Differences between Experimental Oberon and Original Oberon

Andreas Pirklbauer

12.12.1990 / 22.11.2016

Experimental Oberon[1] is a revision of the Original Oberon[2] operating system, containing several simplifications, generalizations and functional enhancements. Some modifications are purely of experimental nature, while others serve the explicit purpose of exploring potential future extensions, for example to add support for touch display devices.

1. Continuous fractional line scrolling with variable line spaces

Continuous fractional line scrolling has been added, enabling completely smooth scrolling of texts with variable lines spaces and dragging of entire viewers with continuous content refresh. Both *far* (positional) and *near* (pixel-based) scrolling are realized[3]. To the purist, such a feature may represent an "unnecessary embellishment" of Oberon, but it is simply indispensable if the system is to support touch display devices where a mouse is absent and viewers may not have scrollbars. In such an environment, continuous scrolling is the only (acceptable) way to scroll and presents a more natural user interface. As a welcome side effect, the initial learning curve for users new to Oberon is *considerably* reduced.

2. Multiple logical display areas ("virtual displays")

Original Oberon was designed to operate on a *single* abstract logical display area, which is decomposed into a number of vertical *tracks*, each of which is further decomposed into a number of horizontal *viewers*. Experimental Oberon adds the ability to create *several* such display areas on the fly and to seamlessly switch between them. The extended conceptual hierarchy of the display system consists of the triplet *(display, track, viewer)*. The Oberon base module *Viewers* exports routines to add and remove logical *displays*, to open and close *tracks* within displays and to open and close individual *viewers* within tracks. There are no restrictions on the number of displays, tracks or viewers that can be created. In addition, text selections, central logs and focus viewers are separately defined for each display area. The scheme naturally maps to systems with multiple *physical* monitors. It can also be used to realize super-fast context switching, for example in response to a swipe gesture on a touch display device.

The command *System.OpenDisplay* opens a new logical display, *System.CloseDisplay* closes an existing one, while *System.ShowDisplays* lists all active logical displays. The command *System.Clone*, displayed in the title bar of every menu viewer, opens a new logical display on the fly *and* displays a copy of the initiating viewer *there*. The user can toggle between the two copies of the viewer (switch displays) with a single mouse click[4].

---

[1] *http://www.github.com/andreaspirklbauer/Oberon-experimental (adapted from the original implementation prepared by the author in 1990 on a Ceres computer at ETH)*
[2] *http://www.inf.ethz.ch/personal/wirth/ProjectOberon/index.html (Original Oberon, 2013 Edition); see also http://www.projectoberon.com*
[3] *The system automatically switches back and forth between the two types of scrolling based on the horizontal position of the mouse pointer.*
[4] *By comparison, the Original Oberon commands System.Copy and System.Grow create a copy of the original viewer in the same logical display area (System.Copy opens another viewer in the same track, while System.Grow extends the viewer's copy over the entire column or display, effectively lifting the viewer to an "overlay" in the third dimension).*

Alternatively, the user can select the command *System.Expand*, also displayed in the title bar of every menu viewer, to expand a viewer "as much as possible" by reducing all other viewers in the track to their minimum heights, and switch back to any of the "compressed" viewers by simply clicking on *System.Expand* again in any of their (still visible) title bars.

3. Unified viewer concept and enhanced viewer management

The previously separate notions of *frame* and *viewer* have been *united* into a single, unified *viewer* concept, in which the nesting properties of the original *frame* concept have been fully preserved. The original distinction between *frames* and *viewers* (a viewer's state of visibility) appeared to be rather marginal and hardly justified in Oberon's display system, which is based on hierarchical tiling rather than overlapping windows. We note that the unified viewer concept is not only simpler, but also more flexible than the original scheme, as *each* sub viewer (called a *frame* in Original Oberon) can now have its *own* state of visibility[5]. This makes it possible to bring the concept of *hierarchical tiling*, which is already used for *top-level* viewers in Original Oberon, all the way down to the sub viewer level as well, enabling "newspaper-style" *multi-column* text layouts for example.

The operations *Viewers.Change, MenuViewers.Modify* and *TextFrames.Modify* have been generalized to handle *arbitrary* viewer modifications, including pure vertical translations (without changing the viewer's height), simultaneously modifying a viewer's bottom line, top line *and* height, and moving multiple viewers with a *single* mouse drag operation.

4. Simplified viewer message type hierarchy

A number of basic Oberon viewer message types (e.g., *ModifyMsg*) and identifiers (e.g., *extend, reduce)* have been eliminated, leading to a more consistent message hierarchy across the entire system. The remaining viewer message types and identifiers now have a single, well-defined purpose. For example, restoring a viewer is accomplished exclusively by means of a *restore* message identifier.

Several viewer message types (or minimal subsets thereof) that appeared to be generic enough to be made generally available to *all* viewer types have been merged and moved from higher-level modules to module *Viewers*, resulting in fewer module dependencies in the process. Notably, module *TextFrames* no longer depends on module *MenuViewers*, making it now possible to embed text viewers into *other* types of composite viewers, for example a viewer consisting of an *arbitrary* number of text, graphic or picture sub viewers.

5. Safe module unloading[6]

The semantics of *unloading* a module has been refined as follows: By default, a module is unloaded and its associated memory is released only if no clients *and* no references to the module exist in the remaining modules and data structures. Such references can be in the form of *type tags* (addresses of type descriptors) in *dynamic* (heap) objects of other loaded

---

[5] For top-level viewers, the field 'state' in the viewer descriptor is interpreted by the viewer manager, whereas for sub viewers it is interpreted by the enclosing viewer.

[6] Historically, module unloading has been handled in a variety of ways in the Oberon system: In Ceres-Oberon, type descriptors are allocated dynamically in the heap at load time and survive beyond the lifetime of their associated module, allowing a module to be physically removed from memory while still enabling other modules to access its type descriptors. This covers the case where a structure rooted in a variable of base type T declared in a module M contains elements of an extension T' defined in (an unloaded) module M'. Access to an unloaded module in Ceres-Oberon (e.g. via dangling procedure variables pointing to the unloaded module) results in a trap on Ceres-1 and Ceres-2, which both use virtual memory (safe module unloading), but goes undetected on Ceres-3, which does not use virtual memory and where code space may therefore be reused (unsafe unloading). In MacOberon 1990, which does not use virtual memory, type descriptors are placed in the module's static block, but a module is never unloaded from memory; instead it is removed only from the list of loaded modules (safe unloading, where however the memory associated with previously loaded modules is never released). In Original Oberon 2013, which does not use virtual memory, type descriptors are also placed in the module's static block, but modules are physically removed from memory upon unloading; therefore, the freed module block (including the area reserved for type descriptors) can be overwritten by modules loaded later, potentially leading to an unstable system state (unsafe unloading).

modules pointing to descriptors of types declared in the module to be unloaded, or in the form of procedure variables installed in *static* (global) or *dynamic* objects of other loaded modules referring to procedures declared in that module[7].

If there *are* references from other modules, however, the module is *not* unloaded, but is removed only from the *list* of loaded modules *without* releasing its associated memory, provided that the *force* option is specified in the unload command *(System.Free M/f)*[8]. Such *hidden* modules (marked with an asterisk in the output of *System.ShowModules*) are later *automatically* removed as soon as there are *no more* references to them. To achieve this, the background task handling garbage collection includes a call to the new command *Modules.Collect*, which collects no longer referenced *hidden* modules. Thus, module data is kept in memory as long as needed and removed from memory as soon as possible.

Dynamic references to a specified module are checked by employing a simple *mark-scan* scheme suitably tailored to this task. In the *mark* phase, the *dynamic* records reachable by *all other* currently loaded modules are marked. This includes records *also* reachable by the specified module itself, but excludes records *only* reachable by that module. The subsequent *scan* phase scans the heap element by element, unmarks marked objects and verifies whether the *type tags* of the encountered (marked) records point to descriptors of *types* declared in the module to be unloaded, or whether *procedure variables* in these records refer to *procedures* declared in that module. The latter check is also performed for all *static* procedure variables of all other loaded modules. The following code excerpt of procedure *Modules.CheckRef* shows a possible realization of this scheme:

```
PROCEDURE CheckRef(mod: Module; VAR res: INTEGER);
   VAR M: Module; pref, pvadr, r: LONGINT;
BEGIN (*mod # NIL*) M := root;
   WHILE M # NIL DO
      IF (M # mod) & (M.name[0] # 0X) THEN Kernel.Mark(M.ptr) END ; (*mark dynamic refs*)
      M := M.next
   END ;
   Kernel.Check(mod.data, mod.var, mod.code, mod.imp, res); (*check dynamic refs*)
   IF res = 0 THEN M := root;
      WHILE (M # NIL) & (res = 0) DO (*check static refs*)
         IF (M # mod) & (M.name[0] # 0X) THEN
            pref := M.pvar; SYSTEM.GET(pref, pvadr);
            WHILE pvadr # 0 DO (*procedure variables*) SYSTEM.GET(pvadr, r);
               IF (mod.code <= r) & (r < mod.imp) THEN res := 3 END ;
               INC(pref, 4); SYSTEM.GET(pref, pvadr)
            END
         END ;
         M := M.next
      END
   END
END CheckRef;
```

[7] *In general, there can be type references, procedure references and pointer references from (static or dynamic) objects of other modules to (static or dynamic) objects of the module to be unloaded, i.e. a total of 3x4=12 combinations. However, only dynamic type references to (statically declared) types, and static and dynamic procedure references to (statically declared) procedures need to be checked.* ***Static type references*** *from (global variables declared in) other loaded modules to (statically declared) types declared in the module to be unloaded don't need to be checked as they are checked via their import relationship (if clients exist, a module can never be unloaded).* ***Pointer references*** *from (static or dynamic) pointer variables of other modules* ***to dynamic objects*** *of the module to be unloaded don't need to be checked – they are handled by the garbage collector (if a module is unloaded, the heap records that become unreachable as a result of unloading will be automatically collected during the next cycle of the garbage collector). There could, at least in theory, also be* ***pointer references*** *from (static or dynamic) objects of other modules M* ***to static module data*** *of the module M' to be unloaded, which could, for example, have been installed in a base module M via an installation procedure M.Install (p: P) that abuses SYSTEM.ADR and SYSTEM.PUT. Even though it would be trivial to add the corresponding checks to procedures Modules.CheckRef (5 additional lines) and Kernel.Check (2 additional lines), we refrain from checking such references, as they are not in the "spirit" of Oberon. In Oberon, pointer variables are meant to point to objects in the dynamic space (heap), while the use of the pseudo module SYSTEM should be confined to a small number of well-isolated and well-justified cases only, for example device drivers. If a programmer deviates from that rule, it should also be his/her responsibility to guarantee a stable system state.*

[8] *In a variant of Experimental Oberon (source code files with suffix "WithForceUnload"), the command System.Free has an additional /u (unload) option, which forces the physical removal of a module from memory even if references to the module exist in the remaining part of the system; in that case however, type descriptors are preserved in memory (in the form of "pseudo" modules containing just the type descriptors) for as long as references to them exist; in addition, references to the unloaded module are "made safe" by setting procedure variables of other modules referring to it to an "empty" procedure – this effectively "signals" to the end user (via the absence of mouse tracking in an installed viewer handler for instance) that the module's code is "really no longer there". However, with the /f option, the /u option is no longer needed (nor recommended).*

where procedure *Kernel.Check* implements the *scan* phase for *dynamic* references:

```
PROCEDURE Check*(type0, type1, proc0, proc1: LONGINT; VAR res: INTEGER);
  VAR p, r, mark, tag, size, offadr, offset: LONGINT;
BEGIN p := heapOrg; res := 0;
  REPEAT SYSTEM.GET(p+4, mark);
    IF mark < 0 THEN (*free*) SYSTEM.GET(p, size)
    ELSE (*allocated*) SYSTEM.GET(p, tag); SYSTEM.GET(tag, size);
      IF mark > 0 THEN SYSTEM.PUT(p+4, 0); (*unmark*)
        IF (type0 <= tag) & (tag < type1) THEN (*types*) res := 1
        ELSIF res = 0 THEN offadr := tag + 16; SYSTEM.GET(offadr, offset);
          WHILE offset # -1 DO (*skip pointers*) INC(offadr, 4); SYSTEM.GET(offadr, offset) END ;
          INC(offadr, 4); SYSTEM.GET(offadr, offset);
          WHILE offset # -1 DO (*procedure variables*) SYSTEM.GET(p+8+offset, r);
            IF (proc0 <= r) & (r < proc1) THEN res := 2 END ;
            INC(offadr, 4); SYSTEM.GET(offadr, offset)
          END
        END
      END
    END ;
    INC(p, size)
  UNTIL p >= heapLim
END Check;
```

In order to make such a validation pass possible, type descriptors for *dynamic* records and descriptors of *global* module data have been extended with a list of *procedure variable offsets*, adopting an approach employed in an earlier implementation of Original Oberon[9]. These additional offsets are simply appended to the existing fields of a descriptor, i.e. their offsets are greater than those of the fields of the *pointer variables* needed for the garbage collector, in order not to impact the latter's performance. The compiler generating these modified descriptors, the format of the Oberon object file containing them and the module loader transferring them from file into memory have been adjusted accordingly.

In sum, unloading a module affects only *future* references to it, while *past* references from other modules remain unaffected. For example, older versions of a module's code can still be executed if they are referenced by static or dynamic procedure variables in other modules, even if a newer version of the module has been loaded in the meantime[10].

Although the operation of *reference checking*, as outlined above, may appear expensive at first, in practice there is no performance issue[11]. An alternative solution would have been to impose *additional conditions* on module unloading. For example, in Original Oberon 2013 *"a module can be dispensed only if (1) it has no clients, and (2) if it does not declare any record types which are extensions of imported types"* (see chapter 6.4 of the book *Project Oberon*, where however condition (2) is not actually implemented). We refrain from adopting this approach for three reasons: First, it does not take into account references from *procedure variables*. Second, modules that meet such *static* conditions can in fact *never* be unloaded and released from memory (even if no references exist), but can only ever be removed from the *list* of loaded modules (determining whether a module can *actually* be unloaded *still* requires reference checking). And third, it would require making a series of changes to the compiler, the object file format and the module loader, which in sum appeared to be a kludge and would not have been as simple as the chosen solution.

[9] *http://e-collection.library.ethz.ch/eserv/eth:3269/eth-3269-01.pdf (The Implementation of MacOberon, 1990)*
[10] *If an older version of a module's code accesses global variables (of itself or of other modules), it will automatically access the "right" version of such variables – as it should.*
[11] *Reference checking is similar to garbage collection and thus barely noticeable (even on the original Ceres computer, whose clock frequency is about 25 MHz).*

6. System building tools

A minimal version of the Oberon system *building tools* has been added, consisting of the two modules *Linker*[12] and *Builder*. They provide the necessary mechanisms and tools to establish the prerequisites for the regular Oberon startup process[13].

The command *Linker.Link* links a set of Oberon binary files together and generates an Oberon *boot file* (a pre-linked binary file containing a set of compiled Oberon modules) from them. The linker is almost identical to the regular Oberon loader (*Modules.Load),* except that it writes the result to a file on disk instead of loading (and linking) the specified modules in main memory.

The command *Builder.CreateBootTrack*[14] loads a valid *regular boot file*, as generated by the command *Linker.Link*, onto the boot area (sectors 2-63 in Oberon 2013) of the local disk, which is one of the two valid Oberon boot sources (the other one being the serial line). From there, the Oberon *boot loader* will transfer it byte for byte into main memory during stage 1 of the regular boot process, before transferring control to its top module.

In sum, to generate a new *regular* Oberon *boot file* and load it onto the local disk's boot area, one can execute the following commands (*on* the system which is to be modified):

```
ORP.Compile Kernel.Mod FileDir.Mod Files.Mod Modules.Mod ~   ... compile the modules of the inner core
Linker.Link Modules ~                                       ... create a regular boot file (Modules.bin)
Builder.CreateBootTrack Modules.bin ~                       ... load boot file onto the disk's boot area
```

Note that the last command overwrites the boot area of the *running* system. A backup of the local disk is therefore recommended before experimenting with new *inner cores*[15].

The modules *Linker* and *Builder* are mainly used to prepare and load new *inner cores* and thus constitute a key tool for the *system builder*. A corollary is that it is not recommended to distribute these two modules to *pure* production systems used by *regular* end users.

When *adding* new modules to a *boot file*, the need to call their module initialization bodies during stage 1 of the boot process may arise, i.e. when the *boot file* is loaded into main memory by the *boot loader* during system restart or when the reset button is pressed.

We recall that the Oberon *boot loader* merely *transfers* the *boot file* byte for byte from a valid boot source into main memory and then transfers control to its *top* module. But it does *not* call the initialization bodies of the *other* modules that are also transferred into main memory as part of the *boot file* (this is, in fact, why the *inner core* modules *Kernel*, *FileDir* and *Files* don't *have* initialization bodies – they wouldn't be executed anyway).

The easiest way to add a new module *with* a module initialization body to an Oberon *boot file* is to move its initialization code to an exported procedure *Init* and call it from the *top* module of the modules contained in the *boot file*. This is the approach chosen in Original Oberon, which uses module *Modules* as the top module of the *inner core*.

---

[12] *The linker has been included from a different source (https://github.com/charlesap/io). It was slightly adapted and extended for Experimental Oberon's object file format.*
[13] *Currently not implemented is a tool to prepare a disk initially – which consists of a single 'Kernel.PutSector' statement that initializes the root page of the file directory (sector 1).*
[14] *Historically, the boot file was located on a separate "track" on a spinning hard disk (or floppy disk) on a Ceres computer. We retain the name for nostalgic reasons only.*
[15] *If you work with an Oberon emulator (e.g., https://github.com/pdewacht/oberon-risc-emu) on a host system, you can simply make a copy of the directory containing the disk image.*

An alternative solution is to extract the starting addresses of the module initialization body of the just loaded modules from their module descriptors now present in main memory and simply call them, as shown in procedure *InitMod* below (see chapter 6 of the book *Project Oberon* for a detailed description of the format of a *module descriptor* in memory; here it suffices to know that it contains a pointer to a list of "entries" for exported entities, the first one of which points to the initialization code of the module itself).

```
PROCEDURE InitMod(name: ARRAY OF CHAR);
   VAR mod: Modules.Module; P: Modules.Command; w: INTEGER;
BEGIN mod := Modules.root;
   WHILE (mod # NIL) & (name # mod.name) DO mod := mod.next END ;
   IF mod # NIL THEN SYSTEM.GET(mod.ent, w);
      P := SYSTEM.VAL(Modules.Command, mod.code + w); P
   END
END InitMod;
```

In the following example, module *Oberon* is chosen as the new top module of the *inner core*, while module *System* is configured to be the new top module of the *outer core*.

Stage 1: Modules loaded by the Oberon boot loader (BootLoad.Mod) & initialized by their top module Oberon

```
MODULE Modules; ..                    ... old top module of the inner core, now just a regular module
   IMPORT SYSTEM, Files;
   ...
BEGIN ...                             ... no longer loads module Oberon (as in Original Oberon)
END Modules.

MODULE Oberon; ...                    ... new top module of the inner core, now part of the boot file

   PROCEDURE InitMod (name: ...);
   BEGIN ...                          ... see above (calls the initialization body of the specified module)
   END InitMod;

BEGIN                                 ... boot loader will branch to here after transferring the boot file
   InitMod(„Modules“);                ... must be called first (establishes a working file system)
   InitMod(„Input“);
   InitMod(„Display“);
   InitMod(„Viewers“);
   InitMod(„Fonts“);
   InitMod(„Texts“);
   Modules.Load(„System“, Mod); ...   ... load the outer core using the regular Oberon loader
   Loop                               ... transfer control to the Oberon central loop
END Oberon.
```

Stage 2: Modules loaded and initialized by the regular Oberon loader (Modules.Load)

```
MODULE System;
   IMPORT ..., MenuViewers, TextFrames;
   ...
END System.
```

We note that this module configuration reduces the number of stages in the Oberon *boot process* from 3 to 2, thereby simplifying it (at the expense of extending the *inner core*). If one prefers to keep the *inner core* as small as possible, one might choose to extend the *outer core* instead with module *System* and its imports, with the (small) disadvantage that the basic viewer complex would be "locked" in the *outer core*. However, an Oberon system without a viewer manager hardly makes sense, even in closed server environments.