Visual Aspect-Oriented Programming of 
Resource Constrained Real-Time Embedded Systems 
using the Port-Based Object Model of Computation

Thomas W. Carley  
Dept. of Electrical and Computer Engineering  
University of Maryland  
College Park, MD 20742  
tcarley@eng.umd.edu

David B. Stewart  
Embedded Research Solutions  
9687F Gerwig Lane  
Columbia, MD 21046  
dstewart@embedded-zone.com

1 Introduction

This paper describes preliminary work towards a Visual Aspect-Oriented Programming (VAOP) environment specifically for Resource-Constrained real-time Embedded Systems (RCES). VAOP is an extension of Aspect-Oriented Programming (AOP) in which some aspect languages are visual. AOP is an evolving paradigm for implementing software [15][17]. It is based on the realization that different parts of a program are best implemented using a different set of techniques, this is called separation of concerns [1].

RCES is a candidate domain that can benefit tremendously from VAOP. The domain is characterized by the very limited memory and processor resources available, such that modern programming techniques, including object-oriented design, are unnatural or too bulky. As a result, many RCES applications still use antiquated practices, like writing code in assembly language, or using C without any clear structure.

Design and implementation of RCES is improving with the advent of software component technologies. For example, the Port-Based Object (PBO) Model of Computation (MoC) can be used to implement component software for real-time control and communication applications that have very limited resources [13][11].

The problem with using the PBO or other similar component model in RCES, is that to keep the code very compact, C is still one of the few high-level languages available. As a result, the component interface must be defined manually through use of cryptic macros and careful disciplined use of coding standards.

VAOP provides the opportunity to improve development of component software through better interfaces, yet still producing code with the same level of efficiency as C by raising the level of abstraction. However, abstraction often results in less efficient code. To offset this tendency, domain specific abstractions must be used. While some generality is lost, domain specific abstractions are more efficient to implement.

In Section 2, the unique characteristics of RCES that differentiate it from desktop or non-resource-constrained embedded applications is described. A review of the PBO MoC, the domain specific abstraction used, is given in Section 3. The aspects that we have identified for RCES, as well as our technical approach to VAOP are summarized in Section 4. Our experiences with a preliminary prototype are then summarized in Section 5.

2 Resource Constrained Embedded Systems

The severe resource constraints in embedded systems are often in the form of very limited memory, processor cycles, and power utilization. For example, the MSP430F1121 microcontroller from Texas Instruments (TI) [14] is a new processor, yet it has only 256 bytes of RAM
and 4K bytes of ROM. This 16-bit processor is designed to operate in the clock range of 4 KHz to 8 MHz, yielding as few as 40 thousand instructions per second (0.04 MIPS), and at most 8 MIPS. This is two to five orders of magnitude less than a desktop computer! The primary advantages to using a microcontroller with so little memory and processing capability is in its extremely small size (half the size of a dime), low cost ($1.75 per unit, even less in large quantities), and ultra-low power (352 uwatts in normal operation at 1 MHz, 1.76 uwatts in standby mode) capabilities.

There are numerous reasons for using resource constrained embedded processors, including the following:

**Unit Cost:** Many embedded systems are part of a mass produced product or inexpensive device. For example, if a $2 microcontroller is used in a VCR instead of a $3 one, and 1 million units are made, that is a savings of $1 million. Many products are priced according to supply and demand, not cost, so a saving in cost is as good as profit. Therefore, the unit cost of RCES is critical.

**Power:** RCES are often mobile or located in remote areas and need to be battery powered. Therefore, power consumption is a very important issue in RCES. For example, consider a cell phone, users demand that they be as small and light as possible, yet have many features and a very long battery life. Power consumption in RCES is on the order of microwatts. The trade-offs are software complexity, lower memory, and CPU cycles, as described in the next few sections.

**Real-Time:** Many RCES have real-time constraints. For example, consider a system filling bottles in a beer bottling plant. If the pouring mechanism is too early or too late as the bottle moves on the conveyor belt, the beer is spilled!. The timing constraints associated with hard real-time systems make designing them complex. Tasks in hard real-time systems must be scheduled to ensure that these timing constraints are met. Analyzing a set of tasks to ensure they can be scheduled is difficult [8].

**Personnel:** In many RCES development costs are significant. While the cost of development tools can be amortized over many projects, the cost of developers can not. Therefore, the cost of developers becomes significant. Also, skilled developers for RCES are in short supply. Therefore, the RCES developer is a costly resource, and his/her time is constrained. Also, the more experienced the developer, the more costly his/her time. This makes programming environments that can be used by less experienced developers, and those that make better use of an expert’s time desirable.

**Development Time:** Time to market is continually a concern, as most embedded software projects run behind schedule and over budget. As consumers demand more advanced technology in devices, there is a push to get the latest technology into a product as soon as possible, and as cheaply as possible. In fact, a recent market survey revealed that products that get to market six months late but on budget generate about 33% fewer profits over a five year span than they would have had the product shipped on schedule [16]. Consequently, there is significant pressure to reduce development time yet still building increasingly complex applications.

**Safety & Reliability:** Many RCES are part of systems that can cause damage or injury in case of failure. Therefore, safety in these embedded systems is very important. An example is the Therac-25, a medical radiation therapy treatment device. Software defects in these embedded devices caused massive radiation overdoses to six patients, killing three [7].

VAOP is a highly promising technology because it can simultaneously address all the above concerns. It enables advanced software engineering techniques to be applied to even the small-
est of processors, thus reducing item cost and minimizing power consumption. The visual interface enables use of complex and proven real-time component methodologies, while simplifying the development so that even rookie engineers can build reliable systems. The abstractions of VAOP also enforce proven programming structure and style, thus making it more suitable for design and code reviews, as well as reducing debugging and maintenance time.

Additional information of embedded and real-time systems, including the many design issues that make it unique as compared to desktop application programming, can be found in [2] and [3]. The next section focuses specifically on the PBO MoC, which is a real-time component-based software technology that has the structure needed to enable VAOP. Subsequently, in Section 4, the many aspects that have been identified for developing RCES are listed, and preliminary research on two of those aspects are detailed.

3 Port-Based Object Model of Computation

Port-Based Object (PBO) Model of Computation (MoC) is a domain specific model of computation developed for real-time control systems [13]. The PBO MoC has proven to be a very useful abstraction for RCES, especially in control and communication based applications [12]. PBO MoC is an object-based methodology in which each component in a system is a PBO. An implementation of the PBO MoC is called a PBO Framework.

Each PBO is a real-time task that communicates only through its ports. A PBO gets data from other PBOs through its input ports, shares its results with other PBOs with its output ports, and interacts with the environment through its resource ports. This interface is depicted in Figure 1. Access to shared resources is handled by the PBO Framework, so no synchronization is necessary. PBOs execute periodically (or sporadically), and always use the most recent data available on their ports when they begin execution of a cycle.

A PBO system consists of a set of PBOs that communicate with each other and the environment. A PBO system can be represented as a graph. This graph is depicted in Figure 2, taken from [13]. A node in the graph is a PBO. Edges in the graph represent communication. An edge that connects one PBO port to another represents communication between PBOs. An edge with only one endpoint represents communication between sub-systems, or interaction with the environment through sensors and actuators. This type of design is similar to the architectural view of hardware description languages such as VHDL and verilog.
A PBO also interfaces with the PBO Framework, through a set of functions that a PBO must implement. This interface is illustrated in Figure 3. The functions that a PBO must implement in this PBO Framework are: `init`, `on`, `cycle`, `off`, and `term`. These functions are called by the PBO Framework according to the state of, and events in, the PBO system. This is very different from the usual method of developing RCES software with a Real-Time Operating System (RTOS), often with POSIX style process, threads and Inter-Process Communication (IPC). With an RTOS, the user must make system calls, handle signals, or define hooks into the RTOS. There are several problems with explicitly interfacing the RTOS. First, the programmer must know the interface to the RTOS which is often cryptic. Also, the code is not portable across RTOSs, and therefore likely not portable across platforms. Defining a standard interface is much better than inserting code to accomplish the same goal. The interface becomes implicit and the programmer is freed from knowing the particular RTOS interface.

The behavior of the PBO Framework is illustrated in Figure 4, and fully explained in [13]. The interface functions are called at specific stages in the PBOs lifecycle. `Init()` is called to create a PBO, after it is allocated, but before it is activated. `On()` is called to activate the PBO and ready it for execution. `Cycle()` is called periodically (or sporadically) while the PBO remains activated. `Off()` is called when the PBO is deactivated, but has not yet been deallocated. At this point it is possible to re-activate the PBO without re-allocating it. `Term()` is called to destroy a PBO, before it is decollated. PBOs have the ability to activate and deactivate so that the system can be dynamically reconfigured, changing the topology of the PBO system.

The PBO MoC was first implemented as a middleware extension in the Chimera RTOS [10]. It was used in several institutions, including: Carnegie Mellon University (where it was developed), NASA's Jet Propulsion Laboratory, Wright Patterson Air Force Base, and National Institute of Standards and Technology (NIST). Chimera was designed for 32-bit multi-processor real-time systems. Next, the PBO MoC was applied to RCES, using a multirate cyclic executive as described in [11], called Featherweight PBO (FPBO).
FPBO is a set of libraries that provide a PBO Framework run-time environment for the C programming language. FPBO provides real-time task scheduling, intertask communication via ports, and task synchronization. FPBO was designed as a self-contained run-time environment instead of an RTOS to reduce the overhead of the PBO Framework. Many RTOS features are not needed in RCES, and can not be supported due to resource constraints. For example virtual memory, separate address spaces, and dynamic linking and loading are not needed in RCES. In fact, many RCES do not even need dynamic memory allocation.

The PBO Framework can be customized to reduce overhead by only supporting the featured needed. Recently we have produced a Tiny PBO (TPBO) Framework. The TPBO Framework for the MSP430 uses only 156 bytes of ROM, and (12*nPBO + dPort) bytes of RAM, where nPBO is the number of PBOs in the system, and dPort is the total size of all port data. This demonstrates the scalability of the PBO MoC from large real-time systems down to small RCES.

While the PBO MoC has made great strides in introducing component-based design to RCES, there are still several limitations in the implementations to be addressed.

For example, PBO Frameworks have only been implemented in the C programming language. The C implementation of the PBO MoC is represented in the code that the user must write, hidden only slightly by macros that define functions and structures needed by the Framework. Therefore the user must understand tricky C code that implements the Framework in order to use it. Further, by using all of C as the programming language of the system, the PBO MoC cannot be enforced. For example, a pointer can be used to bypass the port-based communication, destroying the consistency of shared resources.

Another consequence of implementing the whole PBO Framework in C is that PBOs are not portable. While the code that implements the PBO is portable, to code that the PBO Framework needs is not. Each implementation of the PBO MoC (PBO, FPBO, TPBO) has different glue code and data structures needed for the Framework. While the PBO MoC allows for better modularity, it is a problem that the user must insert the appropriate Framework code manually. However, the Framework code is very systematic, and it is possible to generate it automatically.

Also, the same code describes all aspects of the system. Data types, control algorithms, PBO Framework, port communication, PBO configuration, and task synchronization are all mixed together. This make the code more complex and harder to comprehend. Code that is hard to comprehend is more likely to have bugs, and take longer to develop. Further, both are text based tools. The graph of PBO system represented in text is difficult to both design and comprehend, a visual representation of the PBO system is much better.

The solutions to these issues by using VAOP techniques are discussed next in Section 4.

4 VAOP with PBO MoC

Separation of concerns is our key goal for improving the development of RCES software. This is the concept that parts of a software system that are engineered separately should be seen separately. When a control engineer is implementing a control algorithm, he is not concerned with the task synchronization of the system. When a real-time systems engineer is scheduling tasks in a system he is not concerned with the details of a task implementing a control algorithm, only its period, deadline, and execution time. It is clear that the concerns of the control engineer and systems engineer are separate. Therefore, the way they design the system should also be separate. No current development environments do this. It can be achieved by separat-
ing the design into aspects. Creating real-time control applications with a visual programming environment can reduce days of programming to minutes [4].

The following lists several of the key aspects that we have identified as appropriate for RCES, based on our own design experience: PBO Detailed Design, PBO System Configuration, data types, power management, memory management, device driver, temporal, configuration, and error detection & handling. Some of these aspects are best implemented as visual programming languages, however others are inherently textual or tabular. Therefore, an environment that combines visual and textual programming techniques is needed.

We are building a prototype VAOP tool to evaluate the appropriateness of these aspects. The evaluation of an aspect involves answering the following questions.

- Does this aspect achieve a separation of concerns?
- Is this aspect language natural to the intended user?
- Does this aspect simplify the design of both this aspect, and the rest of the system?
- Is this aspect complete, can it express everything the user needs?

The prototype VAOP tool is written in Java to ensure portability. The environment resides in one application window, and aspects are accessed as internal frames in the environment. A screenshot of the design of an automobile cruise control system is shown in Figure 5. Initially, the prototype is being designed with just the Data Types, PBO Detailed Design, and PBO System Configuration aspects. This is the smallest set of aspects needed to completely design a system with the PBO MoC. The data types aspect is used to design data types, such as integer, floating point, enumerated, array, and record types. It can be expanded to describe abstract-
data types. Since it is not tied to the PBO MoC, we do not discuss it further. PBO Detailed Design is discussed in Section 4.1. PBO System Configuration is discussed in Section 4.2. The implementation of the aspect weaver is discussed in Section 4.3.

4.1 PBO Detailed Design

This aspect is used to build PBOs. The PBO Detailed Design specifies a PBOs interface to the rest of the PBO System (its communication ports illustrated in Figure 1) and its interface to the PBO Framework (implementation of the required functions illustrated in Figure 3).

A PBO port is described by the type of data it expects, whether that data is constant, the direction of data travel (input or output), and the scope of the port. Scope can either be instance when each PBO has a separate copy of the port, or object when the port is shared by all instances. A PBO can make use of data items other than ports, but they are not shared and so do not have a data direction.

A PBO may define more functions than those required by the PBO Framework. A PBO function is described by its prototype (interface to callers) and implementation (code that executes when it is called). The function prototype consists of a set of parameters that have data types and direction. The function implementation consists of textual code that could be written in many programming languages.

The implementation of the PBO Detailed Design aspect is shown in the top half of Figure 5. It consists of four separate frames. The upper left frame is used to add, modify, and delete ports of the PBO. The lower left frame is very similar and is used to add, modify, and delete data members of the PBO. The lower right frame defines a function prototype, and is used to add new functions, or modify existing function prototypes. The upper right frame is used to implement functions. The implementation of functions is done with the C programming language. This was done to allow the use of available C compilers for the many different embedded microcontrollers.

A preliminary evaluation of the PBO Detailed Design aspect with this implementation reveals several observations. The aspect appears to be complete, because all PBO functionality can be described by it. The aspect seems to simplify the design of PBOs by separating it from the design of the PBO System and the PBO Framework. The aspect also seems to simplify the design of the PBO System also by removing the unnecessary information of the PBO Detailed Design from the PBO System. However, parts of the implementation of the aspect are not natural for all users. It would be more natural for systems engineers to define a PBOs ports by adding icons for ports onto a representation of the PBO, similar to Figure 1. Ports on the left are defined to be inputs, while ports on the right are defined to be output. The remaining information can be entered into pop-up frames. Finally, this aspect does achieve a separation of concerns by separating PBO Detailed Design from the PBO System and Framework. A better separation of concerns may be achieved by further dividing this aspect into PBO Interface and PBO Implementation. The PBO Interface defines the PBO ports, while the PBO Implementation defines the behavior of the PBO.

4.2 PBO System Configuration

A PBO is an object, and as such can have multiple instances. A PBO instance consists of a PBO implementation and additional information needed by the PBO System - specifically creation time, activation criteria, real-time scheduling, and synchronization. Creation time determines when a PBO is allocated and initialized. Activation criteria determines when a PBO is allowed to be active. Scheduling information for periodic PBOs include: period, deadline, worst-case execution time. Scheduling information for sporadic tasks include: minimum inter-
arrival time, deadline, worst-case execution time, and event that triggers execution. Synchronization information is used to coordinate activities with multiple PBO instances.

A PBO System consists of a set of PBO instances, and the communication between them. The communication between PBO instances are connections from the port of one PBO to the port of another. There are a few restrictions on the communication allowed: (1) The PBO ports connected must be of the same or compatible data types, and (2) A connection may have only one source port, though it may have multiple destinations. Therefore, a ports fan-in is always one, while the fan-out may be zero or more.

A PBO system can be represented by a graph where nodes are PBO instances, and edges between PBO ports represent communication as described in Section 3. A PBO graph is best represented visually, as in Figure 2. This representation is both easy to comprehend, and design. The prototype implementation takes this approach, demonstrated in the bottom half of Figure 5. The implementation was developed using a Java package called diva [5]. It allows nodes (PBO instances) to be moved, added and deleted, and ports to be connected with edges in a way that is consistent with the restrictions discussed above. PBO instance information is accessed through pop-up frames for each PBO instance.

Preliminary evaluation of the PBO System Configuration aspect reveals several observations. The aspect appears to be complete, though the prototype implementation is not. The prototype uses some simplifying assumptions about PBO instances. First, all PBO instances are created when the system is created. Second, PBO instances are either initially active, or activated by other active PBO instances explicitly at run time. Finally, PBO instance synchronization is also accomplished explicitly in the PBO implementation. The aspect seems to simplify both the PBO System Configuration, and the PBO Detailed Design by separating them. The aspect is very natural to the systems engineer, the graph representation of the PBO System is very convenient. However, the aspect is less natural to the real-time systems engineer, who is only interested in the PBO instance information. This information is still scattered throughout the nodes. A more natural representation for the real-time systems engineer is tabular, such as a table of PBO instances. The PBO System Configuration aspect does achieve a separation of concerns, however further separation is needed. The PBO Instance Management should be an aspect separate from PBO System Configuration. This would allow multiple aspects for instance data so that the systems engineer and the real-time systems engineer can design the PBO System differently. It is okay to have multiple aspects for the same information, so long as they are kept consistent.

4.3 Aspect Weaver

An aspect weaver combines all of the aspect descriptions of a system into one representation. It does this by intermingling, or weaving, the code for these aspects, often producing what the user would have seen had the aspects not existed. The aspect weaver produces an executable of the PBO system that was designed with the aspects described in the preceding sections. It does this by generating the code for each PBO implementation, creating each PBO instance according to the PBO instance data, then synthesizing the PBO Framework code according to the system design. In this way the user never has to see the Framework code, and it solves the problem of porting PBOs across Frameworks. Implementing a new PBO Framework only requires writing a new weaver to synthesize the Frameworks code.

The aspect weaver implemented for the prototype VAOP produces a C implementation of the system. C code is higher level than machine or assembly code, and therefore easier to generate. C is portable and almost every microcontroller used in RCES has a C compiler. Most C compilers produce very efficient target code. Therefore generating C code enables a quality
prototype to be built very rapidly so evaluation could take place early in the design of the VAOP. In the interest of brevity the detail of the aspect weaver implementation are not discussed in this paper.

5 Summary

This paper offers a preview of our research on VAOP in the RCES domain. We use the PBO MoC, a domain specific abstraction enabling component-based design of RCES, as the foundation for our research. Due to resource constraints and a high degree of complexity, RCES are particularly difficult to develop. Therefore, RCES is a good candidate for new software engineering techniques such as VAOP. Our prototype environment demonstrates the applicability of VAOP techniques to the domain of RCES. However, more research is needed to determine the best implementation of VAOP for these systems. Future research in this area will follow two orthogonal courses.

First, while the implementation of the visual language is a working tool, it is a prototype and needs improvement. One improvement involves modifying the aspects implemented as described in Section 4.1 and Section 4.2. Further, the PBO Implementation should not use C as the aspect language. This is because C supports functionality that is not needed to implement a PBO, and has some features, such as void pointers, that compromise the integrity of the PBO Framework. Other improvements would involve the aspect weaver. First, the aspect weaver has information that could be used in optimization that it can not pass on to the C compiler. Therefore, code targeted to a specific instruction set architecture can be generated to allow more fine grained optimizations. The aspect weaver can also implement optimizations unique to RCES and the PBO MoC.

The second, while many issues with programming RCES have been resolved with the PBO MoC, many more aspects for RCES are possible. These aspects need to be researched, and the promising ones evaluated with a method similar to the one outlined in this paper. Candidate aspects are listed below.

- **Power Management:** This aspect addresses power management issues in RCES such as software controllable clock rates and low-power stand-by modes.
- **Temporal:** This aspect describes fine grain timing issues in RCES. Many I/O devices and communication protocols require strict timing when interfacing them, such as serial protocols like RS232, \(^2C, and CAN that have very strict timing requirements.
- **Configuration:** This aspect describes both the hardware and software configuration of the system. Hardware configuration includes: processor, I/O devices, and memory devices. Software configuration includes: scheduling algorithm, inter-task communication mechanism, and communication protocols. This enables both porting and scaling of the PBO MoC.
- **Memory Management:** This aspect not only manages memory dynamically, but also statically. Dynamic memory management deals with memory allocated by the system as it is running like `malloc/free` or `new/delete`. RCES often need to specify specific memory devices (RAM, ROM, Flash, EEPROM), and even exact memory addresses (memory mapped devices) for code and data. This requires static memory management.
- **Device Driver:** Our research on device drivers resulted in a new software engineering model and a mathematical foundation for engineering design and analysis of device drivers based on the PBO MoC [9]. Such a foundation forms the basis for a device driver aspect.
- **Error Detection & Handling:** Our research into error detection and handling has shown that it is possible to incorporate this as an orthogonal extension to the PBO MoC [13][6]. This aspect uses the techniques discovered to design error detection and handling software for RCES.
6 Acknowledgements

The research in this paper is supported in part by Embedded Research Solutions, LLC (http://www.embedded-zone.com), and in part by the Dept. of Electrical and Computer Engineering, University of Maryland. Special thanks to Christine Smit for her programming contributions to the prototype. Support for Smit was provided by the National Science Foundation Research Experiences for Undergraduates program, Award #9912218.

7 References