The ITHACA Technology: A Landscape for Object-Oriented Application Development[1]

Martin Ader — Bull S.A.
Oscar Nierstrasz — Université de Genéve
Stephen McMahon, Gerhard Müller, Anna-Kristin Pröfrock —
Siemens Nixdorf Informationssysteme AG

Abstract

The ITHACA environment[2] offers an application support system which incorporates advanced technologies in the fields of object-oriented programming in general and programming languages, database technologies, user interface systems and software development tools in particular. ITHACA provides an integrated and open-ended toolkit which exploits the benefits of object-oriented technologies for promoting reusability, tailorability and integratability, factors which are crucial for ensuring software quality and productivity. Industrial applications from the fields of office automation, public administration, finance/insurance and chemical engineering are developed in parallel and used to evaluate the suitability of the system.

1. Introduction

The ITHACA project aims to build a development environment for the construction of large-scale applications. Compared with the current situation, the applications are expected to be more reliable and more convenient to handle and to run. The manner in which they interact will, on the whole, be more integrated. At the same time, it is necessary to considerably reduce development costs, minimise the overall development risk and ensure the maintainability of the resulting applications. Moreover, applications built within the environment must be extensible with only a moderate amount of effort, thus allowing them to be adapted to meet changing requirements. This latter point is probably the major drawback of current hardwired application systems. The application domains we envisage in the ITHACA context belong to the domain of information systems, office systems and C* systems (CAD, CAM, CIM, CASE etc.).

It is widely accepted that the object-oriented paradigm promises best to be a basis for meeting the above requirements [Tsichritzis and Nierstrasz '88]. In spite of this, however, object-oriented programming is not yet used for the industrial development of systems in the foreseen domains. To reason this neglect we may draw from our experience in putting object-oriented programming environments to practical use and may identify some major lacks of present-day environments [Müller and Pröfrock '89].

1. Proceedings Esprit 1990 Conference, Kluwer Academic Publishers, Dordrecht, NL, pp. 31-51, 1990.

2. The ITHACA project (Esprit 2705) is funded as a Technology Integration Project by the Commission of the European Communities as part of the ESPRIT II programme. The partners involved are Siemens Nixdorf Informationssysteme AG (Germany), the prime contractor, with the associated partners Trinity College Dublin, University of Zurich, Altaïr (France) and SQL Datenbanksysteme GmbH (Germany); Bull S.A. with the associated partners Delphi S.p.A. (Italy), INRIA (France) and CMSU (Greece); Datamont S.p.A. (Italy) with Politecnico di Milano and Universitá di Milano as subcontractors; TAO (Spain); F.O.R.T.H. (Greece) and the University of Geneva (Switzerland).

The reasons for this are manifold. First, object-oriented programming is not supported by any adequate life-cycle model. Existing life-cycle models are insufficient for they do not support any notion of reusability - the core advantage of the object-oriented paradigm. Second (and most probably due to the missing life cycle model), tools for supporting the early phases of object-oriented programming are still in their infancy. Finally, object-oriented programming does not support any notion of persistence and thus limits the use of object-oriented programming to the development of system software, tools and user interface systems.

The ITHACA project aims to contribute towards providing solutions for these problem areas. The first section of this paper gives an integrated overview of the entire system and the approach adopted. This is followed by a more detailed description of the persistent programming and storage environment (HooDS). Subsequently, an appropriate object-oriented development life-cycle is introduced together with the tools required to support this. A brief description of selected pilot applications follows, including an explanation of end-user assistance and the support provided by groupware for cooperative processes. We conclude by outlining the minimum hardware and software platform on which the system runs and describe the status of development work so far.

2. The ITHACA Approach

As mentioned above, the objective of ITHACA is to provide an environment for supporting the development of large-scale application software systems [Pröfrock et al. '90]. The general approach to large-scale system design and development is to move the emphasis towards engineering rather than programming. To do so, we focus on the reuse of components on a macro-scale, a technique which is understood well and which is common practice in traditional engineering sectors, but not in software engineering, despite its name.

Complexity in engineering design is managed by hierarchical decomposition. This entails the repeated application of the following design cycle: Specifications are laid down and then, using knowledge of existing lower-level components, their properties and possible methods of interconnection, a configuration is proposed. This configuration usually undergoes a number of improvements in order to meet the specifications and criteria which may have been omitted or unquantified intially. The initial configuration together with subsequent improvements includes two major steps. First, the choice of component *types* and their layout and interconnection (topology of the design). Second, the choice of *specific* components (in terms of characteristic parameter values) to meet these specifications. We have assumed that the design is composed of available components. However, there may be a need for some *new* component(s). In this case, a design cycle at a lower level will be initiated to provide the missing component (or components), configuration of which again being attempted using existing components at a lower level still. For any given class of designs, systems and subsystems are constructed using a defined set of building blocks, the primitive components. The primitive components are not universal: gates are the basic building blocks for some designs, although they are themselves configured from more elementary components. At lower design levels, the primitive components are usually standardised. As a field of application matures, standardisation propagates upwards until it reaches final products.

It should be noted that the described approach to engineering is quite different from the stepwise refinement offered by software engineering. Stepwise refinement leads the software engineer to split a complex problem into less complex subproblems by means of iteration. Reuse of existing components is not considered to be a goal of the process and, if at all, is achieved by accident. In general, stepwise refinement results in recoding of similar programs.

Another, more intuitive example can be drawn from an architect's approach to building a house for a specific client. First of all, the architect draws up details of the client's requirements and the budget restrictions. Based on those conditions, the architect decides if he can offer his client a prefab. If not, he has to make a customised design in the manner described above. The more he cannot use standard components, the client's budget is burdened exponentially. He will *thus try to find standard kitchen designs, common bathroom layouts etc. in order to remain* within his budget limitations. Such a "generic kitchen frame" can be specialised by concrete objects which are differentiated with regard to material, colour, form and quality.

Object-oriented programming allows this approach to be followed because it adheres to the philosophy of using prefabricated components. Here, the term "component" covers more than just basic code; it also includes the context in which a piece of software is used. Thus, reuse of components also means the reuse of designs, models, requirements and even experience.

The following scenario can be used to apply the principles of engineering design to software engineering. Well-engineered components are described in catalogs and stored in software information bases in the form of application frames. An application frame is a prefabricated application and constitutes a network which links the application's model, requirements, design, experience, realisation and documentation. The application developer gathers the client's requirements. He navigates through his prefabricated applications with the aim of finding either a solution which satisfies the demands or pieces which fit into a specific layout. This cycle is applied recursively until no specific layout is left. In addition to the support of object-oriented programming and application configuration, the developer is assisted by means of dedicated construction tools (e.g. for user interfaces) and high-level system components (e.g. adequate database systems).

In order to support a manufacturing approach to software development, ITHACA provides an application support system which places an additional, object-oriented layer between the application and the underlying operating system (Figure 1). For the purpose of application programming, ITHACA provides a persistent programming and storage environment which serves as the kernel of the system. This kernel, called HooDS, includes an object-oriented database system, adequate programming languages and the appropriate basic programming environment. In addition, ITHACA presupposes an object-oriented design style. This design style is supported by dedicated tools for engineering and configuring applications. These tools are based on a software information base with appropriate navigation facilities. For end-user support, ITHACA offers a Motif-based user interface and groupware for assisting cooperative work between end-users. In order to have an early evaluation of the technology provided and to allow user participation from a very early stage onwards, four complex "real-world" applications are being designed and developed within the ITHACA framework. In addition to spanning the scope of the appli-

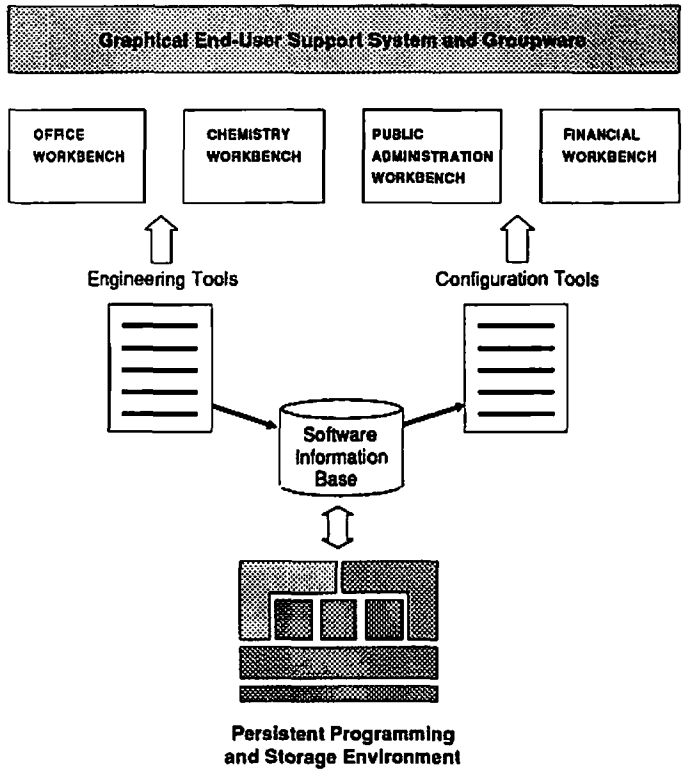

Figure 1 The ITHACA Approach to Application Engineering

cation areas we aim to serve, all four applications have a high generic nature and are expected to be relevant to the market in the near future.

3. The Persistent Object Environment HooDS

HooDS supports object-oriented programming in general and persistent objects in particular. As a programming environment, HooDS provides:

- a multi-programming language approach,
- the basic programming tools for programming in the small and
- an optional desktop environment for convenient access. The desktop can be used by applications as an application interface, as well as by programmers as a system interface to the system.

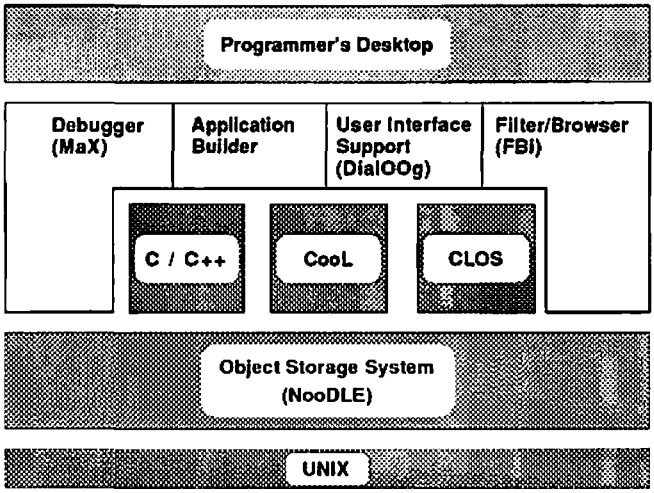

Figure 2 The General System Architecture of HooDS

As an environment which supports persistent objects, HooDS provides the functionality of a fully-fledged object-oriented database system. Thus, HooDS supports persistent objects, multi-user access to shared objects, concurrency control, (closed) nested transactions, object retrieval by both navigation and association and, last but not least, recovery in the case of software failures. Note that in the context of ITHACA, persistence of objects is considered as a feature for those applications which need database support and not as an end in itself in order to store development data - although this would be equally possible and is in fact planned for future versions.

Figure 2 shows the component structure of HooDS. On the highest level of abstraction, the kernel consists of a set of object-oriented programming languages. In order to realise persistent objects, these languages are connected to an object storage subsystem, called NooDLE.

To support the practical needs of programmers, a runtime monitoring and debugging facility, called MaX is supported [Brodde '89], [Brodde '90]. In a world where the reuse of software components written by other programmers is the standard, a facility such as this constitutes more than just one of life's luxuries.

A specialised application builder is under development [Krickstadt '90] in order to ease the process of building a specific application system by connecting class declarations, interconnect this application to the appropriate user interface objects and finally generate the necessary meta-information for the management of persistent objects in NooDLE (i.e. the schema). Due to the internal complexity of object-oriented application systems (e.g. the resolution of inheritance) the

standard UNIX®[1] *make* facility is only of limited use in this context. To nevertheless ensure conformance with the standard, the application builder is built on top of the *make* facility and thus generates standard *make* files.

The design and implementation of appropriate graphical user interfaces is an expensive task. Based on meta-information kept internally, the dialog management system DialOOg is able to generate simple form-oriented, graphical user interfaces on the basis of OSF/Motif for a majority of standard representations.

Finally, in order to retrieve objects interactively the kernel provides a filter/browser system called FBi which acts on both the storage subsystem to retrieve the data of an object, as well as on the languages able to execute the methods of an object.

Unlike current approaches to object-oriented database system development, HooDS focuses on achieving an open, extensible and configurable architecture.

A number of research projects are currently under way which focus on the development of prototypes for extensible database systems [Batory et al. '86], [Carey et al. '86], [Thompson et al. '89]. The general idea of extensible database systems is to modularise the system's architecture by factoring out functionalities into modules. This idea is quite simple, but nevertheless contradicts the common approach to database system design which centres on monolithic systems with a layered architecture.

We may intuitively draw some clear advantages of extensible architectures. First, an extensible architecture allows systems to be configured at installation time in accordance with the needs of different application domains - if a specific functionality is not needed then it is not configured at all. Thus, HooDS permits any number of configurations to be installed in a system related to the different needs of application domains. Nevertheless, HooDS guarantees that applications based on different configurations may also freely co-operate and exchange information.

Second, extensible architectures are better suited for staying abreast of the progress of technology and for absorbing improvements more frequently, faster and mostly without the need to rewrite major portions of the system.

Last but not least, extensible architectures provide a better basis for the construction of open systems. The vital case for openness with respect to object-oriented database systems is the ease with which additional programming languages can be bound into them. Current object-oriented database systems are implemented as huge, monolithic systems, which closely combine a specific programming language with a storage management system for handling persistent objects. Due to the fact that (object-oriented) languages do not support any notion of persistence, the favoured language is usually extended by some model of persistence. The implications of this approach are that the overall conceptual model of persistence is both language-dependent and model-specific. In practice, this approach thus implies that it is neither possible to integrate any additional language into the system nor to introduce any different model of persistence. In fact, current implementations are noticeably closed solutions and thus resemble database system developments of the late 1960s.

1. UNIX is a registered trademark of AT&T.

In order to compensate these drawbacks of current implementations, HooDS supports a formal data model and a related algebra called NO2. This model serves as the explicit binding interface between languages and the storage subsystem NooDLE. The NO2 data model is comparatively general and allows the developer of a specific language binding to decide more or less freely how to extend his favoured (object-oriented or imperative) language by concepts of persistence. It should be noted that the use of a formal data model not only fomalises language binding, but also ensures that different languages (and language extensions) may interoperate within the HooDS system.

In order to facilitate implementation of different language bindings and to increase the flexibility of binding, NooDLE supports different levels of interfaces. At the highest interface level, NooDLE supports an object-oriented extension of SQL which allows embedded oSQL approaches to be built as used in the conventional relational approach. On a medium level, NooDLE provides a converter interface which permits language binding via a call interface. At the lowest level, NooDLE maintains a heap interface which corresponds to an abstract low-level heap as used in most programming languages. Unlike other approaches, this heap is also used internally by NooDLE as a database buffer, hence constituting some sort of stable heap [Kolodner et al. '89]. This interface minimises any overhead and is thus extremely useful when language binders have direct access to the internals of the language's compiler.

NO2 Data Model

The NO2 data model [Elsholtz '89] [Dittrich et al. '90] [Geppert et al. '90] is based conceptually on the O2 data model [Lecluse and Richard '89], but extends this model by ideas from the ENF2 model [Pistor and Anderson '86]. Thus, the NO2 model is equally well suited to serve both object-oriented and imperative programming languages as a storage medium.

NO2 clearly distinguishes between objects and values. Objects have an identity and exist by themselves independent of their values, while values exist only by belonging to an object. An object is thus a pair comprising a surrogate and a value. Objects are always instances of an object type. Complex objects are constructed by orthogonally including objects in values of objects. As with objects, values are instances of value sets. Constructors for tuples, arrays, lists and sets are offered in order to construct structured value sets. To increase the flexibility of data modelling, NO2 also supports the composition of objects, in addition to aggregation of objects. Composition models the IS-PART-OF relationship between objects. Subobjects of a composite object may be owned exclusively on the existence of their parent object. Composition is first of all a value of its own for application domains which have to handle hierarchies of objects as a whole. Moreover, composition provides a basis for the design of specialised (and more efficient) storage and retrieval strategies.

Language Bindings

The initial version of HooDS supports C++, CLOS and CooL as programming languages. Two successive approaches are used for binding C++. In a first attempt, a persistent class library is provided which allows application programmers to achieve persistence for their actual class by subclassing from persistent classes. In a second step, which is currently under design, an extension to C++ in the direction of O++ [Agrawal and Gehani '89] is planned. These extensions,

which are few in number, are processed by a preprocessor and translated to calls to the converter interface. For CLOS (the de-facto standard extension of LISP by object-oriented concepts [Attardi et al. '89]), persistence is transparent to the programmer and is achieved through direct binding on the level of the heap interface.

CooL is a newly designed object-oriented programming language which closely combines the object-oriented approach to programming with current database technology.

CooL

CooL [CooL '90] is especially designed with the objective of giving programmers an adequate language for building large and complex application systems. Large-scale system building does not usually take place in "splendid isolation", but has to consider existing software. Thus, CooL is fully compatible with C and allows any existing C software to be integrated without additional effort. Despite this compatibility, CooL does not inherit the insecurity of C, but rather offers a high level of reliablity by incorporating strong typing, data encapsulation, controlled inheritance and exception handling. Of course, CooL is an object-oriented language which supports the usual features of object-orientation. Nevertheless, the design is not religious and thus extends the language for the benefit of professional programmers by a number of concepts which are not necessarily a 'must' for object-oriented language systems. An example of such an extension is that CooL supports a rich set of type constructors, including sets with a computational complete set algebra.

Support for persistent objects is an integral concept of CooL. For programmers, persistence is transparent and uniform. This implies that they may use any construct together with any object, regardless of whether its property is persistent or volatile. CooL provides transactions, read/write locks and elaborated iterators in order to manage persistent objects. A runtime overhead for persistence only has to be paid if persistent objects are actually used in an application. Otherwise CooL is at least as efficient as C++.

Under the present-day practical conditions encountered in industry, object-oriented database systems only constitute small islands in the world of relational database systems. To succeed commercially integration to SQL systems is thus essential. Hence CooL supports a type constructor relation which is mapped directly onto a SQL system and which allows persistent (complex) objects with tuples from a SQL system to be merged within a CooL program.

NooDLE

The object storage system NooDLE constitutes the core of HooDS and provides a so-called structurally object-oriented storage system. NooDLE supports the NO2 data model and implements the IS-A hierachy based on multiple inheritance, (closed) nested transactions, concurrency control based on read/write locking, query processing (also on type hierarchies), recovery based on *before* images and a n-client/1-server architecture with dedicated cache management. Of course, NooDLE is in itself again extensible and hence configurable.

Figure 31 illustrates the architecture.

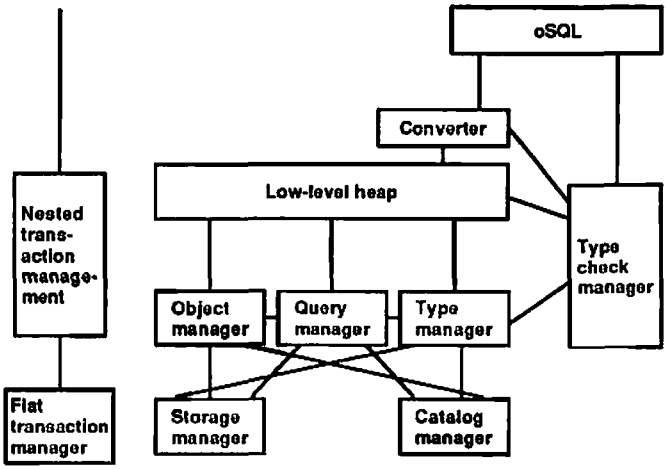

Figure 3 General Architecture of the Storage Manager

In the current implementation, the flat transaction manager and the catalog manager are based on a relational database system. However, dedicated severs are under evaluation.

MaX

MaX provides comfortable monitoring and debugging facilities to the programmer of HooDS. In this paper, we restrict ourselves to emphasising the debugging facility of MaX. The MaX debugger provides multi-language support for language implementations based on the COFF format. MaX debugs at source code level, providing all code printouts in the original programming language. As a debugger suited for object-oriented languages, it also solves inheritance by listing the inherited code when it is executed. Similarly, MaX is able to cope with exception handling and displays raised exceptions and the name of the exception to the programmer. Besides common debugging facilities, the MaX runtime environment supports conditional and unconditional breakpoints and trace points, full variable access (persistent and volatile) and all data types. On the basis of standard UNIX® facilities, MaX is able to debug programs written in different languages, thus providing CooL source code whenever CooL is used, for example, and switching to C source code when C code is called.

Filter/Browser

The filter/browser system constitutes an end-user tool which allows the objects of an application to be retrieved interactively. The retrieval process is carried out in two steps. The filter provides a query-by-example interface [Zloof '82] to the user which allows objects to be queried in a natural way by describing patterns for the results of the query. The evaluation of the query yields a

so-called result set. The user can navigate through this result set by means of the browser. It should be noted that the filter/browser is a standard tool for which the information is generated internally by the system. It need not be implemented by the programmer for a specific application.

DialOOg

This tool is currently undergoing design and will provide a dialog manager which generates user interfaces based on meta-information kept internally within the system. These interfaces will be form-oriented and will incorporate as a default the entire dialog behaviour to permit object creation, modification, display and method activation. References to objects will be marked and may be dereferenced by navigation. This provides an implicit browsing mechanism within the object base conforming to a hyper-object approach. DialOOg is based on OSF/Motif and can be tailored at several levels for supporting specific interface layouts.

4. Application Development Tools

The goal of the ITHACA Application Development Environment is to reduce the long-term costs of application development and maintenance for standard applications in selected application domains. By "standard" applications we mean classes of similar applications that share concepts, domain knowledge, functionality and software classes.

The key requirement is that applications developed using the environment be flexible and open-ended: it should be possible not only to develop applications quickly and flexibly, but it should be relatively easy to reconfigure applications to adapt to evolving requirements.

The key assumption of the approach is that one must be able to adequately characterise the selected application domains in order that individual application can be constructed largely from standard object-oriented software components. Achieving reusability of not just software but of previous *experience* is therefore an essential activity within the approach. This, in turn, implies the need for a different kind of software life-cycle in which the long-term development and evolution of reusable software proceeds in tandem with the short-term development of specific applications.

The ITHACA approach requires the development of "application workbenches" consisting of reusable application domain specific software components and software information. This information is stored in a *Software Information Base* (SIB) and is accessed by an application developer either interactively by a Selection Tool, or indirectly through the use of the other development tools. We distinguish between *application development*, which refers to the development of specific applications by means of the application workbenches and the ITHACA development tools, and *application engineering*, which refers to the activity of preparing the contents of the SIB, that is, developing the application workbenches themselves.

Application engineering is expected to be an incremental, long-term activity, since reusable software can only be developed on the basis of experience gained from the development of specific applications (including existing "precursor" applications). The benefits are to be realised during application development, since one can reuse existing requirements models, existing (ge-

neric) designs and existing software. Most of the activity in application development should ideally take place in the task of matching user requirements to generic designs, and in configuring running applications from available components. Since mostly tried and tested software will be used, less time should be spent on detailed debugging, and since construction of applications from existing parts should rapidly lead to evolutionary prototypes, more time can be spent ensuring that the client's needs are properly met. Except for unusual applications, little effort should be spent capturing exceptional requirements, re-engineering existing designs or programming new custom software. Clearly this scenario depends on the extent to which one can capture application domains, but even non-standard applications can benefit from the approach if the special requirements can be localised to particular subcomponents or to incremental modifications of existing software objects.

The raw material with which the application developer works is the software information contained in the SIB. This information is organised into *Generic Application Frames* (GAFs), which describe how specific applications can be constructed from the available components. In order to ensure that it will be possible to map the application requirements to a GAF, it is essential that requirements collection and specification *start* by using the SIB as a basis for specifying the application. Matching requirements to existing application domain knowledge starts *immediately* as requirements are collected. The *Requirements Collection And Specification Tool* (RECAST) effectively provides a "guided tour" of the SIB, attempting to construct a specific application frame (SAF) on the basis of generic information and user requirements. Software components selected and identified at this stage are then tailored and composed to construct the running application by a combination of programming and *scripting*. The *Visual Scripting Tool* (VISTA) supports the interactive construction of applications by graphically editing and connecting visual representations of application and user interface objects. The tools are tightly integrated to permit, for example, simultaneously development of parts of the application during requirements collection, re-examination and refinement of requirements specifications during development, and access to programming tools during scripting. We shall briefly present the functionality and current status of each of these tools. Figure 4 provides a simplified view of the tools interaction.

The *SIB* provides the underlying mechanisms for storing and representing *descriptions*. Descriptions encapsulate properties of software components and knowledge concerning application domains for use by the other tools. Since descriptions may refer to one another, the contents of the SIB can be seen as a semantic network, sharing properties of both object-oriented (design) databases and of hypertext systems. A prototype of the SIB has been built using the Telos knowledge representation language [Koubarikis et al. '89]. Descriptions are thus represented internally as Telos propositions, but externally may appear as, for example, software templates, requirements collection forms, or application designs. The uniform internal representation permits advanced queries to be posed to the SIB and evaluated by the Telos inferencing mechanisms.

The *Selection Tool* provides the means to retrieve software descriptions from the SIB and to navigate through GAFs. The Selection Tool may be used either interactively or indirectly via the other tools. The two modes supported are (1) querying, in which a thesaurus and a set of filters are used to reduce the focus of interest, and (2) browsing, in which the client can navigate through the software information network (Figure 5). A prototype has been built using the *Lab-*

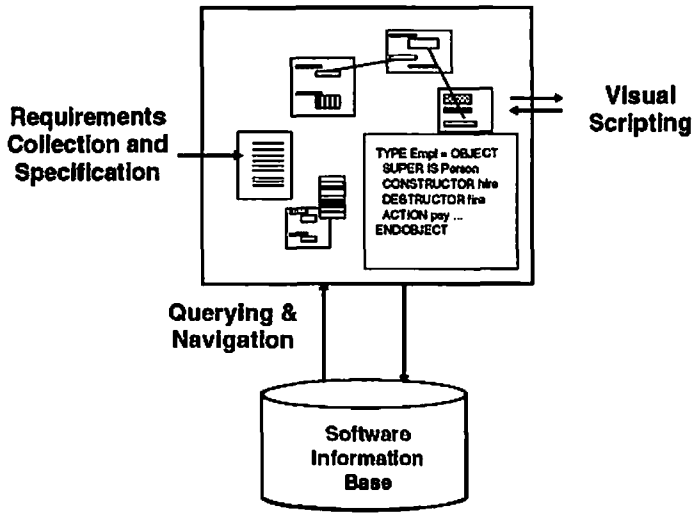

Figure 4 Simplified View of User/tools Interaction

yrinth system (Laby), a constraint-oriented graphics engine which runs on top of X. Laby is used to display various hierarchical views of the SIB during browsing. *RECAST* directs the application developer in the task of collecting requirements and matching them to existing designs and specifications by providing a "guided tour" of the SIB. RECAST is based on the *Objects with Roles Model* (ORM), an object-oriented specification model which considers that objects undergo a life-cycle in which they may play a variety of *roles*, each of which may associate different properties, states, methods, usage rules and constraints with objects [Pernici '90]. A first prototype of RECAST guides the application developer along a series of choices associated with different parameters of a generic application. Current work on RECAST is concerned with refining this specification technique and with the flexible interplay between RECAST and the other tools.

VISTA is an interactive tool for *scripting* running applications from previously programmed components [Nierstrasz et al. '90]. The term "scripting" refers to the idea that components should completely encapsulate some well-defined behaviour, like character actors in a play, whereas the interaction between components and the precise roles they should play should be specified separately, that is, within a script. A script, then, may bind parameters of generic classes, specify which instances of object classes are needed in various parts of an application, and introduce objects to one another. Both user interface components and internal application components are visually presented to the user, and graphical editing facilities are used to link together the components. Scripts encapsulate the bindings between objects, and so may themselves be reused as components within other scripts. Furthermore, a script, with the help of the other tools, can viewed as a generic design: once an application has been built as a script, it is a simple matter

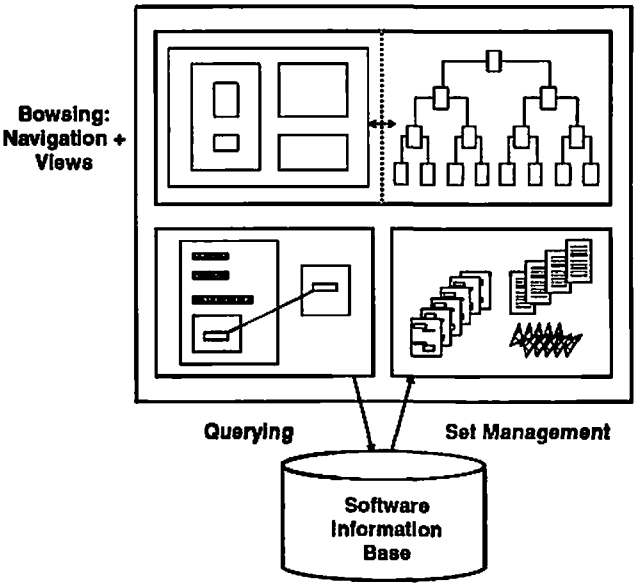

Figure 5 SIB Management

to unpack it and use it as a prototype for a new application, obtaining suggestions for design alternatives via RECAST and the Selection Tool.

Scripting depends heavily on the definition of standard interfaces between components and on the existence of a rich library of components that conform to these interfaces. Object-oriented techniques are essential here as they provide the mechanisms by which subclass hierarchies can share a common abstract interface. A "scripting model" is a description of these standard interfaces and the allowable interconnections supported by components for a given application domain. For example, a scripting model for a user interface kit might specify which methods are to be used to propagate events between application and user interface objects, and which events are valid to pass to which classes of objects.

A proof-of-concept prototype based on a UNIX® scripting model has been implemented [Stadelmann et al. '90]. An object-oriented visual scripting tool (VISTA) based on Laby and the MOTIF toolkit is now under development, and an initial version is expected to be released within the project by the end of 1990. Work on scripting models for various application domains is ongoing.

5. The Applications

Demonstrator applications which reflect real-world situations and common problem areas facing industrial software production are used to test and ensure the suitability of the environment developed in ITHACA. In addition to specifying their requirements, these 'workbenches' are also developing new domain architectures and innovative techniques oriented to their respective, non-IT-specific spheres of activity.

Office applications constitute a real challenge for testing the suitability of ITHACA technology in the commercial/industrial scenario. Offices in general form an application scenario which requires a wide range of different tools and facilities in order to meet the requirements of the different "styles" of office encountered (size, type of work etc.). The general nature of these tools allows them to be used in a variety of other applications to a greater or lesser extent.

Due to the fact that the ITHACA Office Workbench provides a general concept in terms of a reference office model and several basic technologies, it is described in some detail here. The components it includes can be applied to the other workbenches as necessary.

5.1 The Office Workbench

Any office application must (1) deal with a complex environment (distribution, client/server architecture, large objects), (2) integrate existing pieces of software (editors, spreadsheet, printing and filing services, electronic mail services), and (3) provide a very simple interface to non-expert end-users. The major application target selected is computer-supported cooperative work in all its possible variations: from rigid to flexible procedures, from a few instances to hundreds of thousands of instances, from a few steps to one hundred steps, with a total duration ranging from one day to several months, and involving two to dozens of actors.

In order to meet this goal, several interrelated components are provided that can be used together or separately. A CooL Office Model provides an object representation of organisation, information and facilities. A set of CooL Office Operators enable activation of classical office operations (such as editing, filing, mailing, printing) on these objects and the definition of more elaborate actions by assembling them in scripts using the scripting tool. A coordination procedure application provides the means to express procedures in the form of a network of activities which encapsulate actions. A desktop application provides the user with a workstation interface to the overall application. In addition, a Budget Management System provides a decision-making tool using constraint-based artificial intelligence techniques for analysing and optimising budgets in large organisations. This tool will be able to access information provided by commercial information services with access to an information services programmatic interface.

The *Office Model* provides a set of CooL objects representing office concepts in the fields of organisation, facilities and information [Ang '90]. The organisation model includes three main object types: grouping, actor and role. It constitutes the basis for authorisation and access control. The facilities model represents objects to organise the user space in the form of folders, repositories, trash-can, in-tray, out-tray, printer and file servers. The information model represents objects, such as forms, documents, messages, drawings, spreadsheets etc. The various parts of the model are interrelated. A folder can contain a letter whose "to" and "copy" instance

variables can have sets of actors or roles as their value. Together, objects of the office model represent the application domain on which office operators can be applied and combined into scripts to form actions linked together into procedures by using the COP tool.

Office Operators represent an abstraction of most commonly used actions in an office environment, such as editing, mailing, filing and printing. They comprise a combination of objects and methods which encapsulate existing pieces of software and which take into account the objects of the office model. For example, a print operation will involve the information object to be printed, a printer server object representing the available printer server, a trading object able to negotiate the applicability of the service to print the information object, and a conversion object which, if required, is able to convert the format of the content portion of the object to a format acceptable to the printer.

COP (COordination Procedure) is a tool for supporting the application engineer and the end-user in modelling and executing cooperative task processing [Tueni et al. '90]. It is designed to support a large scope of applications, from highly repetitive, highly structured administrative procedures (workflow processing) to highly unstructured and flexible cooperative sessions between office actors (cooperative computer-supported work). COP relies on a language providing constructs to express dynamic behaviour of the coordination procedure, an engine which interprets structures generated by the compiler for supporting the distributed execution of the procedure, and support tools for graphical programming, scripting actions, debugging and for using the COP applications. The principal COP elements are activity, input and output, pre- and post-condition, procedure and object. The language supports the expression of sequences, alternatives, parallelism, loop and rendezvous, composition and refinement, aggregation and specialisation, and an unlimited level of abstractions. The COP engine offers resume, suspension, correction, cancellation, jumping, exception, help, history, event triggering and explanation.

The general *Desktop* is configured to present the user with a metaphor of her/his current working environment that must be easy to use and as flexible as possible to incorporate all personal applications available to her/him [Brady '90]. In order to facilitate speedy access, the Desktop presents all documents which have just been created by the user, or which the user wants to edit immediately and which are therefore available locally on the workstation. The Desktop is used to start all local applications, such as editors for texts, graphics and images. In addition, the Desktop application allows the user to access all office services, such as electronic mail, central filer, printer service etc. Documents no longer required can be stored in a trash-can which is also a component of the desktop. The Desktop is to be implemented in C and CooL, with OSF/Motif for the user interface.

The *Budget Management System* supports the functions of a company top management budgeting process, including budget preparation and budget evaluation using financial analysis [Bicard-Mandel et al. '89]. It incorporates a data model and appropriate knowledge-based tools. The data model provides the means for viewing the budget elements at several levels of abstraction and/or factors of analysis (e.g. company departments, chart of accounts, products, time). The knowledge-based tools provide constraint formalisms tailored to the budgeting domain. These formalisms are used for expressing consistency goals for the data model. Tools are provided for building the semantic network representing a company's budget, where nodes are bud-

get entities (accounts, products, departments, production factors) and where links have budget-oriented semantics with associated methods for regulating information flow through the system.

Access to information services provides a CooL programmatic interface for implementing access to information in a client/server mode. This programmatic interface acts as a client for the service and as a server for the client application or user interface. The services integrated in this way could act as sources of up-to-date information, which could then be transferred to and exploited in local databases, spreadsheets and word processors. A specific access can be integrated as an action part in an activity network controlled by the COP engine.

5.2 Financial Workbench

The activities of the Financial Workbench are organised around the development of both a specific and a generic application for the sale of financial products, in particular insurance [Bruni et al. '89]. Key marketing issues are the development of new insurance services and products, as well as the extension and enhancement of the distribution network. In this context, a sales support tool is being developed which collects and uses all relevant information related to consultancy on the subject of insurance products geared specifically to customer needs.

Activities centered on developing the specific insurance application represent a key step for acquiring the necessary knowledge and expertise concerning the domain to be abstracted in the application frame.

5.3 Public Administration Workbench

The Public Administration Workbench demonstrates the applicability of the ITHACA environment in relation to a real setting within a public administration setup [Garcia et al. '89]. The application is concerned with workflow automation, where a citizen's request must be processed within the administrative setup by different services according to precise rules and at different locations.

5.4 Chemistry Workbench

The Chemistry Workbench belongs to the CIM domain and involves the development of a graphical control desk for controlling plant and machinery in the chemicals production process. This control system is linked to a planning system for solving the logistic problems specific to this particular sector.

6. An Industrial Approach

From the very outset, the ITHACA project chose an industrial approach by selecting standards to be applied for development and by defining a common hardware platform to ensure full integration [Konstantas '90]. The goals of the project already correspond to the established main line of activity of the development units of the main industrial partners involved.

The standards and *de facto* standards which have been adopted within the project include the following:

- UNIX®, conforming to the X/Open portability guide,

- C compiler and GNU C++ compiler,
- X/Windows and OSF/Motif toolkits for user interfaces,
- TCP/IP and NFS,
- SQL for database interface,
- ODA, ODIFF, DFR, X.400 and X.500 for office automation.

Besides ensuring compatibility between the various organisations involved in the project, the consistent use of these standards will also ensure conformance with the market requirements when finished products are ready to be launched.

In addition, both Siemens Nixdorf Informationssysteme AG and Bull S.A. participate actively in the same worldwide organisations for the main elements of the architecture: X/Open for UNIX® portability, the Open Software Foundation for UNIX® extensions, and the Object Management Group for future object-oriented standards.

The ITHACA common hardware platform is an Intel 80386-based workstation delivered either by Bull S.A. or Siemens Nixdorf Informationssysteme AG. The workstations run the UNIX® operating system and conform to the 386 application binary interface. The workstation provides full support of a selection of components, including C, GNU C++, X/Windows System, OSF/Motif, TCP/IP, NFS, INET and ORACLE. All software components produced in ITHACA are qualified on this platform and have to run there before they are classified as accepted and able to be officially delivered to each partner.

7. Conclusions

In order to gain as much exposure to the market as possible, the ITHACA environment will be provided to external companies and universities for evaluation purposes on the basis of evaluation licence contracts as soon as individual components are available. For example, the first version of the CooL compiler has been available since June 1990 and the first version of the Nood-DLE/CooL integrated environment will be available in October 1990. An office application prototype written in CooL will be on show at the exhibition accompanying the ESPRIT Conference 1990.

Prototypes of each of the application development tools were produced during the first year of the project. Redesign and reimplementation of the tools is now under way. A C++/X version of Labyrinth has been made available to partners and is now being used as a graphical front-end to VISTA. A first version of VISTA is being made available to the partners as of October 1990.

References

[1] R. Agrawal and N.H. Gehani, "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++", Proceedings of the Second International Workshop on Database Programming Languages, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1989.

[2] J. Ang, "A Comprehensive Office Modeling Framework for Ithaca", Technical Report, Bull S.A., Paris, 1989, ITHACA.BULL.89.D8.1a.

[3] G. Attardi, C. Bonini, M.R. Boscotrecase, T. Flagella and M. Gaspari, "Metalevel Programming in CLOS", in \IP Cook, British Computer Society Workshop Series, 1989.

[4] D.S. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell and T. Wise, "Genesis: A configurable database management system", Technical Report TR-86-07, Department of Computer Science, University of Texas at Austin, 1986.

[5] J. Bicard-Mandel (Ed.), C. Chen, G. Vlodakis and M. Androulakis, "User Requirements and Architecture Specifications for the Budget Management System", Technical Report, Bull & NTUA-CMSU, 1989, ITHACA.BULL.89.D10.1.

[6] M. Brady, S. Kienapfel and G. Zschoche, "An Office Desktop Approach", Technical Report, Nixdorf Microprocessor Engineering GmbH, Berlin, 1989, ITHACA.NIXDORF.89.D3#1.

[7] B.Brodde, "MaX - Monitoring and X-Ray Tool for ITHACA", Technical Report, Nixdorf Microprocessor Engineering GmbH, Berlin, July 1989, ITHACA.NIXDORF.89.B4.#1.

[8] B.Brodde, "MaX User Handbook", User Handbook, Nixdorf Microprocessor Engineering GmbH, Berlin, August 1990, ITHACA.NIXDORF.90.B4.4.1#1.

[9] G. Bruni, C. Cardigno, M. Damiani and G. Seminati, "Final Report on Insurance Domain Requirement Analysis", Technical Report, DATAMONT R&D, Milano, 1989, ITHACA.DATAMONT.89.D.7.#3.

[10] M. Carey, D. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. Richardson and E. Shekita, "The Architecture of the EXODUS extensible DBMS", Proceedings of Object-Oriented Database Workshop, 1986.

[11] Cool. Development Team, "CooL/0 Language Description", Reference Manual, Nixdorf Microprocessor Engineering GmbH, Berlin, May 1990, ITHACA.NIXDORF.90.L2.#2.

[12] K.R. Dittrich, A. Geppert and V. Goebel, "The Data Definition Language of NO2", Technical Report, University of Zürich, March 1990, ITHACA.ZUERICH.90.X.4#2.

[13] A.Elsholtz, "NooDLE, a New Object-Oriented Database System for Advanced Programming Language Environments", Technical Report, Nixdorf Microprocessor Engineering GmbH, Berlin, November 1989, ITHACA.NIXDORF.89.X.4#1.

[14] J. Garcia, J. Lopez, J. Mongiou and R. Sole, "Design Description of the Administration Workbench. First Draft.", Technical Report, TAO, Barcelona, 1989, ITHACA.TAO.89.D.6.#8.

[15] A. Geppert, K.R. Dittrich and V. Goebel, "An Algebra for the NO2 Data Model", Technical Report, University of Zürich, July 1990, ITHACA.ZUERICH.90.X.4#4.

[16] Brian W. Kernighan and Dennis Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1978.

[17] E. Kolodner, B. Liskov and W. Wheil, "Atomic Garbage Collection: Managing a Stable Heap", in \IP International Conference on the Management of Data, 1989.

[18] D. Konstantas, "The Ithaca UNIX Development Platform", Technical Report, D-Tech, 1990, ITHACA.D-TECH.90.X.#4.

[19] M. Koubarikis, J. Mylopoulos, M. Stanley and A. Borgida, "Telos: Features and Formalization", Technical Report CSI 1989/018, FORTH, Herklion, Crete, 1989.

[20] T. Krickstadt, *CAKE: Towards an Object-Oriented Application Builder*, Nixdorf Microprocessor Engineering GmbH, Berlin, September 1990, ITHACA.NIXDORF.90.E1.#1.

[21] C. Lecluse and P. Richard, "The O2 Data Model", Technical Report, pp. 39-89, Altair, Le Chesnaey Cedex, 1989.

[22] G. Müller and A.-K. Pröfrock, "Four Steps and a Rest in Putting an Object-Oriented Programming Environment to Practical Use", in \IP Workshop Series, 1989.

[23] O.M. Nierstrasz, L. Dami, V. de Mey, M. Stadelmann, D.C. Tsichritzis and J. Vitek, "Visual Scripting – Towards Interactive Construction of Object-Oriented Applications", in *Object Management*, ed. D.C. Tsichritzis, pp. 315-331, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[24] B. Pernici, "Objects with Roles", Proceedings ACM-IEEE Conference of Office Information Systems (COIS), Boston, April 1990.

[25] P. Pistor and F. Andersen, "Designing a Generalized NF2 Model with a SQL-Type Language Interface", Proceedings of the Twelfth International Conference on Very Large Data Bases, IBM Wissenschaftliches Zentrum, Kyoto, August 1986.

[26] A.-K. Pröfrock, M. Ader, G. Müller and D. Tsichritzis, "ITHACA: An Overview", Proceedings of the Spring 1990 EUUG Conference, pp. 99-105, 1990.

[27] M. Stadelmann, G. Kappel and J. Vitek, "VST: A Scripting Tool Based on the UNIX Shell", in *Object Management*, ed. D.C. Tsichritzis, pp. 333-344, Centre Universitaire d'Informatique, University of Geneva, July 1990.

[28] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[29] C. Thompson et al., "Open Architecture for Object-Oriented Database Systems", Technical Report 89-12-01, Texas Instruments, 1986.

[30] D.C. Tsichritzis and O.M. Nierstrasz, "Application Development Using Objects", Proceedings of the First European Conference on Information Technology for Organisational Systems, pp. 15-232, North Holland, Athens, May 1988.

[31] M. Tueni (Ed.), J. Alsina, A. Graffigna, J. Li, G. de Michelis, J. Monnguio and H. Wiegmann, "Towards a Common Activity Coordination System", Technical Report, Bull S.A., Nixdorf Computer AG, TAO, University of Milano, 1989, ITHACA.BULL.89.U2.#1.

[32] M.M. Zloof, "Office-By-Example: a business language that unifies data and word processing and electronic mail", IBM Syst. Journal, vol. 21, no. 3, pp. 272 - 304, 1982.