The Experimental Oberon System

Andreas Pirklbauer

1.9.2019

Experimental Oberon[1] is a revision of the FPGA Oberon[2] operating system and its compiler. It contains a number of enhancements, including smooth scrolling of displayed texts with variable line spaces, multiple logical displays, safe module unloading, system building and maintenance tools, and a *Revised Oberon-2* programming language (implementing as a strict superset of the Oberon-07 language) with various features, including type-bound procedures, a dynamic heap allocation procedure for fixed-length and open arrays, a numeric case statement, exporting and importing of string constants and no access to intermediate objects within nested scopes. Some of these enhancements and features are of a purely experimental nature, while others serve the explicit purpose of exploring potential future extensions.

1. Smooth scrolling of displayed texts with variable line spaces

Experimental Oberon enables completely smooth, fractional line scrolling of displayed texts with variable lines spaces and dragging of entire viewers with continuous content refresh. Both *far* (positional) scrolling and *near* (pixel-based) scrolling are realized. The system automatically switches back and forth between the two scrolling modes based on the horizontal position of the mouse pointer. For 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 the Oberon system and its user interface is *considerably* reduced.

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

The Oberon system 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 (or *displays* for short) on the fly and to seamlessly switch between them. Thus, the extended conceptual hierarchy of the display system consists of the triplet *(display, track, viewer)* and consequently the underlying base module *Viewers* exports procedures to add and remove *displays*, open and close *tracks* within existing displays and open and close individual *viewers* within tracks. There are no restrictions on the number of displays, tracks or viewers that can be created. Focus viewers and text selections are separately defined for each display, but there is only one system-wide log.

[1] *http://www.github.com/andreaspirklbauer/Oberon-experimental*
[2] *http://www.inf.ethz.ch/personal/wirth/ProjectOberon/index.html (FPGA Oberon, 2013 Edition); see also http://www.projectoberon.com*

This scheme naturally maps to systems with multiple *physical* monitors attached to the same computer. 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 name* opens a new logical display with the specified name, *System.CloseDisplay id* closes an existing one. *System.ShowDisplays* lists all open displays, *System.ThisDisplay* shows the display *id* and *name* of the current display, *System.SetDisplay id* switches to a new display, *System.SetDisplayName id name* assigns a new name to an existing display, and *System.PrevDisplay* and *System.NextDisplay* "rotate" through the open displays.

The additional commands *System.Expand, System.Spread* and *System.Clone* are displayed in the title bar of every menu viewer. *System.Expand* expands the viewer *as much as possible* by reducing *all* other viewers in its track to their minimum heights, leaving just their title bars visible. The user can switch back to any of the minimized viewers by clicking on *System.Expand* again in any of these title bars. *System.Spread* evenly redistributes all viewers vertically. This may be useful after having invoked *System.Expand*. *System.Clone* opens a new logical display on the fly *and* displays a copy of the initiating viewer *there*. The user can then toggle between the two copies of the viewer (i.e. switch logical *displays*) with a single mouse click[3].

3. Safe module unloading

In the Oberon system, there exist three possible types of references to a loaded module M:

1. *Client references* exist when other loaded modules import module M. Client modules may refer by name to constants, types, variables or procedures declared in module M.

2. *Type references* exist when *type tags* (addresses of type descriptors) in dynamic objects reachable by other loaded modules refer to descriptors of types declared in module M.

3. *Procedure variable references* exist when procedure variables in static or dynamic objects reachable by other loaded modules refer to procedures declared in module M.

In most Oberon implementations, only *client* references are checked prior to module unloading. *Type* and *procedure variable* references are usually not checked, although various approaches are typically employed to address the case where such references exist. As a result, module unloading has traditionally left an Oberon system in an *unsafe* state, which will become *unstable* the moment another module loaded later *overwrites* a previously released module block and other modules still contain *dangling* references to its *type descriptors* or *procedures*. In addition, unloading of module *groups* with references only among themselves is usually not supported.

In order to make module unloading *safe*, Experimental Oberon checks *all* possible types of references as follows prior to unloading a module or module group (Figure 1):

- If clients exist among other loaded modules, a module or module group is never unloaded.
- If no client, type or procedure variable references to a module or module group exist in the remaining modules or data structures, it is unloaded and its associated memory is released.
- If no clients, but type or procedure variable references exist, the module *unload* command takes no action by default and merely displays the names of the modules and the types of references that caused the removal to fail. If, however, the *force* option /f is specified, the

[3] *By comparison, the Original Oberon commands System.Copy and System.Grow create a copy of the original viewer in the s a m e logical display area – System.Copy opens another viewer in the same track of the display, while System.Grow extends the viewer's copy over the entire column or display, lifting the viewer to an "overlay" in the third dimension.*

modules are initially removed only from the *list* of loaded modules, without releasing their associated memory. Such “hidden” modules will later be physically removed from *memory* using the command *Modules.Collect*[4], as soon as there are no more references to them.

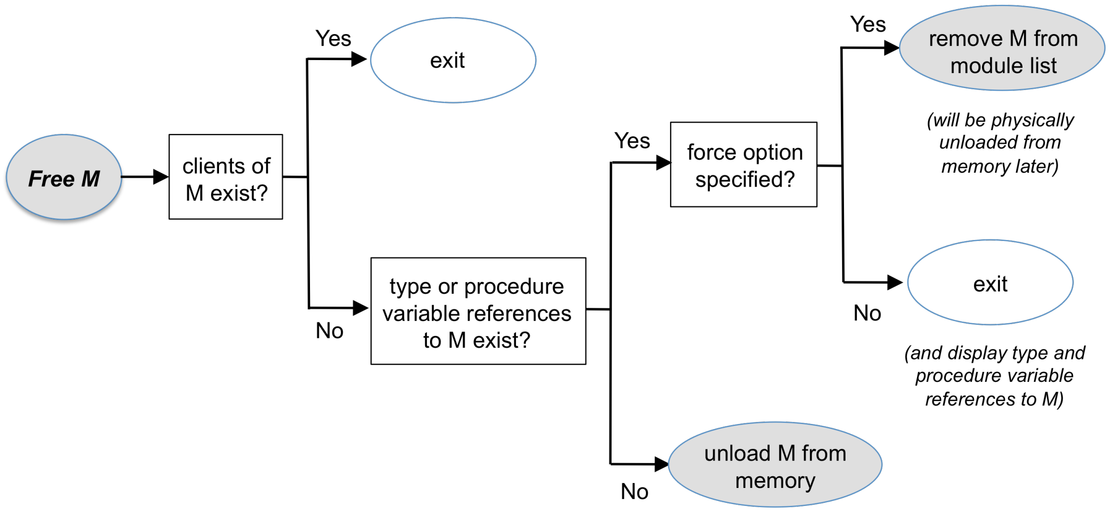
Free M
clients of M exist?
Yes
exit
No
type or procedure variable references to M exist?
Yes
force option specified?
Yes
remove M from module list
(will be physically unloaded from memory later)
No
exit
(and display type and procedure variable references to M)
No
unload M from memory

Figure 1: Safe module unloading in Experimental Oberon

Removing a module only from the module *list* amounts to *renaming* it, with the implication that a newer version of the same module (with the same name) can be reloaded again, without having to unload (from memory) earlier versions that are still referenced. By contrast, unloading a module from *memory* frees up the memory area previously occupied by the module block. In Experimental Oberon, this is only possible when no references to the module exist.

To automate the process of unloading no longer referenced *hidden* module data from memory, the command *Modules.Collect* is included in the background task handling garbage collection. It checks all possible combinations of *k* modules chosen from *n* hidden modules for references to them, and removes those module subgroups from memory that are no longer referenced.

In sum, module unloading does not affect any past or future references, as module data is kept in memory for exactly as long as necessary and removed from it as soon as possible.

If a module *group* is to be unloaded and there exist references *only* within this group, it is unloaded *as a whole*. This can be used to remove module groups with *cyclic* references[5]. It is also possible to release *entire subsystems* of modules. The command *System.FreeImports* attempts to unload the specified modules and all their direct and indirect *imports*. It may be used for conventional module stacks with a single top module, for example a compiler. By contrast, the command *System.FreeClients* unloads the specified modules and all their direct and indirect *client*s. This variant may be used for module stacks written in the object-oriented programming style with their typically inverted module hierarchy, for example a graphics editor.

The mechanism of *safe module unloading* in the Experimental Oberon system is described in more detail in a separate document[6].

[4] *The tool command System.Collect also invokes Modules.Collect*

[5] *In Oberon, cyclic references can be created by pointers or procedure variables, whereas cyclic module imports are normally not allowed. However, through a tricky sequence of compilation and editing steps, it is actually possible to c o n s t r u c t cyclic module imports, which cannot be detected by the compiler. But even though they can be created, such modules are not allowed to be loaded, as the module loader of Experimental Oberon – adopting the approach chosen in Original Oberon and FPGA Oberon – would enter an endless recursion, eventually leading to a stack overflow or out-of-memory condition – a totally acceptable solution for such an artificially constructed case. But even if modules with cyclic module imports w e r e allowed to be loaded, Experimental Oberon would handle them correctly upon unloading, i.e. if no external clients or references exist, such a module group would simply be unloaded as a whole.*

[6] *http://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/Safe-module-unloading-in-Oberon.pdf*

4. Oberon system building and maintenance tools

A minimal version of the Oberon system *building tools*, as outlined in chapter 14 of the book *Project Oberon 2013 Edition*, has been added. They provide the necessary tools to establish the prerequisites for the *regular* Oberon startup process either on an existing *local* system or on a bare metal *target* system connected to a *host* via a data link (e.g., an RS-232 serial line).

The boot linker (procedure *Boot.Link*) links a set of object files together and generates a valid boot file from them. It can be used to either generate the *regular* boot file to be loaded onto the *boot area* of a disk or the *build-up* boot file sent over a data link to a target system. The name of the top module is supplied as a parameter. For the *regular* boot file, this is typically module *Modules*, the top module of the *inner core* of the Oberon system. For the *build-up* boot file, it is usually module *Oberon0*, a simple command interpreter mainly used for system building and maintenance purposes.

The command *Boot.Load* loads a valid *boot file*, as generated by the command *Boot.Link*, onto the boot area of the local disk, one of the two valid *boot sources* (the other being the data link). This command can be used if the user already has a running Oberon system. It is executed *on* the system to be modified and overwrites the boot area of the *running* system.

The tools to build an entirely new Oberon system on a bare metal *target* system connected to a *host* system via a communication link are provided by the module pair *ORC* (for Oberon to RISC Connection) running on the host system and *Oberon0* running on the target system.

The command *ORC.Load Oberon0.bin* loads the Oberon-0 command interpreter over the data link to the target system and starts it there. The commands *ORC.Send* and *ORC.Receive* transfer files between the host and the target system. The command *ORC.SR* ("send, then receive sequence of items") initiates a command on the target system and receives the command's response (if any).

One can also include an *entire* Oberon system in a single *boot file*. Sending a pre-linked binary file containing the entire Oberon system over a serial link to a target system is similar to booting a commercial operating system in a *Plug & Play* fashion over the network or from a USB stick.

There is a variety of other Oberon-0 commands that can be initiated from the host system once the Oberon-0 command interpreter is running on the target system, for example commands for system inspection, loading and unloading of modules or the (remote) execution of commands.

Finally, there are tools to modify the boot loader itself (module *BootLoad*), a small *standalone* program permanently resident in the computer's read-only store (ROM or PROM) – although there generally is no need to do so. Such standalone programs can be created with the regular Oberon compiler. Once compiled, the tool command *Boot.WriteFile* can be used to extract the code section from the boot loader's object file and to convert it to a PROM file compatible with the specific hardware used[7]. Transferring the boot loader to the permanent read-only store of the target hardware typically requires the use of proprietary (or third-party) tools.

The system building tools for FPGA Oberon and Experimental Oberon, as outlined above, are described in more detail in a separate document[8].

[7] *A Xilinx field-programmable gate array (FPGA) contained on the development board Spartan in the case of FPGA Oberon*
[8] *https://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/The-Oberon-system-building-tools.pdf*

6. Revised Oberon-2 programming language

Revised Oberon-2 is a revision of the programming language Oberon-2[9]. The main difference to the original is that it implements a strict superset of *Revised Oberon (Oberon-07)* as defined in 2007/2016[10] rather than being based on the original language *Oberon* as defined in 1988/1990[11].

$$\begin{tikzcd}
\text{Original Oberon (1988/1990)} \arrow[r] \arrow[d] & \text{Original Oberon-2 (1991/1993)} \\
\text{Revised Oberon (2007/2016)} \arrow[r] & \textbf{Revised Oberon-2 (2019/2020)}
\end{tikzcd}$$

Its principal features include type-bound procedures, a dynamic heap allocation procedure for fixed-length and open arrays, a numeric case statement, exporting and importing of string constants, and no access to intermediate objects within nested scopes.

The *Revised Oberon-2* language is described in more detail in a separate document[12].

[9] *Mössenböck H., Wirth N.: The Programming Language Oberon-2. Structured Programming, 12(4):179-195, 1991*
[10] *http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf (Revision 3.5.2016)*
[11] *https://inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf (Revision 1.10.1990)*
[12] *https://github.com/andreaspirklbauer/Oberon-experimental/blob/master/Documentation/The-Revised-Oberon2-Programming-Language.pdf*