Diss. ETH No 11592

Integration of Online Documents

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Ralph Olivier Sommerer
Dipl. Informatik-Ing. ETH
born February 6, 1963
citizen of Niederglatt, Zürich

accepted on the recommendation of
Prof. Dr. J. Gutknecht, examiner
Prof. Dr. N. Wirth, co-examiner

1996

Integration of Online Documents

Ralph Sommerer

Copyright (c) 1996 by Ralph Sommerer

Acknowledgements

I want to thank my advisor Prof. J. Gutknecht for his liberal supervision of this project, for his numerous contributions, and also for his patience. He commented thoroughly on earlier drafts of this thesis and provided valuable criticisms that helped to improve the presentation of this material considerably.

I also wish to thank Prof. N. Wirth for accepting the co-examination of this thesis. His way of thinking and problem solving had a significant impact on my work.

My colleagues at the Institute of Computer Systems contributed to a friendly and intellectually most inspiring working atmosphere that is gratefully acknowledged. In particular, I wish to thank (in alphabetical order) Andreas Disteli, Martin Gitsels, Urs Hiestand, Hannes Marais and Karl Rege for the numerous stimulating discussions during coffee breaks and lunch-hour, and for providing me with valuable input and support.

There are many other people who have contributed in one or the other way to the success of this work. The fact that they are not explicitely mentioned here shall in no way degrade their significance.

Last but not least, my deepest gratitude goes to my mother. I cannot imagine having successfully completed my studies and research without her encouragement, motivation and support.

Contents

Abstract/Kurzfassung

Introduction 1

A Universal Document Oriented Interface 5
- Introduction 5
- System Environment 6
- Services and their Representations 11
- Online Documents 14
- Smart Links 15
- A Document Oriented User Interface Model 18
- Example Service Integration with Smart Links 23
- Related Work 25
- Summary 27

Case Study 1: Dynamic Mathbook 29
- Introduction 29
- The Dynamic Mathbook 30
- Integration 32
- Implementation Aspects 35
- Related Work 45
- Summary and Conclusion 46

Case Study 2: Electronic Newspaper 49
- Introduction 49
- Teletext 50
- Teletext Presentation 51
- Implementation Aspects 56
- Further Applications: The Electronic Investment Bulletin 63
- Related Work 64
- Summary and Conclusion 65

Case Study 3: Network Information Browser 69
- Introduction 69
- Smart Web: A World-Wide Web Browser for Oberon 70
- Integration 72

Summary and Conclusion 81
Outlook 83

Conclusions and Outlook 85

Bibliography and References 89

Abstract

The present work describes a univeral programming environment and user interface for the integration of online multimedia services into the Oberon system. This kind of online services has obtained an immense relevance especially since the trend towards global telecommunication has become manifest.

By generalizing the notion of a document, services can be viewed as a specific type of documents that we have called *online documents*. In contrast to documents in the traditional sense, the contents of online documents possibly must be prepared and collected by services before being presented on the display screen.

We will show that documents provide a suitable user model for the interaction with services, especially in connection with so called *Smart Links*. Smart links are a special type of links (i.e. references to other documents) that combine the semantics of Oberon commands with the functionality of service calls. They are internally modelled as objects which can be included in documents. Hence, smart links allow to access services directly from online documents.

By integrating smart links, the document oriented user interface model of Oberon is extended to a universal document oriented programming and user environment for the interaction with services. Specifically, this programming environment allows to integrate new online documents with minimal effort because only the service-specific functionality of the smart links needs to be implemented.

The thesis describes the integration of three different online documents. On the one hand, their integration serves to examine the suitability of the presented concepts based on real examples. On the other hand, the implemented online documents prove to be useful tools.

The *Dynamic Mathbook* integrates an interactive formula editor and a symbolic algebra system. The application allows formulae to be manipulated and evaluated directly within a mathematical document. Hence, the user can interactively experiment with mathematical expressions, or illustrate the shape of mathematical functions by means of graphical plots.

The electronic newspaper *TeleNews* is an innovative mapping of a Teletext data base by means of dynamically generated hypertext. *Teletext* (in Germany called *Videotext*) is a non-interactive digital information service that is broadcast by television stations. TeleNews modernizes and hence improves

both the representation of the Teletext service on the screen, and its user interface.

The World-Wide Web browser for Oberon results from an attempt to integrate an existing Internet information service based on the document oriented programming environment. The result is a completely integrated, run-time extensible Internet browser that supports most of the graphical facilities of the World-Wide Web including in-line images and forms.

Kurzfassung

Die vorliegende Arbeit beschreibt eine universelle Programmier- und Benutzerumgebung zur Integration von lokalen und externen Online Multimedia Services in das Oberon System. Solche Services haben vor allem im Zuge des Trends hin zu globaler Telekommunikation grosse Bedeutung erlangt.

Eine Verallgemeinerung des Begriffs des Dokumentes erlaubt es, Services als spezielle Dokumente, sogenannte *Online-Dokumente*, aufzufassen. Im Gegensatz zu Dokumenten im herkömmlichen Sinn zeichnen sich Online-Dokumente dadurch aus, dass ihr Inhalt vor der Darstellung auf dem Bildschirm möglicherweise erst von einem Service bereitgestellt werden muss.

Es wird gezeigt, dass Dokumente in Verbindung mit sogenannen *Smart Links* ein geeignetes Benutzermodell für die Interaktion mit Services darstellen. Smart Links sind eine spezielle Form von Links (Verweise auf andere Dokumente), welche die Semantik von Oberon Kommandos mit der Funktion eines Service-Aufrufs verbinden. Sie sind intern als Objekte modelliert, die in einem Dokument mitfliessen können. Smart Links erlauben daher, beliebige Services direkt aus Online-Dokumenten heraus anzusprechen.

Die dokument-orientierte Benutzerschnittstelle von Oberon wird durch die Verbindung mit Smart Links zu einer universellen dokument-orientierten Programmier- und Benutzerumgebung für die Interaktion mit Services. Diese erlaubt im Speziellen, neue Online-Dokumente mit minimalem Aufwand zu realisieren, da jeweils lediglich die service-spezifische Funktionalität der Smart Links implementiert werden muss.

Die Dissertation beschreibt die Integration dreier verschiedener solcher Online-Dokumente. Deren Integration dient einerseits dazu, die vorgestellten Konzepte anhand realer Beispiele zu erproben. Anderseits stellen die Online-Dokumente selbst aber auch praktische Werkzeuge dar.

Das *Dynamische Mathematikbuch* integriert einen interaktiven Formeleditor und ein Symbolisches Algebra System. Im Dynamischen Mathematikbuch können Formeln direkt in einem mathematischen Text editiert und evaluiert werden, z.B. um mit mathematischen Ausdrücken im Text zu experimentieren oder um den Verlauf von Funktionen mittels graphischer Plots zu illustrieren.

Die elektronische Zeitung *TeleNews* ist eine innovative Darstellung einer Teletext Datenbank mittels dynamisch generierten Hypertexts. *Teletext* (in Deutschland *Videotext* genannt) ist ein nicht-interaktiver digitaler Informationsdienst, der von Fernsehanstalten ausgestrahlt wird. TeleNews modernisiert und verbessert damit sowohl die Darstellung als auch die Benutzerschnittstelle von Teletext.

Der World-Wide Web browser für Oberon ist das Resultat eines Versuchs, auf der Basis der dokument-orientierten Programmierumgebung, einen existierenden Internet-Informationsdienst zu implementieren. Das Resultat ist ein vollständig integrierter, zur Laufzeit erweiterbarer Internet browser, der die meisten graphischen Möglichkeiten des World-Wide Web (z.B. Bilder und Formulare) unterstützt.

Introduction

In the field of interactive computing, two of the most important subjects of current research activities deal with multimedia applications and online services. The term *multimedia*, on the one hand, characterizes applications which integrate and process different media such as images, animation sequences and sound. Such applications have become very popular, especially in education. Inspired by the availability of high performance computing resources and high resolution color monitors, these applications introduced a new level of interaction which reaches beyond the mere assistance in searching and browsing through large data bases. This new level of interaction consists of the *activation of the data* itself.

On the other hand, we can observe an increasing amount of local and remote *services* operating in distributed client/server environments. By services we mean entities that provide upon request either information (information service) or the ability to consume and process arbitrary data items (computing service). These services are characterized by an open interface or a well defined access protocol by means of which other applications (called *clients*) can interact with them.

Both of the above research topics have obtained an immense importance and relevance especially since they tend to *converge*. World–wide, a considerable effort is raised in designing and implementing such online multimedia applications. Definitely, the ample potential of the integration of multimedia and online services is the driving force behind activities that are paraphrased by slogans such as *information highway*, *video on demand* etc.

As the influence of these new media on future personal computing cannot be neglected, prospective research has to explore how the inherent potential of such applications can be exploited within a given user environment. For that purpose, a project has been launched that aims at the integration of access to such services as a prerequisite for all further explorations. The scientific goal of the project is the modelling of a programming environment and user interface for a uniform and consistent interaction with services.

One essential requirement of such a programming environment must be *integration*. Services that exhibit a similar behaviour should be presented uniformly and accessed in a similar and consistent way. Another important

issue is extensibility. In order not to restrict the range of services to be integrated and accessed, the design should allow additions of functionality to the system without invalidating existing parts. Last but not least, simplicity is an important design principle. Instead of inflating the system with concepts, the basis should be kept small and simple.

This thesis describes the conceptual and technical aspects of the project and its functional implications. The notion of *online documents* will serve as a common basis for the integration of services. The term denotes documents (i.e. collections of data items) with two specific properties. First, their content and structure does not necessarily have a permanent static global state. Second, their contents may be distributed over several physical locations and therefore must be collected before their presentation on the screen. Online documents thus provide an appropriate model for information that is collected and presented by services.

The result of our explorations is a small document oriented framework that decouples services' internal aspects from aspects of their user interface, and from aspects of the information's presentation on the screen. An essential part of the framework is an internal representation for services whose abstract interface allows to hide service-specific details of their processing from clients.

The presented concepts have been implemented in order to prove their suitability for the integration of online documents. In a case study of three example applications the integration process is studied. Each application emphasizes a different type, aspect and implication of online documents. The actual online documents presented in the corresponding chapters include a *dynamic mathbook*, an *electronic newspaper*, and a *network information browser* as an example of a "world–wide web" of distributed documents. Due to the document oriented programming framework, the integration of these online documents essentially confined to merely internal aspects of the actual services which handle the corresponding online documents. All other aspects related to the user interface and the presentation on the screen are handled by the framework.

The user environment to build upon is the *Oberon* system. Oberon is a fully-fledged single-user operating environment that contains all resources for practically autonomous work. One of its major characteristics is that the separation of system and user programs has disappeared completely. Instead, applications are merely extensions of the system's functional basis, dynamically loaded on demand while the system is operational. Due to Oberon's object-oriented kernel almost all user activities can be applied to almost all objects that are available in the system and visible on the screen. Hence, integration of applications within Oberon is not an option. It is a basic concept.

Outline of the Thesis

The next, second chapter motivates and introduces the basic concepts of this thesis. It introduces the notion of online documents and relates it to that of traditional documents. A simple programming framework for the integration of online documents is presented. A document oriented user interface model for the interaction with services is based on the service call as the unit of interaction. The resulting *Smart Documents* integrate the information that is provided by services and the services' user interface.

The subsequent chapters are dedicated to a case study of the three different online document types mentioned above. They serve to explore the feasibility of the concepts introduced in the second chapter and to investigate the potential and the implications lying within these concrete applications. In order to keep their description concise, only their most important aspects are described.

The description of the case studies follows more or less the same order. Each application is motivated first. After that, the corresponding online document is presented. Then, implementation aspects are discussed. They deal with service-specific protocols and the representation of the services' data items within Oberon. The discussion closes with a note on differences to similar other existing systems, and a short summary of other's work related to the subject of the corresponding chapter.

Although the implemented applications prove to be usable and powerful tools for the mediation of information, they have not been geared towards outstanding sophistication. We have confined both their implementation and their discussion to the general ideas and implications of the corresponding online documents. They basically serve to reason about the integration of online documents in general, and to outline their abilities and inherent potential.

A Universal Document Oriented Interface

Introduction

In this chapter, we discuss general aspects of an integrated access to local and remote services in the Oberon system. Experience with the integration of services reveals that these aspects can be roughly separated into three groups: (1) Communication protocols between server and clients, (2) the design of a stub as a representation of the service on the client's side, and (3) the user interface model on the client's side. In the following we focus on clients and shall formulate requirements and introduce suitable abstractions for a seamless integration of access to services. The objective of such an integration is to allow uniform and consistent access to various different types of services.

The Oberon system provides a document oriented user interface model based on customizable documents serving as menus. Typically, these are ordinary texts that contain prepared collections of commands to be executed directly from the document. We shall present a generalization and extension of this interface model that leads to a universal document oriented user interface for the use of arbitrary (local and remote) services, and thus to an document interface combining contents and user interface. We shall base this integration on a data type called *smart link* that unifies the notions of a (hypertext) reference link and an abstract service. The latter can be viewed as a client's stub of the real service.

We will illustrate the general concepts presented in this chapter with three examples: A *Dynamic Mathbook*, an electronic newspaper called *TeleNews*, and *Smart Web*, a *World-Wide Web* browser.

The *Dynamic Mathbook* is an electronic document that provides access to a symbolic algebra system. Thereby, mathematical expressions are evaluated directly within the document.

TeleNews is a presentation of the set of Teletext pages in the form of an electronic newspaper. *Teletext* (in Germany called *Videotext*) is a textual information service that is broadcast by European TV stations together with the video signal. Teletext information comprises various topics such as political news, sports news, television schedules, advertisement and so on.

Third, *Smart Web* integrates access to the very attractive *World-Wide Web* information system. This models a distributed multimedia document consisting of hypertext pages, bitmap images etc.

System Environment

Oberon

Oberon is both the name of an operating system and a programming language. The coincidence of names is an indication of the integration of programming language and operating system rather than a symptom for lack of imagination [WiGu92].

In particular, it is an integrated operating system that allows system resources to be accessed from any application by the use of abstract data structures. Oberon differs from other operating systems by the absence of a traditional notion of the program as the unit of both action and algorithm description. Instead, the atomic unit of interaction is the procedure (called *command*) that is invoked directly by the user. A command is therefore an activity that operates on the global state of the system, typically represented by a set of viewers (windows) on the screen. Examples of commands are opening a document, sending a prepared text mail, reading the directory of a diskette and so on. The tasks are carried by a single process and may arbitrarily be sequenced by the user. The Oberon system can therefore be called a *single process multi tasking* system.

Other important properties of the Oberon system are the dynamic loading of program modules, automatic memory management by a garbage collector and a central event dispatcher. All of these properties are both necessary and sufficient in order to achieve unlimited extensibility of the system. "Application packages" such as compilers, editors etc. are merely extensions of functionality of the basic Oberon system. They are are added to the system while it is operational.

The evolution to Oberon System 3

Oberon System 3 basically introduces two new concepts to the standard "classical" Oberon system. These are *Object* and *Library*. The new base type *Object* is rooted in the system basis. It stands for a class of entities that share a common message protocol. Objects generalize and unify concepts that are

already present in the classical Oberon system, such as character patterns, texts and display frames.

Libraries are mutually disjoint collections of objects. An object is said to be *bound* to a library, if the library "contains it". If so, there is a reference number such that the pair (reference number, library) uniquely identifies the object within its library. Objects and Libraries prove to be merely separate aspects of one and the same integrated concept of *persistent objects* [Gu93].

Technically, a library is an indexed collection of objects, and serves three different purposes in connection with persistence: Externalization and internalization, object grouping, and invariant identification of objects. Object grouping (i.e. collection) is realized by means of methods to deposit, retrieve and delete objects into and from libraries, respectively. Object identification is implemented with a dictionary mechanism that allows an object within a library to be identified by a *name*.

Libraries exist in two variants which differ in their accessibility: Public and private. On the one hand, *public* or named libraries can be accessed by any authority in the system. They allow to share objects across document and library boundaries. Hence, several documents may share a collection of objects. As a consequence, modifications on objects are immediately available to all clients. On the other hand, *private* or anonymous (i.e. unnamed) libraries are local to some *host* (e.g. a document) and cannot be accessed from "outside". If an object is shared across boundaries of private libraries, each library will eventually have a distinct copy of the object, if its host document is stored.

Public libraries allow a invariant reference scheme for objects of the form *L.O*, where *L* is the name of the library and *O* the name of the object. This naming scheme exactly corresponds to that of a unit of execution in the Oberon system (the Oberon command) which has the form *M.P* where *M* is the name of a module and *P* the name of a command procedure.

Integration of Objects into Text

The original motivation for the evolution of the Oberon system was the seemingly harmless question of how to integrate figures and graphics into a text in a conceptually clean way. In the Oberon system *Text* is a fundamental abstract data type and belongs to the outer kernel of the system. It is looked at as a sequence of *attributed characters* with attributes *type-font*, *color* and *vertical offset*. The Ascii code of a character is now interpreted as an index into a data set of graphical representations (character patterns) that form the look of a character. A detailed discussion on Oberon's text machinery can be found in [WiGu92].

It was obvious that the integration of generalized objects such as figures and graphics into a text had to be achieved in a way that allowed objects to flow freely as elements within a stream of characters. Several different solutions for the integration have been presented. They mainly differ in the level on that the integration takes place, i.e. how deeply the new concept is situated in the system hierarchy. In the *Write* editor package [Szy91] a special object type called *Element* has been introduced. Elements are able to flow within the text like ordinary characters. For that purpose, the definition of a text has been conceptually extended to a sequence of either normal characters or elements. In this model, elements within the text need special treatment by the text machinery. Therefore, elements have been defined and integrated at the hierarchy level of *Texts*. This, of course, implies that elements exist only in textual environments.

Supertexts

In Oberon System 3, a more general approach has been chosen. A simple shift of emphasis in the interpretation of type-fonts led to an elegant and flexible solution. Instead of looking at the font as an *attribute* (of a character), it is interpreted as a *collection* (of character objects). Such a collection is a special variant of a *Library* that is indexed by the character's Ascii code. This interpretation leads directly to generalized texts, called *Supertexts*, that are *sequences of objects* that can be of various types (e.g. pictures, graphics etc.). Of course, character objects will very probably be the most frequent ones. They are specified by pairs of (character code, library). Because objects are located sufficiently low in the system's hierarchy, objects may occur in texts as well as in any other environments.

There are two different ways of including an object in a context: *Inclusion by reference* and *inclusion by integration*. An object may appear in several different texts although it physically exists as a single instance only (e.g. character patterns of a font). This case is implemented by *reference copies* where a reference copy is a reference to an object rather than the object itself. In a text, the reference copies are represented by pairs (character code, library). Inclusion by reference allows objects to occur simultaneously at different locations in the same or other documents. Note an object must be contained in a public library if it is intended to be referenced from different documents.

A *clone* is an object that emerges from the creation of an identical but independent instance of an original object. It has the same initial state as the original object but any modification of its state will only affect the clone. In contrast to reference copies, *clones* are included into a text by *integration*. They

must be stored in the document's private library. It is the responsibility of the corresponding document to maintain its private library.

Script

Supertexts are displayed and manipulated with the help of a tool called *Script* [So94]. Script is a simple editor for formatted text. It does not support a sophisticated document model but is "wysiwyg", and in particular guarantees an identical line breaking on the display and printer.

Supertexts are presented on the screen by a special object class called *ScriptFrames*. Its main purpose is twofold: (a) Mapping of the sequential text to a two dimensional display area (formatting) and (b) support for interactive manipulation of its contents (editing). Editing operations comprise inserting, deleting and copying of text pieces and text attributes. Graphical objects that appear in the supertexts can be displayed, sized, moved and edited *in-place*, i.e. without having to switch to a dedicated graphics environment. Due to the object oriented design and implementation of ScriptFrames, object management basically confines to forwarding all user requests to the corresponding object.

Alternatively to being displayed, objects that flow in a text stream may be interpreted, and thus can have an impact on their context. Script supports a special variant of such objects called *style symbols* that control the look of a subsequent text section. While *font*, *offset* and *color* define the look of a document at character level, style symbols allow to control the look and format of the document at the level of paragraphs. The notion of flowing format style symbols has been borrowed from the *Write* editor package [Szy91]. Attributes of style symbols include left and right margins, formatting mode (left and right adjusted, block mode and centered mode), line spacing, tabulator stops and forced page breaks. Style symbols can be copied to different text locations (an even to different documents) like any other text piece and therefore may be used in different text sections (reference copy). Any change of parameters of a style symbol thus applies to every position where it occurs.

The collection of all style symbols of a document defines its overall look. In addition (and beyond the possibilities of the Write editor), symbols representing a certain style may be grouped in a *style library*. Predefined ready-made style symbols collected in public style libraries can thus be used to define a uniform look of a set of different text documents.

This thesis, for example, has been written and printed completely using Script. Thereby, each chapter is a separate document. However, all chapters

share a uniform look defined by a small number of prepared style symbols that are collected in a common public style library.

Document Oriented Interface Model

In commonly used operating environments, user interfaces are an integral part of the corresponding applications; that is, they aren't viable outside of the application's context, and are always loaded together with their application. In the Oberon system, however, user interfaces are autonomous documents that can be opened without having to start their applications in advance. This model can be called *document oriented interface* (DOI). It is one of the major characteristics of Oberon's user interface.

As stated earlier, the *command* is the unit of interaction in Oberon. A command is activated directly from the user environment by clicking the mouse at its name. A command name *M.P* consists of two parts *M* and *P* where M is a module, and P is a procedure in M. Such commands and their parameters are usually prepared and collected in freely customizable menu text documents, so called *Tools*. Tools are opened, edited and stored like any other textual document in the Oberon system. In addition, commands can be included in (and executed from) ordinary text documents. For example, it is common practice in Oberon to include ready-made command lists in electronic mail messages for a convenient execution by the recipient of the mail. The set of all text documents containing commands can be looked at Oberon's textual user interface (TUI).

In addition, Oberon provides a graphical user interface management system called *Gadgets* [Gu94]. It again applies a document oriented approach. A user interface *panel* is a graphical document that contains a collection of user interface elements like buttons, text fields and sliders (collectively denoted by the term *gadgets*). A panel is interactively opened, composed and stored like any other document. In principle, panels are autonomous. Their composition does not assume the presence of any application at all. However, applications can later be bound to a panel (or a panel can be bound to an application).

Functionality is attached to gadgets (buttons, etc.) again by means of commands. For that purpose, gadgets are equipped with an attribute that indicates the command to be executed when the gadget is clicked at. In fact, the Gadgets user interface can be seen as a graphical clove of ordinary commands, although it extends the capabilities of the original user interface significantly (e.g. by automatically collecting parameters from the different user interface elements).

Binding activities to graphical items by means of command attributes is common practice in the Gadgets graphical user interface and one of its sources of flexibility. This dynamic way of coupling activities with gadgets allows to customize every single type of graphical items for several different purposes. For example, two buttons of the same type initiate different actions if different commands are attached.

Services and their Representations

We define a *service* as an authority able to consume, process and deliver on demand certain well-defined data items. The entity that operates a service is called *server*. Users of a service are called *clients*. The service's environment is called client/server environment. The interaction between clients and servers usually happens in the form of request/response pairs (where the response occasionally may be left out or restricted to an acknowledgement, for example in the case of a print service). Requests for processing or delivering data items are always initiated by clients and processed by servers.

The spectrum of possible services ranges from simple file and print services over static local data bases such as electronic encyclopedias to highly dynamical broadcast information services like *Teletext*, or global network information systems like the *World-Wide Web*.

Implementing Access to Services

We repeat that the construction of client applications for interacting with services involves aspects which can be roughly separated into three groups as listed below:

a) communication protocols between server and clients,
b) the design of a stub as a representation of the service on the client's side, and
c) the user interface model on the client's side.

Usually, an implementor of a client application is confronted with all of these aspects in the above order.

Aspects in the first group are related to the interaction with services and hence with the processing of transactions. Transactions between services and clients involve an exchange of requests and responses whose structure and sequence is called *protocol*. These aspects also include the data representation (encoding and decoding) of the information to be exchanged.

The third group collects aspects of the integration of services into the user environment. This includes the definition of an appropriate user interface model. Favorably, the user should not be forced to learn a different user interface for every different service. Ideally, the user interface model should provide a single, consistent interaction mechanism that applies equally to any arbitrary local and remote service. A seamless integration also requires a representation of the service's data items by facilities and components of the user environment. In such a case, existing applications may immediately profit from the new possibilities (i.e. consume and process such items).

Aspects (a) and (c) can be bridged by an abstract representation of the service on the client's side whose purpose is hiding all service-specific technical details and providing a uniform interface to the different services. *Uniform* in this case means that a similar access mechanism is applied to every service, irrespective of the internal details.

Representation of Services on the Client's Side

Abstracting from internal details of a service, the interface of a service can be defined by two intrinsic operations: Sending an item and requesting an item. *Sending* an item to a service implies that the service is ready to consume and evaluate it. *Requesting* an item from a service implies that the service is ready to create or retrieve it. Note that the actual communication between the client and the server (such as connection establishment etc.) are regarded as implementation details of these basic operations. We assume that these operations completely encase transactions; i.e. there is no service related state that survives any of the operations (so called *stateless* transactions).

Of course, this abstract representation relies on an appropriate individual implementation of these operations for every service. However, once an implementation is provided for a desired service, a client needs no longer care about details of protocol and transport but merely uses the two basic operations for communication with this service.

A service is thus conceptually split into an abstract interface and implementation that resides on the client's side and the actual service that resides on the server's side. The following figure illustrates this situation:

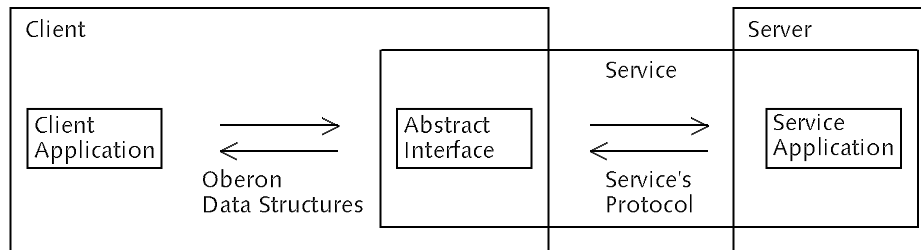


Figure 1: Client/Server Structure of a Service

Because clients interact with the abstract interface of a possibly non-local service, rather than with the service itself, internal Oberon data structures can be passed via the abstract interface. The client representation of the service ("local" from the client's point of view) thus consumes and produces Oberon objects while encapsulating and hiding all service specific behaviour such as protocols, encoding and decoding behind its abstract interface. As from the client's point of view the service's representation stands for the service itself, we don't distinguish between the two from now on and call the abstract interface *Service*. It is modelled by means of the following data type:

```
TYPE
   Service = POINTER TO ServiceDesc;
   ServiceDesc = RECORD(Objects.ObjDesc)
      Send: PROCEDURE(S: Service; parobj: Objects.Object; par: ARRAY OF CHAR);
      Request: PROCEDURE(S: Service; parobj: Objects.Object; par: ARRAY OF CHAR):
                                                                Objects.Object;
   END;
```

Type Service is an extension [Wi88] of type *Object* with two additional methods for sending and requesting objects. The *Send* method of the service allows to deposit an object for being consumed and processed by the service. The desired operation is specified by a string parameter. The *Request* method allows to retrieve an object from the service. The requested object (or operation) is again specified by a string parameter. If the requested operation depends on an object (for example on a mathematical formula to be evaluated) it may be supplied with the request as a parameter. Formally, an object is requested from a service *S* in the following generic way:

```
resultObj := S.Request(S, parObj, par);
```

and sent to a service *T* like this:

```
T.Send(T, obj, par);
```

Remember that different services only differ in the actual implementation of the corresponding send and request methods.

Generic Naming of Services

For a generic interaction with services, the concept of an abstract interface is a necessary but not yet sufficient requirement. While abstract interfaces allows to *use* services in a generic way, a method is still required to *identify* a service in a generic way. For that purpose, a facility has been constructed that allows to retrieve services by their name. The following *GetService* procedure implements the generic naming facility:

```
PROCEDURE GetService(name: ARRAY OF CHAR): Service;
```

Services are created like any other object in the system in a generic way by means of a so called *generator method*. The naming facility is based on this technique: The mapping of names to services is done by generator methods that create instances of services.

As examples, the following lines list accesses to different services, each one identified by its name (given by generator commands):

Request plot *of formula* fm *from the Symbolic Algebra System* Maple

```
maple := Services.GetService("MapleEngine.New");
plot := maple.Request(maple, fm, "plot");
```

Lookup keyword "Oberon" in Encyclopedia

```
dict := Services.GetService("Encyclopedia.New");
obj := dict.Request(dict, NIL, "Oberon");
text := obj(Texts.Text);
```

Online Documents

A document in the traditional sense is a static and self-contained collection of data (a "model") that represents a certain unit of information. The model can be a text, a graphic, a spreadsheet, a video sequence etc. or any combination

thereof. If a document is being "opened", the model is represented on the display in a certain way. Although mapping the model onto the screen may consume significant computing resources (e.g. in the case of on-the-fly decompression of video data), it does not produce any information that is not already present in the model.

The notion of *online documents* generalizes both the model and the mapping of traditional documents on some display space. The model of online documents is possibly distributed over several locations (non-locality of the model). Non-locality implies that the model must be collected before representation, for example via network connections. Additionally, the model does not necessarily need to represent a static state of information (dynamic model). If an online document lacks a well-defined, static global state, (static) views must be computed "on-the-fly" at the time of the document's mapping on the display. A specific view of an online document merely represents a *snapshot* of its current state or, more general, if the mapping itself depends on certain parameters, a *projection* of the current state.

The mapping process of online documents involves either compiling the model or constructing a static view. Each of these tasks requires a program that processes the corresponding task. If we interpret as services the programs that produce the mapping, an online document can be seen as an abstract representation of information that is managed, updated, mediated and visualized by means of services. Hence, we can view an online document as an abstract representation for services.

Examples are the *World-Wide Web* and *TeleNews* (for internal details refer to the corresponding case studies). The model of the World-Wide Web is a distributed pool of hypertext pages, bitmap images etc. Its mapping involves the collection of these items via network connections. Both collection and rendering are performed by applications called *browsers*.

TeleNews is an advanced and expressive representation of Teletext data in the form of an electronic newspaper. The teletext data base is perpetually collected and maintained by a centralized remote server. Static views of the data base are compiled upon request into hypertext overview lists and news articles.

Smart Links

Links are a popular means to abstractly relate arbitrary data items and documents to each other. If a link is activated, the related data item is fetched and processed in an appropriate way. A picture item, for example, is probably

displayed on the screen, whereas a binary data file is downloaded onto the local disk.

In Oberon, abstract relations are usually established with prepared commands that are contained in documents. For example, an electronic mail message that informs the clients about a new release of a software component may directly include an abstract reference to the component. The corresponding reference usually consists of ready-made commands that allow the downloading of the component from a remote file server. Thanks to Oberon's document oriented interface mentioned earlier, these commands can be executed directly from the mail text. For the future, it is planned to physically link (i.e. attach) the component directly to the mail.

In so called *hypertexts*, links provide a technique to structure large documents and to build a network of hierarchical and semantical relations.

Hypertext

Hypertext [BeDe91] is a special way of organizing and presenting textual information that allows the reader to navigate hierarchically or freely through the text. Hypertext is a simple and powerful concept for both structuring and activating text. By simply pointing at highlighted keywords, the corresponding information is fetched and displayed. This behaviour gives the reader the impression of "zooming" into subjects of interest, which proves to be a very efficient way of retrieving information. Hypertext facilities have become very popular. There is hardly any online manual worth mentioning which does not rely on hypertext mechanisms in order to allow the reader to "browse" through the contained information.

The basic idea of hypertext is to enhance the pure sequential structure of text. Instead, references to different text sections, so called *hypertext links*, allow the construction of complex reference structures. Besides acting as a structuring tool for splitting the information into easily readable portions of text, hypertext links allow for building a network of logical and semantic relations.

The links of a hypertext document are directed edges of a graph structure with text sections as nodes. The hypertext graph can be traversed along any of its directed links. Usually, a "context stack" supports backtracking from previously traversed links in the reverse direction.

A link consists of a source anchor and a destination anchor, denoting the origin and the end point of the link, respectively. A source anchor is usually indicated by highlighting a keyword, or by adding a symbol such as the reference arrow that is familiar from dictionaries (e.g. →keyword).

A hypertext look-up is processed as follows: If a link anchor is activated, e.g. by clicking at it with the mouse, the hypertext platform (often called hypertext *browser*) identifies the type of the hypertext link and uses it to address the referenced text section. Possible link types are the reference link pointing to a different text section or, for example, a link yielding a popup note.

Service Links

In [BLCa90] it has already been recognized, that *"potentially, hypertext provides a single user-interface to many large classes of stored information such as reports, notes, data-bases, computer documentation and on-line systems help"*. In other words, by interpreting the link activation as an elementary user interface operation, hypertext can be seen as a unified and integrated model of a user interface for a large class of different textual applications. This kind of interpretation of hypertext in general and the link activation in special has led to network services and other applications that more and more present themselves as active hypertext documents.

It is easy to recognize that an ordinary hypertext lookup scheme as presented above is not general enough as the only basic operation of a user interface model. The reason is that the knowledge about how to interpret a specific link type needs to be contained and centralized in the browser.

Conceptually, link anchors serve as *tokens* defining both the presence as well as the type of a hypertext link. This property requires that the hypertext system "knows" all existing types of links in order to interpret them and react accordingly. Consequently, the actual acquiring and displaying of the referenced text sections is performed by the browser. Such a hypertext system is inherently non-extensible. The addition of new link types and thus of new ways of interpreting links always implies changing the original code. If extensibility is an indispensable requirement, this model of a general user interface is therefore inappropriate. Thus, a more general notion of an elementary operation has to be found. A solution that takes away the burden of service-specific aspects of the link activation from the browser consists of a reinterpretation of link activation as a request to a specific service. Such an extended view is inspired by the similarity of a link activation and an access to a (such as request-response schemes). In this way, links generalize naturally to service calls. In other words, service links are active entities that are *requested* to deliver the associated data item rather than *interpreted* by the browser.

In this light, it is natural to represent such a service link internally as an abstract service. Such internal representations of service links are called *Smart Links*. They are formally defined as follows:

```
TYPE
   Service = POINTER TO ServiceDesc;
   ServiceDesc = RECORD(Objects.ObjDesc)
      Send: PROCEDURE(S: Service; parobj: Objects.Object; par: ARRAY OF CHAR);
      Request: PROCEDURE(S: Service; parobj: Objects.Object; par: ARRAY OF CHAR):
                                                                  Objects.Object;
   END;

   SmartLink = POINTER TO SmartLinkDesc;
   SmartLinkDesc = RECORD(ServiceDesc)
      Resume: PROCEDURE(link: SmartLink)
   END;
```

A smart link is modelled as a type-extension of a service. It inherits the service's *Request* and *Send* methods and adds a *Resume* method that is used for the implementation of a generic context stack that allows traversing links in the reverse direction. With the help of the context stack, the user can conveniently return to a previously visited context (page).

A generic context facility for resuming contexts is necessary, because in a heterogeneous document environment (heterogeneous with respect to services) activation of a link may result in a switch to a totally different context, for example, from a World-Wide Web page to an online encyclopedia page. Usually, a context is indicated with titles etc. which are changed whenever a new page is presented. The setting and resetting of context dependent attributes must be handled by the link itself, because the link processor has no information about contexts. For symmetry reasons, the link thus handles both the entering as well as the leaving of a context.

A Document Oriented User Interface Model

In Oberon, documents serve as containers for textual user interfaces (tools) and graphical user interfaces (panels), respectively. In both cases, the user interface relies on the command as the unit of interaction. In the textual case, the command name is pointed at directly, whereas in the graphical case, the command is installed in a graphical element as one of its attributes.

Regarding our experience with documents as a basis for user interfaces in Oberon, we have designed a similar model for the interaction with arbitrary services. In fact, by defining the service call as the unit of interaction, Oberon's document oriented user interface model can be easily adjusted to the desired generalization.

Our document oriented user interface model for services is based on smart links as a primitive. The corresponding service call (which is an implicit result

of the smart link's activation) may, for example, result in getting and presenting a new page of a hypertext based service. Because smart links are special variants of objects, they can be included directly in the new text page. In that way, the returned text represents the user interface for a subsequent activity.

Documents containing smart are called *Smart Documents*. They unify and integrate both the *information* that is delivered and presented by the services, and their *user interface*. Smart documents thus provide a consistent and extensible user interface model.

Consistency

Smart documents incorporate a consistent representation of an online document and its user interface. Any mismatch between the presented information and the expected behaviour of a document is impossible because the behaviour is concentrated in the links. Consider the following example: An excerpt of an encyclopedia or a dictionary that contains references to further keywords. For any human reader it is obvious that the keywords refer to further encyclopedia entries. Consequently, the browser application must remember the context of the displayed page in order to be able to direct further activities to the appropriate service (in this case an electronic encyclopedia).

In a heterogenous online document environment of online documents (heterogenous with respect to (possibly unknown) links) it is possible that an activity (e.g. the activation of a link) results in a switch of context so that the following user activities make use of a different service. In our document oriented user interface no provisions have to be taken in such a situation: The services that process the (different) activities are contained as links *within* a page of the corresponding context.

Extensibility

The activation of a smart link implicitely has the side effect of a generic service call that allows access to new and yet unknown services. Smart links therefore constitute the basis of a potentially unlimited extensibility of the document oriented interface model. A smart link be seen as a combination of an Oberon command and an abstract service.

Generic Link Processor

Extensibility of the document oriented interface for services relies on a generic link processor. Processing activities within an ordinary hypertext system

requires a specific interpreter that is able to recognize different kinds of hypertext links. In contrast, smart links are *services* which can be invoked in a generic way. In other words, the interpretation of links is replaced by (generic) service calls. Hence, we can outline the implementation of the universal link processor of the document oriented user interface as follows:

```
ProcessLink = {
   get text and position of the user activity within online document;
   if keyword at position is highlighted then
      find first link object after position within text;
      request new item from link;
      if returned item # NIL then
         if item is a text then
            push current text context on context stack;
            replace current text by text section received from link
         end
      end
   else
      execute keyword as Oberon command
   end
}
```

The core of the above link processor is requesting a data item from the smart link. It is implemented as a service call. In principle, the smart link is fully autonomous in the handling of the item returned by the service. Typically, it can return a data item to its interpreter. For example, if the service returns a text, the interpreter replaces the currently visible text page by the returned text in a hypertext-like style. A link may also decide to open a view of the returned data item by itself and not to return any object to the interpreter. For example, links to the symbolic algebra system never return a data item. Instead, the result of an evaluation is directly appended by the smart link to some specific text view that acts as a "sketch paper".

Similarly, the link type handling World-Wide Web requests opens a separate picture view, if the requested item is an image. If the returned item is a text, the smart link still returns it to the interpreter.

The above described special processing of text items by the link processor offers a hypertext-like functionality that can be seen as a special support for a frequently occurring case.

Furthermore, it is noteworty that the conventional processing of unhighlighted keyword as Oberon commands is preserved by the interpreter.

Visual Representation

Up to now, we have not yet fully discussed the visual representation of smart documents. This was intentional, because service related issues should not affect the visual representation of smart documents at all. In fact we shall decouple all the aspects of the smart document's representation on the screen from the link processor. This is comparable to the decoupling of service related aspects (protocol etc.) by encapsulating them in the implementation.

In summary we have the following aspects in connection with smart documents: Invoking links, calling services, and rendering documents (on the display). This leads to the control flow as sketched in the following Figure:

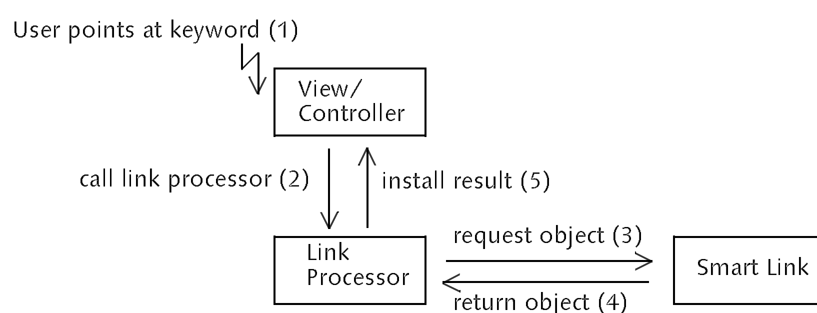


Figure 2: control flow of generic link processing (the sequence of activities is given in parentheses)

Note that the entity that is responsible for the display smart documents (denoted with *View/Controller* in the figure) is involved in two activities: (a) Calling the link processor (step 2), and (b) (possibly) receiving the result of the link activation. Both activities can be performed in a generic way, so that a simple adjustment of ordinary Gadgets text views readily meets our requirements:

Invocation of the link processor. Smart document views differ from ordinary text views by a specific behaviour upon user activities: They invoke the generic link processor for calling the service that is attached to the link. In a graphical user environment that is based on the *Model/View/Controller* ("model" = data, "view" = representation of model, "controller" = handler of interaction) separation, a different behaviour can be represented by a different controller. The Gadgets system allows to parameterize the controller action by a command attribute. Therefore we are done with simply setting the command attribute of a Gadgets text view to the generic link processor.

Display result of the link activation. A message interface is used for the communication between the link processor and the document's view. The following *attach message* notifies the view manager of the text that is to be displayed:

```
AttachMsg = RECORD(Objects.LinkMsg)
  pos: LONGINT
END;
```

The *attach message* is a type-extension of a standard *link message*. The latter is used to retrieve or set the model that is "linked" to a view.

```
LinkMsg = RECORD (Objects.ObjMsg)
  id: INTEGER;
  name: Objects.Name;
  obj: Objects.Object;
END;
```

The link message contains fields *id*, *name* and *obj* which specify the desired operation (set or get), the model's name and the actual model, respectively.

The *attach message* extends the *link message* by a position that indicates the origin of the text view (i.e. the position of the first character (or object) that is displayed).

Note that a text view which handles the standard *link message* but not the extending attach message is still able to display and process smart documents. However, it will only a reduced functionality offer (no automatic positionning of the text).

Generic Context Stack

The bove introduced attach message allows it to build a general mechanism for the maintenance of a context stack. Because text pages that result from a smart link are installed indirectly at the smart document's view via the attach message, the context stack's data structure cannot be anchored at the view. It is therefore anchored directly at the text page to be installed. Hence, by definition, a new text page always represents the head of a chain of previously displayed pages (the context stack).

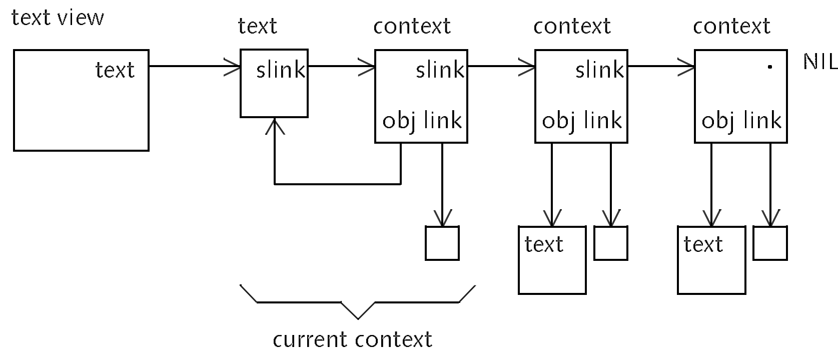


Figure 3: data structure of the context chain

The context stack is a linear list of elements each representing a context, i.e. a text page that lies on the traversed path to the currently visible page. A context consists of the text page and the link which produced the page. If the previous context is resumed, the *resume* method of the corresponding link is executed in order to allow context titles and other context specific information to be restored.

Example Service Integration with Smart Links

The link activation scheme based on smart links allows the integration of arbitrary textual services into one and the same document oriented interface. In the next chapters, the emphasis lies on concrete implications and on service-specific aspects of such an integration. To illustrate the application of links in the case of a concrete class of online documents, we subsequently describe the integration of an electronic encyclopedia.

The *electronic encyclopedia* is based on a commercially available low-cost encyclopedia (Knaur's Lexikon) containing explanations about 40'000 keywords by about 100'000 lines of text. In connection with its integration, we are interested in the raw data only. The browser application supplied with the encyclopedia package is therefore not required. (In fact, the low-cost solution has been chosen mainly for a mere practical reason: It was assumed, that a "cheap" encyclopedia lacks a sophisticated data file encoding scheme that is hard to crack).

Interaction with the online encyclopedia is based on the *keyword lookup* as an elementary operation. It happens in one of two ways: Either a keyword is selected anywhere on the screen, or a reference to a keyword within a page returned by the encyclopedia is activated. In either case, the result of a link activation is an encyclopedia page that contains the explanation of the keyword.

The implementation of the actual *request method* is listed below. It is assumed that the encyclopedia lookup and the generation of the resulting page are performed by the procedures *Lookup* and *WritePage*, respectively. Their detailed implementations are irrelevant in this context and therefore are not explained any further. It suffices to note that the *WritePage* procedure parses the text lines that are returned from the encyclopedia data base, and substitutes all "arrowed" keyword references by smart links. These are then integrated into the text.

```
PROCEDURE RequestPage(enc, obj: Objects.Object; par: ARRAY OF CHAR): Objects.Object;
  VAR entno: EntryNumber;
BEGIN
  Lookup(par, entno); WritePage(W, entno);    (*keyword lookup & page construction*)
  NEW(text); Texts.Open(text, ""); Texts.Append(text, W.buf);
  RETURN text
END RequestPage;
```

Note that the *Lookup* procedure is parameterized by the string parameter supplied with the method. This parameter is the keyword at the link anchor which has been clicked at. Because the keyword supplied as a parameter is sufficient to specify the lookup, the encyclopedia link does not need to store any further private data. It can be defined as follows:

```
Link = SmartLinks.Link;
```

Links are generated by the following generator method:

```
PROCEDURE New*;
  VAR obj: Link;
BEGIN
  NEW(obj); obj.handle := SmartLinks.Handle;        (*create encyclopedia link*)
  obj.Request := RequestPage;        (*install request method*)
  Objects.NewObj := obj
END New;
```

All text based services that are integrated and accessed by means of smart links are implemented in a way similar to that of the encyclopedia service. In fact, they mainly differ in the methods of requesting and processing the information returned from the service, i.e. in the implementations of the *Lookup* and *WritePage* procedures in the example above. Therefore, a new service's integration in principle confines to the implementation of these (service-specific) procedures.

Related Work

Acme [Pi94] is a document oriented interface designed as a *User Interface for Programmers*. It is a windowed user environment especially suited for text based applications. Acme presents itself on a display in a hierarchically tiled form with tracks and windows that are organized in a way similar to Oberon. Acme adopted a command activation method comparable to Oberon, i.e. using clickable text captions that represent activities (built-in Acme commands or programs).

One of Acme's major characteristics is a generic interaction method for services that is based on a file system interface. On one hand, arbitrary clients can access the offered facilities and abstractions by reading or modifying the contents of abstract files. On the other hand, services can be integrated into Acme's user environment and accessed with its tools, if they offer their facilities in the form of abstract files. Note that files need not necessarily be associated with disk space. Instead, a file can be viewed as an abstract service with a well-defined set of intrinsic operations including *reading from* and *writing to* a file. This generalized concept of files is frequently used in UNIX-like systems [ThRi74] e.g. for representing devices.

Acme's state and behaviour, for example the content of its windows, are controlled and represented by such generalized files. These can be manipulated (read and written) by clients in order to modify the window's contents, for example.

The integration of a mail reader into the Acme environment is accomplished by specifying a mail *file directory*. The mailbox entries are represented by mail files. The mail program simply monitors the user's mailbox and updates the mail directory accordingly. There is no need for a special mail browser, because, from Acme's point of view, reading mail is equivalent to reading an ordinary file. As a benefit, all applications operating on files can as well operate on mailbox entries.

In Oberon, the file is not an adequate abstraction for the modelling of a service. This on one hand is due to merely technical reasons (type File is not extensible). On the other hand, using files as a basis for arbitrary services would leave unexploited one of Oberon's sources of power, namely its ability to operate directly on global data structures.

In Acme, a centralized representation of services on a per-document base is used. The window's *current directory* identifies the service to which all activities within a window are directed. In contrast to Acme, we have a *decentralized* representation of services in our interface model. Services are represented by

means of smart links that are distributed in the document. Therefore, a smart document may contain links of very different kinds.

In [Ze88], a document oriented interface is presented that relies on a concept named *script* to construct so called *Active Paths Trough Multimedia Documents*. Multimedia documents are documents that integrate text, images, sound and animations.

A script associates an arbitrary timed action to a location within the document (for example the origin of an active path). The content of the script is usually interpreted without intervention of a user. This allows for example the annotation of links with sound patterns (such as a welcome text) or to play back parts of the multimedia document. Scripts activate *themselves*, i.e. there is no (global) media recorder that replays the multimedia document.

Active paths are the implicit result of the interpretation of scripts that connect two locations. These paths are more general than ordinary hypertext links. In fact a hypertext link can be expressed by a script that connects origin and destination without associating any action.

The destination of an active path does not need to be a physical location in a static document. Instead, the execution of a script may redirect the path to a new location, or even to a document that is computed during the execution of a script. For example, a path in a multimedia document may depend upon earlier activities in the sense that further explanations are abbreviated, if enough information has already be seen (history dependent scripts).

In contrast to our smart links, active paths may spontaneously perform activities after some predefined time, or immediately when they are displayed. Whereas smart links combine the behaviour of abstract services with the semantics of Oberon commands, active paths build on the semantics of Oberon background tasks (i.e. commands that are executed by the system when it runs idle). However, although active paths serve a different purpose (namely to activate documents), they can serve as a basis for the integration of links to services in multimedia documents. Such links are implemented by scripts that call the corresponding services and construct the document the active path leads to.

Conversely, the implementation of active paths by means of smart links would allow the integration with our document oriented interface of the two most interesting aspects of multimedia documents, namely history dependent link processing and automatic link activation.

History dependent link processing would require the maintainance of a list of visited documents. If a smart link is invoked, it could inspect the "history list" to decide which activity to perform (e.g. which document to return). Timed

activation of smart links could be implemented by a timer that is attached to the link. As soon as the link is displayed, the timer would be started. After expiration of a predefined time, the link would be activated by the link processor as if it had been activated by the user. Timers could easily be implemented with the kind of tasks mentioned above, as tasks already have a timer value associated with them.

Summary

In this chapter we have presented a universal document oriented interface for a unified and consistent model of interaction with arbitrary local and remote services. An Oberon object type that we have named *Service* serves as a unifying concept for the representation of arbitrary services on the client's side. This concept offers an abstract and generic accessing scheme for arbitrary services, and allows passing abstract data structures to the service. All involved tasks such as encoding, sequentializing etc. are hidden behind a uniform message interface.

We have introduced the notion of online documents and related it with traditional (static) documents. Online documents provide an powerful model for all kinds of information that is collected or produced by services.

A new interpretation and generalization of a hypertext links led to the concept of *Smart Links*. These are an object-oriented and internal representation of service links, i.e. of links whose interface represents an abstract service.

By integrating smart links in (online) documents, we have constituted a universal document oriented interface for services. It is universal with respect to its potentially unlimited extensibility. Smart documents integrate both the information that is provided by services and their user interface. Based on the document oriented interface model, services are naturally integrated into the Oberon system. They supply the user with an interface that he is familiar with. In the latest implementation, for example, one and the same opening command applies for local documents as well as for online documents.

Case Study 1: Dynamic Mathbook

In the first case study we present the integration of an interactive formula editor and a symbolic algebra system as a *Dynamic Mathbook*. The integration results in an interactive mathematical platform allowing to modify and evaluate mathematical expressions directly within a document containing formulae. Applicable operations include mathematical transformations and the generation of graphical function plots. Activation of mathematical documents introduced by such an integration aims at a new class of interactive applications called *electronic textbooks*. These are textual information environments that allow the contained information not only to be consumed but interactively processed and experimented with *directly within the document*.

Introduction

If we take a look at the evolution of information media from printed matter to multimedia applications, we can observe several steps of (inter–) activation of the transported information. The concept of multimedia currently forms the last and at the same time farthest reaching step. Printed material allows almost no interaction at all, besides the possibility to manually turn over the leaves. The support of fast keyword indexing facilities within an "electronic book" provide support for interactive searching in large data bases. The concept of multimedia even activates the data itself by allowing play-back video sequences, sound patterns and animated slide shows.

Multimedia as well as hypertext (whose unification is occasionally denoted by the term *hypermedia*) are very effective concepts for activating static information. However, there is a level of interaction which has not yet been developed extensively in the fields of interactive computing in general and hypermedia applications in special. We allude to the ability to *process* the information presented in the text *itself*.

Essentially, the only operation applicable to multimedia documents is replay. Hence, the flexibility that lies inherently within the concept of software is not exploited. What we really want is an *interactive extension* of multimedia

platforms allowing to interpret and manipulate the information they mediate. In particular, this is true for mathematical expressions within a text.

It is a usual feature of sophisticated text document editors that graphical objects like figures and pictures can be integrated into the text. Sometimes they can also be manipulated *in place*, i.e. without having to switch to a different application. The set of such objects often includes mathematical formulae with their typical two dimensional rendering. The different formula preparation systems mainly differ in the way formulae can be constructed and manipulated. The mathematical meaning of a formula is normally irrelevant in such systems.

Conversely, today's symbolic algebra systems have reached a state, where they cope with almost every mathematical problem of whatsoever complexity. However, their formula manipulating and rendering facilities has not always reached a satisfactory state (*"For the most part, it is still true that programs that format mathematics can't do mathematics and programs that do mathematics can't do good math editing."* [Leong90]). The integration of a formula manipulating system and a symbolic algebra system can close the gap between manipulation and rendering of formulae on the one hand, and their evaluation on the other. But their integration aims beyond that by opening the door to a new class of interactive applications which allow their contents to be processed, rather than only to be consumed. Such an integration, by the way, manifests itself as a new level of interaction in particular in the fields of formula processing. The levels under discussion are (ordered by increasing interactive power): Formula creation by means of markup sequences (no interaction at all, e.g. TEX [Knuth84]), direct *manipulation* (cf. [Schär91]) and finally direct *evaluation* of formulae within the document.

The Dynamic Mathbook

The Dynamic Mathbook is an online document that is linked to a symbolic algebra system. Mathematical formulae that appear in the document can be interactively edited *in place*, i.e. without having to switch to a dedicated formula editor. Furthermore, formulae can be evaluated directly within the document. Evaluation results in a formula again, which can immediately be modified and processed further for interactive experimentation with the expression.

The Dynamic Mathbook's functionality essentially consists of two operations that can be performed on formulae. They correspond to the two possible resulting object types: *Evaluating* a formula results in a *formula* that represents the mathematical value of the expression, whereas *plotting* a

function results in a *graphical figure* which displays its geometrical shape. If such an activity is initiated, a text view is opened that acts as an electronic "sketch paper" where the results are displayed.

The user initiates activities within the Dynamic Mathbook by selecting an expression and executing one of the Dynamic Mathbook's commands. The user may either evaluate the formula or compute a graphic plot of it. The formula is then passed to the symbolic algebra system for evaluation. The returned object is appended to the sketch text.

Evaluation of a formula means that the given expression is simplified. Simplification usually includes evaluation of complex expressions such as integrals etc. It is performed by selecting a formula somewhere within a document and executing the *Evaluate* command of the electronic math book. A special *Apply* command allows computing an arbitrary function with the selected expression as an argument.

$$\frac{\sin(x)+x^2}{2^x-x}$$

formula

$$x+(2-\ln(2))*x^2+\left(-\frac{1}{6}-\frac{1}{2}*\ln(2)^2+(-2+\ln(2))*(\ln(2)-1)\right)*x^3+O\left(x^4\right)$$

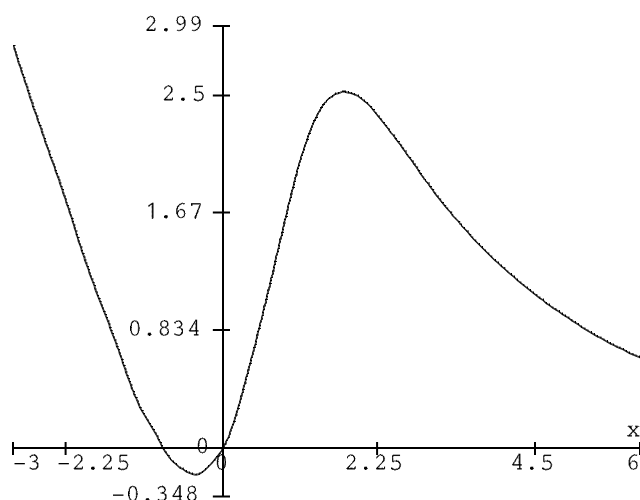
taylor series of formula

$$\frac{\cos(x)+2*x}{2^x-x}-\frac{\left(\sin(x)+x^2\right)*\left(2^x*\ln(2)-1\right)}{\left(2^x-x\right)^2}$$

differentiation of formula

$$-\frac{-\cos(x)*2^x+\cos(x)*x-2^{1+x}*x+x^2+\sin(x)*2^x*\ln(2)-\sin(x)+x^2*2^x*\ln(2)}{2^{2*x}-2^{1+x}*x+x^2}$$

simplification of above



graphic plot of formula

Figure 1: Sample session with comments

This is useful, for example to compute a Taylor series of an expression that cannot be simplified by the Evaluate command.

A graphic plot of an expression is computed within the Dynamic Mathbook by selecting a formula and executing the *Plot* command. A plotting range may be specified with the command.

The graphical figure which results from the Plot command is appended to the sketch paper like any other result of the math book. The figure can be manipulated (e.g. copied to a different document) like any other graphical object in Oberon.

In the case of three-dimensional graphical plots, a specific viewer object is returned which visualizes polygonal descriptions of geometrical shapes.

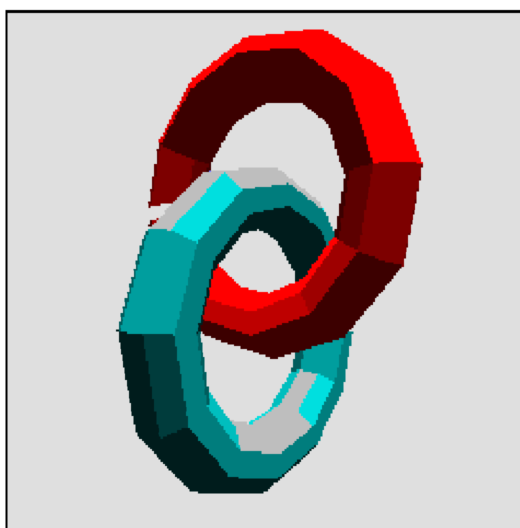


Figure 2: Polygon viewer showing two ring-shaped objects

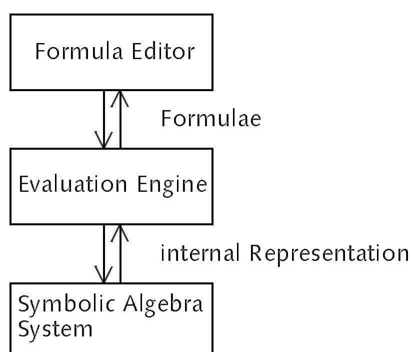
With the help of this viewer, the user can interactively move around in the visualized "scene" and inspect the three-dimensional graphic plot from different distances and angles without having to recompute the shape of the object.

Integration

The symbolic algebra system is a service that consumes formulae and produces either formulae, or graphical objects (if a plot is computed). To integrate a formula editor and a symbolic algebra system, a data representation is required that allows exchanging objects between the two applications. Each symbolic algebra system provides a specific (private) access protocol and data representation. To transparently interact with arbitrary symbolic algebra systems, an abstract interface is required that shields clients from the mathematical services' internal access methods. How this abstract interface looks like, depends on the architecture of the mathematical service.

Local Service Architecture

An interface that acts as an *evaluation engine* for formulae is inserted between the formula editor and the symbolic algebra system. It transparently accepts and produces formulae. However, instead of *evaluating* formulae, it merely *translates* them into the internal representation of the symbolic algebra system which in turn processes the actual computation. Hence, the evaluation engine shields the formula editor from internals of the symbolic algebra system in use.



local service

Figure 3: Architecture of a local mathematical service

With this architecture, it is possible to exchange the symbolic algebra system by adjusting the corresponding evaluation engine only. Furthermore, the evaluation engine can directly access the symbolic algebra system's facilities for translating formulae into its internal representation.

Remote Service Architecture

If the symbolic algebra system is residing on a remote computer, the architecture of the mathematical service splits into two parts which are connected by a network. Because the evaluation engine shields the formula editor from the internals of the underlying symbolic algebra system, it is reasonable to split the evaluation engine into parts which reside on different computers. Note that this architecture matches exactly the specification of the abstract interface as discussed in the previous chapter. The properties which the evaluation engine hides from clients hence include the non-locality of the service. The following figure shows the resulting structure of the mathematical service.

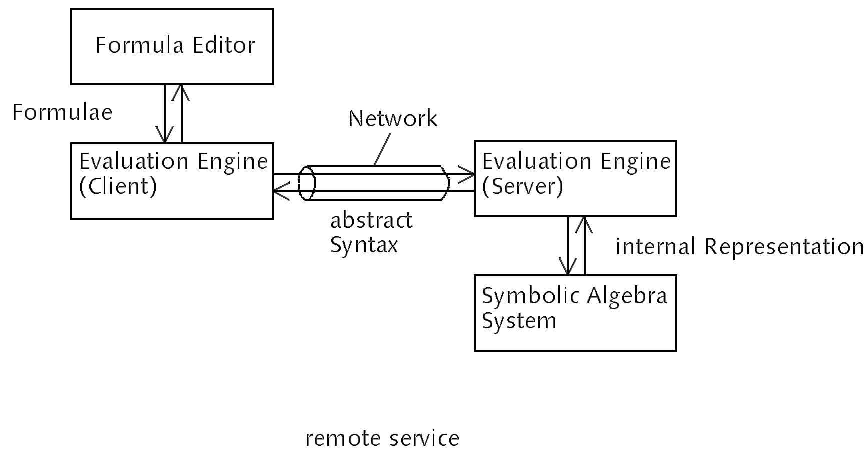


Figure 4: Architecture of a remote mathematical service

In order to let the two parts of the translator interact with each other through the network connection, a serialization scheme for mathematical expressions is required. Serialization methods for mathematical expressions are usually based on an *abstract syntax*.

Internal Representation of the Evaluation Engine

As we have seen, the evaluation engine forms the abstract "link" between the formula editor and the symbolic algebra system. From the formula system's point of view, the engine unifies both the local and the remote service's architecture by hiding their differences behind an abstract interface. Through the interface, the engine consumes and produces only formulae, irrespective of which internal format is used for processing or transmission to a remote service. Therefore it is natural to represent an evaluation engine as a *smart link*. Interaction with the evaluation engine thus happens by means of the link's abstract interface.

```
PROCEDURE Request(S: Services.Service; fm: Objects.Object; op: ARRAY OF CHAR):
                                                          Objects.Object;
  VAR argument, result: InternalRepresentation;
BEGIN
  argument := ParseFormula(fm);
  result := Evaluate(op, argument);
  fm := BuildFormula(result);
  RETURN fm
END Request;
```

The above *Request* method illustrates the principle of a formula's evaluation. First, the formula is transformed into an internal representation that is suitable for the specific architecture or implementation of the symbolic algebra service (for example, an abstract syntax for a remote service). Then, the requested operation is applied to the transformed expression. If no specific operation is specified, most symbolic algebra systems *simplify* the given expression. Finally, the result of the evaluation is transformed back into a formula (or figure, respectively, if a plot has been computed).

Implementation Aspects

Formulae

Formulae are typical examples of recursively defined objects. A formula consists of a formula symbol (e.g. a root or an integral sign) and several subexpressions, which are formulae themselves. Because of their recursive structure, an infinite multitude of different formulae may be constructed by means of only a small set of elementary formula types such as fractions, roots, integrals, powers and so on.

$$\sqrt[3]{x^2 + \frac{x - \sqrt{x^2 - 2}}{x + \sqrt[3]{2*x - x^2}}}$$

Within the text flow, a formula may appear as a separate paragraph or simply within a text line. If it appears within a text line, like $\sqrt{x+1}$, it is treated as a (complex) character, i.e. line breaks don't take place within formulae and in block adjusted formatting mode there is no distribution of blank space into the formula.

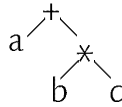
Adding Meaning to Formulae

The connection with a symbolic algebra system has implications on the formula system because of the necessity to interpret the formula's value. If the graphical shape of a formula is relevant only, its structure does not need to match its mathematical value. However, if formulae are to be actually evaluated, the meaning of formulae becomes essential. Any structural discrepancy of look and meaning must be strictly avoided to prevent evaluation errors.

Consider the following expression:

a+b⁎c.

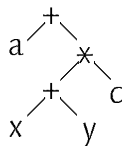
An appropriate internal representation that reflects the different binding strengths of the involved operators might be:



If we substitute the factor *b* in the formula by the expression "x+y", the formula reads

a+x+y⁎c

and has the following internal representation:



The tree, however, does not match the visual appearence of the expression *without the addition of parentheses*. The expression a+x+y⁎c reads as a+x+(y⁎c) while the formula tree implies a+(x+y)⁎c. The formula system must either automatically insert parentheses in the visual representation, or adjust the internal formula tree to match the mathematical precedence rules. Both solutions have advantages and disadvantages. Automatic modifications on the formula image may be very disturbing for the user. Adjusting the formula tree can be a very complex task (try to imagine the necessary modifications on the tree, if someone deletes the operator between the two identifiers a and x to get "ax").

Some formula editors avoid any discrepancy of look and meaning by allowing only predefined formula patterns to be expanded, guaranteeing thus the formula to be structurally correct at any time. The imposed lost of flexibility during the formula construction is significant (and might be considered obstructive).

We chose to avoid ambiguous *interpretation* of formula structures by avoiding ambiguous formula *structures* at all. Infix operators such as "+" or "⁎" in *expressions* don't impose ambiguities due to their implied structure (mathematical binding strenght). Formulae with a two dimensional placing of operands don't impose ambiguities, either. For example, the numerator and the

denominator of a fraction need not to be put in parentheses even if their precedence is lower than that of the fraction.

$$\frac{x+1}{x-1} \text{ instead of } \frac{(x+1)}{(x-1)}$$

Formulae, however, which are placed on the parent formula's base line may result in ambiguities, if parentheses are missing. A composed base of a power, for example, almost always has to be put in parentheses.

A simple and safe solution that prevents ambiguities is to rely on the structure that is implied by infix operators. Formulae are nested only if necessary because of the two dimensional placing of operands. A *string formula* type is used to represent such a sequence of text (i.e. identifiers, numbers, mathematical operators) and formulae that are placed on a common base line. The string formula may contain arbitrary text. Therefore, within its contents no specific explicit structure is assumed. Only the binding strenghts of mathematical operators – if present – introduce a structure within the string formula. It is in the responsibility of a *parser* to derive this structure.

The mentioned exponential formula thus behaves like an infix operator which connects the base and the exponent.

$$x \uparrow {}^{a}$$

Because the exponent is placed above the base line, it needs not to be put in parentheses. The exponentiation applies to the immediately preceding operand.

$$x{*}y^{a} = x{*}y \uparrow {}^{a} = x{*}\left(y^{a}\right)$$

Other formulae are handled in a similar way. The usual precedence rules of the parser prevent ambiguities. This solution puts more responsibility to the formula translator, but imposes less restrictions to the usage of the formula editor.

Maple

Maple is a powerful symbolic algebra system which has been designed and implemented by the Symbolic Computation Group at the University of Waterloo in Canada and which is now commercially distributed.

Maple provides a large number of facilities for mathematical calculations including simplification, factorization, symbolic differentiation and integration of mathematical expressions, exact solution of linear and polynomial systems

of equations, and so forth. Maple's mathematical libraries contain functions to solve problems within the fields of linear algebra, linear optimization, number theory, group theory, statistics, financial calculus etc. Two and three dimensional graphic plots can be computed and the latter displayed from different points of view.

From the architectural point of view, the Maple symbolic algebra system is split into a so called *kernel* and a *library*. The kernel implements the basic operations of the system such as data management, input/output, language interpreter etc. The library covers Maple's computer algebra part. The library code is loaded selectively *on demand*. Because the mathematical functionality of Maple is decoupled from the pure computing aspects, the kernel can be kept small. In contrast, the amount of library code has grown tremedously.

Maple offers an interactive shell for input and manipulation of mathematical expressions in an efficient way comparable to advanced scientific calculators. Additionally, it supports a programming notation that allows the definition and implementation of algorithms. In fact, the library code is mainly formulated in this notation.

Maple supports different kinds of user interfaces depending on the facilities offered by the underlying operating system. They range from simple shells in a terminal style up to sophisticated interactive user environments with different windows. Within these, previously processed input lines can be modified and re-evaluated.

If Maple runs in a graphical environment, output is displayed by means of two dimensional graphical formula figures. On a terminal oriented platform the results are printed with several appropriately formatted text lines. In both cases the resulting formula image can neither be edited nor evaluated further.

Maple for Oberon is an implementation of the Maple system especially designed to be ported to different operating systems. It is based on an interpreted Maple kernel that is executed by a software emulator, called the *Maple Machine* [Vorkoetter89]. The Maple Machine emulates a stack oriented "computer" with its private memory and a UNIX-like [ThRi74] operating system base for memory management, file I/O etc. It is designed essentially to run *a single program* – namely the Maple kernel – and is formulated in the programming language *Oberon*.

Local Service

Maple for Oberon is running under control and within the memory space of the Oberon system. Considering this, an obvious and straight solution to implement an evaluation engine is to directly build and manipulate *data*

structures within Maple's private memory, i.e. the fraction of Oberon's memory space that is managed by Maple.

In addition to its interactive terminal shell, Maple for Oberon has been extended with a procedural interface that allows an Oberon program to create, manipulate and evaluate Maple expressions within Maple's private memory. These expressions are recursive data structures which Maple uses to represent numeric values, expressions, functions, algorithms (i.e. lists of statements), function plots etc. All internal data structures are built with nodes of the same format. The nodes consist of a header encoding their types and lengths (plus some internal details) and a certain number of fields that are usually references (pointers) to further nodes. We will refer to them as *Maple nodes*.

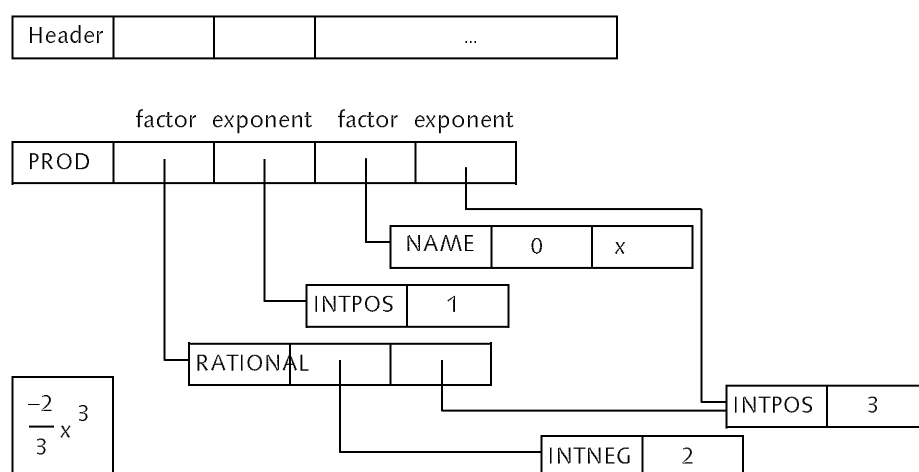


Figure 5: General structure of Maple node and full expansion of a mathematical expression.
Note, that the PROD (product) structure contains pairs of factors with their numeric exponents.

To manipulate the Maple data structures, the following procedures have been implemented:

```
PROCEDURE CreateNode (id,len: LONGINT): LONGINT;
PROCEDURE WriteNode (addr,n,value: LONGINT): LONGINT;
PROCEDURE ReadNode (addr,n: LONGINT): LONGINT;
PROCEDURE Formula (addr: LONGINT): LONGINT;
PROCEDURE CreateName (name: ARRAY OF CHAR): LONGINT;
```

CreateNode creates a node with type *id* and *len* entries. It corresponds to the standard Procedure NEW under Oberon. *WriteNode* and *ReadNode* allow for reading and writing of values from and into the slot *n* of the node referred to by *addr*. *CreateName* returns a reference to a node for storing identifiers. The name is registered in an internal name table which Maple maintains to store

and retrieve identifiers. *Formula* evaluates a given expression tree, i.e. performs basic simplification or function evaluations.

Remote Service

If Maple is running as a *remote service*, a single installation can serve a multitude of clients. However, if Maple is running remotely, interaction cannot be based on direct manipulation of data structures. A serialization scheme (protocol) for mathematical expressions is required instead.

The Open Math Proposal

In a distributed mathematical computing environment we can distinguish different classes of applications which perform, use or handle mathematics. Such applications are formula editors on one side and symbolic algebra systems or typesetting engines on the other. The integration of such facilities requires a standardized data representation and communication scheme. Usually, mathematical expressions are represented by means of an abstract syntax.

Approaches for the integration of different mathematical facilities based on an abstract syntax have been around for a long time already (cf. for example [Ar87]). A new syntax proposed by the developers of the Maple system is called *Open Math* [Vorkoetter94a].

Open Math defines a serialization method for nested mathematical expressions in LISP–style. Its specification merely defines the lexical grammar and the syntactical structure of mathematical expressions, and identifies the most important operations and functions. Because the set of allowed symbols is is left open, Open Math is an *extensible* communication scheme. Thanks to its generic structure, Open Math does not anticipate any specific symbolic algebra system or typesetter. The actual communication between Open Math clients and servers happens by means of Ascii strings.

The Open Math specification described in the *Open Math Proposal* consists of a lexical syntax definition of the Open Math language and a set of semantic rules for the interpretation of Open Math expressions. The lexical syntax as derived from the specification is listed below ("––" in the *symbol* definition below means "not containing"):

```
Expression = integer | basedInt | real | float | string | symbol |
   backRef | "(" ExpressionSequence ")".
ExpressionSequence = [label] Expression {[label] Expression}.
integer = [–] digit {digit}.
```

```
basedInt = "$" digit [digit] "," hexDigit {hexDigit}.
float = "%" {hexDigit}*16.
string = """" {char | "\" ("\" | """")} """".
backRef = "#" digit {digit}.
label = backRef ":".
symbol = "–" | (char –– ("%", "$", "(", ")", """", "#", "–", ".", digit)) {char –– ")"}.
```

The language supports several data formats: Integers, based integers (whose bases are different from 10), reals, floating point numbers (with a hexadecimal representation), character strings and symbols (operators and identifiers). It allows the labelling of subexpressions so that labels may be used to refer to subexpressions that occur several times.

Unfortunately, the Open Math proposal suffers from several ambiguities and design shortcomings. These render the implementation of an Open Math parser more difficult. For example, the first item in a sequence of expressions in the syntax above is an *expression* although the specified semantics cogently require it to be a *symbol*. The parser thus has to recognize and discard invalid Open Math expressions although they are syntactically correct.

The Open Math proposal is currently subject to a substantial redesign [Vo94b]. Therefore, the following discussion mainly refers to the integration of Maple as a local service. However, results of an experimental implementation of the Open Math proposal are mentioned in the corresponding contexts.

Transforming Formulae to the Service's Internal Representation (Forth Transformation)

The recursive (nested) structure of both representations of mathematical expressions (formulae on one hand and Maple's data structures on the other) implies that the transformation of any of them into the other is again a recursive process.

In principle, the conversion of any formula results in an appropriate Maple node whichs contains the results of the conversions of the formula's children. A fraction, for example, is translated into a product node containing two factors, one with a positive and the other with a negative exponent of absolute value 1. Because Maple performs trivial simplifications on expressions automatically, a costly optimization of the resulting structure may be omitted.

To translate string formulae, a parser for a subset of the Maple language has been provided. Expressions and terms within the string formula are represented by sum and product nodes. Although in Maple these nodes can have arbitrary lengths, a binary representation is used. This dispenses us from counting the number of expressions and terms in the formula.

Special formulae like the integral or the summation are represented in Maple as functions with corresponding names ("int" and "sum"). If bounds are omitted, Maple computes the undefined integral and sum, respectively.

Properties within a formula that cannot be deduced safely are defined explicitely. The determination of the integration variable is an example of such a property. While integration *bounds* have their fixed location within a formula (above and below the integral symbol), the integration variable is defined implicitly in the form of a factor prefixed by a "d" (e.g. dx). This factor can be located anywhere within an integration term (e.g. on top of a fraction). As there exists no general method to locate the integration variable within an integration term, a syntax has to fix its location *explicitly*. The following syntax defines the location of the integration variable within the term to be integrated. Restrictions like this are the price to be paid for the formula to be evaluatable.

integrand = term intident.
intident = "d" identifier.

Serialization of Formulae (Open Math). The general processing scheme of the formula transcription into Open Math is based on the basic structure of its expressions

"(" symbol *expr0 expr1* ... ")".

The symbol depends on the formula's type or the current arithmetic operator within a string formula. The expressions *expri* are the recursively resolved operands. Note that the prefix notation of the Open Math language requires operands to be known before subexpressions can be processed.

Transforming the Service's Internal Representation to Formulae (Back Transformation)

The construction of formulae is done by parsing the data structure resulting from an evaluation process, and simultaneously building formulae. To reconstruct formulae of a high level of abstraction, for example to return roots instead of powers with rational exponents, "patterns" must be isolated of the low level description that Maple uses to represent mathematical expressions.

The construction process is highly context dependent. A given Maple node thus is not necessarily treated the same way if it appears in two different contexts.

Trivial conversions can be found for terminal tokens like integers, names, and for rationals and all Maple functions for which there does not exist a specific formula (sin, cos, exp, ...). These are simply included in the "current" string formula, i.e. they are appended without changing the nesting level of formulae. Special care has to be taken for the correct setting of parentheses, or – even more important – their omission if parentheses are not required. This is necessary to avoid an inflation.

$(x+y)^{a+b}$ instead of $(x+y)^{(a+b)}$

2*x*y instead of 2*(x*y)

Nodes of a sum type that are returned from Maple consist of pairs of terms and numeric factors and are translated into a string formula. The sign of the factor (integer, float or rational) determines the sign of the term and therefore must be interpreted. Real factors of absolute value 1 are suppressed.

Nodes of a product type consist of pairs of factors and numeric exponents and are translated into a string formula, if there are no negative exponents, otherwise into a fraction. Factors with rational exponents are translated into roots. Otherwise, and if the absolute value of the exponent is not equal to 1, a formula of an exponential type is built.

Functions of special types are transformed into a formula of the corresponding type if such a definition exists (e.g. integral or summation). Summation or integration bounds are extracted out of the RANGE clause, if they are supplied. The integration variable is prefixed by a "d" and placed behind the integration term (cf. Forth Transformation of formulae). All other functions including transcendental or unknown functions are transformed into the form *f(arguments)*, with *f* being the name of the function, and its arguments being set in parentheses.

De–serialization of Open Math expressions. String formulae are represented as straight forward expansions of corresponding Open Math terms. The operator prefixing the sequence of expressions in the Open Math language is to be distributed between pairs of expressions within the (sequential representation of) string formula. If the "current" operator has lower precedence than the "outer" operator, the whole expression must be enclosed in parentheses.

The construction of formulae may involve pattern isolation as in the case of Maple nodes. Because of the sequential structure of the input stream (and some ambiguities of the Open Math definition) it is possible that the "role" of

an expression cannot be identified unless the full expression has been parsed. The following example illustrates the problem:

(+ a (– b c)) results in a + b – c
(+ a (– b) c) results in a – b + c

Note that the sign of b can be identified only after b has been parsed completely, and either a closing parenthesis or a new expression has been found. A similar problem occurs in the case of the unary division (inversion). A single pass conversion of such an expression is only possible, if a scratch buffer is used. After the expression has been parsed completely, and the operator identified successfully, the scratch buffer is copied into the formula under construction.

Construction of Graphic Plots

Besides the pure mathematical functions, Maple's libraries contain functions for two and three dimensional graphic plots of arbitrary expressions. Such graphic plots are computed by applying a plotting function like `plot()` or `plot3d()` to an arbitrary expression (the former computes a two-dimensional, and the latter a three-dimensional graphic plot). The result of a plotting function is a data structure which contains information about the graphical *shape* of the function instead of its expression or *value*. The graphical shape is defined by means of pseudo functions like `CURVE` or `MESH` that are applied to nested lists of spline curve sections. These either define the shape of the function or the surface of a three-dimensional object, respectively. Instead of transforming the data structure into a formula which only reflects the *kind of coding* a graphic plot, it is obviously preferable to transform the plot data structure into a *graphical object*. Hence, if a function with the name `PLOT` or `PLOT3D` is parsed, it is not represented as a formula but interpreted as a graphic plot.

In the two dimensional case a plot data structure contains information about the graphical shape of the function, about axis, ranges, labelling and so on. With this information a graphical figure is constructed that consists of horizontal and vertical lines for the axis, spline curve sections for the shape of the function and text captions for the lettering. All these graphical elements are borrowed from the integrated and extensible graphics package *Illustrate* available under Oberon System 3. This reduces the programming effort needed to implement the plot facility, because a suitable *composition* of existing parts

is sufficient. Additionally, the graphic plot can be manipulated by the graphics package, and by any other application that is able to consume such objects.

In the three dimensional case, a slightly different approach has been chosen. The graphic plot should have a three-dimensional, plastic shape to be interactively inspected from different points of view. To achieve this, a polygon renderer is used which evolved from a diploma thesis project [Os93]. The polygon viewer allows polygonal descriptions of shapes to be visualized, and geometrical viewing transformation interactively to be applied. The graphical structure used in the polygon viewer (whose details are irrelevant in our scope) allows a very fast rendering of the three dimensional shape.

Related Work

There is a lot of work relevant to interactive WYSIWYG mathematical systems in their different aspects such as mathematical editors (e.g. Theorist, Formulator, MacEqn, MathWriter to name just a few), symbolic algebra (Eureka, PowerMath, Mathematica etc.), numeric mathematics (MathCAD, Mathlab) and so on. We therefore confine the discussion on related work and applications which are most relevant to the subject of this chapter.

CaminoReal. CaminoReal is a system for direct manipulation of mathematical expressions. It has been developed at the Xerox Palo Alto Research Center [ArBe*88]. Its goal was to integrate several categories of mathematical software such as mathematical typesetting and symbolic computation into a single package with the objective of an *interactive mathematical notebook* (as states the subtitle of the paper).

The CaminoReal package consists of a formula editor that allows interactive (but not in-place) construction of formulae by successive combination of formula templates. The evaluation part of the system consists of a collection of symbolic algebra packages that are accessible over the network. With these computational facilities, the user can interactively explore mathematical expressions from a document that contains formulae.

An interesting component of the CaminoReal system functions like a mathematical spreadsheet. Based on this facility, so called *computed documents* can be constructed. Such documents may contain sequences of dependent expressions whose values are automatically updated if an expression within such a sequence is modified. Mathematical expressions within technical or mathematical papers can thus be computed "on-the-fly", e.g. when the

document is printed. This facility helps for example to avoid typing errors within mathematical documents.

In contrary to our project, interactive aspects have not been fully developed in CaminoReal. Formulae, for example, are created with the help of a menu-driven *external* application, i.e. they cannot be edited in-place. While the menu-driven construction guarantees structural correctness of formulae at any time, the imposed restrictions during the formula construction might be considered obstructive. CaminoReal is also a tool for interactive experimentation with mathematical expressions. However, in our project we have put slightly more emphasis on interactivity. For instance, formulae can be manipulated directly in the text. Furthermore, interactive exploration of mathematical expressions extends to the visualization of their graphical shape (plotting facility).

Maple User Interface. The Maple system also tends to extend its interactive facilities. The possibility to return nicely rendered formulae (in bitmap form, though), and the inclusion of versatile 2D/3D plotting packages that allow visualizing complex mathematical properties are obvious steps in this direction. In particular, in Maple's online documentation the contained expressions and statements can be evaluated directly in place. Both steps, however, have only be done halfway so far: The formula bitmaps, on the one hand, can neither be manipulated nor evaluated any further. The graphic plots, on the other hand, are internal Maple data structures whose graphical shape has to be recomputed each time one of the viewing attributes (color scheme, viewing angle etc.) is modified, rather than autonomous graphical objects. It is easily foreseeable, however, that future releases of Maple will extend their interactive facilities comparable to those presented in this chapter.

Summary and Conclusion

A formula editor and a symbolic algebra system have been integrated in order to form a *Dynamic Mathbook*. This type of dynamic online documents extends the notion of interaction to direct manipulation and evaluation of its contents, and it opens up the possibility of *integrated computing* within a textual document. In contrast to pure multimedia applications, the result of such a computation can be *new* or derived information that was not contained in the original data.

By providing an interactive choice of representations of the result – either as a mathematical expression or a graphic plot – the presented system extends the functionality of comparable existing systems.

The presented evaluation engine provides an interface to the underlying symbolic algebra system. It performs all necessary transformations and shields the algebra system's internals from the formula editor.

Typographical Quality of the Result

The typographic formatting rules that are applied by the interactive formula formatter have not been discussed because they are of minor relevance in our context. One problem, however, needs a further discussion as it normally does not appear in interactive formula editors. Formulae that are created by human users not only respect the correct mathematical meaning but also show a well–balanced image. In contrast, formulae which are constructed by a machine (including those created by our evaluation engine) tend to be large and intricate. They may easily exceed the width of a text line.

There are mainly two possible solutions to the problem of rendering large formulae. The first is to break such formulae into appropriate pieces each of which fitting on one text line. The second is to substitute suitable subexpressions by place holders, and to add the sequence of subexpression declarations to the output.

The first solution follows from the associative property of sum and product. To break a sum or a product type of expression into pieces is trivial, because both products and sums can obviously be split up between any two operands. In principle, this holds for a fraction type expressions as well. In this case both the numerator and denominator have to be split appropriately into a product of two (or more) shorter fractions.

The second solution is compatible with structural properties of Maple's handling of internal data structures because Maple stores multiple occurrences of the same simplified expression as multiple references to a single and unique representative. As a consequence, common subexpressions can always be identified by simply comparing their references. Hence, subexpressions referred from multiple locations within an expression are suitable candidates to be represented by a place holder. Replacement of multiple occurrences of an expression makes the resulting formula simpler and smaller. This specific property, by the way, has been exported in the Open Math Proposal through the possibility of labelling an expression and afterwards referring to it again.

Unfortunately, experiments revealed that the effect of the corresponding size reduction is not quite as significant as expected. Further research needs to be done in this field.

Case Study 2: Electronic Newspaper

In this chapter we discuss the integration of a central *Teletext* service into the Oberon environment. Teletext is a non-interactive textual information service that is broadcast together with the television video signal. Teletext information comprises various topics such as political news, sports news, television guides, advertisement and so on. By integrating the service into Oberon, current online information can be accessed by everyone over the local area network. A *Teletext gadget* displays the information with the service's typical look and functionality. Furthermore, the online document *TeleNews* is presented. It is a different and novel way of presenting and organizing (text based) Teletext information as an online document. With *TeleNews*, Teletext information can be accessed and represented in hypertext form. It is based on the model of an *electronic newspaper*.

Introduction

Printed information on topics that are subject to continuous change almost always has a serious shortcoming: It is not up-to-date anymore when a subscriber eventually gets his copy. This kind of printed information is inherently out-dated because of the delay between the acquisition of the information and its eventual delivery during which the information may change.

In order to keep the information as up-to-date as possible, this delay has to be minimized, in the extreme case by acquiring and collecting the information immediately before delivery. Such immediate delivery cogently requires that the information is requested from the information supplier only at the time of its consumption, e.g. via an online connection. In practice, parts of the information that don't become obsolete too quickly can probably be delivered as some kind of static information framework. The volatile parts of the information, on the contrary, are gathered and compiled only at the time the user consumes the information.

In the following, an online document called *TeleNews* is presented which simulates an electronic newspaper. It is based on information that is collected by a *Teletext* server which has recently been installed in our institute, and which

takes the role of an online news data base. TeleNews eliminates the cited drawbacks of printed information carriers, because information acquisition and presentation happen at the same time. Information is acquired by means of smart links.

Teletext

Teletext is a page oriented, non-interactive information service that is broadcast by television stations together with the video signal. Teletext pages contain alphanumeric text and raw graphics assembled with block graphic characters. They are displayed on the television screen. Teletext has originally been developed by the research department of the British Broadcasting Corporation (BBC) in order to supply subtitles and captions for television programs. Since that time, it has become an attractive information medium of its own. It is now supported by almost all European television companies. Information presented via the Teletext service covers political, sportive and economic news, weather forecasts, television programs, entertainment, advertisement and so forth.

Technical Background

Teletext is a digital information system whose data is transmitted during the vertical blanking period between the transmission of subsequent video half-images. To receive and decode Teletext, the television set must be equipped with special decoding circuits. Usually, these circuits are integrated in a specific Teletext processor which also controls the visual representation of the data on the screen.

A Teletext page consists of 24 lines containing 40 character cells each. A character cell is occupied by either an alphanumeric letter, a block graphic symbol, or a control code. These control codes are used to specify display attributes and visual effects including different colors, double height printing, flashing and so on.

Teletext pages are identified by three-digit page numbers that range from 100 to 899. All pages are repeatedly broadcast one after the other in a non-stop broadcast loop. Although the transmission of pages may happen in arbitrary order, they are normally enumerated from 100 up to the highest available page number. A sequence of several related pages can be assigned to the same page number. These pages are called *rolling pages*. In the transmission slot of the corresponding page number always the next page of the rolling page sequence is transmitted.

A specific page is displayed on the television screen by specifying its page number on the television's remote controller. As soon as the corresponding page (or any of its rolling pages) is passing by, it is extracted from the video signal and displayed on the screen. If the page with the corresponding number passes the next time, the display is not updated unless the page's contents have changed, or it is the next sub page out of a rolling page sequence.

Teletext Presentation

It is not necessary to care about a fancy presentation of Teletext on a computer display, if one is interested in consuming the information only. Instead, a simple text terminal that displays the alphanumeric contents of a Teletext page is obviously sufficient. However, the integration of an external online medium cannot be said to be complete if its typical look is not adopted reasonably well. Colors and graphics, for example, are not just gimmicks on an otherwise dull information service. Color is used to structure the text on a cramped text area. With Teletext's block graphics facility, simple graphical illustrations like company logos on advertising pages can be represented.

Teletext Panel

In Oberon System 3, the Teletext service is displayed in a so called *functional unit*. These are self-contained operating panels that collect a visual display and all necessary user interface elements in a panel document [Gu94a].

Figure 1: Teletext gadget in an operating panel

The *Teletext panel* contains a *Teletext gadget*, and is equipped with buttons and text fields that serve to control the Teletext display.

The Teletext gadget produces an image of the service comparable to that on a television screen. It supports all visual effects like different colors, double height display, and flashing. In contrast to the service on a television receiver, however, the Teletext gadget is much more convenient to use. In particular, the two major deficiencies of the original service's user interface have been circumvented: Page selection, and the waiting times for rolling pages. First, the user can point directly at three-digit numbers in the Teletext gadget to obtain the corresponding page. Page numbers thus have to be typed only rarely.

Second, all pages of a rolling page sequence are always fetched from the service together at once. Hence, the reader can browse forward and backward, and read the pages in any order and pace. Furthermore, a short history list allows returning to a previously visited page by pressing a button only. On a television receiver, the reader has to retype the (remembered!) page number.

TeleNews

Having a Teletext terminal on one's computer display ready to display Teletext information with its familiar look is doubtlessly very impressive. However, experience revealed that the presentation of the Teletext terminal on the screen does not really satisfy the frequent reader. There are two reasons for that: First, the very small amount of information presented at once (less than 1000 characters), and second, the small terminal display. Although the possibility to recognize the service on the computer display should not be disregarded, imitating a Teletext terminal and hence all its limitations and drawbacks is unsatisfactory. Taking into account the capabilities of modern high resolution computer displays, we may conclude that the mere enhancement of the Teletext *presentation* may impose a progress over the old fashioned terminal look.

To avoid the mentioned drawbacks of the Teletext service we have invented a different and novel way of presenting and organizing (text based) Teletext information. Hence we have achieved, that the origin of the information does not need to be noticeable any more by the end-user.

TeleNews is an online document that maps the Teletext data base to the form of an "electronic newspaper". With TeleNews, news articles from the Teletext service can be interactively accessed and obtained in hypertext form.

TeleNews consists of an operating panel which contains two tracks: The *headline track* and the *article track*. Above the headline track there is a menu of buttons. Each refers to a different news rubric such as national, foreign or

economic news. By pressing one of the buttons the corresponding news section is acquired from the Teletext server, and its contents displayed in the headline track. The headline track can be viewed as a menu that lists the currently available news articles in a compact form.

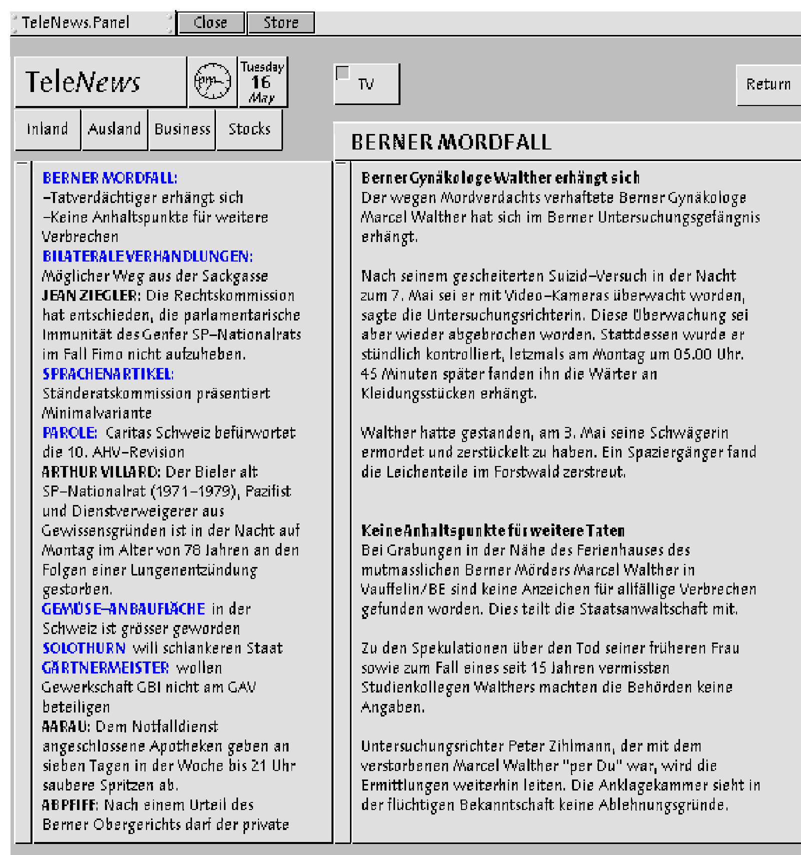


Figure 2: TeleNews panel

Headlines are anchors of smart hypertext links that reference the associated news articles. Pointing at the highlighted part of the headline initiates the request for the corresponding news articles which in turn are displayed in the article track. If the article consists of several pages or chapters, they are all delivered at once simulating thus a grouping of articles or paragraphs familiar from ordinary newspapers.

Electronic TV Guide

The electronic newspaper provides selective access to the dynamic information base: Only those parts of the information that the reader is interested in is fetched and presented. The *Electronic TV Guide* extends this kind of selective access by accepting requests for selective *projections* of the data base; that is, a collection of data items matching certain parameters can be requested.

This application is inspired by the very probable availability of more than 100 satellite television programs in the near future. A printed program guide would have the size of a telephone book with the unpleasant consequence that the desired information ("is there a feature film tonight ?") is almost undiscoverable. An electronic version of a TV guide can include searching and browsing facilities and different ways of organizing the information. Depending on the query, TV schedules selected by broadcast time, subject of the program or television channel can be provided. Additional information about the programs including short trailers can be supplied as well. With online connections to an information provider also short-term changes can be considered and even current information be supplied.

The electronic TV guide is an online document that implements the basic functionality of such a package. It processes the program overview pages of the Swiss Teletext service. As a special feature, the Swiss Teletext service additionally includes television program schedules of the most important television companies of the German speaking part of Europe. Although these television program pages originate from different Teletext suppliers, they have an identical structure.

Information on program schedule pages consists of the starting time and length, the program's title, and some technical attributes such as the presence of subtitles and the sound quality (stereo or multilingual). Additionally, program items may contain references to further Teletext pages with detailed information about a television feature.

Based on this collected information, the electronic TV guide computes excerpts (or *projections*) of the TV schedules that match certain parameters. The electronic TV guide can thus be seen as an example of on-the-fly computed hypertext. Currently, four different projections of the TV guide's data base can be requested.

TV channel. The current TV schedule for the day can be listed for each of the TV channels whose information is available.

Broadcast time. All program items of the current day of all available TV channels falling in a range of time are listed (i.e. whose broadcast times intersect with the range).

Program type. Feature films and news programs of the current day of all available TV channels can be listed.

Program attribute. It is possible to list all program items of the current day which are broadcast either with subtitles or with stereo sound or in multilingual mode. For people who are hard of hearing, for example, a TV

schedule might be helpful that lists program items which are supplied with subtitles.

The electronic TV guide implements the television program rubric of an electronic newspaper. Consequently, its user interface is integrated into the TeleNews panel.

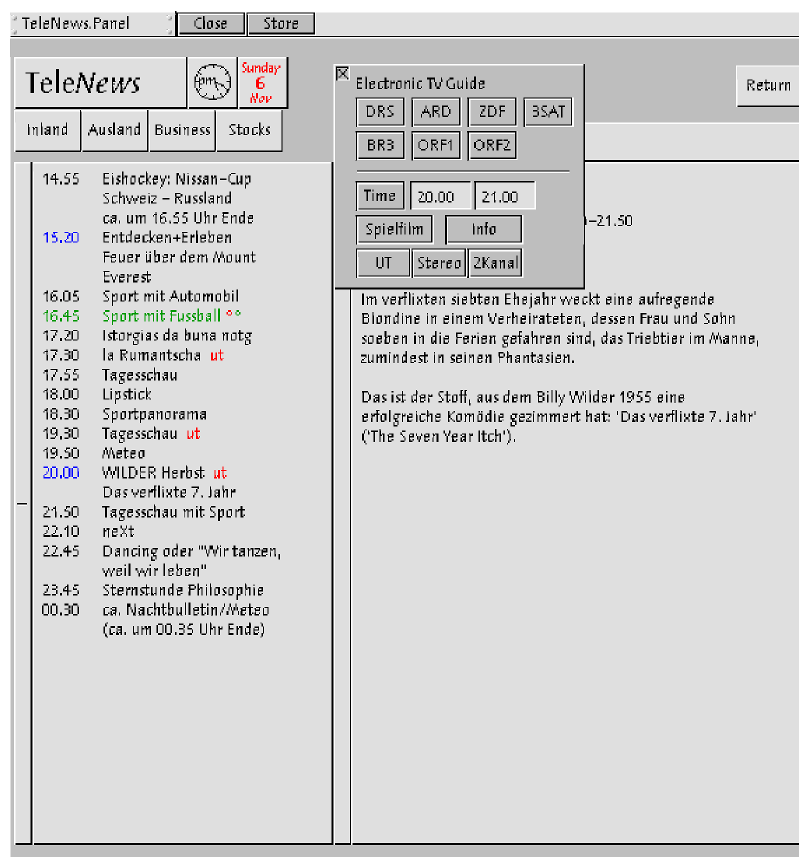


Figure 3: Electronic TV Guide

A popup menu within the panel collects all the functional elements of the electronic TV guide. For each type of request a different user interface element is provided in the panel. Depending on the type of the request, projections of the information can be constructed and listed in the headline track of the TeleNews panel. Within the list, the program item currently on air is highlighted. If additional information is associated to a program item, a smart hypertext link is added to the corresponding entry. If the link is activated, the information is displayed in the panel's article track.

Implementation Aspects

To provide access to the Teletext service within a computing environment, a Teletext processor has to be connected to a computer. Such devices designed for their integration in television receivers contain all necessary circuits for the separation, acquisition and decoding of the teletext signal. We will refrain from stressing the reader's patience with the details of the corresponding hardware implementation. Hence, we will leave the subject by mentioning that the hardware implementation is based on a simple Teletext board whose blueprints can be found in [Ba92]. The teletext processor that we use is equipped with four independent receiver channels [Ph90]. Therefore, it can simultaneously scan for 4 different pages. The board is connected to the computer via the modem lines of a serial RS–232 connector.

Centralized Teletext Server

A centralized server for Teletext is not only an issue of economy – only a single Teletext decoding device is needed for a group of clients. Several of its properties make it not only desirable but necessary to centralize it on a server and access it remotely. The most influencing property is the cyclic transmission scheme of Teletext.

Due to this cyclic transmission scheme, scanning for a given page on a heavily loaded service may take several seconds. The Swiss Teletext service, for example, consists of about 1200 pages including sub-pages. About 250 of those are single pages which are broadcast once every cycle. A full enumeration cycle takes about 14 seconds. Rolling pages are often rolled forward only once in two cycles. In addition to that, a user accessing a rolling page will very probably step somewhere in the middle of the subpage sequence having thus to wait for the first page of the sequence to reappear the next time.

Connecting a user directly with a Teletext decoder is thus inappropriate. While the waiting time for a single page is just barely acceptable, the waiting time for a sequence of rolling pages is prohibitive. This is especially true because the reader usually wishes to receive all parts of a rolling page anyway, which multiplies the waiting time by the number of subpages.

For these reasons, the Teletext service has been centralized on a dedicated computer and made accessible over a local area network. All transmitted pages from the Teletext service are downloaded and cached on a local disk. Hence, page requests can be immediately served. Therefore, the Teletext server's first task consists of supervising the Teletext broadcast loop to update the data

base on the disk, if pages or page constellations change. Its second task is twofold: To handle and serve requests, and to deliver (sequences of) pages.

Downloading Strategies

The teletext service is a unidirectional (one-way) information service that applies the (passive) operation model familiar from ordinary radio and television broadcast; that is, the teletext decoder is "tuned in" to a certain page number just as the television receiver is to a certain TV channel. If the page passes by, it is extracted. A certain teletext page is thus downloaded by "tuning in" the teletext decoder to the corresponding page's number and waiting for the page to appear the next time.

The time needed to download a collection of pages (or even the whole teletext data base) is basically the total of the waiting times for each of the individual pages. An efficient downloading strategy thus aims at the limitation of the sum of all waiting times. This can be achieved by extracting as many pages as possible pages in one enumeration cycle, and by avoiding subpage misses in rolling page sequences. Note that missing a subpage within a rolling page sequence delays the complete downloading time of the corresponding page by the duration of a total rolling page transmission cycle (i.e. until the missed subpage arrives again). If we favor one of the goals (multiple pages vs. avoiding subpage misses) to the debit of the other, respectively, we can distinguish two strategies:

Defensive strategy. In the defensive strategy, completeness has priority over efficiency; that is, a subpage miss is avoided at all circumstances. In order to guarantee not to miss a subpage of a rolling page sequence, the receiver channel (of which four are available in the Teletext device we use, as mentioned earlier) is immediately restarted with the *same page number* (and thus the next subpage of the rolling page sequence). In other words, the receiver channel is assigned to the corresponding rolling page until all subpages have been extracted. Hence, a subpage is never missed. However, by assigning a receiver channel exclusively to a rolling page number, the page arrival rate for one receiver channel is one page in two enumeration cycles or *one page every about 30 seconds*!

Aggressive strategy. With the aggressive strategy, efficiency has highest priority. The downloading algorithm is designed to download as many pages as possible within a certain period of time. It thus catches *every page* that passes by. Its efficiency arises from the fact, that no receiver channel is monopolized

for a single page number but immediately reused for the next available page being transmitted. If a page does not arrive within a certain time, it is assumed to be a rolling page that does not appear in this round. In that case, a new page number is searched for. The page arrival rate for a receiver channel is about 10 pages per cycle.

However, if the receiver channel is released after the arrival of a subpage, it is not necessarily at hand again before the next subpage of this sequence arrives. Subpage misses thus become the normal case. If a page miss happens in a rolling page with only a few subpages, there is a good chance that the missing page will be caught during one of the next cycles. However, rolling pages that consist of 60 pages (leading to a full transmission time of 30 minutes and more) are very probable never to be completely downloaded *at all*!

Implemented approach – weakened completeness. Whereas the defensive strategy leads to prohibitive downloading times, the efficient strategy (efficiency measured in terms of page arrival rate) leads to an unacceptable state with pages to be occasionally outdated for days. Some measures have to be taken, either of the two drawbacks to be avoided.

The downloading strategy that has been implemented in the course of a term project [Do93] is a relaxed defensive strategy. The longest possible downloading time is needed only when the algorithm starts from scratch, i.e. with an empty data base. As rolling pages normally change more rarely than single pages, and as normally *all* subpages change, if *any* of it does, it suffices to inspect *one subpage* out of a sequence, to find out, if the whole rolling page has to be downloaded. If the page has not changed, no further subpage with its page number is inspected until every other page has had its turn. This strategy drastically reduces the downloading time. By preserving the defensive strategy only for those rolling pages which have to be downloaded *completely* (e.g. due to a change) and using the efficient (offensive) scanning algorithm for fast inspection, the downloading time may be reduced further. The mean update rate varies between 12 and 20 minutes (depending on the update load), which is acceptable.

TeleNews

A newspaper is structured in different rubrics by (geographical) region or subject. News articles are usually organized by geography into global, regional and local news, and/or by subject into political news, business news, sports news etc. Within this raw structure, news articles about the same subject are grouped together, and are otherwise arranged arbitrarily on the page.

A typical news article consists of a *headline*, and a short *summary*, followed by the detailed *news text* itself. The main purpose of a headline is to catch the reader's interest in reading the whole news article. These structural properties imply the usual way of reading newspapers that consists of first browsing through the headlines and selecting articles depending on one's interests (very few people read a newspaper sequentially from the first page to the last).

This typical way of reading newspapers hierarchically suggests a possible structure of an electronic version of a newspaper. It is based on a first-level hypertext document that allows "zooming in" from the headlines to the text. The reader is first presented headlines and summaries. The actual news texts can then be obtained by pointing at the corresponding headline.

The hypertext structure introduces interactivity of a finer granularity into the electronic version of a newspaper compared to ordinary (printed) ones. In ordinary newspapers, interaction essentially restricts to the action of turning over the pages. By adding hypertext facilities, interaction is refined to navigating through the newspaper. If headlines are presented in a compact list, they can be overviewed very easily. Selecting articles is thus very efficient.

Using *smart links* as a basis for the hypertext structure adds two more advantages. If the actual news articles are called from the information server only at link resolution time, information that is not required needs not to be transported. Furthermore, with such *delayed compilation* of the actual news texts, latest updates of their contents can still be considered. The delay between the information's acquisition and delivery can thus be minimized. This keeps the electronic newspaper up-to-date at any moment.

Automatic Hypertext Construction

TeleNews is based on the news section of the Swiss Teletext service. The news section consist of overview pages and the text pages which contain the actual news texts in full length, respectively. The construction of the hypertext structure, i.e. the links from the headlines to the corresponding articles, is done by analyzing overview pages. Analysis is based on some heuristically determined properties of the involved pages.

An overview page in the Teletext contains a list of briefly paraphrased news topics, each ending with the number of the page that contains the news text. The relation between different pages is derivable only from these page numbers, i.e. there is no auxiliary structure explicitly linking pages to others. For a human reader it is obvious that a three-digit number associated to a news topic is a Teletext page number. The automatic determination of the reference structure is more intricate. As the Teletext service mixes content and structure,

it is not always obvious if a number indicates a Teletext page or an arbitrary numeric value. To distinguish between numeric *content* and numeric *structure indication*, the numbers' contexts have to be considered.

In the mentioned Teletext gadget with built-in pointing facility, page numbers are not distinguished from arbitrary three-digit values. Therefore, users often intentionally click at arbitrary three-digit values (for example a price within a list of stock rates) to "random walk" through the service.

Whereas this property is a rather harmless artefact of the Teletext gadget, it leads to wrong references of an automatically generated hypertext document. If the hypertext reference structure is based on Teletext data, heuristics are required to correctly identify page numbers.

Page Analysis Algorithm

On an overview page of the news section of Teletext, each news article is represented by a particular entry that we have called *overview topic*. Sometimes, the news about a certain subject splits into several articles each represented by an own topic.

Using conventional terminology of newspapers, one may call an overview topic a *headline*. Its associated text page contains the *article*.

When the hypertext structure of the news service is constructed, the headlines must be linked to their corresponding articles. To construct this reference structure, overview pages are parsed to construct headlines from topics, and to separate groups of headlines of different (news) subjects.

The extraction of the properties of an *overview page*, i.e. the identification of its elements (topics), is the most important and at the same time the most critical task of the heuristic parser. It uses the following heuristics based on the structure of an overview page, to identify the different types of items (headlines etc.).

Headline: A headline typically starts with a yellow title (usually in capital letters), continues with a sequence of characters (possibly extending over more than a single line) and ends, if the foreground color changes. If such a sequence of characters terminates with a three-digit number in the range of 100 to 899, it is assumed to be a *headline*.

Example:

```
SELTENER FUND: Skelett eines Ty-
rannosaurus Rex in Kanada entdeckt..130
```

Subsequent items of the above type are assumed to belong to the same (i.e current) headline as long as they do not start with a yellow (capital) title and are not printed with one of the colors that identify *colored items* as specified below. The whole headline refers to the whole set of pages referred to by its items.

Example:

```
ZUGUNGLÜCK IN LAUSANNE:
-Drei Gründe führten zum Unfall /
 Probleme bei der Bergung des Giftes105
-Zugverkehr stoppt vor Lausanne /
 Erste Massnahmen der SBB...........106
-Chemische Industrie
 gegen Transportstop................107
```

The identification of groups of related items is one of the conceptual advantages of the TeleNews application compared to the terminal display of the Teletext gadget. It relieves the user from getting the corresponding pages one by one. Furthermore, it organizes the related news articles in a way similar to that of an ordinary newspaper.

Colored Items: Overview items printed in either green, magenta or yellow that extend over a single line and end with a three-digit number in the range of 100 to 899 are assumed to be colored headlines. These items are sometimes used to indicate background information about a subject or to identify pages that contain news flashes etc.

```
Kurzmeldungen...................115/116
```

Page references may cover a range of pages, which is indicated either by a slash or a dash character. The headline then refers to the whole range of pages.

If a headline has been successfully identified to be of one of the above types, a smart link is assigned to its title. Headlines that don't end in a three-digit number are assumed to be *news flashes*. These are short articles that consist of a few lines of text and completely cover a subject. Therefore, they don't refer to a further news article.

Text pages, i.e. pages holding the news article, are terminal nodes of the hypertext structure and thus don't contain further links. Their processing essentially confines to reformatting the fixed 40 characters-per-line format. The hypertext viewer is expected to perform implicit line breakings.

Words that have been separated by means of hyphens to fit on the 40 character teletext lines are connected again, if the next line starts with a small letter. This admittedly very simple heuristic almost always results in correctly connected words. Only one (!) frequent special case needs to be handled specially. The same procedure is also applied to headlines.

On a teletext page, different paragraphs are separated from each other by a different text color or by a blank line. If such a paragraph separation is detected, an explicit line break is inserted into the text. Using different fonts for titles and flow text supplies the text with yet more structure.

Smart Links

Headline pages contain smart links that refer to the associated news articles and that allow to access them in a hypertext-like fashion. The specific smart link descriptor of the TeleNews application is listed below.

```
Link = POINTER TO LinkDesc;
LinkDesc = RECORD(SmartLinks.LinkDesc)
  pn0, pn1: INTEGER    (*range of teletext page numbers*)
END;
```

The TeleNews link descriptor contains the range of Teletext page numbers that are covered by the smart link.

```
PROCEDURE Request*(S: Services.Service; obj: Objects.Object; par: ARRAY OF CHAR):
                                                          Objects.Object;
  VAR link: Link; text: Texts.Text; pno: INTEGER; c: PageClass;
BEGIN link := S(Link);
  pno := link.pn0;
  WHILE pno <= link.pn1 DO
    c := Class(pno);    (*identify type of teletext page*)
    c.Process(c, W, pno);    (*get page from teletext server and process depending on type*)
    INC(pno)
  END;
  NEW(text); Texts.Open(text, ""); Texts.Append(text, W.buf);
  RETURN text
END Request;
```

The link's *Request* method listed above iterates through all teletext pages that are covered by the smart link. The actual processing of a page depends on its type (called *page class*) which is identified via the page number. Page classes and their attributes are listed in a configuration text.

```
PageClass = POINTER TO PageClassDesc;
PageClassDesc = RECORD
   ...
   Process: PROCEDURE (C: PageClass; VAR W: Texts.Writer; pno: INTEGER)
END;
```

Besides the fields that hold technical attributes of the corresponding page class, the descriptor contains a *Process* method which is invoked for each page. Processing consists of two steps: First, the teletext page is requested from the server and, second, the class specific page analysis discussed above is applied.

Further Applications: The Electronic Investment Bulletin

Information bulletins on financial investment are other examples of printed information carriers that suffer from the drawbacks mentioned in the introduction. Such bulletins are sometimes offered by banks and investment companies on subscription. The *electronic investment bulletin* is an application aiming at this topic. It simulates an electronic version of an edition of the investment bulletin available at subscription from the *Zürcher Kantonalbank* (state bank of Zurich).

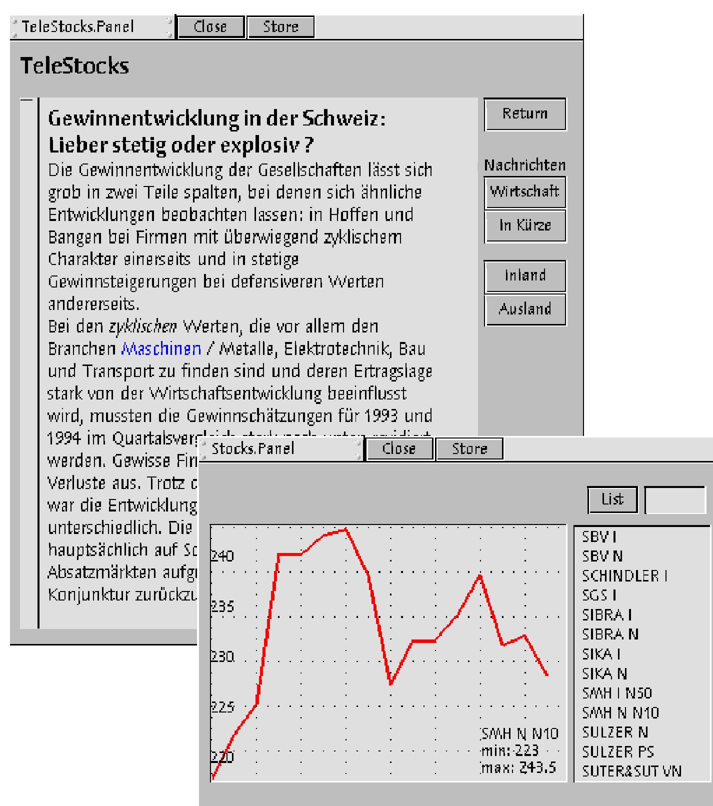


Figure 4: Electronic Investment Bulletin

The electronic investment bulletin is a hypertext application which mixes static parts such as the global market analysis (which is usually not as volatile as stock market data) with dynamic parts such as economic news and stock exchange rates. The latter is downloaded from an online provider (simulated by the Teletext server). The electronic investment bulletin is an ordinary (static) hypertext document that contains smart links to access dynamic online data. The interactive functionality of the electronic investment bulletin can even be extended by integrating a *stock data base service* (a corresponding online service has been prepared as a diploma thesis [Su95]). By storing the stock rates successively over a certain period of time in a data base, also the *course* of a stock can be visualized (in addition to the current *state*).

Related Work

Electronic Newspaper. A "real" electronic newspaper is the goal of a project called *Newsbox* of the Evening Standard in London [Di93]. It will be an electronically available edition of a newspaper which continues to appear also in printed form. Besides text, it is intended to include images and graphics transmitted as *Postscript* files. The idea is both to preserve the traditional newspaper "look and feel" as well as to extend the functionality by menu driven searching and browsing including hypertext facilities. The newspaper is downloaded as a whole on the user's private interactive newspaper terminal (the actual *Newsbox*). Besides still images, future versions are intended to contain short video sequences.

The Newsbox project aims at an electronic newspaper with "electronic" subsuming paper-less, dynamic and interactive. This is obviously an advantage over classical printed newspapers. The downloading of the newspaper immediately before reading, for example, allows to include aspects of actuality, because the electronic newspaper can continuously be edited and updated throughout the day. This is in strict contrary to printed newspapers where sometimes around midnight the editorial process is stopped to print and deliver the newspapers.

If the electronic newspaper is not intended to be "just" a daily renewed multimedia data set, it is necessary additionally to provide online connections for *selective* access to newspaper items. Such a personalized information acquisition may, for example, allow the interested reader to access additional information that is not contained in the ordinary newspaper release.

Visual Appearance of an Electronic Newspaper. Aspects of the overall presentation of TeleNews such as a sophisticated placing of articles on a page have not been investigated in detail. Such a layout of articles would probably be more desirable to achieve a typical newspaper look. In [HüHa93], for example, such a tool for an automated placing of articles on a page in a newspaper-like fashion is presented. Its placing algorithm is based on structure and properties of the documents involved. These properties are derived from their formatting and translated into an abstract document description.

The Teletext service does not include enough meta-information such as the relevance or the mere kind of the news information (feature, leading article, etc.) to decide how to present an article on a page. The only features of an article are the ones derivable from the Teletext overview page. These restrict to information such as the number of Teletext pages the article consists of or the color of the headline item referring to it (color sometimes indicates background information). However, these features are not occurring frequently and consistently enough to justify their using as heuristics for a page layout formatter. The lack of features would almost always result in either exactly the same page layout (for example n by m articles on a page) or – even worse – yield a misunderstandable grouping and highlighting of articles introduced by a wrong assignment of relevance and importance.

Summary and Conclusion

We have discussed the integration of a *Teletext* information service into the Oberon environment. A centralized Teletext server collects and stores all available pages on the local disk to provide efficient access. The Teletext client consists of a special gadget which represents the data by mimicing a TV screen, however, with extended usability. Additionally, an experimental online document modelling an electronic newspaper has been realized which presents Teletext information in hypertext form. The hypertext structure is constructed on-the-fly based on heuristically derived properties of Teletext overview pages.

The integration of Teletext into the Oberon environment provides an attractive source of always up-to-date information. By means of the centralized server, some of the service's inherent deficiencies such as the annoying waiting times could be mitigated at the cost of a slightly reduced actuality (the time for a complete update of the data base).

The first approach for the visualization of the service consists of a Teletext gadget. Its implementation effort essentially confines to aspects of rendering. All other user interface aspects can be solved with *interactive composition* of predefined items (buttons, text fields and the more). The Gadgets user interface toolkit thus proves to be an adequate basis for the implementation of such service displays.

The TeleNews application provides full integration of Teletext information into Oberon because it is based on Oberon's fundamental data structure *Text*. TeleNews information thus becomes reusable by arbitrary other applications in Oberon: All commands which operate on text can also process TeleNews information. By using smart links for its hypertext structure, all activities related to acquisition, compilation and delivery of the actual news text can be deferred until the customer reads the newspaper. Hence, latest updates can be considered as well and thus the newspaper content be kept current at any time.

Finally, TeleNews presents Teletext information in a more expressive manner compared to the Teletext terminal. Thanks to the hypertext structure, related articles can be delivered together, relieving thus the reader from fetching associated pages one by one. Using different type fonts for titles and text helps to add even more structure to the text. All these properties thus enhance the readability of the Teletext information. Only the absence of the conventional newspaper look based on a multi-column arrangement of articles on a page might perhaps be considered a drawback.

Although TeleNews is the most innovative step in the whole teletext integration process, it has also revealed the limits of the integration. Teletext proved more than once to be indeed a medium rather for humans than for computers (of course, this does not hold for computer data that is broadcast via Teletext, e.g. stock rates). Even non-editorial tasks such as the assignment of page numbers are still done by hand. Typing errors in page numbers (e.g. the letter l instead of the digit 1) and inconsequent grouping of information impose several serious problems for automated processing of teletext. Because Teletext is a non-formally defined data base, automated processing can only rely on heuristics.

A professional realization of an electronic newspaper that is still based on teletext requires more reliability. To avoid the weakness of heuristics, the structure of teletext items such as tables, titles, overviews or flow text must be identifyable explicitly. This can be done with interspersed (invisible) markup codes, or by adding special teletext pages which carry such structural information.

Sophisticated online information services such as an electronic newspaper with integrated images and video sequences will certainly consume a

significant amount of network resources (i.e. bandwidth). As long as broad access to high speed networks is not available, a solution has to be found that avoids the need to transport large amounts of data over the network. The *electronic investment bulletin* shows a possible approach. By separating the information into static and dynamic parts, a reasonable compromise can be found in order to simulate online media. Locally stored static information on mass storage is combined with small amounts of volatile online data accessed through dialup connections. Initially, the static information base can be transported by yellow mail e.g. on CD-ROM. Later it is selectively updated via online connections. The information is thus organized as a distributed pool of data that is principally stored at the customers' sites and synchronized with the central service via the network. The new graphical information service of the German telecommunication provider *Telekom* is based exactly on that operation model [DT94].

Case Study 3: Network Information Browser

In the last case study of our series we discuss the integration of *Internet services* into the Oberon system. We present the *Smart Web* browser which provides access to the World-Wide Web service. The browser constitutes both an interesting extension of the set of services accessible in Oberon as well as an illustration of distributed online documents in a global extent.

Introduction

Currently, the *Internet* offers most probably the widest range of publicly accessible sources of online information to the network community [Krol92]. By means of application protocols such as the file transfer protocol *ftp* [PoRe85] or the terminal protocol *telnet* [PoRe83], the corresponding information services can be connected and the information be read and/or downloaded. The mere number of information servers and the amount of data that is available renders an efficient use of the offered network facilities more and more difficult. Therefore, to help accessing and structuring or merely finding the information available over the Internet, several kinds of so called *information browsers* have been developed. Archive servers such as *Archie* [Krol92] assist in finding both the desired data file and the nearest possible host server holding a copy of it. A few years ago, information services have been developed which help structuring and accessing textual rather than binary information. Thanks to their power and simplicity, some of these text based information services have become very popular. Currently among the most popular is the *World-Wide Web* [HTTP0].

Network information services will rapidly become more and more important in the near future. In order to explore the potential of these network tools it is obviously desirable to access such services from within Oberon. With the successful implementation of the Internet protocol suite TCP/IP [Co95] the technical basis has been laid which makes such an access possible.

From our point of view, the integration of a network information system is interesting for several reasons: First, the suitability of the generic hypertext

facility based on smart links can be explored for a "real" application. Using an existing network service, of course, is both the most challenging and the most convincing test. Second, by successfully integrating access to network information services we lay the foundation for a new class of applications for the Oberon system, namely as an advanced client platform for the mediation and integration of online information.

As the concepts of the World-Wide Web as a globally distributed hypertext document and its accessing scheme match very closely the abstract model of online documents presented earlier, we decided to implement access to this service.

World-Wide Web

The *World-Wide Web* (*WWW* or *Web* for short) is an information service based on the model of a globally distributed hypertext document. Its development started at the European Particle Physics Laboratory (*CERN*, *Centre Européenne de la Recherche Nucléaire*) in Geneva, Switzerland. The original goal of the WWW project was to supply high energy physicists with a unified information utility for consistent access to a multitude of textual information services. Since the beginning of 1994 the Web has become very popular, and the number of Web users and information providers is currently growing exponentially.

WWW hypertext documents contain *reference links* which allow switching to logically related items (further hypertext documents, but also images, video sequences and sound patterns). These items may be physically residing at very different locations in the world. If a link is activated, the corresponding referenced document is requested and downloaded via the network. The document is subsequently displayed by a WWW client application (called WWW *browser*).

Smart Web: A World-Wide Web Browser for Oberon

Smart Web is a fully integrated network browser that supports all important graphical facilities of the World-Wide Web service like in-line images, text styles, fill-out forms etc. It can be operated in a way similar to that of WWW browsers for other operating systems, i.e. by pointing at highlighted captions in order to download and view the linked documents.

The browser is represented in a panel document that contains a hypertext view for World-Wide Web pages and user interface elements for initiating

activities such as the requesting of documents and items. The panel document is opened like any other graphical document.

Figure 1: Operating Panel with text view displaying a WWW page

The Smart Web browser provides unlimited extensibility *at runtime*, both with respect to new network protocols and new document types accessible via the WWW. The WWW browser delegates activities like the requesting of network items, or the mapping of these items to data types of Oberon to specialized objects which process these activities in a generic way.

Because the implementation of the Smart Web browser is based on our document oriented user interface framework, it integrates completely with other online documents available in Oberon. For example, if a corresponding link is activated, the browser will allow to navigate *in place* also in TeleNews pages, or pages of the mentioned electronic encyclopedia.

Integration

To provide access to a network information service, first its application protocol has to be implemented. Additionally, appropriate mappings of the service's data formats to those of the operating environment have to be provided.

The data model of the World-Wide Web – namely hypertext – is suitable for representing information from several already existing network information services. These services frequently use lists, for example directories or menus, to organize their information. Because such lists can easily be mapped onto a hypertext structure, WWW browsers provide an integrated user interface for all these information services.

The *Gopher* service [AnMC*93], for instance, is usually supported in WWW browsers. It is a simple network service based on the model of a global file directory. Items accessed with the Gopher service are either directory lists or (textual, graphical or binary) documents. Gopher documents, however, do not contain further references, in contrast to Web documents. Therefore, they can be viewed as terminal nodes of the world-wide Gopher directory graph.

Besides integrating different network services, the Web also integrates different *media*, i.e. data formats of the information. We have already mentioned the hypertext format of textual information. In graphical browsers, at least the so called *Graphics Interchange Format* (GIF) for bitmap images has to be supported as well. The following table shows common network information systems which are usually supported by WWW browsers and some of their possible data formats.

Protocols\Formats	plain text	hypertext	bitmap	sound	menu list	binary file
http	*	*	*	*		
gopher	*	(*)	(*)	(*)	*	*
news	*				*	
ftp	*				*	*

Various additional formats for photographic images, sound files, video sequences, and even three-dimensional sceneries can potentially be integrated into the Web, too. Hence, the World-Wide Web is extensible in two dimensions: New services (i.e. new protocols) may be added as well as new media types (i.e. new types of documents).

Technical Background

The specification of the Web consists of three parts that form its technical basis: (a) a network protocol for the exchange of requests and responses, i.e.

the *hypertext transfer protocol* (HTTP) [BL93], (b) a specification of hypertext documents by means of markups called *hypertext markup language* (HTML) [BLCo93] and (c) a naming scheme for data items that are accessible via the internet called *universal resource locator* (URL).

The *hypertext transfer protocol* is an application protocol that controls activities on the World-Wide Web such as the request of items. It is a *stateless* protocol in the sense that there is no relation of a HTTP activity to any previous operation. Requests and responses are encoded as Ascii strings i.e. in a human readable form. A (minimal) transaction for obtaining an object has the following structure (C stands for the client's requests, S for the server's responses):

C: GET *object identification* HTTP/1.0 <CRLF>
C: Accept: *type* <CRLF>
C: <CRLF>

S: HTTP/1.0 200 OK <CRLF>
S: Content–type: *type* <CRLF>
S: Content–length: *length* <CRLF>
S: <CRLF>
S: *data*

The *GET* command is used to request the delivery of an item. The acceptable representation (data format) of the item (hypertext, bitmap graphic etc.) can be specified by a corresponding *Accept* entry in the request. The data format is specified by means of a so called *Internet Media Type* [Po94]. Some HTTP servers even respect such media requests and kindly avoid sending data the client is not able or not willing to process.

The response consists of a result line, a header that includes (among others) information about the actual representation of the item, and the length of the data. The header is followed by the data itself.

Textual data is transmitted by means of Ascii strings delimited by line feed characters. Depending on the media type of the data announced in the header, the browser interprets and processes the data as a text document, a raster image, a sound file, an animation sequence etc.

The *hypertext markup language* is used to express reference links to other hypertext documents. Additionally, it allows to specify the formatting of the document and hence the look of the resulting hypertext page in a device independent way. There are markups which structure the document by defining paragraphs, headings, enumeration lists etc. Other markups don't break the text flow but specify the rendering of the text with physical styles like "bold" or "italic", or with logical styles such as "emphasized". It depends on the

graphical abilities of the used browser if such formatting styles are applied or ignored. With a corresponding markup tag, inline images can be integrated into a hypertext page for representing icons and logos.

New versions of the markup language include support for fill-out forms. These extend the suitability of the WWW for interactive applications for which the ordinary hypertext interface is not adequate anymore. The following overview lists some of the most frequent markup tags.

HTML Tag Overview

Format Item	HTML Coding
Anchor	<A HREF=*item–url*> ... </A>
List	<UL> ... </UL>
List item	<LI>
Glossary	<DL> ... </DL>
Term	<DT>
Definition	<DD>
Paragraph	<P>
Explicit Linebreak	
Horizontal Ruler	<HR>
Styles	
Heading (levels)	<H1> ... </H1>, <H2> ... </H2>, ... , <H6> ... </H6>
bold	<B> ... </B>
italic	<I> ... </I>
emphasized	<EM> ... </EM>
strongly emphasized	<STRONG> ... </STRONG>
Inline Image	<IMG SRC=*image–url*>
Fill-out Form	<FORM ACTION=*action–url*> ... </FORM>
Submit Button	<INPUT TYPE=submit>
Checkbox	<INPUT TYPE=checkbox>
Radio Button	<INPUT TYPE=radio>
Textfield	<INPUT TYPE=text>

The *universal resource locator* specifies the referenced item including the coresponding protocol in order to acquire it in a generic way. By using the resource locator, also items from already existing network information services can be specified and thus be referenced from WWW documents. Note, that in

this sense HTTP is just one of several possible protocol types. It is the minimum a Web browser has to support. The URL uses the following format:

```
URL = method ":" ["//" host [":" port]] item.
```

Method specifies the method (i.e. the protocol) applicable in order to obtain the item. *Host* is the internet address of the server holding a copy of the *item* which in turn is identified in a protocol-dependent way. HTTP items, for example, are identified by a UNIX–like file path. The *port* number is to be supplied only if it is not equal to the dedicated standard port of the service.

Examples of URLs:

http://www.ethz.ch/swiss/Switzerland–Info.html	*hypertext document*
http://www.cs.indiana.edu/cstr/search?oberon	*keyword query*
gopher://gopher.ethz.ch/	*gopher item*
news:comp.lang.oberon	*news group list*
telnet://rereth@orion.inf.ethz.ch/	*telnet terminal session*
ftp://ftp.inf.ethz.ch/Oberon/System3/DOS/system.exe	*item accessible with ftp*

In order to supply arguments to interactive WWW services, HTTP items may additionally contain an argument list, separated from the item specifier by a question mark (see second example above).

Processing different Document Types

We start the discussion on the integration of the WWW with the presentation of the different data formats of HTTP items. The data format of a requested document is specified in the *Content–Type* field of the response header. The corresponding media specification corresponds uniquely to a method of processing the data. In our implementation, items are requested only, if and only if their file extensions indicate one of the currently supported media types. This behaviour prevents the browser from requesting items that it cannot process anyway, e.g. because its media type is not supported yet (Actually, the browser exposes all media types that it can handle in the *Accept* field of the request. However, some of the servers simply ignore the field).

Deriving (or better guessing) the document's *type* from its file extension is common practice in WWW browsers. However, the conclusive *determination* of the document's type is favourably done with the Content–Type field of the response header (if present).

In order to allow independently the sets of protocols and supported media types to be extended, the protocol dependent *item request* has to be decoupled

from the media type specific *item processing*. Such decoupling can be achieved by downloading the item into a file and subsequently process the content of the file. Some WWW browsers for other platforms apply this scheme and even delegate the actual processing of the item to an external program.

A more elegant, more efficient, and more integrated solution is applied in the WWW browser for Oberon. The solution is based on an installable parser method which is determined as soon as the type of the document is recognized during its downloading. Parser methods have the following structure:

```
PROCEDURE ContentType (C: NetSystem.Connection; VAR obj: Objects.Object;
                                                    inline: BOOLEAN);
```

The parser method is called after the desired item has been requested and the response header completely processed. The corresponding item's data can therefore immediately be downloaded and parsed directly "from the wire". For that purpose, the method is supplied with an network connection *C* as parameter. If the *inline* flag is true, the parser method returns in the *obj* reference parameter a ready-made object for being inserted into a text. Otherwise (i.e. inline equals false), the parser method is expected to generate an appropriate view of the object by itself and return NIL instead.

All media types for which there exists a parser method are enumerated in a configuration text. An entry in the configuration text lists the data item's file suffixes and an Oberon command which assigns the type-specific parser method to the global procedure variable *parse*:

```
parse: PROCEDURE (C: NetSystem.Connection; VAR obj: Objects.Object; inline: BOOLEAN);
```

The text file contains an entry for each of the currently supported document types with the following structure:

```
entry = media-type command file-suffix {file-suffix} CR.

"text/plain" WebHtml.Text "*.text" "*.txt" "*.c" "*.h" "*.conf"
"text/html" WebHtml.Html "*.html" "*.htm" "*"
"image/gif" WebImage.Gif "*.gif"
"image/x-xbitmap" WebImage.Xbm "*.xbm"
```

The following *LoadDoc* procedure passes control to the parser method that is associated to the *media* type specified as parameter.

```
PROCEDURE LoadDoc (C: NetSystem.Connection; media: ARRAY OF CHAR;
                                VAR obj: Objects.Object; inline: BOOLEAN);
  VAR res: INTEGER;
BEGIN
  parse := NIL;   (*reset parser method*)
  search media in list of currently known document types;
  IF found THEN   (*deposit parser method*)
    Oberon.Call(installer belonging to media, Oberon.Par, FALSE, res);
    IF parse # NIL THEN parse(C, obj, inline)   (*forward processing to parser method*)
    ELSE obj := NIL   (*error: no parser method installed*)
    END
  ELSE obj := NIL;
    Out.String("cannot handle "); Out.String(type); Out.Ln
  END
END LoadDoc;
```

The most important document type is the one which actually constitutes the World-Wide Web, namely structured text encoded with the hypertext markup language.

HTML Documents

The parser method which processes HTML encoded data has to map the properties specified in the HTML document to abstractions present in the Oberon system. The following table lists *properties*, their HTML markups and the corresponding abstraction in the Oberon system that is used to represent it.

Property	Markup	Representation in Oberon
Anchors	`<A HREF= ...> </A>`	Smart Links
Text Looks	`<B> </B>, <EM> </EM>, ...`	Fonts
Headings	`<H1> </H1>`	Fonts
Inline Images	`<IMG SRC= ...>`	Pictures
Lists	`<UL> </UL>, <DL> </DL>, ...`	Style Symbols
Form Elements	`<INPUT TYPE= ...>`	Gadgets

The HTML parser is based on a lexical scanner that turns markup sequences and character entities (character encoding by means of a literal paraphrase, such as Ä for Ä) into single characters.

The deficiency of discipline in the markup language's usage forces the paser to be extremely tolerant. Therefore, we have implemented it as a state machine where arriving markup tags either set or reset graphical states and modes. A state machine is easily reset to a reasonable state if illegal use of markups is discovered. For properties which may be nested such as lists and character styles (fonts) an attribute stack is maintained which collects the current

attribute state up to a certain nesting level. The whole processing of a HTML document and its conversion into a hypertext page happens in a *single pass*.

Mapping of Data Formats

As mentioned, textual data is transmitted by means of Ascii strings terminated by line feed characters. The line structure used for transmission is totally unrelated to the resulting hypertext presentation (except for so called *preformatted text*, where the line structure has to be preserved). Instead, the browser is assumed to support implicit line breaking as soon as a text line exceeds a certain width. Line breaks are therefore converted to space characters.

If an *anchor* markup tag is found, the corresponding document's address (specified with a universal resource locator) is extracted and a smart link is constructed to be inserted later into the text. All subsequent characters up to the ending anchor markup are printed in a highlighted fashion with a special color. After the ending markup has been found, the prepared smart link is inserted into the text flow and the text color reset.

Headings and the various different text looks are represented with different fonts. Indentation and nesting of lists and glossaries is done by means of appropriately attributed style symbols that are inserted into the text.

Inline images are immediately processed as soon as they are discovered; that is, processing of the containing document is suspended until the processing of an inline image has completed. The actual processing of the image data is performed by a media-specific parser method. It results in an Oberon picture object which is inserted into the text. Image data is cached locally to avoid unnecessary network traffic (decorative images are often referenced several times within a single page).

All other graphical items within a WWW page are mapped to standard components of Oberon System 3's graphical user interface toolkit Gadgets. These items include horizontal rulers, bullets in front of enumeration lists, and the fill-out forms elements like text fields and buttons. Their customization to the browser's needs confines to appropriate settings of attributes (sizes, commands etc.).

Processing different Network Services

Different network services imply different access protocols for requesting their items. Our concept of *Smart Links* suggests a simple solution for the integration of a potentially infinite number of different network services. In principle, the

integration of a new network service confines to the definition of a new type of smart links.

However, this general rule leads to the problem that the HTML parser must be able to generate smart links for all network services of which items are referenced in the current page. Remember that, while processing the current HTML page (cf. previous paragraph), hypertext links referring to further items are represented by smart links. These are inserted directly into the text of the page under construction. If the type of a smart link depends on the network service, the HTML parser has to know all currently supported network services in order to choose the appropriate smart link.

Hence, there is justification in this case to refine the rule stated above and to decouple the aspects of the smart link (which references an item) from the actual service (which processes the network access to retrieve the item). This decoupling leads to a two-level structure of the smart link activation: A single unified variant of a smart link is used to *reference* network items (irrespective of their access methods), whereas a service-specific service object is used to *process* the requests (parameterized smart links). The following figure illustrates the two-level structure of the smart link activation and compares it with the direct solution (on the left):

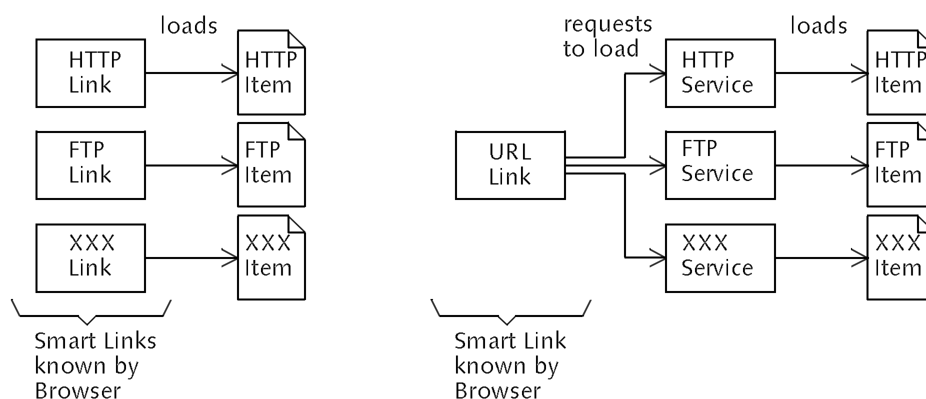


Figure 2: Decoupling of Smart Links and Services

As a consequence, only a single type of smart links has to be known to the browser even for different (and yet unknown) network services.

The smart link known to the parser thus represents a (general) *universal resource locator* instead of a (specific) network service. Its data type is listed below.

```
Link* = POINTER TO LinkDesc;
LinkDesc* = RECORD(SmartLinks.LinkDesc)
    method*: ARRAY 8 OF CHAR;
    host*: ARRAY 64 OF CHAR;
    port*: INTEGER;
    path*: ARRAY 256 OF CHAR
END;
```

The data type contains the different sections of the resource locator. The *method* field in the smart link descriptor specifies the service which actually loads the item. In the data type, the service which loads the item is thus represented only with its *name*. By means of the generic naming facility, the name is mapped to the actual service.

The decoupling of smart links and service implementations introduces a new level of indirection. This indirection allows to delay the identification of a specific network service up to the first resolution of a smart link referring to one of its items. It additionally allows to delay the *loading* of the corresponding service's modules into memory up to that moment.

The indirection leads to a two phase process of requesting an item. The following Request method of the smart link implements this process.

```
PROCEDURE Request(S: Services.Service; obj: Objects.Object; par: ARRAY OF CHAR):
                                                                    Objects.Object;
    VAR s: Services.Service;
BEGIN
    s := Services.GetService(S(Link).method);    (*find appropriate service*)
    IF s # NIL THEN
        RETURN s.Request(s, S, par)    (*download item*)
    ELSE
        Out.String(S(Link).method); Out.String(" not supported"); Out.Ln;
        RETURN NIL
    END
END Request;
```

If the smart link is requested to deliver the referenced item, it orders an appropriate service from the central service manager (which implements the naming of services). The corresponding service is identified by the *name* of the access method (which is very probably registered as an alias name for the service's generator command). The service manager returns the corresponding service which, in turn, is requested to load the item.

Summary and Conclusion

We have made accessible information of the World-Wide Web network service to users of the Oberon environment. The WWW browser tool that evolved from the project allows to bring textual and graphical information from anywhere in the world directly on the Oberon desktop.

For the implementation of the browser, we had to care only about the mere protocol oriented aspects of the World-Wide Web and the appropriate representation of WWW items with Oberon's basic data types. The latter could be achieved by customizing existing items of Oberon System 3's graphical user interface toolkit to the needs of the browser application. User interface aspects are handled completely by the document oriented application framework. Finally, the generic hypertext machinery introduced earlier in this thesis covered those parts of the browser tool which are not supported by the basic Oberon system already. As a consequence, the implementation of the World-Wide Web browser for Oberon confined to a module for WWW's transfer protocol and a module for the HTML parser. It is obvious, that the integration of further network information systems into the Oberon environment can be achieved with a similar implementation effort.

Differences to existing Web browsers

Although the functionality of the Web browser for Oberon is far from being complete and therefore cannot compete with existing Web browsers for other operating platforms (e.g. Netscape [HTTP2]), the presented browser is unique for several reasons.

In contrast to World-Wide Web browsers for other platforms, Web for Oberon is *fully integrated*; that is, the Web browser is not a separate application but a collection of customized system components. It presents itself as a document that can be opened like any document in the system. User activities within the Web document (e.g. the requesting of Web items) have identical semantics as user activities in arbitrary other documents (e.g. the invoking of commands). In fact, there is no Web "browser" in Oberon at all. There is only a type of smart links and collection of services that are embedded in an ordinary text which, in turn, is displayed in an arbitrary document viewer.

Web for Oberon is *fully integrating* with arbitrary other online documents, because links to any other local and remote services (e.g. an online encyclopedia) may be embedded as well. For example, if the user invokes the

online encyclopedia on a selected keyword in the currently visible text, the result is displayed as if it resulted from a Web request. As long as the user remains within the encyclopedia context, further activities are directed to the online encyclopedia. After returning to the Web context (e.g. by means of the context stack) the user can continue navigating in the Web document space.

Web for Oberon is *arbitrarily extensible at runtime* with respect to new network services as well as to new document types (i.e. data representations of network items). The Web platform provides only the basic functionality such as connection establishment etc. The actual (service- and media-specific) functionality, however, is located in the different components that handle either different network services or different data representations. These components can be freely added at runtime without the necessity to reboot the application, i.e. while the Web environment is loaded and operational.

Whereas almost all available WWW browsers allow to extend the set of supported document types in a similar way, typically, browsers for other platforms only fetch the corresponding data and store it in a temporary file. The actual *interpretation* of the data is delegated to an external program which is associated with the corresponding document type in an initialization file. In the Web for Oberon, a slightly more integrated solution is provided by means of a parser method that is installed *within* the scope of the Web engine. This results in a more elegant and more efficient processing of different media types.

Extensibility with respect to *network services* is usually not possible in browsers for other platforms. Instead, such Web browsers are equipped with a certain number of currently supported network services that are "burnt in". In the Web for Oberon new network services (represented as new service types) can be added freely. These services are not loaded into memory until one of its items is requested. Instead, they are loaded one by one during navigating through the WWW space. The mere reference to an item within the visible page therefore does not require the presence of the corresponding service's program module!

The concept of autonomous smart links allows *customizable World-Wide Web "tool" texts* (in the sense of Oberon *Tools*) to be constructed. Because its smart links are objects freely flowing in the text, such a "tool" can be constructed by simply copying the highlighted hypertext anchors with their links into a local text document. The latter can be stored as usual. Within a hypertext view (i.e. a text view equipped with the mentioned activation command), the tool text can immediately become active.

Outlook

The World–Wide Web seems to become a standard platform for all kinds of network information systems accessible over the Internet. Its extensibility of protocols as well as document types due to a generic identification scheme allows also to integrate new and yet unrecognized possibilities and features.

A vision of the future of the Web is that of world-wide distributed multimedia books extended with interactive facilities such as the manipulation of visual data over the network. In fact, due to the accessing scheme of the World-Wide Web, the presented data needs not to be permanently present for ready-made access but can be immediately constructed at the occasion of an incoming request [So95]. This opens a completely new interactive potential which still has to be explored.

Recently, a development group in the UK offered the possibility to inspect the current level in the coffee pot of their video supervised coffee machine through the World-Wide Web [HTTP4]. This interactive application of the "internet coffee pot" can only be considered a caricature of this potential.

Conclusions and Outlook

Conclusions

The project presented in this thesis aimed at the integration of access to local and remote services into the Oberon system. The goal was to obtain a unified programming model and user environment for the interaction with these services. Such an environment constitutes a prerequisite and starting point for successfully exploring the inherent potential of interactive online multimedia applications.

The integrated user environment and programming model is essentially based upon two concepts: First, the *online document* as a model for information that is processed and displayed by services and, second, *smart links* as user interface elements that represent abstract services.

An important design principle was the strict separation of the three aspects (and phases) of the smart link's activation. These include the link's initiation, its processing (i.e. the service call), and the rendering of the result on the screen. The actual service call very often proves to be the only phase that involves a service-specific processing; that is, the other phases of the link resolution can be handled identically for all services. Therefore, the implementation of a new service is basically confined to these service-specific issues such as the service's protocol and its data representation. The separation of the three aspects thus allows to integrate new services very efficiently and hence to add new functionality into the environment. A rough integration of a simple service (for example the gopher service mentioned in the previous chapter) can thus be achieved in only about an afternoon.

The integration of different online media and sources of information requires a consistent user interface and presentation model. We believe that documents provide a suitable abstraction for both the information that is delivered by services as well as their user interface. The smart document as the integration of an online document (as the model of the information) and the document oriented user interface offers the user a familiar presentation and behaviour. For example, a smart document is opened like any other document in the system, and service calls are initiated like ordinary commands.

Experience with the integration of several online documents allows to draw the conclusion that Oberon is not only a suitable but an ideal platform for accessing information services. Oberon especially supports and simplifies such functional extensions due to the following properties and characteristics:

Oberon's *document oriented interface model* extends naturally to a universal user interface for services. Accessing a service can be interpreted as opening a document (an online document, respectively). Furthermore, Oberon's textual user interface and hypertext have very similar (stateless) semantics. The most conspicious difference is that the hypertext link arbitrarily *paraphrases* an activity that is invoked while the command name exactly and explicitely *identifies* the activity.

Oberon consists of *powerful system components* that restrict the effort of the implementation of service access essentially to aspects related to the actual interaction with the service. These components include attributed text and graphical objects which can be used to build user interface platforms with. Furthermore, these components can be easily customized for example to be used as mappings of the service's abstractions.

Oberon is a *powerful programming platform* that allows an efficient realization of a service integration project. For example, the mentioned system components can be customized also with respect to their programmed functionality, e.g. by installing a command. Most of the user interfaces can even be built by interactive composition, i.e. without any programming.

Outlook

Objects that are delivered by services are constructed locally by the (local) representative of the (possibly remote) service. Remote services thus only provide the *data* that the local objects consist of, but not the objects themselves. For example, the (remote) service delivers a binary image file that is used to construct a local picture object.

To extend the range of deliverable items in a generic way, objects *themselves* must be delivered, instead of only their data. The inclusion of such portable objects with autonomous functionality is therefore obviously the next step. However, this raises the question of how to transmit the necessary executable code with the object.

In a homogeneous computing environment with compatible processing facilities, the object's machine-code can be transmitted directly. In an environment with different and incompatible computing platforms, a portable

representation of the functionality is required (so called portable or virtual object code).

There are two interesting proposals for such portable object code. The *Java* project [Ja95] aims at small portable applications, so called *applets*. These can be transported over network connections, for example, to equip World-Wide Web pages with autonomous functionality. Java applets are executed on a runtime system that emulates a virtual machine. The *Oberon Module Exchange* (OMEGA) proposal [Fr94] is based on a code generating loader. It compiles an intermediate representation of an Oberon module *at loading time* to the target machine's native object code. Both the *Java* project and also the *Oberon Module Exchange* proposal provide a promising foundation for such portable objects.

Future research will be doubtlessly directed towards even higher integration. Information that is provided by one service can be combined with information that originates from another source. For example, the data base of the electronic encyclopedia might be extended with links to the teletext service. If the user requests information from the encyclopedia service, additional information that is currently found in the teletext data base can be delivered as well.

Also possible is an integration of other online documents like the electronic encyclopedia or TeleNews with World-Wide Web. Looking at the World-Wide Web as a broadly accepted transport media for online services, such an integration has two important advantages: On the one hand, the programmer needs not to care about the definition of the transport protocol. On the other hand, the online document is immediately accessible by anyone who has access to the WWW. However, integration of online resources into the WWW requires a server software that handles and processes the corresponding requests. In [So95] a very simple server framework is suggested that allows to install such resources with minimal effort.

Bibliography and References

[Ar87] D. Arnon, *Report of the Workshop on Environments for Computational Mathematics*, RFC 1019

[ArBe*88] D. Arnon, R. Beach, K. McIsaac, C. Waldspurger, *Caminoreal: An Interactive Mathematical Notebook*, Proc. on Electronic Publishing, Document Manipulation and Typography (EP88), Cambridge University Press 1988

[AnMC*93] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. John, D. Torrey, B. Alberti, *The Internet Gopher Protocol (a distributed document search and retrieval protocol)*, RFC 1436, 1993

[BLCa90] T. Berners–Lee, R. Cailliau, *WorldWideWeb: Proposal for a HyperText Project*, http://info.cern.ch/hypertext/WWW/Proposal.html, 1990

[BL93] T. Berners–Lee, *Hypertext Transfer Protocol*, Internet Draft, 1993

[BLCo93] T. Berners–Lee, D. Conolly, *Hypertext Markup Language*, Internet Draft, 1993

[Ba92] L. Bauer, *Videotext für alle*, c't 7/92, pp 176–182

[BeDe91] E. Berk, J. Devlin, (ed.), *Hypertext/hypermedia handbook*, McGraw Hill, 1991

[ChGe*88] B. W. Char, K. O. Geddes, G. H. Gonnet, M. B. Monagan, S. M. Watt, *Maple(TM) Reference Manual 5th Edition*, Watcom, 1988

[Co95] D. E. Comer, *Internetworking with TCP/IP*, Prentice–Hall, 1988, 1991, 1995

[Di93] S. Dickman, *Die Zeitung der Zukunft steht auf Chips*, Tages Anzeiger, 21. 10. 1993

[Do93] H. Domjan, *Ceres-TXT*, term project ICS–ETHZ, 1993

[DoWi85] J. Donahue, J. Widom, *Whiteboards: A Graphical Database Tool*, CSL–85–4, PARC, 1985

[DT94] Deutsche Telekom AG, *KIT: Window-based Kernel for Intelligent Communication Terminals*, Technical Specification, 1994

[EBU92] European Broadcasting Union, *Teletext Specification*, Interim Technical Document, SPB 492

[ElGi89] C. A. Ellis, S. J. Gibbs, *Active Objects: Realities and Possibilities*, in: Object–Oriented Concepts, Databases and Applications, Won Kim ed., ACM Press, 1989

[GoRo83] A. Goldberg, D. Robson, *Smalltalk–80: The Language and its Implementation*, Addison–Wesley, 1983

[Gu93] J. Gutknecht, *Oberon System 3 – A Realm of Persistent Objects*, Internal Draft, 1993

[Gu94a] J. Gutknecht, *Oberon System 3: Vision of a Future Software Technology*, Software Concepts & Tools 15:1, 1994

[Gu94b] J. Gutknecht, *Oberon–Perspectives of Evolution*, Proc. on Joint Modular Languages Conference (JMLC), 1994

[HüHa93] Ch. Hüser, A. Haake, *The Hypermedia Presentation Composer: A Tool for Automatic Hyperdocument Delivery*, Proc. Hypermedia '93, Springer, 1993

[ISO86] ISO 8879:1986, *Information Processing Text and Office Systems Standard Generalized Markup Language (SGML)*, 1986

[Kr92] E. Krol, *The Whole Internet. User's Guide and Catalog*, O' Reilly & Associates Inc., 1992

[Kn84] D. E. Knuth, *The TEXbook*, Addison Wesley, 1984

[LRG76] Xerox Parc Learning Research Group, *Personal Dynamic Media*, SSL–76–1, PARC 1976

[Le90] B. L. Leong, (indirect) Electronic communication, 1990

[Me86] N. Meyrowitz, *Intermedia: The Architecture and Construction of an Object–Oriented Hypermedia System and Applications Framework*, Sigplan Notices 21:11, 1986

[Os93] E. Oswald, *Polyworlds*, Diploma Thesis ETHZ, 1993

[Ph90] *SAA5246 Integrated VIP and teletext (IVT)*, Datasheet, Philips, 1990

[Pi94] R. Pike, *Acme: A User Interface for Programmers*, AT&T Bell Laboratories, 1994

[PoRe83] J. Postel, J. Reynolds, *Telnet Protocol specification*, RFC 854, 1983

[PoRe85] J. Postel, J. Reynolds, *File Transfer Protocol*, RFC 0959, 1985

[Po94] J. Postel, *Media Type Registration Procedure*, RFC 1590, 1994

[Re91] M. Reiser, *The Oberon System*, Addison Wesley, 1991

[ReWi92] M. Reiser, N. Wirth, *Programming In Oberon*, Addison Wesley, 1992

[Schä91] H. R. Schär, *Integrierte interaktive Bearbeitung mathematischer Formeln im Dokumenteneditor Lara*, Diss. ETH, 1991

[So94] R. Sommerer, *Script User Guide*, User Guide as part of the official Oberon System 3 Release, 1994

[So95] R. Sommerer, *Integrating Oberon Resources into the World-Wide Web*, Proc. GISI'95, Springer, 1995

[Su95] A. Suter, *Online Monitor für Börsenkurse basierend auf Teletext*, diploma thesis ETHZ, 1995

[Szy91] C. A. Szyperski, *Write – An Extensible Text Editor for the Oberon System*, Tech. Report 151, Institut für Computersysteme, ETH Zürich, 1991

[ThRi74] K. Thompson, D. A. Ritchie, *The UNIX Time-Sharing System*, Comm. ACM, 17:2, 1974

[Vo89] S. Vorkoetter, *Software Portability by Virtual Machine Emulation*, Diss. Thesis, University of Waterloo, 1989

[Vo94a] S. Vorkoetter, *Open Math Proposal*, Draft, 1994

[Vo94b] S. Vorkoetter, private communication

[WeGa*89] A. Weinand, E. Gamma, R. Marty, *Design and Implementation of ET++, a Seamless Object–Oriented Application Framework*, Structured Programming 10:2, 1989

[Wi88] N. Wirth, *Type Extensions*, ACM Trans. on Programming Languages and Systems, 10:2, 1988

[WiGu92] N. Wirth, J. Gutknecht, *Project Oberon*, Addison Wesley, 1992

[Ze88] P. T. Zellweger, *Active Paths Trough Multimedia Documents*, Proc. on Electronic Publishing, Document Manipulation and Typography (EP88), Cambridge University Press, 1988

Electronic References (without explicit autorship)

[HTTP0] http://info.cern.ch/hypertext/WWW/TheProject.html
[HTTP1] http://info.cern.ch/hypertext/WWW/History.html
[HTTP2] http://home.netscape.com/
[HTTP3] http://info.cern.ch/hypertext/WWW/MarkUp/HTMLPlus/htmlplus-1.html
[HTTP4] http://www.cl.cam.ac.uk/coffee/coffee.html