Oberon with Gadgets - A Simple Component Framework

Jürg Gutknecht, ETH Zürich
Michael Franz, UC Irvine

Abstract

We present a simple component framework that, "in a nutshell", addresses many of the archetypal aspects of component-oriented software environments, with a particular emphasis on homogeneity and unified concepts. Some of the topics focussed upon are a fully hierarchic notion of compound objects, persistent object representation, interactive and descriptive tools for object composition and self-contained and mobile objects. Methodological highlights are (a) a novel kind of generic object interfaces in combination with a message protocol that strictly obeys the principle of parental control, (b) a multi-purpose concept of indexed object libraries and (c) an alternative approach (compared to the Java virtual machine) to portable code, accompanied by dynamic compilation. Our framework is based on and integrated into Oberon, a language and system in the heritage of Pascal and Modula that runs both natively on Intel-based PCs or on top of a commercial operating system. Of the many projects having made use of our framework so far, three applications of a pronounced interdisciplinary character shall be mentioned briefly.

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

1 From Object Oriented Languages to a Component Culture

Perhaps the most important conceptual cornerstone of today's hardware industry is its pervasive component culture. Devices are typically composed of functional components of all kinds and granularities that have been developed and fabricated by highly specialized teams, possibly elsewhere. Interestingly, no similar culture has been able to establish itself in the software industry, probably for lack of (a) a well-defined and globally accepted notion of interface and (b) a corresponding "market".

In [1] Brad Cox considers a more advanced component culture as paramount for future stages of software development and use. From this perspective, we have developed an experimental component-oriented framework that, "in a nutshell", deals with many of the archetypal aspects of component-orientation. Our framework integrates smoothly with the original Oberon language and system as described in [2, 3, 4] but goes beyond the ordinary object-oriented level in three respects: (a) Persistent representation of individual objects in their current state outside of the runtime environment, (b) construction and composition of individual objects and (c) object mobility.

To put it differently: While the underlying object-oriented language provides both a compositional framework for object classes (allowing the derivation of specialized classes) and a production factory for generic object instances, our component-oriented framework in addition supports the construction, maintenance, organization and reuse of individual, prefabricated components. The following example may illustrate our point.

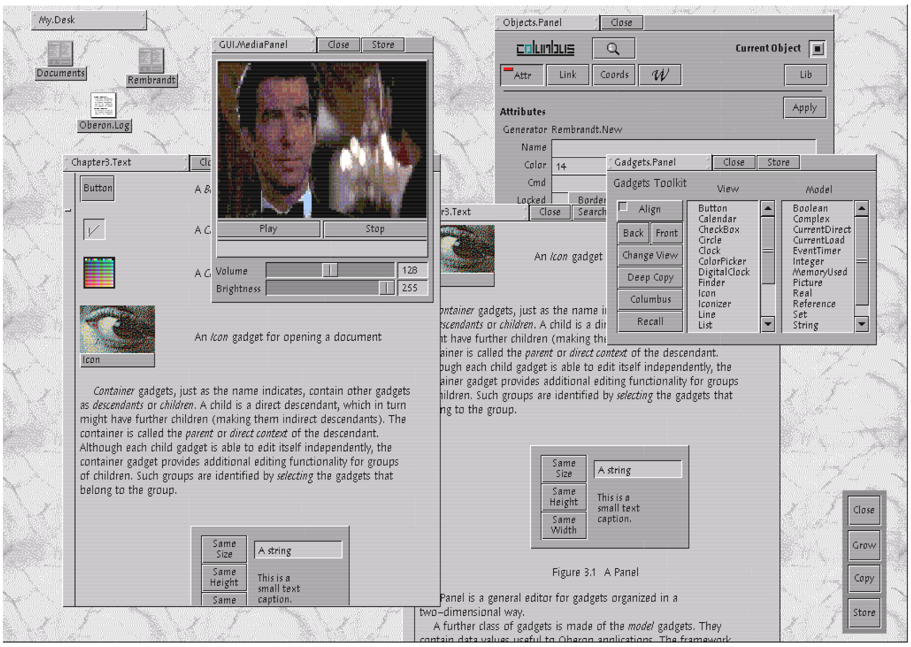

Figure 1: Sample Oberon Display Space with Multimedia Panel, Text Document Views, Gadgets Tool and Columbus Inspector on a Desktop

Figure 1 shows a snapshot of the Oberon display screen. The multimedia panel in the upper left quarter 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 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 again the conceptual difference between *generic* objects like an empty panel, a caption, a slider, a text field etc. that can be obtained directly from the system's class library by instantiation and *prefabricated* objects like the movie sub-panel and the multimedia panel itself.

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 construction and composition* respectively. From a methodical view the two axes are orthogonal. Developing generic components is class-centered. It essentially amounts to object-oriented programming within a given *framework* of classes, i.e. to deriving subclasses from existing classes. In contrast, object composition is instance-centered and primarily a static matter of design specification. For example, a generic scaling panel is defined by a subclass *ScalingPanel* that has been derived from class *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 A "Light-Weight" Component Framework

Different software component systems are available today. Among the most prominent ones are Microsoft's COM/OLE/ACTIVE-X [8] and Sun's JavaBeans [9]. In addition, technologies like OpenDoc [10] and OMG's CORBA [11] address related aspects like compound documents and standardized client-/server-communication respectively.

In the following sections we present an alternative "light-weight" component framework. We shall orientate our presentation according to four topics that we consider as (a) essential to the field of component-software in general, (b) building the conceptual base of and "connecting glue" in our framework and (c) being solved somehow originally in our system. These topics are (1.) message protocols in compound objects, (2.) object data bases, i. e. persistent representation of object collections, (3.) object composing tools and (4.) self-containedness and mobility.

3 Message Protocols in Compound Objects

The concept of *compound object* is fundamental in every component architecture. In favor of a concrete terminology, we restrict ourselves to compound objects of *container types* that are particularly popular as units in graphical user interfaces. 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. We emphasize that our rigorous hierarchic view pays in terms of a highly uniform object model, where coarse-grained objects like desktops, medium-grained objects like panel and fine-grained objects like push-button are completely unified.

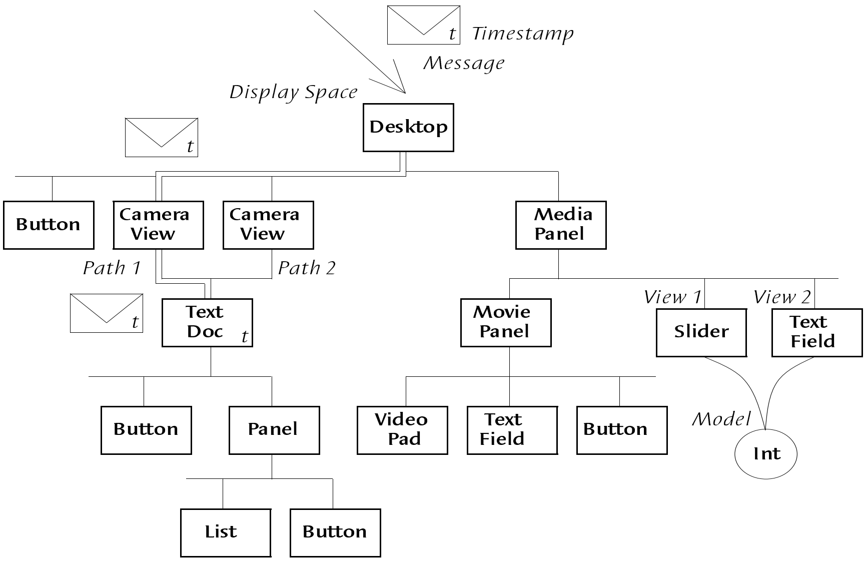

Figure 2: Data Structure of the Sample Display Space

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 brokering 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.

For this reason we make use of a novel kind of *generic object interface* that relies on a built-in interpreter which is able to interpret and dispatch arbitrary arriving messages appropriately. Technically, if *Message* is the base type of messages accepted by instances of some given object type, and *MessageA* and *MessageB* are subtypes of *Message*, then the structure of the dispatcher is this:

```
PROCEDURE Dispatch (me: Object; VAR M: Message);
BEGIN
  (* common preprocessing *)
  IF M IS MessageA THEN (* handle message of subtype MessageA *)
    ELSIF M IS MessageB THEN (* handle message of subtype MessageB *)
    ELSE (* default handling *)
  END
END Dispatch;
```

Note in particular that the dispatcher (a) is able to perform 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 any need to change its interface.

For the sake of uniformity, we use generic interfaces for atomic (i.e. non-container objects) as well. With that, message processing in the display space manifests itself in a hierarchic traversal of the structure, directed by dispatchers of the above kind. The ordinary case is *target-oriented* dispatching, where the target-object is specified as part of the message. Typical examples of target-oriented messages are requests to get, set and enumerate attributes ("properties") of a certain object. However, interesting strategic variations of message dispatching exist. For example, universal notifications without a specific target are typically *broadcast* within the display space or within any one of its sub-spaces. Other options concern *incremental processing* (incremental contributions to message handling by individual dispatchers) and *context dependent behavior* (behavior depending on the path the message arrives from). Applications will be given in the subsequent sections.

Camera Views

The *MVC scheme* [12] 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 *developer 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 visual objects that are able to display visual models. As a desirable consequence, common parts of the data structure representing the display space are automatically shared by camera views and unnecessary duplication is avoided as shown in Figure 2. 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 developer views.

The following simplified examples of message processing in compound objects in general and in the display space in particular may promote a better understanding of the concepts discussed in this section and of their combination.

Update Notifications

Update notifications are sent by models or controllers (in the case of smart bundling) 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 Requests

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 are handled in the same manner, albeit with different absolute coordinates and overlap masks.

Copy Requests

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 *deep copy* of a compound object consists of a real copy of both the original container and its contents, while a *shallow copy* typically would just include new views on the original 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 as a fine point 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 merely return some new view on itself or even to return itself (leading to *copy by reference*) instead of a genuine copy.

4 Object Libraries as a Versatile and Unifying Concept

Object persistence is a trendy expression for a facility that allows individual objects to be stored on some external device (typically a disk) in their current state. The essential part of every such facility are two transformation methods called *externalizer* and *internalizer* respectively. Externalizers are used to transform objects from their internal representation into an invariant, linear form, internalizers are used for the inverse transformation. The problems of externalizing and internalizing are similar in their generic nature to the copy problem just discussed. However, there is one additional aspect to be considered: 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 of 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 }
```

Obviously, acyclicity of the relation of containment is a precondition for this algorithm. Further note that the statements marked "*" must be implemented as object methods 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 internalizing a library is a potentially recursive process, because indices in internalized object descriptors may refer to foreign libraries. And, again, the statement marked "*" must be implemented as an object method.

Object libraries are a surprisingly versatile and thereby unifying concept. The spectrum of their application is much wider than one would perhaps expect. Beyond supporting externalization and internalization of individual objects they are simple *object data bases* that serve the purpose of organizing any local or distributed space of objects. Some typical manifestations are: (a) Collection of logically connected reusable *components*, (b) collection of *public* objects shared by a set of documents and (c) set of objects *private* to some document.

Objects Flowing in Text

Another unifying application of the concept of object libraries are *generalized texts*. A simple reinterpretation of ordinary (multifont) texts as sequences of references to character patterns (where the reference numbers are Ascii-codes) leads the way to a far-reaching generalization. By allowing references to arbitrary object libraries instead of just to fonts, we immediately get to texts with integrated objects of any kind, including pictures, links, formatting controls, entire functional panels and other units that are similar to *Java* "applets". Additional flexibility is provided by the possibility to embed both private objects (collected in the so-called *private library* of the text) as well as public objects (belonging to some public library).
Such a high degree of integration of text with objects has proved to be incredibly useful mainly in the area of documentation. Thanks to it, functional units of any complexity and granularity that have been developed anywhere can simply be copied to and integrated with their textual documentation. An illustrative example is the chapter of the electronic Oberon book shown in Figure 1.

5 Object Composition Tools

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 with *descriptive methods* that typically rely on some formal language and a corresponding interpreter. In most cases, the two methods are interchangeable. However, while the interactive method is more suitable for the construction of visual GUI-objects, the descriptive method is preferable for the construction of regular layouts and indispensible for program-generated objects (such as "property-sheets" etc.) and for non-visual (model) objects.
The following table summarizes:

Kind of object	Suitable construction method
Visual GUI	interactive
Regular layout	descriptive
Program generated	descriptive
Non-visual model	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* toolkit [5, 6, 7] and the *Columbus inspector* tool shown in Figure 1 offer functionality for

- generating new instances of an existing type

- calling prefabricated instances from an object library
- aligning components in containers
- establishing model-view links
- inspecting state, attributes and properties of objects
- binding Oberon commands to GUI objects

On the *view level*, the earlier mentioned developer views enable editing of visual objects. On the object level, support for in-place 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. Because layout specification is functional, we decided in favor of a functional language with a LISP-like syntax.

We basically distinguish two modes of processing of a functional object description: (a) *compilation* and (b) direct *interpretation*. Separate descriptions are compiled into an object library, while descriptions that are embedded in some document context (e. g. in a HTML page) are typically interpreted and translated directly into an inline object. Figure 3 visualizes the compiling mode. Note that generic objects are retrieve by the compiling composer from the class library by cloning, while prefabricated objects are imported from any object library either by cloning or by reference.

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

```
(LIB GUI
  (FRAME MediaPanel (OBJ Panels.NewPanel)
    (Volume (OBJ BasicGadgets.NewInteger (Value 100)))
    (Brightness (OBJ BasicGadgets.NewInteger (Value 200)))
    (GRID 2:50 1:* @ 1:25% 1:50% 1:25%)
    (PAD 2 @ 2)
    (FRAME (POS 1 @ 1) (OBJ TextFields.NewCaption)
      (Value "Brightness"))
    (FRAME (POS 1 @ 2) (OBJ BasicGadgets.NewSlider)
      (Max 255)
      (Model Brightness)
      (Cmd "Movie.SetBright #Value Movie"))
    (FRAME (POS 1 @ 3) (OBJ TextFields.NewTextField)
```

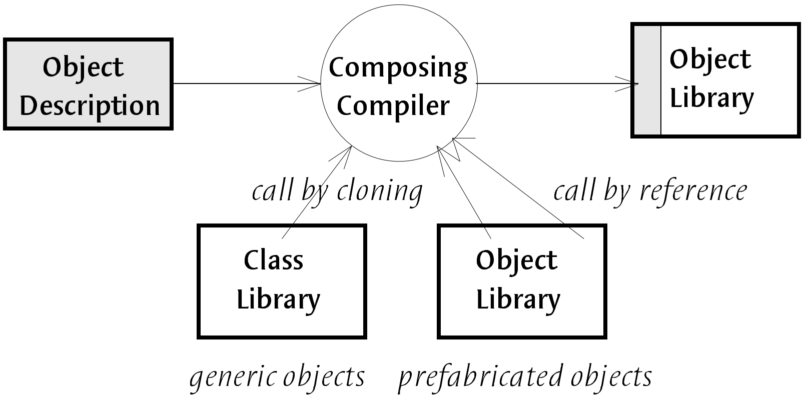

Figure 3: Construction of Objects by Compilation of a Functional Description

```
    (Model Brightness)
    (Cmd "Movie.SetBright #Value Movie"))
  (FRAME (POS 2 @ 1) (OBJ TextFields.NewCaption)
    (Value "Volume"))
  (FRAME (POS 2 @ 2) (OBJ BasicGadgets.NewSlider)
    (Max 255)
    (Model Volume)
    (Cmd "Movie.SetVol #Value Movie"))
  (FRAME (POS 2 @ 3) (OBJ TextFields.NewTextField)
    (Model Volume)
    (Cmd "Movie.SetVol #Value Movie"))
  (FRAME (POS 3 @ 1:3) (OBJ MoviePanel.Movie)
    (SIZE 296 @ 246))))
```

Comments

(a) A (compound, visual) object is generally specified as a nested hierarchy of *frames*, where each frame may optionally use the OBJ-clause to define a carrier-object.

(b) The compilation of the above declaration results in an object library file called *GUI* containing an instance of the constructed object called *MediaPanel*.

(c) Frames may optionally declare local objects that are typically used as models. In the example, two such model objects are declared, one for volume control and one for brightness control.

(d) Within the *OBJ* construct, the first identifier specifies either a generator procedure *Module.Procedure* (representing the class of the desired object) or a prefabricated object *Library.Object* that is *imported* by reference or copy from an object library.

(e) Visual objects typically specify a *grid* of type rows @ columns. In our case the grid consists of three rows and three columns respectively. The heights of the first two rows are 50, while the height of the third row is generic, i. e. determined by the contents. Column widths are indicated in percents, where the total width is generic again. The first two rows from the bottom represent brightness and volume control respectively. Each row consists of a caption, a slider and a text field, where the slider and the text field are coupled by a local model. The third row has a column-span of three and displays a prefabricated object called *MoviePanel* that is imported from a library called *Movies*.

We should note that the functional object description tool just discussed is only the leaf-end of a much more general experimental system called *Powerdoc* that allows the descriptive construction of arbitrarily general documents that may include any kind of passive or active objects ("applets") and data streams.

6 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 this facility 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 request to *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 *program modules* (units of the class library).

Unfortunately, it is impossible for any central authority to find out the entirety of resources used by some general object. The reason is that resources are frequently specified implicitly by an inconsipcuous name string. For example, command names *Module.Procedure* typically hide in attribute strings of push-buttons and object names *Library.Object* may well occur within any scrolling list.

Our solution to the resource detection problem 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 asking X to report its resources;
  Externalize M;
  Externalize X }
```

In the case of containers, the broadcast strategy is again used beneficially for the dispatch 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 any global coordination. As a consequence, they define their own scope of resources that, in case of a migration, needs 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 to be checked for consistency, perhaps with the help of *fingerprints* [13].

We emphasize that our approach to mobile objects is generic in the sense that any persistent object can be made mobile in principle. 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.

7 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 i80386 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. 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* [14] that satisfies all of the above requirements. In contrast to approaches like *p-code* and *Java byte-code* [15] 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 externalizing sub-trees and simultaneously steadily extending the "vocabulary" that is used in the encoding of subsequent program sections.

This 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). This is an advantage that cannot be estimated high enough. In fact, experience has shown that on-the-fly code generation can be provided at almost zero cost if a highly compact program representation is chosen, since the additional computational effort can be compensated almost completely by a reduction of I/O overhead [16].

Second, the tree-based structure of our distribution format constitutes a considerable advantage if the eventual target machine has an 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 more difficult to extract on the level of byte-codes.

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 more difficult.

Fourth, we can make use of our slim-binaries technology to optimize code across module boundaries. Such global optimizations as, for example, procedure inlining and inter-procedural register allocation, pay particularly well in a software component environment that is made of a large number of relatively small and separate parts of code.

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.

Applications

Our framework has been used in numerous projects. Three substantial interdisciplinary applications are a *Computer Aided Control System Design* (CACSD) environment [17], a generic *robot controller architecture* [18] and a *real-time audio-/video-stream service* in a local switch-based network. Noteworthy technical highlights in the CACSD environment are matrix- and plot-gadgets that are coupled (behind the scenes) with a Matlab engine and powerful tree-node objects that abstract control system design actions in an action-tree. In the robot controller project object libraries are beneficially used for a persistent but decoupled representation of *I/O-objects*, i. e. sensors, actuators etc.. Another interesting facility are *remote views*, i.e. views that, on the development system (connected to the robot via Internet), display state models of the robot. Finally, in the stream-server project, a new kind of visual objects displaying remotely supplied contents (video streams, not passing the client's memory) has been integrated smoothly into Gadgets.

Conclusion and Outlook

Building on the original version of the Oberon language and system, we have developed a component framework that, strictly speaking, splits into three interacting sub-frameworks:

- a framework of user interface components that can be configured interactively or descriptively
- a framework for externalizing and internalizing object structures
- a framework for shrink-wrapping object structures in a self-contained way

The sub-frameworks are connected by two universal and unifying facilities, (a) a "software bus" in the form of a generic message protocol obeying the principle of parental control and (b) an object data base in the form of a hierarchy of indexed object libraries.

In contrast to *COM* and *Corba*, emphasis was put in our system on homogeneity rather than on platform independence. In fact, Oberon components are not viable in any environment apart from Oberon, with the important exception of a HTML context with an Oberon plug-in. On the other hand, the range of Oberon components covers the whole bandwidth from simple character glyphs to entire documents and desktops.

Our architecture further distinguishes itself by a clear decoupling of object composition from component development. While generic components (atomic ones and containers) are represented by Oberon classes and programmed in Oberon (essentially by implementing the message protocol of parental control), compound objects are created interactively under usage of built-in in situ editors or descriptively in a LISP-like notation. Unlike, for example, Microsoft's Visual Basic and Developer Studio wizards, Borland's Delphi and Sun's *JavaBeans*, the Oberon composer tools do not map compound objects to classes and constructors but directly create data structures of DAG-type that are able to externalize to identifyable instances of some object library. Each compound object can therefore be called and used alternatively by reference or by cloning.

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

Several substiantial projects have made intensive use of our component framework. They have demonstrated not only its power and limits but also an ample potential for further generalizations.

References

[1] Brad J. Cox; "Object Oriented Programming, An Evolutionary Approach"; Addison Wesley, 1986.

[2] N. Wirth, "The Programming Language Oberon", Software – Practice and Experience, 18(7), 671-690.

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

[4] N. Wirth and J. Gutknecht, "Project Oberon", Addison-Wesley, 1992.

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

[6] 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.

[7] A. Fischer and H. Marais, "The Oberon Companion", A Guide to Using and Programming Oberon System 3, vdf Hochschulverlag AG an der ETH Zürich, 1998.

[8] K. Brockschmidt, "Inside OLE"; Microsoft Press; 1993.

[9] J. Feghhi, “Web Developer’s Guide to Java Beans,
Coriolis Group Books, 1997.

[10] The OpenDoc Design Team, “OpenDoc Technical Summary”,
Apple Computer, Inc.; October 1993.

[11] R. Ben-Natan, “CORBA: A Guide to Common Object Request Broker Architecture”,
Mc Graw-Hill; 1995.

[12] 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.

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

[14] M. Franz and T. Kistler, “Slim Binaries”,
Communications of the ACM, 40:12, 87-94; December 1997.

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

[16] 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.

[17] X. Qiu, W. Schaufelberger, J. Wang, Y. Sun, “Applying O3CACSD to Control System Design and Rapid Prototyping”,
The Seventeenth American Control Conference (ACC’98), Philadelphia, USA, June 24-26, 1998.

[18] R. Roshardt, “Modular robot controller aids flexible manufacturing”,
Robotics Newsletter, International Federation of Robotics, No. 16, Dec. 1994.

Acknowledgement

Our thanks go to Niklaus Wirth, without whose initiative and dedication the original Oberon language and system would not have been built. Among the numerous contributions to the current state of the presented system, we particularly acknowledge the work of Hannes Marais whose Gadgets framework and toolkit has had led the way to go. 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 compiler construction respectively. Our sincere thanks also go to Pieter Muller, implementor of the native PC Oberon kernel. And, last but not least, we are most grateful for many constructive comments by the anonymous referees.