DESIGNING GRAPHICAL USER INTERFACES WITH ENTITY-LIFE MODELING

Douglas L. Smith

16 December 1993

ISSE TR 93 110

Abstract

This paper explores using entity-life modeling (ELM) as an effective method for designing concurrent graphical user interfaces in a message processing environment. In ELM, multiple threads of control in the software are modeled on threads of events in the problem environment, and software objects are modeled on objects in the problem. The goal is to identify a minimum number of threads in the problem environment that drive the system while operating on objects. ELM states that threads are primarily those processes that must be rescheduled at particular points in time and those that vie for resources.

The intent of this paper is to show that the ELM approach to problem analysis needs to be modified in order to produce optimal designs in a message passing environment. The paper suggests that some reschedulable processes do not need their own threads of control. Instead, the desired effect can be achieved by timers that insert a message into the message queue of a given process at the appropriate time. The paper also proposes a new category of *background entities* to account for time-consuming calculations that must be carried out concurrently with ordinary message processing.

1 Introduction

Entity Life Modeling [Sanden 94, Sanden 89] was introduced to address the individual shortcomings of two popular paradigms, the process paradigm and the object paradigm. In the process paradigm, the common elements between the problem environment and the software environment are communicating processes where each process represents a single thread of events. The process paradigm emphasizes the sequential aspect of the problem by focusing on the order in which events occur. This method of analysis does not adequately address the order independent aspects of a problem. Such aspects are often found in simple but useful objects that do not change in significant ways over time. Furthermore, when such a design is applied to a multi-thread environment the efficiency of the design is not optimal because of the large number of processes that are usually generated.

In the object paradigm, the common elements between the problem environment and the software environment are objects. Each object in the software environment corresponds to a real object in the problem environment and has a well-defined set of operations. Operations are invoked when an object receives a message from another object. These objects are typically implemented as concurrent, communicating processes in the software system. Each object and its corresponding thread of control is encapsulated in a module. This approach leads to an architecture with modules communicating via messages. The object paradigm deals well with behavior patterns that fit into a single object but may miss patterns that extend beyond objects. Like the process paradigm, the object paradigm also leads to inefficient architectures when applied to a multi-thread environment. The modules produced do not exploit the concurrent aspects of the problem rather they synchronize on messages from other objects to execute operations.

Entity Life Modeling combines the expressive power of these paradigms by giving equal weight to both objects and behavior patterns. ELM first sets out to identify threads of events (behavior patterns) that encompass all the relevant events. For each thread of events exhibiting a sequential behavior pattern in the problem environment, an entity is defined. This entity is then considered a subject candidate if it moves the action forward. ELM then identifies objects in the reality. Each object must be operated on by one or more of the subject candidates. Subjects and objects are then combined into one or more subject-object structures that describe the problem environment.

The goal of ELM is to describe the problem environment in terms of subjects and objects that exploit its concurrent nature optimally.

2 Software Environment

Entity-Life Modeling requires an execution environment where threads of control share an address space and each thread is scheduled for execution dynamically at run-time. Furthermore, a synchronization mechanism is required so that a thread can request exclusive access to a shared resource and remain idle until the resource becomes available. Threads with this capability are said to be queueable. In addition to being queueable, ELM requires threads to be reschedulable also. A rescheduable thread must be capable of suspending execution for a specified interval of time. Finally, ELM requires that the software environment provide a means for expressing objects. That is, the ability to encapsulate persistent data and operations in information hiding modules must be available.

OS/2 [Microsoft Corp. 1989] is an advanced micro-computer operating system which provides a software environment that fulfills the requirements of ELM. OS/2 provides the function DosCreateThread to instantiate a thread of execution and DosExit to terminate a thread of execution. DosSuspendThread gives a thread of execution in a process the ability to suspend any active thread executing in the same process. Likewise, DosResumeThread allows a thread of execution in a process to resume any inactive thread in that process. An entire suite of semaphore functions provide a synchronization mechanism to accommodate threads vying for shared resources. The DosSleep function allows a thread to suspend execution for a specified interval, which makes the OS/2 threads reschedulable. The operating system also supports object oriented languages such as C++ which provide all the necessary tools for encapsulating data and operations in information hiding modules. This operating environment is an adequate platform for implementing real-time systems designed with ELM. The graphical functions native to OS/2, however, are unsophisticated and considered to be inadequate for building state-of-the-art user interfaces.

Presentation Manager is a graphical user interface designed to provide OS/2 with an application user interface capable of producing sophisticated windowing environments [Petzold 1989]. Windowing environments are event

driven by design because they need to respond to user input rapidly. Presentation Manager transforms the operating environment into an event driven, message based architecture. Since all the OS/2 functions are available, this new environment by default fulfills the requirements of ELM. Traditional OS/2 kernel functions used to implement reschedulability and synchronization of shared resources can have an undesirable effect on the performance of systems using Presentation Manager, however.

3 Presentation Manager and the User Entity

Presentation Manager programs usually involve a human user who sustains some kind of ongoing dialog with the system. In such systems, the user is usually the primary candidate for subject entity. This class of entity is more generally referred to as the user entity. This discussion will be limited to systems where the user entity is of primary concern.

A user entity in a Presentation Manager environment must have at least one thread of execution running in one process. A process and a thread can be established by executing the OS/2 function DosExecPgm from the OS/2 environment. This function allocates a child process with a single thread of execution. The user entity thread begins executing as soon as the process is dispatched. Once execution has begun the user entity must create a message queue by calling WinCreateMsgQueue. It must then create one or more windows capable of receiving and processing user input. Presentation Manager directs user input to the window that holds the *input focus*. The window that has focus will be the top-level window and is further identified by some unique change in color to the title bar or border. Window focus is controlled through the WM_SETFOCUS message.

The Presentation Manager uses the message queue to store messages for all windows created in the user thread. The user entity must then loop on the function WinGetMsg. This Presentation Manager function suspends execution of the thread until a message is stored in the queue. After a message has been stored in the message queue the user entity is allowed to resume execution and retrieve the message at the head of the queue. Retrieval of messages occurs without interruption if the message queue is

not empty when the WinGetMsg function is executed. Once the message has been retrieved by the user entity, the function WinDispatchMsg is used to dispatch the retrieved message to the appropriate window procedure for handling. This process of retrieving and dispatching messages continues as long as WinGetMsg completes successfully and returns a non-zero value. When the user entity retrieves the WM_QUIT message WinGetMsg returns zero allowing the thread to exit the loop and terminate.

The user entity may communicate with itself or other entities possessing message queues through the presentation manager functions WinPostMsg or WinSendMsg. WinPostMsg operates asynchronously by placing a message in the message queue associated with a particular window and returns immediately. WinSendMsg operates synchronously by calling the window procedure directly and returning after the window procedure has handled the message. Communication with entities without message queues occurs through traditional means such as global or shared memory.

It should be apparent from the preceding discussion that the user entities primary objective in a graphical user interface is to process input events expediently. Any lengthy delay in the processing of an event will have an undesirable effect on the user. This effect is described in more detail later.

4 ELM Applied to Presentation Manager

As mentioned, the goal of ELM is to identify a minimum set of entities that operate on real-world objects such that all relevant events are taken into account. If ELM is applied to a sequential problem, it will identify the intuitive solution of a single entity operating on one or more real-world objects. The method is most useful when applied to problems involving more than one entity (thread of execution). ELM aids the analyst by classifying reschedulable and queueable entities. Entities in these classes are subject candidates and must have their own thread of control. These classifications must be reexamined for the message passing environment of Presentation Manager.

4.1 Reschedulable entities

ELM defines a reschedulable entity as a thread of events that captures the requirement that certain actions must be taken at a particular time or with a defined interval. Furthermore, it states that each reschedulable entity requires its own task even if there are multiple instances. In traditional environments such as the Ada run-time environment or OS/2, this is a necessary requirement because it allows each instance to be rescheduled independently. This requirement exists because these traditional environments rely on thread suspension mechanisms such as Ada's delay statement or OS/2's sleep primitive. As a result, it is impossible for a task or thread to service more than one reschedulable entity.

A Presentation Manager application can use the OS/2 functions DosCreateThread and DosSleep to implement rescheduable entities as described above. However, this approach does not utilize the message passing architecture of Presentation Manager which provides its own mechanism for rescheduling. The function WinStartTimer accepts as parameters, a timer interval, a window handle, and a timer identification value. The timer interval value indicates the interval of time that should transpire between timer events. The timer identification value and window handle uniquely identifies the timer event. When a timer event occurs, the appropriate window procedure receives a WM_TIMER message which carries the timer identification value as a message parameter. The function WinStopTimer will discontinue a specific timer event allowing for independent rescheduling. It should be clear, that the timer mechanism provided by Presentation Manager provides the capability of rescheduling without thread suspension.

The message passing architecture of Presentation Manager eliminates the requirement that a reschedulable entity must have its own task or thread. The primary reason a reschedulable entity requires its own task is because the reschedulabilty of the entity dominates the processing of the thread. That is, when a thread executes the DosSleep function it remains idle until the interval expires and processing resumes. The timer mechanism in Presentation Manager relieves the thread from the burden of waiting for the time interval to expire by scheduling a unique event. This allows the thread to continue processing messages waiting in the message queue.

Eliminating the requirement of separate threads for reschedulable entities raises several issues. The first issue is one of precision. Often in real-time

systems a reschedulable entity requires precise processing. Consider an application that maintains a graphical stop watch programmed to display elapsed time down to 1/10th of a second. For this application to maintain an accurate display, timer messages must be processed expeditiously. Any significant delay servicing the timer event could result in an inconsistent or inaccurate display. Consequently, the amount of time a message spends in the message queue must be addressed.

The dynamics of the message queue itself must be considered first. All messages in the message queue at any one time are not treated equally. Several specific messages have a unique priority and all remaining messages share a priority. Messages of the same priority are serviced in FIFO order as they appear in the queue. Message sets of differing priority are serviced in order from highest priority down to lowest priority. This is a significant consideration because WM_TIMER messages carry a lower priority than the majority of all the messages. Most notably, their priority is less than the priority of user defined messages. Presentation Manager provides the function WinPeekMsg to examine messages in the message queue. WinPeekMsg operates like WinGetMsg but it returns immediately if the message queue is empty. Also, a NOREMOVE option allows examination without removal of the message from the queue. This method of ensuring that WM_TIMER messages are serviced as they arrive would be burdensome and unstructured. Therefore, analysis must guarantee that reschedulable entities are processed properly.

I propose that ELM could produce more efficient designs for message processing systems by qualifying which reschedulable entities require their own thread of execution and introducing a new class of entities that require a thread of execution. The new entity takes background for processing an event into account. I will discuss each of these in turn.

4.2 Independent reschedulable entities

The traditional ELM definition of a reschedulable entity captures a requirement that particular actions must be taken at a certain time or according to a specific time interval. This definition is too broad to allow the categoric classification of reschedulable entities as entities that require their own thread of execution. From this point forward, I will refer to those reschedulable entities that require their own thread of execution as *independent reschedu-*

lable entities. There are two compelling reasons for categorizing independent reschedulable entities in a message processing system. The first reason is precise calibration. Any behavior pattern exhibiting a timing requirement and whose successful operation requires immediate execution is a independent reschedulable entity. This entity must have its own thread of execution and message queue. Entities of this nature are not common in graphical user interfaces but design methodologies like ELM must account for them.

The second and less compelling reason for defining an independent reschedulable entity is longevity. Often in graphical user interfaces a user entity is more long-lived than any other entity in the system. As a result, timed behavior patterns that do not require expedient execution can be part of the user entity. Analysis, however, must account for timed behavior patterns that outlive all suitable message processing entities in the system. Therefore, the new definition of an independent reschedulable entity must include these behavior patterns.

This new definition of an independent reschedulable entity exposes a fundamental concern in message processing systems. Currently, there is no guarantee that the processing associated with an event (message) will not take an unacceptable amount of time. Lengthy event processing disrupts the handling of messages in the message queue. This is a serious problem for a graphical user interface because they are designed to allow input focus to be switched instantaneously. However, a user entity may only give up focus by processing a message instructing it to do so. Since the user entity is busy processing a message, the focus message sits in the message queue and the user entity is unable to give up input focus. Many of the advantages offered by a preemptive multi-tasking operating system are crippled when a user entity is unable to relinquish focus in a timely manner. Furthermore, the new definition for an independent reschedulable entity relies on timely processing of timer messages in the message queue. For these reasons, ELM needs to identify a new entity type to account for processing time.

To address this issue, I will first qualify what an unacceptable amount of time is. The popular book *Programming the OS/2 Presentation Manager* [Petzold 1989] suggests 1/10th of a second as the longest acceptable time to process a message. For the purposes of this paper, this limit is a suitable guideline. It also assumes that recognizing a event with a potential to exceed the guideline regularly is intuitive. For example, consider an application that searches several files for the occurrence of a string or logical combination of

strings. The ELM designer would probably be tempted to define the files as objects and the search as an operation on those objects. In this design, the user entity would then be chosen as the subject to execute search operations on the file. This design, however, would be plagued with the focus and timer problems mentioned earlier. To correct this design, a separate entity must be in place to carry out the search operations.

4.3 Background entities

The ELM methodology for message passing systems can be corrected by categorizing a new class of entities called background entities. This class will contain those entities that have a lengthy time requirement. Similar to independently reschedulable entities, background entities require their own thread of execution. However, unlike reschedulable entities, background entities are not candidates for subject entities. The primary purpose of this class of entity is to off load work from a message queued subject entity.

This new class of entity takes advantage of the flexibility provided by a message passing system. That is, the message queue absorbs the delay due to processing and allows the subject to proceed. Upon completion of processing the background entity can report the results to the subject by sending it a message. The background entity will be most effective if it is implemented with a message queue which allows subjects to queue up a request by posting a message. To put this in the proper perspective, I will revisit the string search example. With the new approach the ELM designer would recognize the search operation as one with a lengthy time requirement. Consequently, the designer creates a search entity and designates the user entity as the subject that requests the search entity to carry out its operation by sending it a message. The user entity is free to process other messages while it waits for the asynchronous response. It should be apparent that the proper identification of this new class of entity will guarantee that message queue servicing is not significantly delayed by message handling in implementing ELM designs.

4.4 Queueable entities

ELM defines a queueable entity as any entity that competes for a shared resource. Furthermore, it recognizes queueable entities as subject candidates

that require their own thread of execution. This classification holds without deviation in the message passing environment of Presentation Manager as well. There are, however, aspects of queueable entities that should be examined.

Upon examination of the definition of reschedulable entities, it was discovered that a refined classification would produce a better design. This refined classification allowed certain reschedulable entities to be absorbed into other message processing entities such as the user entity. The opposite is true with queueable entities. Absorption into message processing entities is absolutely prohibited because synchronization delays during message processing would cause the same focus and timer message problems mentioned earlier.

The message passing architecture can be used to avoid synchronization delays in subject threads. In the Ada run-time environment, a special purpose guardian task can be used to implement mutual exclusion. Guardian tasks protect one or more critical sections by implementing a rendezvous for each critical section which is implemented in the body of the accept statement. The disadvantage to the Ada guardian task is that competing tasks are suspended and wait in a queue for the desired resource. Similar to the Ada run-time environment, mutual exclusion can be implemented in special purpose threads. These threads must be message processing threads and I will refer to them as *guardian threads*. A guardian thread may protect one or more critical sections by associating a unique message with each critical section. When a message arrives in the guardian threads message queue it retrieves the message and executes the critical section for that message without interruption.

The advantage of guardian threads over guardian tasks is the message queue. Again, the message queue absorbs the delay allowing the subject thread to proceed. Contrast this with guardian tasks where the subject tasks are suspended and queued. Although it is more efficient, the guardian thread has its disadvantages as well. Operation of the guardian task is completely asynchronous. Therefore the subject task must be able to proceed independently of critical section execution. If this is not the case the designer must enforce synchronization by implementing a return message to the subject thread. If the subject thread cannot split its processing in this fashion then traditional means of critical section synchronization must be employed.

5 Conclusion

The primary goal of ELM is to identify a minimum set of subject threads that operate on objects. The subjects are then combined with objects in subject-object structures that best describe the problem environment. This paper shows that the ELM approach to problem analysis can be successfully modified to achieve its goal in the message passing architecture of Presentation Manager. The reclassification of reschedulable entities as independent rescheduable entities eliminates unneeded threads. Likewise, the identification of background entities enables the model to avoid processing delays that locks up the message passing system. As a result, these changes allow ELM to be applied in manner that is both optimal and correct.

Bibliography

Microsoft Corporation, OS/2 Programmer's Reference, Microsoft Press, Redmond, WA, 1989.

Petzold, Charles, Programming the OS/2 Presentation Manager, Microsoft Press, Redmond, WA, 1989.

Sanden, Bo, Software Systems Construction with Examples in Ada, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1994.

Sanden Bo, "An entity-life modeling approach to the design of concurrent software", Commun. ACM, 32:3, March 1989, pp. 330-343.