Repositories for Software Reuse: The Software Information Base

Panos Constantopoulos, Martin Doerr and Yannis Vassiliou

Institute of Computer Science
Foundation of Research and Technology - Hellas
Heraklion, Crete, Greece
e-mail: {panos|martin|yannis}@ics.forth.gr

Abstract
Repositories play a pivotal role in an integrated reuse- based application development environment. Reusing software components implies their persistent storage and maintenance, and the ability to efficiently find them. Repositories built with reuse in mind can be considered as special-purpose information systems, required to support powerful semantic modelling, flexible retrieval of varied software descriptions of multimedia nature, and efficiency optimization directed towards a large variety of classes rather than large populations per class. The Software Information Base (SIB), described in this paper, is an illustrative case where issues of organisation, multi-paradigm access and challenging implementation choices are set forth.

Keyword Codes: D.2.2; D.2.9; H.3.3
Keywords: Software Engineering, Tools and Techniques; Management; Information Storage and Retrieval, Information Search and Retrieval

1. INTRODUCTION

Software reuse [Bigg87, Bigg89, Meye87] is being touted as a very promising approach to deal with the software crisis constituents (late deliveries, unreliable software, prohibitive development and maintenance costs, etc.) It is commonly agreed that software reuse does not only concern code; reuse of a software object implies the concurrent (re)use of objects associated with it, plus information about the objects. Thus, designs requirements specifications, development processes, and decision experiences are also candidates for reuse.

Software artifacts are stored in a repository which supports their efficient selection. Comprehending their functionality and usage before adaptation and subsequent use in the composition of a new application, is also recognised as essential in the reuse process.

Adopting the reuse-based development cycle of Figure 1.1, one can immediately detect the important role played by a reuse-targeted repository, together with its associated functions/tools. Such special-purpose repositories are often lightly mentioned or assumed while discussing reuse issues (e.g., designing-for-reuse), without the proper consideration of the idiosyncrasies, research challenges, and implementation difficulties they present. This paper reports the experiences in building a repository system (the *Software Information Base - SIB*) targeted to support the component-based development of very large software systems within the ITHACA project (Esprit No.2705) [Ader90]. The SIB can be seen as a complex functionality information system (like a large DBMS) providing for persistent storing, sharing, accessing, managing and controlling data. What mostly distinguishes it from a traditional DBMS is its support for conceptual modelling which

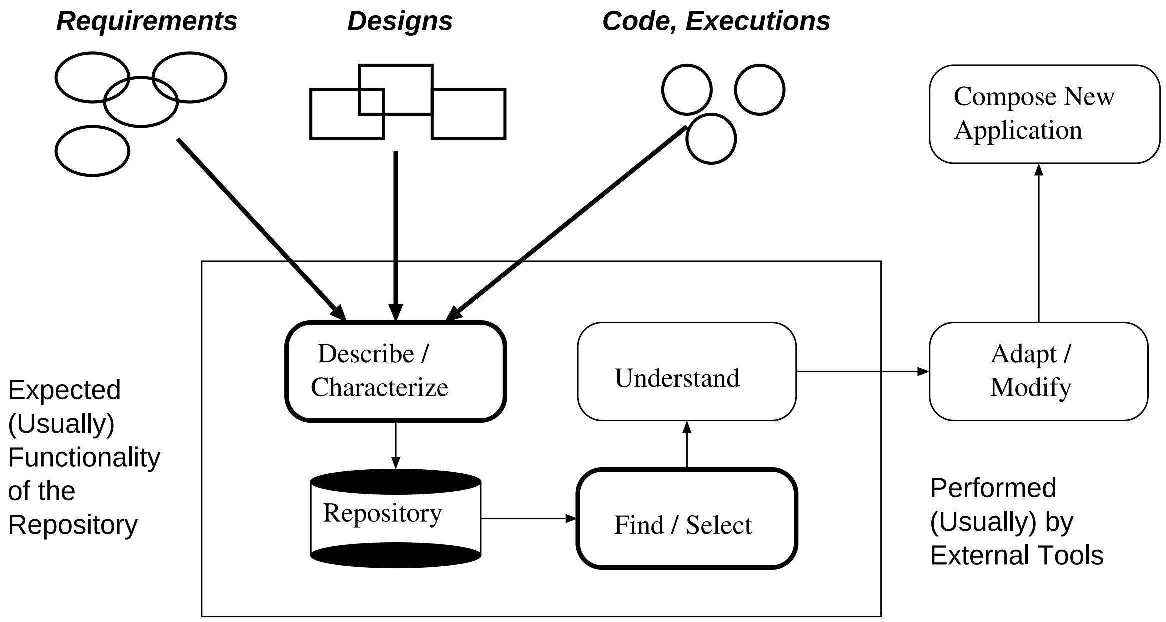

Figure 1.1: The Reuse Process

features, in addition to the usual constructs, a capacity for uniform treatment of schemata and data allowing schema definition at runtime and a uniform treatment of entities and relations. Other distinguishing characteristics are its capacity for flexible retrieval and browsing of multimedia data; and the optimization of efficiency for network structures consisting of a large variety of classes rather than relatively few classes with large populations per class (typical in a DBMS).

As can be seen in Figure 1.2, which applies the reuse process in our setting, the TELOS conceptual modelling language is used for the uniform description and organisation of software artifacts. TELOS is a specialised knowledge representation language for the development of information systems [Mylo90]. A corresponding graphical visualisation for each artifact description is readily available. The software components are located and retrieved via a selection mechanism comprising querying and browsing modes. Finally, the semantic nature of TELOS descriptions assists in understanding the functionality, usage and behaviour of the software components described. The latter is assisted by associated multimedia annotations.

Software reuse via repositories is not a new concept; there have been several approaches, not necessarily resembling the framework presented in our work, which can be characterised on two orthogonal dimensions. The first dimension concerns the methodologies for organising the reusable components in the repository, while the second concerns the methodologies for selecting the appropriate object(s) for reuse.

Library cataloguing (e.g., mathematical libraries - [Hopk88]) has been a prevalent way to organise code. In recent years we have seen shifts from simple *hierarchical classification* schemes to *faceted* ones (e.g., [Prie91]). *Hypertext* software organisation forms [Conk87] have also been applied for instance at HP's Kiosk, while semantic organisations range from simple object-oriented libraries (Eiffel [Meye90], Objective-C) to ER models (e.g., IBM's Repository [Low88]), to richer semantic models (LaSSIE at AT&T [Deva91]). *Hybrid* organisations (combining principles from the above) appear

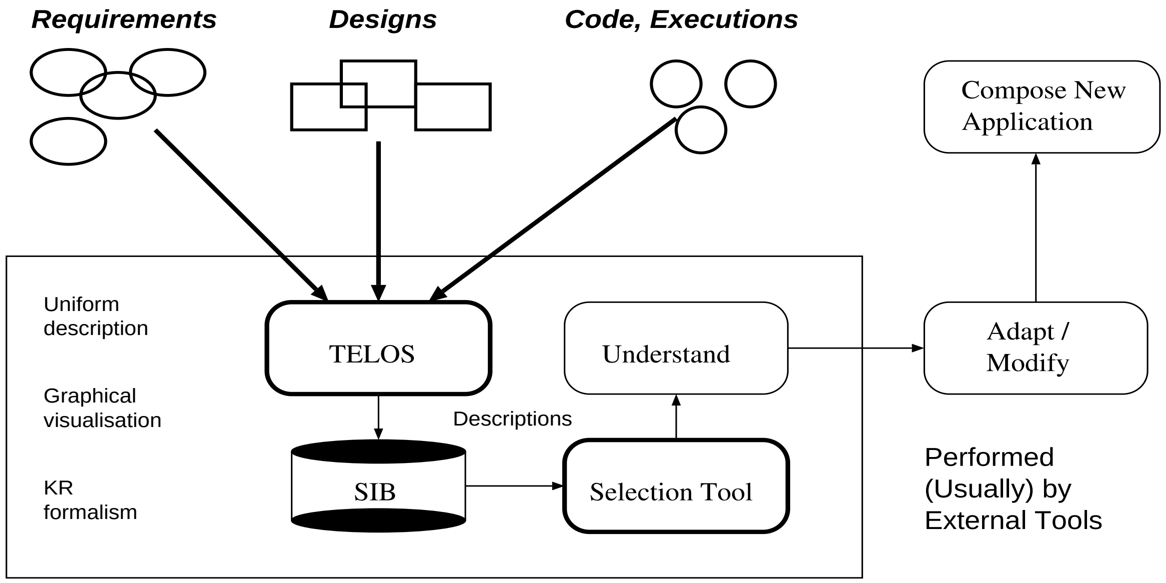

Figure 1.2: The Reuse Process with the SIB

promising [Cons93].

In terms of selection/retrieval methodologies, one can distinguish between *formal specifications* (e.g., VDM, ESF-Rose), to *database retrieval* (Cactis of Arcadia [Tayl88], PCTE [Boud88]), to *Knowledge-based* forms of reasoning (Practitioner, LaSSIE) , to *hypertext style* browsing (e.g., DIF [GaSc87]), to *Information Retrieval* (e.g., Catalog, Proteus), finally to *Wide-Spectrum* approaches like the one followed in ISHYS [GaSc89].

Our conjecture and the motivation behind this work is that the software repository necessary to support reuse should be seen as a special-purpose information system providing a complete and effective functionality. Its development necessitates the harmonious coexistence and co-use of methodologies and techniques from a variety of areas, including (object-oriented) *databases, knowledge representation, hypermedia* and *information retrieval*.

Section 2 summarises the structure of the SIB, while Section 3 presents the functionality and the architectural principles of the system. An example illustrates the usage of the SIB in Section 4, followed by the implementation decisions and principles in Section 5. Our concluding remarks are made in Section 6. The interested reader can find more information about the SIB in [Vass90, Cons91, Cons93].

2. DESCRIPTION OF SOFTWARE

The SIB is structured as an attributed directed graph the nodes and links of which represent descriptions of software objects and semantic relations respectively. There are three kinds of descriptions:
(1) requirements descriptions (RD);
(2) design descriptions (DD); and
(3) implementation descriptions (ID).

These descriptions provide three corresponding views of a software object:
(1) an application view, according to a requirements specification model (e.g., SADT);

(2) a system view, according to a design specification model (e.g., ER or DFD); and
(3) an implementation view, according to an implementation model (e.g., set of C++ classes together with documentation).

Descriptions can be simple or composite, consisting of other descriptions. The term *descriptions* reflects the fact that these entities only describe software objects. The objects themselves reside outside the SIB. Descriptions are related to each other through a number of semantic relations listed below. In addition to the usual isA, instanceOf and attribute relations supported by object-oriented data models and knowledge representation schemes, several special attribute categories have been defined for the purposes of the SIB. The SIB is defined, as mentioned, in terms of the Telos knowledge representation language which supports creating an infinite instantiation hierarchy and treats attributes as objects in their own right (which, therefore, can also have attributes). These features of Telos are fully exploited in structuring the SIB.

The modelling constructs used in the SIB can be distinguished into four categories:

(I) *General structural/semantic relationships*, including attribution, classification and generalisation. These are the basic modelling mechanisms offered by the knowledge representation language.

(II) *Special structural/semantic relationships*, including aggregation, correspondence, similarity and genericity. These have been defined as a minimal set of system-supplied special descriptors for the purpose of software description.

(III) *User-defined and informal links*, including versioning, hypertext, etc. The attribute definition facility supported by TELOS can be used to form this set of links.

(IV) *Associations*. These are sets of descriptors along with private symbol tables, which allow for the construction of materialised views (through queries) and of contexts (or workspaces).

In what follows we present the above constructs in a sequence that facilitates the exposition, rather than dealing with each of the three groups in turn. The link representation referring to each relation is the one appearing on the user interface of the Graphical Browser of the SIB (see Figure 4.1).

(1) *Attribution*, represented by *attribute* links. This is a general, rather unconstrained representation of semantic relations, whereby the attributes of a description are defined to be instances of other descriptions. An attribute can have zero or more values.

Example:
```
Description SoftwareObject with
     attributes
          author: Person
          version: VersionNumber
```

`SoftwareObject` has attributes `author` and `version` whose values are instances of `Person` and `VersionNumber` respectively.

(2) *Aggregation*, represented by *hasPart* links. This relates an object to its components.

Example:
```
Description SoftwareObject with
 ...
```

```
    hasPart
        components: SoftwareObject
```

The components of an object have a distinct role in the function of the object and any possible changes to them affect the aggregate object as well (e.g., new version).

(3) *Classification* (opposite *instantiation*), represented by *instanceOf* links. Objects sharing common properties can be grouped into classes. An object can belong to more than one classes. Classes themselves are treated as generic objects, which their members are instances of, and which, in turn, can be instances of other, more generic objects. In fact, every SIB object has to be declared as an instance of at least one class. Effectively, an infinite classification hierarchy is established starting with objects that have no instances of their own, called tokens. Multiple instantiation is allowed. Instantiation of a class involves instantiating all the associated semantic relations. Thus relations are treated as objects themselves.

Example:

```
Description BankIS instanceOf SoftwareObject with
    author
        : Panos
    version
        : 0.1
    components
        : CustomerAccounts, Credit, Investments
```

The `attribute` and `hasPart` links of `BankIS` are instances of the corresponding `attribute` and `components` links of `SoftwareObject.`

Classification is perhaps the most important modelling mechanism in the SIB ([Veze91, PeVe91] give a detailed account of the construction of models and descriptions in the SIB).

(4) *Generalisation* (opposite *specialisation*), represented by *isA* links. This allows multiple, strict inheritance of properties between classes leading to the creation of multiple generalisation hierarchies. A class inherits all the attributes of its superclasses (possibly more than one - multiple inheritance), however inherited properties can only be constrained, not overridden (strict inheritance).

(5) *Correspondence*, represented by *correspondsTo* links. A software object can have zero or more associated requirements, design and implementation descriptions. Correspondence relations concern the identity of an object described by different descriptions and can have as parts other correspondence relations between parts of the corresponding descriptions. Correspondence links actually indicate that the descriptions they link describe the same object at different levels of abstraction. The correspondences of the parts need not be one-to-one. For instance, a requirements specification may correspond to more than one designs and a design may have more than one alternative implementations. Similarly, a single implementation could correspond to more than one design entities.

An important type of controlled correspondence in the SIB is the *Application Frame (AF)*. Application frames represent complete systems or families of systems and comprise (*hasPart*) at least one implementation and optional design and requirements descriptions. AFs are further distinguished into specific and generic (SAFs and GAFs) while the RDs, DDs and IDs of an AF should properly be considered as groupings of such descriptions (i.e., other associations.)

A SAF describes a complete system (be it a linear programming package, a text processor or an airline reservation system) and includes exactly one ID. A GAF is an abstraction of a collection of applications pertinent to a particular application domain and includes one RD, one or more DDs and one or more IDs for each DD. For an extensive presentation of the concepts and structure of the SIB, as well as the role of Application Frames in the Ithaca application development methodology, the interested reader is referred to [Vass90, Cons91, deAn92].
.sp (6) *Similarity*, represented by *similarTo* links. The similarity relation is defined between objects of the same kind (i.e., RD, DD, ID, or AF) and has two attributes: a similarity criterion and a similarity measure. Two objects are said to be similar with respect to some criterion if they can substitute one another with regard to this criterion. The similarity criterion can be endogenous defined in terms of relations already stored in the SIB, or exogenous, provided by the user who specifies a particular similarity link. Accordingly, similarity links can be computed or user-defined. The similarity measure expresses the degree to which the substitution of two similar objects, in either direction, is satisfactory and is a number in the range [0,1]. The similarity criterion is a mandatory attribute while the measure is optional.

(7) *Genericity*, represented by *specialCaseOf* links. This relation is defined only between application frames (see below) to denote that one application frame is less parameterized than another. E.g., a bank accounting and a hotel accounting application frame could both be derived from a more general, parametric accounting application frame.

(8) *Informal and user-defined links*. When users have foreseeable needs for other types of links they can define them using the attribute definition facility of TELOS. For instance, versioning can be modelled by special correspondence links labeled *derivedFrom*. Furthermore, random needs for representation and reference can be served by informal links, such as hypertext links which allow the attachment of multimedia annotations to SIB objects.

(9) *Association* is an encapsulation mechanism intended to allow the grouping of descriptions that play together a functional role [Brod84]. For example, we may define as an association the descriptions that constitute a design specification for a hotel information system, or all the classes that define an implementation of that same system. The contents of an association can only be accessed through the entry points defined in a private symbol table. Thus, an association is, actually, a tuple:

Association = (setOfDescriptions, symbolTable)

The SIB itself is a global association containing all objects included in any association. Its symbol table contains all the external names of every object. Name conflicts can be resolved by a precedence rule.

Associations can be derived from other associations through queries or set operations. Furthermore, associations can be considered as materialised views. Non-materialised views, or simply *views*, differ from associations in that they cannot be updated directly, but rather, through updates of the associations which they are derived from.

3. FUNCTIONALITY AND ARCHITECTURE OF THE SIB

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 adapt themselves to the type of object to be edited by reading out schema information from the database itself. There is no distinction between schema and data, giving the possibility to deal with a continuously growing number of classes and special cases, and to refine existing descriptions to more and more detail in a formal way. Selection functions include querying and browsing the SIB for purposes of selecting reusable artifacts and understanding their structure and functionality. Workspace management involves the dynamic definition and modification of workspaces to provide easier and more efficient interaction with the SIB.

Maintenance and selection operations are performed on *workspaces* which are application-specific and/or user-specific subsets of the SIB represented as associations. Between several workspaces, the notion of identity of shared objects is kept by reference to the identifiers in the global workspace. Classification and generalization links are also shared between the workspaces, being an integral part of the object's identity. Using workspaces carries several benefits, such as focusing attention to 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 (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 one generated as a result of navigational commands in the browsing mode, or one of particular significance and frequent occurence, "pre-canned" for ease of use and offered as a button or menu option. Browsing commands and explicit queries can also be issued through appropriate interfaces from external tools such as the Visual Scripting Tool of ITHACA [deMe91].

The selection of software descriptions from the SIB is accomplished through the *Selection Tool* (ST) by an iterative process comprising alternate stages of explicit queries (retrieval) and browsing. Browsing usually is the final and often the only stage of the process. The functional difference between the retrieval and the browsing mode is that the former supports the retrieval of an arbitrary subset of the SIB based on some user's knowledge about the SIB contents, while the latter supports local exploratory searches withing a given subset of the SIB without need of previous knowlegde. Operationally, both selection modes address queries to the SIB.

We now present the basic functions of the SIB in the manner of defining the operations on an abstract data type:

Maintenance functions:
Insert: Descriptions X Associations -> Associations
Delete: Identifiers X Associations -> Associations
Update: Descriptions X Associations -> Associations
NewVersion: Descriptions X 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 materialisation of the association. The delete function takes the name 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: Query X Associations -> SetOf (Descriptions X Weights)
Browse: Identifier X Links X Associations -> Views

The *Retrieve* function takes as input an association and a (compound) non-Boolean query and returns a subset of the association with weights attached indicating the degree to which the descriptions in the answer set match the query. In the current implementation only Boolean queries are supported, yet the extension to non-Boolean is currently undertaken.

Browsing is a special retrieval operation. It starts at a specified SIB description which is the focus of attention and is called *current object* and produces a view of a neighbourhood of interest of the current object within a given association. Since the SIB has a network structure, the neighbourhood of the current object (node) is defined in terms of the links of interest adjacent to it. As we shall later see, the size of the neighbourhood 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 of interest and an association, and determines a local view centred around the current object. This local view can be extended to the transitive closure of one or more relations (link classes), e.g. hasPart. By calling *Browse* again with the identifier argument equal to the name of one of the objects contained in the browser's view, that object is made current and the view is updated. Effectively, the *Browse* function provides a moving window with controllable filters and size, which allows navigational search over subsets of the SIB network. The default association is the entire SIB.

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, thus completely integrated in the SIB network model. Hypermedia annotations include text, graphics, raster images and algorithm animations.

The SIB system consists of the following major parts (Figure 3.1):

(1) The *SIB Interactive User Interface* generates and coordinates the other parts, including the interface tools of the Data Entry Forms and the Selection Tool, except for the Graphical Browser. It is implemented using the OSF/Motif toolkit.

(2) The *Graphical Browser* presents parts of the SIB network graphically and allows the user to browse through it by sending messages to the SIB Interactive User Interface in response to user actions. The Graphical Browser is a LABY graphical editor with only the working area present. LABY is a general purpose graphical editor developed in part within the ITHACA project at the Institute of Computer Science, FORTH [Kate90].

(3) The *Display Forms* are used to provide information about the current node or another selected node in a form layout. They are designed to support multimedia information (text, graphics, images, animation, etc.).

(4) A special *Data Entry Form* provides for entering data into the SIB through a form interface.

(5) The *Query Interface* handles queries about SIB objects, issued to it by the various components of the ST or by the Data Entry Form. As a result of processing a query it constructs a file suitable for display by the Graphical Browser or the Data Entry 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 ST user interface consists of the following windows (Figure 4.1):

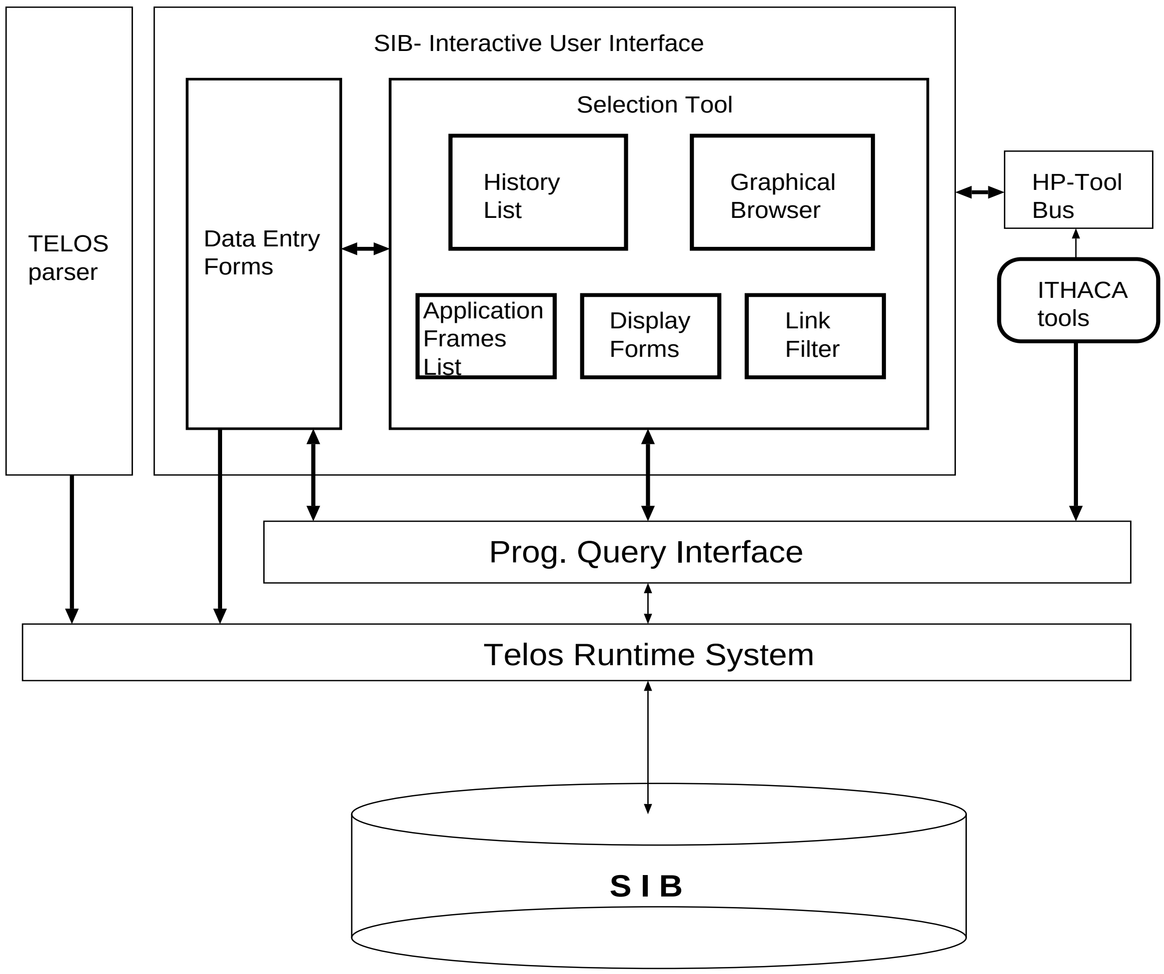

Figure 3.1. SIB architecture

(1) *Graphical Browser*:
The Graphical Browser, built using the LABY graphical editor,
displays a part of the SIB network around a selected object (*current object*). The structures currently displayed have the form of a star whose central node is the current object. The window of the Graphical Browser is semantically divided in two parts. From each node appearing in the lower part emanates at least one link pointing to the current node. Similarly, 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 denoted by a colour code. For example the isA relation is shown with red links while the instanceOf relation with green links. The colour code is shown in the Link Filter. Nodes appearing in the graph of the browser are selectable with the mouse. When selected, a node becomes current, it is placed in the middle of the display and a move in the SIB network results. The links displayed include direct links from the current object to 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 class (i.e. selected by the Link Filter) 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 (Figure 4.2). The objects on this list are selectable just as those displayed graphically.

(2) *Link Filter*:
The Link Filter provides buttons corresponding to link classes and is used for activating/deactivating links thus controlling the information displayed in the Graphical Browser. Each button acts as a filter on a certain relation. If the button is selected (on), the corresponding relation is displayed and vice versa. The isA and instanceOf buttons further offer the option of displaying computed in addition to direct isA and instanceOf links. All buttons show the colour code of the link classes and have a help facility. When moving from node to node in the Graphical Browser, the state of the Link Filter does not change unless the user does so explicitly.

(3) *History List*:
The History List is a navigation aid intended to prevent users from getting lost, a common problem in hypertext systems [Conk87]. 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 in 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.

(4) *Application Frames List*:
The Application Frames List contains the names of the application frames, reflecting the overall structure and contents of the SIB. The purpose of the Application Frames List is to compensate for the limited scope of the Graphical Browser, which is a shortcoming if an extended area of interest is sought by navigation rather than by explicit querying. The application frames are displayed in an indented list representing the existing hierarchical structure. Each item on the list is selectable, which effectively allows big steps over the SIB network in the browsing mode. Moreover, the Application Frames List serves as the initial entry point to the ST.

(5) *Main Form*:
The Main Form is the top right-most window of the ST and invariably displays information about the current object. It shows the abstracted definition of the object in a form layout. The information presented is generated by unparsing the SIB. The form can also contain multimedia annotations to the object, each one displayed in a separate window. At present, this annotation is textual and graphical (Figure 4.3), but can be of any other type with no additional effort, provided that appropriate tools exist in the working environment.

(6) *Auxiliary Form*:
To get information about a displayed object, other than the current one, without changing the actual view in the Graphical Browser, the user can select the name of the desired object on the Main Form in a hypertext-like fashion (see Figure 4.1). An Auxiliary Form will appear at the bottom right of the ST and display information about the selected object in the manner of the Main Form. To prevent user distraction, only one auxiliary form can be open at a time; also objects are not selectable on auxiliary forms. The auxiliary form is actually a preview mechanism and is offered as an orientation aid.

(7) *Button Panel*:
Through the Button Panel, several other windows for performing various 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 (Figure 4.4). The Query Form, currently under construction, will have a similar appearance.
KeepObject: Keeps a retrieved object in a local workspace.
Iconify and *Quit*: Closes and iconifies a window; and quits the ST respectively.

4. USAGE EXAMPLE

Suppose we would like to include in the SIB an application dealing with processing of letters, which will assist secretaries, managers, and others to write professional letters, check them, and, after final approval, mail them through post or electronic mail. Looking at the Application Frames List, we observe that there is already an Office Information System called `WooRKS.` The basic concepts in `WooRKS` are actors, roles and procedures.

Our starting point will be `WooRKS`, which we select through the Application Frames List. As we can read in the natural language comment attached to the corresponding Main Form, `WooRKS` is a work flow system for offices, which handles a variety of activities (see Figure 4.1). So this might be one of the candidate places to search for a letter processing application. `WooRKS` has three attributes which are further explained in its Main Form. One of them, *reqDescr*, deals with `WooRKS` requirements descriptions and it will probably provide us with more information about what the `WooRKS` system actually does. Before making it current, we preview its contents on the Auxiliary Form, by selecting `WooRKS1_RD_FORM` from the Main Form, and we decide to visit it.

In Figure 4.2 `WooRKS1_RD_FORM` is current in the Graphical Browser window. Among the *classes* of activities that it handles, `OrderProcessing` appears to be the closest such as checking and archiving. We decide to visit `Order Processing` to see if it includes what is needed for letter processing in general. After we conclude that this is not true. We backtrack to `WooRKS1_RD_FORM` through the History List and decide to preview `WarehouseProcessing` and `AccountProcessing` to see if there is anything relevant there. By previewing these nodes we realise that none of them fullfils our needs. We further notice that they are all instances of `FormProcessClass`.

We take a closer look on `FormProcessClass` by moving to it through the GotoObject facility (Figure 4.3). As `FormProcessClass` is a subclass of `FormClass`, it inherits its attributes. By previewing `FormClass`, we find out that it has two attributes, *roles* and *baseRole* (see Figure 4.3) and decide to create a new instance of `FormProcessClass`, called `LetterProcessing`, whose *roles* will correspond to the initial requirements imposed on our letter processing application. In particular, the *baseRole* of `LetterProcessing` will be `sp_base_role`, and the *roles* will be `sp_letterCompose`, `LP_letterCheck`, `sp_letterApprove`, `LP_letterSend`, `LP_letterReceive`, and `sp_letterArchive`.

We have chosen this convention for naming the *roles* by analogy to the existing *roles* of the other activities. To see these names we first made `FormRole` current using the GotoObject facility (see Figure 4.4). Since `FormRole` has too many instances to be displayed on the Graphical Browser, an Instance Box appears by clicking on the "MANY INSTANCES" box of the Browser.

At this point we start creating the *roles* and *baseRole* of `LetterProcessing`. Before creating `sp_letterArchive`, we visit `OP_orderArchive` by selecting it from the Instance Box (Figure 4.4). This act is worthwhile because we find a *correspondsTo* link from `OP_orderArchive` to `ArchiveAct`, which is an instance of `ADMActivity` (see the Main Form in Figure 4.5). Knowing that `ADMActivity` handles the design descriptions, we can further proceed by defining the ADMActivity corresponding to the new sp_letterArchive in a similar way, or even use the ArchiveAct as the ADMActivity of sp_letterArchive. Similarly, we may use CompileRefAct and/or EvaluationOrderAct which *correspondTo* OP_orderCheck as the ADMActivity of sp_letterCheck, etc.

Finally, we are in a position to define the new LetterProcessing node. We move to FormProcessClass using the GotoObject option and use the EnterData facility, which will make our task easy, even if we have no knowledge of the syntax of Telos. The Data Entry Form for LetterProcessing is shown in Figure 4.6. In the same figure you can see the results of entering the information.

5. IMPLEMENTATION

The SIB is an essential component of a complete application development environment, interfaced cleanly with several tools and the object store as shown in Figure 5.1. A zoom at the runtime system is made in Figure 5.2

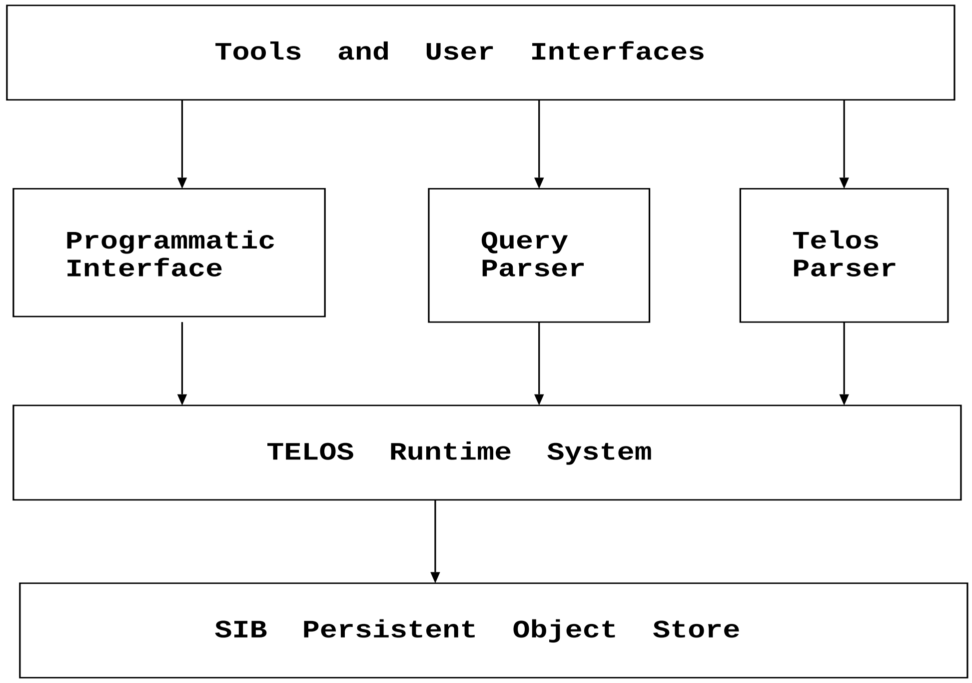

Figure 5.1: Interfaces of the TELOS Runtime with Tools and the Object Store

The TELOS data model and system without its time reasoning mechanism and its assertional language was used for the construction of the SIB. TELOS was chosen for its powerful representation and modelling facilities, as well as for superior flexibility and efficiency over other database systems (relational or object-oriented), as indicated by early experimentation. The current implementation of TELOS, all using C++ as opposed to the earlier Prolog implementation, specifically attends to the needs of the SIB.

Architecturally, the system contains all main components of a typical object-oriented DBMS implementation (see Figure 5.2), but low-level optimizations and data structures are rather different. All data fields are set-valued. Stress is laid on efficient handling of network-like structures and fast retrieval of transitive closures with respect to certain link

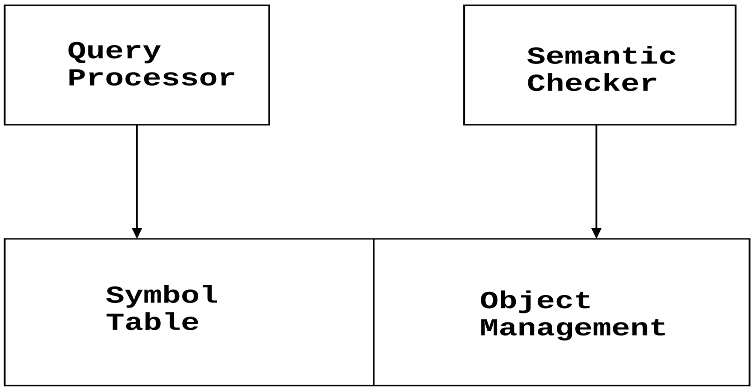

Figure 5.2: Structure of the TELOS Runtime system

types. No optimization is done for range queries on primitive values (integers, floats etc.). The schema is completely kept at data-level, which allows fast schema extensions at run-time. The data of an information system are in general rather static, with infrequent and bulk changes, which leads to a strong preference of query optimization against update optimization.

TELOS objects identify themselves to the user by logical names, which are the links to the real world objects they describe. Internally they carry a unique system identifier [KhCo86, KiKi89] which is invisible to the outside. The TELOS logical names are kept in symbol tables, one for each association, which establish mappings between system identifiers and logical names. The symbol tables may map arbitrary subsets of the database, with the only restriction that the database must be semantically consistent within each association, and on system identifier level as a whole [WiAl88]. There are no restrictions on the logical names one system identifier is mapped to in different symbol tables, but the user will be enabled to check under which name an object within one association will appear in another.

Five C++ classes are used for representing the TELOS objects. The contents of the five classes are sets of system identifiers for the *instanceOf*, *isA*, *attribute* and *trigger* relations. All these relations are kept in both directions to allow very fast query processing.

Triggers are either built-in or user-supplied, in which case they must be linked to the runtime system. They allow access even to data outside the database, thus providing a very powerful interface mechanism.

The TELOS objects are kept persistent on disk and copied to memory in a cache-like way. We implement demand loading in the first step, which will later be replaced by a prefetch mechanism [Low88]. During one transaction, all modifications done on existing objects and newly created objects are kept in memory only. They become persistent at the end of the transaction. Queries may be allowed on these objects before end of transaction under certain restrictions. All dynamic memory allocations are done on a fixed granularity base, first to reduce allocation time and, still more important, to avoid cluttering the virtual address space after longer execution times. Similar issues have been discussed in [KhFr88, MeLa87, SkZd86] too.

The association of system identifiers with object locations in 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. As system identifiers are only internal, there are no identity conflicts as with other database identifiers. Density allows a virtual memory - like indexing of the system catalogue with use of page tables. Requiring an additional indirection step after a growth of the database by a factor of 1000, this design leads to nearly size-independent performance.

Symbol tables translate from logical names to system identifiers and vice-versa. The symbol table tuples are organized as the system catalogue, giving the same high performance for system identifier to logical name translation. The translation from logical names to system identifiers is supported by B-Trees, which are the only element introducing some degradation of performance with growing database size. Fortunately, the frequency of the translation from logical names to system identifiers is roughly an order of magnitude lower than that of the inverse. The reason is that query answer sets are usually larger than the respective argument sets, and all internal query processing is done on the basis of system identifiers. Symbol tables as well as the system catalogue are cached.

Concurrency control in network-like structures poses particular problems, as database parts to be locked are hard to determine. Under the assumption that the frequency of updates is low, only read and write locks for the whole database are provided. This implies that locks may be held only for short time intervals. Interactive data entry forms must check consistency when the data are actually to be committed to the database. Cache invalidation is done at lock grant on demand of each server instance, which prevent degradation of performance as the number of clients increases. The lock mechanism works reliably on local area networks.

Emphasis in the implementation is placed on *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. Porting to IBM Risc machines is currently undertaken. The X window system is required, preferably with a colour monitor. The system, based on a client-server architecture, may run in a local area network. 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. Concurrent access is foreseen for about eight to ten users (members of a software development team) at a time. *Scalability* and *size-independent performance* have been the main principles of the implementation. This is strengthened from the fact that for an information base, the more items it contains - the more useful it becomes.

6. CONCLUSION

We presented the Software Information Base, a reuse-targeted repository system, and first experiences from the development of a high-performance prototype. While this is a continuing effort, some conclusions can be drawn.

Pivotal role in being able to locate software components is played by the rich organisation facilities offered by a proper (meta-)modelling system (e.g., several extensible semantic/structural/referential links which are bi-directional and machine-supported). This should be coupled by graphical and others aids for the user interaction and component comprehension. In terms of implementation, no single technology is sufficient; a harmonious co-use of techniques from several areas is being found effective. Prime role among those is played by the technologies of databases, knowledge representation, hypermedia and information retrieval.

This paper concentrated on presenting functionality and implementation aspects of the SIB system. An empirical evaluation of the SIB model and system in the context of a specific methodology developed in the project ITHACA is reported in [Cons93]. Moreover, we have started evaluating the model with other object-oriented analysis and design methodologies (e.g., the Booch design methodology).

REFERENCES

[Ader90] M. Ader, O. Nierstrasz, S. McMahon, G. Mueller, A-K. Proefrock, The ITHACA Technology: A Landscape for Object-Oriented Application Development, *Proc. ESPRIT'90 Conf.*, Kluwer Academic Publishers, November 1990.

[Bigg87] Biggerstaff, T. et al., "Information Management Challenges in the Software Design Process", *IEEE Database Engineering*, Vol 10, March 1987.

[Bigg89] T. J. Biggerstaff, A. J. Perlis, *Software Reusability, Volume I: Concepts and Models, Volume 2: Applications and Experience*, Addison-Wesley, Reading, Massachusetts, 1989.

[Boud88] Boudier, G., et al., "An overview of PCTE and PCTE+", Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM Software Engineering Notes, Vol. 13, No. 5, pp. 248-257, November 1988.

[Brod84] 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*, Springer-Verlag, 1984.

[Conk87] Conklin J., "Hypertext: An Introduction and Survey", *IEEE Computer*, Sept. 1987.

[Cons91] Constantopoulos, P., et al., The Software Information Base - Selection Tool Integrated Prototype, ITHACA.FORTH.91.E2.#3, Institute of Computer Science, Foundation of Research and Technology - Hellas, January 1991.

[Cons93] Constantopoulos, P., Jarke, M., Mylopoulos, J., and Vassiliou, Y., The Software Information Base: A Server for Reuse, Technical Report, Institute of Computer Science, Foundation of Research and Technology - Hellas, Heraklion, Crete, March 1993 (submitted for publication).

[deAn92] De Antonellis, et al, Ithaca Object-Oriented Methodology Manual, ITHACA. POLIMI-UDUNIV.E.8.6, Politecnico di Milano, 1992.

[deMe91] de Mey, V. et al., "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.

[Deva91] Devanbu, P., et al., "LaSSIE: A Knowledge-Based Software Information System", *Communications of the ACM*, May 1991.

[GaSc87] P. Garg, W. Scacchi, "On Designing Intelligent Hypertext Systems for Information Management in Software Engineering, DIF", Proceedings of Hypertext 87, pp. 409-431, November 1987

[GaSc89] P. Garg, W. Scacchi, "ISHYS: Designing an Intelligent Software Hypertext System", IEEE Expert, pp.52-62, 1989

[Hopk88] T. Hopking, C. Phillips, *Numerical methods in practice: using the NAG library*, Addison Wesley, New York, 1988.

[Kate90] Katevenis, M., et al., LABY User's Manual, Version 2.10, ITHACA.FORTH.90.E.3.3.#7, Institute of Computer Science, Foundation of research and Technology - Hellas, 1990.

[KhCo86] Khosafian S., Copeland G.P., "Object Identity", *ACM OOPSLA '86*.

[KhFr88] Khosafian S., Frank D., "Implementation Techniques for Object Oriented Databases", *Proc. AOODS*, Springer 1988.

[KiKi89] Kim, W., Kim, K.C., and Dale, A., "Indexing Techniques for Object Oriented Databases", in: *Object-Oriented Concepts, Databases, and Applications*, ACM Press New York 1989.

[Low88] Low C., "A Shared, Persistent Object Store", *Proc. ECOOP'88*,1988.

[MeLa87] Merrow T., Laursen J., "A Pragmatic System for Shared Persistent Objects", Proc. *OOPSLA'87*, 1987.

[Meye87] B.Meyer, Reusability: The Case for Object-Oriented Design, *IEEE Software*, March 1987.

[Meye90] B. Meyer, *Eiffel: the libraries*, Prentice-Hall, New York 1990.

[Mylo90] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis, Telos: Representing Knowledge About Information Systems, *ACM Transactions on Information Systems*, p.p. 325-362, October 1990.

[PeVe91] Petra, E., and Vezerides, C., SIB Contents' Manual, ITHACA.FORTH.91.E.2.#4, Institute of Computer Science, Foundation of Research and Technology - Hellas, January 1991.

[Prie91] Prieto-Diaz, R., "Implementing Faceted Classification for Software Reuse", *Communications of the ACM*, May 1991.

[SkZd86] Skarra A.H., Zdonik S.B., "The Management of Changing Types in an Object-Oriented Database", *Proc. OOPSLA'86*, 1986.

[Tayl88] Taylor et al, "Foundations for the Arcadia Environment Architecture", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engin. Symposium on Practical Software Development Environments, ACM Software Engineering Notes, vol.13, no.5, pp.1-13, November 1988.

[Veze92] Vezerides, C., The organization of a Software Information Base for software reuse by a community of programmers, Master's Thesis, Department of Computer Science, University of Crete, May 1992.

[WiAl88] Wile D.S., Allard D.G., Worlds: Aggregates for Object Bases, USC/Information Sciences Institute, Marina del Rey,CA 90292, 1988.

[Vass90] Vassiliou, Y., et al., Technical Description of the SIB, Technical Report, Institute of Computer Science, Foundation of Research and Technology - Hellas, Heraklion, Crete, January 1990.

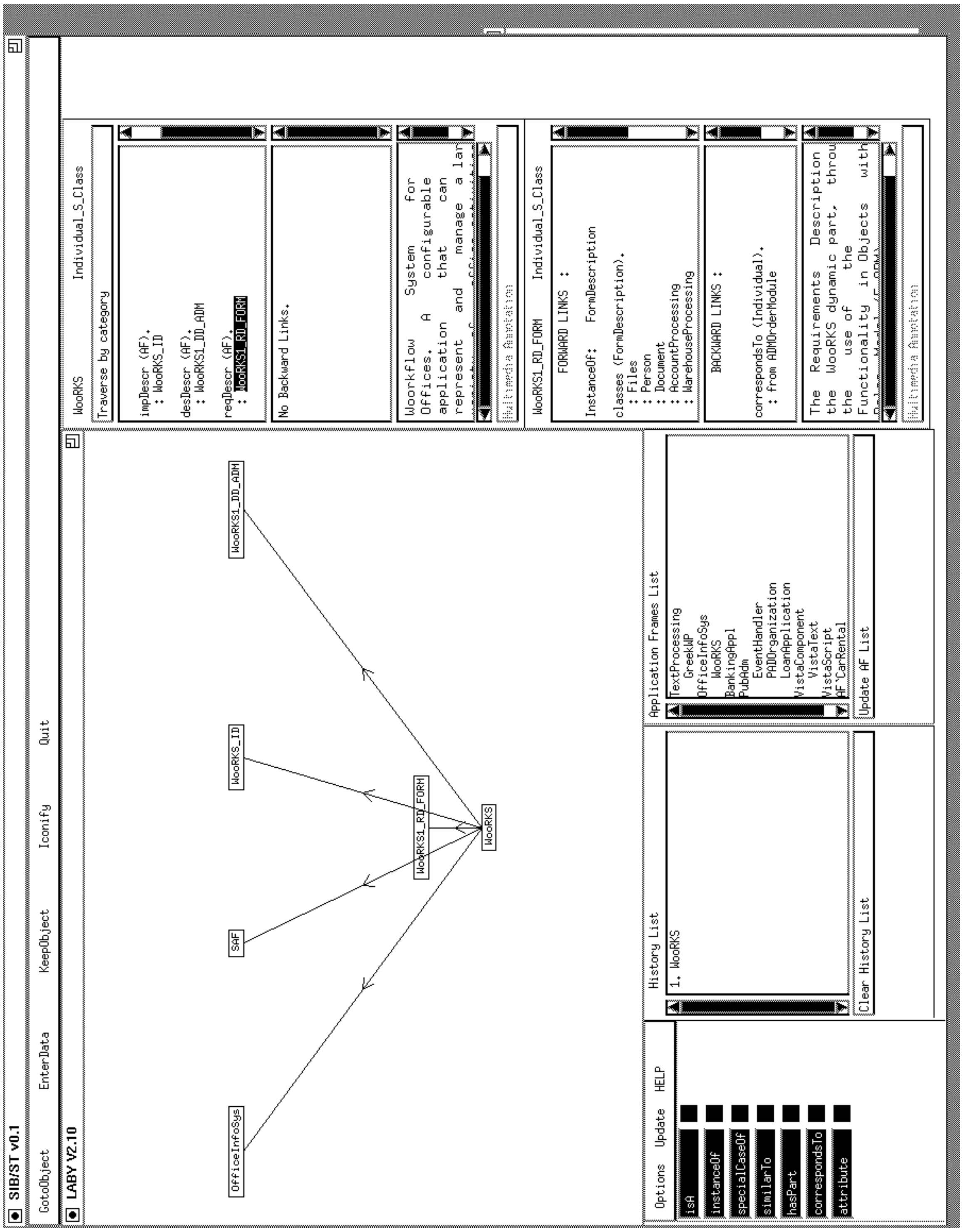

Figure 4.1. : The Selection Tool of the Software Information Base. Current in the Graphical Browser is WooRKS, the starting point of the usage example.

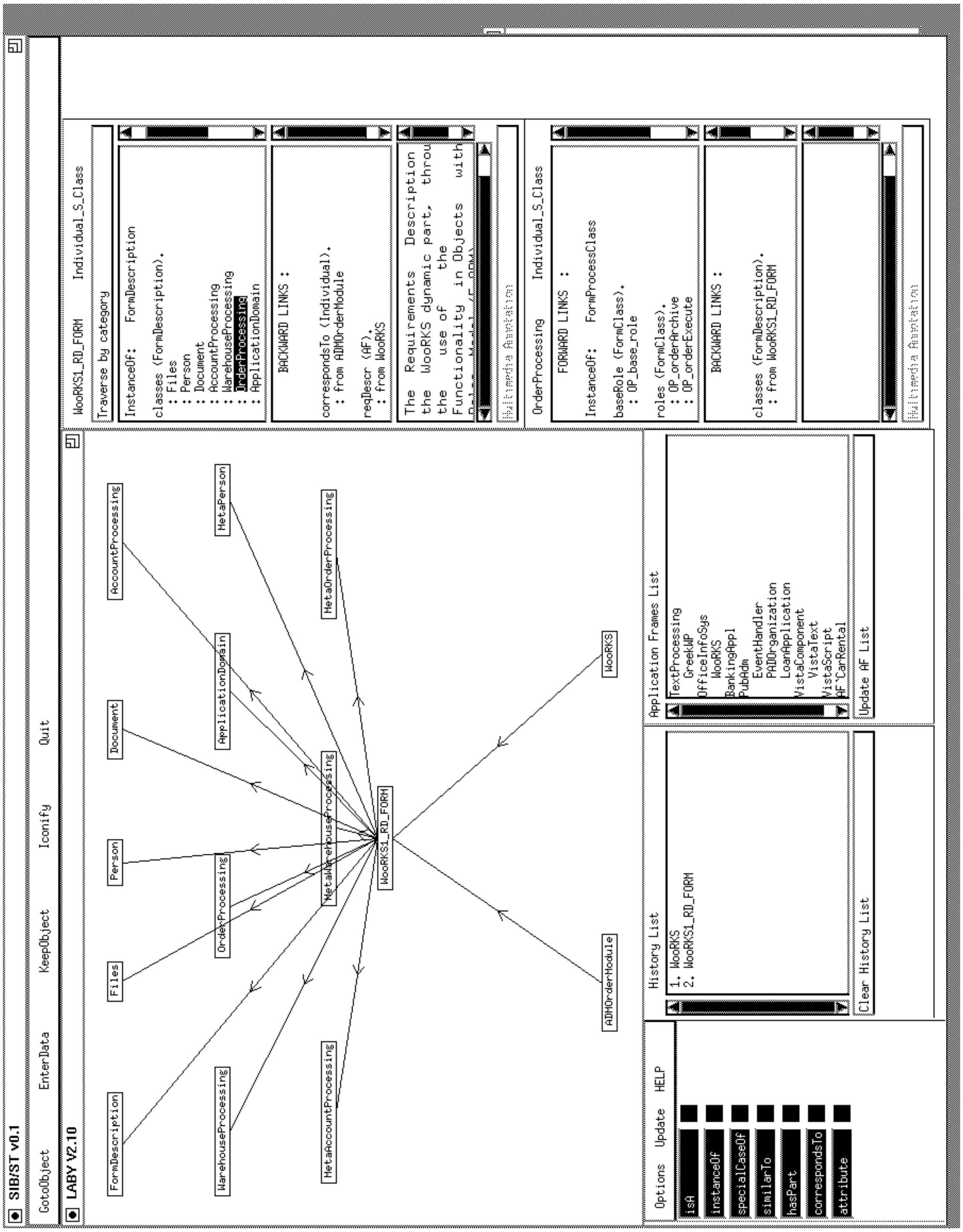

Figure 4.2. : Inspecting the requirements of WooRKS and previewing OrderProcessing.

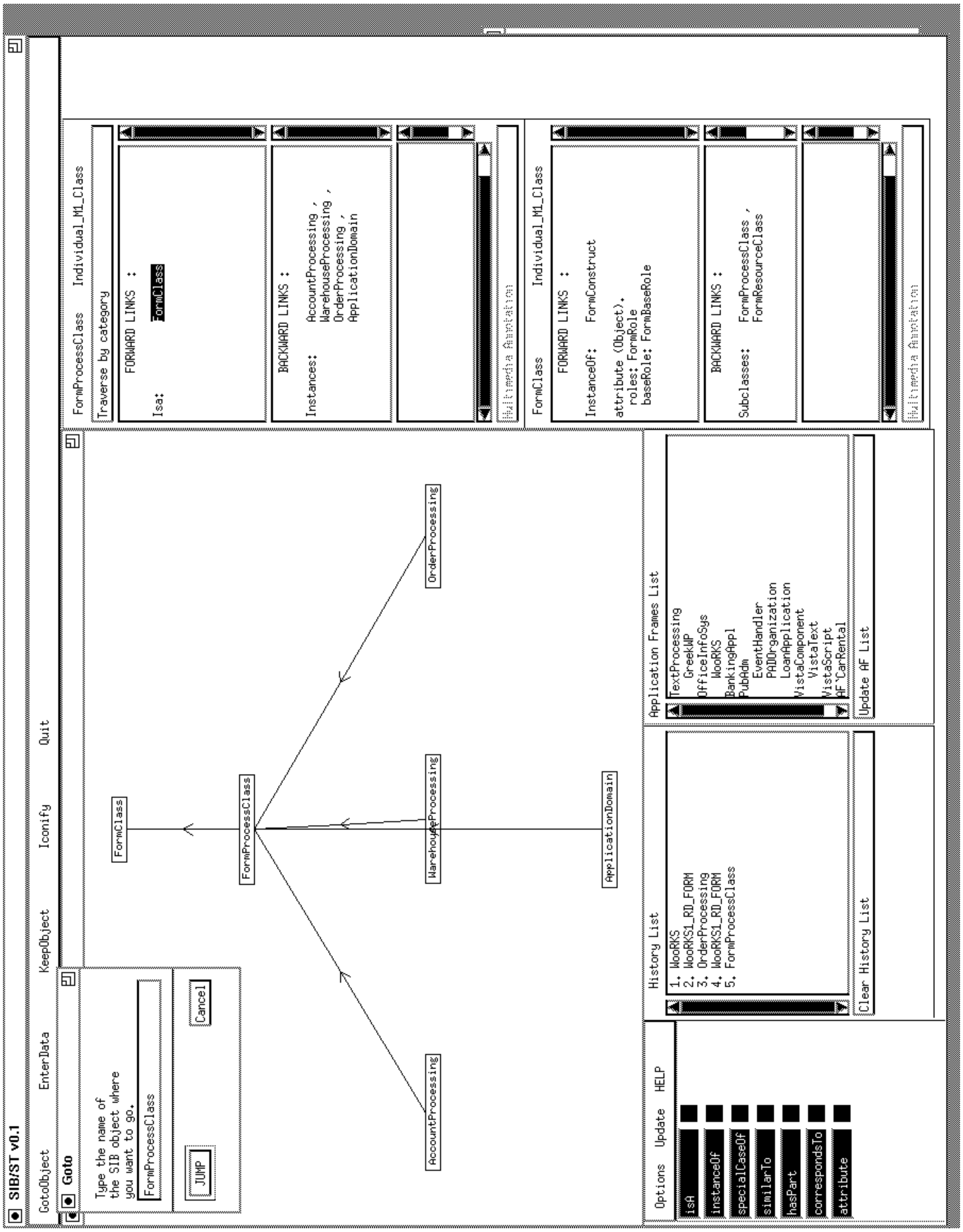

Figure 4.3. : Visiting FormProcessClass and previewing its superclass, FormClass.

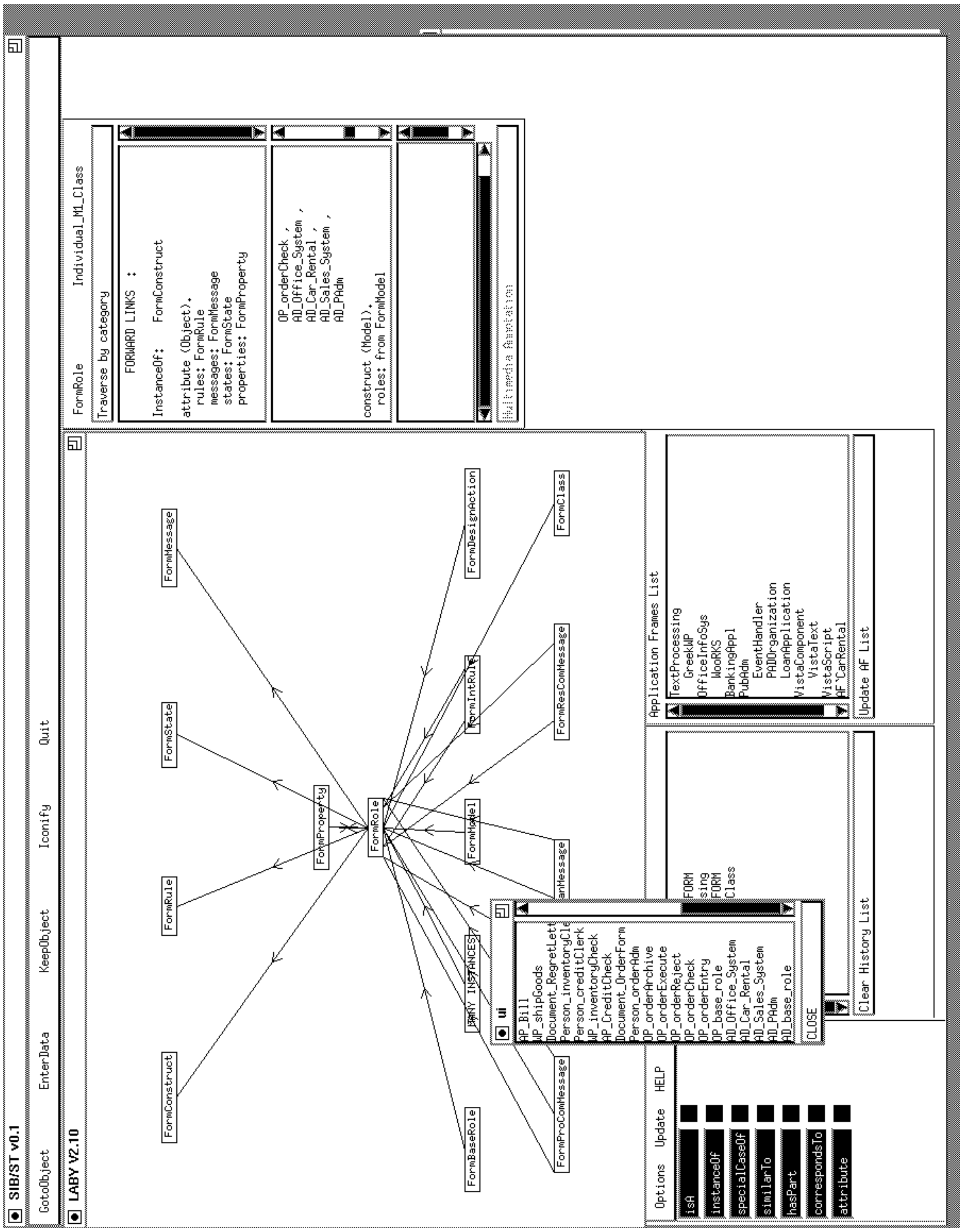

Figure 4.4. : The Instance Box handles the instances of FormRole.

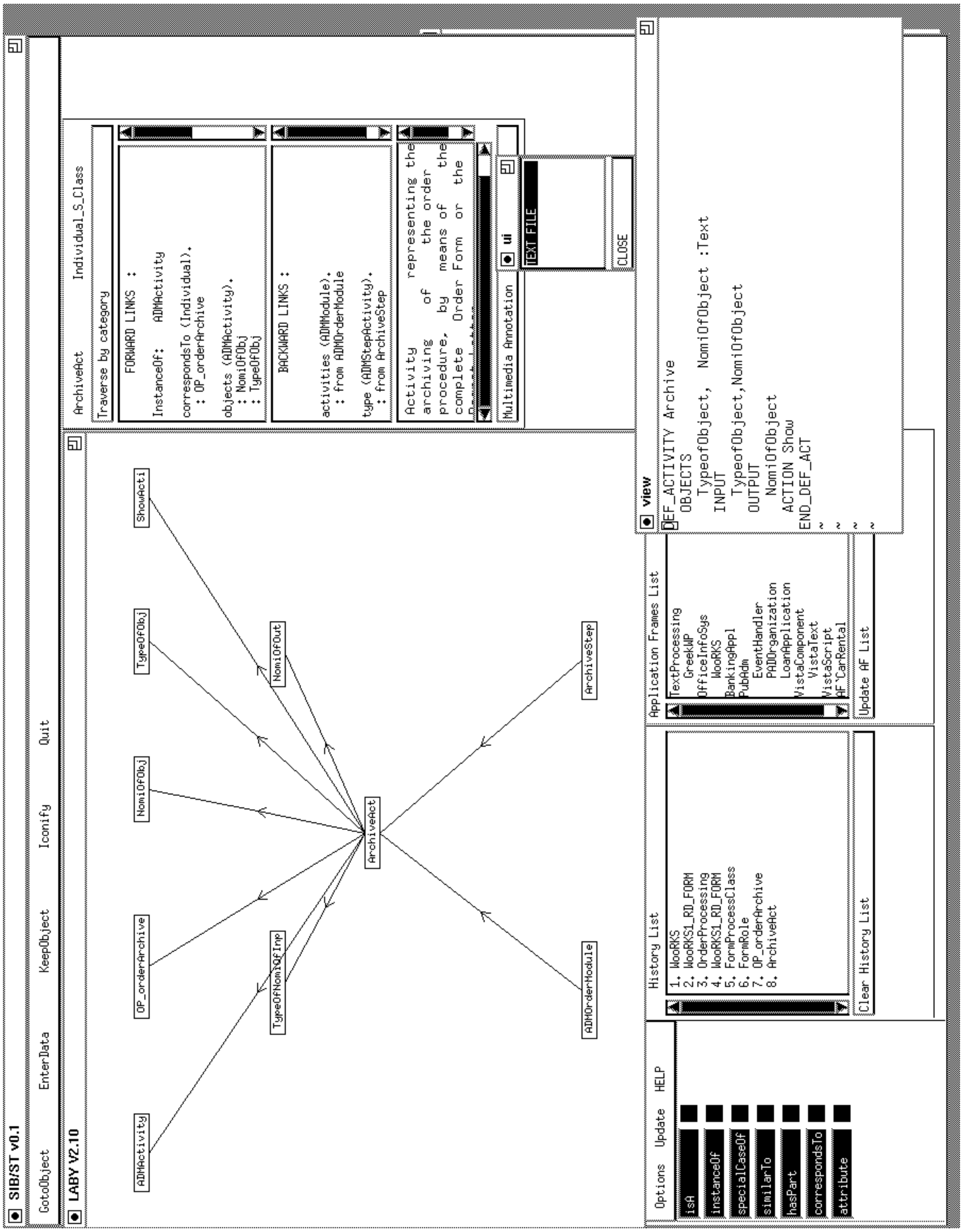

Figure 4.5. : An annotation attached to ArchiveAct.

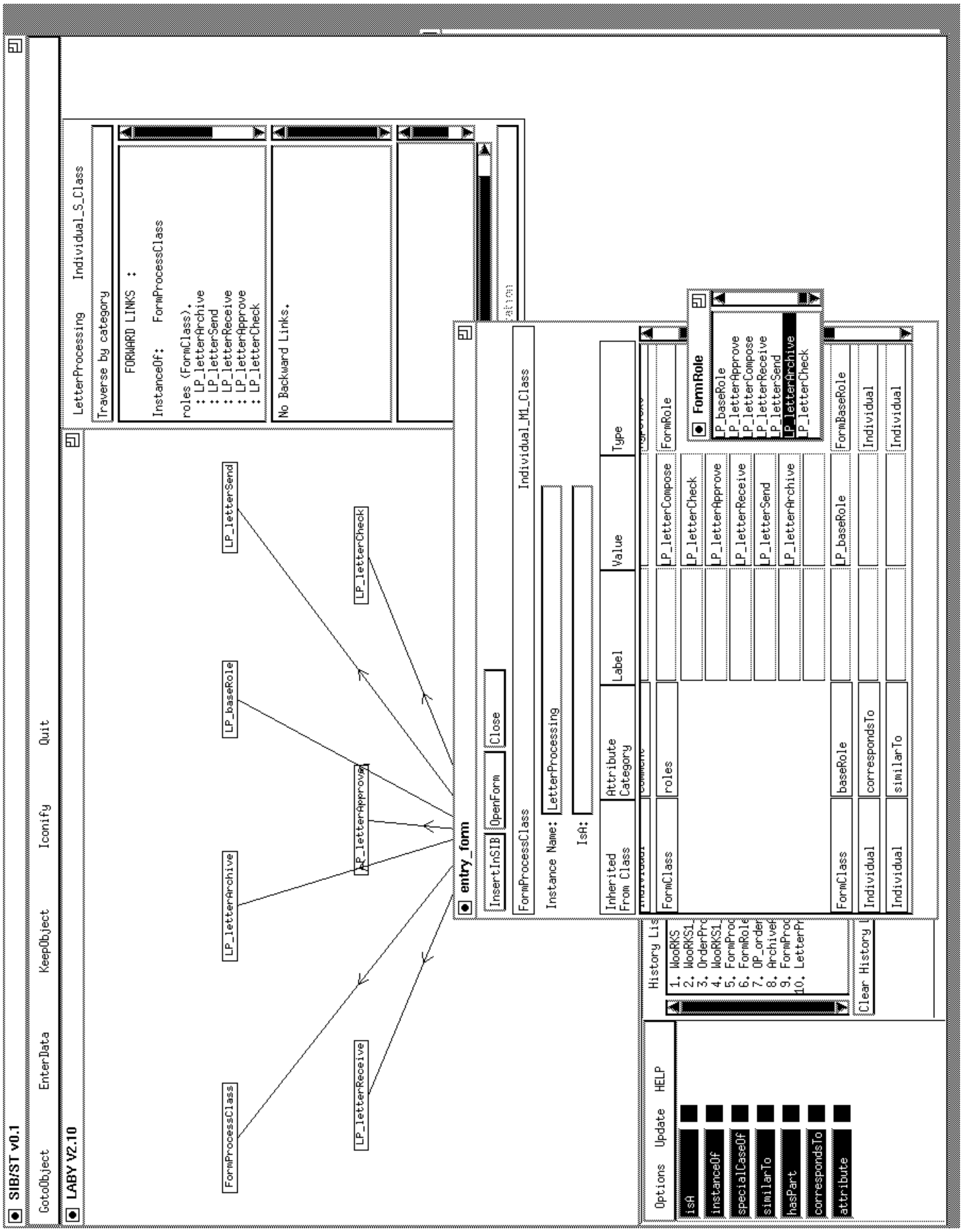

Figure 4.6. : The EnterData facility used to insert LetterProcessing in the SIB.