Computer Science Journal of Moldova, vol.**3**, no.2(8), 1995

Message composition and its application to event-driven system construction

A.Colesnicov

Abstract

Due to the object-oriented technology of event-driven system construction, the message composition may be used. Rules of message composition are alike those of program statement composition. The interpreting message queue is described which produces primitive messages from compound ones. The proposed conception lets to include the information on message dependence to compound messages themselves, which permits to simplify programs.

1 Introduction

When a system is developed using the event-driven object-oriented technology, its behavior is controlled by events. An event may be external, e.g., mouse movement, and internal—execution of a program block or subroutine. Events change the computation state space.

Inside the system, events are reflected by messages. An external event is processed by the corresponding interrupt handler, and one or more messages are generated. These messages are dispatched to instances of the program objects. Objects handle messages and execute the corresponding code; they may produce new internal messages as the result, and so on. In many cases the executing program processes may generate internal messages without any external signals. E.g., the compilation process may produce messages finding an error.

All messages are passed firstly to the system message queue, are kept by it and then are dispatched to program objects. An object may

process a message and destroy it, or it may pass the message to its subordinate objects, or even return the message to the queue. And it may generate several new messages and pass them to the queue. These message flows circulate into the system and control its operations.

The ideal case for such system is "sleep-and-wait", when all possible events are external and independent. The system stays still and waits the user action. When the user presses a key or operates with the mouse, the system handles the corresponding messages and waits again.

This ideal case is not so rare. Meanwhile, there are many programs in which:

1. There are several independent and asynchronous event sources except the user.

2. Events (and representing messages) are not fully independent.

Both problems may be solved with standard message management methods, but the price paid is the complication of program structure.

The message composition was proposed in [4] to resolve problems of message flow synchronization. We are to note here that in that previous article we had followed the terminology of our used software [2], where messages are called 'events'. Afterwards it was found that this approach resolves the above mentioned problems.

The present article contains in Section 2 the detailed description of the message composition rules. The compound message generation and insertion into the queue is described in Section 3, and their interpretive decomposition is described in Section 4.

The message queue is to be able to reply requests for its state. The importance of this property is discussed in Section 5.

An example of message flows in a program and their management using proposed technique are referred in Section 6.

The proposed message queue can use files as message sources. The problem of creating such files if they are not logs from some previous runs is discussed in Section 7.

In literature, the conception of processing compound event scripts can be found in [1], in connection with reversing (undoing) program

actions and testing different event sequences. In [1] each event is represented in the script by three pointers to procedures which perform the action, undo the action, and perform the inverse action. The event script is organized as a tree.

2 The Message Composition

2.1 Primitive Messages

For simplicity, we restrict our discussion by the following five classes of primitive messages:

1. The empty message.
2. External primitive messages, namely:
   (a) the mouse message;
   (b) the keyboard message.
3. Internal primitive messages, namely:
   (a) the command message;
   (b) the broadcast message.

Primitive messages are the only messages that are dispatched by the message queue to program objects and that are processed by these last. You do not need to program your objects to process compound messages.

Primitive messages contain the corresponding information, e.g., a keyboard message contain the key code, the keyboard status, etc. The particular layout is indifferent for our purposes.

2.2 Block Messages

Rules of message composition are exactly the same as rules of program statement composition. Having program statements, we can recursively organize them in blocks, selections (**if**, **case**), and loops (**while**, etc.).

Block messages are composed from one or more messages (primitive or compound) that are dispatched in sequence. In implementation, one possible way is to represent the block message by the pointer to the first submessage in sequence, and to attach to each submessage the pointer to the next message in the sequence, with **nil** attached to the last, i.e., by the simple list.

2.3 Special Types of Block Messages

There are the following special types of block messages:

1. A text message contains a string and acts exactly as if the string was typed at the keyboard.

2. A keystroke message contains a sequence of key codes including not only characters as in the previous case, but keystrokes like Alt-X.

3. A text file message contains a file name and acts as if the file contents was typed at the keyboard.

4. A keystroke file message contains a file name, and the corresponding file contains keystrokes to be played.

5. A message file message contains a file name, and the corresponding file contains messages to be played.

In the case of message file message it is possible to restrict the file contents by the primitive messages only, or to permit compound messages there. See Section 7 below.

2.4 Selection Messages

A selection message contains the pointer to a selector function which returns the discrete (**Boolean**, **integer**, etc.) value, and the pointer to a sequence (list) of *(value, pointer to a message)* pairs. More general, a pair may contain, instead of a single value, a list of values or value intervals, or the special **else** value. As a particular case, an if message

may be defined which contains pointer to a **Boolean** selector function and pointers to two submessages.

2.5 While Messages

A while message contains the pointer to a **Boolean** test function and the pointer to a message.

3 Compound Message Generation

Compound messages are recursive structure. The best language to describe such structures is LISP. If we work in procedural language like Pascal, we are to define several procedures creating compound messages, e.g. (Pascal syntax):

```
function CreateWhileMessage(PTest:  PBooleanFunction;
                            Body:   PMessage):  PMessage;
function CreateIfMessage(PTest:  PBooleanFunction;
                         IfPart, ElsePart:  PMessage):
                         PMessage;
function CreateBlockMessage(Head, Tail:  PMessage):
                         PMessage;
```

Here `PMessage` is 'pointer to message' and `PBooleanFunction` is 'pointer to function returning Boolean value'.

Suppose we want to create a while message whose body is a block message of 3 messages. We write in Turbo Pascal[1]:

```
CreateWhileMessage(Addr(TestFunc1),
                   CreateBlockMessage(Addr(Message1),
                   CreateBlockMessage(Addr(Message2),
                   CreateBlockMessage(Addr(Message3),
                   nil))));
```

[1] *Turbo Pascal* is the registered trademark of Borland International, Inc.

The `Addr` function yields pointer to its argument. Instead of
`Addr(Message1)`, `Addr(Message2)`, and `Addr(Message3)`
we can write another message creating function calls, for compound or primitive messages. We see that the structure is quite LISP-like.

There is one possible problem. The interpretive message dispatching algorithm described in Section 4 below will work right only in the case when the structure of any compound message is a tree, with different[2] primitive messages as leafs. The corresponding checking is simple and may be performed by message generating functions or by message using procedures.

With the restriction described above, we may compare the compound message structure with the textual structure of a program in structural programming paradigm, which is a tree. The case of message composition in an arbitrary graph (a net with nodes and transition conditions) is not so simple to implement and is now under investigation. The obvious parallel to the last case is the program structure with labels and **goto**'s.

See Section 7 below for further discussion on creating compound messages.

4 Compound Message Interpretation

The message queue dispatches messages to instances of program objects. The only dispatched messages are primitive ones. Compound messages are decomposed as follows.

The block compound message is decomposed to its constituent submessages in sequence. It means that if the block message is the one to dispatch, the first of its constituents is extracted, and the non-empty remainder, being the new block message, is included anew into the queue. Next time the next constituent will be extracted, etc. Of course, any constituent may be compound, so the decomposition is recursive.

[2]You may use several copies of a message; unpermitted are pointers to the same place.

The selection compound message is decomposed by calling its selector function, and then extracting the constituent submessage selected by the corresponding value. The original selection message disappears. This selected constituent may be compound as well.

The while compound message is interpreted by calling its test function. If this call returns **true**, the constituent body submessage is selected to process, and the original while message remains into the queue. If the test function call returns **false**, the while message is deleted from the queue without any processing. The constituent submessage may be compound.

When the compound message is interpreted, it causes the dispatching of its terminal constituent primitive messages, in the sequence defined by the compound message structure and selector and test function values. It would be wrong to dispatch them as the continuous flow, because each primitive message send to program objects may cause the new message generation, and in many cases those new messages are to be processed before the next constituent of the original compound message.

One possible solution is to attach to each message the absolute (initial) priority, the priority increment and the current priority. When the message is included into the queue, its current priority is set equal to its absolute priority. The queue selects to process messages with the maximal current priority. If the message is compound and after it remains non-processed message (block, or while) then the priority of this remaining part is reset to its absolute priority, and the current priorities of other messages are increased by their priority increments. This excludes a situation when some messages would forever stay in the queue and implements a conception of the *fair merge* ([3, p. 132]): "A fair merge ... merges messages from every sender and may not ignore any sender infinitely...".

The other solution is to organize messages in several lists and to attach priorities to the lists, as it was described in [4].

5 Additional Message Queue Properties

The message queue itself is to be an object with corresponding methods. Message inclusion and dispatching are the obvious ones, but it was experienced that among queue methods are to be requests on its state. These requests are extremely useful for compound message selector and test functions when the problem of synchronization of many message flows appears (see Section 6 below). In selectors and tests, it may became necessary to ask the queue length, to check the existence in the queue of particular type messages, to access some information from messages in the queue, etc.

Another class of useful queue methods are requests on compound message interpreter state. Examples of useful requests are the current number of repetition of while message body, the current depth of compound or while message nesting, etc.

6 An Example of Message Flow Management

Let us refer to a spelling checker. After the user selects a text file to check, we have three independent message sources. The first message flow comes from the checking process, which includes text scan, vocabulary search, screen scrolling etc. The second source is the user which can at any moment press a button or click the mouse, e.g., to interrupt the checking. The third source is the suggestion process. When the scan stops after the error, it is desirable to search in the vocabulary words which are under some criteria near to the erroneous one. At any moment user may wish to stop this search if he/she sees the proper word in the suggestion list.

The implementation of the case by standard means is not so trivial, see [4]. It included the reprogramming of the event queue manager to make it to generate a broadcast message `FindTextScanProcess` every time when no other messages existed in the system. The scan process message handler, if such process exists, reacts to this message by executing its `Update` method. This includes taking the next word, checking it against the vocabulary etc. It is difficult to manage many

independent processes in such manner.

Using the approach with compound messages, we only need, after text selection, to put into the queue the while message with `ScanNextWord` command message as the body, and with the test function yielding **false** at the file end, or at the user request (the user presses the *Esc* button, the external keyboard message is processed, and a flag is set), or when the erroneous word occurs.

7 Message File Creation

The property to replay message files may be useful for demonstrations and educational software, or in debugging. There are several methods to create the message file.

The first method is to log primitive messages dispatched from the queue. The difficulty is that not all messages are to be logged—we need only those one which were not results of processing previous messages, i.e., only those external or generated by an object's own initiative. One possible solution is to mark each message with a log flag (0—do not log it, 1—log it). For compound messages, the log flag value of the message is to be used for all its submessages. The log is to be one of the queue object actions.

A possible variance of the first method is to log the topmost level messages, primitive or compound, selected to process. See a discussion on pointers later in this section.

The second method is to write special program for message file creation which will contain message generation function as arguments of the `WriteMessage` procedure. The method is less desirable because it is necessary to change the corresponding program each time as we want to generate slightly different message sequence.

The third method is to define a special language to describe messages, and to develop the corresponding compiler etc. A difficulty arises with selector and test functions. This approach may be used, however being restricted by argument length limits, in generating messages in the program. We can have a function compiling a message script to a message.

The fourth method is to create a special kind of resource editor to create message sequences in conversational manner.

Two problems exist in all cases. A problem with selector and test functions may be solved by storing their names and reconstructing links dynamically.

The principal problem of message storing is the same as of object storing—the storing of pointers. We can dereference pointers to constituent messages, but even the primitive message may contain a pointer as the information. Example of the technique may be found in [2], and another known solutions exists: using identifiers (handles) instead of pointers etc.

8 Conclusions

If several independent sources asynchronously produce messages which are in some cases dependent, we are to complicate program structure in many places to deal with the situation. The proposed approach uses more complicated message queue manager and compound messages, and permits to encapsulate the information on message dependence in messages themselves and in their selection and test functions, simplifying programs and object design, and increasing readability and function isolation.

References

[1] D.W.Singer, SCENARIOS: An Event Management Package. Software—Practice and Experience, **vol. 11**, 521–529 (1981).

[2] Turbo Pascal, Version 6.0, Turbo Vision Guide. Borland International, Inc., 1990.

[3] Gul Agha, Concurrent Object-Oriented Programming. Communications of the ACM, vol.33, No. 9, September 1990.

[4] A.Colesnicov, L.Malahova, Event synchronization in object oriented environment—a case study.

9th International Conference on Control Systems and Computer Science CSCS9, Conference Preprints, Bucharest, 25–28 May 1993, **vol. II**.

A.Colesnicov,
Institute of Mathematics,
Academy of Sciences of Moldova,
5 Academiei str., Kishinev,
277028, Moldova
phone: (373–2) 738058
e–mail: *kae@math.moldova.su*

Received 3 July, 1995