Applying Object-Oriented Techniques in Embedded Software Design

Sumithra Bhakthavatsalam and Stephen H. Edwards

Department of Computer Science
McBryde Hall, Virginia Polytechnic Institute and State University
Blacksburg, VA-24061 USA

***Abstact*—In order to study the impact of using object oriented (OO) techniques, a real-time micro-kernel, its application programming interface, and several power electronic control applications are being redesigned using an OO approach. The design, implemented in C++, of the kernel is presented and explored.**

I. Introduction

Efficiency is a major concern for real-time systems. Real-time software is often written in assembly or other low level languages like C. Programmers hesitate to use newer object-oriented techniques since these are perceived as less efficient. This work explores the OO design techniques for embedded operating system, taking into account real-time goals so that the design is efficient and meets system deadlines.

The embedded system kernel being designed is based on Dataflow Architecture. This architecture, which supports asynchronous communication amongst the various components through communication channels, was chosen since it addresses the objectives of modularity, standardization and reusability of Power Electronics Building Block (PEBB) systems, by the use of Elementary Control Objects (ECOs). An ECO is a functional entity in the PEBB system that can be used independent of the system details, in a variety of systems. It thereby serves as a modular, standardized and reusable component.

An entity like the ECO comes across as ideal to be modeled and implemented as an object; i.e., using the OO approach. More over, since the advantages of object-orientation are in line with our objectives of modularity, standardization and reusability, it is ideal to implement the system as an object-oriented one. For instance, an ECO can be represented by a class. If this class is used to represent the general attributes and certain generic functions of all ECOs, then we can inherit specific types of ECOs from this class, that differ in their implementation code due to their functional differences. The ECOs are connected with each other to form the required PEBB system, and communicate with each other through communication channels, known as Data Channels since they carry data between ECOs to facilitate inter-ECO data flows. The Data Channels are also represented by classes. Data Channel objects differ in their type based on the type of data that they carry. ECOs operate based on Firing Rules. A firing rule is a set of priority firing masks, each of which defines one of the possibilities that will cause an ECO to “fire” or begin operation. A firing mask is a bit pattern that has bits set in positions corresponding to input data channels that need to have data for the ECO to fire and a priority firing mask is the combination of a firing mask and an integer priority representing the priority that the ECO will acquire when it gets triggered by that mask.

The paper discusses the design of the object-oriented kernel and comparisons of the C version of the kernel (DARK – Dataflow Architecture Real-time Kernel) and its C++ version, DARK++.

The following section gives an insight into the performance problems that object-oriented systems pose and the reasons for them. Section III describes the OO design of DARK++. Section 1V does a comparison of the C and the C++ kernels. The paper concludes with the most important lessons learnt so far from the research and a brief note on the future work.

II. Performance Issues in OO Systems

Several aspects of object-oriented programming have a significant impact on run-time performance. The core design issues that lead to this overhead are the use of heap-allocated memory, dynamic binding, and method call overhead.

A. Heap-Allocated Memory

Object-oriented systems impose performance issues due to the extensive use of dynamic memory allocation, dynamic creation and destruction of objects, and use of pointers. This causes overhead in memory management and the repeated dereferencing of pointers. Heap memory management in particular leads to unpredicatable time delays in underlying system calls. In addition, “pointer chasing,” or following indirect references from one object to another as computation proceeds through a series of related objects, is a well-known problem in the context of OO systems. Pointer chasing increases memory traffic and due to lack of data locality, causes cache inefficiency.

B. Dynamic Binding

Dynamic binding refers to deferring until run-time the association between a reference to a program operation (such as a class method) in calling code and the actual entry point of the corresponding segment of machine code that implements that operation. Typically dynamic binding in object-oriented systems occurs due to the use of virtual methods. Virtual methods are operations that are declared in a base class, but which may have different implementations in subclasses. When other program code that is written to operate on any object—either an object of the base class or of any of its subclasses—makes method calls, the correct implementation of the method in question must be used. As a result, no fixed code address can be associated with that method call—a different address must be used depending on the particular (sub)class to which the object belongs.

Dynamic binding is usually implemented using indirection. The actual entry point is looked up at run-time in a dispatch table. The cost of performing the lookup and using the extra level of indirection to execute operations significantly increases the overhead of calling methods.

C. Method Call Overhead

While dynamic binding can be avoided in many situations, method calls still introduce performance concerns to the time spent in setting up the stack frame and the actual transfer of control to the called methods. While this need not take any longer in an OO language than regular procedure or function calls in a non-OO language, the OO paradigm inherently means more method calls. Because classes typically have a greater number of smaller methods, most behaviors use a greater number of method calls arranged in deeper call trees. This pattern leads to additional overhead in comparison to non-OO languages such as C.

It should be noted that having large methods to reduce the number of method calls would conflict with the goal of software reusability, because the more generic a component is, the more reusable it is. Thus, in general terms, methods in OO systems are small and generic, with behavior being extensible through inheritance. The subclasses usually have methods that, after doing a more specific operation, make calls to the methods in the base class. Thus, the overhead of method calls is a significant performance issue in OO systems.

III. OO DESIGN OF THE KERNEL

The various entities of the DARK++ system are modeled as objects using C++ classes. The primary types of objects in our system are: elementary control objects (ECOs), data channels, the DARK++ kernel, and the ready queue.

A. Data Channels

Data channels may store different types of data such as int, char, bool, float, short, or double; i.e., scalar types, or vector data, which are essentially arrays of various dimensions. They could also store strings or data in the form of raw bytes. All these types of data channels are subclasses that inherit from the base class, `Data_Channel`. To create scalar data channel types, there is a template called `Scalar_Data_Channel`. This template inherits from the `Data_Channel` class. The data channels for scalar data types are created from this template by passing the appropriate data type as a parameter. Since strings are handled differently, there is a separate `String_Data_Channel` class that inherits from the base `Data_Channel` class, but is not constructed out of the template. Essentially, `read` and `write` operations on data channels have to read / write data in the form of bytes. Hence the `Data_Channel` base class contains methods `Read_bytes` and `Write_bytes`, which are protected and can be accessed only from any of the derived data channels. These methods are used by all the scalar type data channels and the `Byte_Data_Channel`. `String_Data_Channel` has its own `read` and `write` methods that do not make calls to the base class's `Read_bytes` and `Write_bytes` methods. Interfaces for these are shown in Figures 1 through 4. Data channels for user-defined data types like multidimensional arrays can be created by using the `Vector_Data_Channel` template that takes in the user-defined type and the traits. The relationships among the data channel classes are shown in Figure 7.

```
Data Channel  Base Class
Public:
Capacity();     // max. no. of entries
Entries();      // current no. of entries
Available_Capacity();
Flush(int );    // remove given no. of entries
                // from   end of queue
Blocked();      // returns true if blocked
Protected:
Register_OS(); //registers with DARKpp
setBlocked();
resetBlocked();

/* Read_bytes follows: pass  char pointer to
    read, no. of bytes to    be read */
Read_bytes(char* , int );

/* Write bytes follows: pass char array to be
    written,no. of bytes*/
Write_bytes(char* , int );
```

Figure 1. Data Channel Interface

```
 Scalar Data Channel<Scalar T> Template Class
Public:
Read(Scalar_T& );// pass reference parameter
                 //of appropriate type to read
Write(Scalar_T );// pass parameter of
                 // appropriate type to write
```

Figure 2. Scalar_Data_Channel template class Interface

```
String Data Channel Class
Public:
Read(char* );  //pass char pointer to read in
               //string
Write(char* ); //pass char array write
```

Figure 3. String_Data_Channel class Interface

```
Byte Data Channel Class
Public:
/* Read_bytes follows: pass  char pointer to
    read, no. of bytes to be read */
Read_bytes(char* , int );
/* Write_bytes follows: pass char array to be
    written, no. of bytes */
Write_bytes(char* , int bytes);
```

Figure 4. Byte_Data_Channel Class Interface

B. ECOs

In the OO system every ECO is an object. Since there are certain common attributes for all ECOs, we have an abstract base class called `ECO` from which specific ECOs inherit. The implementation function in this base class is a pure virtual function because different ECOs can provide vastly different behavior. The ECO interface is shown in Figure 5.

An ECO can be viewed as a process that executes its `Implementation` code provided by the ECO designer. The various possible states of these processes are: `nascent`, `ready`, `blocked`, `wait_for_fire`, `timed_wait`, `timed_wait_for_fire` and `dead`. When the kernel starts, every process is in `nascent` state. A process waits for data on its input channels and when the channels have data, the process is triggered, and it goes into the `ready` state. The process is `blocked` when it tries to read from an empty data channel or write to a full data channel. Every ECO object "knows" its input and output data channels—these are assigned in the constructor and so need to be specified when the ECO object is being created. As explained in Section I, ECOs operate based on Firing Rules. After every `read` operation on a data channel, the status of the source ECO (ECO that writes to this data channel) is checked. If the source ECO is blocked, then it is unblocked. After every `write` operation, the mask of its sink ECO (ECO that reads from this data channel) is updated; i.e., the bit corresponding to the data channel in question is set. Thus, while a `read` operation could unblock a process blocked on a data-channel, a `write` operation could fire it.

The `wait_to_fire` function can be used to fire the ECO again. If the ECO is not ready for firing, it goes into `wait_for_fire` state. The user can also delay the execution of the ECO for a pre-determined time, which puts the ECO into `timed_wait` state. The `timed_wait_-for_fire` state is a combination of `wait_for_fire` and `timed_wait`. If an ECO is in this state, it can be fired if a firing mask becomes true or the time period elapses. The ECO goes into `dead` state, when it finishes execution.

An ECO designer should take the following steps to write a class for a specific ECO:

- Call the base ECO class constructor from the constructor of the new class with the parameters: number of input and output ports, the firing rule for the ECO, the initial priority, an array of pointers to the input ports and an array of pointers to the output ports.
- Set the specific configuration information for the ECO in the new class's constructor.
- Write the implementation function and any other new functions if required.

```
Class ECO
Public: Implementation() = 0;   // pure virtual function
        Wait_To_Fire();         //returns true or returns to OS
        Wait(Time );
        Register_OS();          // registers with DARKpp
        Current_Priority();     // returns current ECO priority
        SetPriority(Priority );// sets ECO priority
        Set_OS_Call();                 // sets flag in_OS_call to true
        Reset_OS_Call();
        in_OS_Call();          // returns true if process is in OS call
        SetIn_Ports_Ready(Firing_Mask );//set mask after write
        Timed_Wait_To_Fire(int );      // explained in subsection "ECO"
        Delay(int );
        In_Ports_Ready();      // returns mask showing ready ports
        SetProcess_State(ProcessState );
        Blocked();             // returns true if process is blocked
        SetWakeup_Call(Firing_Mask ); // sets the mask that fires the ECO
        Wakeup_Call();         // returns the mask that woke it  up
        Process_State();       // returns process state
        FIRE_Rule();           // returns firing rule
        Get_Heap_Position();  // gets process's position in ready queue heap
        Set_Heap_Position(int );    // used to alter position of ECO in ready queue
        swap(ProcessState );   // changes process state and goes to OS
        ECO_Env();             // returns context information for the process
        Stack();               // returns pointer to the stack in  which process is running
        StackSize();           // returns size of process stack
Protected: Register_As_Source(int ); // register as source ECO of output data channel(s)
           Register_As_Sink(int );   // register as sink ECO of input data channel(s)
```

Figure 5. ECO Class Interface

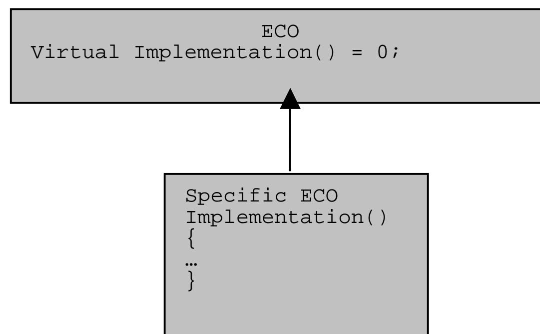

Figure 6. Class Diagram for ECO Classes

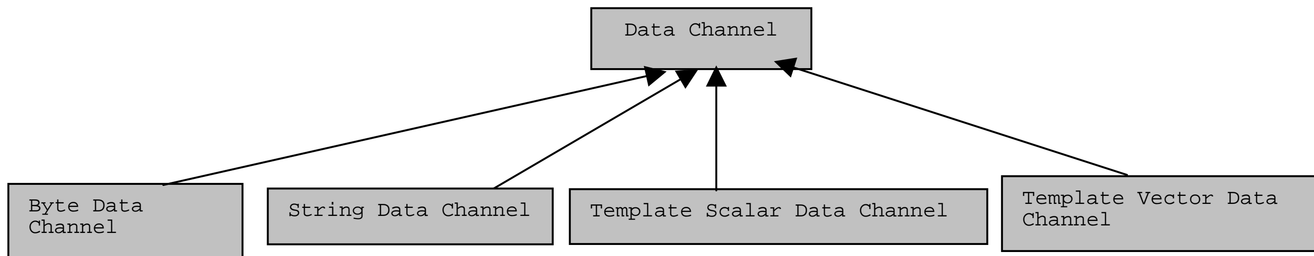

Figure 7. Class Diagram for Data Channel Classes

C. *Ready_Queue*

The qeady queue is a heap that is used by the kernel to organize the set of processes that are ready to run. It is constructed based on process priorities, with the process with the highest priority being at the head of the queue. Processes (ECOs) are put on to the ready queue once they are found to be in the `ready` state, usually because data has arrived on enough input data channels to trigger the ECO's firing rule.

D. The DARK++ Kernel

The kernel is responsible for running ready ECOs. Processes (ECOs) are initially in the nascent state. When data is written to their input data channels, they are triggered and move to the `ready` state. As soon as a read / write is completed, the sink ECO is put on to the ready queue if required. The OS (DARKpp) removes the head of the ready queue each time it needs to execute a process. The processes iterate to execute a special function called `wait_to_fire` every time through. This function causes the ECO to relinquish control since it has finished executing once, and the control is transferred to the kernel.

After getting control, the kernel compares the priority of the current process with that of the next available process in the ready queue and the process of higher or equal priority gets executed. The execution of a process with equal priority as the current process is done to ensure fairness – preventing a single high priority process to run indefinitely, thereby blocking all other processes, causing the problem of starvation.

Figure 8 summarizes the main responsibilities of each class.

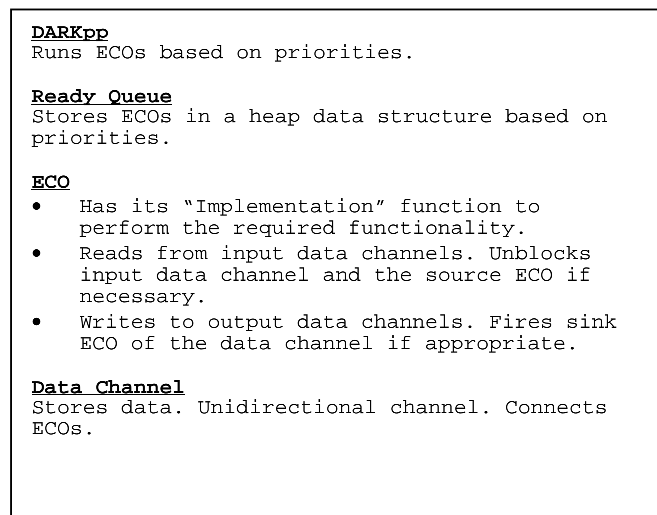

Figure 8. Responsibilities of the classes

IV. Comparison of the C and the C++ Kernels

This section covers an interesting and often debated topic of whether OO approach is preferable in the context of real-time embedded systems, to non-OO approaches, and gives an insight into the main differences in features from the user's point of view.

In DARK, the application designer gives as input, the Data Flow Graph (DFG). This contains node and edge descriptions for the DFG. DARK runs through this graph to allocate memory for all the ECOs and Data Channels, initializes them and then begins execution of the processes. In DARK++, the user declares all the Data Channel objects and then all the ECO objects with the appropriate information passed as parameters to the respective constructors. This creates the

Data Channel and ECO objects and then DARK++ can be called to begin execution.

While in the C version, ECOs are written as functions, in DARK++ these are written as classes. It is here that the OO advantage of reusability can be seen. Currently, we have a base class for an ECO that has a pure virtual function for the implementation code and this function can be written appropriately for any particular ECO. However the use of the OO approach is not limited to this. If there are ECOs that have certain common functionality, then we can have a common class from which they inherit. Development of one ECO from another is also possible as and when required.

The data channels can also be extended to carry any arbitrary user-defined.

One of the primary advantages of using C++ is the type safety that it ensures. For example, the configuration parameters for ECOs have their specific structure defined within the subclass for that particular ECO. So in the constructor for the ECO subclass, one needs to pass the configuration parameter in the right format, failing which compilation does not go through. However, in the C version, all ECOs are treated in a uniform way and are just functions. A void pointer points to the configuration information, increasing the scope for errors.

While the advantages are substantial to talk about, the perceived performance issues are equally important to consider, especially in the context of real-time systems since they have time-critical tasks. These issues have already been outlined in Section II. The general expectation is that these disadvantages would far outweigh the advantages, making OO techniques unsuitable for real-time embedded systems. Experimental comparisons and data will lead to an appropriate conclusion on this issue.

V. CONCLUSIONS AND FUTURE WORK

Redesigning DARK in C++ revealed a number of advantages of using OO techniques most of which were the usually proclaimed features of OO approach.

The next step in this research will be experimental comparisons of the performance of DARK with that of DARK++. Several power control applications will be redesigned in C++. Visual DSP++ which is a compiler and emulator for the Sharc microprocessors provided by Analog Devices will be used to run both these and gather profiling data. This data will give us an idea of the overhead of using the OO approach in the design of the kernel.

ACKNOWLEDGMENT

This work was supported primarily by the ERC Program of the National Science Foundation under Award Number EEC-9731677.

REFERENCES

[1] Roy H Campbell, Gary M Johnston, Peter W Madany, Vincent F Russo, “Principles of object-oriented operating system design”, Technical Report UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign, April 1989.

[2] Yau-Tsun Steven Li, Sharad Malik , “Performance analysis of real-time embedded software”, Kluwer Academic Publishers, Boston, Nov 1998.

[3] Bjarne Stroustrup: “The C++ programming language”, III Edition, Addison Wesley.

[4] Jinghong Guo, Stephen Edwards and Dushan Boroyevich, "Improved architecture of PEBB plug and play power electronics systems: Elementary Control Object (ECO) and dataflow," in *CPES 2001 Power Electronics Seminar and NSF/Industry Annual Review,* April, 2001.

[5] Marc Tarpenning, “Cooperative multitasking in C++”, Dr. Dobb’s Journal, April 1991.

[6] Tom Green, “A C++ multitasking kernel”, Dr. Dobb’s Journal, February 1989.