A Case Study in Concurrent Programming with Active Objects

Ulrike Glavitsch and Thomas M. Frey

Computer Systems Institute, ETH Zurich,
8092 Zurich, Switzerland
ulrike.glavitsch@inf.ethz.ch, thomas.frey@alumni.ethz.ch

Abstract. The recent product development of processors shows that multi-core computer architectures are rapidly becoming reality. Therefore, in order to use the available processing power, operating systems and programming languages supporting the development of multi-threaded software will be needed. In this paper, we present a small case study that shows how elegant and safe concurrent programming can be if a powerful programming language and thread-safe libraries are used. The case study is a simple search tool written in Active Oberon. The application uses a thread-safe GUI framework that relieves the programmer from synchronizing requests.

1 Introduction

Present and future multi-core computer architectures require multi-processor operating systems and support for multi-threading on the level of the programming language and the environment. Current operating systems are capable to schedule processing tasks to multiple processors. The complexity is hidden, thus, giving the IT user benefits in terms of efficiency and response times. The structuring of a programming task into threads and their synchronization is one of the challenges of current programming and can hardly be automated. In standard programming languages and environments (Java, C#/.NET), multi-threading is supported by a number of specialized library or language calls. However, their use is cumbersome and very often, the programmer has to understand the implementation to make correct use of them. In addition, standard libraries and frameworks for graphical user interfaces (GUI) are seldom thread-safe (e.g. Java Swing, .NET WinForms). This means that it is the programmer's responsibility to synchronize threads that operate on the same component. What is desirable is a programming environment in that developing complex multi-threaded applications becomes more focused on the actual problem and releases the programmer from burdens that can be performed effectively by the environment.

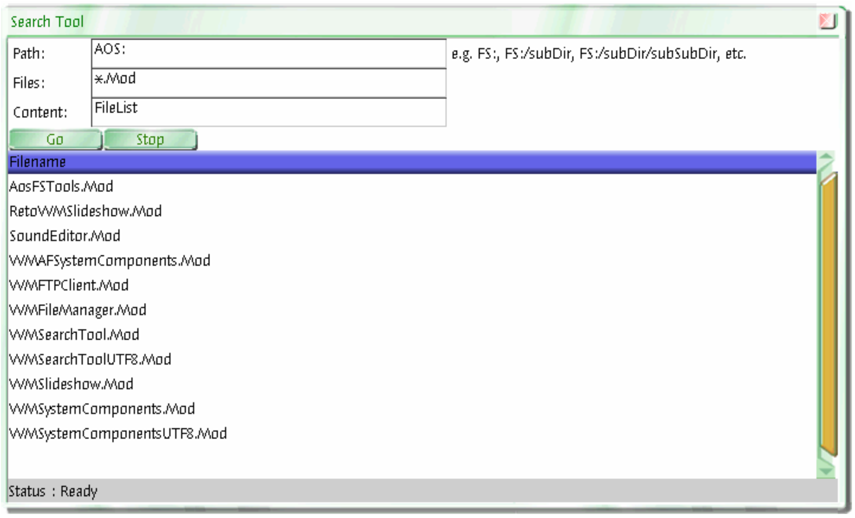
Active Oberon is a type-safe, modular object-oriented programming language that contains dedicated language constructs for threads and their synchronization [1]. Threads are declared as activities encapsulated in objects. Such an active object contains variables and methods like a regular object but its body is executed as a separate thread. In addition, there exist language constructs to declare critical sections with respect to an object scope and a powerful wait statement that allows waiting for a conditional expression to become true instead of waiting for a primitive signal like it is commonly used in other languages and threading libraries. With these, it is possible that an object or a module performs like a monitor [1][2]. It has been shown that these multi-threading specific language constructs can be adopted by other programming languages [3].

Bluebottle is an operating system consisting of a lean multiprocessor kernel and a thread-safe multimedia and GUI framework [1][4]. It is fully programmed in Active Oberon. The particular thread-safety mechanism of Bluebottle's GUI framework makes developing applications easier. Programmers do not need to explicitly synchronize requests to the same GUI component. This built-in synchronization strategy uses a message queue for asynchronous events, a thread for processing the messages and a lock for ensuring a consistent view on inter-component relations [5].

This paper presents the design and implementation of a sample application programmed in Active Oberon under Bluebottle. This application, a search tool, is a typical concurrent GUI application. We demonstrate that the combination of the language constructs of Active Oberon and the synchronization strategy of Bluebottle's GUI framework is perfectly well suited for this type of problem. While the implementation of this application in other environments is typically complex and cumbersome, the program code in Active Oberon becomes clear and concise, and thus, less error-prone.

2 Sample Application

Our case study is a GUI application that allows the user to search for files that contain a given character string. The GUI of the search tool is shown in Fig. 1. The input


Search Tool

Path: AOS: e.g. FS:, FS:/subDir, FS:/subDir/subSubDir, etc.

Files: *.Mod

Content: FileList

```
AosFSTools.Mod
RetoWMSlideshow.Mod
SoundEditor.Mod
WMAFSystemComponents.Mod
WMFTPClient.Mod
WMFileManager.Mod
WMSearchTool.Mod
WMSearchToolUTF8.Mod
WMSlideshow.Mod
WMSystemComponents.Mod
WMSystemComponentsUTF8.Mod
```

Status : Ready

Fig. 1. Graphical user interface of search tool. It shows the results of a search request given the directory path "AOS:", the file mask "*.Mod" and the search string "FileList".

parameters of a search request are a directory path, a file mask (e.g. *.Mod), and a search string. The search is started by pressing the start button labeled with "Go" and interrupted by clicking on the button "Stop". The search results are displayed as soon as they are found. A status line at the bottom of the window indicates the status of an ongoing request. It can either show "Processing" or "Ready". As searching through files is time-consuming it is possible to open some of the found files while the search is still ongoing.

A user may open multiple instances of the search tool on his desktop and start several search requests in parallel.

3 Design

The graphical user interface of the search tool is built using Bluebottle's GUI components. Bluebottle provides a number of standard GUI components to build the most common user interfaces. The search tool uses two threads for searching through the files and for displaying the results. The two threads communicate by a buffer and, thus, represent a classical producer-consumer scheme. The current search results are written to the GUI using a model-view-controller (MVC) pattern. The consumer process updates the model of the GUI component. Changes to the model implicitly lead to an update of the views. Fig. 2 shows the threads (active objects) and the main regular objects involved in this application. The arrows between objects represent method calls.

The singleton object *WindowManager* receives asynchronous mouse and keyboard events and forwards them to the corresponding window [4]. In our case, starting and stopping a search request as well as opening a file for inspection are triggered by mouse clicks.

The main window of the search tool is represented by an object of base type *FormWindow*. It contains the GUI components ordered in a hierarchical structure that control the appearance of the application. For instance, the top element of the component hierarchy is a component of type *Panel* that among others contains a *StringGrid* component to display the search results on the GUI.

The active object *Searcher* processes a search request. It traverses all files that match the given mask and checks if the provided string is contained. The *Searcher* object waits for a new request after finishing a search task. The results are written to a buffer of type *ListBuffer*. The active object *Dispatcher* reads the elements of the buffer chunk by chunk as soon as they are available and updates the object *StringGridModel*. The object *StringGridModel* represents the model in the MVC pattern whereas the view is a GUI component of type *StringGrid* that is part of the component hierarchy contained in object *FormWindow*. The model *StringGridModel* manages a dynamic two-dimensional array of strings that are linked with some more context data. Every change to the object *StringGridModel* automatically leads to an update of its view in the *FormWindow* object.

Status messages are displayed in the GUI component by means of delegate procedures that are registered with the *Dispatcher* object.

Requests to the search tool window are serialized. This means that mouse events from the *WindowManager* object, requests to update the view and status messages are implicitly synchronized. This is done by a sequencing mechanism implemented by a sequencer object that is stored with each *FormWindow* object. The sequencer object contains a thread, a message queue and a lock that protects the hierarchy of GUI components [5]. Each request to the *FormWindow* object that is not called by the sequencer thread is put in the message queue that is processed by the sequencer thread. The object *FormWindow* corresponds to the Active Object pattern that decouples method execution from method invocation in order to allow synchronized access to an object that resides in its own thread of control [6].

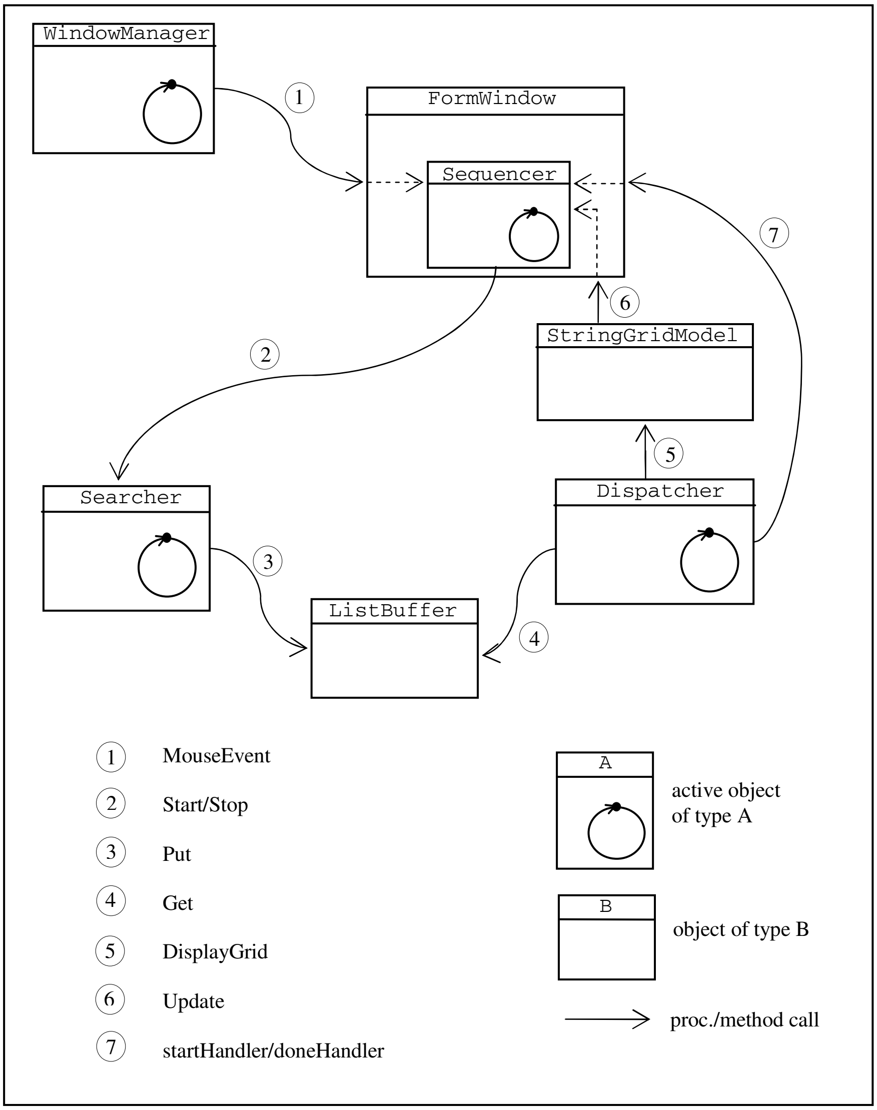


Fig. 2. Call graph of search tool

4 Implementation

The following section presents the implementation of the objects *Searcher*, *Dispatcher* and *Listbuffer*. It describes the mechanism used to update the *StringGridModel* object and the status display. Finally, we explain the wiring of the objects.

Recall that the object *Searcher* administers the search actions. Results are written to the object *ListBuffer* and are forwarded by the object *Dispatcher* to the model *StringGridModel*.

4.1 Searcher

The active object *Searcher* executes an infinite loop that waits for a new search request, then copies the input parameters from the new search request to those of the current request, resets the buffer and starts the search. It notifies the end of the search request to the buffer and waits for the next search request.

Active objects are declared as regular objects but their body is annotated by the keyword *ACTIVE* to denote that the object body is executed as a separate thread. The active object *Searcher* contains some state variables, the parameters of the ongoing and the new search request and a reference to the *Listbuffer* object. The parameters of a search request are packed in a record type *SearchPar* for ease of use. The *Searcher* object acts as monitor that requests mutual exclusion for its methods [2]. Mutual exclusion for a method is denoted by the keyword *EXCLUSIVE* after the first *BEGIN*. A new search request is started by calling the method *Start* and is interrupted by calling the method *Stop*. The relevant code fragments of the Searcher object are given in the following. The language constructs that support multi-threading are highlighted.

```
TYPE
  SearchPar = RECORD
    path, fmask, content : ARRAY 1024 OF CHAR
  END;

  Searcher = OBJECT
  VAR
    newlyStarted, stopped : BOOLEAN;
    currentPar, newPar : SearchPar;
    lb : ListBuffer;

  PROCEDURE &Init(lb : ListBuffer);  (* constructor *)
  BEGIN
    newlyStarted := FALSE;
    stopped := FALSE;
    SELF.lb := lb
  END Init;

  PROCEDURE Start(VAR searchPar : SearchPar);
  BEGIN {EXCLUSIVE}
    newPar := searchPar;
    newlyStarted := TRUE
  END Start;
```

```
    PROCEDURE AwaitNewStart;
    BEGIN {EXCLUSIVE}
      AWAIT(newlyStarted);
      newlyStarted := FALSE;
      stopped := FALSE
    END AwaitNewStart;

    PROCEDURE CopySearchParams;
    BEGIN {EXCLUSIVE}
      currentPar := newPar;
    END CopySearchParams;

    PROCEDURE Stop;
    BEGIN {EXCLUSIVE}
      stopped := TRUE
    END Stop;

    PROCEDURE HasStopped() : BOOLEAN;
    BEGIN {EXCLUSIVE}
      RETURN stopped
    END HasStopped;

    PROCEDURE SearchPath;
    VAR mask, name : ARRAY 1024 OF CHAR;
      e : AosFS.Enumerator;
      d : DirEntry;
    BEGIN
      …
      NEW(e);
      e.Open(mask, {});
      WHILE e.HasMoreEntries() DO
        IF HasStopped() THEN RETURN END;
        IF e.GetEntry(name, …) THEN
          IF ContainsStr(name, currentPar.content) THEN
            NEW(d);
            …
            lb.Put(d);
          END
        END
      END
    END SearchPath;


  BEGIN {ACTIVE} (* body *)
    LOOP
      AwaitNewStart;
      CopySearchParams;
      lb.Reset;
      SearchPath;
      lb.Finished
    END
  END Searcher;
```

The *AWAIT* statement as in procedure *AwaitNewStart* is noteworthy. If the condition in the argument of *AWAIT* returns false, the current process is suspended and put in a list of waiting processes. Additionally to the process, a helper function and the base pointer of the current stack frame are stored. The helper function is generated by the compiler and is used to evaluate the condition of the *AWAIT* statement in a given stack frame. When a process leaves the end of a critical section, the runtime system traverses the list of waiting processes and for each process evaluates the condition using the helper function. If a condition of a waiting process evaluates to true, the lock of the process that leaves the critical section is atomically transferred to the waiting process which is then scheduled [1]. In C#/.NET and Java, suspending a process and waking up one or all waiting processes are realized by special library calls and built-in procedures, respectively. In both C# and Java, the programmer has to place the statements for waking up waiting processes at each location in the code where a condition for any of the waiting processes may become true. This is a burden for the programmer and, in fact, these statements are easily forgotten while developing concurrent C# or Java programs. In addition, if there are processes waiting on different conditions and only one of the conditions becomes true there is no other way than to wake up all waiting processes and to suspend those whose condition is not satisfied yet. Besides that the programs thereby become less readable and less compact this may result in a number of unnecessary context switches that reduce the efficiency of the system. Active Oberon wakes up exactly one of those waiting processes whose condition has become true. Thus, unnecessary context switches are avoided. The cost is that the conditions of waiting processes are evaluated every time a process leaves a critical section. Since these evaluations can be performed without a context switch, this overhead is comparatively small [1].

The procedure *SearchPath* performs the actual search over all files that match the given directory path and the file mask. Before inspecting the next file, *SearchPath* checks whether the flag *stopped* is set and returns if this is the case. The algorithm for finding the occurrence of a given string in a file is the Boyer-Moore string search algorithm.

4.2 Dispatcher

The *Dispatcher* thread waits for a new search request and then continuously reads the search results from the buffer and updates the model of the GUI component that displays the results. It displays status messages to the GUI of the search tool denoting that a search is ongoing and when it has finished. The search results are read from the buffer in chunks to avoid too frequent model updates in the case of very frequently occurring search strings. The delegate mechanism of Active Oberon is used for both the model and the status updates. Delegates are declared similar to procedure types [2]. Formally, a delegate variable is a pair of references that point to an object and to a type-bound procedure.

A new data type *RetrievedList* to contain chunks of the buffer is defined. A buffer element is of type *DirEntry* (see Sec. 4.3). It is a record structure that contains the directory information of a file, e.g. the file name, its size, creation date, etc.. Fragments of the program code of *Dispatcher* are shown below. Dedicated language constructs are again highlighted.

```
TYPE
  RetrievedList = RECORD
    data : ARRAY RListSize OF DirEntry;
    noEl : INTEGER
  END;

  TYPE
    GridDisplayHandler = PROCEDURE {DELEGATE} (VAR rl :
  RetrievedList);

    SearchStatusHandler = PROCEDURE {DELEGATE} ();

    Dispatcher = OBJECT
    VAR
      newlyStarted, stopped : BOOLEAN;
      rl : RetrievedList;
      display : GridDisplayHandler;
      startHandler, doneHandler : SearchStatusHandler;
      lb : ListBuffer;

      (* constructor *)
      PROCEDURE &Init(lb : ListBuffer;
                      d : GridDisplayHandler;
                      sh, dh : SearchStatusHandler);
      BEGIN
        SELF.lb := lb;
        display := d;
        startHandler := sh;
        doneHandler := dh;
        stopped := FALSE
      END Init;

      …
      (* procedures Start, AwaitNewStart, Stop and
         HasStopped as in Searcher *)
      …

    BEGIN {ACTIVE}
      LOOP
        AwaitNewStart;
        startHandler;
        LOOP
          lb.Get(rl);
          IF rl.noEl = 0 OR HasStopped() THEN EXIT END;
          display(rl);
        END;
        doneHandler;
      END
    END GridDisplayHandler;
```

4.3 ListBuffer

The *ListBuffer* data structure is implemented as a circular buffer. The *Searcher* thread puts the search results into the buffer one by one whereas the *Dispatcher* thread consumes them in chunks.

The variables and signatures of the methods of *ListBuffer* are given below. The *Listbuffer* object contains a variable *chunkSize* that denotes the minimum number of buffer elements returned by procedure *Get* in case of an ongoing search request. This number is computed dynamically. It is initialized to 1 and adapted after each call to *Get*. Procedure *Get* returns only if the number of available elements in the buffer are greater or equal to *chunkSize* or if the search is finished. If the number of available buffer elements is greater than *chunkSize* the variable *chunkSize* is adapted. All methods can only be accessed by one process at a time, i.e. they are declared with the *EXCLUSIVE* keyword. Thus, an instance of *ListBuffer* like the instances of *Searcher* and *Dispatcher* acts as a monitor.

```
TYPE ListBuffer = OBJECT
  VAR data : ARRAY RListSize OF DirEntry;
    in, out, chunkSize : INTEGER;
    finished : BOOLEAN;

  PROCEDURE &Reset; (* constructor *)

  PROCEDURE Put(d : DirEntry); (* produce *)

  PROCEDURE Get(VAR rl : RetrievedList); (* consume *)

  PROCEDURE Finished(); (* signal end of searching *)

END ListBuffer;
```

4.4 StringGridModel Update

The model of the GUI component that displays the search results is updated by the delegate *DisplayGrid* that is a method of the type *FileList*. The *FileList* declaration contains the GUI component, its view, as a variable and provides further methods like opening a file in the GUI component.

Excerpts of the program code of *DisplayGrid* and how it is embedded in the declaration of *FileList* is shown below. The variable *grid* denotes the GUI component that shows the search results. It has a reference to the underlying model and provides a locking mechanism such that changes to the model are synchronized. The methods to lock the model are *Acquire* and *Release*. They are highlighted in the code fragment below. The method *Release* implicitly performs an upcall to the observers of the model to update the view.

```
TYPE FileList = OBJECT
  ...
  grid : WMStringGrids;
  ...

  (* delegate *)
  PROCEDURE DisplayGrid(VAR rl : RetrievedList);
  VAR i : LONGINT;
    d : DirEntry;
  BEGIN
    grid.model.Acquire;
    FOR i := 0 TO rl.noEl - 1 DO
      d := rl.data[i];
      …
      (* add the new search result d to the model *)
      …
    END;
    grid.model.Release (* performs an implicit update
                          of the view *)
  END DisplayGrid;

    …

END FileList;
```

4.5 Status Messages and Interconnection of Objects

The delegates for displaying the status messages of the search tool are two very short methods of the object of type *FormWindow*. The constructor of *FormWindow* creates the instances of type *Searcher*, *Dispatcher* and *ListBuffer* and connects them as shown in Fig. 2. The buffer of type *ListBuffer* is registered with the active objects *Searcher* and *Dispatcher* and the delegate procedures are installed with the object *Dispatcher*. The following program code shows fragments of the constructor that creates the objects and their connections. We also present the two delegate procedures that display the status messages. The important pieces of code are marked with highlighted comments.

```
  TYPE
    Window = OBJECT(WMComponents.FormWindow)
    VAR
      (* GUI component that displays status messages *)
      label : WMStandardComponents.Label;
      …
      filelist : WMSystemComponents.FileList;
      lb : ListBuffer;
      s : Searcher;
      d : Displayer;
      …

      PROCEDURE &New();
      BEGIN
        …
        NEW(filelist);    (* object creation and wiring *)
        NEW(lb);
        NEW(s, lb);
        NEW(d, lb, filelist.DisplayGrid,
            SearchStartHandler, SearchDoneHandler);
        …
      END New;
      (* delegate *)
      PROCEDURE SearchStartHandler;
      BEGIN
        label.caption.SetAOC("Status: Processing ...")
      END SearchStartHandler;

      (* delegate *)
      PROCEDURE SearchDoneHandler;
      BEGIN
        label.caption.SetAOC("Status: Ready ...")
      END SearchDoneHandler;
      …
    END Window;
```

It must be noted that this easy way of programming the updating of status messages is due to the synchronization mechanism of Bluebottle's GUI framework. The delegate procedures *SearchStartHandler* and *SearchStopHandler* are called from the *Dispatcher* thread and are executed in the context of the GUI thread, i.e. the sequencer thread of *FormWindow*.

Behind the scenes, the GUI framework checks whether the calling process is the same as the sequencer thread. If this is the case, it puts the method call into the message queue of the sequencer thread. Otherwise, the call is executed immediately within the sequencer thread. Checking this condition costs only a few clock cycles in Bluebottle and, thus, can easily be done within the framework [4].

In C#/.NET, the programmer has to check explicitly whether a context switch to the GUI process is required and is forced to handle the two cases appropriately. Correct handling of these cases requires knowledge of the GUI framework that in our opinion should be hidden from the programmer. Bluebottle's GUI framework is a set of libraries where the programmer does not need to know any implementation details to perform his task.

5 Conclusions

We showed that concurrent programs written in a powerful programming language (Active Oberon) using thread-safe libraries (Bluebottle's GUI framework) are compact, readable and less error-prone. The constructs for multi-threading are integrated in the programming language which facilitates their use. In addition, the dedicated language constructs are lean and very effective such that the program code remains clear and concise. The thread-safety of the GUI framework relieves the program developer from synchronizing requests to the same component. This contributes in a similar way to the readability and compactness of the program code.

References

1. Muller, P. J.: The Active Object System – Design and Multiprocessor Implementation. Ph.D. thesis, Institut für Computersysteme, ETH Zürich (2002)
2. Hoare, C. A. R.: Monitors: An operating systems structuring concept. Comm. ACM (1974) 17(10):549-557

3. Güntensperger, R., Gutknecht, J.: Activities and channels: C# language extensions for concurrency control and remote object communication. IEE Proceedings – Software 150(5):315-322 (2003)
4. Frey, T.: Bluebottle: A Thread-safe Multimedia and GUI Framework for Active Oberon. Ph.D. thesis, Institut für Computersysteme, ETH Zürich (2005)
5. Frey, T. M.: Architectural Aspects of a Thread-safe Graphical Component System Based on Aos. Lecture Notes in Computer Science 2789, Springer (2003)
6. Lavender, R. G., Schmidt, D. C.: Active Object: An Object Behavioral Pattern for Concurrent Programming. In Pattern Languages of Program Design 2 (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Addison Wesley (1996)