repositories for software reuse: The software information base. *Proceedings of the IFIP WG 8.1 Conference on Information System Development Process,* Como, Italy, 1993.

Dadouris, C., Doerr, M, Kizlaridou, S. Mavroidis, D., Pataki, E., Theodorakis, E., and Yeorgiannakis, G. Implementation of the SIB System, Report ITHACA.FORTH. 92.E2.#2, Institute of Computer Science, FORTH, January 1992.

De Antonellis, V., Pernici, B., and Samarati, P. Object Orientation in the Analysis of Work Organization and Agent Cooperation, *Proceedings of the International Conference on Dynamic Aspects in Information Systems,* Washington, D.C., 1991.

de Mey, V. Junod, B., Renfer, S., Stadelmann, M., and Simitsek, I. The implementation of VISTA: A visual scripting tool. In: Tsichritzis, D., ed., *Object Composition,* Centre Universitaire d'Informatique, Universite de Geneve, 1991, pp. 31-56.

Devanbu, P., R. J. Brachmann, P. Selfridge, and B. Ballard. LaSSIE: A knowledge-based software information system, *Communications of the ACM,* 34(5):34-49, 1991.

Dittrich, K. et al. DAMOKLES: A database system for software engineering environments. *Lecture Notes in Computer Science,* 244, 1987.

Esposito, F., Malerba, D., and Semeraro, G. Classification in noisy environments using a distance measure between structural symbolic descriptions. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 14(3):390-402, 1992.

Fichman, R.G. and Kemerer, C.F. Object-oriented and conventional analysis and design methodologies. *IEEE Computer,* 25(10):22-39, 1992.

Fugini, M., Niestrasz, O., and Pernici, B. Application development through reuse: The ITHACA tools environment. *SIGOIS Bulletin,* 13(2):00-00?, 1992.

Garg, P. and Scacchi, W. On designing intelligent hypertext systems for information management in software engineering, DIF. *Proceedings of Hypertext 87,* November 1987.

Garg, P. and Scacchi, W. ISHYS: Designing an intelligent software hypertext system, *IEEE Expert,* 4(3):52-62, 1989.

Hudson, S.E. and King, R. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems,* 14(3):291-321, 1989.

Jarke, M., Mylopoulos, J., Schmidt, J., and Vassiliou, Y. DAIDA: An environment for evolving information systems. *ACM Transactions on Information Systems,* 10(1):1-50, 1992.

Jarke, M., ed., *Database Application Development with DAIDA,* Heidelberg: Springer-Verlag, 1993.

Jeusfeld, M. Change Control in Deductive Object Bases, DISKI Vol. 17, St. Augustin, Germany, INFIX Publ., 1992 (in German, also Doctoral Dissertation, University of Passau).

Jobes, B. Repository comes of Age. *Database Programming and Design,* March, 1990, pp. 18-30.

Jones, M.R. Unveiling repository technology. *Database Programming and Design,* April, 1992, pp. 28-35.

Katevenis, M., Sorilos, T., Georgis, C. and Kalogerakis, P. *LABY User's Manual,* Version 2.10, ITHACA Report, ITHACA.FORTH.90.E.3.3.#7, 1990.

Katz, R. Towards a unified framework for version modelling in engineering databases. *ACM Computing Surveys,* 22(4):375-408, 1990.

Katalagarianos, P. and Vassiliou, Y. Employing genericity and case-based reasoning to effectively reuse code. *Proceedings of the International Computer Science Conference,* Hong Kong, 1992.

Khosafian S. and Copeland G.P. Object identity. *Proceedings of the ACM OOPSLA,* 1986.

Khosafian S. and Frank, D. Implementation techniques for object-oriented Databases. *Proceedings of the AOODS,* Springer, 1988.

Kim W., Kim K.-C., and Dale A. Indexing techniques for object-oriented databases. In: Kim, W. and Lochovsky, F., eds., *Object-Oriented Concepts, Databases, and Applications,* New York: ACM Press, 1989.

Korson, T. and McGregor, J. Technical criteria for the specification and evaluation of object-oriented Libraries. *Software Engineering Journal,* 1992.

Krueger, C.W. Software reuse. *ACM Computing Surveys,* 24(2):131-183, 1992.

Low C. A shared, persistent object store. *Proceedings of the European Conference on Object-Oriented Programming,* 1988.

Merrow T. and Laursen J. A pragmatic system for shared persistent objects. *Proceedings of OOPSLA,* 1987.

Meyer, B. Software knowledge bases. *Proceedings of the International Conference on Software Engineering,* London, 1985.

Michalski, R. Learning from observation: Conceptual clustering. In: *Machine Learning: An AI Approach,* Vol. 1, San Mateo, CA: Morgan Kaufmann, 1986.

Mylopoulos, J., Borgida, A., Jarke, M. and Koubarakis, M. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems,* 8(4):325-362, 1990.

Ostertag, E., Hendler, L., Prieto-Diaz, R., and Brown, C. Computing similarity in a reuse library system: An AI-based approach. *ACM Transactions on Software Engineering and Methodology,* 1(3):205-228, 1992.

Patil, R., Fikes, R., Patel-Schneider, P., Mckay, D., Finin, T., Gruber, T. and Neches, R. The DARPA knowledge sharing effort: Progress report. *Proceedings of the Third International Conference on Knowledge Representation and Reasoning,* Boston, 1992.

Pernici, B. Class design and metadesign. In: Tsichritzis, D., ed., *Object Management,* Centre Universitaire d'Informatique, Universite de Geneve, 1990.

Plotkin, D. Selecting a repository. *Database Programming and Design,* 1992, pp. 28-35.

Prieto-Diàz, R. Implementing faceted classification for software reuse. *Communications of the ACM,* 34(5):88-97, 1991.

Proefrock, K. and McMahon, S. *ITHACA Final Report,* Berlin: Siemens-Nixdorf, 1992.

Rose, T., Jarke, M., Gosek, M., Maltzahn, C., and Nissen, H. A decision-based configuration process environment. Special Issue on Software Environments and Factories, *Software Engineering Journal,* 6(5):332-346, 1991.

Ross, D.T. Structured analysis (SA): A language for communicating ideas. *IEEE Transactions on Software Engineering,* 1977.

Russel, S. Analogy by similarity. In: Hellman, D.H., ed., *Analogical Reasoning,* Dordrecht, The Netherlands: Kluwer Academic, 1988.

Schwanke, R. An intelligent tool for re-engineering software modularity. *Proceedings of the International Software Engineering Conference,* Austin, TX, 1991.

Skarra A.H. and Zdonik S.B. The management of changing types in an object-oriented database. *Proceedings of the OOPSLA,* 1986.

Spanoudakis, G. and Constantopoulos, P. Similarity for analogical software reuse: A conceptual modelling approach. *Proceedings of the Fifth International Conference on Advanced Information Systems Engineering,* Paris, 1993.

Tsichritzis, D., ed., *Object Composition,* Centre Universitaire d' Informatique, Universite de Geneve, 1991.

Tversky, A. Features of similarity. *Psychological Review,* 1977.

Wegner, P. The object-oriented classification paradigm. In: Shriver, J. and Wegner, P., eds, *Research Directions in Object-Oriented Programming,* MIT Press, 1987.

Winston, M. and Chaffin, R. A taxonomy of part-whole relations. *Cognitive Science,* 11:417-444, 1987.