Fine–grained Integration of Oberon into Windows using Pluggable Objects

Emil Johann Zeller

Diss. ETH No 14877

Fine–grained Integration of Oberon into Windows using Pluggable Objects

A dissertation submitted to the Swiss Federal Institute of Technology Zürich

for the degree of Doctor of Technical Sciences

presented by
Emil Johann Zeller
Dipl. Informatik–Ing. ETH
born January 14, 1969
citizen of Appenzell, AI

accepted on the recommendation of
Prof. Dr. Jürg Gutknecht, examiner
Prof. Dr. Thomas Stricker, co–examiner
Prof. Dr. Clemens Szyperski, co–examiner

2002

Acknowledgements

First and foremost, I would like to thank Prof. Jürg Gutknecht, for the opportunity to work in his research group, and for his patient and liberal supervision. I am also very grateful to Prof. Thomas Stricker and Prof. Clemens Szyperski. Both readily agreed to co–supervise this thesis and provided valuable suggestions which helped to enhance the structure of this book.

Special thanks go to André Fischer, Jörg Rentsch, and Wolfgang Bock for proofreading the first draft of the manuscript and helping me improve my writing.

I want as well to thank all my colleagues at the Institut of Computer Systems for contributing to the lively and inspiring working atmosphere.

Last but not least, my deepest gratitude goes to my friends and family. I simply can not imagine having done my studies and research without their encouragement and support.

Contents

Abstract **xi**

Kurzfassung **xii**

1 Introduction **1**
1.1 Motivation 1
1.2 Overview 1
1.3 Component Software 2
1.3.1 Java Beans 4
1.3.2 COM 6
1.3.3 .NET Framework 8
1.4 The ETH Oberon System 13
1.5 Related Work 21
1.5.1 Juice / Gazelle 21
1.5.2 BlackBox Component Builder 23
1.5.3 Java Beans Tools for ActiveX 25
1.5.4 VMWare, Virtual PC 26

2 Emulating a System on Top of Another **29**
2.1 Bootstrapping the System 30
2.2 Automatic Memory Management 30
2.3 IO and CPU Utilization 31
2.4 Filesystem 32
2.5 Display System 33

3 A new Oberon Display System 35
3.1 Introduction . . . 35
3.2 A new Display Scheme . . . 36
3.3 A new Desktop Model . . . 41
3.4 Documents and Applications . . . 44
3.5 Using standard Menus and Dialogs . . . 45

4 A Plug–in Kernel for Oberon 51
4.1 Introduction . . . 51
4.2 Bootlinker and Bootloader . . . 52
4.3 Compiler Extensions . . . 56
4.4 Exception Handling . . . 59
4.5 Multithreading Support . . . 62

5 Pluggable Objects as Web Applets 67
5.1 Introduction . . . 67
5.2 Pluggable Oberon Object . . . 69
5.2.1 Packages . . . 70
5.2.2 PlugIns . . . 74
5.3 Security Issues . . . 76
5.4 The Oberon Web Browser . . . 78

6 Oberon Netscape Browser Plug–in 81
6.1 Introduction . . . 81
6.2 Implementation of the Plug–in DLL . . . 82
6.2.1 NPInitialize . . . 83
6.2.2 NPGetEntryPoints . . . 84
6.2.3 NPShutdown . . . 85
6.2.4 Plug–In Window . . . 85
6.2.5 Plug–In Streams . . . 89
6.2.6 Linking the Plug–In DLL . . . 90
6.3 Installing the Plug–in DLL . . . 92
6.4 Interfacing to Java using LiveConnect . . . 93
6.4.1 Java Runtime Interface . . . 94
6.4.2 LiveConnect Implementation . . . 95
6.4.3 JavaScript Example . . . 99

7 Oberon ActiveX Components **103**
7.1 Introduction . . . 103
7.2 An ActiveX and DCOM Server . . . 108
7.3 OLEObjects . . . 113
7.3.1 IUnknown . . . 113
7.3.2 IDispatch . . . 115
7.3.3 IPersist . . . 118
7.3.4 IDataObject . . . 119
7.3.5 Events . . . 121
7.4 OLEFrames . . . 124
7.4.1 IOleObject . . . 124
7.4.2 IOleInPlaceObject . . . 127
7.4.3 IOleInPlaceActiveObject . . . 128
7.4.4 IViewObject . . . 128
7.4.5 IOleControl . . . 129
7.4.6 IQuickActivate . . . 130
7.5 OLEDataServices . . . 130
7.5.1 Clipboard . . . 131
7.5.2 Client Site . . . 131
7.5.3 Drag and Drop . . . 133
7.5.4 Document Files . . . 134
7.6 OLEPlugIns / Internet Explorer . . . 135
7.7 OLEObjectScripts . . . 136

8 Case Studies **139**
8.1 Web Applets . . . 139
8.1.1 Simple Applets . . . 139
8.1.2 Applets using Network Streams . . . 143
8.1.3 Applets and Scripting . . . 146
8.2 ActiveX Controls . . . 148
8.2.1 Voyager and Microsoft Office . . . 151
8.2.2 Visual Basic . . . 151

9 Conclusions **155**
9.1 Evaluation . . . 155
9.2 Future Perspectives . . . 156

A Module Statistics **157**

B Performance Evaluation **165**

Bibliography **177**

Trademark notice

Abstract

The ETH Oberon System has been implemented in two fundamentally different ways: as native operating system and as an application running on top of another operating system. The embedded versions of Oberon are typically implemented as full screen applications, which are poorly integrated in their host environment. On the on hand, this approach has the advantage of maintaining the look–and–feel of a native Oberon implementation. On the other hand, developers of Oberon software would sometimes want to reuse it or make it available on a fine–grained level as part of commercial applications.

In this thesis, a generic solution to this problem using a software component approach is presented. Visual and non–visual Oberon components — ranging from simple gadgets such as buttons, to complex containers such as desktops — are subject to be plugged into foreign contexts of different kinds, e.g. web pages, spreadsheets, etc. Such pluggable objects are able to cooperate and communicate with their context and with other components using technologies like in–place activation, scripting, automation, clipboard, drag–and–drop, persistency, etc.

Although the implementation described in this thesis is specific to the Windows platform, most of the fundamental ideas can easily be adopted for other platforms.

Kurzfassung

Das ETH Oberon System wurde bisher auf zwei grundsätzlich verschiedene Arten implementiert. Einerseits als eigenständiges Betriebssystem und andererseits als Anwendung, die auf einem anderen Betriebssystem abläuft. Die eingebetteten Versionen von Oberon werden typischerweise als Vollbild–Anwendung realisiert, welche nur schlecht in die jeweilige Gastumgebung integriert sind. Dieser Ansatz hat den Vorteil, dass eine sehr genaue Emulation eines eigenständigen Oberon Systemes angeboten werden kann. Entwickler von Oberon Software möchten aber oft ihre Programme als integrierten Teil einer kommerziellen (nicht Oberon) Anwendung benutzen können.

In dieser Dissertation wird ein allgemeiner Ansatz zur Lösung dieses Problems vorgestellt. Damit können sichtbare und unsichtbare Oberon Komponenten, von elementaren Objekten bis hin zu komplizierten Objekt–Hierarchien, in fremden Umgebungen verschiedenster Art (z.B. Web Seiten oder Tabellen) eingesetzt werden. Diese Komponenten können mit ihrer Umgebung und anderen Komponenten mit einer Vielzahl von Standard Techniken kooperieren.

Obwohl die in dieser Dissertation beschriebene Implementation auf dem Windows Betriebssystem basieren, können die meisten der beschriebenen Ideen auch auf andere Betriebsysteme angewandt werden.

Chapter 1

Introduction

1.1 Motivation

The Oberon system has been implemented on different computer platforms, where it either runs natively or on top of another operating system. All existing ports of the Oberon system running on top of another system are implemented to provide a complete emulation of a native Oberon system. Unfortunately none of these ports provide additional functionality taking advantage of the underlying system.

In this thesis different extension to the Oberon system are discussed. All these extensions are made possible by introducing relatively small extensions in the system core.

1.2 Overview

This thesis is organized as follows:

- This first chapter gives a survey of topics closely related to this thesis. This are:
  - popular component software system standards
  - the ETH Oberon System and the Oberon programming language

  - other projects which try to embed Oberon in a foreign context

- Chapter 2 discusses typicall problems encountered when embedding run–time systems into each other.
- Chapter 3 discusses the implementation of a new display system for Oberon. This makes it possible to integrate existing frames into a wide range of different non–Oberon window contexts.
- Chapter 4 discusses the techniques needed to run a minimal Oberon system on Windows. This includes the following topics:
  - bootlinker and bootloader
  - interfacing with the Win32 API
  - compiler support for different calling conventions
  - exception handling
  - multithreading
- Chapter 5 describes extensions to Oberon needed to use Oberon objects as web applets.
- Chapter 6 describes the implementation of a browser plug–in for using Oberon applets with the Netscape browser.
- Chapter 7 describes an Oberon framework for developing COM components. This framework is then used to implement ActiveX wrappers for visual and non–visual Oberon objects.
- Chapter 8 presents case studies where Oberon objects are used as plug–ins into web browsers and other container applications.
- Chapter 9 presents conclusions and reflections.

1.3 Component Software

Object–oriented programming has long been advertised as a solution to many software problems at hand. However, while object–oriented

programming is powerful, it has yet to reach its full potential. Because no standard framework exists through which software objects created by different vendors can interact with one another. The major result of the object–oriented programming revolution has been the production of 'islands of objects' that cannot talk to one another across the sea of application boundaries in a meaningful way.

The solution is a system in which application developers create reusable software components. A component is a reusable piece of software in binary form that can be plugged into other components from other vendors with relatively little effort. For example, a component might be a spell–checking facility sold by one vendor that can be plugged into several word processing applications of different vendors. Software components must adhere to a binary external standard, but their internal implementation is completely unconstrained. They can be built using procedural languages as well as object–oriented languages and frameworks, although the latter provide many advantages in the component software world.

Software components are much like integrated circuit (IC) components, and component software is the integrated circuit of tomorrow. The software industry of today is very much where the hardware industry was 30 years ago. At that time, vendors learned how to miniaturize transistors and put them into a package so that no one ever had to figure out how to build a particular discrete function — a NAND gate for example — ever again. Such functions were made into an integrated circuit, a neat package that designers could conveniently buy and design around. As the hardware functions got more complex, the ICs were integrated to make a board of chips to provide more complex functionality and increased capability. As integrated circuits got smaller yet provided more functionality, boards of chips became just bigger chips. So hardware technology now uses chips to build even bigger chips.

Currently there are two widely used component software systems: Sun's Java components called 'Java Beans' [Sun97] and Microsoft's 'Component Object Model' COM [Mic95].

1.3.1 Java Beans

A Java Bean is a reusable software component that can be manipulated visually in a builder tool. Some Java Beans may be simple GUI elements such as buttons and sliders, whilst others may be sophisticated visual software components such as database viewers, or data feeds. Some Java Beans may have no GUI appearance of their own, but may still be composed together visually using an application builder.

Individual Java Beans will vary in the functionality they support, but the typical unifying features that distinguish a Java Bean are:

- Support for **introspection** so that a builder tool can analyze how a Bean works
- Support for **customization** so that when using an application builder a user can customize the appearance and behaviour of a Bean.
- Support for **events** as a simple communication metaphor than can be used to connect up Beans.
- Support for **properties**, both for customization and for programmatic use.
- Support for **persistence**, so that a Bean can be customized in an application builder and then have its customized state saved away and reloaded later.

To expose a Bean's features, Java introduces the notion of method pattern, a combination of rules for the formation of a method signature, its return type, and its name. The three most important features of a Java Bean are the set of properties it exposes, the set of methods it allows other components to call, and the set of events it fires.

Properties are named attributes associated with a Bean that can be read or written by calling appropriate methods on the Bean. For example, a Bean might have a color property that represents its foreground color. This property might be read by calling a *getColor* method and updated by calling a *setColor* method.

The methods a Java Bean exports are just normal Java methods which can be called from other components or from a scripting environment. By default, all the public methods of a Bean will be exported, but a Bean can choose to export only a subset of its public methods.

Events provide a way for one component to notify other components that something interesting has happened. Under the new AWT (Abstract Window Toolkit) event model, an event listener object can be registered with an event source. When the event source detects that something interesting happens, it will call an appropriate method of the event listener object.

The following example demonstrates the complete implementation of an elementary visual Bean.

```
import java.awt.*;
import java.beans.*;
import java.io.*;

public class Skeleton extends Component implements Serializable {
  private Color col;

  public Skeleton() {
    col = Color.red;
  }

  public synchronized void paint(Graphics g) {
    g.setColor(col);
    g.fillRect(0, 0, getWidth(), getHeight());
  }

  public Dimension getPreferredSize() {
    return new Dimension(50, 50);
  }

  public void setColor(Color col) {
    this.col = col;
  }

  public Color getColor() {
    return this.col;
  }
}
```

The current Java Beans specification does not address issues of containers. But the new 'Glasgow' specification does include container support as well as other new features:

- Support for aggregating several objects to make a Bean, based on the *Beans.isInstanceOf* and *Beans.getInstanceOf* APIs provided in Beans 1.0 [Sun97].

- Support for Bean contexts. This provides a containment hierarchy for Java Beans and provides ways for Beans to find context–specific information such as design–mode versus run–mode.
- Support for using Beans as MIME viewers. This includes the ability for a Bean to act as a viewer for a given kind of MIME typed data.
- Drag–and–drop support.

1.3.2 COM

The Component Object Model [Mic95] is Microsoft's foundation on which all component software of its (Windows) platforms are based. COM provides the following features:

- binary standard for component interoperability
- programming language–independence
- local/remote transparency
- robust evolution of component–based applications and systems
- support for multiple platforms (Windows 9X, Windows NT, Macintosh, UNIX)

Binary Standard

For any given platform (hardware and operating system combination), COM defines a standard way to lay out virtual function tables (vtables) in memory (see figure 1.1), and a standard way to call functions through them. Thus, any language that can call functions via pointers can be used to write components that can interoperate with other components written to the same binary standard.

Interfaces

In COM, applications interact with each other and with the system through collections of functions called *interfaces*. A COM interface is a strongly–typed contract between software components to provide

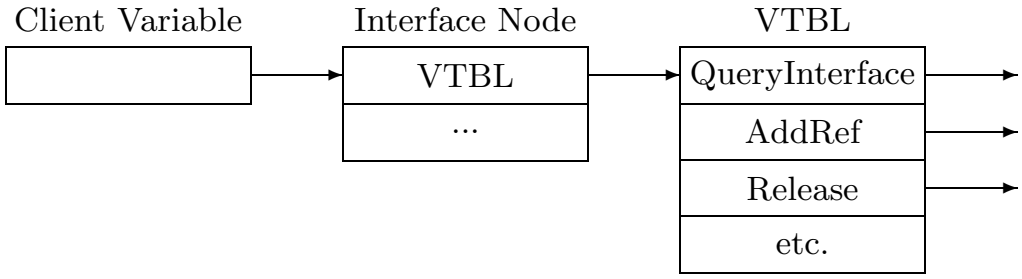

Figure 1.1: COM interface and virtual function table.

a small but useful set of semantically related operations (methods). An interface is the definition of an expected behaviour and expected responsibilities. A pointer to a component object is in fact a pointer to one of the interfaces that the component object implements. Thus, a component object pointer can only be used to call a method, but not to access or modify data directly.

Every interface has its own interface identifier, a globally unique ID (GUID), thereby eliminating any chance of collision that could occur with human–readable names. To create a new interface, a developer must also create a new identifier for that interface. To use an interface, a developer must use the identifier for the interface to request a pointer to the interface.

COM interfaces are never versioned, which means that version conflicts between new and old components are avoided. A new version of an interface, created by adding more functions or changing semantics, is an entirely new interface and is assigned a new unique identifier.

Globally Unique Identifiers (GUIDs)

COM uses globally unique identifiers — 128–bit integers that are guaranteed to be unique in the world across space and time — to identify every interface and every component object class. These globally unique identifiers are UUIDs (universally unique IDs) as defined by the Open Software Foundation's Distributed Computing Environment [Ope97]. Developers create their own GUIDs when they develop component objects and custom interfaces, using Microsoft supplied tools (uuidgen, guidgen) that automatically generate GUIDs.

IUnknown

COM defines one special interface, *IUnknown*, to implement some essential functionality. All component objects are required to implement the *IUnknown* interface, and all other COM interfaces must be derived from *IUnknown*. Using MIDL (Microsoft Interface Description Language), the definition of *IUnknown* is written as follows:

```
interface IUnknown {
  HRESULT QueryInterface(
      [in] REFIID riid,
      [out, iid_is(riid)] void **ppvObject);
  ULONG AddRef();
  ULONG Release();
}
[ object, uuid(00000001-0000-0000-C000-000000000046) ]
```

AddRef and *Release* are simple reference counting methods. Each component object maintains a reference count. As long as the reference count is greater than zero the object must remain in memory. When the reference count becomes zero, the component object can safely unload itself. *QueryInterface* is the mechanism that allows clients to dynamically discover (at run–time) whether or not an interface is supported by a component object. At the same time, it is the mechanism that a client uses to get an interface pointer from a component object.

1.3.3 .NET Framework

The .NET Framework [Mic01] is a new computing platform that simplifies application development in the highly distributed environment of the Internet. The .NET Framework is designed to fulfill the following objectives:

- To provide a consistent object–oriented programming environment whether object code is stored and executed locally, executed locally but Internet–distributed, or executed remotely.

- To provide a code–execution environment that minimizes software deployment and versioning conflicts.

- To provide a code–execution environment that guarantees safe execution of code, including code created by an unknown or semi–trusted third party.
- To provide a code–execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows–based applications and Web–based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework has two main components: the common language runtime (CLR) and the .NET Framework class library. The common language runtime is the foundation of the .NET Framework. You can think of the runtime as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that ensure security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as managed code, while code that does not target the runtime is known as unmanaged code. The class library, the other main component of the .NET Framework, is a comprehensive, object–oriented collection of reusable types that you can use to develop applications ranging from traditional command–line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.

The .NET Framework can be hosted by unmanaged components that load the common language runtime into their processes and initiate the execution of managed code, thereby creating a software environment that can exploit both managed and unmanaged features. The .NET Framework not only provides several runtime hosts, but also supports the development of third–party runtime hosts.

Internet Explorer is an example of an unmanaged application that hosts the runtime (in the form of a MIME type extension). Using Internet Explorer to host the runtime enables you to embed managed

components or Windows Forms controls in HTML documents. Hosting the runtime in this way makes managed mobile code (similar to Microsoft ActiveX controls) possible, but with significant improvements that only managed code can offer, such as semi–trusted execution and secure isolated file storage.

The Common Language Runtime (CLR)

The common language runtime manages memory, thread execution, code execution, code safety verification, compilation, and other system services. These features are intrinsic to the managed code that runs on the common language runtime.

With regards to security, managed components are awarded varying degrees of trust, depending on a number of factors that include their origin (such as the Internet, enterprise network, or local computer). This means that a managed component might or might not be able to perform file–access operations, registry–access operations, or other sensitive functions, even if it is being used in the same active application.

The runtime enforces code access security. For example, users can trust that an executable embedded in a Web page can play an animation on screen or sing a song, but cannot access their personal data, file system, or network. The security features of the runtime thus enable legitimate Internet–deployed software to be exceptionally feature rich.

The runtime also enforces code robustness by implementing a strict type– and code–verification infrastructure called the common type system (CTS). The CTS ensures that all managed code is self–describing. The various Microsoft and third–party language compilers generate managed code that conforms to the CTS. This means that managed code can consume other managed types and instances, while strictly enforcing type fidelity and type safety.

In addition, the managed environment of the runtime eliminates many common software issues. For example, the runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. This automatic memory management resolves the two most common application errors, memory leaks and invalid memory references.

The runtime also accelerates developer productivity. For example,

programmers can write applications in their development language of choice, yet take full advantage of the runtime, the class library, and components written in other languages by other developers. Any compiler vendor who chooses to target the runtime can do so. Language compilers that target the .NET Framework make the features of the .NET Framework available to existing code written in that language, greatly easing the migration process for existing applications.

While the runtime is designed for the software of the future, it also supports software of today and yesterday. Interoperability between managed and unmanaged code enables developers to continue to use necessary COM components and DLLs.

The runtime is designed to enhance performance. Although the common language runtime provides many standard runtime services, managed code is never interpreted. A feature called just–in–time (JIT) compiling enables all managed code to run in the native machine language of the system on which it is executing. Meanwhile, the memory manager removes the possibilities of fragmented memory and increases memory locality–of–reference to further increase performance.

.NET Framework Class Library

The .NET Framework class library is a collection of reusable types that tightly integrate with the common language runtime. The class library is object oriented, providing types from which your own managed code can derive functionality. This not only makes the .NET Framework types easy to use, but also reduces the time associated with learning new features of the .NET Framework. In addition, third–party components can integrate seamlessly with classes in the .NET Framework.

For example, the .NET Framework collection classes implement a set of interfaces that you can use to develop your own collection classes. Your collection classes will blend seamlessly with the classes in the .NET Framework.

As you would expect from an object–oriented class library, the .NET Framework types enable you to accomplish a range of common programming tasks, including tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized development scenarios. For example, you can use the

.NET Framework to develop the following types of applications and services:

- Console applications.
- Scripted or hosted applications.
- Windows GUI applications (Windows Forms).
- ASP.NET applications.
- XML Web services.
- Windows services.

For example, the Windows Forms classes are a comprehensive set of reusable types that vastly simplify Windows GUI development.

Active Oberon for .NET

Active Oberon for .net [Gut00a] is an evolution of the programming language Oberon in the context of the new Microsoft .net technology.

Its highlights are:

- an explicit notion of object type including "active objects" with integrated thread of control
- a uniform concept of abstraction called definition representing both "facet" and unit of use
- a module construct simultaneously acting as name–space and singleton object

Definitions are abstractions. More precisely, a definition is an interface, optionally equipped with a state space and predefined method implementations. Definitions can be refined (that is extended in terms of state, functionality or implementation) to new definitions or implemented by object types. Each definition implemented by an object type corresponds to a facet or service unit of the object type exposed to clients.

Programs in Active Oberon for .net are typically structured as populations of active objects or agents communicating with each other via

definitions. Consequently, the Active Oberon programming model integrates seamlessly with the architecture of distributed systems. Also provided in Active Oberon for .net is a generic type OBJECT, optionally followed by a set of postulated definitions. However, there is no (non–trivial) type hierarchy or class hierarchy in Active Oberon for .net. In particular, Oberon's type extension is re-interpreted in Active Oberon for .net as implemented abstraction.

Also new to the language is a block statement construct of the form

```
BEGIN { modifiers } ... ON EXCEPTION ... END;
```

where modifiers is a list of directives like ACTIVE (separate thread), (mutually) EXCLUSIVE, and CONCURRENT, and the optional ON EXCEPTION clause is used to handle any exception occurring within the block statement. In addition, the language features enumeration types along the Pascal/ Modula lines.

1.4 The ETH Oberon System

Oberon is simultaneously the name of a programming language [Wir88] and of a modern operating system. The Oberon project [GW92] was started at the Swiss Federal Institute of Technology in Zürich in 1985 by Niklaus Wirth and Jürg Gutknecht. It was originally targeted towards in–house built hardware (Ceres workstation, based on the National Semiconductors 32000 processor family). Later, the decision was made to port the system to popular computer hardware, where it would run natively or on top of another operating system. Today, Oberon is available for many computer platforms.

In 1991, Jürg Gutknecht and his group continued the development towards the ETH Oberon System. The goal was to exploit the inherent potential and features of Oberon to a much larger degree, upgrade the system by a concept of composable and persistent objects, complement the textual user interface by a graphical companion and provide support for the ubiquitous network. In 1995, the first official Oberon System 3 release was finished. Since then, the system has been constantly improved and extended. In 1997, Release 2.2 including a large palette of applications was published together with a comprehensive hypertext–based documentation [FM98]. In March 2000, a new release

was ready and the system was renamed 'ETH Oberon System'. Some of the ETH Oberon System highlights are:

- **Advanced Textual User Interface**. The basic system layer comes with a highly powerful textual user interface (texts with embedded Oberon commands). It is as lean and compact as the original system.

- **Integrated Object Support in the Kernel**. The ETH Oberon kernel is an upgrade of the original Oberon kernel by a unifying object machinery that

  - integrates and generalizes existing concepts. In particular, all substantial ingredients of the original Oberon system like character, font, text, display frame and viewer can be expressed uniformly in terms of two newly introduced concepts object and object library (an indexed collection of objects).
  - defines a generic message protocol ('software bus') for composite objects and thus provides a universal and extensible platform for future object types.

- **Object Autonomy and Persistence**. All objects are autonomous and persistent by nature, i.e. they are viable in any environment and are portable to any other store or machine together with their current state. The ETH Oberon System supports persistence by a binding mechanism that allows objects to be bound to an object library.

- **Extensibility by Software Bus Technology**. Objects are consistently equipped with generic message interfaces. Based on this genericity, a common message protocol is defined, that can be regarded as a kind of software bus in the sense that objects implementing this protocol can participate and cooperate in system data structures. The protocol is defined by a set of basic message types and rules that can be extended individually for specific classes or groups of objects. A message can be sent directly to a receiver object, or it can be broadcast.

- **Fully Hierarchical Composability**. Objects are components by nature, i.e. they can be freely embedded in any other container object or text. Object composition can be nested up to any arbitrary level.

- **Generalized MVC Scheme**. The display space is a heterogeneous hierarchy of composite and elementary visual objects and it supports arbitrary partial and total overlapping. The hierarchy is based on a generalized Model–View–Controller scheme that allows different camera views of one and the same visual object.

- **Powerful GUI Framework Gadgets**. The Gadgets package [Mar96] provides a very powerful framework for object composition. It includes a rich library of predefined visual objects and model (or abstract) objects, called 'gadgets', and some effective tools for their interactive and descriptive composition and inspection.

  Gadgets object types cover a large spectrum. Most of them are of a visual nature and are seen on the display, that is, they are part of the display space. Examples are text gadgets, viewers, menu bars, buttons, sliders, panels etc. In contrast to visual gadgets, non–visual gadgets operate behind the scenes and are able to manipulate and store information. They are acting as models in the MVC scheme.

  The Gadgets framework allows the run–time construction of graphical user interfaces (GUI) from gadgets as building blocks. Each dialog element, or gadget, can be embedded in any UI (textual or graphic) or application. All gadgets can be integrated and reused in any other ETH Oberon System environment. Gadgets can float in a text, they can be embedded in a panel, in a graphic diagram etc. Container gadgets manage other gadgets as their 'children'. The principal containers are panel gadgets (two–dimensional edit surfaces) and text gadgets (complete text editors with support for embedding), although more refined containers are available.

  Two different methods for object construction and composition are available: interactive and descriptive. The interactive method

makes use of the built–in editing support of objects. Visual objects are modified and used interactively wherever they are located. ETH Oberon users create new UIs or modify existing ones in a typical drag–and–drop fashion. In effect, UI construction is reduced to pure document editing. In addition, two supporting tools are available: the Gadgets tool offers a rich and extensible arsenal of predefined components and some layouting functionality, the Columbus inspector allows gadgets to be inspected together with their attributes and links. (see figure 1.2)

The descriptive method relies on a formal language and a corresponding interpreter. The description language LayLa is of a LISP–like functional style.

- **Document–Oriented Interface**. A concept of generic and self–contained documents is part of the Gadgets framework. Such documents are identified by a name which specifies the access method and path to the document's data. Arbitrary new access methods and document types can be added to the document system, e.g. http, html, new image formats, etc.

- **Extensibility on Different Levels**. Openness and extensibility were key goals of the Oberon project from the beginning. With the ETH Oberon System we can distinguish extensibility on three different levels:

  - The lowest level corresponds to the simplest case. It comprises the creation of composite objects, i.e. of user interfaces and documents from existing components. This level is accessible to programmers as well as to end–users. It merely requires some familiarity with either the interactive composition tools and the inspector or the description language.
  - The next level is programmed use of existing gadgets and GUIs, in particular for adding 'glue logic' to components. This level is supported by a rich procedural interface and in particular by the modules Attributes, Links, GadgetsIn and GadgetsOut. No object–oriented programming is required on this level.

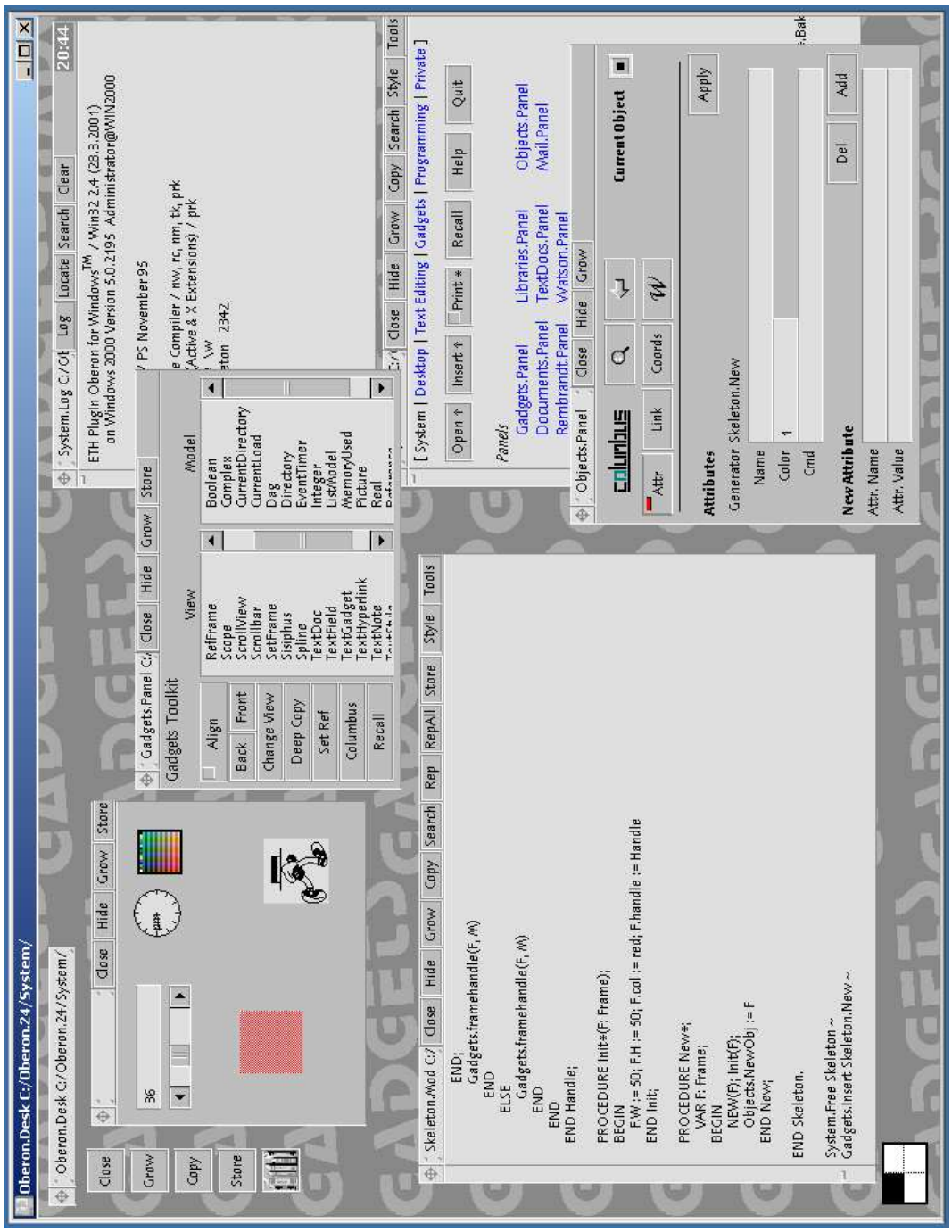

Figure 1.2: Gadgets toolkit and Columbus object inspector on the Oberon desktop.

– The third level involves developing new components. Here, one should distinguish between two kinds of components: elementary gadgets that do not contain any further gadgets and container gadgets that are able to manage other gadgets. In this context, 'developing' means extending the type and message handler code of an existing gadget or defining a new type and programming a new message handler. To support this activity, the ETH Oberon System release provides code skeletons, in other words, templates that can easily be modified and tailored to individual needs.

The following example demonstrates the complete implementation of an elementary visual gadget. The gadget displays a colored rectangle and has the following features:

- The gadget is persistent (*FileMsg*).
- The gadget can be copied (*CopyMsg*).
- The color can be changed by setting the attribute named 'Color' (*AttrMsg*).

Procedure *Handle* implements the message handler, unknown messages are forwared to the default handler implementation (*Gadgets.-framehandle*). New instances are created by the generator command *New*.

```
MODULE Skeleton; (** portable *)
IMPORT Files, Objects, Display, Display3, Gadgets;

TYPE
  Frame* = POINTER TO RECORD (Gadgets.FrameDesc)
    col: Display.Color
  END;

PROCEDURE Restore(F: Frame; Q: Display3.Mask; x, y, w, h: INTEGER);
BEGIN
  Display3.ReplConst(Q, F.col, x, y, w, h, Display.replace);
  IF Gadgets.selected IN F.state THEN
    Display3.FillPattern(Q, Display3.white, Display3.selectpat,
      x, y, x, y, w, h, Display.paint)
  END
END Restore;
```

```
PROCEDURE Attributes(F: Frame; VAR M: Objects.AttrMsg);
BEGIN
  IF M.id = Objects.get THEN
    IF M.name = "Gen" THEN
      M.class := Objects.String; M.s := "Skeleton.New"; M.res := 0
    ELSIF M.name = "Color" THEN
      M.class := Objects.Int; M.i := F.col; M.res := 0
    ELSE Gadgets.framehandle(F, M)
    END
  ELSIF M.id = Objects.set THEN
    IF (M.name = "Color") & (M.class = Objects.Int) THEN
      F.col := M.i; M.res := 0
    ELSE Gadgets.framehandle(F, M)
    END
  ELSIF M.id = Objects.enum THEN
    Gadgets.framehandle(F, M); M.Enum("Color")
  END
END Attributes;

PROCEDURE Copy*(VAR M: Objects.CopyMsg; from, to: Frame);
BEGIN
  Gadgets.CopyFrame(M, from, to); to.col := from.col
END Copy;

PROCEDURE Handle*(F: Objects.Object; VAR M: Objects.ObjMsg);
  VAR x, y: INTEGER; Q: Display3.Mask; F1: Frame;
BEGIN
  WITH F: Frame DO
    IF M IS Display.FrameMsg THEN
      WITH M: Display.FrameMsg DO
        IF (M.F = NIL) OR (M.F = F) THEN
        (* message addressed to this frame *)
          x := M.x + F.X; y := M.y + F.Y;
          IF M IS Display.DisplayMsg THEN
            WITH M: Display.DisplayMsg DO
              IF M.device = Display.screen THEN
                Gadgets.MakeMask(F, x, y, M.dlink, Q);
                IF (M.F # NIL) & (M.id = Display.area) THEN
                  Display3.AdjustMask(Q, x+M.u, y+F.H-1+M.v,
                    M.w, M.h)
                END;
                Restore(F, Q, x, y, F.W, F.H)
              ELSE Gadgets.framehandle(F, M)
              END
            END
          ELSE Gadgets.framehandle(F, M)
          END
```

```
        END
      END
    ELSIF M IS Objects.AttrMsg THEN
      Attributes(F, M(Objects.AttrMsg))
    ELSIF M IS Objects.CopyMsg THEN
      WITH M: Objects.CopyMsg DO
        IF M.stamp = F.stamp THEN (* non-first arrival *)
          M.obj := F.dlink
        ELSE (* first arrival *)
          NEW(F1); F.stamp := M.stamp; F.dlink := F1;
          Copy(M, F, F1); M.obj := F1
        END
      END
    ELSIF M IS Objects.FileMsg THEN
      WITH M: Objects.FileMsg DO
        Gadgets.framehandle(F, M);
        IF M.id = Objects.store THEN
          Files.WriteLInt(M.R, F.col)
        ELSIF M.id = Objects.load THEN
          Files.ReadLInt(M.R, F.col)
        END
      END
    ELSE Gadgets.framehandle(F, M)
    END
  END
END Handle;

PROCEDURE Init*(F: Frame);
BEGIN
  F.handle := Handle; F.W := 50; F.H := 50;
  F.col := Display.RGB(255, 0, 0)
END Init;

PROCEDURE New*;
  VAR F: Frame;
BEGIN
  NEW(F); Init(F); Objects.NewObj := F
END New;

END Skeleton.
```

1.5 Related Work

1.5.1 Juice / Gazelle

Juice [FK96] is a web applet development system based on Oberon. Key components of Juice are:

- An architecture–neutral software distribution format based on 'Slim Binaries' [FK97].
- An Oberon based framework for developing web applets.
- A plug–in that enables the Netscape and Microsoft Internet Explorer browser to run Juice applets.

The 'Slim Binaries' software distribution format was designed with on–the–fly compilation in mind. So, Juice always compiles each applet into native code of the target machine before it begins execution. This in contrast to Java where applets are normally interpreted. The on–the–fly compilation process is not only exceptionally fast, but it generates object code that is comparable in quality to commercial C compilers. The current distribution supports the Microsoft Windows (Intel) and the Mac OS (68k and PPC) platforms.

The Juice applet framework consists of an ETH Oberon compatible kernel with some additional modules specific to web applets. This framework is not compatible with ETH Oberon's component system Gadgets. For the development of Juice applets a CDK (Cross Development Kit) consisting of authoring tools (compiler, applet file packager) and a Gadgets–based applet viewer are provided for the ETH Oberon System.

Juice applets are inserted into a HTML page using the EMBED tag. This tag specifies the size (WIDTH, HEIGHT) and the applet description file (SRC) to be used for creating the applet. The applet file itself contains the generator command followed by optional parameter definitions. Other resources required for creating and executing the applet are not included in the description file. The Juice module loader requestes object–files from the web server if there exists no local copy of the requested module. Other files required by the applet must be requested explicitly from the server by the applet itself. Juice does not provide any mechanism to automatically download and

install resources. The following sample applet implements the same functionality as the Skeleton gadget in the previous section.

JuiceSkeleton.html:

```
<HTML>
<HEAD><TITLE>Juice Skeleton Applet Test page</TITLE></HEAD>
<BODY>
<EMBED SRC="JuiceSkeleton.adf" WIDTH=50 HEIGHT=50>
</BODY>
</HTML>
```

JuiceSkeleton.adf:

```
JuiceSkeleton.New
  ColorRed=255 ColorGreen=0 ColorBlue=0
```

Juice uses a message handler and generator command similar to Gadgets, but a different message protocol.

JuiceSkeleton.Mod:

```
MODULE JuiceSkeleton;
  IMPORT Applets := JuiceApplets, Devices := JuiceDevices;

  TYPE
    Applet = POINTER TO RECORD (Applets.AppletDesc)
      col: Devices.RGBColor
    END;

  PROCEDURE Init(me: Applet);
    VAR par: Applets.Param;
  BEGIN
    Applets.GetParam(me, "ColorRed", par);
    IF par.class = Applets.Int THEN me.col.red := par.i END;
    Applets.GetParam(me, "ColorGreen", par);
    IF par.class = Applets.Int THEN me.col.green := par.i END;
    Applets.GetParam(me, "ColorBlue", par);
    IF par.class = Applets.Int THEN me.col.blue := par.i END
  END Init;

  PROCEDURE Update(me: Applet);
  BEGIN
    Devices.Setup(me.device);
    Devices.SetForeColor(me.col);
```

```
    Devices.FillRect(0, 0, me.device.w, me.device.h);
    Devices.Restore(me.device)
  END Update;

  PROCEDURE *Handler(me: Applets.Applet; VAR M: Applets.AppletMsg);
  BEGIN
    WITH me: Applet DO
      IF M IS Applets.DisplayMsg THEN
        WITH M: Applets.DisplayMsg DO
          IF (M.id = Applets.update) OR
              (M.id = Applets.resize) THEN
            Update(me)
          ELSE
            Applets.AppletHandler(me, M)
          END
        END
      ELSIF M IS Applets.InitMsg THEN
        Init(me)
      ELSE
        Applets.AppletHandler(me, M)
      END
    END
  END Handler;

  PROCEDURE New*;
    VAR a: Applet;
  BEGIN
    NEW(a); a.handle := Handler;
    a.col.red := 255; a.col.green := 0; a.col.blue := 0;
    Applets.newApplet := a
  END New;

END JuiceSkeleton.
```

Gazelle [Paz97] is an implementation of both the Juice CDK and run–time system using the Oberon/F environment. This package is provided by MicroWorks LLC as a commercial Internet Development Framework package.

1.5.2 BlackBox Component Builder

BlackBox Component Builder [Obe97] is an integrated visual development system for the rapid development of software components. It is the successor to Oberon/F, with many new features. It uses the programming language Component Pascal, a refined version of Oberon–2

[WM91] optimized for component software development. BlackBox Component Builder is available for Windows and Mac OS.

The BlackBox Component Builder includes a component framework with a compound document architecture. On Windows, the framework and document architecture is integrated into OLE (object linking and embedding). The highlights of the component framework are: extensible text and text processing classes, a visual form designer, and an open database access architecture.

The Direct–to–COM (DTC) compiler is a Component Pascal compiler that directly supports Microsoft's Component Object Model (COM). With the DTC compiler, objects are declared and implemented in a superset of the Component Pascal language. The compiler implements the COM reference counting methods *AddRef* and *Release* automatically for all COM objects.

The following sample component implements the same functionality as the Skeleton gadget in the previous section.

```
MODULE ObxSkeleton;
IMPORT Views, Ports, Properties;

TYPE
  View = POINTER TO RECORD (Views.View)
    c: Ports.Color
  END;

PROCEDURE (v: View) Restore(f: Views.Frame; l, t, r, b: INTEGER);
BEGIN
  f.DrawRect(l, t, r, b, Ports.fill, v.c)
END Restore;

PROCEDURE (v: View) HandlePropMsg(VAR msg: Properties.Message);
  VAR stdProp: Properties.StdProp; prop: Properties.Property;
BEGIN
  WITH msg: Properties.SizePref DO
    IF (msg.w = Views.undefined) OR (msg.h = Views.undefined) THEN
       msg.w := 50*Ports.point; msg.h := 50*Ports.point
    END
  |msg: Properties.PollMsg DO
    NEW(stdProp);
    stdProp.color.val := v.c;
    stdProp.valid := {Properties.color};
    stdProp.known := {Properties.color};
    Properties.Insert(msg.prop, stdProp)
  |msg: Properties.SetMsg DO
```

```
    prop := msg.prop;
    WHILE prop # NIL DO
      WITH prop: Properties.StdProp DO
        IF Properties.color IN prop.valid THEN
          v.c := prop.color.val
        END
      ELSE
      END;
      prop := prop.next
    END;
    Views.Update(v, Views.keepFrames)
  ELSE  (* ignore other messages *)
  END
END HandlePropMsg;

PROCEDURE Deposit*;
  VAR v: View;
BEGIN
  NEW(v); v.c := Ports.red; Views.Deposit(v)
END Deposit;

END ObxSkeleton.

"ObxSkeleton.Deposit; StdCmds.PasteView"
```

1.5.3 Java Beans Tools for ActiveX

Java Beans Architecture Bridge for ActiveX

Java Beans provides a packager tool [Sun99a] which automatically converts a Bean into an ActiveX component. This packager creates for each converted Bean a registry file for registering the component with COM and a TypeLib file which describes the components properties, events and methods. The ActiveX server is implemented as a generic bridge that can handle any Bean together with the COM information generated by the packager tool.

Java Beans Architecture Migration Assistant for ActiveX

The Java Beans architecture Migration Assistant [Sun99b] is a tool that generates Java Beans specification code from ActiveX controls. The tool analyzes the properties of an ActiveX control and creates a framework for a subsequent product meeting the 100% Pure Java certification standards that takes on the features of the ActiveX control.

1.5.4 VMWare, Virtual PC

Products like *VMWare* [VMW01], or *Virtual PC* [Con01] provide a complete PC environment implemented in software. This allows a user to simultaneously use different virtual PC environments (guest OSes) on a single physical PC (host OS). Each virtual machine operates as if it were a stand–alone machine with its own sound card, video board, network adapter, and processor.

To the host system this virtual machine software looks like an ordinary application software. Thus all the hardware components of the virtual machine are emulated by using the API of the host OS. For example a virtual video board uses the graphics programming interface of the host system. To allow efficient operation of such an emulation just–in–time (JIT) compilation and bridge drivers are used.

JIT compilation is used to translate code between incompatible host and guest machine CPUs, and to replace privileged instructions by calls to the corresponding virtual machine implementations. For example *Virtual PC* uses JIT technology to efficiently emulate an x86 system on a PowerPC.

Bridge drivers are used to call APIs of the host OS directly without the need of a complete software emulation of a hardware component. For example if a Windows system is running as guest OS on top of another Windows system; the guest OS can use a special graphics library, which calls the corresponding routines in the host OS's graphics library, without the need of emulating an virtual video adapter.

Using virtual machine software not only makes it possible to combine several individual systems running different OSes into a single system; but the virtual machine software enables new features, which are not feasible when using individual systems.

A virtual machine allows users to containerize a virtual machine's hard drive into a single file on the host system. This feature allows for easy portability, backup, and recovery and provides an elegant mechanism for deploying pre–configured systems.

Another benefit of running multiple operating systems on a single host system are special integration features. Copy & paste allows the user to transfer text, and graphics between different virtual machines and the host environment. Another integration feature is drag & drop, it allows the user to drag files between the virtualized guest environ-

ment and the host environment or vice versa. Yet another variant of integration is the use of an virtual intra–net connecting the guest environments with the host environment, using routing technologies like NAT (network address translation).

Chapter 2

Emulating a System on Top of Another

The Oberon System [GW92] has originally been designed and implemented as native operating system for the Ceres workstation [Ebe87], but has since beend ported to a number of other machine architectures. Ports have been implemented both as native operating system (Native Oberon for PC, ...), and as an operating system emulation running on top of another (MacOberon, Oberon for Windows, ...).

As the Oberon System module hierarchy contains only few hardware dependent modules it is relativly simple to port the Oberon System to a new enironment. Specially when implementing the Oberon System on top of another operating system, the low–level modules must be designed very carefully to get both a well behaving application in the host environment, and a fully compatible Oberon implementation (reference platform for ETH Oberon is Native Oberon for PC [Mul02]).

An application can be called well behaving, when it does well cooperate with other applications, specially concerning the sharing of limited system resources like: CPU time, memory, and system object handles (files, graphics, ...). A second point where an application can be well behaving, is the conformance with application guidelines (e.g. UI) as defined by the host system's designers.

A fully compatible Oberon implementation on the other hand should

adhere to the look & and feel of a native Oberon implementation and provide full compatibility with all existing and new Oberon applications.

The following sections provide an overview on the problems and solutions typically found when embedding a framework or operating system in a new environment. Although these sections reflect the actual problems and solutions found when implementing Oberon for Windows, most statements made can easily be adapted to the implementation of other systems like: Jave, Virtual PC, etc.

2.1 Bootstrapping the System

A prerequisite for the implementation of a system or framework like Oberon on a new platform is the availability of a compiler for the new target architecture. As there have allready been different Oberon implementations for x86 based platforms, the existing compiler from Native Oberon for PC has been used. When implementing a new compiler, a portable compiler framework like OP2 [Cre90] greatly simplifies the task.

For each new platform a specific runtime library must be provided. Typically parts of this library have to be programmed with tools (SDK, C compiler, ...) provided by the manufacturer of the target platform. These problems are discussed in more details in chapter 4.

An alternative approach for bootstrapping a framework (progamming language), is the use of an interpreter. Typicall examples are the Pascal p–code interpreter [PD82, JW91], and the Java VM [LY99].

2.2 Automatic Memory Management

An emulated system must allocated its own heap on the heap of the host system. The heap may be allocated as one big contiguous memory block, or as individual heap blocks as needed.

Using one big block has the advantage, that one contiguous range of addresses can be used for memory managment functions like garbage collection. There are however two big disadvantages with using one big block. As the system allocates one big block befor actually needing all of it, memory possibly needed by other applications or the system

(file buffers) is wasted. A second problem is, that the size of the block can't be changed at runtime, thus the emulator might stop with an out of memory exception.

Allocating individual blocks from the host systems heap solves the problem of wasting memory. However this approach is rather inefficent when using small block sizes, as the hosts systems memory allocation scheme is optimized for allocating memory pages (4 kBytes). Another problem is, that this scheme does not allow the efficient implementation of a mark and sweep garbage collector. The Oberon implementation uses a mix of the two methods: small blocks are (sub–) allocated from a pool of medium sized heap blocks (256 kbytes), big blocks (> 256 kbytes) are directly allocated.

Another memory managment releated problem is, that an emulators garbage collector might cause 'trashing' [JL96, Mil02] when running on top of a demand paging system. 'Trashing' is caused by a mark and sweep collector in low–memory situations, as all active cells are visited in the mark phase, and all cells are examined by the sweep phase. The current Oberon mark and sweep implementation does ignore this problem; since any modern PC is equipped with more than an order of magnitude more memory (e.g. 256 MB) than a typical Oberon environment will need (e.g. 8 MB).

2.3 IO and CPU Utilization

A second critical resource which must be shared between the embedded system and the host system is the CPU. Operating systems are normally optimized to take full advantage of the CPU power available. One technique used to achieve this, are idle time tasks. These are applications which run when all regular processes are blocked waiting for some resource to become available. Examples of idle time tasks are power saving actions like slowing down the CPU or screen savers or long running background calculations like SETI. Obviously such idle time activities should not be run by an emulated system; as other applications running on the same host system might have some more important work to be done. In the Oberon system the garbage collector is run as idle task every 100 seconds. This alone does certainly not result in a high CPU utilization. However in a low memory situation

running the GC at a fixed intervall might result in some 'trashing'.

Another problem with sharing the CPU are systems which use a simple polling model for keyboard and mouse input. Typicall 'polling applications' are interactive DOS applications or the Oberon system. An operating systems which provides a DOS emulation will handle the keyboard and mouse input releated software interrupts to minimize the CPU load. The implementation of the Oberon (input) loop has been adapted to the event model of the host system. More details on this approach are given in 3.2.

2.4 Filesystem

An entire class of hardware resources to be shared are the different storage drives found in a PC. Typicall examples are floppy drives, harddrives, CD–ROM drives, etc. For removable devices like floppies a simple sharing strategy is to assign the affected device exclusively to the application currently using it. This approach gives the application full control over the removable device. For example an operating system emulator might use a file system not supported by the host system.

For harddisks this simplistic 'exclusive access' solution is not feasible, as the harddisk will be used concurrently by both the guest system and the host system. There are however other solutions, which use the host systems file system in different forms.

The most efficient solution is to use a separate partition for the emulators file system. The emulator still must cooperate with the host operating system at the level of the disk driver, but is not affected by any restrictions of the hosts OS's file system. For example an emulated Linux system running on a Windows host system might access Unix specific partitions. The downside of this approach is that a new partition for the guest OS must be created. This usually involves resizeing of existing partitions, a proccess which requires special tools like *Partition Magic* [Pow02]. Another problem is that files can't be easily exchanged between the guest and the host.

A second possibility is to use an ordinary host file to emulate a partition. Thus sectors in the guest OS's partition are not directly mapped to physical sectors on the disk, but are emulated by reading

and writing blocks to the file. This approach has the advantage, that emulator partitions can be easily created and resized. It is even possible to provide features which are not feasible using ordinary partitions. For example as the guest partitions are just ordinary host files they can be used to redistribute preconfigured environments.

Yet another method for implementing a guest OS's file system is to map all the file systems calls to corresponding calls of the hosts OS's API. This method is used by most Oberon implementations. One problem with this approach is, that the host file system might not provide a feature required by the guest system. For example not all file system do allow long file names (e.g. 8.3 names using FAT). A solution to this problem is to store the extra information required in a file header preceding the actual file content.

2.5 Display System

A very important aspect of every modern operating system is it's graphical user interface. The graphical user interface consists of two subsystems: the graphics system and the user interface.

The graphics system has two interfaces. At the hardware level there is a driver interface to the display adapter and at the programming level there is an graphics programming interface. Operating systems like Windows or Mac OS, implement a complex framework between this two interfaces. For example on Mac OS X the programming interface is PDF (*Adobe Portable Document Format*). An emulated system can either implement it's low–level driver interface or the programming interface on top of the graphics API of the host. Although emulating a display adapter at a low–level (hardware) is very slow, there are applications which require this kind of emulation (e.g. graphical DOS applications). Another problem with emulating a graphics interface on top of another is, that the two systems might use completely different paradigms. Oberon's graphics system for example is based on a small set of simple raster operations (module *Display*). This makes it impossible to implement fast graphics on top of another graphics API. To get reasonable fast display output, some of the otherwise portable graphics modules must be replaced by host system specific variants. For example text output on Windows is much faster using the *GDI32.TextOut*

routine, than calling *GDI32.BitBlt* for each single character in a string [Fra93].

The user interface of an emulated system can be embedded into the host environment in two different ways. The emulators user interface can either be implemented as single–window (or full-screen) application, or as an application integrated with the hosts window manager.

The single–window approach is a good solution if one want's to maintain the native look & feel of the emulated systems. This includes exclusively capturing the mouse and keyboard, when the emulator runs in the foreground. The downside of this solution is, that the emulator does not integrate well in the host environment.

When implementing a better integrated version, more parts of the emulated system must be adapted to the host enironment. However it is still possible to provide a fully compliant emulation. In chapter 3 such an implementation for the ETH Oberon system is presented.

Chapter 3

A new Oberon Display System

3.1 Introduction

In the original implementation of Oberon for Windows one fixed size window is used for display output. Within this one window, the non–overlapping tiled viewers system is used as user interface. On one hand this approach provides a good emulation of a Native Oberon look–and–feel, but on the other hand this approach does not fit well into the desktop model of Windows. This chapter presents extensions to Oberon's viewer and display system, which allows for a better visual integration of Oberon, and still provides full compatibility with existing applications.

The extensions are based on two new concepts. *Displays* are abstract drawing contexts which can be used for graphics output to a wide range of different devices. Possible implementations of displays are: application windows, control windows, offscreen bitmaps, etc. *Events* are used to decouple the Oberon loop from the Windows thread message queue. Any instance within an application using an Oberon subsystem can produce such events to communicate with Oberon. Thus by introducing only a minimal set of new concepts, Oberon can be used in a much more general way.

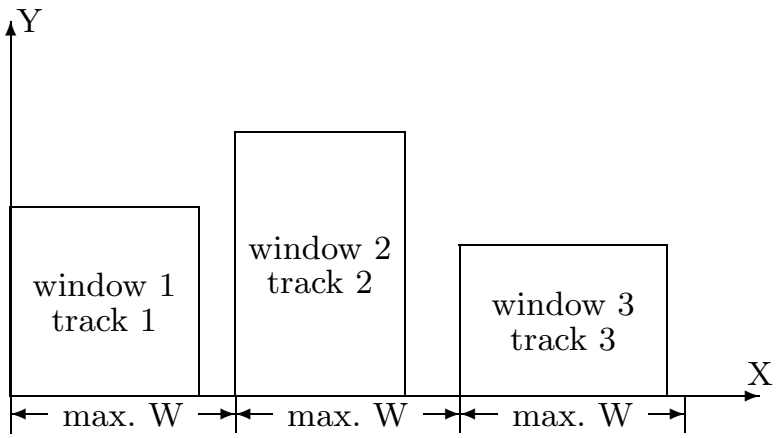

Figure 3.1: Multi–windowed Oberon using global coordinates.

3.2 A new Display Scheme

In [AB94] an implementation of a multi–windowed Oberon system using a global coordinate system is discussed. The system works by arranging all windows along the X–axis of the Oberon coordinate system. The windows are placed at offsets which are multiples of the maximum window width (1280 pixels), as shown in figure 3.1. Using this approach, coordinates can easily be transformed from the Oberon coordinate system to the local coordinate system of the individual window containing a given X coordinate. Each such window contains one Oberon track. Inside the track viewers are arranged in a tiled fashion like in the normal Oberon display layout. This model fits well into the existing Oberon system, in which only modules *Display*, *Viewers*, *Oberon* and *System* must be adapted. To the module *Display*, a table of all available windows together with their offset in the Oberon display space must be added. The module *Viewers*, together with modules *Oberon* and *System*, must provide new commands to manage tracks in windows of their own. The drawbacks of this method are, that the number and the size of the windows is limited, and that there is no support for mapping components smaller than entire tracks to windows. When using *Gadgets*, it would be much more desirable to map documents or even individual frames to a window.

For the reasons mentioned above, a completely new approach has been chosen for the implementation of a multi–windowed Oberon sys-

tem. This new approach exploits the fact that frame messages like *display*, *modify*, and *input* are always broadcast to the destination frame according to the parental control principle. In this way, a parent frame can easily set up a graphic context for its children when processing frame messages. The destination frame can then draw into the already established graphics context without the need for context switches or coordinate transformations in every graphics primitive. This approach is much more general than the solution presented in [AB94]. Thus, not only tracks, but also frames can easily be displayed in a foreign window context. This is achieved by implementing a specialized display class for the different types of window contexts.

The multi–windowed implementation is based on a new module *Displays*, introducing two new data types: *Display* and *Event*. Instances of *Display* encapsulate a graphics context together with clipping information and cached resources [Fra93] like fonts, brushes, etc. Module *Display* maintains a global pointer to the current display context to be used for output. This current context is setup by module *Viewers* during broadcast, and is used for subsequent display output until it is reset to another display context by a new broadcast. The following excerpt of *Displays.Def* shows the definition of type *Display*.

```
Display = POINTER TO RECORD (Objects.Object)
  hWnd: User32.HWND;  (* handle of this window *)
  hWndParent: User32.HWND;  (* handle of the parent window *)
  hDC: User32.HDC;  (* the display device context *)
  width, height: LONGINT;  (* current size *)
  clipL, clipR, clipT, clipB: LONGINT;  (* clipping rectangle *)
  link: Display (* list of all displays *)
END;
```

Type *Display* is an extension of *Objects.Object*, and is therefore equipped with a standard Oberon message handler. Thus displays can be controlled in a generic way using standard object messages like: *Objects.AttrMsg*, *Display.ModifyMsg*, *Display.ControlMsg*, and so on. Using standard object messages a viewer embedded inside a display can control its surrounding display without knowing its exact type or even its implementation. For instance, the title of a document window can be set by setting the 'Title' attribute. Internally, most display implementations have a second message handler (or window procedure), which is responsible for handling messages sent to the display

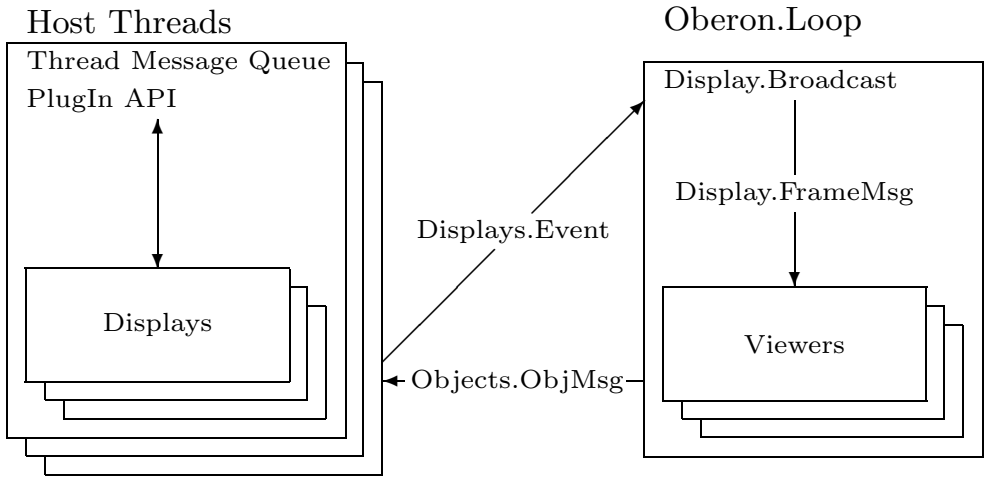

Figure 3.2: Inter–thread communication.

(or window) by the host system. This includes messages emanating from mouse and keyboard input, window movement and resizing, etc. Since these messages are typically sent to the handler from the thread dispatching the Windows thread message queue, there must be a mechanism for thread–safe communication with the 'Oberon Loop'. For this purpose, event objects are introduced in module *Displays*.

Event objects are used for thread–safe communication between — possibly different — threads generating display–related events with the 'Oberon Loop'. Event objects are extensions of the base type *Displays.Event* and are put into a FIFO queue by event producers (any thread) and are fetched from the queue by the single event consumer 'Oberon Loop', see figure 3.2. The following excerpt of *Displays.Def* shows the definitions of the mouse and keyboard input events.

```
CONST
(* Event ids: *)
  consume = 9; track = 10; (* keyboard or mouse data available *)
  execute = 12; (* request to execute a command *)

TYPE
  (* Event object *)
  Event = POINTER TO RECORD
    disp: Display;  (* target display, NIL for broadcast *)
    id: LONGINT;
    done: Threads.Event
    (* done # NIL:
      call Threads.Set(event.done) when event done *)
```

```
  END;

  InputEvent = POINTER TO RECORD (Event) (* id: consume, track *)
    keys: SET;  (* mouse buttons, numbered from right to left *)
    X, Y: LONGINT;  (* mouse position (relative to display) *)
    ch: CHAR  (* character typed *)
  END;

  CommandEvent* = POINTER TO RECORD (Event) (* id: execute *)
    cmd: FileDir.FileName; (* command string to be executed *)
    executor: Objects.Object
  END;

VAR
(* this event is set when new event objects (nEvents > 0)
  get available *)
newEvents: POINTER TO RECORD (Threads.Event)
    nEvents: LONGINT
END;

(* read the next event from the event queue *)
PROCEDURE GetEvent(): Event;

(* put a new event into the event queue, returns TRUE if the
  event was dispatched within timeOut milliseconds *)
PROCEDURE PutEvent(event: Event; disp: Display; id: LONGINT;
  timeOut: LONGINT): BOOLEAN;
```

Using event objects, the 'Oberon Loop' no longer has to poll the message queue but it can be suspended using a blocking wait operation until new event objects become available. In case there are background tasks to be handled, a timeout value for the blocking wait can easily be calculated. A second advantage of using event objects is that additional event types can easily be introduced by defining new event identifiers. Thus, special character codes are no longer needed. A command event type is provided to execute Oberon commands. And it is used to implement shortcut keys supported by module *Oberon*: *Setup*, *Neutralize* and *Update*. The code below shows an excerpt from the 'Oberon Loop' rewritten to use the new module *Displays*.

```
LOOP
  event := Displays.GetEvent();
  WHILE event # NIL DO
    CASE event.id OF
      |Displays.consume: broadcast consume message
```

```
      |Displays.track: broadcast track message
      |Displays.execute: execute command string
    (* handle all other event ids *)
    END;
    IF event.done # NIL THEN Threads.Set(event.done) END;
    event := Displays.GetEvent()
  END;
  lastTask := NextTask;
  WHILE Displays.newEvents.nEvents = 0 DO
    time := Kernel32.GetTickCount();
    CurTask := NextTask; NextTask := CurTask.next;
    IF (CurTask.time-time) <= 0 THEN
      CurTask.handle(CurTask); nextTaskTime := CurTask.time;
      lastTask := CurTask; CurTask := NIL
    ELSIF (CurTask.time-nextTaskTime) < 0 THEN
      nextTaskTime := CurTask.time
    ELSIF NextTask = lastTask THEN
      IF (time-nextTaskTime) < 0 THEN
        Threads.Wait(Displays.newEvents, nextTaskTime-time)
      ELSE
        nextTaskTime := CurTask.time
      END
    END
  END;
  CurTask := NIL
END
```

In the above code, we can see that events are dispatched from the event queue in chronological order. This is in contrast to the standard implementation of the 'Oberon Loop', where the event ordering is ignored (see section 4.5). The most important difference however is, that there is no longer a direct polling of the Windows thread message queue. This brings forward two advantages:

1. Any thread within the Oberon process can produce events, even if there is no thread message queue attached to the producer thread. This is required to support (lightweight) windowless controls (ActiveX) and applets (Netscape).

2. There are now enough free slots in the Oberon ASCII table to implement both the ISO–8859–1 encoding and Oberon's own encoding. Special characters are no longer needed to encode events like neutralize or setup.

3.3 A new Desktop Model

Module *Viewers* has been extended to support the new multi–display possibilities. For this purpose, a new type *Window* was introduced. A window is an instance of a display containing one or more viewers. Module *Viewers* supports three different types of window and viewer combinations:

1. A window containing classical tiled viewers.
2. Window containing a document viewer (menu and main frame).
3. Window containing a view on a single frame (typically non–document Gadgets).

Type *Viewer* has been extended by two new fields *kind* and *win*. These fields are needed by the broadcast to set up the correct display context and by procedures changing the state of viewers (e.g. *Close*, *Grow*, *Hide*, etc.). Procedures for old style viewers (*IsTrack*, *IsFiller*, *IsViewer*) are implemented in module *Viewers*, whereas for new style viewers (*IsDocument*, *IsControl*) an upcall using the object handler of *win* is used. The following excerpt of *Viewers.Def* shows the extended definition of type *Viewer*.

```
CONST
(* the different types (kind) of viewers: *)
  IsDisplay = 0; IsTrack = 1; IsFiller = 2; IsViewer = 3;
  IsDocument = 4; IsControl = 5;

TYPE
  Viewer = POINTER TO ViewerDesc;
  ViewerDesc* = RECORD (Display.FrameDesc)
    state: INTEGER;
    (* > 1: displayed, 1: filler, 0: closed, < 0: suspended. *)
    kind: INTEGER;
    (* IsDisplay, IsTrack, IsFiller, IsViewer,
       IsDocument, IsControl *)
    win: Displays.Display
  END;

  Window = POINTER TO RECORD (Displays.Display)
    viewer: Viewer;
    track: Displays.InputEvent
  END;
```

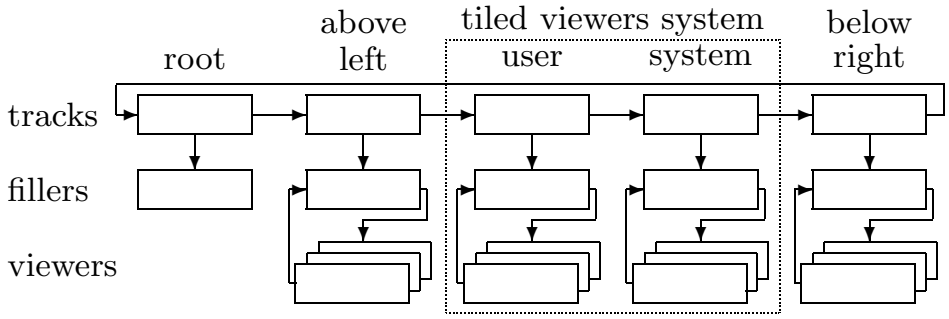

Figure 3.3: Extended track arrangement in the display space.

```
(* default handler for document viewers *)
PROCEDURE DocHandler*(V: Objects.Object; VAR M: Objects.ObjMsg);

(* default handler for control viewers *)
PROCEDURE CtrlHandler*(V: Objects.Object; VAR M: Objects.ObjMsg);
```

All tiled viewers share a common window, corresponding to the single fixed size window of the classical Oberon implementations. The new non–tiled viewers are integrated into the display space [GW92] by introducing two new tracks *left* and *right*. Whereas viewers in the *left* track are above (overlapping) viewers in the tiled viewers system, viewers in the *right* track are below viewers in the tiled viewers system. The extended structure of the display space is illustrated in figure 3.3. When the overlapping of non–tiled viewers (or their windows) changes, then the affected viewers are moved from the left track to the right track or from the right track to the left track depending on their new placement relative to the tiled viewers system window. Tiled viewers always remain in their tracks. Using this track arrangment of the display space, a broadcast is guaranteed to traverse the tracks and windows from top to bottom. This is important for some messages assuming a global coordinate system to work correctly, e.g. *LocateMsg*.

Each of the three different viewer types (document, control, tiled viewer) requires a specific implementation of a viewer class. For tiled viewers there are the already existing implementations like: *MenuViewers.Viewer*, *Desktops.DocViewer*, etc. Module *Viewers* provides default implementations of handlers for document and control viewers. Module *Desktops* has been extended to implement a new document

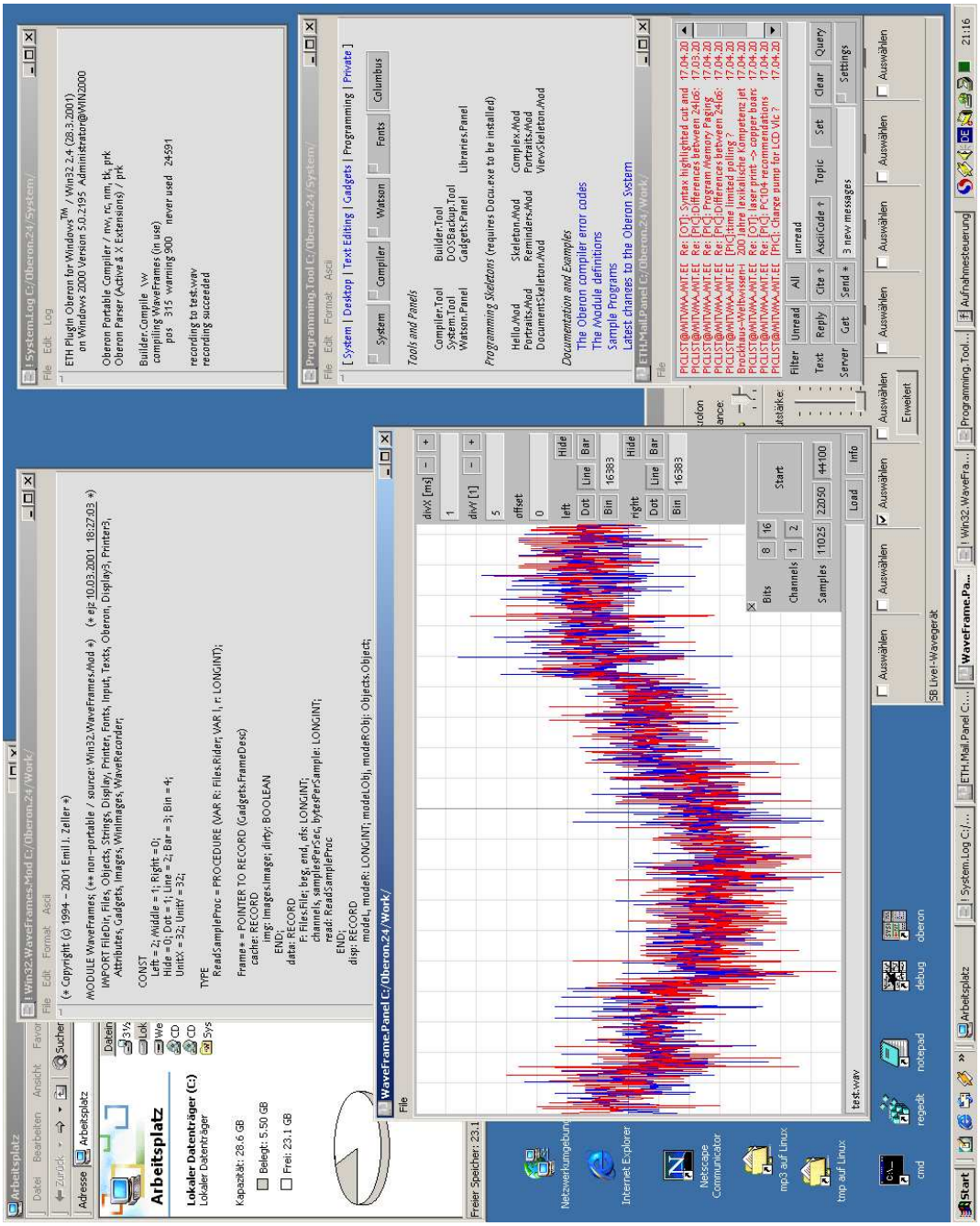

Figure 3.4: Oberon documents on the Windows desktop.

viewer class, embedding documents in overlapped windows (*Viewers.Window*) as provided by the host system. For the Windows platform, module *Windows* implements windowed displays. The two new commands *Desktops.OpenWinDoc* and *Desktops.OpenWin* are provided to open documents and desktops in a window on their own (see figure 3.4). The syntax of these new commands is the same as that of their counterparts *Desktops.OpenDoc* and *Desktops.Open*. The command *Desktops.OpenDoc* opens documents depending on the context in which the command was called. It opens:

- a new viewer in the viewer system
- a new desktop document gadget on the Oberon desktop
- a new windowed document on the Windows desktop

The implementation of a new control viewer class is discussed in sections 6.2 and 7.4.

3.4 Documents and Applications

When Oberon is used on the Windows desktop it seems as each Oberon document was an Windows application on it's own. But actually all the Oberon documents shown in figure 3.4 share one instance of the Oberon run–time system. This is the normal mode of operation in the Oberon system, as there is no concept of applications running in separate processes. The composition of an instance of the Oberon run–time system is closely related with the definition of an application or process in the Windows operating system. The Microsoft Platform SDK [Mic00] defines a process as follows:

A process is a collection of virtual memory space, code, data, and system resources. A thread is code that is to be serially executed within a process. A processor executes threads, not processes, so each application has at least one process, and a process always has at least one thread of execution, known as the primary thread. A process can have multiple threads in addition to the primary thread.

An instance of an Oberon run–time system is created, when Oberon is either started as a process on its own or as a plug–in running inside the process of a host application. In such a setup all Oberon objects or

documents share the heap, modules, and files. This allows for very efficient communication, as objects of arbitrary complexity can be shared between documents directly. Thus there is no need to copy complex data structures or even store them to files.

When Oberon objects are plugged into a host application — like a word document, or web-page — all objects inside the same host environment share the same Oberon run–time system. Thus this objects can be used much like in a normal Oberon document. When Oberon is used by different processes, each process runs it's own instance of the run–time system. In the latter case, standard mechanisms for interprocess communication like the clipboard, files, etc. must be used.

3.5 Using standard Menus and Dialogs

The new document viewers can be moved, resized, minimized, maximized and closed just like normal Windows windows. All these features are implemented by the new document viewer class and there is no need for changing the implementations of the document (main) frame and menu frame inside the document viewer to work with the new display model. But this solution is rather incomplete since in this way the border of documents behaves like a Windows application window and the contents still uses Oberon–style menus with buttons and a three button mouse. Module *WinMenus* provides commands and wrapper objects to use standard drop–down menus, context menus and standard dialogs in Oberon document implementations. These additional features can be used with existing documents simply by updating the menu libraries with new menu item objects. Module *Windows* provides an improved support for two–button mice and a generic mechanism to map shortcut key combinations to arbitrary Oberon commands (e.g. Ctrl–C for Clibpoard.Copy). Note that all these features are completely transparent to all existing document and frame implementations.

Module *WinMenus* implements frame object wrappers which encapsulate Windows drop–down menu items. Menu frame objects can be used much like normal menu buttons: their caption and command can be set using the attribute message, they can be stored in object libraries and they can be copied with the copy message. Although the menu frame objects are an extension of *Display.Frame*, they do not

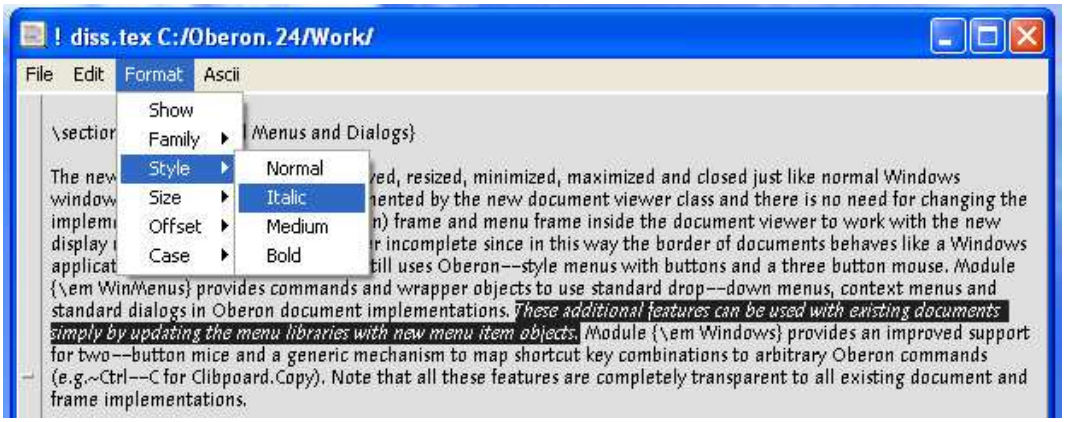

Figure 3.5: Oberon text editor with WinMenus.

have a visual representation in Oberon. Menu frames install themselves in the window of the current (surrounding) viewer when receiving a display message for the first time. Since menu frame objects do not have a visual representation in Oberon, menus can not be composed visually. For the creation of new menus module *WinMenus* provides a simple programming interface.

```
(* create a new empty menu bar *)
PROCEDURE NewMenu(): Popup;

(* append a a sub-menu to menu *)
PROCEDURE AppendMenu(menu: Popup; caption: ARRAY OF CHAR): Popup;

(* append a new leaf item to menu *)
PROCEDURE AppendItem(menu: Popup; caption, cmd: ARRAY OF CHAR);
```

Using the above programming interface the *File* entry in the standard document menu can be programmed as follows:

```
VAR bar, file, fnew: Popup;

bar := NewMenu();
file := AppendMenu(bar, "&File");
fnew := AppendMenu(file, "&New");
AppendItem(fnew, "Text", "Desktops.OpenDoc (TextDocs.NewDoc)");
AppendItem(fnew, "Panel", "Desktops.OpenDoc (PanelDocs.NewDoc)");
AppendItem(fnew, "Pict", "Desktops.OpenDoc (RembrandtDocs.NewDoc)");
```

```
AppendItem(fnew, "Log", "Desktops.OpenDoc (TextDocs.NewLog)");
AppendItem(fnew, "Columbus", "Desktops.OpenDoc (Columbus.NewDoc)");
AppendItem(file, "&Open...", "WinMenus.Open");
AppendItem(file, "&Save", "Desktops.StoreDoc");
AppendItem(file, "Save &As...", "WinMenus.SaveAs");
AppendItem(file, "---", "");
AppendItem(file, "Page Set&up...", "WinPrinter.Setup");
AppendItem(file, "&Print...", "Desktops.PrintDoc Default");
AppendItem(file, "---", "");
AppendItem(file, "Copy", "Desktops.Copy");
AppendItem(file, "Name", "WinMenus.DocName");
AppendItem(file, "---", "");
AppendItem(file, "Close", "Desktops.CloseDoc")
```

In situations where an Oberon viewer does not have control over its parents (window) menu bar, e.g. in a control window, menus may as well be used as local popup–menus (context menus).

A second feature where Oberon document viewers can be presented in a more Windows–like GUI are standard dialogs. Module *WinMenus* provides commands to load and store documents using the 'Open' and 'Save As' standard dialogs. They can easily be provided in a transparent way since the dialogs are only used to choose the file name of a document, the actual loading and storing of the document is done by the document itself. The Windows printer driver *WinPrinter* has been upgraded to use the standard dialogs provided by Windows for page setup and printing.

The handling of the mouse is another GUI feature where Oberon differs radically from more common user interfaces like Windows or Macintosh. The three–button mouse interface of Oberon is a very powerful feature for the expert user, a beginner however usually experiences difficulties using the rather uncommon three–button mouse interface. Most standard PCs and laptops are usually equipped with a two–button mouse only. On such computers the third mouse button is emulated using the Ctrl–key. This emulation scheme is rather inconvenient and even interferes with standard keyboard short–cuts. Module *Windows* implements a more complex emulation scheme for the missing middle mouse button. The left mouse button is interpreted as either middle or left button depending on the type of frame under the mouse cursor; that is, a left button clicking is interpreted as middle click and controls the execution of the command (attribute) associated to the button. A left button clicking in a textfield is interpreted as a

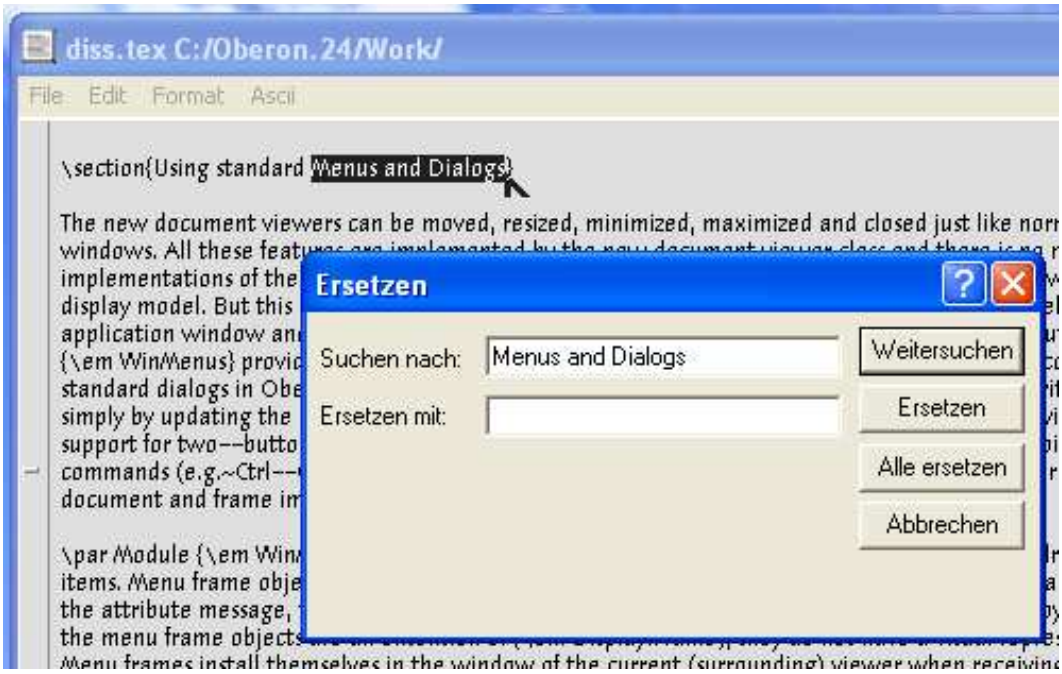

Figure 3.6: Oberon text editor with "Find & Replace" dialog..

left click and sets the caret at the mouse focus. The mapping of the physical left mouse button to a virtual left or middle Oberon mouse button is stored as as lookup table in Oberon's system configuration text (*Oberon.Text*). This table contains for each (frame) generator command the suitable mapping of the physical left mouse button. If there is no mapping defined for a given generator, a middle mouse button is emulated by default. By holding down the Ctrl–key while clicking the left mouse button, the mapping can be inverted. This is needed for more complex frames which use both the left and middle mouse button. For example, to activate a text hyperlink or a command the middle mouse button is used, to set the caret the left button is used. Note that this does not interfere with keyboard short–cuts, since the action is triggered by the mouse button and 'Ctrl' acts only as modifier. This button mapping scheme works well if no interclicks are used. Interclicks are mouse interactions, where two or even all three mouse buttons are involved. To use interclicks a third button or key is needed, the 'Windows context menu' key is used to replace the missing third button in such cases.

Note, that only recently some of the standard Gadgets have been updated to directly support two–button and three–button mice. For these Gadgets, the mapping described above can be disabled.

Chapter 4

A Plug–in Kernel for Oberon

4.1 Introduction

The first version of Oberon (Oberon System V2) for Windows was implemented by Matthias Hausner in 1991. In the following years, Hannes Marais upgraded this version to Oberon System 3. This implementation was named 'Spirit of Oberon' and led later on to the development of 'MacOberon System 3' and 'Oberon System 3 for Linux'. Although this version (Spirit of Oberon for Windows) provides a fully working System 3, there are several limitations which make it less useful for more complex applications. These limitations are:

- boot–loader written in C
- fixed–size heap
- short file names (FAT)
- error–prone interfacing to the Win32 API
- no exception handling
- high CPU load

- one fixed–size window only
- etc.

In the following sections, different aspects of the design and implementation of a new Oberon kernel for Windows are discussed. This kernel not only solves the problems listed above but also provides additional new features. The most important innovations are:

- The kernel can be used either as stand–alone application (EXE), or as dynamic link library (DLL) running within a non–Oberon application.
- A new display system allowing a closer integration of Oberon viewers with the windowing system of the underlying OS.

4.2 Bootlinker and Bootloader

Since Oberon uses its own proprietary object file format, a tool is required to boot a minimal Oberon system on Windows. This minimal set of modules is called 'inner–core' and consists of all the modules required for Oberon's own module loader (module *Modules*) to work. Normally Oberon modules are loaded and linked dynamically when they are needed, the inner–core modules however must be statically linked into a boot image. When Oberon is started the boot image is loaded into memory by a boot loader. Since the statically linked modules loaded from the boot image are not loaded by the normal module loader, the boot loader must also do the initializations and fixups normally done by the dynamic module loader.

Most Oberon systems running on top of a host system use a boot loader implemented in C. This has the advantage, that features provided by the host system or the C run–time library [KR88] can easily be used by the Oberon kernel through an up–call mechanism provided by the boot loader. Such features are:

- heap allocation using malloc
- implementation of modules Math/MathL (sin, etc.)
- calls to the host systems API (application program interface)

- export of callbacks (e.g. WindowProc, etc.)
- access to the process environment (command line parameters, standard input/ouput, etc.)

The major disadvantage of this C based solution is its inflexibility. Whenever a new kernel feature is required, the (C) boot loader, the boot image and some of the kernel modules have to be updated. When different projects require different kernel features, things get even worse and we end up having different incompatible versions of the boot loader and kernel. A first step towards our more flexible boot scheme is the implementation of a new boot linker and boot loader which is able to create a self–contained Windows executable file directly from Oberon object files. There are thus no longer any dependencies on a C–based boot loader or the C run–time library. The rest of this section describes the implementation of the new boot linker and boot loader.

On Windows, executable files use the PE (portable executable) file format [Lev00], which is an extension of the well–known COFF (common object file format) format. A PE file is structured into different sections, among which typical sections are:

.text executable code

.data initialized variable data

.rdata initialized constant data

.idata import table

.edata export table

.reloc relocations

.rsrc resources (icons, cursors, version information, etc.)

The Oberon *PELinker* tool works by emulating an Oberon systems kernel and module loader. All the modules to be statically linked are loaded into the heap of the emulated system (pseudo heap). This heap, together with a small stub, is located in the *.text* section of the PE file. The stub does fixups for APIs imported from DLLs,

initializes the Oberon heap and calls all module bodies of the statically linked modules. Since Oberon calls APIs imported from DLLs using procedure variables, the fixup code in the stub copies the address values from the import table — generated by the windows PE loader — to the procedure variables.

The Oberon PE–linker supports two different variants of executable files: DLL and EXE. An EXE file is needed when Oberon is started as stand–alone Windows application. A DLL file is needed when Oberon is loaded as library within the process and the address space of an other Windows application. In the second case, the stub must not only handle initialization of the Oberon heap when the DLL is loaded, but it must as well handle cleanup when the DLL is unloaded. This cleanup is important since a DLL shares its address space and other resources (file handles, memory, etc.) with its host process. Resources allocated by an EXE file are automatically released by the system when the process ends.

The cleanup is implemented in module *Kernel*. The stub calls the *Kernel* body twice: a first time when the DLL is loaded and a second time when the DLL is unloaded.

The skeleton of a DLL stub has the form:

```
// BOOL WINAPI DllMain(HINSTANCE hinstDLL,
//    DWORD fdwReason, LPVOID lpvReserved);
  PUSH EBX, EBP, ESI, EDI
  MOV EBX, fdwReason
  CMP EBX, DLL_PROCESS_DETACH
  JNE Else
  JMP Unload
Else:
  CMP EBX, DLL_PROCESS_ATTACH
  JNE Exit
Load: // loading Oberon.DLL
  API fixups for all imports
  initialize Oberon heap
  call module bodies
  JMP Exit
Unload: // unloading Oberon.DLL
  CALL Kernel
Exit:
  POP EDI, ESI, EBP, EBX
  MOV EAX, TRUE
  RET 12
```

The *PELinker* uses a textual description file — similar to the module definition files used by the Visual C++ linker — which contains all the parameters needed by the linker. The following code is an excerpt from the description file used to build Oberon.DLL. In the exports section, we can see how exported Oberon procedures are assigned a named entry in the export table. The imports section illustrates how entry points exported by a DLL are assign to global Oberon variables. In the imports section, we can also see how the pseudo heap loaded from the executable file is mapped to module Kernel (*HeapAdr*, *HeapSize*, etc.).

```
LIBRARY Oberon
MODULES Kernel32, Kernel, ..., COM
EXPORTS
  DllGetClassObject = COM.DllGetClassObject,
  DllRegisterServer = COM.DllRegisterServer,
  DllUnregisterServer = COM.DllUnregisterServer,
  DllCanUnloadNow = COM.DllCanUnloadNow
IMPORTS
  Kernel.heapAdr = HeapAdr,
  Kernel.heapSize = HeapSize,
  Kernel.hInstance = hInstance,
  Kernel32.AllocConsole = KERNEL32.AllocConsole,
  Kernel32.CloseHandle = KERNEL32.CloseHandle,
  Kernel32.CopyFile = KERNEL32.CopyFileA,
  Kernel32.CreateDirectory = KERNEL32.CreateDirectoryA,
  ...
```

Since the new Oberon inner–core is built using *PELinker*, features previously provided by the C run–time libraries or the boot loader have to be re–implemented in plain Oberon.

Heap In the C–based boot loader, the Oberon heap was allocated as one big block using *malloc*. This had the inconvenience that the heap size could only be chosen at startup. Thus, if the heap size was chosen to small, Oberon would stop with a heap overflow trap, if the heap size was chosen to large memory needed by other applications was wasted. In the new kernel, the heap is dynamically allocated and released in 256 kByte blocks. Thus, memory is only allocated when needed.

Math and MathL The modules Math and MathL have been replaced by the implementations of Native Oberon for PC.

	parameters	caller	callee
Pascal	left to right	-	ESP, EBP
C	right to left	ESP	EBP, EBX, ESI, EDI
WinAPI	right to left	-	ESP, EBP, EBX, ESI, EDI

Table 4.1: Intel calling conventions.

Exception Handling A mechanism for structured exception handling based on metaprogramming (see section 4.4) has been implemented.

Import and Export Importing APIs and exporting callbacks are directly supported by the new boot linker (see above).

4.3 Compiler Extensions

This section describes extensions made to the Oberon compiler to support direct calls to and from non–Oberon procedures. The sysflag [Cre90] notation is used to mark system specific calling and parameter passing conventions. When using sysflags, module *SYSTEM* must be imported, since such code is not portable.

The different calling conventions differ by the order in which actual parameters are passed on the stack and the set of registers which have to be saved by the caller or callee. By default, Oberon for Windows uses the Pascal calling convention. Table 4.1 gives an overview on the different calling conventions typically used on Intel platforms.

Non–Oberon procedures are marked using the *winapi* (Win32 API) or *c* (C) sysflag:

```
PROCEDURE [ "[" ( "winapi" | "c" ) "]" ] ... .
```

This applies to procedure type declarations as well as to procedure implementations. Procedures with different calling conventions are not (assignment) compatible.

The following code fragment shows an excerpt from the implementation of a Windows callback procedure in Oberon. The first variant

uses the old–style Oberon compiler, the second one uses the new compiler supporting different calling conventions. In the first variant the programmer must revers the parameterlist and add register save and restore code explicitly. In the second variant this is automatically done by the compiler.

```
PROCEDURE *WndProc(lPar, uPar, msg, hwnd: LONGINT):
    LONGINT;
  VAR EBX, ESI, EDI: LONGINT;
BEGIN
  SYSTEM.GETREG(S.EBX, EBX);
  SYSTEM.GETREG(S.ESI, ESI);
  SYSTEM.GETREG(S.EDI, EDI);
  ...
    SYSTEM.PUTREG(S.EBX, EBX);
    SYSTEM.PUTREG(S.ESI, ESI);
    SYSTEM.PUTREG(S.EDI, EDI);
    RETURN 1
  ...
  SYSTEM.PUTREG(S.EBX, EBX);
  SYSTEM.PUTREG(S.ESI, ESI);
  SYSTEM.PUTREG(S.EDI, EDI);
  RETURN 0
END WndProc;

PROCEDURE [winapi] *WndProc(hwnd, msg, uPar, lPar: LONGINT):
  LONGINT;
BEGIN
  ...
    RETURN 1
  ...
  RETURN 0
END WndProc;
```

When interfacing to non–Oberon libraries, the compatibility of basic and complex types must be considered. Mapping of basic data types does not present a problem, because all have the same word size both in Oberon and in C. In Oberon, all numeric types are signed, with the exception of type *CHAR* which is unsigned. These definitions are mapped to C–types as shown in table 4.2.

In Oberon, when calling a procedure with an open array parameter or reference record parameter, a hidden descriptor containing further type information is passed as additional parameter. This is not compatible with non–Oberon libraries where typically no run–time type

Oberon	C
CHAR	unsigned char
SHORTINT	signed char
INTEGER	signed short int
LONGINT	signed long int
SET	DWORD
REAL	float
LONGREAL	double

Table 4.2: Oberon to C type mapping.

information is provided. To pass open array and record parameters in a C compatible way, their types must be marked with the *notag* sysflag.

```
RECORD [ "[" "notag" "]" ] ... .
ARRAY [ "[" "notag" "]" ] ... .
```

To maintain type–safety, different restrictions apply to types marked with *notag*. *notag* record types may only be used if the type tag is not used, thus the compiler does not support any type tests or guards on such records (*IS*, *WITH*, (Type)). To maintain type–safety the dynamic type of a *notag* record must always be the same as its static type. *notag* record types may neither be extended nor can they be an extension of another record.

notag open array types can only be used if the hidden array descriptor is not accessed directly *(LEN)* or indirectly (index checks, *COPY*, etc.). Thus, *notag* open arrays can only be used to define procedure types and variables but can not be used as formal parameters by procedures implemented in Oberon. However, using notag open arrays as actual parameters when calling external (procedure variable) procedure is allowed.

For all reference parameters and structured value parameters, *NIL* may be passed as parameter value to procedures marked with a sysflag. This feature is specially usefuly to call C–API procedures where *NULL* is often used as default value for optional parameters.

In typical APIs (e.g. Win32) most procedures are implementated as functions, returning an error value indicating success or failure of the

procedure called. Checking this return value after each API procedure invocation makes source code less readable and is often not needed at all. The extended Oberon compiler allows calling functions marked with a sysflag as proper procedures or as functions.

4.4 Exception Handling

ETH Oberon implementations provide no mechanism for structured exception handling like Java or C++ do. This is due to the fact that Oberon's approach using a system–wide trap handler has proven to be a very efficient solution in most exception situations. The trap handler displays the calling sequence (stack frames) which finally led to the exception in the so–called 'Trap Viewer'. In standard Oberon imlementations, there is only one thread of execution, the 'Oberon Loop'. Thus after an exception occured, the Oberon loop can simply be restarted. If Oberon however is embedded into a host application, there are situations where a more complex exception handling scheme is required.

When a host application calls a call–back routine in an embedded Oberon object, Oberon must be able to return an error code to the caller when the called procedure runs into a exception. Using the standard trap handler the failing thread is aborted without notification, thus the host application will lose all information attached (thread message queue, thread local data) to this thread. This will most likely lead to an abnormal termination of the host application.

The exception handling technique implemented for Oberon is programmed as an ordinary Oberon module using metaprogramming and does not need any special language construct or compiler support [HMP97]. Exception handlers are detected by the trap handler using metaprogramming based on reference information generated by the compiler. In case of an exception, the trap handler calls matching handlers before aborting with a trap.

Exceptions are objects of an exception class, which is a subclass of *Exceptions.Exception*. There exist system exceptions (*SysException*) and user exceptions (*UserException*). System exceptions (e.g. access violation, division by zero, etc.) are triggered automatically by the system, while user exceptions are raised by the library call *Excep-*

tions.Raise(exception). Exceptions are caught by an exception handler which is an ordinary procedure H with the following characteristics:

- H is declared local to a procedure P in the call–stack of the failing procedure.
- H has a single reference parameter of type E, which is the type of the exception to be caught or a superclass thereof. The return type of H is *LONGINT*. Valid values for this return type are:

  Ignore Ignore the exception and continue execution. This is only valid for exceptions raised by a call to *Exceptions.Raise*, or by a *ASSERT* or *HALT* statement.

  Forward Continue searching an exception handler in the caller of P.

  Abort Abort execution, display the trap viewer.

  ReturnFail(context) Return to the procedure that called *Exceptions.Failed(context)* and return *TRUE*.

The following code fragments show the usage of a user exception:

```
TYPE
  MyException = RECORD (Exceptions.UserException) END;

  PROCEDURE P();
    VAR e: MyException;
    PROCEDURE Handler(VAR e: Exceptions.Exception): LONGINT;
    BEGIN
      IF e IS MyException THEN
        (* handle MyException here *)
      ELSE
        (* another exception occured *)
        RETURN Exceptions.Forward
      END
    END Handler;
  BEGIN
    ...
    Exceptions.Raise(e)
    ...
```

Since exception handlers are discovered by the trap handler at run–time, there is no run–time overhead for error–free programs. This method is however limited, since there is no mechanism to continue

execution after an exception, and an exception will always result in an *Abort* or *Ignore*.

This is sufficient if only some cleanup, for example of a global state, is needed. For more advanced exception handling a mechanism for catching exceptions within an protected block is provided. This mechansim is required to implement exception handling with continue semantics, thus the thread causing an exception can continue execution at a predefined point.

The mechanism implemented is similar to the *setjmp* and *longjmp* feature of the C programming language [KR88]. The implementation of jump (*ReturnFail* and *Fail*) in Oberon is however limited to 'structured' jumps. Calling *Fail* or *ReturnFail* will result in a trap if the matching *Failed* procedure was not called within the current call–stack.

```
TYPE
  MyException = RECORD (Exceptions.UserException) END;

PROCEDURE P();
  VAR e: MyException; context: Exceptions.Context;
  PROCEDURE Handler(VAR e: Exceptions.Exception): LONGINT;
  BEGIN
    IF e IS MyException THEN
      RETURN Exceptions.ReturnFail(context)
    END;
    RETURN Exceptions.Forward
  END Handler;
BEGIN
  IF ~Exceptions.Failed(context) THEN (* no error occured *)
    ...
    Exceptions.Raise(e)
    ...
  ELSE (* error occured and Exceptions.ReturnFail was called *)
    ...
  END
```

Using *Exceptions.Fail*, this mechanism may as well be used without an exception handler.

```
PROCEDURE P();
  VAR context: Exceptions.Context;
BEGIN
  IF ~Exceptions.Failed(context) THEN (* no error occured *)
    ...
    Exceptions.Fail(context)
    ...
```

```
  ELSE (* Exceptions.Fail was called *)
    ...
  END
```

4.5 Multithreading Support

Oberon is a single–threaded, single–user, co–operative multi–tasking operating system. The single thread used by Oberon is the 'Oberon Loop'. This loop acts as a dispatcher for keyboard input, mouse input and background tasks. The code below shows an excerpt from the single–threaded implementation of the 'Oberon Loop' in Oberon for Windows. The *Win32.PeekMsg* call checks the windows message queue for new messages and the *Win32.DispatchMsg* call invokes a message handler (window procedure) which dispatches the different window messages. Messages handled by Oberon are: keyboard messages, mouse input messages and window update messages. Keyboard and mouse messages are directly translated into their Oberon equivalents and are buffered by module *Input*. Screen update messages are translated to special character codes. All other messages are delegated to the Windows default message handler.

```
Input.Mouse(keys, X, Y);
LOOP
  IF Win32.PeekMsg(...) # 0 THEN Win32.DispatchMsg(...) END;
  IF Input.Available() > 0 THEN (* keyboard input *)
    Input.Read(ch);
    IF special char THEN
      handle special chars
    ELSE
      broadcast consume message
    END;
    Input.Mouse(keys, X, Y)
  ELSIF keys # {} THEN (* mouse click *)
    REPEAT
      broadcast track message;
      IF Win32.PeekMsg(...) # 0 THEN Win32.DispatchMsg(...) END;
      Input.Mouse(keys, X, Y)
    UNTIL keys = {}
  ELSE (* mouse move and background tasks *)
    M.X := X; M.Y := Y;
    broadcast track message;
    Input.Mouse(keys, X, Y);
    WHILE (keys = {}) & (X = M.X) & (Y = M.Y) &
```

```
        (Input.Available() <= 0) DO
      handle next ready task;
      IF Win32.PeekMsg(...) # 0 THEN Win32.DispatchMsg(...) END;
      Input.Mouse(keys, X, Y)
    END
  END
END
```

In the above code, we can see that keyboard input has the highest priority followed by mouse clicks. Background tasks have the lowest possible priority, they run only when there is no user interaction. The implementation illustrated above has three big drawbacks:

- Oberon uses polling to dispatch the windows message queue. This results in a high CPU load, thus slowing down other applications.
- A long running command prevents Windows messages from being dispatched. This often blocks the message sender, e.g. the Windows desktop!
- Oberon can only dispatch messages sent to the message queue of the thread executing the 'Oberon Loop'.

These deficiencies can be avoided by using multithreading. Hence, there must be a separate thread responsible for dispatching the Windows message queue and a so–called worker thread running the 'Oberon Loop'. The implementation of multithreading requires some changes and additions to the standard Oberon system.

- The inner core must be made thread–safe. This affects the modules: *Kernel*, *FileDir*, *Files* and *Modules*.
- The garbage collector must be extended to check for candidates on all thread stacks.
- A mechanism for inter–thread communication between the Windows message queue dispatcher and the 'Oberon Loop' threads is required.
- Existing code should still work without modification.

A new module *Threads* is provided to program multithreaded applications.

```
DEFINITON Threads;

  CONST
  (* priority levels *)
    Low = -1; Normal = 0; High = 1;

  (* infinite timeout value *)
    Infinite = -1;

  TYPE
  (* thread descriptor, used to store thread information. *)
    BodyProc = Modules.Command; (* Thread body procedure. *)
    Thread = POINTER TO RECORD (Kernel32.Object)
      stackBottom: Kernel32.ADDRESS;
      name: ARRAY 64 OF CHAR; (* Name of thread. *)
      safe: BOOLEAN (* Restart the thread after a trap. *)
    END;

  (* threads enumerator *)
    EnumProc = PROCEDURE (t: Thread);

  (* base type for critical section objects. *)
    Mutex = POINTER TO RECORD count-: LONGINT END;

  (* base type for events *)
    Event = POINTER TO RECORD (Kernel32.Object) END;

  VAR
    oberonLoop: Thread; (* thread executing Oberon.Loop *)

(* start a new thread executing p. *)
  PROCEDURE Start(t: Thread; p: BodyProc; stackLen: LONGINT);

(* get the current thread beeing processed. *)
  PROCEDURE This(): Thread;

(* enumerate all threads. *)
  PROCEDURE Enumerate(p: EnumProc);

(* stop execution of thread t. *)
  PROCEDURE Kill(t: Thread);

(* register the calling thread as non-Oberon thread *)
  PROCEDURE Register(name: ARRAY OF CHAR): Thread;
```

```
(* unregister a thread previously registered with Register *)
  PROCEDURE Unregister(t: Thread);

(* change the priority of thread t to prio *)
  PROCEDURE SetPriority(t: Thread; prio: LONGINT);

(* get the priority for thread t *)
  PROCEDURE GetPriority(t: Thread; VAR prio: LONGINT);

(* suspend execution of thread t *)
  PROCEDURE Suspend(t: Thread);

(* resume execution of thread t *)
  PROCEDURE Resume(t: Thread);

(* set the calling thread to sleep for the specified amount
    of milliseconds *)
  PROCEDURE Sleep(ms: LONGINT);

(* pass control to the next ready thread. *)
  PROCEDURE Pass();

(* start non-interruptable section *)
  PROCEDURE BeginAtomic(): BOOLEAN;

(* end non-interruptable section *)
  PROCEDURE EndAtomic();

(* initialize a new mutex *)
  PROCEDURE Init(mtx: Mutex);

(* wait for ownership of the mutex *)
  PROCEDURE Lock(mtx: Mutex);

(* release ownership of the mutex *)
  PROCEDURE Unlock(mtx: Mutex);

(* try to take ownership of the mutex without blocking *)
  PROCEDURE TryLock(mtx: Mutex): BOOLEAN;

(* set an event *)
  PROCEDURE Set(event: Event);

(* reset an event *)
  PROCEDURE Reset(event: Event);

(* initialize a new event *)
  PROCEDURE Create(event: Event);
```

```
(* wait for an event for at most time milliseconds *)
  PROCEDURE Wait(event: Kernel32.Object; time: LONGINT): BOOLEAN;

END Threads.
```

The primary mechanism used for synchronization is critical section objects. Their introduction offers an efficient way to implement mutual exclusion. The inner–core modules *Kernel*, *FileDir*, *Files* and *Modules*, are implemented as monitors using critical section objects. Since Oberon does not feature a concurrent garbage collector, the collector must run in an atomic mode. This is implemented by module *Threads* in the procedure *BeginAtomic* and *EndAtomic*. Checking all thread stacks does not present a problem, since module *Threads* already maintains a list of all threads together with their initial context (register dump).

Since the single process multi–tasking model works well for user driven applications, the Oberon loop is still responsible for processing all keyboard and mouse events, controlling the screen and accessing globally shared data structures related to input and display handling. As a consequence, multi–threading is transparent for ordinary Oberon programs that are executed from within the Oberon loop, hence such programs can freely access global data types without explicit synchronization. Other threads may access such data structures only in a controlled way to avoid inconsistencies. The mechanism used for inter–thread communication between the Windows message queue dispatcher and the Oberon loop threads is discussed in the following chapter.

Chapter 5

Pluggable Objects as Web Applets

5.1 Introduction

Most web pages seen on the Internet have a relatively primitive, static character. Even though many web pages seem to be very active, their activity is typically limited to animated pictures and sounds. Behind the scene, these active elements have a rather static character, thus there is no non–trivial behaviour or even user interaction.

Although forms and image maps allow a measure of user interaction, all the main processing is done remotely on the web server. For any frequently visited site, this incurs a considerable load on the server. Moreover, the final result of all the server's hard work is static (HTML) content, which is downloaded only to be again passively displayed by the web browser.

Web applets promise an alternative model for web content — a model where as much processing as possible is done on the local computer and not on the server. The crucial difference between an applet and a traditional application is that applets are, by nature, network–aware and truly distributed. Currently, there are two competing applet models widely in use: Java applets and ActiveX controls (ActiveX is discussed in more detail in section 7.1). There are three major issues

to be considered with web applets:

- **portability.** Portability or platform independency is a must for applets which should be usable on a wide range of different software and hardware platforms. Today, the standard solution to this problem is the Java virtual machine (JVM). Java compilers normally compile Java source code into Java byte code, a platform independent binary format understood by the JVM. Thus Java classes compiled into byte code can be interpreted on any platform where an implementation of the JVM is available.

  ActiveX controls are always provided in a platform specific form, mostly as dynamic link libraries for the Windows platforms. ActiveX controls are thus not portable at all, and their use is limited to the Windows platforms. Only recently, Microsoft has launched a new technology '.net' which provides portability for COM based components similar to the JVM.

  Oberon provides two different techniques for platform independent code:

  - *Slim Binaries* is a software distribution format optimized for efficient on–the–fly compilation of Oberon modules (see also section 1.5.1).
  - *SourceCoder* is a format based on a highly compressed token stream representation of Oberon source codes.

- **efficiency.** As most people still access the Internet using a realtively slow analog modem connection, the download size of applets should be as small as possible. For this purpose most of the different applet technologies use compressed archives for transfering applet data, e.g. Java archives (JAR) or cabinet files (CAB).

  Once an applet has been downloaded to the client machine the run–time efficiency of applets becomes an import issue. Java byte code is typically interpreted and thus inefficient, even using a just–in–time (JIT) compiler results still in slow code and high memory consumption. ActiveX does not have this problem since

only compiled code is used. Oberon implements efficient on–the–fly compilation, which is fast and generates reasonable quality code.

- **security.** Using applets presents a certain level of risk, for example it is easy to imagine, that by simply loading a web page, viruses or other hostile software are installed and executed on the client computer. An efficient and effective security system is thus required to protect against potential hostile attacks. In section 5.3 an overview of the different approaches taken by Java, ActiveX and Oberon is presented.

5.2 Pluggable Oberon Object

ETH Oberon features an extensible document–oriented user interface [Zel97]. The document system can be extended in two ways:

- by providing implementations for new document access protocols
- by implementing new document classes

Named documents are accessed by the end–user and the system using the URL (Uniform Resource Locators) notation. As all Oberon documents are stored as binary files with a standardized header layout, an implementation of a document access protocol can handle all Oberon document types in a generic way. Thus any combination of access protocol and document type will work. Since Oberon documents can be arbitrary nested within other Oberon documents, they can immediately be used as simple applets. This approach is much more general than the hyper–text centric web, since the document system of Oberon allows for browser–style navigation through any combination of hyper–documents (panels, texts, poly worlds, etc.).

As such document applets require resources that may not already be available on the client system, self–contained documents [Zel97, Mar96] are used for distribution. Self–contained documents are documents that have their resources integrated into themselves as *document meta data*. Should the resources required on opening a document not be available on the destination platform, the resources of the

self–contained document are installed automatically before the document contents is loaded and displayed. Nothing happens should all the required resources already be available. Unfortunately, using self–contained documents as web applets has several disadvantages:

- Only documents can be used as applets, but often a simple frame (gadget) would be more suitable for a given task. In these cases an unnecessary document wrapper has to be provided.
- Each instance of a self–contained document contains all the resources needed. This is very inefficient since common resources of different documents are transferred and installed many times. This is similar to statically linked applications, where each instance comes with it's own copy of all the run–time libraries.
- The consistency checking abilities on installation of a self–contained document and its resources are rather limited. A more general installation facility which could also be used for installing ordinary application packages would be preferable.

To overcome the limitations of self–contained documents a new framework for installing applications, and for plugging frames as applets into web documents has been developed. This framework is based on the two modules *Packages* and *PlugIns*. Module *Packages* provides tools to pack application data together with installation scripts and version information into archives; as well as tools to unpack and install such archives. Module *PlugIns* implements background downloading and installation of applets provided as *Packages* archives.

5.2.1 Packages

Module *Packages* is used for authoring new application packages as well as to install (unpack) packages. A package is defined using a simple textual description language. The EBNF syntax of this simple description language is:

```
package = "PACKAGE" name version gen { data | url | cmd } .
data    = "DATA" name [ version ] [ ":=" file ] .
url     = "URL" name [ version ] [ ":=" url ] .
cmd     = filecmd | "MSG" text | "SET" key ":=" value .
filecmd = ( "COMPILE" | "COPY" | "DEF" | "DEST" ) name .
version = major "." minor .
```

The following list gives a short description for each of the keywords used in the above grammar.

PACKAGE Create a new package *name* with generator *gen*. *version* specifies the version of the package, consisting of a major and a minor number. The generator will be used to create an object if the package is opened as document or as plugin. The generator is ignored if the package is installed using *Packages.Install*. *gen* has the following syntax:

```
gen = ( "obj" | "doc" | lib ) ":" name .
lib = "libref" | "libdeep" | "libshallow" .
```

The generator consists of a prefix and a name. The prefix specifies how the name should be interpreted to create an object. *obj* is used to create a new object instance by calling the generator command. *doc* is used to open an existing document or create a new document instance. The *lib* prefixes are used to retrieve an object from a public object library by reference, or either as shallow, or deep copies.

DATA Include the file *name* in the package. Using the ':=' notation, the file in the package can be given a name different from the name of the actual file being copied. If no version is specified, the version of the package is used. The date and time the original file was last changed are included in the package as additional implicit version information.

URL Reference to another package required to be installed before the package is installed. By default, the name of the referenced package is interpreted as URL relative to the URL context of the object which requested its installation. If this scheme does not fit, the ':=' notation can be used to specify some other (full or relative) URL. The version number of an URL entry is used to determine if the referenced package needs to be requested at all. Thus if the version number of the URL entry specifies a version which is already installed the package is not installed.

DEST Set the destination directory or volume for the following file commands (COMPILE, COPY, DEF). The following directory names are pre–definied:

SYSTEM The directory where all the system files are installed. E.g. .Tool, .Fnt, .Lib, etc.

OBJ The directory where all the compiled modules (.Obj) are installed.

SRC The directory where all the source code files are installed.

APPS The directory where all the miscellaneous application files and package files are installed.

DOCU The directory where all the tutorials and other documentation files are installed.

WORK The Oberon working directory.

OBERON The Oberon (boot) root directory.

COMPILE Compile the source file *name* in plain or encoded form (SourceCoder).

COPY Copy the file *name*.

DEF Update *Definitions.Arc* with a new module interface definition for the module *name*.

MSG Display text in the log.

SET Store a key–value pair in Oberon's system configuration text (*Oberon.Text*).

To avoid name conflicts when distributing software as packages, a name prefix registry is provided on the Oberon homepage (http://www.-oberon.ethz.ch/native/Registry.html). This registry can be used to register module names as well as names for arbitrary (data) files.

When a package is being installed, the version information of the package itself and that of the resources contained in the package are checked against version information of already installed resources. The installer performs the following heuristic:

If the resource is already installed (thus, if there is version information stored in the registry):

1. If the major version is different, the installer aborts with a version conflict error.

2. If the minor version is smaller, the installer ignores the resource.

3. If the minor version is greater, the installer installs the resource.

4. If the minor version is identical, then if the time stamp of the resource is newer the resource is installed otherwise ignored.

The resource is not yet installed (thus, if there is no version information stored in the registry):

1. If there exists already a file of the given name, the installer aborts with an error.

2. If there exists no such file, the installer installs the resource.

Since Oberon is a modular system using dynamically linked modules, it is not possible to identify all resources (modules, libraries, etc.) needed by an object (or composed object) to work correctly. *ResourceFinder* is a tool which recursively traverses all dependencies of a root object to identify all resources required by that object. All the resources which are part of the basic system or of other already installed packages are excluded from the recursive search. The dependencies traced by *ResourceFinder* are:

Modules A natural starting point for finding all the modules needed by an object is its generator string. All other string attributes containing a valid command name (M.P) must be considered as well.

Links Typically an object maintains links to other objects, e.g. model objects or contained objects. This linked objects can be traced by following all the links provided by an object and by traversing the part of the display space rooted by the object (container frame).

Libraries Public object libraries (e.g. fonts) are another class of resources handled by *ResourceFinder*.

Data Files Any additional resources required by the object must be specified explicitly by the programmer.

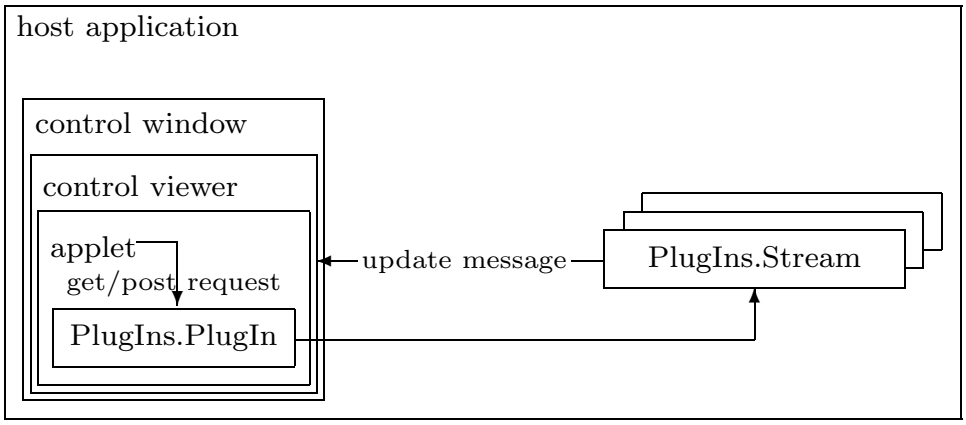

Figure 5.1: Embedding of an applet frame.

5.2.2 PlugIns

Module *PlugIns* defines two new data types: *PlugIn* and *Stream*. These types are introduced to abstract from the specific implementation details of the different plugin or applet technologies. *PlugIn* is a non–visual object type, since it does only provide the infrastructure to create a visual applet. The applet itself is a standard Oberon frame. This frame is embedded in a special viewer or view (see section 3.3) which is integrated in the host container application (e.g. Netscape Navigator), see figure 5.1.

```
GetURLProc = PROCEDURE (plugin: PlugIn; url: ARRAY OF CHAR);
PostURLProc = PROCEDURE (plugin: PlugIn; dest, src: ARRAY OF CHAR);
ErrorProc = PROCEDURE (plugin: PlugIn; msg1, msg2: ARRAY OF CHAR);

PlugIn = POINTER TO RECORD (Objects.Object)
  V: Display.Frame; (* view displaying the applet *)
  getURL: GetURLProc; (* GET request *)
  postURL: PostURLProc; (* POST request *)
  error: ErrorProc; (* display an error message *)
  base: Stream (* stream (SRC) to load the applet *)
END;

URLStr = ARRAY 1024 OF CHAR;

Stream = POINTER TO RECORD (Objects.Object)
  file: FileDir.FileName; (* file name of local copy *)
  url: URLStr; (* the requested URL *)
  state: LONGINT
```

```
  (* Done, NetworkErr, UserBreak, OtherError, Transferring *)
END;
```

The definitions of *PlugIn* and *Stream* allow for the implementation of asynchronous access to data objects on the web identified by URLs. The URLs may either be relative to the context of the plugins, or absolute. Clients requesting an URL are notified by broadcasting Oberon update messages (for *Stream* objects) whenever the state of a pending request changes. This asynchronous scheme is used by module *PlugIns* to implement background installation of resources required by an applet. For this purpose, the view embedding the applet frame captures all the update messages generated by streams. Once the applet is successfully installed, the update messages are forwarded to the contained frame, following the parental control principle.

Pseudo code for background applet installation:

```
PROCEDURE AppletViewHandler(V: Object; VAR M: Msg);
  VAR stream: Stream; plugin: PlugIn;
BEGIN
  IF M IS StreamUpdateMsg THEN
    stream := M.stream; plugin := V.win.plugin;
    IF plugin is not running THEN
      IF stream in plugin.queue THEN
        IF stream completed THEN
          add stream to plugin.queue
        ELSIF stream request failed THEN
          display error msg;
          abort installation of plugin
        END
      END
    ELSE
      V.dsc.handle(V.dsc, M)
    END
  END
END AppletViewHandler;

PROCEDURE AppletInstaller(me: PlugIn);
BEGIN
  WHILE streams available in me.queue DO
    IF stream completed THEN
      remove stream from me.queue;
      check integrity of package on stream;
      install package, this may require additional streams
    END
```

```
  END;
  IF no error occured THEN
    create applet frame;
    me.V.dsc := new applet frame
  ELSE
    display error msg
  END
END AppletInstaller;
```

Objects of type *PlugIn* are moreover used to map parameters provided by the host environment (e.g. param tags in html) to Oberon attributes. Attribute 'Name' is used to dynamically find objects in a given context by their name. For plugin objects, this context is normally a non–Oberon object. Finding objects is implemented by the view embedding the applet frame. The view knows both the plugin object and the applet frame and thus can handle the *FindMsg* accordingly.

Pseudo code for finding applets by name:

```
PROCEDURE AppletViewHandler(V: Object; VAR M: Msg);
BEGIN
  IF M IS FindMsg THEN
    IF M.obj = NIL THEN
      find plugin object with name = M.name on current web page
    END;
    IF M.obj = NIL THEN
      find child object with name = M.name of this applet
    END
  END
END AppletViewHandler;
```

5.3 Security Issues

In a dynamic environment such as the web, each resource accessed may contain some form of hostile software (viruses, etc.). Thus an effective security system is required to protect against potential hostile attacks. It is important to note, that the process and user based security mechanisms incorporated in operating systems like UNIX or Windows–NT are not sufficient for this purpose. A security level determined by the user or process identity is too coarse, hence security attributes must be associated with single applets, typically living as components in

the same process or even in the same thread together with their host application. For effective system protection, two problems have to be solved. The first problem is that of authentication. Applets (or applications) are not resistant to tampering, as the large number of viruses can attest. A trusted applet can be turned into a virus by 'adjusting' a few bytes. A digital signature testifying the authenticity and integrity is required. A second and much more difficult problem is to decide how and where to restrict system access for authenticated applets. Various approaches are possible, the following list summarizes the most common ones:

Prevention Untrusted applets are restricted as much as possible, but with enough freedom so that at least something useful can be done. This approach is often called 'sandbox' approach, since it allows only for the development of 'toy' applets.

Confirmation Each potentially 'dangerous' operation is verified by asking the user if it can be completed or not [Dät96]. This approach has the drawback, that the user will quickly lose his or her patience clicking away popup dialogs. It is thus often used in a restricted form, where the user confirms only the association of an applet with a pre–configured security zone.

Authenticated trust Only applets are used, that have been digitally signed by its trusted manufacturer or author. This approach has the downside, that nobody will trust a new and unknown author.

In practice a combination of the approaches listed above is used. Thus, applets run in 'sandbox' or security zone, with a set of pre–configured restrictions, depending on the manufacturer (authenticated trust) or the end–user (confirmation).

The current Oberon implementation uses only 'authenticated trust' since in the rather small Oberon community it is possible to know all trustworthy authors of applets. [Dät96] describes Oberon System extensions supporting prevention and confirmation based security as well. Oberon's inherent type safety allows for the efficient implementation of the latter .

5.4 The Oberon Web Browser

The Oberon system includes a web browser, which is fully integrated with the extensible document–oriented user interface. Web pages can be opened by just clicking at their URL. An interesting property of the browser is, that an HTML text is converted into normal Oberon text while downloading the page. This has the advantage, that an HTML page can be used and edited like any Oberon text. Thus, text, hyperlinks, and embedded objects can be manipulated like normal Oberon text and objects. For example a hyperlink copied to another text does still maintain the correct link target even after the original document is no longer available. A disadvantage of using the Oberon text format is that advanced formatting features (Tables, etc.) can not be implemented, since there is no equivalent construct in the Oberon text format. By using the Oberon browser plug–in for Netscape (see section 6.2) or Internet Explorer (section 7.6), Oberon applets can be used in a full featured web browser. The applets will still work in this foreign environment much like in their native environment. To allow efficient operation, and communication between applets only one instance of the Oberon run–time system is started (see section 3.4).

Module *HTMLPlugIns* extends the Oberon web browser with support for applets. Applets are integrated into a HTML page using the EMBED tag. This tag has the following attributes:

SRC URL of a web resource containing the applet data. The type of applet is determined by the MIME type of the resource. The current implementation of module *HTMLPlugIns* supports only the Oberon format for application packages described in the previous section.

WIDTH Width in pixels of the applet area.

HEIGHT Height in pixels of the applet area.

Name Used to access the applet object by name. Used by Oberon to identify objects in a HTML page.

Gen Generator for the applet object, only needed when the generator is different from the generator of the package.

By using the standard EMBED tag the same HTML pages and Oberon applets can be used with any browser, for which an implementation of an Oberon applet plug–in is available.

Chapter 6

Oberon Netscape Browser Plug–in

6.1 Introduction

Formatted text, graphics, images, and audio streams belong to different media types which can be externalized in countless different file formats. On the web, these file formats are called 'media types' and are identified by their MIME (Multipurpose Internet Mail Extensions) type. MIME types are denoted by unique strings, registered by the developers of new formats, with the IANA (Internet Assigned Numbers Authority) [Int01]. Examples of MIME types are: `text/html` and `image/jpeg`.

By default, the Netscape browser supports only the most commonly used media formats like HTML or JPEG. There are two different ways to extend the capabilities of the browser with the ability to handle additional data formats:

1. The browser maintains a table which maps MIME types to external programs, so–called helper applications. When the browser attempts to load a resource of the specified type, the data is passed to the associated helper application. Once the data transfer is finished, the helper application runs independently from the browser. Any program handling a file format of a defined MIME

type can be used as a helper application. The only pre–condition is, that the application is able to read the data from standard input or from a file specified on the command line.

2. Plug–ins are helper applications which run inside the browser, enabling seamless integration much like the browser internal data types. To accomplish this, the helper application must be provided as a plug–in library (DLL) which implements the interfaces as defined in the Netscape browser plug–in SDK [Net98].

The first method should be chosen for (large) documents which are referred to by a hyperlink, for example a word document. By including the command *Miscellaneous.OpenCmdLineDocs* in the Oberon startup script *Configuration.Text*, Oberon can immediately be used as a helper application for Oberon document files.

The second method is ideal for (small) documents which should be presented as embedded objects, typically floating in a hypertext document. The embedded object is part of the document data structure of the browser and can interoperate with other objects and the browser. In the following two sections, the implementation of a plug–in DLL for the Netscape browser running on the Windows system is discussed. This library allows the interoperation of embedded Oberon objects (applets) with the Netscape browser as well as with other applets in the browser.

6.2 Implementation of the Plug–in DLL

On startup, the Netscape browser builds a list of all available plug–in DLLs and of the media types they support. A DLL must meet the following conditions to qualify as a valid browser plug–in:

- The DLL file must be stored in the plug–ins directory with a name starting in 'NP' and ending with '.DLL'. For Oberon, a typical file name is:
  `c:\program files\netscape\communicator\program\plugins\NPOberon.DLL`

- The MIME types and filename suffixes for the data formats supported by the plug–in must be included in the version resource

MIME type	application/oberon
suffix	oaf
description	ETH Oberon Applet File

Table 6.1: Oberon plug–in DLL version information.

section of the DLL file. The version information for the Oberon plug–in DLL contains the information for the applet file format described in the previous chapter (see section 5.2.1).

- A Netscape browser plug–in DLL must provide implementations for the three predefined entry points: *NPInitialize*, *NPGetEntryPoints*, and *NPShutdown*.

6.2.1 NPInitialize

NPInitialize passes the browser function table to the plug–in DLL. This table is used by plug–ins to call functions in the browser. There are functions to read and write asynchronous URL streams, and functions to access the Java environment of the browser. To simplify calling these browser functions, wrapper function starting with a *NPN* prefix are used. As not all functions are implemented by the different browser versions, the wrappers must first check the availability of a given function befor invoking it. As the current implementation of the browser functions are not thread–safe, the wrappers must as well implement marshalling of the parameters and return values from the callers thread to the browser thread. As the plug–ins thread runs in the browser process, marshalling can be implemented by sending a message containing pointers to the function and parameters to the browser thread.

```
(* call PostURL from the browser thread *)
PROCEDURE npnPostURL(plugin: PlugIn; adr: ADDRESS): LONGINT;
  VAR url, file: ADDRESS; len: LONGINT; ch: CHAR;
BEGIN
  GET(adr-8, url); GET(adr-16, file); len := 0;
  REPEAT
    GET(file+len, ch); INC(len)
  UNTIL ch = 0X;
```

```
    INC(len);
    IF hasNotification THEN
      RETURN netscapeFuncs.posturlnotify(plugin.instance, url,
        NULL, len, file, True, 0)
    ELSE
      RETURN netscapeFuncs.posturl(plugin.instance, url, NULL,
        len, file, True)
    END
  END npnPostURL;

  (* wrapper called by plug--in *)
  PROCEDURE NPNPostURL*(plugin: PlugIn; urlDest, urlSrc:
    ARRAY OF CHAR): NPError;
    VAR cmd: NPNCommand; ret: LResult;
  BEGIN
    IF hasStreamOutput THEN
      cmd := npnPostURL;
      ret := SendMessage(plugin.hWnd, WMNPNCommand,
        VAL(User32.WParam, cmd), ADR(plugin));
      RETURN SHORT(ret)
    ELSE
      RETURN NPErrIncompatibleVersionError
    END
  END NPNPostURL;

  (* plug-in window handler, dispatches messages from
     the browser's thread message queue *)
  PROCEDURE [WINAPI] WndProc(hwnd: HWND; uMsg: LONGINT;
    wParam: WParam; lParam: LParam): LResult;
    VAR plugin: PlugIn; res: LResult; cmd: NPNCommand;
  BEGIN
    plugin := VAL(PlugIn, GetProp(hwnd, prop));
    IF uMsg = WMNPNCommand THEN
      cmd := VAL(NPNCommand, wParam);
      res := cmd(plugin, lParam)
    ELSE
      ...
    END;
    RETURN res
  END WndProc;
```

6.2.2 NPGetEntryPoints

NPGetEntryPoints requests the DLL to fill in the plug–in function table, which is used by the browser to call functions in the plug–in. The plugin–in function table contains functions to manage the creation,

presentation and removal of individual plug–in instances. The actual implementations of the different plug–in functions use *NPP* as a prefix in their name.

6.2.3 NPShutdown

NPShutdown is called immediately before the plug–in DLL is unloaded. A plug–in DLL is unloaded when either the browser application quits or when there are no more plug–in objects left.

6.2.4 Plug–In Window

The Oberon implementation *NPPlugIns* of a Netscape plug–in DLL is based on a port of Netscape's Windows template (WinTemp.c) provided with the plug–in SDK. When the (Netscape) browser parses an EMBED tag containing a SRC attribute referring to a file with media type *oaf* (associated with Oberon), a new instance of a plug–in object is created. The *NPPNew* callback in the plug–in function table (returned by *NPGetEntryPoints*) is called to create a new, uninitialized plug–in object. Plug–in objects (or instances) must be of type *NPP*. Whenever the size or position of the object changes, the function *NPPSetWindowPos* is called to notify the plug–in. This information is stored in a object of type *NPPWindow*. The following excerpt of module *NPPlugIns* shows the definition of types *NPP* and *PlugInWin*.

```
(* NPP is a plug-in's opaque instance handle *)
NPP = POINTER TO RECORD [NOTAG]
  pdata: PlugInWin; (* plug-in private data *)
  ndata: Kernel32.ADDRESS (* netscape private data *)
END;

(* a plug-in's control window *)
PlugInWin = POINTER TO RECORD (Windows.Window)
  plugin: PlugIns.PlugIn; (* plugin object for this window *)
  instance[UNTRACED]: NPP;
  window[UNTRACED]: NPPWindow
END;
```

Each plug–in instance consists of two objects. The first object is an instance handle of type *NPP* as specified by the plug–in SDK. The second object is an Oberon specific control window of type *PlugInWin*.

NPP instances are destroyed by the browser as soon as they are no longer needed. The callback *NPPDestroy* is called to allow the plug–in to cleanup resources and possibly store the state of the instance in a persistant form. After a plug–in instance has been created (*NPPNew*) and displayed (*NPPSetWindow*) its contents is loaded from the resource indentified by the SRC attribute.

Control windows (of type *PlugInWin*) are integrated in the display space as illustrated in figure 5.1. Module *WinFrames* implements a new viewer class, which allows the direct embedding of frames in a viewer without the need for any additional container or an Oberon viewer frame. This viewer must provide a complete environment for the applet frame, as it would have to in an ordinary frame container (e.g. a Panel). This includes the following:

ModifyMsg

The embedded frame must be constrained to the position and size of the surrounding control window (*NPPSetWindowPos*). For this purpose, the viewer handler captures all *Display.ModifyMsg* messages for it's child (parental control). As there is only one direct child in a control window, the offset of the child is always set to (0, 0), and the size of the child is always set to the full size of the control window.

Frame Printing

When printing a web page, the browser calls the *NPPPrint* method to ask the plug–in instance to print itself. An embedded plug–in shares printing with the browser. The plug–in prints the part of the page it occupies, and the browser handles the rest of the printing process, including displaying print dialog boxes, getting the printer device context, and, of course, printing the rest of the page.

As Oberon's own printer interface supports only page oriented printing (Open, Page, Close), two new procedures (StartPrint, EndPrint) have been added to allow printing to an arbitrary rectangle on a given device context.

```
StartPrint(P, hdc, x, y, w, h)
  save current page and frame setup
  set page and frame to printing area
  P.savedhdc := SaveDC(hdc)
```

```
  setup Oberon mapping mode and colors

StopPrint(P)
  RestoreDC(P.savedhdc)
  restore page and frame setup
```

FindMsg

If the applet sends a *FindMsg* to its parent, other Oberon applets located on the same web page must be found as if the web page was a normal Oberon text. If an named object can't be found within the applet, all sibling applets (with the same parent) are checked.

```
  PROCEDURE FindSelf(win: Window; VAR M: FindMsg);
    VAR name: Name;
  BEGIN
    GetObjName(win, name);
    IF name = M.name THEN
      M.obj := win.viewer.dsc
    ELSE
      GetObjName(win.viewer.dsc, name);
      IF name = M.name THEN
        M.obj := win.viewer.dsc
      END
    END
  END FindSelf;

  PROCEDURE FindObj(win: Window; VAR M: FindMsg);
    VAR disp: Window;
  BEGIN
    disp := root;
    WHILE (disp # NIL) & (M.obj = NIL) DO
      IF IsSibling(disp, win) THEN
        FindSelf(disp, M)
      END;
      disp := disp.link
    END
  END FindObj;

  PROCEDURE ViewerHandler*(V: Object; VAR M: ObjMsg);
    ...
  BEGIN
    WITH V: Viewer DO
      ...
      ELSIF M IS FindMsg THEN
        WITH M: FindMsg DO
          V.dsc.handle(V.dsc, M);
```

```
        IF M.obj = NIL THEN FindObj(V.win, M) END
      END
    ELSE
      ControlHandler(V, M)
    END
  END
END ViewerHandler;
```

Context Menu

There are no Oberon tool texts or menus available in the browser environment. Thus, another method is needed to activate standard Oberon actions which are not provided by the internal editor of the frame. The new viewer class implements context menus for this purpose (see figure 6.1). A context menu is displayed, when the frame does not consume a right click.

```
PROCEDURE ViewerHandler*(V: Object; VAR M: ObjMsg);
  ...
BEGIN
  WITH V: Viewer DO
    IF M IS InputMsg THEN
      WITH M: InputMsg DO
        ControlHandler(V, M);
        IF (M.res < 0) & (M.keys = {Right}) THEN
          menu := CopyPublicObject("ContextMenu");
          ContextMenu(V.win, menu);
          M.res := 0
        END
      END
    ...
    ELSE
      ControlHandler(V, M)
    END
  END
END ViewerHandler;
```

The default menu (ContextMenu) contains entries for the actions: *Select*, *Neutralize*, *Refresh*, *Inspect*, and *Oberon.Log*. *Select*, *Neutralize*, *Refresh*, are provided in the menu since the browser itself captures most of the function keys and it is thus not possible to use the standard Oberon function keys (F1, F2, F9) for plug–ins. As there are no Oberon tool texts available in the browser environment, the menu entries *Inspect*, and *Oberon.Log* are provided to open the Oberon log viewer and to inspect the properties of a plug–in object.

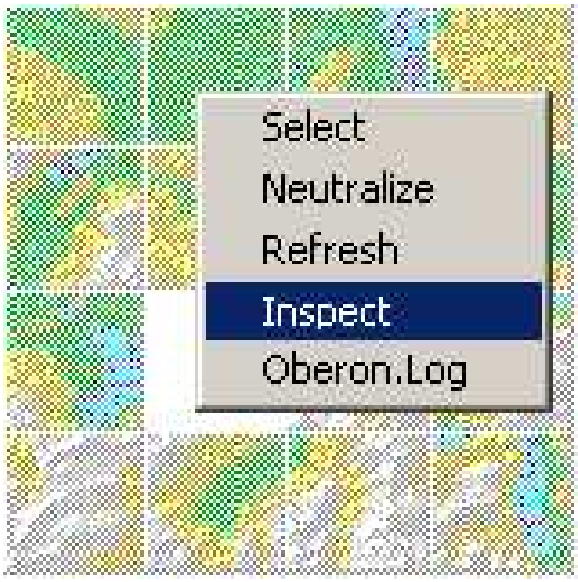

Figure 6.1: Context Menu.

6.2.5 Plug–In Streams

Network streams are implemented using the stream features provided by the browser. In this way, Oberon applets can take advantage of all the different protocol and caching services provided by the browser without beeing implemented to be used with a specific protocol.

After a new plug–in instance has been created, the *NPPNewStream* callback is called to pass the new instance a handle to its persistant data specified with the SRC attribute. Stream handles are of type *NPPStream*.

```
NPPStream = POINTER TO RECORD [NOTAG]
  pdata: Stream; (* plug-in private data *)
  ndata: Kernel32.ADDRESS; (* netscape private data *)
  url: Kernel32.LPSTR; (* full URL of the stream *)
  end, lastmodified: LONGINT;
  notifyData: Kernel32.ADDRESS
END;

Stream = POINTER TO RECORD (PlugIns.Stream)
  stream[UNTRACED]: NPPStream
END;
```

A plug–in instance may request additional streams through the *NPNGetURL* or *NPNGetURLNotify* browser functions. This functions

check the syntax of the URL passed and put the request into a browser internal queue. The plug–ins *NPPNewStream* callback is called as soon as the requested streams gets available. The plug–in instance then has to select a transmission mode for the data to be downloaded. In 'normal mode', the browser uses the *NPPWrite* method to 'push' stream data to the instance incrementally as it becomes available. In 'file mode', the browser saves the entire stream to a local file and passes the file path to the plug–in instance through the *NPPStreamAsFile* method.

While a plug–in object (applet) is beeing installed from it's package files, all stream notifications for that instance are captured by it's container. After the applet has been installed succesfully stream requests and notifications are handled as described in section 5.2.2. Error and status messages related to stream requests and applet installation are displayed in the browsers status bar as well as in the Oberon log viewer.

The Netscape plug–in API provides several functions releated to writing to streams. One group is related to writing data to other browser windows and frames (*NPNWrite*, ...), another to writing data to resources identified by an URL (*NPNPostURL*). The current Oberon plug–in framework does only support the later case, through an plug–ins *PostURL* method (see section 5.2.2).

6.2.6 Linking the Plug–In DLL

Module *NPPlugIns* can directly be linked into *NPOberon.dll* using the linker tool described in section 4.2. Since DLLs are statically linked *NPOberon.dll* will include all modules required by *NPPlugIns*. If modules included in the DLL are changed in the normal Oberon system, these changes are not automatically included in the (plug–in) DLL. Thus, the plug–in will continue to use outdated versions of the statically linked modules, resulting in an erroneous behaviour. To overcome this danger of inconsistency, COM is used to realize a dynamic linking scheme. The three entry points required by Netscape as well as all method stubs required by the Java classes for LiveConnect (see following section) are provided as COM interface *INPPlugIn*. Module *NPPlugIns* implements a singleton COM object providing only this interface.

```
(* COM interface used by NPOberon. *)
```

```
INPPlugInVTBL = POINTER TO RECORD (COM.IClassFactoryVTBL)
  GetEntryPoints: PROCEDURE [WINAPI] (pFuncs: PluginFuncs): NPError;
  Initialize: PROCEDURE [WINAPI] (pFuncs: NetscapeFuncs): NPError;
  Shutdown: PROCEDURE [WINAPI] (): NPError;
  UseJava: PROCEDURE [WINAPI] (plugin: PlugIn);
(* NPOberon *)
  javaGetObj: JavaMethodStub;
(* NPOberonObjectProxy *)
  javaGetBool, javaGetInt, javaGetReal, javaGetLongReal,
  javaGetString, javaGetLink,
  javaSetBool, javaSetInt, javaSetReal, javaSetLongReal,
  javaSetString, javaSetLink,
  javaFindObj, javaUpdate, javaExecute: JavaMethodStub
END
```

For the reasons stated above, *NPOberon.dll* is implemented by the single module *NPOberon*. This module acts as a wrapper for the actual implementation of the plug–in functionality provided by module *NPPlugIns*. The COM system is used by module *NPOberon* to connect to the implementation in module *NPPlugIns*. As this connection is established dynamically (at runtime) the plug–in DLL always uses the most recent Oberon modules.

```
PROCEDURE Init();
  VAR clsid: CLSID; iid: IID;
BEGIN
  CoInitialize(NULL);
  GUIDFromString(CLSIDNPOberonStr, clsid);
  GUIDFromString(IIDINPPlugInStr, iid);
  IF Succeeded(CoCreateInstance(clsid, NIL, CLSCTXInprocServer,
    iid, npPlugIn)) THEN
    npPlugInVTBL := npPlugIn.vtbl(INPPlugInVTBL)
  ELSE
    npPlugIn := NIL; npPlugInVTBL := NIL
  END
END Init;

(* Called immediately after the plug-in DLL is loaded. *)
PROCEDURE [WINAPI] Initialize*(pFuncs: NetscapeFuncs): NPError;
BEGIN
  IF npPlugInVTBL = NIL THEN Init() END;
  IF npPlugInVTBL # NIL THEN
    RETURN npPlugInVTBL.Initialize(pFuncs)
  END;
  RETURN NPErrModuleLoadFailedError
END Initialize;
```

```
(* Called immediately before the plug-in DLL is unloaded. *)
PROCEDURE [WINAPI] Shutdown*(): NPError;
  VAR err: NPError;
BEGIN
  CoUninitialize();
  IF npPlugInVTBL # NIL THEN
    npPlugInVTBL.Release(npPlugIn);
    err := npPlugInVTBL.Shutdown();
    npPlugIn := NIL; npPlugInVTBL := NIL;
    RETURN err
  END;
  RETURN NPErrModuleLoadFailedError
END Shutdown;

(* Fills in the func table used by Navigator to call
  entry points in plug-in DLL. *)
PROCEDURE [WINAPI] GetEntryPoints*(pFuncs: PTR): NPError;
BEGIN
  IF npPlugInVTBL = NIL THEN Init() END;
  IF npPlugInVTBL # NIL THEN
    RETURN npPlugInVTBL.GetEntryPoints(pFuncs)
  END;
  RETURN NPErrModuleLoadFailedError
END GetEntryPoints;
```

6.3 Installing the Plug–in DLL

Before a web page containing Oberon applets can be used, the Oberon plug–in DLL, its Java classes, and Oberon for Windows must be installed. The Oberon system is easily installed by unpacking some self–extracting archives containing all files and directories needed for a working Oberon environment. This simple approach does not work fot the plug–in files, as their destination directory depends on the pecific installation of the browser. There are three different ways to install a new plug–in:

- The user copies the plug–in files to the appropriate directory by hand. The current Oberon implementation uses this approach, as the user needs only to copy three files into the browsers *plugins* folder.
- The plug–in files are copied to the appropriate directory using an installer application like *InstallShield*.

- To automatically install a plug–in when the browser does not yet support a given MIME type, the JAR Installation Manager (JIM) can be used. The plug–in is insgtalled from a Java archive (JAR) containing a JavaScript installation script. To enable this feature, the web page author must include the *PLUGINURL* atttribute in the plug-in's *EMBED* tag. This attribute must point to the JAR archive to be used for installation.

6.4 Interfacing to Java using LiveConnect

The interoperability of Oberon plug–in applets as described in the previous section is rather limited, since only communication with other Oberon components is possible, but not with non–Oberon applets. LiveConnect is a proven technology that lets JavaScript, Java and plug–ins talk to one another. A plug–in must provide a Java class derived from *netscape.plugin.Plugin* to be LiveConnect compatible. Java and JavaScript can access instances of this class using the document naming scheme of the browser. For example the public method *M* in plug–in applet *P* can be called using the name *document.P.M*. The Java class *NPOberon* implements such an extension of *netscape.plugin.Plugin*. The only method added to the base class is *GetObj*. The purpose of this method is to create a Java proxy object for the Oberon object embedded in the plug–in.

```
import netscape.plugin.Plugin;
import NPOberonObjectProxy;

class NPOberon extends Plugin {
  native public NPOberonObjectProxy GetObj();
}
```

The class *NPOberonObjectProxy* implements the Java proxy object for Oberon objects and provides public methods for communication with the embedded object. Using these methods it is possible to access attributes and links of Oberon objects. When retrieving a link, the result is a new instance of a Java proxy object referring to the linked object. All the public methods are declared as *native*, thus they are implemented in a compiled (native) non–Java language. The native methods are implemented in the Oberon module *NPLiveConnect*. Only the constructor and finalizer methods are implemented in

Java. These are needed to remove proxy object collected by the Java garbage collector from the list of proxies maintained by Oberon (in module *NPLiveConnect*).

```
class NPOberonObjectProxy extends Object {
  native public boolean GetBool(String name);
  native public int GetInt(String name);
  native public float GetReal(String name);
  native public double GetLongReal(String name);
  native public String GetString(String name);

  native public void SetBool(String name, boolean b);
  native public void SetInt(String name, int i);
  native public void SetReal(String name, float x);
  native public void SetLongReal(String name, double y);
  native public void SetString(String name, String s);

  native public NPOberonObjectProxy GetLink(String name);
  native public void SetLink(String name, NPOberonObjectProxy obj);
  native public NPOberonObjectProxy FindObj(String name);

  native public void Update();
  native public void Execute(String cmd);
}
```

6.4.1 Java Runtime Interface

Non–Java applications interoperate with the Java runtime by using the *Java Runtime Interface* (JRI, [Net96]), the standard interface to Java services. The primary goal of the JRI is to allow application programs to be decoupled from the internals of a given Java runtime implementation. In *NPLiveConnect*, the JRI is used to provide Java with the implementations for the native methods in the classes *NPOberon* and *NPOberonObjectProxy*.

The state associated with a Java runtime system is embodied in a Java runtime instance, a structure that contains all the global data necessary to operate a Java interpreter. A runtime instance comprises the heap containing all Java objects, the classes which have been loaded into the runtime, and the mapping from class names to the class objects they denote.

Runtime instances are also used to construct *Java execution environments*. An execution environment contains the state associated with a particular thread of control, or a particular invocation of the

interpreter. This includes the native thread (the plug–in execution stack), the Java execution stack, and all interpreter registers. An execution environment also maintains a table of references. The JRI manipulates all Java objects via *references*. References are opaque tokens for Java objects. Users of the JRI always manipulate Java objects indirectly by passing references to the various JRI operations.

Native methods are callback procedures that are registered with the Java runtime system and called in response to the interpreter invoking a method specified as native. Native method procedures may be found by the Java runtime by using a dynamic linking mechanism, or may be optionally registered explicitly with the runtime via the *JRIRegisterNatives* operation. In the Netscape implementation for Windows, native methods are found by using a fixed naming scheme for the stub procedures exported by the plug–in DLL. For example the implementation for the native method *FindObj* belonging to the class *NPOberonObjectProxy* uses the name:

```
Java_NPOberonObjectProxy_FindObj_stub
```

6.4.2 LiveConnect Implementation

To enable *LiveConnect* for a plug–in, the author of a web page must add the attribute *UseJava* to the *EMBED* tag. If the *UseJava* attribute has a positive value ("Yes", "TRUE", "On"), the browser must be notified of the Java class associated to the plug–in. The plug–in assigns a (JRI) reference to the class in the data structure passed to the plug–in with the *NPGetEntryPoints* call.

JRI uses a data structure similar to the function tables used by COM. This is, the client calls the JRI methods indirectly through an *environment pointer* (see figure 6.2). The first parameter for all JRI methods is the environment pointer, the second parameter is the number of the function entry in the function table.

```
CONST
  FindClassOp = 4;
  NewGlobalRefOp = 10;
  ...

TYPE
  Env = POINTER TO RECORD [NOTAG]
    vtbl: Interface
```

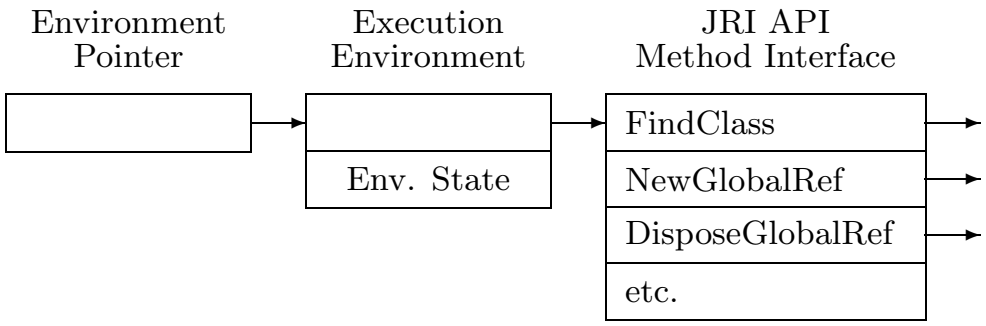

Figure 6.2: Java Execution Environment.

```
  END;

  Interface = POINTER TO RECORD [NOTAG]
    reserved: ARRAY 4 OF LONGINT;
    FindClass: PROCEDURE [C] (env: Env; op: JInt;
      a: ARRAY [NOTAG] OF CHAR): JavaLangClass;
    notimp: ARRAY 5 OF LONGINT;
    NewGlobalRef: PROCEDURE [C] (env: Env; op: JInt;
      a: JRef): JGlobal;
    ...
  END;

VAR
  NPOberonClass: JGlobal;
  idGetPeer: JInt;
  NPOberonObjectProxyClass: JGlobal;
  idNPOberonObjectProxy, idGetProxyID, idSetProxyID: JInt;

PROCEDURE FindClass(env: Env; a: ARRAY OF CHAR): JavaLangClass;
BEGIN
  RETURN env.vtbl.FindClass(env, FindClassOp, a)
END FindClass;

PROCEDURE NewGlobalRef(env: Env; a: JRef): JGlobal;
BEGIN
  RETURN env.vtbl.NewGlobalRefenv, NewGlobalRefOp, a)
END NewGlobalRef;

...
```

When initializing a LiveConnect enabled plug–in instance, *UseJava* is called to initialize the required Java class references. For all methods

required later on in the implemenation of the native methods, the method identifiers (*GetMethodID*) are cached.

```
PROCEDURE InitJava(env: Env);
  VAR class, ref: JRef;
BEGIN
  IF NPOberonClass = NULL THEN
    ref := FindClass(env, "NPOberon");
    NPOberonClass := NewGlobalRef(env, ref);
    class := GetGlobalRef(env, NPOberonClass);
    idGetPeer := GetMethodID(env, GetMethodIDOp, class,
      "getPeer", "()I")
  END;
  IF NPOberonObjectProxyClass = NULL THEN
    ref := FindClass(env, FindClassOp, "NPOberonObjectProxy");
    NPOberonObjectProxyClass := NewGlobalRef(env, ref);
    class := GetGlobalRef(env, NPOberonObjectProxyClass);
    idNPOberonObjectProxy := GetStaticMethodID(env, class,
      "NPOberonObjectProxy", "()V");
    idGetProxyID := GetMethodID(env, class,
      "GetProxyID", "()I");
    idSetProxyID := GetMethodID(env, class,
      "SetProxyID", "(I)V")
  END
END InitJava;

PROCEDURE UseJava();
  VAR env: Env;
BEGIN
  env := NPNGetJavaEnv();
  InitJava(env);
  IF (NPOberonClass # NULL) & (NPOberonObjectProxyClass # NULL) THEN
    SetPlugInJavaClass(NPOberonClass)
  END
END UseJava;
```

When releasing the plug–in library, the class references must be released explicitly. At this stage all Java instances have allready been destroyed, as a plug–in library is only released after all its plug–in instances have been freed.

```
PROCEDURE FreeJava(env: Env);
BEGIN
  IF NPOberonObjectProxyClass # NULL THEN
    DisposeGlobalRef(env, NPOberonObjectProxyClass);
    NPOberonObjectProxyClass := NULL
  END;
```

```
  IF NPOberonClass # NULL THEN
    DisposeGlobalRef(env, NPOberonClass);
    NPOberonClass := NULL
  END
END FreeJava;

PROCEDURE Shutdown(): NPPlugIns.NPError;
  VAR env: Env;
BEGIN
  env := NPNGetJavaEnv();
  FreeJava(env);
  SetPlugInJavaClass(NULL);
  ...
END Shutdown;
```

For each Oberon object instance accessed by Java, a Java proxy object of class *NPOberonObjectProxy* is allocated. Each Java proxy object uses a unique identifier. This identifier is used by the Oberon implementation of the native methods to find the Oberon proxy and object associated to the Java proxy object. When a Java proxy object is no longer needed, it is collected by the Java garbage collector. Oberon is notified of this action by calling the native method *Finalize*.

```
class NPOberonObjectProxy extends Object {
  private int proxyID;

  public void SetProxyID(int proxyID) {
    this.proxyID = proxyID;
  }

  public int GetProxyID() {
    return proxyID;
  }

  public void NPOberonObjectProxy() {
    proxyID = -1;
  }

  native public void Finalize();

  protected void finalize() throws Throwable {
    Finalize();
    proxyID = -1;
    super.finalize();
  }
}
```

```
Proxy = POINTER TO RECORD
  obj: Objects.Object;
  id: LONGINT;
  parent: Proxy
END;
```

All native method implementations use the same procedure interface. The first parameter contains a pointer to the top of the Java stack, and the second parameter contains a pointer to the current execution environment. The native method can access the Java parameters by popping them from the stack. The result is pushed to the stack and the new pointer to the top of the stack is returned.

```
PROCEDURE FindObj(stack: JavaStack; env: Env): JavaStack;
  VAR
    self, ref: JRef; ret: LONGINT;
    args: ARRAY 1 OF LONGINT;
    proxy: Proxy; name: Objects.Name;
    obj: Objects.Object;
BEGIN
  POP(stack, self);
  ret := CallMethod(env, self, idGetProxyID, args, 0);
  proxy := proxies[ret];
  POP(stack, ref);
  CopyString(env, ref, name);
  Oberon.LockOberon();
  obj := Gadgets.FindObj(proxy.obj, name);
  Oberon.UnlockOberon();
  ref := NewNPOberonObjectProxy(env, obj, proxy);
  PUSH(stack, ref);
  RETURN stack
END FindObj;
```

6.4.3 JavaScript Example

The following example illustrates how a standard HTML fill–out form can be used together with JavaScript to control an Oberon plug–in applet. The applet is composed of an Oberon panel named 'Panel' containing a skeleton frame named 'Skeleton'. The purpose of the JavaScript function is to change the color of the skeleton frame.

```
<HTML>
<HEAD>
<TITLE>LiveConnect Example</TITLE>
<SCRIPT LANGUAGE="JavaScript">
```

```
function SetColor(color) {
  obj = document.Panel.GetObj();
  obj = obj.FindObj("Skeleton");
  obj.SetInt("Color", color);
  obj.Update();
}
</SCRIPT>
</HEAD>

<BODY>

<EMBED Name="Panel" SRC="LiveConnect.oaf"
  WIDTH=220 HEIGHT=200 UseJava=Yes>

Modify the "Color" attribute of "Skeleton".
<FORM>
<INPUT TYPE=button VALUE="Red" onClick="SetColor(1)">
<INPUT TYPE=button VALUE="Green" onClick="SetColor(2)">
<INPUT TYPE=button VALUE="Blue" onClick="SetColor(3)">
</FORM>

</BODY>
</HTML>
```

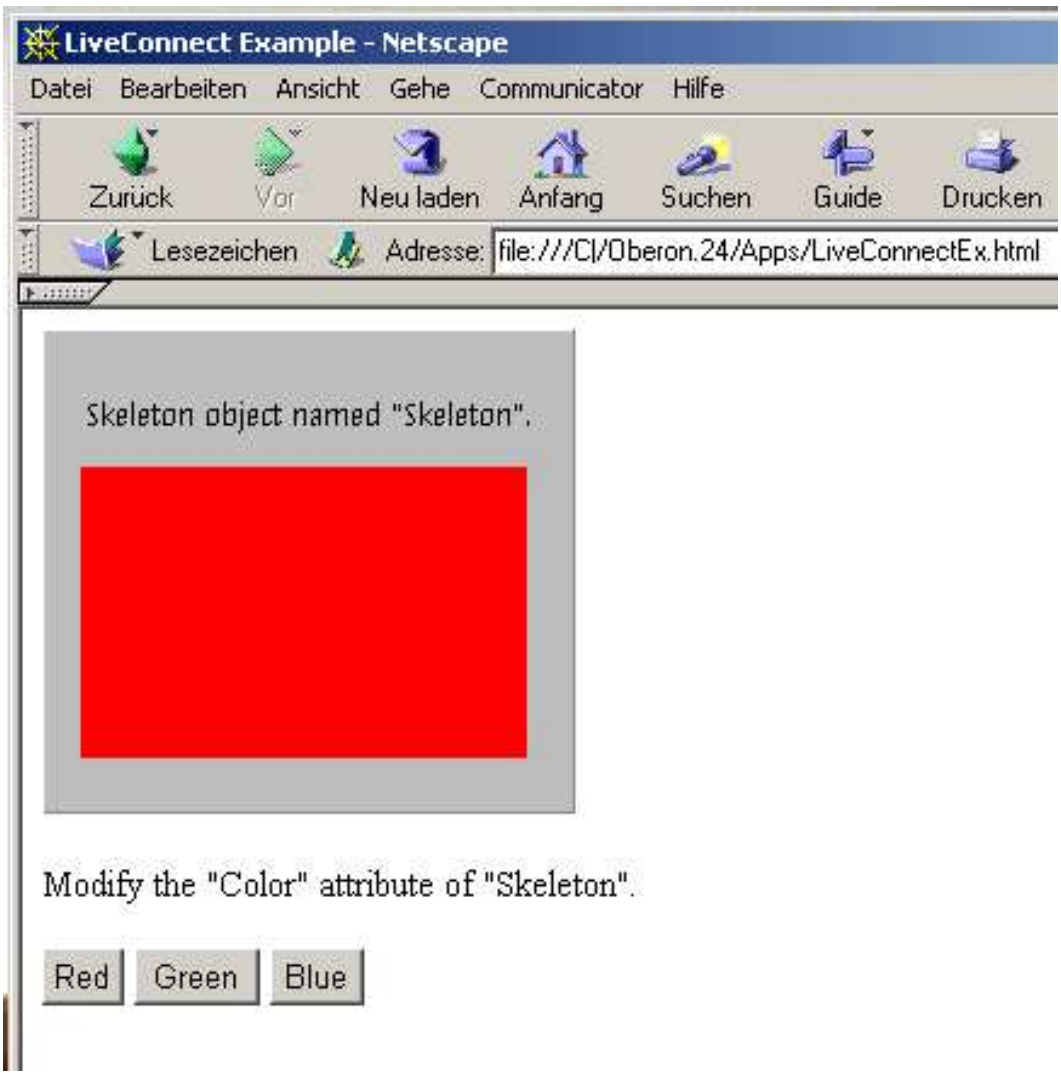

Figure 6.3: Oberon applet, JavaScript and fill–out form communication.

Chapter 7

Oberon ActiveX Components

7.1 Introduction

Object linking and embedding (OLE) is Microsoft's compound document standard. OLE1 was created to migrate existing legacy applications to a single document–centric paradigm. This first version of OLE was implemented on top of DDE (dynamic data exchange). DDE is an interprocess communication mechanism utilizing thread message queues to pass data between threads or processes. This results in a huge communication overhead, specially in an in–process situation, where such a marshalling step is not required. Since the performance and reliability of this initial version was very poor, OLE1 did never gain broad acceptance.

OLE2 is a completely new version of object linking and embedding based on Microsoft's component object model COM (see section 1.3.2). OLE can be summarized as a large collection of predefined COM interfaces. Several of the key technologies required by OLE are delivered by COM services. This includes the following:

- structured storage (compound document files)
- monikers (linking document parts)

- uniform data transfer (clipboard, drag–and–drop)
- connectable objects (events and change notification)
- automation (scripting, properties, and events)

In OLE, a distinction is made between document containers and document servers. A document server provides a content model together with methods to display and manipulate that content. A document container can accept parts provided by arbitrary document servers. Many document containers are also document servers, that is, they have their own native document content but do also support foreign parts. Popular combined servers and containers are the Microsoft Office applications, Word, Excel, and PowerPoint.

Fundamental to the user's illusion of working with a single document is the ability to edit everything where it is displayed. This is called in–place editing. In OLE, an embedded object is activated for in–place editing by double–clicking on it. This action is called in–place activation. As long as an embedded object is not activated, a snapshot of its last known state is displayed. As soon as the object is activated for editing, the objects document server is started to open an embedded editing window. This includes also merging of the menus and toolbars of the document container with the menus and toolbars provided by the embedded document server.

Besides embedding, OLE also supports linking of documents or document parts. Linking is implemented using monikers. The document container stores a moniker to the linked object, this in contrast to embedded objects, where the persistent state is stored in the container. A container can establish an advisory connection with a linked object to be notified when the latter changes. OLE does not allow in–place editing of a linked object. Instead, a fully separate document window is opened to edit a linked object.

Visual Basic controls (VBXs) was the first successful component technology released by Microsoft. Visual Basic uses a simple and fixed model in which controls are embedded into forms. A form binds the embedded controls together and allows the attachment of scripts. There are hundreds of different VBXs, ranging from simple controls to complete applications. The main disadvantages of VBXs are their tight coupling to Visual Basic, and the restrictive form model of Vi-

sual Basic. OLE controls (OCXs) were introduced to migrate the VBX concept to a more powerful platform, that of general OLE containers.

To qualify as an OLE control, a COM object has to implement a large number of interfaces. An OCX implements all of an OLE document server's interfaces, plus a few more to emit events. OCXs can be used in any OLE document container, as they implement a full–featured document server. An OCX–aware container provides extra features like scripting when embedding such a control server. The downside of such a full–featured control server is, that even the most minimal control has to carry much baggage to qualify as OCX. This is particularly cumbersome when downloading components across the Internet.

Recently, OLE controls were renamed to 'ActiveX Controls'. 'ActiveX Controls' is not just a new name for OLE controls, but it is also a new revised specification. The minimal requirements for an ActiveX control are, that it is implemented by a self–registering server and the it implementats the interface *IUnknown*. Self–registering allows a server, when started and asked to do so, to register its class with the COM registry. This is useful when a server's code has been downloaded from the Internet. All other features defined in the ActiveX specification are optional. With ActiveX, it is more difficult to implement a useful container. A container can not rely on anything but the implementation of *IUnknown* when interacting with an embedded (ActiveX) control. Hence it has to inspect at run–time which interfaces are supported by a control and to react accordingly. This is why many so–called ActiveX containers function only correctly when the embedded controls implement the full set of OLE control interfaces.

Due to the complexity of the OLE/ActiveX framework, new servers (controls) and clients (containers) are typically implemented using some supporting tool. Prominent examples of such aiding tools are Microsoft's MFC (Microsoft Foundation Classes) and ATL (Active Template Library). Both provide interactive wizards which aid the programmer in the implementation of a component object. After this interactive design phase the wizards generate the source code for a complete component and the programmer often needs only to fill–in gaps with some glue logic code. In Oberon's OLE/ActiveX implementation, a completely different and more transparent approach has been chosen: architecture bridges (or wrappers), similar to Java Beans

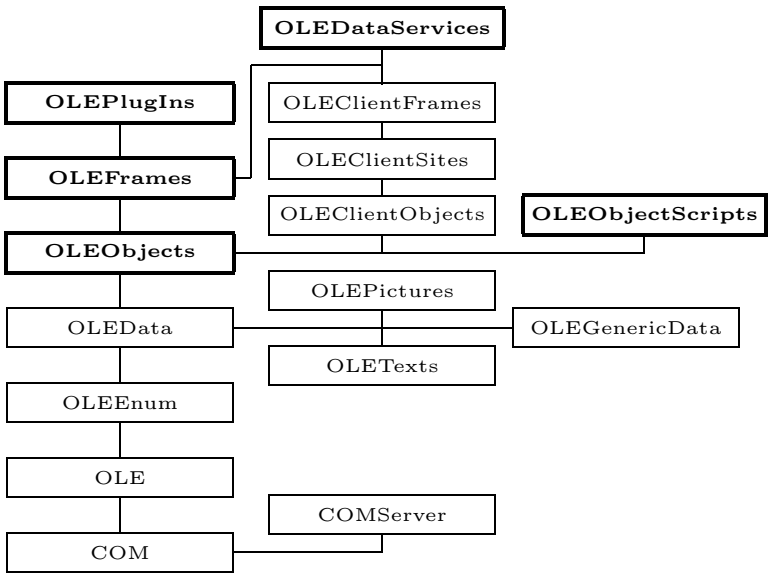

Figure 7.1: Oberon OLE framework: module hierarchy.

Architecture Bridge for ActiveX [Sun99a], are provided. A programmer of an Oberon component does not need to care about OLE or ActiveX at all. The finished Oberon component implementation can just be plugged into the appropriate wrapper component. The Oberon wrappers are completely generic, thus new Oberon components can be plugged into such a wrapper without the need to compile any generated source code or to register new COM classes.

In the following sections of this chapter the implementation of Oberon's OLE framework is discussed in more details. Figure 7.1 gives an overview of its module hierarchy.

COM This module provides the basic types and APIs needed for implementing COM (OLE, ActiveX) object classes and implements a generic in–process multi–class server. Concrete class implementations can be plugged into this generic server at run–time. Module *COMServer* implements an out–process server providing remote access to any object class hosted by the generic in–process server.

OLE This module defines all the standard OLE interfaces and miscellaneous OLE data structures needed by the Oberon *OLE** modules. Module *OLEEnum* provides a small framework for the

implementation of *IEnumX* interfaces on arbitrary linear data structures.

OLEData This module provides a small framework to convert between Oberon objects and the different data types defined for uniform data transfer in COM. *OLETexts* implements conversion routines for different text formats and *OLEPictures* implements conversion routines for different image formats. For unsupported formats *OLEGenericData* can be used to transfer the raw binary data.

OLEObjects This module implements a non–visual ActiveX wrapper for arbitrary Oberon objects. This wrapper provides standard services like: persistency, scripting and uniform data transfer.

OLEObjectScripts This module implements an ActiveX scripting engine for scriptlets. Any ActiveX scripting host can be used to attach (Oberon) scripts to COM objects based on *OLEObjects*.

OLEClientX The *OLEClientX* modules implement an OLE container for arbitrary linked OLE objects. *OLEClientObjects* handles the creation of linked objects, typically as a result of a uniform data transfer transaction. *OLEClientSites* provides a host environment for the linked objects and *OLEClientFrames* implements an Oberon frame for visual manipulation of linked objects.

OLEDataServices This module implements the UI for uniform data transfer transactions, these are: clipboard, drag–and–drop, object linking, and document files.

OLEFrames This module implements a visual ActiveX wrapper for arbitrary Oberon frames. In addition to the services already provided by *OLEObjects*, this object adds support for in–place activation and in–place editing, thus providing a full featured ActiveX and OLE control

OLEPlugIns Extends the *OLEFrames* wrapper with the ability to load the embedded Oberon frame from an asynchronous internet connection. The same download and install mechanism as discussed in section 5.2.1 is used for this ActiveX–based browser plug–in (Internet Explorer).

7.2 An ActiveX and DCOM Server

COM uses an elaborate interface concept in order to decouple component implementations from clients. The main aspects which must be considered when adding COM support to a new programming language are:

1. IDL (interface description language) specifications must be translated into the new programming language.
2. Multiple interfaces per object and reference counting as defined by COM must be modelled appropriately.

IDL is a notation used in language independent frameworks like CORBA or COM to describe interfaces formally. Although the COM IDL is language–independent, the data types used are those of the C programming language. When translating IDL specifications to Oberon, C data types are mapped to Oberon data types as described in section 4.3.

A second issue to be dealt with is the implementation of multiple interfaces and the binding of method procedures to interfaces. In languages like C++ or Java, multiple interfaces and binding of methods can be modelled using multiple inheritance of abstract base classes. As (standard) Oberon does neither feature multiple inheritance nor methods, a completely different approach has been chosen. Explicit interface objects are used to provide multiple interfaces per object instance. The disadvantage of this approach is, that VTBLs and interface nodes must be filled–in explicitly by the programmer. On the other hand, this solution provides great flexibility. For example, it is possible to exchange method tables at run–time on a per instance base. This feature is specially useful when one wants to implement a different behavior of the same object depending on its context.

A client of a COM object never has direct access to the object itself. All accesses to the object are provided through a pointer to an interface node (see figure 7.2). The first field of an interface node points to the function table containing all the functions (or methods) for that interface. The pointer to the interface node is passed on all method invokations as *this* parameter implicitely.

In Oberon, interface nodes are represented by variables of type *Interface*. Interfaces provided by Oberon COM objects use the extended

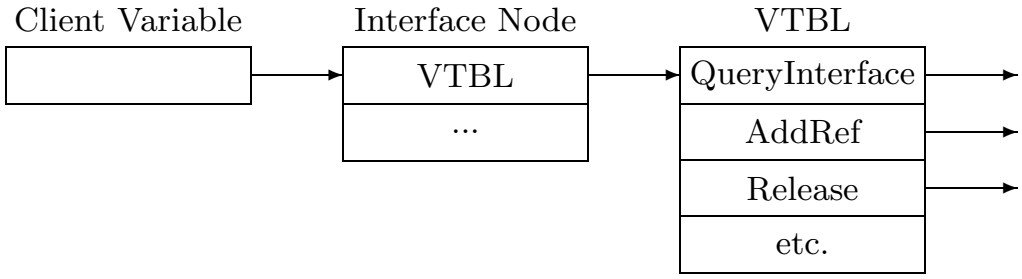

Figure 7.2: COM interface and virtual function table.

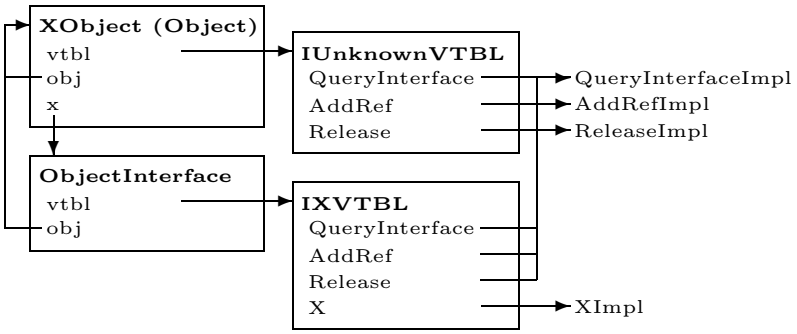

Figure 7.3: Memory layout of an Oberon COM object.

type *ObjectInterface*, as the *obj* field is needed to get the pointer to the object instance from the *this* pointer passed on any method invokation. Actual COM objects are of type *Object* or an extension thereof. As this type is an extension of *Interface*, a primary interface can be provided without the need for additional interface nodes. For additional interfaces however, extra interface nodes of type *ObjectInterface* have to be allocated.

```
MODULE COM;

  Interface = POINTER TO RECORD
    vtbl: IUnknownVTBL (* virtual function table *)
  END;

  IUnknownVTBL = POINTER TO RECORD
    QueryInterface: PROCEDURE [WINAPI] (this: Interface;
      VAR riid: IID; VAR ppvObject: Interface): HRESULT;
    AddRef: PROCEDURE [WINAPI] (this: Interface): LONGINT;
    Release: PROCEDURE [WINAPI] (this: Interface): LONGINT
```

```
  END;

  ObjectInterface = POINTER TO RECORD (Interface)
    obj: Object
  END;

  Object = POINTER TO RECORD (ObjectInterface)
    refCount: LONGINT;
    class: Class; next: Object
  END;
```

In the COM model, a class factory object is required for creating new object instances. Each class is identified by a unique class identifier (CLSID). This identifier is used by the COM system libraries as a key into the registry to retrieve information about a class and its server. Once the server is loaded, a pointer to the class factory's implementation of *IClassFactory* can be requested. The latter is subsequently used to create new object instances by calling the *CreateInstance* method. To get the class object for an in–process COM server, the DLL entry point *DllGetClassObject* must be called. For out–process COM servers, a new process must be started from the server's EXE file. Once the server is running, it will register its class object with the COM system. Normal COM clients do not need to know all these server–specific details. The COM library routine *CoCreateInstance* can be used to create object instances by a simple procedure call, independent of their server type.

In the Oberon COM framework, module *COM* implements a generic ActiveX in–process server. This server is generic in that concrete class implementations can be added dynamically to the server at run–time, without the need to rebuild the server DLL (Oberon.DLL). When implementing a new class, the new CLSID together with a generator command are registered in the Oberon registry (Oberon.Text). When the DLL entry point *DllGetClassObject* is called, the requested CLSID is located in the Oberon registry and the associated generator command is called. This generator registers a new class object (of type *Class*) with module *COM* by calling *RegisterClass*.

```
MODULE COM;

  IClassFactoryVTBL = POINTER TO RECORD (IUnknownVTBL)
    CreateInstance: PROCEDURE [WINAPI] (this: Interface;
      pUnkOuter: Interface; VAR riid: IID;
```

```
    VAR ppvObject: Interface): HRESULT;
  LockServer: PROCEDURE [WINAPI] (this: Interface;
    fLock: Kernel32.BOOL): HRESULT
END;

ClassProc = PROCEDURE (class: Class): HRESULT;
CreateObjectProc = PROCEDURE (class: Class; data: PTR): Object;
ReleaseObjectProc = PROCEDURE (class: Class; obj: Object);

Class = POINTER TO RECORD (Interface)
  clsid: CLSID; objs: Object;
  refCount, locks: LONGINT;
  RegisterServer, UnregisterServer, CanUnloadNow: ClassProc;
  CreateObject: CreateObjectProc;
  ReleaseObject: ReleaseObjectProc;
  apartment: Threads.Thread
END;

PROCEDURE RegisterClass(class: Class);

PROCEDURE [WINAPI] DllGetClassObject(VAR rclsid: CLSID;
  VAR riid: IID; VAR ppv: Interface): HRESULT;

PROCEDURE [WINAPI] DllRegisterServer(): HRESULT;

PROCEDURE [WINAPI] DllUnregisterServer(): HRESULT;

PROCEDURE [WINAPI] DllCanUnloadNow(): HRESULT;
```

When the Oberon COM server (Oberon.DLL) is registered or unregistered with a standard tool like *regsvr32*, all classes known to Oberon (entries in Oberon.Text) are registered or unregistered. To register individual COM classes, the Oberon specific command *COM.RegisterDLL* must be used.

As COM objects provided by Oberon may be in use by a client but no longer referenced by any Oberon data structure, each class object (of type *Class*) maintains a list of all available object instance of that class. As soon as the COM reference count reaches zero, the object is removed from its class list and thus can be collected by the Oberon garbage collector. Garbage collection is further complicated because an in–process COM object runs in a thread controlled by its client (and not its server). Upon calling *DllGetClassObject*, the client's thread (apartment thread) is therefore registered with the Oberon garbage collector.

In–process servers are best suited for applications where the server and the client have to share data structures in memory. An example of such a data structure is a client's graphics context. A server providing in–place editing or activation must have direct access to the graphics context for display output. An out–process server would not work in this scenario since the client's graphics context would be in the wrong address space. Out–process servers are needed for complex applications where it is not feasible to have separate instances for each client. Possible reasons for using an out–process server are:

- A legacy application is used as server.
- The server runs on a remote system.
- The design of the server application does not allow running multiple instances of the server.

The module *COMServer* implements an out–process server on top of the generic in–process server described in the previous sections. The server provides two modes of operation:

1. It can be started automatically by the COM system when a client requests an out–process server for a class registered by Oberon. In this case, the server runs as hidden (thus no UI) process under the control of the COM system, this mode is called *embedding*.
2. The server can be started ad hoc from within a running Oberon system.

The main difference between an in–process and an out–process server is, that in the latter case methods can no longer be called directly. Since the server and client run in different processes or even on different machines, the actual parameters and return value of a method called, must be assembled into a standard format that both processes understand. This process is called marshalling (sender) and unmarshalling (receiver). For most standard interfaces marshalling is done by the COM system, thus server and client do not need to care about cross–process or cross–machine communication. This is accomplished by introducing in–process objects *proxy* and *stub* which emulate the remote server (proxy) or client (stub).

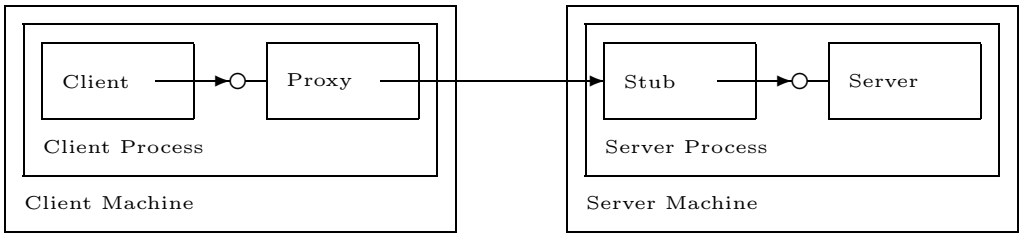

Figure 7.4: Remote method invokation using proxy and stub.

7.3 OLEObjects

Module *OLEObjects* implements a non–visual ActiveX control container for arbitrary Oberon objects. This control is best described as an architecture bridge filling the gap between an ActiveX client and the Oberon server. The bridge is transparent to both the server and the client; that is, any combination of client and server will immediately work without the need to create or register additional helper objects. The features of this 'bridge' control are best explained by describing its interfaces.

7.3.1 IUnknown

IUnknown implements garbage collection using a reference count and *QueryInterface* for requesting pointers to all additional interfaces supported by the control. The control allows two different methods of reuse:

1. The most common reuse mechanism in COM is containment or delegation. This is the normal way for an outer object (container, client) to contain an inner object (control, server). But this method can as well be used for implementation inheritance.

2. Aggregation is another reuse mechanism, in which the outer object exposes interfaces from the inner object as if they were implemented on the outer object itself. This is useful when the outer object would always delegate every call to one of its interfaces to the same interface in the inner object.

For aggregation to work properly, the inner object must explicitly support this feature. That is, the *IUnknown* methods of any of

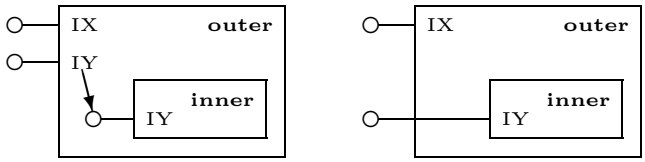

Figure 7.5: Containment vs. Aggregation.

the inner objects interfaces must be delegated to the outer objects *IUnknown* methods. The outer object's *IUnknown* interface must be passed to the inner object on creation, using the *pUnkOuter* parameter in *IClassFactory.CreateInstance*. In the Oberon implementation each COM object is equipped with two implementations of *IUnknown*, a non–delegating variant and a delegating variant. Thus delegation (to the outer object's *IUnknown* interface) is used here to implement aggregation.

```
TYPE
  Object = POINTER TO RECORD (COM.Object)
    ...
    (* non-delegation IUnknown *)
    unknown: ObjectInterface;
    (* outer IUnknown *)
    unkOuter[UNTRACED]: Interface;
    ...
  END;

(* delegating implementation of QueryInterface. *)
PROCEDURE [WINAPI] QueryInterface(this: Interface;
  VAR riid: IID; VAR ppvObject: Interface): HRESULT;
  VAR obj: Object;
BEGIN
  obj := this(ObjectInterface).obj(Object);
  RETURN obj.unkOuter.vtbl.QueryInterface(obj.unkOuter,
      riid, ppvObject)
END QueryInterface;

(* non-delegating implementation of QueryInterface. *)
PROCEDURE [WINAPI] NDQueryInterface(this: Interface;
  VAR riid: IID; VAR ppvObject: Interface): HRESULT;
  VAR obj: Object;
BEGIN
  obj := this(ObjectInterface).obj(Object);
  IF IsEqualGUID(riid, IIDIUnknown) THEN
    OutInterface(ppvObject, obj.unknown)
```

```
  ELSIF IsEqualGUID(riid, IIDIOberonOleObject) THEN
    OutInterface(ppvObject, obj)
  ELSIF IsEqualGUID(riid, IIDIPersistStreamInit) THEN
    OutInterface(ppvObject, obj.persistStreamInit)
  ...
END NDQueryInterface;

PROCEDURE [WINAPI] CreateInstance(this: Interface;
  pUnkOuter: Interface;
  VAR riid: IID; VAR ppvObject: Interface): HRESULT;
  VAR obj: Object;
BEGIN
  obj := CreateObject(this(Class));
  IF pUnkOuter # NIL THEN
    obj.unkOuter := pUnkOuter
  ELSE
    obj.unkOuter := obj.unknown
  END;
  RETURN NDQueryInterface(obj, riid, ppvObject)
END CreateInstance;
```

7.3.2 IDispatch

The *IDispatch* interface provides methods to enumerate and manipulate the features (properties, methods and events) of a control at run–time. The main application of *IDispatch* are scripting languages like VBScript or JavaScript, where the actual type and features of an object are checked at run–time only (late binding). The features exported by a control via its *dispatch interface* are typically described in a type library, a binary file compiled from an IDL description file. Using this information, a client may as well statically check the access to a control (early binding). For this purpose, the control in addition often implements a *custom interface* providing direct access to its features. When a control provides both a dispatch interface and a custom interface, the two interfaces are called *dual interfaces*.

The Oberon ActiveX wrapper control is described in the IDL files 'OberonOleObject.idl' and 'IOberonOleObject.idl'. The first IDL file describes the Oberon COM class *OberonOleObject* and its type library *OberonOleObjectLib*.

OberonOleObject.idl:

```
import "oaidl.idl";
```

```
import "ocidl.idl";
import "IOberonOleObject.idl";

[ uuid(91C4D7F1-A0E8-11D3-A2BE-005004435167),
  version(1.0),
  helpstring("ETH Oberon, OLE Objects 1.0 Type Library") ]

library OberonOleObjectLib {
  importlib("stdole32.tlb");
  importlib("stdole2.tlb");

  [ uuid(91C4D7F0-A0E8-11D3-A2BE-005004435167),
    helpstring("ETH Oberon, OLE Objects Class") ]

  coclass OberonOleObject {
    [ default ] interface IOberonOleObject;
    [ default, source ] dispinterface IOberonOleEvent;
  };
};
```

The second IDL file describes the dual interface *IOberonOleObject*.

IOberonOleObject.idl:

```
import "oaidl.idl";
import "ocidl.idl";

[ object, dual,
  uuid(91C4D7F2-A0E8-11D3-A2BE-005004435167),
  helpstring("ETH Oberon, OLE Objects Interface") ]
interface IOberonOleObject : IDispatch {
  [propget] HRESULT Gen([out, retval] BSTR* gen);
  [propget] HRESULT Name([out, retval] BSTR* name);
  [propput] HRESULT Name([in] BSTR name);

  HRESULT GetBool([in] BSTR name, [out, retval] BOOL* b);
  HRESULT GetInt([in] BSTR name, [out, retval] int* i);
  HRESULT GetReal([in] BSTR name, [out, retval] float* x);
  HRESULT GetLongReal([in] BSTR name, [out, retval] double* y);
  HRESULT GetString([in] BSTR name, [out, retval] BSTR* s);

  HRESULT SetBool([in] BSTR name, [in] BOOL b);
  HRESULT SetInt([in] BSTR name, [in] int i);
  HRESULT SetReal([in] BSTR name, [in] float x);
  HRESULT SetLongReal([in] BSTR name, [in] double y);
  HRESULT SetString([in] BSTR name, [in] BSTR s);

  HRESULT GetLink([in] BSTR name,
```

```
      [out, retval] IOberonOleObject** obj);
  HRESULT SetLink([in] BSTR name,
      [in] IOberonOleObject* obj);
  HRESULT FindObj([in] BSTR name,
      [out, retval] IOberonOleObject** obj);

  HRESULT Update();
  HRESULT Execute([in] BSTR cmd);
};
```

The *Get* and *Set* method collections provide access to the attributes and links of the embedded Oberon object. These methods correspond to the *Get* and *Set* procedures provided by modules Attributes and Links. *FindObj* searches for an object named 'name' inside a container object (e.g. a Panel). All Oberon attribute types can be mapped to matching IDL types. In methods with an object type parameter such as *GetLink*, *SetLink* and *FindObj*, the Oberon object type is mapped to an *IOberonOleObject* interface pointer to that object. Behind the scene, an *OLEObjects* or *OLEFrames* wrapper control is generated for any Oberon object reference passed to a COM client.

As the *IOberonOleObject* methods are called in the thread apartment of the client, all accesses to the embedded Oberon object must be synchronized with the thread running the Oberon loop. This is done using the *LockOberon* and *UnlockOberon* calls. The code fragment below shows how the *GetString* method is implemented in module *OLEObjects*.

```
TYPE
  Object = POINTER TO RECORD (COM.Object)
    ...
    object: Objects.Object; (* contained object *)
    ...
  END;

PROCEDURE [WINAPI] *GetString(this: Interface;
    name: BStr; VAR s: BStr): HRESULT;
  VAR obj: Object; oname, str: Objects.Name;
BEGIN
  obj := this(ObjectInterface).obj(Object);
  IF obj.object # NIL THEN
    FromBStr(name, oname);
    Oberon.LockOberon();
    Attributes.GetString(obj.object, oname, str);
    Oberon.UnlockOberon();
```

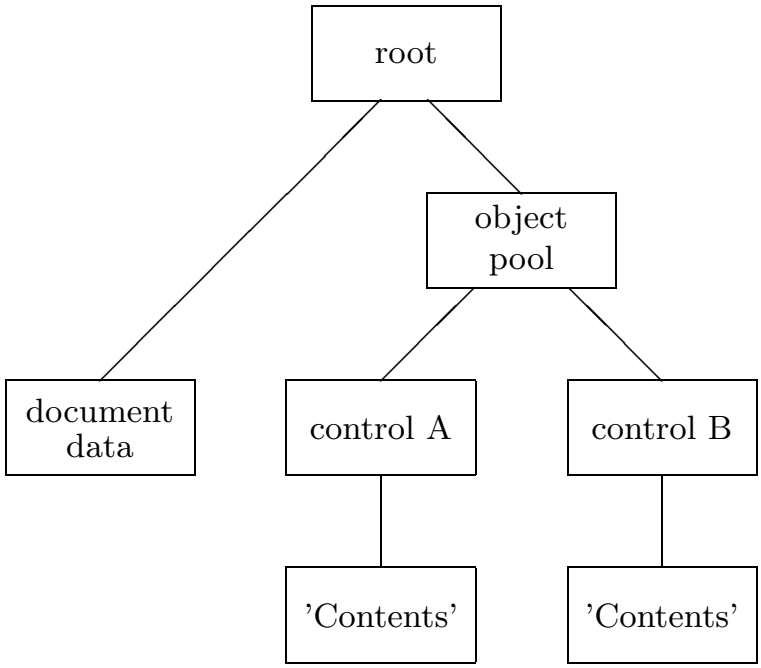

Figure 7.6: Compound document file using Structured Storage.

```
    AllocBStr(str, s);
    RETURN SOk
  END;
  RETURN EUnexpected
END GetString;
```

7.3.3 IPersist

The purpose of the *IPersist* interfaces is to save and load the state of an object. COM defines three different variants of persistency interfaces. These are:

1. The *IPersistStream* interface provides methods for saving and loading objects to and from a sequential byte stream. To save an Oberon object to a COM stream, the object is first stored in an anonymous file using the Oberon object library mechanism, the contents of that file is then copied to the COM stream. To load an Oberon object from a COM stream, the stream is first copied to an anonymous file from where the object is loaded using the Oberon library mechanism.

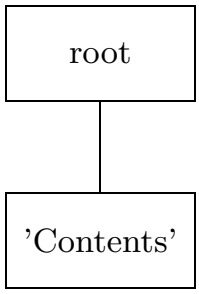

Figure 7.7: Single object compound file using Structured Storage.

2. The interface *IPersistStorage* is based on structured storage (see figure 7.6), the OLE technology for storing compound files in native file systems. A compound file consists of storages and streams, these objects are analogous to the directories (Storage) and files (Stream) of common hierarchic file systems. Each contained object has its own storage that is nested within the storage of the container. In Oberon, *IPersistStorage* is implemented as a stream named 'Contents' allocated in the object's storage and the object's data is stored to this stream by calling the *IPersistStream* methods. This is the standard method used by most non–container OLE controls.

3. The *IPersistFile* interface provides methods for loading and saving an object directly from and to a compound file. For non–container controls like the Oberon wrappers, the objects data can be saved and loaded by calling the *IPersistStorage* methods on the root storage object of the compound file (see figure 7.7).

7.3.4 IDataObject

The *IDataObject* interface specifies methods enabling data transfer, and notification of changes to data. Data transfer methods specify the format of the transferred data along with the medium through which the data is to be transferred. Optionally, the data can be rendered for a specific target device. In addition to methods for retrieving and storing data, the *IDataObject* interface specifies methods for enumerating available formats and managing connections to advisory sinks for handling change notifications.

Module *OLEData* provides a generic implementation of *IDataObject*. That is, concrete implementations of converters between OLE data formats and Oberon data formats can be registered and unregistered with *OLEData* as needed. An OLE data item consists of two data structures, the first *FormatEtc* describes the kind of data (e.g. bitmap) together with the aspects or views available on that data (e.g. full contents, preview, etc.). The second data structure *StgMedium* describes the storage medium where the data is stored (e.g. a file name). An Oberon data item consists of a single (model) object which encapsulates the data (e.g. Texts.Text or Pictures.Picture).

A filter for converting between the OLE and Oberon data formats consists of the three methods *get*, *set* and *check*. *get* is called when converting data from Oberon to OLE, *set* is called for converting data from OLE to Oberon and *check* is used to test whether a given filter is able to perform a given conversion or not.

```
Filter = POINTER TO RECORD
  get: PROCEDURE (this: Interface; obj: Objects.Object;
    pformatetc: FormatEtc; VAR pmedium: StgMedium): HRESULT;
  set: PROCEDURE (this: Interface; VAR obj: Objects.Object;
    pformatetc: FormatEtc; pmedium: StgMedium): HRESULT;
  check: PROCEDURE (this: Interface; obj: Objects.Object;
    dwDirection: LONGINT; pformatetc: FormatEtc): HRESULT;
   next: Filter
END;
```

All the data format related methods in *IDataObject* use the *check* method to find a filter capable of performing the desired action. The *get* and *set* methods are then used to transfer the data from one format to another. The implementation of *IDataObject.QueryGetData* shown below determines whether the data object is capable of rendering the data described in the *FormatEtc* structure.

```
PROCEDURE CheckFormat(this: Interface; obj: Objects.Object;
  dwDirection: LONGINT; pformatetc: FormatEtc): HRESULT;
  VAR filter: Filter;
BEGIN
  filter := filters;
  WHILE (filter # NIL) &
    (filter.check(this, obj, dwDirection, pformatetc) # SOk) DO
    filter := filter.next
  END;
  IF filter # NIL THEN
```

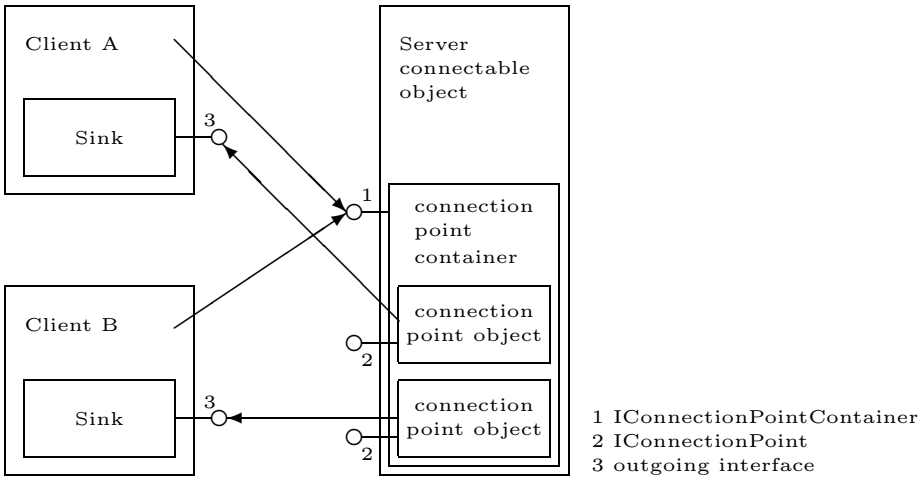

Figure 7.8: A server connected with two clients.

```
    RETURN SOk
  ELSIF obj = NIL THEN
    RETURN OLEENotRunning
  ELSE
    RETURN DVEClipformat
  END
END CheckFormat;

PROCEDURE [WINAPI] QueryGetData(this: Interface;
  pformatetc: FormatEtc): HRESULT;
  VAR obj: Object;
BEGIN
  obj := this(ObjectInterface).obj(Object);
  RETURN CheckFormat(this, obj.object, DataDirGet, pformatetc)
END QueryGetData;
```

7.3.5 Events

The natural flow of communication is from the client to the server, where the client invokes methods on interfaces provided by the server. There are however situations where the server has to call back on a client. The *IDataObject* provides such a callback facility, which however is specialized for change notification on data objects. A general mechanism for a server to signal any kind of events to clients is needed. A typical setup where such events are used, is a GUI control (e.g. a push button) which triggers some scripting procedures in the client

(e.g. a Visual Basic form).

The *Connectable Objects* technology provides a much more general two–way communication. If a server object needs to callback to its client, the interfaces defined by Connectable Objects offer a standard way to establish a logical connection between the object and its client, no matter what interface the object uses. Because all that is specified is the mechanics of setting up and tearing down this relationship, the *Connectable Objects* interfaces can be used for all kinds of two–way communication.

To talk back to its client, an object must support one or more *outgoing* interfaces. Supporting an outgoing interface simply means that the object knows how to act as a client for that interface. That same interface is an incoming interface from the point of view of the client that implements it. An object can describe its outgoing interfaces using IDL and can store descriptions of outgoing interfaces in a type library.

To qualify as a connectable object, an object must support an interface called *IConnectionPointContainer*. Through this interface, the client of an object can learn which outgoing interfaces the object supports. Each of those interfaces is represented within the object by a separate connection point object. Each connection point handles only one type of outgoing interface, and each supports (at least) an interface called *IConnectionPoint*. Finally, given that a connectable object supports an outgoing interface and thus knows how to invoke the methods of the interface, some other object must effectively implement this interface. This other object is called a *sink* and is part of the connectable object's client.

Outgoing interfaces can be implemented either as dispatch interfaces or as custom interfaces. Custom interfaces can be implemented much more efficiently than dispatch interfaces, as the latter require explicit dispatching at run–time, whereas methods on custom interfaces can be called directly. The problem with custom interfaces is, that a client must explicitly implement that specific interface, whereas a dispatch interface can easily be synthesized at run–time. The Oberon ActiveX implementation provides an event interface *IOberonOleEvent* togethter with a connectable object, as described in 'IOberonOleObject.idl'. This event interface features notification methods for all important state changes in the server object.

IOberonOleObject.idl:

```
[ uuid(91C4D7F3-A0E8-11D3-A2BE-005004435167),
  helpstring("ETH Oberon, OLE Objects Event Interface") ]
dispinterface IOberonOleEvent {
  methods:
    [id(1)] void OnUpdate();
    [id(2)] void OnExecute();
    [id(3)] void OnLoad();
    [id(4)] void OnSave();
    [id(5)] void OnInitNew();
    [id(6)] void OnChangeContent();
    [id(7)] void OnClose();
    [id(8)] void OnActivate();
    [id(9)] void OnDeactivate();
};
```

The next code fragment shows how an outgoing (dispatch) interface is called. With controls, this is usually called *firing an event*. As the implementations supports different outgoing (dispatch) interfaces, the IID is explicitly checked against the IID of the desired interface. The id parameter passed to the *FireEvent* procedure is one of the id values defined in the above IDL specification.

```
TYPE
  Connection = POINTER TO RECORD
    dispatch[UNTRACED]: DispatchInterface;
    iid: IID;
    next: Connection
  END;

PROCEDURE FireOberonEvent(obj: Object; id: LONGINT);
  VAR c: Connection; disp: DispParam; err: LONGINT; hr: HRESULT;
BEGIN
  c := obj.connections;
  WHILE c # NIL DO
    IF IsEqualGUID(c.iid, IIDIOberonOleEvent) THEN
      disp.rgdispidNamedArgs := NIL; disp.cNamedArgs := 0;
      disp.cArgs := 0; disp.rgvarg := NIL;
      hr := c.dispatch.vtbl.Invoke(c.dispatch, id, GUIDNULL,
        LocaleNeutral, DispatchMethod, disp, NIL, NIL, err)
    END;
    c := c.next
  END
END FireOberonEvent;
```

7.4 OLEFrames

Module *OLEFrames* extends the non–visual ActiveX control (*OLEObjects*) with the interfaces needed by a visual ActiveX control. The most important extensions are for supporting in–place activation and in–place editing. As the method tables for the different interfaces are allocated explicitly, inheritance must be programmed explicitly as well. Thus, a copy of the original method table is allocated and extended methods are assigned as needed. Upcalls can be simulated by explicitly calling the corresponding method in the original method table.

```
MODULE OLEFrames;

  TYPE
    Object = POINTER TO RECORD (OLEObjects.Object)
      ...
    END;

PROCEDURE [WINAPI] StrInitNew(this: Interface): HRESULT;
  VAR obj: Object; hr: HRESULT;
BEGIN
  hr := OLEObjects.persistStreamInitVTBL.InitNew(this);
  IF COM.Succeeded(hr) THEN
    obj := this(ObjectInterface).obj(Object);
    ...
  ELSE
    RETURN hr
  END
END StrInitNew;

...

NEW(persistStreamInitVTBL);
persistStreamInitVTBL^ := OLEObjects.persistStreamInitVTBL^;
persistStreamInitVTBL.InitNew := StrInitNew;
```

As the interfaces already implemented by *OLEObjects* need no or only small changes, only the additional interfaces introduced by *OLEFrames* are discussed here.

7.4.1 IOleObject

The *IOleObject* interface is the principal means by which an embedded object provides basic functionality to, and communicates with, its

container. The *IOleObject* interface encompasses over twenty methods, but the implementation of most of them is trivial. The basic OLE concepts implemented by these methods are:

Client Site Within a compound document, each embedded object has its own client site — the place where it is displayed and through which it receives information about its storage, user interface, and other resources.

Advisory Connection A container establishes with each embedded object an advisory connection, through which the object can inform its container's advise sink of close, save, rename, and link–source change events in the object. The advisory connection methods are best implemented by delegating to an advise holder object provided by the OLE libraries through a call to *OleCreateAdviseHolder*.

Verbs Request an object to perform an action in response to an end–user's action. Each action is assigned a different verb value. OLE defines verbs for the different levels of in–place activation, but an object may provide additional verbs to its container through an *enumerate verbs* method.

Pseudo–code implementations for in–place activation as well as deactivation of an embedded object appear below. The major problem with implementing in–place activation is, that there are five different instances which must be notified in the correct order. The individual methods and interfaces involved in in–place activation are well documented in the OLE specifications. But there is no detailed discussion on how to use all these interfaces together.

In the Oberon implementation, the following interfaces provided by the container are used:

- **clientSite** — *IOleClientSite*
- **inPlaceSite** — *IOleInPlaceSite* temporary client site object for an in–place active object.
- **frame** — *IOleInPlaceFrame* pointer to the container frame window containing the embedded object. It is used to negotiate border space on the document or frame window.

- **win** — *IOleInPlaceUIWindow* pointer to the containers document window containing the UI of the document. It is used to negotiate merging of the menus and toolbars.
- **adviseHolder** — *IOleAdviseHolder* helper object for maintaining advisory connections.
- **adviseSink** — *IAdviseSink* advise sink for notifying a container of view changes (see: *IViewObject*).

together with the following miscellaneous objects:

- **obj.hwnd** — window handle of the embedded objects control window.
- **pos** — position and size of the control window.
- **clip** — visible region of the control window.

```
PROCEDURE Activate(obj: Object);
  VAR site: Interface;
BEGIN
  obj.clientSite.QueryInterface(IIDIOleInPlaceSite, site);
  IF site.CanInPlaceActivate() &
     site.OnInPlaceActivate() THEN
    ActivateWindow(obj, site); obj.inPlaceSite := site;
    FireOberonEvent(obj, IDActivated)
  END
END Activate;

PROCEDURE ActivateWindow(obj: Object; site: Interface);
BEGIN
  site.GetWindow(hwnd);
  site.GetWindowContext(frame, win, pos, clip, info);
  IF obj.hwnd = NIL THEN
    CreateWindow(obj, hwnd, pos);
    frame.SetActiveObject(obj); win.SetActiveObject(obj);
    frame.SetBorderSpace( NIL); win.SetBorderSpace(NIL);
    IF ~obj.inPlaceActive THEN
      ShowWindow(obj.hwnd, SWShow);
      obj.clientSiteOnShowWindow();
      obj.inPlaceActive := TRUE
    END;
    SetObjectRects(obj, pos, clip);
    site.ShowObject();
    SetFocus(obj.hwnd)
  END
```

```
END ActivateWindow;

PROCEDURE Deactivate(obj: Object);
  VAR site: Interface;
BEGIN
  IF obj.inPlaceActive THEN
    site := obj.inPlaceSite;
    site.GetWindow(hwnd);
    site.GetWindowContext(frame, win, pos, clip, info);
    frame.SetActiveObject(NIL); win.SetActiveObject(NIL);
    site.OnUIDeactivate(); obj.inPlaceActive := FALSE;
    site.OnInPlaceDeactivate();
    ShowWindow(obj.hwnd, SWHide);
    FireOberonEvent(obj, IDDeactivated)
  END
END Deactivate;

PROCEDURE Close(obj: Object);
BEGIN
  Deactivate(obj);
  DestroyWindow(obj);
  IF obj.dirty THEN
    obj.clientSite.SaveObject()
  END;
  obj.adviseSink.OnClose();
  obj.adviseHolder.SendOnClose();
  FireOberonEvent(obj, IDClosed)
END Close;
```

In addition to the methods mentioned above, the *IOleObject* interface provides many methods which are not required by most control implementations and can thus be left empty. An 'empty' method implementation simply returns a predefined result value.

```
PROCEDURE [WINAPI] Empty(this: Interface; ...): HRESULT;
BEGIN
  RETURN ENotImpl
END Empty;
```

7.4.2 IOleInPlaceObject

The *IOleInPlaceObject* interface manages the activation and deactivation of in–place objects, and determines how much of the in–place object should be visible. The implementation of in–place activation

and deactivation has already been discussed earlier. Method *SetObjectRects* is called to update the position and clipping region of an embedded object.

7.4.3 IOleInPlaceActiveObject

The *IOleInPlaceActiveObject* interface provides a direct communication channel between an in–place object and the associated application's outer–most frame window and the document window within the application that contains the embedded object. The communication involves the translation of messages, the state of the frame window (activated or deactivated), and the state of the document window (activated or deactivated). Also, it informs the object when it needs to resize its borders, and manages modeless dialog boxes.

As the Oberon control does not need a menu or toolbar, the implementation of this interface is completely empty. The interface is nevertheless provided since some containers only support controls with a complete set of in–place related interfaces and methods.

7.4.4 IViewObject

The *IViewObject* interface enables an object to display itself directly without passing a data object to the caller. In addition, this interface can create and manage a connection with an advise sink so that the caller can be notified of changes in the view object. This is similar to *IDataObject*, except that *IViewObject* places a representation of the data onto a graphics device context while *IDataObject* places the representation onto an abstract transfer medium.

Oberon implements a windowed control. That is, the control has its own display context and window handler when activated. However, when the control is inactive, the state of the window is set to hidden and the display context is discarded. To update the visual representation of an inactive control, the container must provide a device context through the *IViewObject.Draw* method. This method is not only used to display the control on screen, but also for printing, and rendering into bitmaps. Oberon implements the draw method, by displaying the embedded frame to an offscreen bitmap. This bitmap is copied to the device context provided by the container using the *StretchBlt* routine.

```
PROCEDURE Snapshot(V: Viewers.Viewer; hdc: User32.HDC;
                   x, y, w, h: LONGINT);
  VAR
    D: Display.DisplayMsg; B: Bitmaps.Bitmap;
    cur: Displays.Display;
BEGIN
  Oberon.LockOberon();
  Oberon.FadeCursor(Oberon.Pointer);
  B := Bitmaps.New(V.W, V.H);
  cur := V.win; V.win := B;
  D.F := NIL; D.device := Display.display; D.id := Display.full;
  Viewers.Send(V, D); V.win := cur;
  GDI32.StretchBlt(hdc, x, y, w, h, B.hDC,
      0, 0, V.W, V.H, GDI32.SrcCopy);
  Oberon.UnlockOberon()
END Snapshot;

PROCEDURE [WINAPI] *Draw(this: Interface; dwDrawAspect: LONGINT;
  hdcDraw: User32.HDC; rect: User32.RectL): HRESULT;
  VAR obj: Object; V: Viewers.Viewer;
BEGIN
  obj := this(ObjectInterface).obj(Object);
  IF dwDrawAspect = DVAspectContent THEN
    GDI32.SaveDC(hdcDraw);
    IF (obj.win # NIL) & (obj.win.viewer # NIL) THEN
      V := obj.win.viewer(Viewers.Viewer);
      Snapshot(V, hdcDraw, rect.left, rect.top,
          rect.right-rect.left, rect.bottom-rect.top)
    ELSE
      GDI32.Rectangle(hdcDraw, rect.left, rect.top,
          rect.right, rect.bottom);
      GDI32.MoveToEx(hdcDraw, rect.left, rect.top, NIL);
      GDI32.LineTo(hdcDraw, rect.right, rect.bottom);
      GDI32.MoveToEx(hdcDraw, rect.left, rect.bottom, NIL);
      GDI32.LineTo(hdcDraw, rect.right, rect.top)
    END;
    GDI32.RestoreDC(hdcDraw, -1);
    RETURN SOk
  ELSE
    RETURN DVEDVAspect
  END
END Draw;
```

7.4.5 IOleControl

The *IOleControl* interface provides the features for supporting keyboard mnemonics, ambient properties, and events in control objects.

Oberon implements only the *GetControlInfo* method, which is needed to capture the ESC and Return keys.

7.4.6 IQuickActivate

The *IQuickActivate* interface allows controls and containers to avoid performance bottlenecks on loading controls. It combines the load–time and initialization–time handshaking between a control and its container (see: *IOleObject* and *IViewObject*) into a single call.

7.5 OLEDataServices

The Oberon OLE/ActiveX objects *OLEObjects* and *OLEFrames* discussed above, implement wrappers for arbitrary visual and non–visual Oberon objects. When new instances of these OLE objects are created, they are empty; that is, they do not contain (wrap) any Oberon object. Appropriate mechanisms have to be supplied to transfer native Oberon objects to a wrapper. There are two possible solutions to this bootstrap problem:

1. A new COM class is implemented for each Oberon object class. A client can then create uninitialized Oberon objects directly as COM objects.

2. Empty wrappers are created by the client and the object content is transferred from a running Oberon system using a uniform data transfer method like the clipboard or drag–and–drop.

Using the first approach, Oberon objects can be directly created by a client by their COM CLSID. But a COM class must be created and registered for every possible Oberon object (–class), thus only a limited set of preconfigured objects can be provided this way. The second approach uses only one COM class for the empty wrapper, while the contents is copied from an object in a running Oberon system. A big advantage of using the second method is, that objects can be customized in their 'natural' Oberon environment and a ready–to–use object can be transferred to the client.

Module *OLEDataServices* implements the UI for uniform data transfer transactions: clipboard, drag–and–drop, and document files.

7.5.1 Clipboard

The Copy and Cut commands place an *OLEObjects* or *OLEFrames* wrapper object onto the OLE clipboard depending on whether the selected Oberon object to be copied is visual or not. A client pasting the object from the clipboard will request an appropriate representation of the object through the *IDataObject* interface. An Oberon client, such as the Oberon System or the Oberon OLE wrapper objects, will retrieve a copy of the original Oberon object by specifying an Oberon specific clipboard data format. The Oberon clipboard format is registered with the string 'ETH Oberon, Object Library').

The Paste command creates an Oberon object from the data object on the OLE clipboard, using the conversion routines provided by *OLEData*. With the PasteAs command, a specific data format for the conversion can be enforced. For example, a bitmap editor might use format 'Bitmap' whereas a text editor might use format 'Rich Text Format'.

Using the PasteAsLink command, the copied object (client) keeps a link to the original object (server). The purpose of this link is to receive notification messages when the original object changes. Behind the scene, Oberon must implement a client site object to host the linked object.

7.5.2 Client Site

Normally, when consuming a data object from the clipboard or as a drop target, the data of the object is copied and the link to the original is immediately released. This simple procedure has two disadvantages: change notifications are lost, and only the representation or aspect requested in the consume action is available. An alternative approach is to create a link object to the data object to be consumed. The link object acts as proxy for the external data object. The proxy object can be registered with the data object to receive change notifications. The client can request the original data object to render its data in different formats as needed.

The Oberon implementation of this proxy or link object is composed of at least three different (client–) objects as illustrated in figure 7.9.

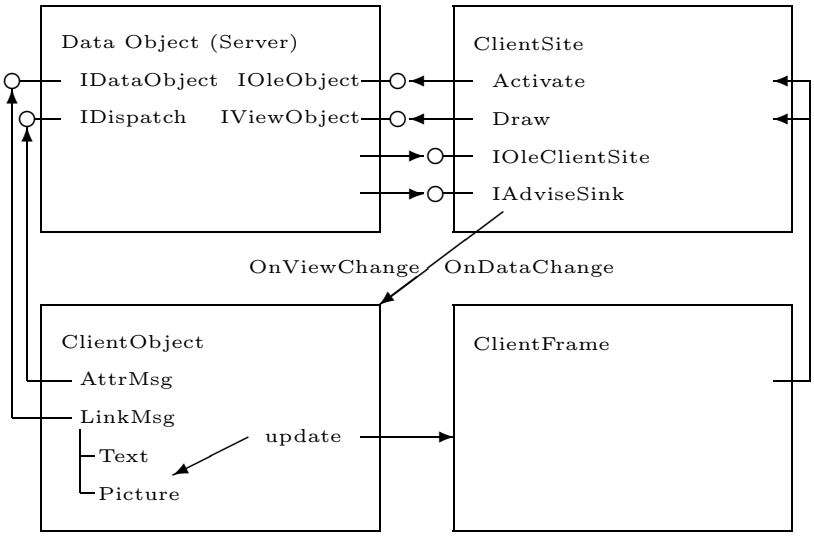

Figure 7.9: Client site and helper objects.

ClientSite Implements the COM interface *IOleClientSite* required to host the link object. The *IAdviseSink* interface is registered with the data object to receive change notification messages.

ClientObject An Oberon object (extension of Objects.Object) providing access to the data of the object properties (IDispatch) using a standard *AttrMsg*. The different possible conversions of the data (IDataObject) are provided through a standard *LinkMsg*. For example, for a data object providing text data, a link 'Text' is enumerated. The corresponding text model object (Texts.Text) is created when the 'Text' link is retrieved for the first time. The *ClientObject* implementation ensures that such model objects are automatically updated, when receiving change messages through the client site.

ClientFrame A visual Oberon object (extension of Display.Frame) displaying the contents as rendered by the data object itself. This frame acts also as UI to the client site; that is, by clicking the frame, the data object is activated for editing in a separate window.

7.5.3 Drag and Drop

The only difference between the clipboard (copy and paste) and drag–and–drop is the different UI. That is, the clipboard functions are triggered *explicitly* by calling corresponding commands, whereas drag–and–drop functions are activated *implicitly* by moving visual objects between windows.

Each window created by Oberon must be registered with OLE as a drop target. For this purpose, a (single) drop target COM object implementing the interface *IDropTarget* must be supplied.

```
PROCEDURE InitDropTarget(win: Windows.Window);
BEGIN
  IF dropTarget = NIL THEN
    dropTarget := CreateDropTarget();
    CoLockObjectExternal(dropTarget, TRUE)
  END;
  RegisterDragDrop(win.hWnd, dropTarget)
END InitDropTarget;
```

A drop action is initiated by the drop source calling the *IDropTarget.DragEnter* method. In this call, a data object is passed to the target. Depending on the data formats provided by the data object, the target will accept or reject the drop action (no parking sign). While dragging the icon over the target window, the *IDropTarget.DragOver* is repeatedly being called to update the feedback icon. When the dragging is aborted without a drop, the *IDropTarget.DragLeave* method is called. When the dragged icon is dropped over the target window, the *IDropTarget.Drop* method is called, and the data object is consumed in the same way as with the paste command for the clipboard. As not all drop sources do provide all possible drop effects in the *DragOver* method, the Oberon implementation shows a context menu at the drop location allowing the user to choose from all possible variants (copy, move, link).

To drag an object from Oberon to a non–Oberon window, a wrapper object must be created as is done in a clipboard copy. Most of the drag–and–drop functionality is provided by the OLE library function *DoDragDrop*. In addition to the data object (wrapper), only an implementation of the interface *IDropSource* must be supplied. This interface contains only two methods. The *QueryContinueDrag* method

is called to determine whether a drag–and–drop operation should be continued, canceled, or completed.

```
PROCEDURE [WINAPI] *QueryContinueDrag(this: Interface;
  escPressed: BOOL; keyState: SET): HRESULT;
BEGIN
  IF escPressed THEN
    RETURN DragDropSCancel
  ELSF ~(MKLButton IN keyState) THEN
    RETURN DragDropSDrop
  ELSE
    RETURN SOk
  END
END QueryContinueDrag;
```

The *GiveFeedback* method is called to give visual feedback to the end–user during a drag–and–drop operation. As Oberon uses the OLE–provided default cursors, the implementation of this method can be left empty.

7.5.4 Document Files

In addition to the clipboard and drag–and–drop, many OLE containers provide the possibility to 'insert object from file'. The so–called 'Document File' must contain the object to be inserted in its root storage (see figure 7.7). Module *OLEDataServices* provides the command *StoreDocfile* to store an Oberon object (using a COM wrapper) in its persistent state in such a document file. Such a file can subsequently be used to insert the Oberon object into OLE containers like Word or Excel.

Unfortunately, Visual Basic does not support any of the above methods to insert an object in its persistent state into a form: only new empty object instances may be inserted. Module *OLEDataServices* provides a special command to manipulate existing Visual Basic form files. The *ChangeVBFrame* command can be used to exchange any named frame in a form file by a new one. In contrast to document files implemented as structured storage, Visual Basic stores objects by a textual description of their properties (*IPersistPropertyBag*) or as a simple stream using the *IPersistStream* interface.

7.6 OLEPlugIns / Internet Explorer

The Netscape browser plug–in discussed in section 6.2 does also run under Internet Explorer. This setup is however suboptimal, as optional features like scripting or writing to network streams are not supported by the Netscape plug–in system of Internet Explorer. Module *OLEPlugIns* implements an Oberon plug–in for Internet Explorer based on the *OLEFrames* ActiveX wrapper object described earlier in this chapter.

Modules *OLEPlugIns* extends the *OLEFrames* control with the three additional interfaces.

IPersistPropertyBag This interface is used to store the persistent data of an object as a set of properties, each of which is a character string. For *OLEPlugIns* objects, the persistent data is stored as HTML text using PARAM tags. The persistent data for an Oberon plug–in consists of the parameters 'WIDTH', 'HEIGHT', 'Name', 'Src', and 'Gen' described in section 5.4. The values of these parameters are parsed from the HTML page by calling the *IPropertyBag.Read* method on the property bag object passed to the plug–ins load method (*IPersistPropertyBag.Load*).

IBindStatusCallback In the Internet Explorer, URLs can be requested using the *URLDownload* API function. The plug–in object issuing the request can be registered to receive notification messages as the download process proceeds. The interface *IBindStatusCallback* must be implemented by the receiver of such notification messages. This interface is implemented by *OLEPlugIns* to provide asynchronous network streams for background applet installation as described in section 5.2.2.

IObjectSafety This interface is used by Internet Explorer to configure the different safety options for scripting and initialization. This interface is required for any object to be embedded in the browser, its implementation however can be left empty.

The following HTML code implements the same functionallity as the example given earlier in section 6.4. The main difference is that the OBJECT tag replaces the EMBED tag. The OBJECT tag has a more verbose syntax and requires an explicit CLSID, whereas in the

EMBED tag the objects implementation is found by the media type of the data file. A second minor difference is the use of a Visual Basic Script in place of the JavaScript. The data file for the applet as well as the HTML fill–out form remain unchanged.

```
<HTML>
<HEAD>
<TITLE>VBScript  Example</TITLE>
<SCRIPT LANGUAGE="VBS">
Sub SetColor(color)
  Dim obj
  Set obj = Panel.FindObj("Skeleton")
  Call obj.SetInt("Color", color)
  Call obj.Update
End Sub
</SCRIPT>
</HEAD>

<BODY>

<OBJECT ID="Panel" WIDTH=220 HEIGHT=200 CLASSID="CLSID:...">
<PARAM NAME="Name" Value="Panel">
<PARAM NAME="Src" Value="LiveConnect.oaf">
</OBJECT>

Modify the "Color" attribute of "Skeleton".
<FORM>
<INPUT TYPE=button VALUE="Red" onClick="SetColor(1)">
<INPUT TYPE=button VALUE="Green" onClick="SetColor(2)">
<INPUT TYPE=button VALUE="Blue" onClick="SetColor(3)">
</FORM>

</BODY>
</HTML>
```

7.7 OLEObjectScripts

The browser plug–in technologies presented so far are all limited to visual applets. There are however applications where non–visual applets are useful as well. One class of such applets is that of *Scriptlets*.

Scriptlets are dynamically installed applet objects able to parse and execute a special purpose scripting language. This in contrast to the general purpose scripting languages which are typically statically built into the browser (JavaScript, Visual Basic Script).

Scriptlets may either be used by other scriptlets or by regular scripting languages, much like procedures or functions in a programming language. But they may as well be used as texutal descriptions of a data model for another applet in the browser. An example of such a textual description language used on the Web is VRML (Virtual Reality Modelling Language).

In the Oberon System, there exist different such descriptive languages which could be used as scriptlets. Examples of such languages are: Vinci [Osw00], Dim3 [Ulr96, Osw94], LOLA [Wir95], PowerDoc, and LayLa [Der96]. A complete scriptlet example using the Vinci graphics description language is discussed later in section 8.1.3.

Module *OLEObjectScripts* implements a script object which can be bound to an arbitrary Oberon applet within the same context (HTML text). The task of the script object is to convert the scriptlet text into Oberon format and to locate the target object by its name. The actual interpretation of the script text is the task of the receiving object. This ensures that this approach will work with any Oberon object capable of interpreting a description text. Stand–alone scripts (with no target object) are not supported with this method. The HTML skeleton below illustrates, how a scriptlet (... scriptlet text ...) is associated with a web applet (named 'Frame').

```
<OBJECT ID="Frame" WIDTH=... HEIGHT=... CLASSID="CLSID:...">
<PARAM NAME="Src" Value="... .oaf">
</OBJECT>

<SCRIPT LANGUAGE="Oberon.OLEObjectScripts" FOR="Frame">
... scriptlet text ...
</SCRIPT>
```

For a scriptlet to be recognized as a valid ActiveX scripting engine, the following three interfaces must be provided at a minimum:

IActiveScript The lifetime of the script engine and the communication with its container are controlled through this interface, which is similar to the *IOleObject* interface. The container calls the method *SetScriptSite* to inform a scripting object (engine) of its hosts *IActiveScriptSite* interface. The engine needs the latter to access other objects within the same container.

IActiveScriptParse This interface is used in environments where the persistent state of the script is intertwined with the host

document and the host is responsible for restoring the script, rather than through an *IPersistX* interface. The primary examples are HTML scripting languages that allow scriptlets of code embedded in the HTML document to be attached to objects or to events.

```
PROCEDURE AddScriptlet(this: Interface; code, item,
    subItem: String);
  VAR
    sobj: ScriptObject; itm: Interface;
    oobj: OberonOLEObject; text: Texts.Text;
BEGIN
  sobj := this(ScriptObject);
  (* find object item "." subItem, e.g. document.Frame *)
  itm := FindObj(FindObj(sobj.site, item), subItem);
  IF itm IS OberonOLEObject THEN
    oobj := itm(OberonOLEObject);
    text := GetText(code);
    oobj.SetLink("Model", text);
    oobj.Update();
    RETURN SOk
  END;
  RETURN EUnexpected
END AddScriptlet;
```

IObjectSafety This interface is used by the Internet Explorer to configure the different safety options. As the Oberon scripting engine implementation does not feature code signing or another standard safety mechanism, the implementation of this interface can be left empty.

Chapter 8

Case Studies

This chapter illustrates the concepts presented in this thesis, by discussing concrete applications. The applications are divided into the two sections 'Web Applets' and 'ActiveX Controls'.

8.1 Web Applets

This section demonstrates the use of pluggable Oberon objects as Web (HTML) applets. The applets presented in the sections 8.1.1 and 8.1.2 can be used with any browser supporting applets as presented in chapter 5.

8.1.1 Simple Applets

An Oberon applet can be called *simple* if the applet does not require any resources other than the ones installed with its source package (see section 5.4). One popular variant of such applets are small puzzle or card games. A classical example of such a game is the Oberon version of scramble. The goal of the game is to unscramble a puzzle with a picture on one side and numbered pieces on the other side.

A simple applet consists of two parts: package files containing the different resources and a HTML page embedding the applet. As the top package referenced in the HTML text is transferred every time the page is accessed, this package should be kept as small as possible. In the

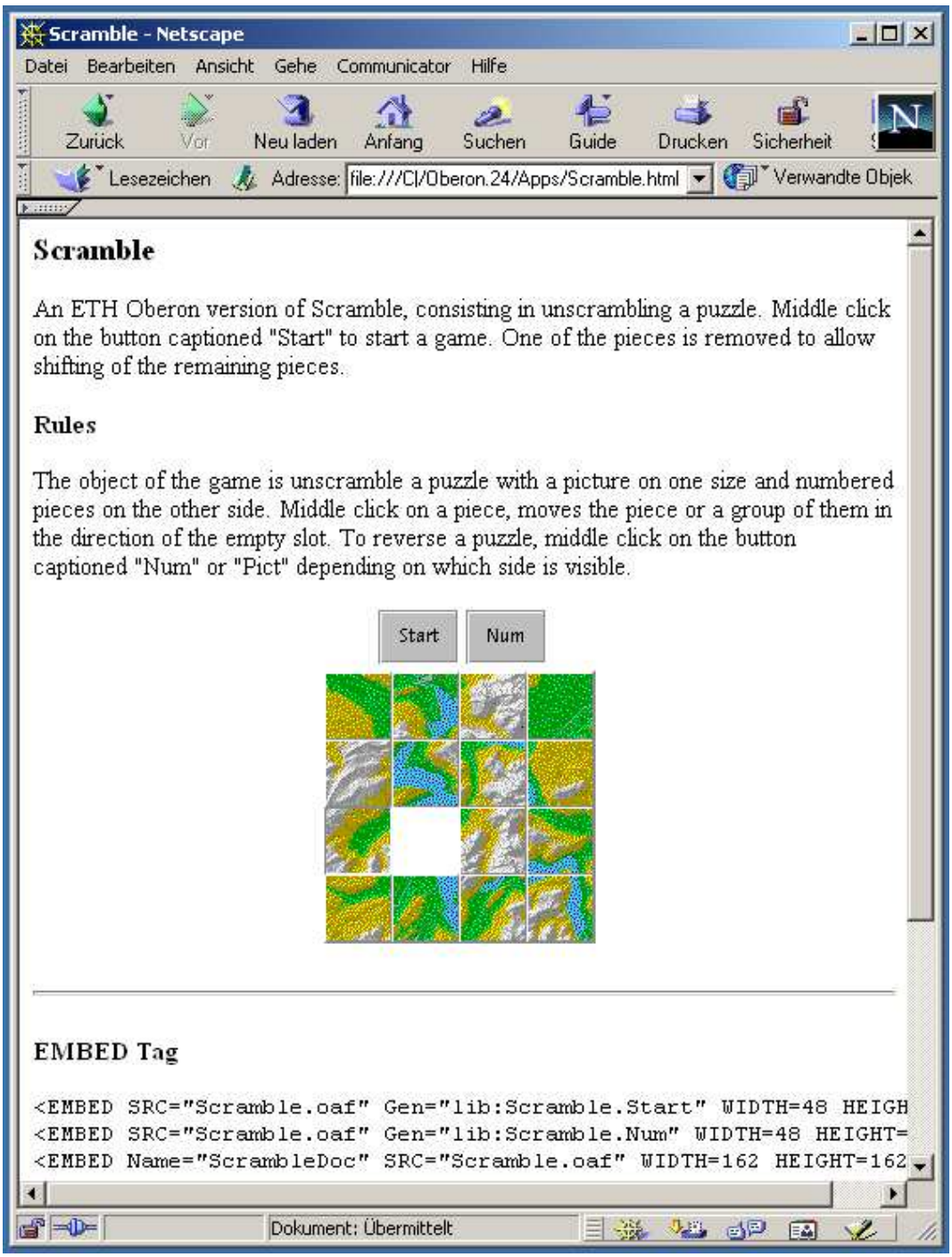

Figure 8.1: Scramble game as Netscape applet.

Scramble example, the top package file contains only the two (menu) buttons required by the game, all the other resources are contained in the package file *Games.oaf*. As the latter is referenced by *Scramble.oaf* using a link inside the package (URL Games.oaf), *Games.oaf* is only transferred the first time the page is loaded.

Description to build the Scramble package file:

```
PACKAGE Scramble.oaf 2.4 "doc:(Scramble.NewDoc)"
  DATA Scramble.Lib
  URL Games.oaf
  DEST System
  COPY Scramble.Lib
PACKAGE Games.oaf 2.4 "doc:Games.Tool"
...
```

Scramble.html:

```
<HTML>
<HEAD><TITLE>Scramble</TITLE></HEAD>
<BODY>

<P ALIGN=CENTER>
<EMBED SRC="Scramble.oaf" Gen="lib:Scramble.Start"
 WIDTH=48 HEIGHT=32>
<EMBED SRC="Scramble.oaf" Gen="lib:Scramble.Num"
 WIDTH=48 HEIGHT=32><BR>
<EMBED Name="ScrambleDoc" SRC="Scramble.oaf"
 WIDTH=162 HEIGHT=162>
</P>

</BODY>
</HTML>
```

Standard browsers do not provide any tools to manipulate plug–in applets, but the developer of a new applet would sometimes like to inspect or change an applet inside the browser. For this purpose the control viewer containing an applet (see section 6.2) provides a context menu (see figure 8.2) for accessing important standard functions such as: the Oberon log viewer or the object inspector.

A second source for simple applets are Oberon texts with embedded objects. For example the text document 'GettingStarted.Text' provides an introduction into using Oberon with Gadgets. This tutorial uses embedded panels called 'Sandboxes' to illustrate the basic

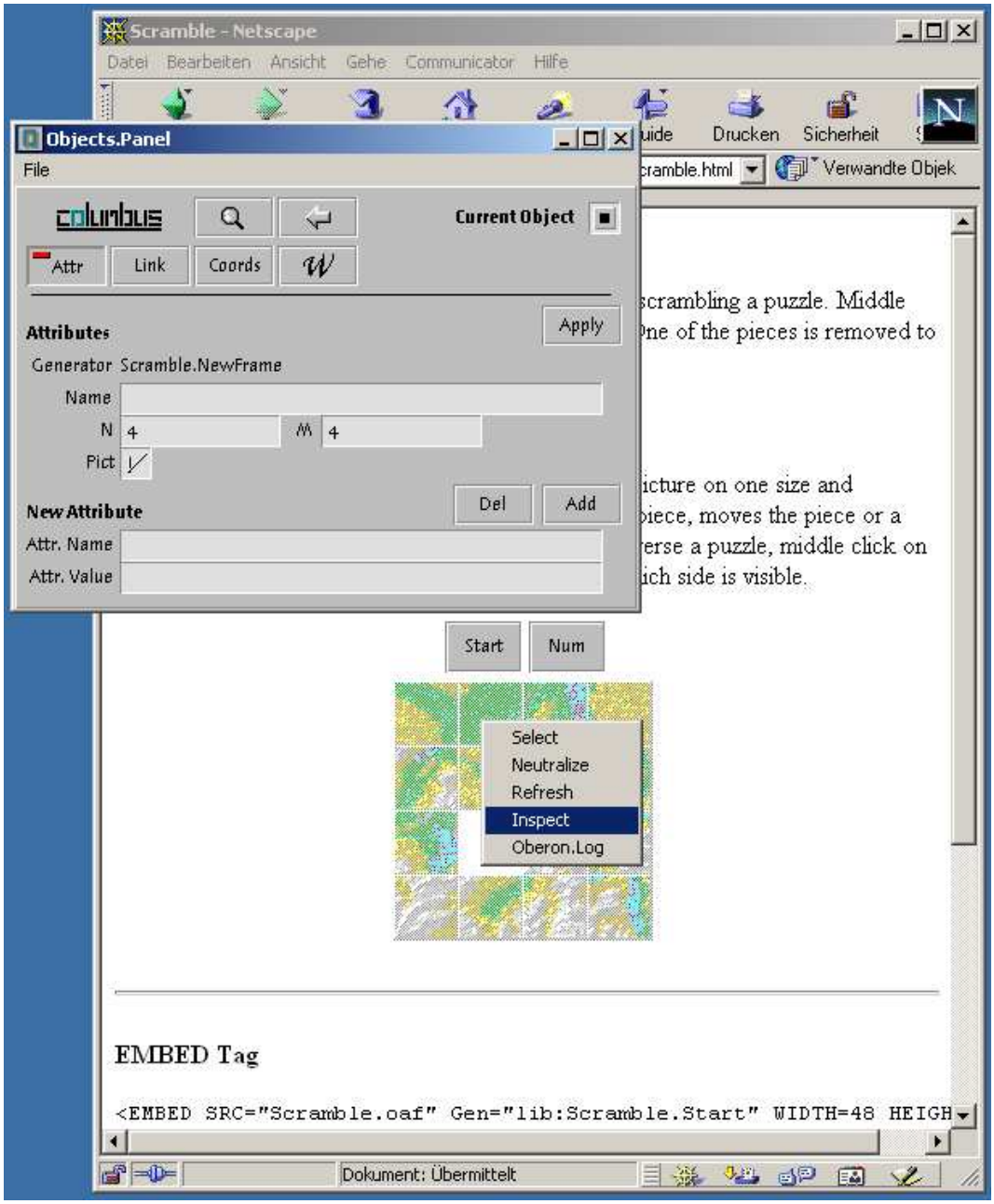

Figure 8.2: Applet Context Menu.

Gadgets concepts in a interactive fashion. Of course these sandboxes are ideal simple applets, as each of them implements a self–contained small application.

Oberon texts with embedded objects can be converted into HTML texts with applets using the tool *Text2HTML*. This tool is able to map the text attributes *font*, *color*, and *text* to the corresponding HTML tags. Using some heuristics it is as well possible to recognize the structure of the text document and to generate corresponding title and paragraph tags. Embedded objects are stored in a public object library, this library is used to build a package file referenced by the EMBED tags in the generated HTML text.

Excerpt of generated HTML file 'GettingStarted.html':

```
<P><FONT FACE="Syntax" SIZE=3 COLOR=#000000>
Or we can use <I>container</I> components like this <I>panel</I>
to group gadgets together:
</FONT></P>

<P>
<EMBED NAME="ObjRef97" SRC=GettingStarted.oaf
WIDTH=510 HEIGHT=159 GEN="lib:GettingStarted.ObjRef97">
</P>
```

8.1.2 Applets using Network Streams

One limitation of the simple applets presented in the previous section is, that the applets can only use the resources contained in their application package files. Using the network stream abstraction (see section 5.2.2) makes it possible to implement applets which are capable of downloading additional resources at run–time or which can store their persistent state on the Web.

Figure 8.4 shows a snapshot of a Web–based Oberon programming tutorial. The reader can try out the programming examples directly inside the browser without the need for starting any additional applications or tools. To load additional programming examples or to save changes between sessions, the reader of the tutorial can use the embedded *Load* and *Store* buttons. The source or destination path specified in the corresponding textfield is not limited to local file names, but may be any URL where the reader has sufficent permissions to read or write files.

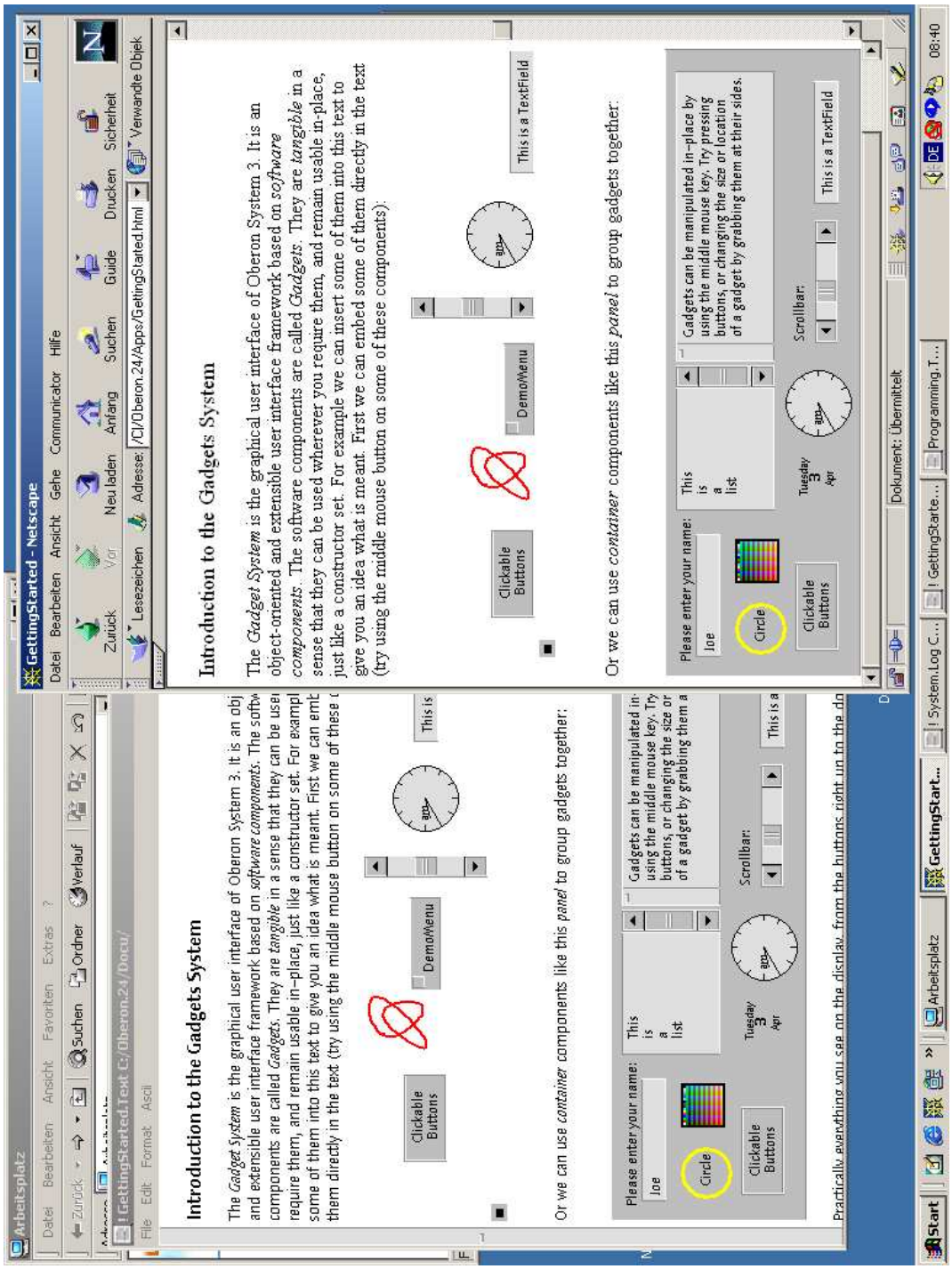

Figure 8.3: Original Oberon text and HTML text generated with Text2HTML.

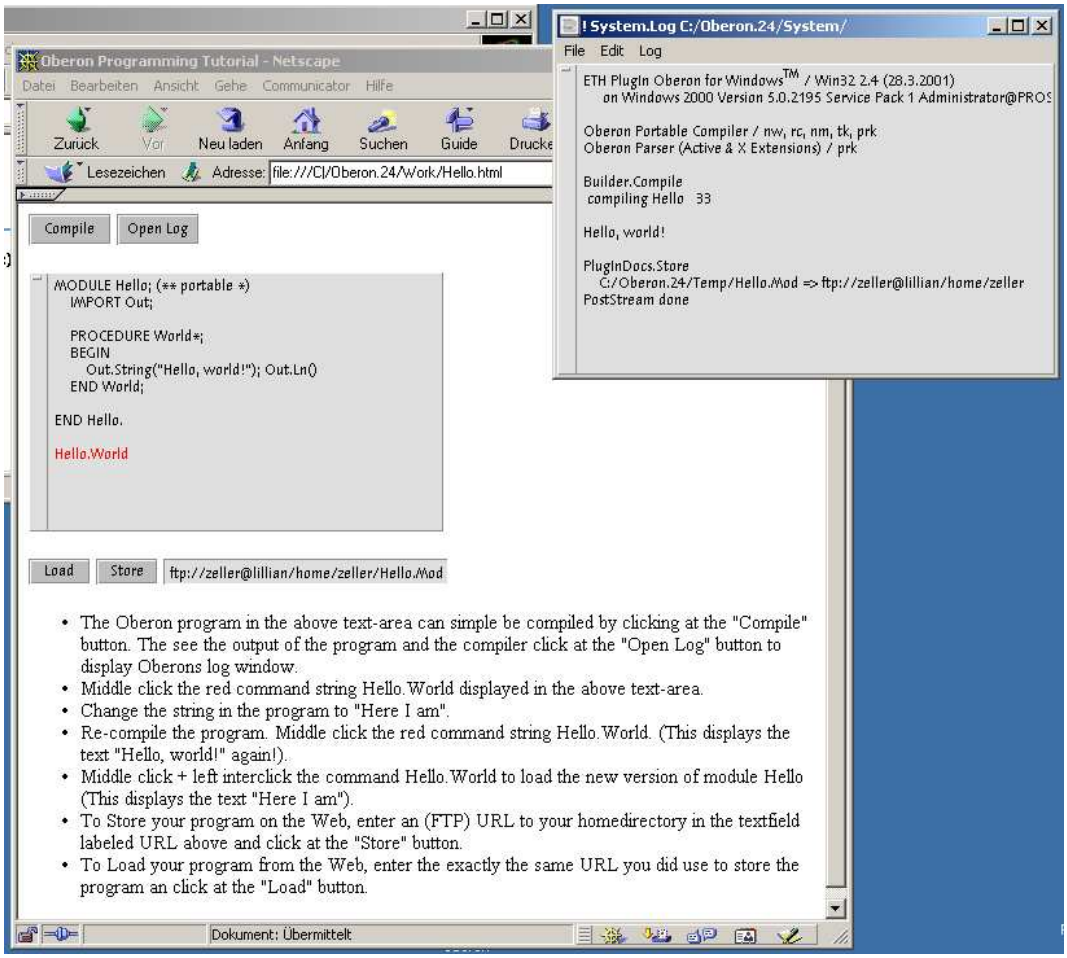

Figure 8.4: Web–based Oberon programming tutorial.

Loading and storing of Oberon document data on the Web is a feature which can be useful with any document–based applet. It could for example be used with the game applet presented earlier, to store the state of the game.

Module *PlugInDocs* implements commands for loading and storing of arbitrary Oberon document data on the Web.

8.1.3 Applets and Scripting

Scripting enables the implementation of applets which can communicate with other applets and HTML controls. This communication is not limited to applets of the same technology, but enables the interoperation of applets of different cultures such as Java or Oberon. This flexibility is achieved by defining a common communication infrastructure. The Netscape browsers use a Java based technology called *LiveConnect* (see section 6.4) for this infrastructure. The Internet Explorer uses *Automation* also called *Programmable Objects*, this technology depends on the *IDispatch* interface (see section 7.3.2).

Scripts can be used as glue logic code to bind standard HTML fill–out form controls to an applet. Using the browsers standard controls has the advantage, that the Web page can be presented in a unified look. Figure 8.5 shows a snapshot of the *Dim3* viewer using its own Oberon controls and using standard HTML controls. *Dim3* [Ulr96] is a package for visualizing and exploring three–dimensional scenes that are built from polygons. It is based on an earlier package called *PolyWorlds* [Osw94], which it extends by adding specular reflections, Gouraud shading and textured surfaces.

Dim3Java.html:

```
...
function SetFieldView(field) {
  obj = document.camera.GetObj();
  obj.Execute("Dim3Frames.SetViewAngle " + field);
}
...
<INPUT TYPE=button VALUE="FieldView"
  onClick="SetFieldView(FieldView.value)">
<INPUT NAME=FieldView TYPE=int MIN=50 MAX=120 VALUE="60"
  onChange="SetFieldView(FieldView.value)">
...
```

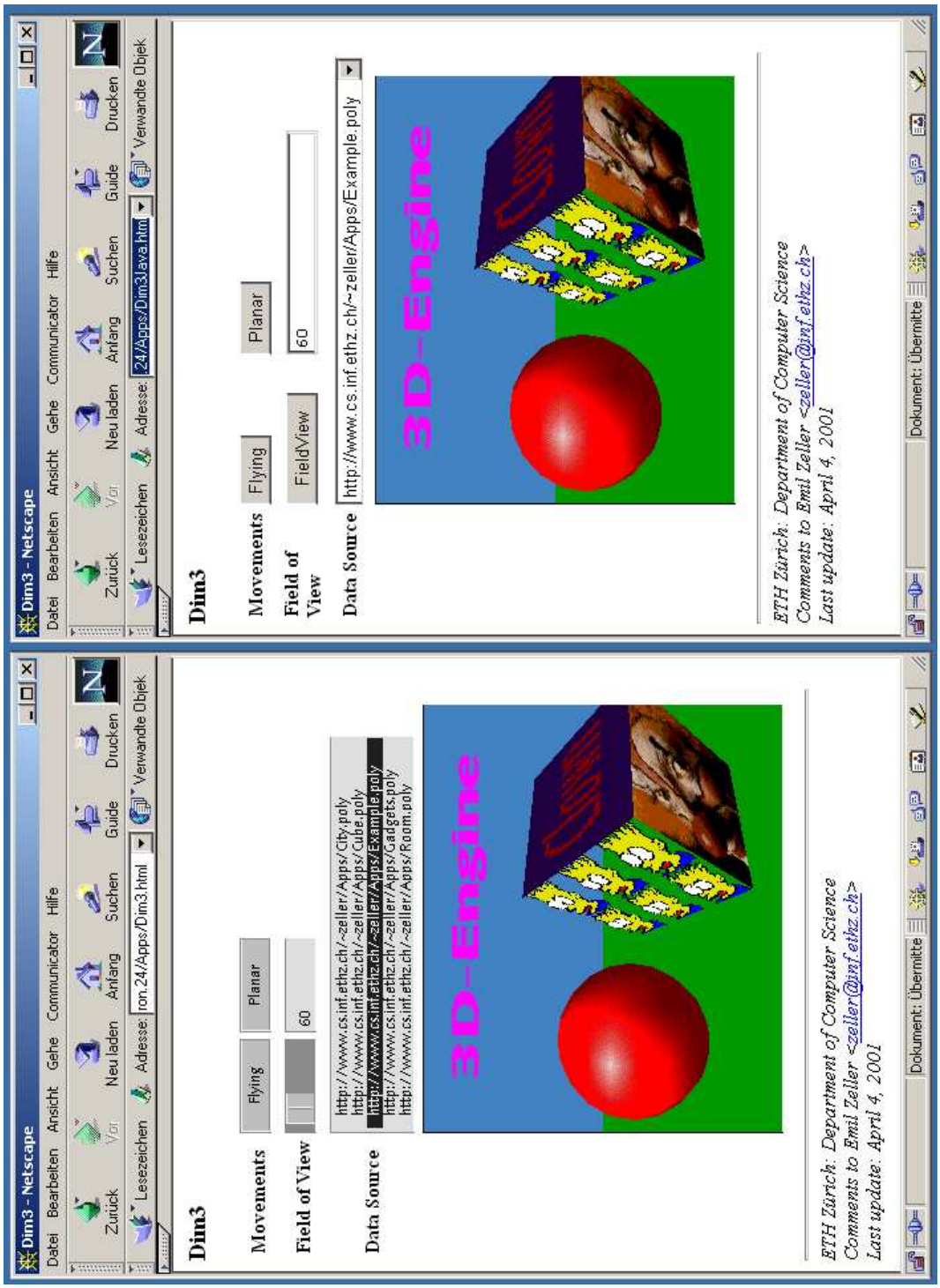

Figure 8.5: Dim3 viewer with Oberon and HTML controls.

Normally scripts are only executed, when an applet or some other browser object fires a corresponding event. For example a push–button firing an *onClick* event (see HTML/JavaScript code above). The Internet Explorer allows arbitrary scripting languages to be plugged into the browser. These *Scriptlets* are (non–visual) applets which are able to execute their own scripts.

The Oberon System features different special purpose scripting languages. The most recent of these languages is the Vinci [Osw00] graphics description language. Some examples of Vinci generated graphics are shown in figure 8.6.

OLEVinci.html:

```
...
<OBJECT ID="Smiley" WIDTH=256 HEIGHT=256 CLASSID="CLSID:...">
<PARAM NAME="Base" Value="OLEVinci.oaf"></OBJECT>
...
<SCRIPT LANGUAGE="Oberon.OLEObjectScripts" FOR="Smiley">
import colors;

with width=3 do
  with color=colors.yellow do
    fill circle(100, 100, 80) end
  end;
  stroke circle(100, 100, 80) end;
  fill circle(70, 110, 10) end;
  stroke
    from (120, 110) arc (130, 130) to (140, 110);
    from (60, 70) arc (100, 140) to (140, 70)
  end
end
</SCRIPT>
...
```

8.2 ActiveX Controls

Most examples presented so far are Web based. In this section some non–Web applications of pluggable Oberon objects are presented. In this applications, objects are plugged into non–Oberon applications using the ActiveX/OLE wrapper *OLEFrames* (see section 7.4).

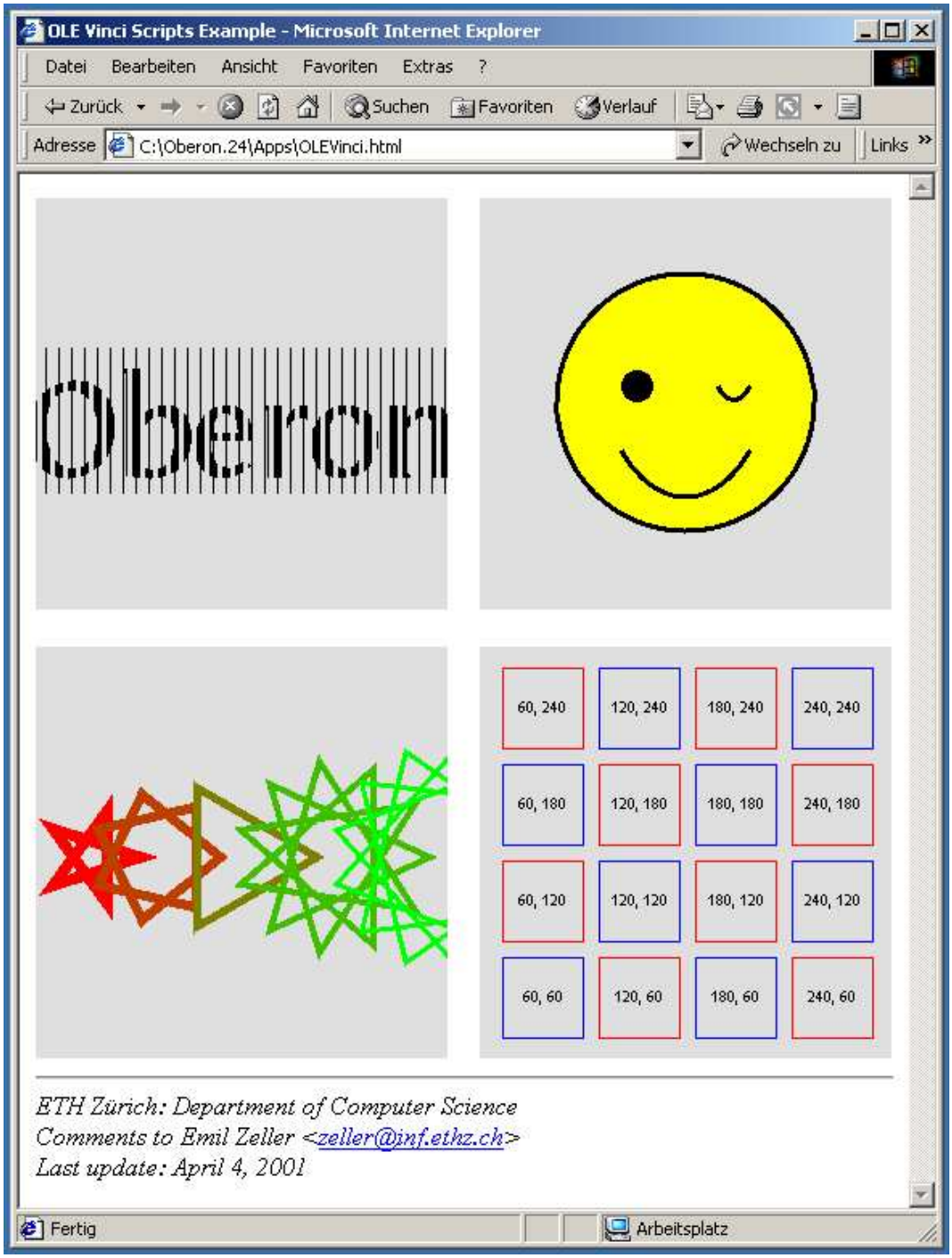

Figure 8.6: Some Vinci graphics examples.

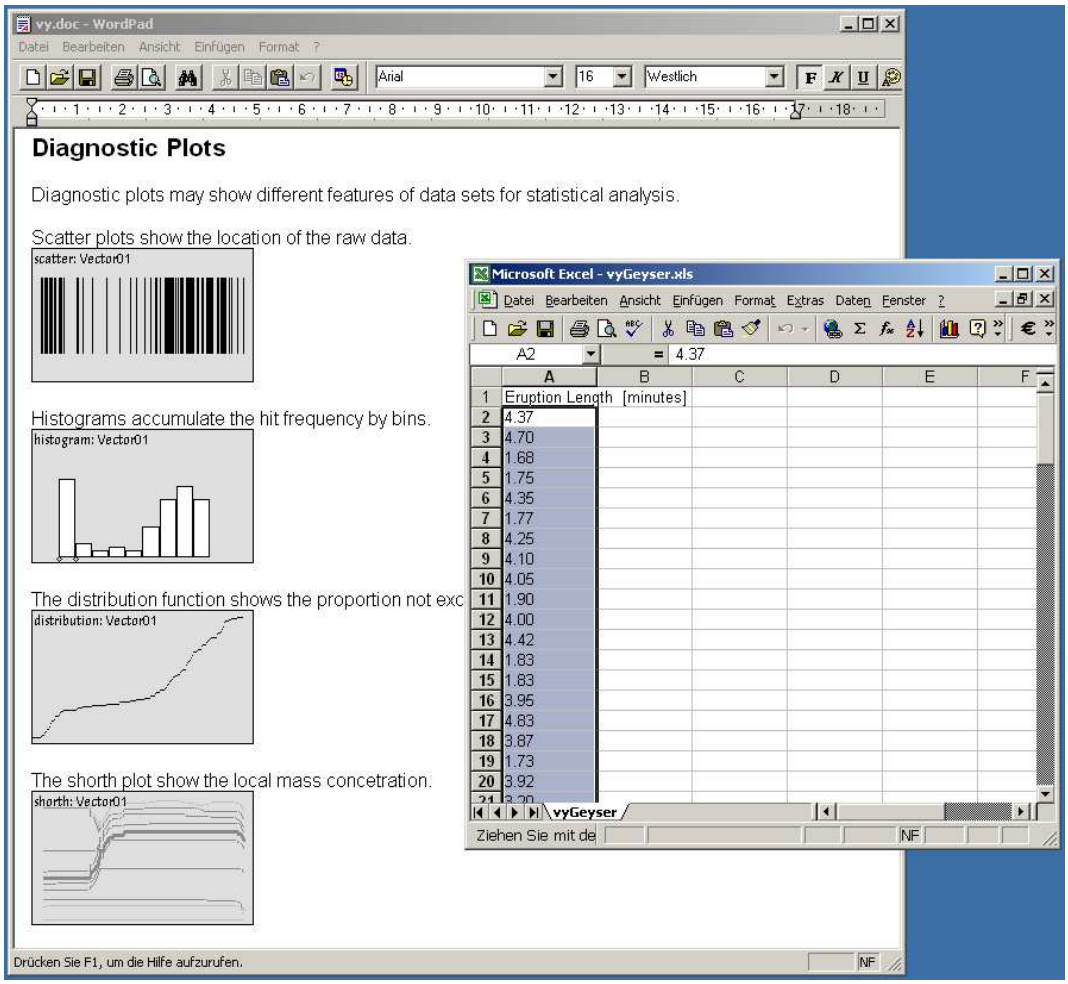

Figure 8.7: Voyager Gadgets as ActiveX control in a Word document with data feed from an Excel sheet.

8.2.1 Voyager and Microsoft Office

Voyager [Saw96, Saw00] is a project to explore the feasibility of a portable and extensible system for simulation and data analysis systems. The Voyager software consists of a huge number of visual and non–visual Gadgets.

In the example shown in figure 8.7, a Word document introduces different methods for statistical data visualization. The embedded Voyager Gadgets are all linked to the same data model object. The data model object can be exchanged by a new one by simply dragging data items from a suited data source and dropping them over one of the statistics plots. In the example shown in the snapshot, cells from an Excel sheet — containing eruption length data of a geyser — are used as data source.

To implement this example no programming is required. The plot Gadgets can be created using the standard Voyager tools with Oberon for Windows. These preconfigured plots then can be stored in document files using the command *OLEDataServices.StoreDocfile*. In Word these plots can be created from their document files using the 'insert object from file' menu.

8.2.2 Visual Basic

Visual Basic is the most popular programming environment for developing Windows applications composed of predefined and custom controls. Visual Basic supports controls complying with its own standard *Visual Basic Controls* (VBX) as well as *OLE Controls* (OCX) and *ActiveX Controls*.

Teletext is a page oriented, non–interactive information service that is broadcast by television stations together with the video signal. Even though the Internet is very popular nowadays, Teletext is still a valuable source of information (weather forecast, television programs, etc.).

In [Som96] the implementation of a Gadgets user interface, for a Teletext decoder attached to a PC [Bau92], is discussed. A networked version of the Oberon Teletext software using an intranet server with a proprietary protocoll is discussed in [Dis97].

Figure 8.8 shows a snapshot of a Visual Basic Teletext viewer application. The Teletext Gadget is integrated on the form using the special command *OLEDataServices.ChangeVBFrame* as described in section

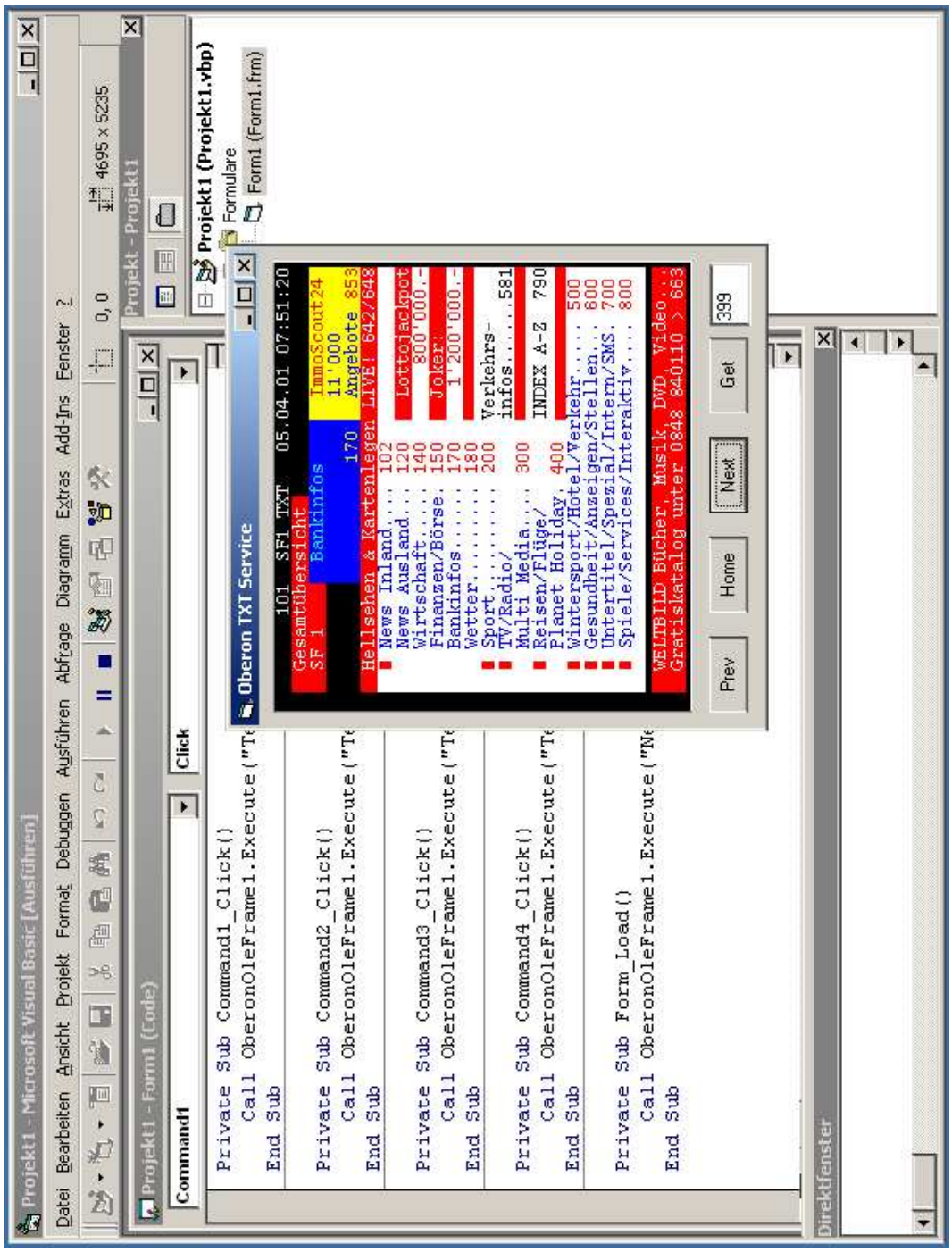

Figure 8.8: Teletext Gadget as ActiveX control on a Visual Basic form.

7.5.4. The push buttons and the textfiles are standard Visual Basic form controls. The Teletext Gadget is controlled by this elements by calling the *Execute* method of the Gadgets *IOberonOleObject* as described in section 7.3.2.

Chapter 9

Conclusions

By introducing a small extension in how display coordinates are handled in the hierarchic display space, many new ways to deploy Oberon applications have been made possible. The most obvious application of this extension is the smooth integration of the Oberon GUI into that of a given host system. More advanced applications are embeddings where small Oberon components are integrated into standard applications like: web browser, spreadsheets, ...

The GUI integration has been successfully implemented for the Windows platform (as described in this thesis) as well as for UNIX platforms [Chr98]. So far the component level integration has only been implemented for different standard applications on the Windows platform. As most of the extensions are based on basic Oberon concepts, similar embeddings could easily be realized by an expert of the desired target platform.

9.1 Evaluation

The Oberon implementation presented in this thesis provides new ways to deploy Oberon applications, while still maintaing full compatibility with existing applications. All the extensions have been implemented with minimal changes to existing (standard) Oberon modules. This has two important benefits: most modules are identical with those from other Oberon implementations, and there are only few system

specific modules. Most of the extensions can be loaded and unloaded dynamically as they are needed.

The big advantages of Oberon over other systems like Java or .net are its efficency and simplicity. Although Oberon's component framework is very simple compared to that of Java or .net, it seems to be sufficently complete as basis for a wide range of non–trivial applications. There are two major reasons for this: the framework uses a very flexible and open message protocol, and the framework can easily be customized or extented.

As Oberon features a very fast compiler, modules required by applets can be recompiled on the target platform. This approach seems to be much more appropriate than, slow interpretation of byte–codes, or complex run–time optimizations. For typical applications the benefits of compiler or run–time optimizations are neglectible. Other advantages of using source code are: a high level of portability among different platforms, type–safety, and other security checks done by the compiler (prior to generating executable) code.

Module statistics and performance evaluation data can be found in appendixes A and B.

9.2 Future Perspectives

The most obvious follow–up project is to implement similar Oberon versions for other platforms. Candidates for such implementations are all platforms, where Oberon allready is used as application on–top of another operating system. Examples are different UNIX platforms, and MacOS.

A limitation of the current implementation is, that the communication between a host application and embedded Oberon applets is limited to the heavy–weight standard mechanisms (Java, OLE). A solution to this problem is "module level integration", thus an Oberon compiler which directly generates standard executable files. This would permit any application with a programming interface to call Oberon modules directly.

Appendix A

Module Statistics

To give an impression of the size of the various modules that are described in this thesis, tables A.1 up to A.10 list the sizes of involved Oberon modules. For each module, its name, its function, its size in number of statements, and its size in number of lines of source code are given. The number of statements are calculated by the Oberon parser *OPP*. The number of lines of source code are calculated by the *EditTools* tool. Few statements per line of code are typical for modules that contain many definitions.

Table A.1 lists the sizes for the Oberon inner–core modules. These are the modules required for Oberon's own module loader and, memory manager to work. These modules must be statically linked into a self–loading 'boot file' to allow further Oberon modules to be loaded. Table A.2 lists the sizes of the static–linker tools.

Module	Function	Statements	Lines
Kernel32	interface to Kernel32.DLL	68	610
Kernel	heap and garbage collection	577	821
ADVAPI32	interface to ADVAPI32.DLL	0	59
Registry	interface to Windows Registry	195	273
FileDir	directory managment	317	455
Files	file managment	421	697
Modules	Oberon module loader	606	869
Total		2184	3784

Table A.1: Oberon inner–core

Module	Function	Statements	Lines
BootLinker	Oberon static linker	1556	1977
PELinker	DLL and EXE linker	1135	1721
Total		2691	3698

Table A.2: BootLinker and PELinker

Table A.3 lists the sizes of all modules required to implement a minimal Windows Oberon system. On top of these modules different user–interfaces can be used. Windows Oberon can run either the classical tiled text–based UI known from 'Project Oberon' [GW92], the different variants of the Gadgets GUI system [Mar96], and the different plug–in technologies. All these UIs can be used concurrently. Other UIs like in the 'Bluebottle Project' [Gut02] could be ported to this base–system as well.

Module	Function	Statements	Lines
Objects	persistent objects library	476	878
Reals	IEEE numbers library	156	300
Dates	date and time library	116	186
Strings	string library	935	955
Threads	threads library	369	636
Types	type names and type tags	23	78
Exceptions	exception handling	276	476
User32	interface to User32.DLL	1	471
GDI32	interface to GDI32.DLL	14	366
Displays	virtual display driver	310	567
Display	Windows display driver	465	823
Fonts	font system	16	66
Viewers	display space manager	658	731
Input	mouse and keyboard	27	95
Texts	text abstract data type	1170	1229
Windows	Windows displays	736	1176
WinFonts	Windows fonts	387	492
Oberon	event dispatcher	607	991
Total		6742	10516

Table A.3: Base–system with basic Windows GUI

Table A.4 lists the sizes of the modules required to use Oberon documents as Windows style applications. This includes support for pull–down menus, pop–up menus (context menus), standard dialogs, and GDI printing.

Module	Function	Statements	Lines
COMDLG32	interface to COMDLG32.DLL	13	131
WinMenus	Windows menus and dialogs	796	1182
WinPrinter	Windows printer driver	635	834
WinFrames	control viewer	192	284
Total		1636	2431

Table A.4: Advanced Windows GUI

Table A.5 lists the sizes of the Oberon browser plug–in framework including the plug–in implementation for Oberon's own browser (HTMLDocs).

Module	Function	Statements	Lines
Packages	Package file manager	857	1095
PlugIns	applet installer	200	373
HTMLPlugIns	HTMLDocs applets	202	340
PlugInDocs	streaming documents	147	197
Total		1406	2005

Table A.5: Oberon browser plug–in

Table A.6 lists the sizes of the modules required by the Netscape browser plug–in implementation including the Java interface. The module *NPLiveConnect*, and the two Java classes *NPOberon* and *NPOberonProxy* are only needed when using the Java interface (e.g. from a JavaScript).

Module	Function	Statements	Lines
WinPlugIns	applet viewer	135	265
NPPlugIns	Netscape plug–in	422	935
NPLiveConnect	plug–in Java interface	330	630
NPOberon	Java class	0	8
NPOberonProxy	Java class	6	43
NPText2HTML	Text converter	273	296
Total		1166	2177

Table A.6: Netscape browser plug–in

Table A.7 lists the sizes of the generic COM in–process and out–process servers. Both servers can be used as either stand–alone servers with no UI at all, or as part of a GUI.

Module	Function	Statements	Lines
COM	in–process server	410	744
COMServer	out–process server	102	178
Total		512	922

Table A.7: In–process and out–process COM servers

Table A.8 lists the sizes of the complete OLE uniform data transfer implementation. These modules form the basis for the OLE integration of Oberon (see table A.9 and table A.10).

Module	Function	Statements	Lines
OLE	OLE APIs and interfaces	298	1287
OLEEnum	IEnumXXX	47	109
OLEData	uniform data transfer	269	479
OLETexts	text conversion	126	196
OLEPictures	bitmap conversion	46	93
OLEGenericData	generic data	18	43
Total		804	2207

Table A.8: OLE APIs and interfaces including uniform data transfer

Table A.9 lists the sizes of the OLE wrapper implementations for visual and none–visual Oberon objects, and the Internet Explorer browser plug–in.

Module	Function	Statements	Lines
OLEObjects	Objects.Object wrapper	1183	1946
OLEFrames	Display.Frame wrapper	851	1285
OLEPlugIns	PlugIns wrapper	261	449
Total		2295	3680

Table A.9: OLE wrappers for Oberon objects, frames, and plug–ins

Table A.10 lists the sizes of the extensions to the OLE implementation for hosting active OLE data servers and custom Oberon scripting.

Module	Function	Statements	Lines
OLEClientObjects	linked object creation	117	219
OLEClientSites	linked object container	306	512
OLEClientFrames	linked object GUI	223	402
OLEDataServices	uniform data transfer UI	551	953
OLEObjectScripts	Oberon OLE scriptlet	304	496
Total		1501	2582

Table A.10: OLE uniform data transfer and scripting engine

Appendix B

Performance Evaluation

This section presents performance measurments of Oberon's Netscape browser plug–in implementation. The results are compared with a Java applet providing the same functionality. The Netscape browser plug–in has been choosen for two reasons.

1. The Netscape browser and its plug–in mechanism are provided with the same functionality on many different platforms. Thus similar results can be expected on other platforms running Netscape and Oberon.

2. The Netscape browser runs as a standard application without any hidden dependencies on the underlying operating system. Thus the numbers presented here are reproducible (repeated measurments give the same numbers).

The measurments are made using the *Scramble* puzzle game presented in section 8.1.1. A similiar Java based applet found on the Web [Rad97] is taken for comparison. The measurments are made using the following setup:

- Noname PC with AMD K6–II 350 MHz, 256 MB of RAM
- Microsoft Windows 2000
- Netscape Communicator 4.78

- latest Version of PlugIn Oberon for Windows

Measurments are taken for the following four different configurations:

A First the resources used by the browser displaying a local text–only page (no images, no plug–ins) is measured. These numbers are required to calculate relative values.

B In this configuration a minimal subset of Oberon, containing only the modules needed by the Netscape plug–in, is used. To keep this configuration very small, compiled modules are used by the applet. This configuration consists of 25 files with a total size of 800 kB.

C In this configuration a complete installation of Oberon is used. The applets modules are compiled from the source files.

D In this configuration the Java version of Scramble is used.

Table B.1 lists the absolute numbers measured. The first colum titled "startup" lists the number of seconds it takes to start the browser with the page under test as startup page. This numbers have been measured using a standard stopwatch.

config	startup	startup	minimized	restored
A	4 s	9140 kB	1068 kB	2388 kB
B	4 s	10496 kB	1108 kB	2836 kB
C	5 s	11620 kB	1124 kB	2976 kB
D	12 s	16584 kB	2592 kB	6988 kB

Table B.1: Measurment results.

The remaining three colums list the memory usage of the browser process. This includes the memory used by the Oberon plug–in or the Java applet, as both run in–process. The memory usage has been measured using the *Task–Manager*, a standard Windows tool. The taskmanager reports the resident set of memory allocated to a process. A process' resident set is that part of a process' address space which is currently in main memory [Rav01].

The second colum titled "startup" lists the number of kBytes used by the browser after startup and playing the game. As there was enough of free memory available in the test setup (no swapping), this number is the total amount of memory allocated by the browser and plug–in.

The third colum titled "minimized" lists the number of kBytes used by the browser after the browser has been minimized for 5 minutes. Windows actively start swapping out unused memory pages when an application (all of its windows) is minimized [Moz02]. Thus this number is the size of the minimal working set required by the browser and plug–in.

The fourth colum titled "restored" lists the number of kBytes used by the browser after the browser has been restored and playing the game again. As all unused memory pages have been swapped out by minimizing the application, this number is the working set required by the browser when active.

config	startup	startup	minimized	restored
A	0 s	0 kB	0 kB	0 kB
B	0 s	1356 kB	40 kB	448 kB
C	1 s	2480 kB	56 kB	588 kB
D	8 s	7444 kB	1524 kB	4600 kB

Table B.2: Overhead by using applet.

Table B.2 lists the overhead of the two plug–in configurations B, C and the Java applet D. The overhead is the difference to the minimal browser configuration A. Table B.3 lists relative values for the overhead numbers in table B.2.

The measurments have shown, that the Oberon browser plug–in can be loaded without any noticable delay. Loading the Java VM on the other hand will completely block the browser for 8 seconds. Once the Oberon plug–in or the Java VM are loaded, applets (modules or classes) can be loaded very fast on both systems.

The memory usage measurments have shown, that the memory footprint of Oberon based applets is much smaller than that of Java based applets. In configuration C a standard Oberon system is used as

config	startup	startup	minimized	restored
A	0%	0%	0%	0%
B	0%	15%	4%	19%
C	25%	27%	5%	25%
D	200%	81%	143%	193%

Table B.3: Relative overhead by using applet.

applet environment. This includes the OP2 compiler and the graphical text–editor *TextDocs*. The memory footprint for this configuration would even be smaller, when using a classical single–pass Oberon compiler.

Table B.4 lists the size of the different applet files used in configurations B, C and D. In configuration C2 the source code is compressed using the *SourceCoder* tool [Zel97]. The Oberon applet files (.oaf) use the older LZSS compression algorithm [Nel96]. Using the latest inflate algorithm used in the *ziplib* [RlG02] these files can be further shrinked by 20% or more. The uncompressed Java class file *Puzzle.class* is very compact, it is only have the size of a corresponding (Intel) object file used by Oberon. Using a Java archive (JAR) [Sun98] the Java class file can be compressed to a size similar to the compressed Oberon source code (.POM).

config	file	size in bytes
B (.Obj)	Scramble.oaf	10166
C (.Mod)	Scramble.oaf	7453
C2 (.POM)	Scramble.oaf	6594
D	Puzzle.class	9132

Table B.4: Size of applet data files.

Bibliography

[AB94] Erwin Achermann and Daniel Blank. Multi-Window Oberon. Master's thesis, Institute for Computer Systems, ETH Zürich, 1994.

[Arm98] Tom Armstrong. *Active Template Library – A Developer's Guide*. M & T Books, 1998.

[Bak97] Seán Baker. *CORBA Distributed Objects Using Orbix*. Addison–Wesley, 1997.

[Bau92] Lukas Bauer. Teletext für alle. *c't magazin für computer technik*, (7):176 – 182, July 1992.

[BCFT95] M. Brandis, R. Crelier, Michael Franz, and Josef Templ. The Oberon System Family. *Software – Practice and Experience*, 25(12):1331–1366, 1995.

[Bro95] Kraig Brockschmidt. *Inside OLE*. Microsoft Press, second edition, 1995.

[Cha96] David Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[Chr98] Beat Christen. Integration von Gadgets in den Desktop des Host–Systems. Master's thesis, Institute for Computer Systems, ETH Zürich, 1998.

[Con01] Connectix Corporation. The Technology of Virtual Machines. Web Page, 2001. http://www.connectix.com/-downloadcenter/pdf/vpcw-wp/vpcw-techwp-sep1301.pdf.

[Cox96] Brad Cox. *Superdistribution.* Addison–Wesley, 1996.

[Cre90] R. Crelier. OP2: A Portable Oberon Compiler. Technical Report 125, Institute for Computer Systems, ETH Zürich, 1990.

[Dät96] Markus Dätwyler. Executable Content in Compound Documents. Master's thesis, Institute for Computer Systems, ETH Zürich, 1996.

[DeL99] Robert DeLine. *Resolving Packaging Mismatch.* PhD thesis, CMU, 1999.

[Den97] Adam Denning. *ActiveX Controls Inside Out.* Microsoft Press, 1997.

[Der96] Jörg Derungs. Layout Language – eine Beschreibungssprache für Gadgets. Master's thesis, Institute for Computer Systems, ETH Zürich, 1996.

[DGL95] S. Drew, K. J. Gough, and J. Ledermann. Implementing Zero–Overhead Exception Handling. Technical Report 95–12, FIT, 1995.

[Dis97] Andreas Disteli. *Integration aktiver Objekte in Oberon am Beispiel eines Serversystems.* PhD thesis, Institute for Computer Systems, ETH Zürich, 1997.

[dM95] Mark de Munk. Concurrent Oberon for Windows NT. Master's thesis, Institute for Computer Systems, ETH Zürich, 1995.

[Ebe87] H. Eberle. *Development and analysis of a workstation computer.* PhD thesis, Institute for Computer Systems, ETH Zürich, 1987.

[EE99] Guy Eddon and Henry Eddon. *Inside COM+ Base Services.* Microsoft Press, 1999.

[Eng97] Robert Englander. *Developing Java Beans.* O'Reilly & Associates, Inc., 1997.

[Fis00] André Fischer. ETH Oberon Homepage. Web Page, 2000. http://www.oberon.ethz.ch/.

[FK96] Michael Franz and Thomas Kistler. The Juice Homepage. Web Page, 1996. http://caesar.ics.uci.edu/juice/.

[FK97] Michael Franz and Thomas Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, 1997.

[Fla97] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., second edition, 1997.

[FM98] André Fischer and Hannes Marais. *The Oberon Companion – A Guide to Using and Programming Oberon System 3*. vdf Hochschulverlag AG, ETH Zürich, 1998.

[Fra93] Michael Franz. Emulating an Operating System on Top of Another. *Software – Practice and Experience*, 23(6):677–692, 1993.

[GF96] Jürg Gutknecht and Michael Franz. Towards a Framework for Mobile Objects in Oberon – A Concept–Oriented Tour. Technical Report 96–55, Department of Information and Computer Science, University of California, 1996.

[GF99] Jürg Gutknecht and Michael Franz. *Implementing Application Frameworks: Object–Oriented Frameworks at Work*, chapter Oberon with Gadgets: A Simple Component Framework. John Wiley & Sons, 1999.

[Gri98] Frank Griffel. *Componentware – Konzepte und Techniken eines Softwareparadigmas*. dpunkt.verlag, 1998.

[GS98] Richard Grimes and Alex Stockton. *Beginning ATL COM Programming*. Wrox Press Ltd., 1998.

[Gut96] Jürg Gutknecht. Oberon, Gadgets and Some Archetypal Aspects of Persistent Objects. Technical Report 243, Institute for Computer Systems, ETH Zürich, 1996.

[Gut00a] Jürg Gutknecht. Active Oberon for .net. Web Page, 2000. http://www.oberon.ethz.ch/oberon.net/.

[Gut00b] Jürg Gutknecht. Oberon as an Implementation Language for COM Objects. In *Proc. of Joint Modular Languages Conference (JMLC). LNCS 1897*. Springer Verlag, 2000.

[Gut02] Jürg Gutknecht. Bluebottle Project. Web Page, 2002. http://bluebottle.ethz.ch/.

[GW92] Jürg Gutknecht and Niklaus Wirth. *Project Oberon – The Design of an Operating System and Compiler*. Addison–Wesley, 1992.

[HMP97] M. Hof, H. Mössenböck, and P. Pirkelbauer. Zero–Overhead Exception Handling using Metaprogramming. In *Proceedings SOFSEM'97*. Springer Verlag, 1997.

[Int01] Internet Assigned Numbers Authority. Media Types Directory. Web Page, 2001. http:///www.isi.edu/in-notes/iana/-assignments/media-types/.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection – Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[JW91] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report. ISO Pascal Standard*. Springer Verlag, 1991.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice–Hall, 1988.

[Lai00] Rolf Laich. COM Development Tools für Oberon for Windows. Master's thesis, Institute for Computer Systems, ETH Zürich, 2000.

[Lal94] Spyridon Gerassimos Lalis. *Hermes – Supporting Distributed Programming in a Network of Personal Workstations*. PhD thesis, Institute for Computer Systems, ETH Zürich, 1994.

[Lev00] John R. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.

[Lip96] Stanley B. Lippman. *Inside the C++ Object Model*. Addison–Wesley, 1996.

[LO98] Eric Ladd and Jim O'Donnell. *Using HTML 4.0, Java 1.1, and JavaScript 1.2*. Que Corporation, 1998.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison–Wesley, second edition, 1999.

[Mar96] Johannes L. Maris. *Design and Implementation of a Component Architecture for Oberon*. PhD thesis, Institute for Computer Systems, ETH Zürich, 1996.

[Mic95] Microsoft Corporation. The Component Object Model Specification. Web Page, 1995. http://www.microsoft.-com/com/.

[Mic00] Microsoft Corporation. Microsoft Developer Network Online. Web Page, 2000. http://msdn.microsoft.com/.

[Mic01] Microsoft Corporation. Overview of the .NET Framework. Web Page, 2001. http://-msdn.microsoft.com/library/en-us/cpguide/html/-cpovrintroductiontonetframeworksdk.asp.

[Mil02] Bartosz Milewski. Disk Thrashing & the Pitfalls of Virtual Memory. *Dr. Dobbs's Jounal*, 27(5):34 – 40, May 2002.

[Mor97] Mike Morgan. *Netscape Plug-Ins Developer's Kit*. Que Corporation, 1997.

[Moz00] The Mozilla Organization. Mozilla Plug-in API Guide. Web Page, 2000. http://www.mozilla.org/docs/plugin.html.

[Moz02] The Mozilla Organization. Demystifying Footprint. Web Page, 2002. http://www.mozilla.org/projects/footprint/-footprint-guide.html.

[Mul02] Pieter Muller. Native Oberon Operating System. Web Page, 2002. http://www.oberon.ethz.ch/native.

[Nel96] Mark Nelson. *The Data Compression Book*. M & T Books, second edition, 1996.

[Net96] Netscape Communications Corporation. The Java Runtime Interface. Web Page, 1996. http://home.netscape.com/-eng/jri/.

[Net98] Netscape Communications Corporation. Plug-in Guide. Web Page, 1998. http://developer.netscape.com/docs/-manuals/communicator/plugin/index.htm.

[NR69] P. Naur and B. Randell, editors. *Software Engineering*. NATO Scientific Affairs Division, 1969.

[Obe97] Oberon Microsystems. BlackBox Component Builder. Web Page, 1997. http://www.oberon.ch/prod/BlackBox/-index.html.

[Obj99] Object Managment Group. CORBA 2.3 Specification. Web Page, 1999. http://www.omg.org/.

[OHE95] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, 1995.

[Ope97] The Open Group. DCE 1.1: Remote Procedure Call – Universal Unique Identifier. Web Page, 1997. http://www.-opennc.org/onlinepubs/9629399/apdxa.htm.

[Osw94] Erich Oswald. Polyworlds. Master's thesis, Institute for Computer Systems, ETH Zürich, 1994.

[Osw00] Erich Oswald. *A Generic 2D Graphics API with Object Framework and Applications*. PhD thesis, Institute for Computer Systems, ETH Zürich, 2000.

[Paz97] Elan Paznesh. Gazelle: An Oberon/F Based Internet Developmen Framework. *The Oberon Tribune*, 2(1), 1997.

[PD82] S. Pemberton and M. C. Daniles. *Pascal Implementation, The P4 Compiler*. Ellis Horwood, 1982.

[Pet99] Charles Petzold. *Programming Windows*. Microsoft Press, 1999.

[Pla97] David S. Platt. *The Essence of OLE with ActiveX*. Prentice–Hall, 1997.

[Pow02] PowerQuest inc. PowerQuest: PartitionMagic product information. Web Page, 2002. http://www.powerquest.com/-partitionmagic/.

[PS98] František Plašíl and Michael Stal. An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. *Software – Concepts & Tools*, 19(1):14–28, 1998.

[Rad97] Timothy M. Radonich. Scramble Puzzle in JAVA. Web Page, 1997. http://www.whatrain.com/javatest/-scrampuzz/puzzle.html.

[Rav01] Ravenbrook Limited. The Memory Management Glossary. Web Page, 2001. http://www.memorymanagement.org/-glossary/.

[RlG02] Greg Roelofs and Jean loup Gailly. zlib. Web Page, 2002. http://www.gzip.org/zlib/.

[Rog96] Dale Rogerson. *Inside COM*. Microsoft Press, 1996.

[Saw96] Günther Sawitzki. Extensible Statistical Software: On a Voyage to Oberon. *Journal of Computational and Graphical Statistics*, 5(1):263–283, 1996.

[Saw99a] Günther Sawitzki. Diagnostic Plots for One–dimensional Data. Web Page, 1999. http://www.statlab.uni-heidelberg.-de/projects/onedim/.

[Saw99b] Günther Sawitzki. Software Components and Document Integration for Statistical Computing. In *Proceedings ISI Helsinki 1999 (52nd session)*. Bulletin of the International Statistical Institute, Tome LVIII, Book 2, 1999.

[Saw00] Günther Sawitzki. Project Voyager. Web Page, 2000. http://www.statlab.uni-heidelberg.de/projects/voyager/.

[Som96] Ralph Sommerer. *Integration of Online Documents*. PhD thesis, Institute for Computer Systems, ETH Zürich, 1996.

[SP00] Tom Schotland and Peter Petersen. Exception Handling in C Without C++. *Dr. Dobbs's Jounal*, 25(11):102 – 112, November 2000.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison–Wesley, third edition, 1997.

[Sun97] Sun Microsystems. JavaBeans 1.01 Specification. Web Page, 1997. http://www.javasoft.com/beans/docs/-beans.101.pdf.

[Sun98] Sun Microsystems. JAR – Java Archive. Web Page, 1998. http://java.sun.com/products/jdk/1.1/docs/guide/jar/.

[Sun99a] Sun Microsystems. JavaBeans Development Kit. Web Page, 1999. http://www.javasoft.com/beans/software/-index.html.

[Sun99b] Sun Microsystems. Using JavaBeans with Microsoft ActiveX Components. Web Page, 1999. http://java.sun.com/-products/plugin/1.1.1/docs/script.html.

[SW96] George Shepherd and Scot Wingo. *MFC Internals – Inside the Microsoft Foundation Class Architecture*. Addison–Wesley, 1996.

[Szy98a] Clemens Szyperski. *Component Software: Beyond Object–Oriented Programming*. ACM Press and Addison–Wesley, 1998.

[Szy98b] Clemens Szyperski. Emerging component software technologies – a strategic comparison. *Software – Concepts & Tools*, 19(1):2–10, 1998.

[Ulr96] David Ulrich. 3d–engine. Master's thesis, Institute for Computer Systems, ETH Zürich, 1996.

[VMW01] VMWare inc. VMWare Workstation. Web Page, 2001. http://www.vmware.com/pdf/ws-specs.pdf.

[Wir88] Niklaus Wirth. The Programming Language Oberon. *Software – Practice and Experience*, 18(7):671–690, 1988.

[Wir95] Niklaus Wirth. *Digital Circuit Design – An Introductory Textbook*. Springer Verlag, 1995.

[WM91] N. Wirth and H. Mössenböck. The Programming Language Oberon–2. *Structured Programming*, 12(4):179–195, 1991.

[You97] Douglas A. Young. *Netscape Developer's Guide to Plug-ins*. Prentice–Hall, 1997.

[Zel97] Emil J. Zeller. Seamless Integration of Online Services in the Oberon Document System. In *Proc. of Joint Modular Languages Conference (JMLC). LNCS 1024*. Springer Verlag, 1997.

Curriculum Vitae

Emil Johann Zeller

January 14, 1969	Born in Altstätten, SG, citizen of Appenzell, AI Son of Emil and Rösli Zeller–Baumgartner
1976 – 1981	Primary school in Oberriet
1982 – 1983	Secondary school in Oberriet
1984 – 1988	Kantonsschule Heerbrugg
1988	Matura Typus C
1989 – 1994	Studies in Computer Science at the Federal Institut of Technology, Zürich
1994	Dipl. Informatik–Ing. ETH
1994	Working for ASI Products AG
1994 – 2000	Teaching and research assistant at the Institut for Computer Systems at the Federal Institut of Technology, Zürich, in the research group of Prof. Dr. Jürg Gutknecht
2000 –	Collaboration in EU project Paper++
2001 –	Working for MCT Lab GmbH on a new Oberon based software development environment