UC Irvine
ICS Technical Reports

Title
Towards a framework for mobile objects in Oberon : a concept-oriented tour

Permalink
https://escholarship.org/uc/item/9hz4z330

Authors
Gutknecht, Jurg
Franz, Michael

Publication Date
1996-12-05

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

SL BAR
Z
699
C3
no.96-55

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Towards a Framework for Mobile Objects in Oberon - A Concept-Oriented Tour

Jürg Gutknecht and Michael Franz

Technical Report 96-55
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

5th December 1996

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Towards a Framework for Mobile Objects in Oberon – A Concept-Oriented Tour

Jürg Gutknecht, ETH Zürich
Michael Franz, UC Irvine

Abstract

The key topics of this article are object composition and mobility. Its main concern is a concept-based presentation of a simple component architecture that has evolved from project Oberon, an integrated language and system project in the heritage of Pascal and Modula. Some of the highlights specially focussed on are a notion of fully hierarchic compound objects, persistent object representation, object construction methods, mobile objects and, in particular, a new and very effective approach to portable code in combination with dynamic compilation. A novel combination of generic object interfaces and hierarchic message processing in compound objects provides a uniform solution to a diversity of archetypal problems.

Keywords: Object Composition, Object-Oriented Systems, End-User Objects, Persistent Objects, Portable Code, Just-in-time Compilation, Mobile Objects, Oberon.

1 From Object Oriented Languages to a Component Culture

Citing Brad Cox in [1], the current state of software manufacturing is "*like musket making was before Eli Whitney*". This metaphor vividly puts the current premature state of handcrafted software construction in contrast with a more advanced culture that is governed by component fabrication and reuse.

Undoubtedly the most promising approach to a component culture is the object-oriented programming paradigm. Object-oriented languages typically support the construction of class libraries and thereby serve the double purpose of a compositional framework for object classes and a production plant for object instances. However, in a component culture, at least two additional facilities rarely provided by bare programming languages are needed: (a) Persistent representation of individual objects outside of the runtime environment and (b) object composition.

For example, the multimedia panel in Figure 1 is a persistent composition of a number of components: A main panel, a movie sub-panel, two captions, two sliders and two textfields. The movie sub-panel is itself a composition of

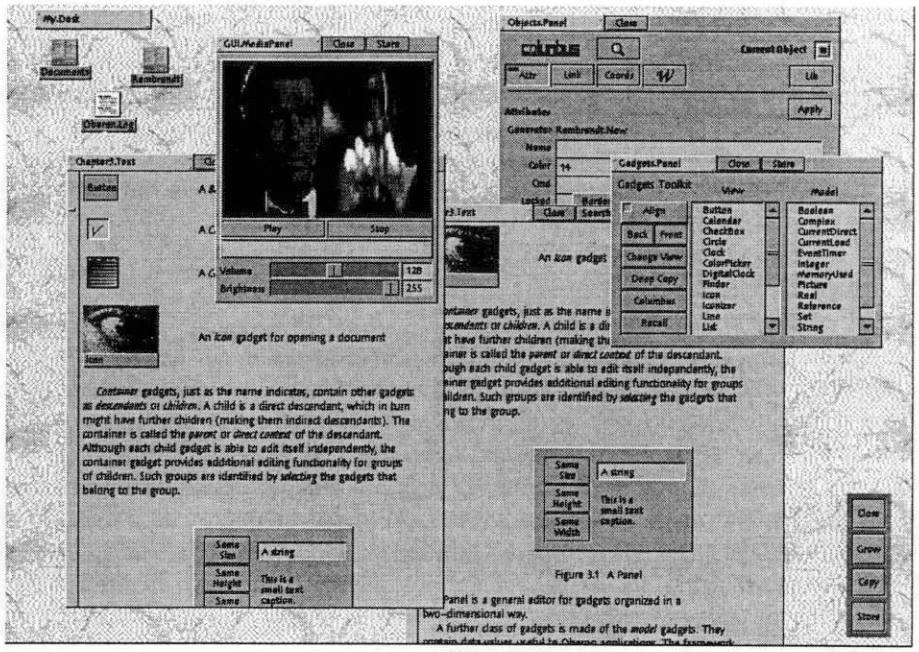

Figure 1: Sample Oberon Display Space with Multimedia Panel, Text Views, Gadgets Tool and Columbus Inspector on a Desktop

a scaling panel (auto-scaling its components), a video pad, two push-buttons and a textfield. It is important in this connection to point out the conceptual difference between *generic* objects like an empty panel, a caption, a slider, a textfield etc. that can be obtained directly from the system's class library by instantiating object types and *compound* objects like the multimedia panel and the movie sub-panel that need to be fabricated individually.

The example reveals the two-dimensional structure of the "space of software construction" in a component culture whose axes represent development of generic components and object composition respectively. From a methodical view the two axes are quite different. Developing generic components essentially amounts to object-oriented programming, i.e. to deriving subtypes from existing types, while object composition is a pure matter of static design specification. For example, a generic scaling panel is defined by a subtype *ScalingPanel* that has been derived from type *Panel* by adding some programmed auto-scaling functionality, while the specific movie sub-panel is an individual composition of a generic scaling panel, a video pad, two buttons and a text field.

2 Towards an Integrating Component Architecture

Despite its theoretical and practical relevance, object composition is not a well-established discipline of software engineering today. Nevertheless, numerous component architectures and related technologies have been developed, the better known among them being COM/OLE [2], OpenDoc [3] and CORBA [4]. We can get a representative cross section of the relevant topics in this field by simply enumerating the foci of these exemplary systems: Components, compound objects, compound documents and interobject communication.

The aim of this and the following sections is a systematic presentation of an alternative "light-weight" component architecture that has emerged from an evolutionary step of project Oberon [5], [6], [7]. The architecture distinguishes itself by a high degree of integration, conceptual uniformity and efficiency. It consists of four parts: (a) An integrating object management in the Oberon kernel, (b) a library of reusable object types, (c) a programming framework for the creation of customized object types and (d) a toolkit for interactive and descriptive construction of objects. The latter three parts together constitute the *Gadgets* package [8].

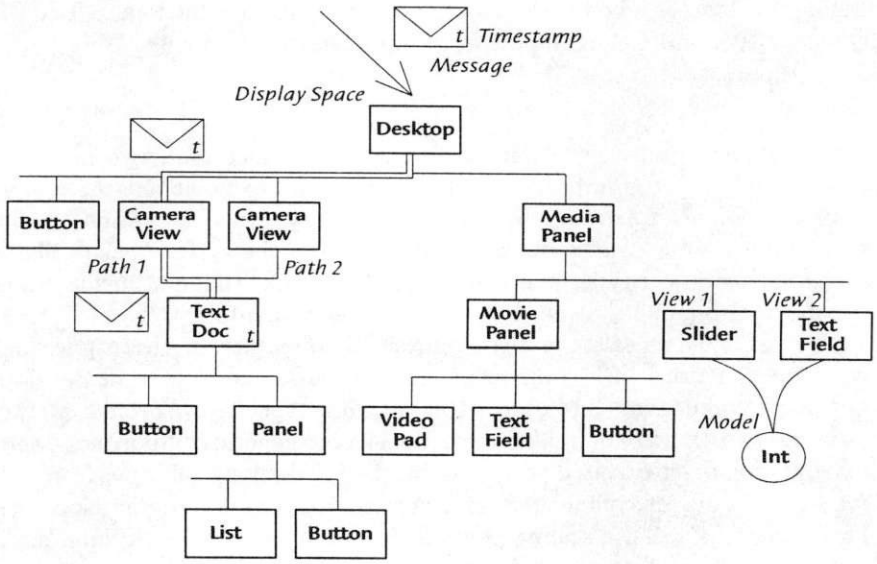

Figure 2: Data Structure of the Sample Display Space

The concept of *compound object* is fundamental in every component archi-

tecture. In favor of a concrete terminology, we restrict ourselves to compound objects of *container* type. Containers are particularly popular in visual contexts. Taking up Figure 1 again, we recognize an entire hierarchy of nested containers and a terminating *atomic* object: desktop → media panel → movie sub-panel → push button. Figure 2 shows an excerpt of this hierarchy in terms of a data structure.

The Principle of Parental Control

The role of containers is characterized in our framework by the single postulate of *parental control*, imposing on containers both full authorization and full responsibility for the management of their contents. This postulate has far-reaching consequences. First of all, it basically rules out any message traffic to content objects that by-passes their container. In other words, parental control indispensibly implies readiness of containers for message dispatching and request broking in-the-small.

The term *message traffic* needs clarification. As in every object-oriented environment, messages are used in our framework to specify and answer requests. However, static object interfaces as they are commonly provided by object-oriented languages are incompatible with the principle of parental control, at least in combination with generic containers that are prepared to include contents of potentially unknown (future) types and, correspondingly, to dispatch potentially unknown messages.

We make use of a novel kind of *generic object interface* that relies on subtyping applied to messages. The generic interface merely defines the base type of accepted messages and expects the dispatcher to discriminate arriving messages according to their subtype. If *Msg* is the base type of messages accepted by instances of some given object type, and *MsgA* and *MsgB* are subtypes of *Msg*, then the structure of the dispatcher is this:

```
PROCEDURE Dispatch (VAR M: Msg);
BEGIN
  (* common preprocessing *)
  IF M IS MsgA THEN (* handle message of subtype MsgA *)
    ELSIF M IS MsgB THEN (* handle message of subtype MsgB *)
    ELSE (* default handling *)
  END
END Dispatch;
```

Note in particular that the dispatcher (a) is able to do some common (pre)processing, (b) makes use of Oberon's safe runtime *type test IS* for message discrimination, (c) calls some default handler to handle common cases and (d) is extensible with respect to message subtypes without a need to change its interface.

For the sake of uniformity, we use generic interfaces for atomic objects as well. With that, message processing in the display space manifests itself as a hierarchic traversal of the message through dispatchers of the above kind. The ordinary case is *target-oriented* dispatching. However, there exist some

interesting strategic variations. For example, messages with no specific target are typically *broadcast* within the display space or within any one of its sub-spaces. Other variations concern *incremental processing* (incremental contributions by individual dispatchers) and *context dependent behavior* (behavior depending on the path the message arrives from). Applications will be given later in this section.

Camera Views

The *MVC scheme* [9] is a fundamental design pattern and an integral aspect of every object architecture that provides a conceptually clean separation of the concerns of *modelling*, *viewing* and *controlling* of objects. In our case, a very general interpretation of MVC is applied. A simple case is one or more *visual objects* (with a representation in the display space) serving as view of some *abstract object*. Examples of this kind are (a) checkbox view of a *Boolean* object, (b) slider view or textfield view (or both) of a *Integer* object and (c) color palette view of a color vector (red, green, blue).

A more intricate case is given by views of views, in the following called *camera views*. Camera views are useful for a number of purposes. They provide a conceptual frame for multiple views on one and the same visual document on one ore several display screens. For example, Figure 1 depicts a double view on some text document with integrated visual objects. An interesting variant of camera views are *functional views* that are combined with some specific functionality. For example, in addition to ordinary *user views*, special *constructor views* can be offered for the support of interactive editing and construction of a visual object or GUI in situ.

Camera views are implemented as a special kind of container object with the visual model acting as contents. As an immediate consequence and as shown in Figure 2, parts of the data structure representing the display space may thus be shared by camera views. Obviously, this adds both complexity and flexibility to message processing in the display space. Messages may now arrive at objects along different paths and therefore need to be *time-stamped* for detection of multiple arrivals. On the other hand, *context sensitive* processing is now possible and can be used beneficially, for example, to implement the above mentioned constructor views.

The following simplified examples of message processing in the display space may serve a better understanding of the concepts discussed in this Section and of their combination.

Update Message

Update messages are sent by models or controllers to notify potential views of a state change. They are always addressed to the display space as a whole with an implicit *broadcast request*. Affected views then typically reestablish consistency with their model after querying its actual state. Message broadcast is simpler and more generic than alternative methods such as callback lists but claims an efficiency penalty that, however, has proved to be not noticeable in practice.

Optimizations could easily be implemented, for example by adding knowledge to critical containers. We emphasize that generic message interfaces are absolutely essential for this broadcast method to be applicable within a strongly typed framework.

Display Message

This type of message is used to request a visual *target object* in the display space to display itself. For example, such a request would be issued by a reorganized container for every of its content objects or by the recipient of an update message to adjust its own display. Display requests are again addressed to the display space as a whole. They require incremental processing while traversing the container hierarchy in two respects: Successive accumulation of relative coordinates and successive calculation of an *overlap mask*. If camera views are involved, multiple paths may lead to the target object, so that it must be prepared for multiple arrivals of a message. All arrivals must be handled equally, though with different absolute coordinates and overlap masks.

Copy Message

Copying or *cloning* is an elementary operation on objects. Nevertheless, in the case of compound objects, it is quite intricate. Obviously, a generic copy operation on objects is equivalent with an algorithm to copy any arbitrary and truly heterogeneous data structure. Moreover, different possible variants of copies exist. For example, a *shallow copy* of a compound object consists of a real copy of the original container and new views on the original contents, while a *deep copy* typically consists of real copies of both container and contents.

Our implementation of the copy operation is again based on message broadcast. This time, multiple arrivals at an object must be handled with more care. The following is a rough sketch of copy message handling by a container:

```
IF first arrival of message THEN
  create copy of container;
  IF deep copy request THEN
    pass on message to contents;
    link copy of contents to copy of container
  ELSE (* shallow copy request *)
    link contents to copy of container
  END
END;
RETURN copy of container
```

Note that recipients in fact have some freedom in the handling of a copy request. For example, a "heavy-weight" object receiving a deep copy message could decide to return just some new view on it or even to return itself (leading to *copy by reference*) instead of a genuine copy.

3 Object Libraries as a Unifying Tool for Persistence

Object persistence is a trendy expression for a facility that allows individual objects to be stored in their current state on some external device (typically a disk). The essential part of every such facility is a method for the transformation of objects from their internal representation into an invariant, linear form (called *externalization*) and vice versa (called *internalization*). This problem is similar in its generic nature to the copy problem just discussed. However, there is one additional aspect: Invariant representation of pointer links.

Our approach to invariant pointer links is based on an institution of indexed sets of objects called *object libraries*. The idea is to implement object linearization by (recursively) registering components in some object library, thereby replacing pointers with reference indices. With that, externalization and internalization become distributed two-pass processes that again rely on broadcasting messages within the desired object:

The Externalizing Algorithm

```
Externalize(object X) =
   { Create(library L); Register(X, L); Externalize(L) }

Register (object X, library L) = {
  WITH X DO
*  FOR ALL components x DO Register(x, L) END
  END;
  IF X is unregistered THEN
    assign index and register X in L
  END }

Externalize (library L) = {
  WITH L DO
    FOR index i := 0 to max DO
      WITH object X[i] DO store generator of X[i];
*      replace pointer links with index references
          and externalize descriptor of X[i]
      END
    END
  END }
```

Note that the statements marked "*" must be implemented as object method because they are type-specific.

The Internalizing Algorithm

```
Internalize (library L) = {
  WITH L DO
    FOR index i := 0 to max DO
      load generator of X[i]; generate descriptor of X[i]
    END;
    FOR index i := 0 to max DO
*    internalize descriptor of X[i]
       and replace index references with pointer links
    END
  END }
```

Note that (in this simple version) the entity of internalization is a library rather than an individual object. Moreover, internalizing a library is a potentially recursive process, because indices in internalized object descriptors may refer to foreign libraries. Also note that the statement marked "*" must again be implemented as an object method.

The spectrum of applications of object libraries is much wider than mere externalization and internalization of individual objects. Object libraries are a simple form of *object data bases* and, as such, serve the purpose of organizing any local or distributed space of persistent objects. Some typical forms of their appearance are: (a) Collection of logically connected reusable *components*, (b) collection of *public* objects shared by a set of documents, (c) set of objects *private* to some document.

We conclude this Section with an interesting new and unifying characterization of *generalized texts*, i.e. multifont texts with possibly embedded objects like pictures, links, formatting controls and even functional units similar to *Java* "applets". With the background of object libraries, such a text is just a sequence of object references, where each reference is either an Ascii-code specifying a character pattern in some type font or an index into some public or private object library.

4 Interactive vs. Descriptive Object Composition

In principle, two different kinds of methods for object construction and composition exist: Interactive and descriptive. *Interactive methods* are based on direct editing in contrast to *descriptive methods* that typically rely on some formal language and a corresponding interpreter. The following table relates kinds of objects with suitable construction methods:

Kind of object	Suitable construction method
Irregular visual/GUI	interactive
Regular visual/GUI	descriptive
Abstract non-visual	descriptive
Program generated	descriptive

Independent of the construction method, components can be acquired alternatively from (a) generators for atomic objects, (b) generators for container objects and (c) prefabricated object libraries.

Interactive Construction

Our framework supports interactive construction on different levels. On the *tool level*, the *Gadgets* tool and the *Columbus inspector* tool shown in Figure 1 offer functionality for

- generating new instances of existing types
- calling prefabricated instances from object libraries
- aligning components in containers
- establishing model-view links
- inspecting state and attributes of objects
- binding Oberon commands to GUI objects

On the *view level*, the earlier mentioned developer views enable direct editing of visual objects. On the object level, support for in-situ editing is provided by built-in *local editors*. Mouse event messages are tagged with a pair of absolute mouse coordinates and undergo a location oriented dispatching in the display space. Because mouse events should be handled differently in user and developer contexts, most mouse event handlers make beneficial use of context-dependent message processing.

Descriptive Construction

Descriptive construction requires a formal description language as a basis. Two choices must be made: *Paradigm* and *syntactic style*. While the obvious paradigm in this case is *functional*, the choice of a style is more open. In the interest of internet compatibility, we decided for a HTML-kind of syntax.

We basically distinguish two modes of processing of an object description: (a) *compilation* and (b) direct *interpretation*. Separate descriptions are compiled into an object library, while descriptions floating in some HTML page are typically interpreted and translated directly into an inline object. Note as a fine point that such inline descriptions advantageously are wrapped by an *image tag*, so to enable non Oberon-speaking browsers to display a substitute at least.

The subsequent commented example of a separate description of the multimedia panel in Figure 1 may suffice to give an impression of the use of descriptive construction in our framework.

```
<LIBRARY ID="GUI">

  <OBJECT CLASSID="Gadgets.NewPanel" ID="MediaPanel">
    <PARAM NAME="Color" VALUE=13>
    <LIBRARY>
      <OBJECT CLASSID="Gadgets.NewInt" ID="Volume">
        <PARAM NAME="Value" VALUE=128>
      </OBJECT>
      <OBJECT CLASSID="Gadgets.NewInt" ID="Brightness">
        <PARAM NAME="Value" VALUE=255>
      </OBJECT>
    </LIBRARY>
    <VLIST ALIGN=CENTER>
      <OBJECT ID="Movie.Panel" WIDTH=240 HEIGHT=180>
        <PARAM NAME="Name" VALUE="Movie">
      </OBJECT>
      <TABLE BORDER CELLPADDING=2>
        <TR>
          <TD><OBJECT CLASSID="Gadgets.NewCaption">
            <PARAM NAME="Value" VALUE="Volume">
          </OBJECT>
          <TD WIDTH=80><OBJECT CLASSID="Gadgets.NewSlider">
            <PARAM NAME="Max" VALUE=255>
            <PARAM NAME="Cmd" VALUE="Movie.SetVol #Value Movie">
            <PARAM NAME="Model" VALUE="Volume">
          </OBJECT>
          <TD WIDTH=35><OBJECT CLASSID="Gadgets.NewTextField" HEIGHT=18>
            <PARAM NAME="Cmd" VALUE="Movie.SetVol #Value Movie">
            <PARAM NAME="Model" VALUE="Volume">
          </OBJECT>
        <TR>
          <TD><OBJECT CLASSID="Gadgets.NewCaption">
            <PARAM NAME="Value" VALUE="Brightness">
          </OBJECT>
          <TD WIDTH=80><OBJECT CLASSID="Gadgets.NewSlider">
            <PARAM NAME="Max" VALUE=255>
            <PARAM NAME="Cmd" VALUE="Movie.SetBright #Value Movie">
            <PARAM NAME="Model" VALUE="Brightness">
          </OBJECT>
          <TD WIDTH=35><OBJECT CLASSID="Gadgets.NewTextField" HEIGHT=18>
            <PARAM NAME="Cmd" VALUE="Movie.SetBright #Value Movie">
            <PARAM NAME="Model" VALUE="Brightness">
          </OBJECT>
      </TABLE>
    </VLIST>
  </OBJECT>

</LIBRARY>
```

Comments

(a) The compilation of this description results in a public object library called *GUI* containing one public object called *MediaPanel* or *GUI.MediaPanel* with its full name.

(b) Used in an inner scope the *LIBRARY* tag specifies an auxiliary private library of the containing object. In the example the multimedia panel uses an auxiliary library for the definition of two model objects coupling the slider and textfield views of volume and brightness control.

(c) Within the *OBJECT* tag, *CLASSID* and *ID* specify a generator procedure *M.P* (representing the type of the desired object) and a prefabricated library object *L.X* respectively.

(d) Components in a container object are either freely placed objects (at a specified position), vertical lists (*VLIST* with alignment specification), horizontal lists (*HLIST* with alignment specification) or tables (*TABLE* as in HTML).

(e) Unlimited nesting of lists and tables is conceptually possible.

(f) The layout structure of a panel is an intrinsic attribute. It is preserved under changes of size or shape.

(g) The *movie panel* is a prefabricated object that is imported from an object library called *Movie*. It is built from an auto-scaling panel, a custom video pad, two buttons and a text field. All other elements used in this construction are generic and imported from the *Gadgets* package.

5 From Persistent Objects to Mobile Objects

In the previous discussion we have developed our component architecture to a state that includes a mechanism for an external, linear representation of general objects. So far, we have used it for object persistence on secondary store only. However, there is a second potential application: *Mobility*. In this case, the linear representation must obviously be accompanied by (a) *self-containedness* and (b) *portability*. These notions are the topics of this and the next Section.

The essential point with *self-contained objects* is their completeness in terms of resources used. Thanks to our highly unified architecture, only two types of resources exist: *Object libraries* and *code modules* (packages).

Unfortunately, it is impossible for any authority to centrally find out the entirety of resources used by some general object. The reason is that resources are frequently specified dynamically at run-time by a name string. For example, command names *M.P* typically hide in attribute strings of push-buttons and object names *L.X* may well occur within any scrolling list.

Our solution consists of two new ingredients: (a) a *resource query message* used to collect resources and (b) a *resource management object* acting as a shrink-wrapper for self-contained objects.

IF X is any self-contained object and M is a resource management object, the shrink-wrapped composition MX is then externalized like this:

```
Externalize self-contained object MX = {
  Send resource query message Q to X;
  Externalize M;
  Externalize X }
```

In the case of containers, the broadcast strategy is again used beneficially for the dispatching of resource query messages:

```
Receive resource query message Q = {
  pass on Q to contents
  report own resources to resource manager }
```

In combination with *mobility*, there are some significant areas of problems behind the apparent simplicity of this algorithm. Among them are (a) scoping of resource names and (b) protection of the target system from malicious or erroneous program code. We now briefly touch problem (a), while we postpone a short remark on problem (b) to the next Section.

Mobile objects are developed in general without global coordination. They define their own scope of resources that, in case of a migration, need to be mapped to a separate space on the target system. However, it is reasonable to distinguish some global set of *kernel resources* that are assumed to be identically available on every target system. Obviously, kernel resources need not be transported with every individual object but need be checked for consistency, perhaps with the help of *finger prints* [10].

We emphasize that our approach to mobile objects is generic in the sense that any persistent object can in principle be made mobile. The spectrum of potential mobile objects therefore covers an impressive range: From simple buttons and checkboxes to control panels and documents and finally to desktops representing an entire Oberon system.

6 An Effective Approach to Portable Code

The final aspect of mobile objects that merits consideration is *cross-platform portability* of their implementation code. As mobile objects are expected to long outlive their creation environments and all currently existing hardware architectures, the choice of a software *distribution format* should be guided less by the present-day prevalence of specific processor families but rather by more fundamental considerations. While it might be a smart tactical decision at this moment to distribute mobile objects in the form of Intel 80386 binary code that can be executed directly on the vast majority of computers currently deployed (and interpreted on most of the others), this would be a bad choice in the long run, as the Intel architecture is nearing the end of its life-cycle. A future-oriented software distribution format needs to meet three primary requirements: It must be (a) well suited for a *fast translation* into native code of today's and future

microprocessors, it must (b) not obstruct advanced *code optimizations* required for tomorrow's super-scalar processors and, considering the anticipated importance of low-bandwidth wireless connectivity in the near future, it should be (c) *highly compact.*

Our concept of portable code incorporates a distribution format called *slim binaries* [11] that satisfies all of the above requirements. In contrast to approaches like *p-code* and *Java byte-code* [12] that are based on an underlying virtual machine, the slim binary format is an adaptively-compressed representation of *syntax trees.* In a slim-binary encoding, every symbol describes a sub-tree of an abstract syntax tree in terms of the sub-trees preceding it. Roughly spoken, the encoding process proceeds by successively adding sub-trees to the slim-binary representation and thereby steadily extending the "vocabulary" that is used in the encoding of subsequent program sections.

The proposed software distribution format has the obvious disadvantage that it cannot be decoded by simple pointwise interpretation. The semantics of any particular symbol in a slim-binary-encoded data stream is revealed only after all the symbols preceding it have been processed. Random access to individual instructions as it is typically required for interpreted execution is impossible.

However, in return for giving up the possibility of pointwise interpretation (whose value is limited due to low efficiency anyway), we gain several important benefits. First of all, our software distribution format is exceptionally compact. For example, it is more than twice as dense as Java byte-code and it performs significantly better than standard data compression algorithms such as *LZW* applied to either source code or object code (for any architecture). A reduction of network-transfer time of mobile components by a factor of two or more is an advantage that cannot be estimated high enough in view of the rising complexity of components in combination with a strong tendency towards wireless connections of low bandwidth. Moreover, experience has shown that on-the-fly code generation can be provided at almost zero run-time cost if a highly compact program representation is chosen. In fact, the additional computational effort can be compensated almost completely by a reduction of I/O overhead [13].

Second, the tree-based structure of our distribution format constitutes a considerable advantage if the eventual target machine has a super-scalar architecture that requires advanced *optimizations.* Many modern code-optimization techniques rely on structural information that is readily available on the level of syntax-trees but is lost on the level of byte-codes and is not reconstructable efficiently. For example, program code can often be improved by *instruction scheduling*, i.e. by reordering individual instructions so that functional units can operate in parallel. Our tree-based representation preserves all the control knowledge required for this and related optimizations, giving it an edge over linear representations requiring an additional pre-processing step.

Third, unlike most other representations, slim-binary encoding preserves *type* and *scope* information. It thus supports detection of violations by malicious or faulty code that potentially compromises the integrity of the host system. For

example, it is easy to catch any kind of access to private variables of public objects that may have been allowed by a rogue compiler. In the case of byte code, a corresponding analysis is very difficult.

A Pseudo-Monolithic Execution Environment

The *granularity* of compiled code pieces has a significant influence on the quality of the resulting machine code. In general, the finer grained the pieces are the less opportunities for optimizations they provide. Unfortunately, optimizations like procedure inlining and inter-procedural register allocation can often not be performed across boundaries of compilation units at a reasonable cost. Hence, software component systems that are made of a large number of relatively small and separate parts usually pay a performance penalty for added flexibility.

However, we can heal this undesired side-effect of modular programming by making inspirational use of our scheme of dynamic code generation. The key idea is to repeat the translation from one code format into another one. When a piece of slim binary code is initially loaded into the system, it is translated into native code in a single burst, trading code efficiency for compilation speed. After its creation, the piece of code is immediately subject to (a) execution and (b) optimization in the background. After completion of the optimization step, the previous generation of code is simply exchanged for the new one. Obviously, in conjunction with run-time profiling, this procedure can be iterated continually to produce ever better generations of code.

Note that optimization steps are time-uncritical because idle machine cycles can be used. Separate pieces of code can now be integrated into the current run time context, and far more aggressive optimization strategies can be employed in comparison with the primary compilation at loading time.

Quintessentially, dynamic re-compilation in a component environment can produce the same run-time efficiency as globally optimized monolithic applications typically show. This leads to a new execution model combining the advantages of two worlds: Traditional monolithic application programs and modular software.

Iterative run-time optimization integrates a collection of distributed software components into a single global and fully optimized code image without compromising the abstractions provided by modularization. On one hand, this relieves programmers of performance-related decisions in connection with modularization and, on the other hand, makes profitable use of idle machine cycles or temporarily unused processors in a multi-processor environment. As processor costs decline, it may become perfectly affordable to add an extra processor to a computer system specifically for the purpose of dynamic re-optimization.

Conclusion and Outlook

Building on the original version of the Oberon language and system we have developed a component framework whose most important design goals were conceptual clarity and uniformity. Architectural design highlights comprise an

integrated management for atomic and compound objects, a simple hierarchic data base facility for persistent object store, interactive and descriptive support for object construction and composition, a shrink-wrapper based generic solution for mobile objects of any granularity and a concept of portable code in combinaton with dynamic re-compilation.

Descriptive object construction is supported in our framwork by a separate package consisting of a HTML-compatible language, an interpreter and a compiler. In the interest of decoupling different concerns, we have consciously resisted any temptation to follow the trend of integrating object construction with type-oriented programming as, for example, Microsoft's *Visual C++* and Borland's *Delphi* do.

The system described is running on bare Intel PC platforms. Versions "hosted" by *Windows* and *Macintosh* platforms are also available. In its current state, the system includes all of the above mentioned local facilities, numerous applications and an advanced navigator that is designed to handle compound Oberon documents and objects separately or integrated in HTML hypertexts. Mobile objects and dynamic re-compilation are currently under development. Plans eventually aim at an "Oberon in a gadget" paradigm.

References

[1] Brad J. Cox; "Planning the Software Industrial Revolution"; IEEE Software; November 1990.

[2] K. Brockschmidt; "Inside OLE2"; Microsoft Press; 1993.

[3] The OpenDoc Design Team; "OpenDoc Technical Summary"; Apple Computer, Inc.; October 1993.

[4] R. Ben-Natan; "CORBA: A Guide to Common Object Request Broker Architecture"; Mc Graw-Hill; 1995.

[5] N. Wirth; "The Programming Language Oberon"; Software-Practice and Experience, 18(7):671-690; July 1988.

[6] N. Wirth and J. Gutknecht; "The Oberon System"; Software-Practice and Experience, 19(9); September 1988.

[7] J. Gutknecht; "Oberon System 3: Vision of a Future Software Technology"; Software-Concepts and Tools, 15:26-33; 1994.

[8] Johannes L. Marais; "Towards End-User Objects: The Gadgets User Interface System"; Advances in Modular Languages, volume 1 of Technology transfer series (Peter Schulthess, editor), Universitaetsverlag Ulm; September 1994.

[9] G.E. Krasner and S.T. Pope; "A Cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80"; Journal of Object-Oriented Programming, 1(3):26-49; August 1988.

[10] R. Crelier; "Extending Module Interfaces without Invalidating Clients"; Structured Programming, 16:1, 49-62; 1996.

[11] M. Franz and T. Kistler; "Slim Binaries"; Communications of the ACM, to appear, also available as Technical Report No. 96-24, Department of Information and Computer Science, University of California, Irvine; 1996.

[12] T. Lindholm, F. Yellin, B. Joy, and K. Walrath; The Java Virtual Machine Specification; Addison-Wesley; 1996.

[13] M. Franz; Code-Generation On-the-Fly: A Key to Portable Software; Doctoral Dissertation No. 10497, ETH Zurich, simultaneously published by Verlag der Fachvereine, Zürich, ISBN 3-7281-2115-0; 1994.

Acknowledgement

Our foremost thanks go to Niklaus Wirth who launched project Oberon in 1986 together with the first author of this article. Among the many contributions to the current state of the presented system we particularly acknowledge the work of Hannes Marais whose Gadgets framework has had a significant influence on the presented component architecture. We also gratefully acknowledge contributions by Emil Zeller, Ralph Sommerer, Patrick Saladin, Joerg Derungs and Thomas Kistler in the areas of compound and distributed documents, descriptive object composition and on-the-fly compilation respectively. And, last but not least, our sincere thanks go to Pieter Muller, implementor of the native PC Oberon kernel.