C++ Idioms for Concurrent Operations

Fernando Náufel do Amaral, Patricia Garcês Rabelo, Sergio E. R. de Carvalho[*]

Laboratório de Métodos Formais, Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente 225, Rio de Janeiro, RJ, 22453-900, Brasil
email: {fnaufel, rabelo, sergio}@inf.puc-rio.br

Abstract. Well-integrated development tools, allowing automatic code generation from visual representations of analysis and design decisions, are important assets in handling the complexities of today's software. This paper describes several message passing semantics for the expression of concurrency in a new object-oriented visual development system, along with the C++ idioms generated for asynchronous messages.

Keywords: Message semantics, concurrent operations, automatic code generation, development tools, C++ idioms

1 Introduction

Expected increases in developer productivity, thus reducing time to market, correctness by construction, and design and code reuse suffice to justify large investments in the construction of object-oriented CASE tools. The automatic generation of quality, executable code from visually represented designs is one of the most desirable aspects of such tools. In this paper we present C++ code automatically obtained from several message passing semantics available in a new development environment, 2GOOD/DDL.

2GOOD (Second Generation Object-Oriented Development) (Carvalho+98) is a new object-oriented design tool. 2GOOD designs are automatically represented as code in DDL (Design Description Language) (Carvalho97), a very high level

* This work was supported by CNPq, in the context of project ARTS.

intermediate language, to smooth out transformations from visual designs to C++. The translation from DDL to C++ is accomplished with TXL (Tree Transformation Language) (Cordy+95), a programming language whose basic paradigm involves the use of correctness-preserving transformation functions and rules to process input data after it has been converted to tree format.

Section 2 gives an overview of object behavior in 2GOOD/DDL, and a more detailed rendition of its message passing features. Section 3 describes, via small examples, the C++ idioms used in modeling asynchronous messages. Section 4 briefly discusses other message semantics, and section 5 presents our conclusions.

2 Object Behavior in 2GOOD/DDL

Several semantics may be used in the specification of object behavior: procedural, concurrent, cooperative, exceptional. This paper concentrates on message handlers, which promote concurrency while hiding from users platform resources as message queues and execution threads, and lower level concurrency constructs like semaphores and mutexes.

Modern execution platforms, operating systems and run-time libraries, include implementation facilities for these semantics, such as queues for message passing and threads for concurrency. The selection of the correct semantics for the situation at hand may improve system quality. This can be useful, for example, in relaxing synchronization constraints imposed by the procedural behavior, which may not exist in the domain being modeled or in the application being developed.

There are three kinds of message handling operations in 2GOOD/DDL: *asynchronous*, *handshake* and *future*.

Asynchronous Messages. An object that sends an asynchronous message does not block: execution of the sending operation continues normally after an asynchronous message send. If the receiver has a handler for that specific message, it may be executed concurrently with the sending operation. Asynchronous message handlers can only have input parameters, and cannot have return values, since caller and callee never resynchronize. Figure 2.1 outlines a DDL program that includes an asynchronous message handler. In procedure Main, an Msg1 message is sent to the 'objx' object, modeled by class X. Upon receiving the Msg1 message, 'objx' will execute the body of the Msg1 handler.

Handshake Messages. Unlike asynchronous messages, when a handshake message is sent, the sender is blocked until it receives an acknowledgment from the receiver. This behavior allows synchronization of sender and receiver at the beginning of handler execution. Handshake handlers can have only input parameters. Local variables are permitted; return values are not, since no processing takes place before the acknowledgment is sent.

Future Messages. Future message handlers may have output parameters. When an object sends a future message, it can continue executing until an output parameter becomes necessary. At this point, the sender blocks issuing to itself a WAIT message, naming as an argument the receiver it is waiting upon. When the handler terminates, output parameters are passed back and then the sender is released. If a future message handler has no output parameters, the WAIT message becomes merely a synchronization point. If the execution of the handler terminates before the WAIT message is issued by the sender, the sender is not suspended.

```
CLASS Main;                        CLASS  X;
     PROCEDURE Main;                ASYNC HANDLER Msg1 (IN Integer n)
            X  objx;                              Integer  i;
     BEGIN                                 BEGIN
            objx <- Msg1 (3);                     …
     END PROCEDURE                  END HANDLER
END CLASS                          END CLASS
```

Figure 2.1: Asynchronous message send

3 Mapping Messages onto C++

One way to achieve concurrency within an application is through the use of multiple *threads of control*. The C++ programs generated by our transformation process use threads to simulate DDL's concurrent operations. There are now many libraries that allow C++ programs to manage the creation and use of threads.

In this section, we describe platform-independent solutions to the problem of simulating the semantics of DDL's message handlers in C++. The example C++ programs used include comments describing the creation and destruction of threads, and specific inter-thread communication and synchronization mechanisms, such as semaphores and critical sections.

First, we detail the solution for asynchronous messages; next we discuss the new elements introduced in the simulation of handshake and future messages.

Elements involved. Figure 3.1 shows the elements used to simulate asynchronous message handlers in C++, and a trace of the events that take place when an asynchronous message is sent and handled.

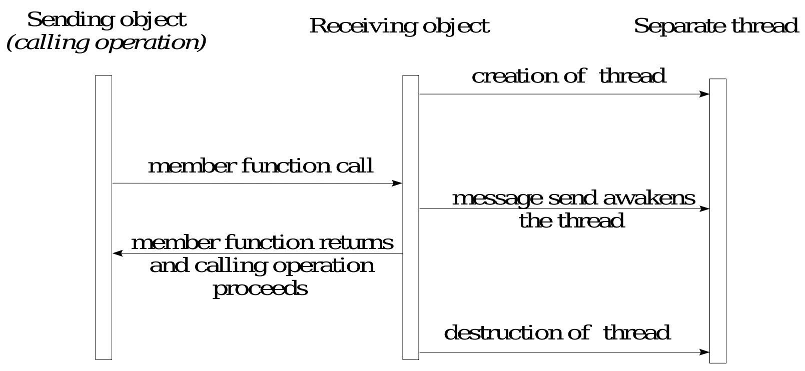

Figure 3.1: Event trace for asynchronous message handling in C++

Message send**:** sending an asynchronous message to an object is simulated by calling the C++ member function created to correspond to its handler. All member functions receive pointers to their arguments, rather than copies of them. The identity of the sending object (i.e., the pointer *this*) is passed as an additional argument (Figure 3.9).

Member function**:** the body of the member function does *not* contain the commands of the DDL handler it simulates, since a C++ member function is an inherently non-concurrent operation. Instead, its sole purpose is to send a message to a separate thread that will actually execute the commands of the original handler. After this message is sent, the member function promptly terminates, freeing the sender.

Arguments**:** the message sent to the separate thread must carry along the arguments passed by the sender. Copies of the arguments are gathered in an instance of a special class defined for this purpose, and a pointer to this instance is sent to the separate thread.

Separate thread**:** whenever a class contains message handlers, its C++ constructor will include commands to create a separate thread for each one of them. In our solution, the code to be executed by the separate thread is defined in a global function.

The following subsections discuss the finer points of the simulation and include outlines of the C++ code generated by the translation of the DDL example of Figure 2.1.

```
              1.    void Main :: main_proc (void)
2.    {
3.      X * objx;
4.      objx = new X (...);
5.      objx -> msg1 (Integer (3), this);
6.      delete objx;
7.    }
```

Figure 3.9: C++ code for DDL procedure Main in Figure 2.1

Ensuring that every object can receive messages. According to the DDL language definition, every object is able to receive messages, even when its current class and superclasses do not define handlers for them; in this case no action is taken by the receiver.

Because all classes in the C++ program must offer some common basic features, they are defined as descendants of the Object class, which implements these features. Given this arrangement, one simple way to ensure that every object can receive messages is to alter the definition of the Object class so as to make it include virtual member functions to handle all possible messages in the program (Figure 3.9, line). These virtual member functions do nothing (i.e., their bodies have no commands), immediately returning control to the caller whenever invoked. Thus, if a particular class has a handler for a certain message, it will override the corresponding member function inherited from Object (Figure 3.9, line); otherwise, the inherited version will remain in effect.

Sending the arguments to the separate thread. When the member function passes the message on to the separate thread, it must also send the corresponding arguments and the identity of the original sender. Before the member function sends its message to the separate thread, the arguments and the identity of the sender are bundled together (Figure 3.9, line) in an instance of a special class defined for this purpose (Figure 3.9). What is sent along with the message is a pointer to this instance.

The member function contacts the separate thread via whatever inter-thread communication mechanisms provided by the thread package being used. The strategy of marshaling the arguments in one instance of ArgClass and then sending a pointer to it requires that those communication mechanisms allow at least one argument to be passed in an inter-thread message. This is more reasonable than requiring that inter-thread messages support the same number of arguments as any message handler in a DDL program. Note that, because different threads of the same process share a common address space, the separate thread can freely access the instance of ArgClass, no matter where it is allocated. Since asynchronous message handlers can only have value input parameters, what is stored in the instance of ArgClass are pointers to *copies* of the arguments, thus simulating argument passing by value.

```
1.    class Object
2.    {
3.      // (...)
4.      virtual void msg1 ( Integer *_n, Object *sender ) {};
5.      // (...)
6.    };
7.
8.    class Main : public Object {
9.      public :
10.     Main (...);
11.     ~ Main ();
12.     void main_proc ( void );
13.   };
14.
15.   class X : public Object {
16.     public :
17.     // Declare MUTEX;
18.     virtual void msg1 ( Integer *_n, Object *sender );
19.     X (...);
20.     ~ X ();
        21.   };
```

Figure 3.9: C++ class definitions for asynchronous message example

```
        1.    void X :: msg1 ( Integer *_n, Object *sender )
2.    {
3.      ArgClass_msg1 *arg_obj_msg1;
4.      arg_obj_msg1 = new ArgClass_msg1 ( _n, sender );
5.      // Send message to separate thread, with arg_obj_msg1;
6.    }
```

Figure 3.9: C++ member function corresponding to Msg1 handler

Thread creation and destruction. Constructors and destructors of classes that have message handlers must include actions to manage the creation and destruction of the separate threads. Every message handler causes one separate thread to be created in the constructor of the class to which it belongs (Figure 3.9, line). The code to be executed by the separate thread is that contained in a global function that has the same name as the message handler (Figure 3.9). A pointer to this global function may be passed as an argument to the package-specific service for thread creation.

```
1.    class ArgClass_msg1
2.    {
3.      public :
4.      Integer *n;
5.      Object *sender;
6.      ArgClass_msg1 ( Integer *_n, Object *_sender );
7.      ~ ArgClass_msg1 ();
8.    };
9.
10.   ArgClass_msg1::ArgClass_msg1 (Integer *_n,Object *_sender)
11.   {
12.     n = new Integer;
13.     *n = *_n;
14.     sender = _sender;
15.   };
16.
17.   ArgClass_msg1 :: ~ArgClass_msg1 ()
18.   {
19.     delete n;
20.   };
```

Figure 3.9 Definition of C++ class to hold message arguments

To ensure that an instance will have only one of its operations executing at any given time, a package-specific mutex mechanism is declared as a member of the modeling class (Figure 3.9, line) and initialized in the constructor (Figure 3.9, line); the entire body of the message loop in each separate thread is considered a critical section. Each instance defines its own mutex, thus preventing different messages to the same instance from being handled concurrently, which could cause the instance to enter inconsistent states.

When the separate thread is created, the code in the global function is executed (i.e. the thread is created in an active state). First, it must declare a buffer to hold the contents of incoming messages (Figure 3.9, line 3). Next, if the message handler takes parameters, memory positions to hold them are declared and allocated (Figure 3.9, lines 4 – 6). After this initialization, the separate thread enters a message loop, where its execution is suspended until a new message arrives (Figure 3.9, lines 9 – 28).

The separate thread is closed and destroyed on the occasion of the destruction of the instance it was created to serve (Figure 3.9, line). The destructor of the instance's modeling class causes the separate thread to exit its message loop, whereupon the memory positions previously allocated for the parameters are freed (Figure 3.9, line 30).

```
X :: X (...) : Object (...)
{
  // Initialize MUTEX;
  // Create thread to execute code in ::msg1 ( X* );
}

X :: ~ X ()
{
  // Terminate separate thread;
  // Terminate MUTEX;
}
```

Figure 3.9: Constructor and destructor of class X

```
void msg1 ( X *tthis )
{
  // Declare buffer to hold incoming message;
  Object *sender;
  Integer *n;
  n = new Integer;
  ArgClass_msg1 *argptr;

  // Message loop:
          // Wait until new message arrives;
          // Enter critical section;
          argptr =       // ptr to instance of ArgClass_msg1
                         // received with message;
          sender = argptr->sender;
          *n = *( argptr->n );

          {       Integer * i;
                  i = new Integer;
                  // Translation of handler statements
                  delete i;
                  goto end_handler;
          }

          end_handler:

          delete argptr;
          // Exit critical section;
  // End of message loop

  delete n;
}
```

Figure 3.9: C++ global function to simulate Msg1 handler

The message loop. The message loop is where the separate thread spends most of its life. Package-specific mechanisms suspend the execution of the thread when there are no messages to be handled. Incoming messages are put on the receiving instance's message queue and handled one at a time. When a new message arrives, the thread is awakened and the following events take place (Figure 3.9): the critical section is entered (line 11); copies of the arguments and the sender's identity are obtained from the instance of ArgClass (lines 14 – 15); the statements originally contained in the body of the DDL message handler are executed, including the allocation of locally declared variables (lines 17 – 22) (if these statements need to access features of the instance of the class in which the handler was declared, they can do so via the *tthis* pointer, passed to the separate thread as an argument on the occasion of its creation); the instance of ArgClass created by the member function is destroyed (line 26); the critical section is exited (line 27).

Multiple handlers for the same message. Each class in a DDL program can define its own handler for a message. Besides, each class can also handle the same message differently, depending on the sending object. In Figure 3.9, if an instance of class X receives a message Msg1 from object 'objy', the code of the first handler will be executed. If the sending object is 'objz', the second handler will be invoked. If the message was sent by any other object, the last handler will be executed.

```
CLASS Y;... END CLASS
CLASS Z;... END CLASS
CLASS X;
     OBJECT Y objy; Z objz;
     ASYNCHRONOUS MHANDLER Msg1 FROM objy;
       ...
     ASYNCHRONOUS MHANDLER Msg1 FROM objz;
     ...
     ASYNCHRONOUS MHANDLER Msg1;
     ...
END CLASS
```

Figure 3.9: Multiple handlers for the same message

Figure 3.9 shows the C++ global function generated from the example in Figure 3.9. The C++ statements that correspond to the bodies of the handlers are included in the global function whose code is executed by the separate thread. The body of the global function contains *if* statements (lines and) to determine the sender's identity and execute the appropriate section of code – i.e., the one corresponding to the body of the associated sender's handler.

```
1.    void msg1 ( X *tthis )
2.    {
3.              // Declare buffer to hold incoming message;
4.              Object *sender;
5.
6.              // Message loop:
7.              // Wait until new message arrives;
8.             // Enter critical section;
9.             argptr = // ptr to instance of ArgClass_msg1
10.                     // received with message;
11.
12.            sender = argptr->sender;
13.
14.            if ((sender == (tthis -> objy)))
15.            {
16.               // Translation of commands in DDL handler for objy;
17.               goto end_handler;
18.            }
19.
20.            if ((sender == (tthis -> objz)))
21.            {
22.               // Translation of commands in DDL handler for objz;
23.               goto end_handler;
24.            }
25.
26.            {
27.               // Translation of handler body without FROM clause;
28.               goto end_handler;
29.            }
30.
31.            end_handler:
32.
33.            delete argptr;
34.            // Exit critical section;
35.            // End of message loop
36.   }
```

Figure 3.9: Global function for multiple handlers

If the DDL class has a default handler for the message (one whose signature does not include a FROM clause), the C++ code for the body of this handler will be the last one in the sequence of blocks in the global function, and will not be guarded by a test of the sender's identity (lines –). To allow this kind of behavior in the C++ program, the identity of the sending object must be sent to the separate thread by the member function, as stated above.

4 Other message semantics

The simulation of handshake messages in C++ differs from that for asynchronous messages only in the synchronization between sending and receiving objects, which occurs at the beginning of message handling. This we accomplish with condition variables, associated with each handshake message handler. These variables are declared as data members of the class, initialized in its constructor and destroyed in its destructor.

When the member function corresponding to the handler is invoked, it resets the condition variable, sends the message to the separate thread and then suspends itself on the condition variable. The separate thread has access to the features of the receiving object, including the condition variable; as soon as it enters the critical region, it signals the conditional variable, thus releasing the member function from its wait, returning control to the sending object.

Future message handlers may declare output parameters; besides, the calling operation and the handler may have to re-synchronize upon reception of the WAIT message by the sender. These are the main differences between the C++ code generated for asynchronous messages and that generated for future messages. Output parameters generate members in ArgClass that, like input parameters, are pointers to the corresponding arguments.

As with asynchronous messages, member functions associated with future message handlers also receive pointers to their arguments, but these are treated in different ways: arguments corresponding to input parameters are copied, and pointers to the copies are stored in the instance of ArgClass; arguments corresponding to output parameters are not copied – instead, pointers to the original values are stored in the instance of ArgClass. This enables the separate thread to alter the values of the arguments when it finishes handling the message, correctly simulating output argument passing.

In the body of the calling operation, before any other commands, a semaphore is created for each object to which a future message is sent. Upon creation, a semaphore is in the unsignaled state. On the occasion of a message send, this semaphore is passed as an extra argument to the member function, which in turn includes it in the information sent to the separate thread.

When execution reaches a WAIT message associated with the corresponding receiving object, the calling operation waits on the corresponding semaphore. If the semaphore is unsignaled, the calling operation is suspended until the semaphore becomes signaled.

It is the separate thread that signals the semaphore when it finishes handling the message, thus releasing the calling operation from its wait. In the event that the separate thread finishes handling the message and signals the semaphore before the WAIT message is reached by the calling operation, the calling operation is not suspended.

In the simulation of future messages, because the calling operation always waits on the semaphore at some point in time after calling the member function of the receiver, the virtual member function included in the Object class to ensure that any object can receive messages must now include a command to signal the semaphore. Thus, calling the member function of an object whose modeling class does not override the implementation inherited from the Object class will cause the semaphore to be signaled, and the calling operation will not be suspended when it reaches the WAIT message.

5 Conclusions

In modeling dynamic views for object-oriented systems, the highly synchronized procedural semantics may impose severe restraints on execution. The current emphasis on the development of distributed applications enforces the need for more ways to define object behavior, as suggested in (Johnson97). Using these features designers can fine-tune their systems, without having to resort to class libraries, typically difficult to use, or to directly access platform resources, compromising system reuse.

However, to a high level specification, there must correspond executable code. In this report we describe the specification and the implementation of message passing features in 2GOOD/DDL. We use the TXL transformation process to automatically generate C++ code corresponding to asynchronous, handshake and future message handlers.

We used Version 8 of the TXL Programming Language. The resulting C++ code was compiled using Visual C++ 5.0 on Windows NT 4.0. Our transformation system has been tested on a large number of DDL programs, including real time software for the operation of a PABX system. One small example was a 196-line DDL program, which made significant use of concurrency, and which corresponded to 753 lines of C++ code. This increase in size is due in part to the fact that C++ classes must handle concurrent features explicitly, as seen above.

References

(Carvalho97) S. Carvalho, “The Design Description Language”, MCC29/97, PUC-Rio, Dept. de Informática, , Sept 97, available from ftp://ftp.inf.puc-rio.br/ftp/pub/docs/ techreports

(Carvalho+98) S. Carvalho, S. de O. e Cruz, T. C. de Oliveira, “Second Generation Object-Oriented Development”, Electronic Notes in Theoretical Computer Science, URL: http:// www. elsevier /nl/ locate/entcs/volume14.html

(Cordy+95) J. Cordy, I. Carmichael, R. Halliday, “The TXL Programming Language Version 8”, Legasys Corp., April 1995

(Johnson97) R. Johnson, “Frameworks = (Components + Patterns)”, CACM vol.40 (10), pp.39-42, October 1997