Attribute/Service Model: Design Patterns for Efficient Coordination of Distributed Sensors, Actuators and Tasks in Embedded Systems

Ying Zhang, Mark Yim, Craig Eldershaw, Kimon Roufas and Dave Duff

Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, California 94304
yzhang,yim,celdersh,kroufas,dduff@parc.com

Abstract

This paper proposes the Attribute/Service Model (ASM) and associated design patterns as a general and simple framework for applications that require programming with multiple tasks on multiple embedded processors. This model enables the programming of complex tasks with multiple sensors and actuators on highly distributed yet tightly coupled systems by: using a simple unified protocol for communication; allowing the access to attributes or the running of services to be independent of where such attributes or services reside; protecting shared resources, and simplifying the synchronization of multiple processes in multiple processors. Associated design patterns such as the event/trigger mechanism and general event-driven control are developed on ASM. ASM is designed for the coordination of distributed sensors, actuators and computational tasks on modular self-reconfigurable robots. However it may be used for any multi-threaded distributed embedded control network. Unlike the most existing distributed objects, ASM can be implemented on embedded systems with small footprints. ASM has been implemented both in C on top of VxWorks on the MPC555 embedded microprocessor, and in Java on PC. The Controller Area Network (CAN) has been used as the communication medium. It could be equally implemented on any real-time operating system using any communication media.

Keywords: *distributed embedded system model, networking architecture*

1. Motivation and Introduction

In general, real-time distributed control systems require synchronization and coordination of sensors, actuators and computation across multiple processors. In particular, **PolyBot** (see Figure 1) [8,9], a *modular reconfigurable* robot developed at Palo Alto Research Center, challenges the software design for massively *distributed*, largely *scalable*, deeply *embedded*, tightly *coupled*, and highly *responsive* control systems. Modular reconfigurable robotic systems are those systems that are composed modules that can be disconnected and reconnected in different arrangements. Each arrangement forms a new system with unique capability. In many cases, the number of modules is much larger than the types of modules within such systems, i.e., the systems tend to be more homogenous than heterogeneous. The general philosophy underlying these systems is to simplify the design and construction of components while enhancing functionality and versatility through larger numbers of modules.

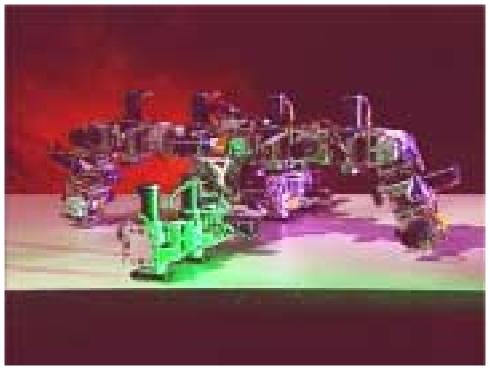


Figure 1. PolyBot modules in a spider configuration

PolyBot is distributed and scalable since it will consist of 10s to 100s of connected modules; it is embedded since each module has an embedded microprocessor with multiple *local* I/O channels for sensing and actuation, as well as *remote* channels for inter-module communication; it is coupled since controls in modules have to be *synchronized* for most of the tasks; it is responsive, requiring *multi-threading* and *real-time event handling*. **PolyBot** has demonstrated its flexible capabilities in locomotion and manipulation (see videos in http://www.parc.com/modrobots). However, programming its various tasks effectively remains a challenging problem.

Real-time operating systems (RTOS) provide a set of general APIs and mechanisms, e.g., task scheduling, interrupt handling, and semaphores. However they vary considerably with different RTOSs and communication media, making programming and porting costly. Various higher-level programming models have been developed for such distributed control and coordination. However, most of these are impractically large or slow for use in embedded environments.

This paper proposes the *Attribute/Service Model* (ASM) as a general and simple framework for applications that require programming with multiple threads on multiple processors. *Attributes* are abstractions for resources shared among multiple threads located in one or more processors. *Services* are abstractions of hardware or software routines. In general, hardware services correspond to settings in registers controlling hardware peripherals, and software services are threads that may be run for particular tasks.

ASM extracts some widely used features in distributed multi-threaded applications: task synchronization, resource protection, and transparent remote accessing. The presence of these features makes it a valuable *design pattern*. A design pattern [3] names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.

ASM supports a *component-based* architecture, where components are either attributes or services distributed over the communication network. Component-based software architectures have been highly promoted in the software engineering community [7]. Much work has been done on Real-time CORBA[2,6] and there are several Java packages for the coordination of services in distributed environments such as Sun's JavaSpaces [12] and IBM's TSpaces [13]. ASM borrows some of the ideas from these architectures. However, few of these implementations are suited for embedded systems with small footprints. ASM is more lightweight, focused more on coordination among sensors and actuators in multi-tasking and multi-processor environments.

ASM also serves as *middleware* that resides between the RTOS and the application software. This enables it to provide certain basic features independent of the RTOS and communication media. ASM is not application specific; rather it applies to the general domains of distributed coordination of sensors, actuators and tasks. Also ASM is not implementation specific; it can be implemented on most RTOSs and communication media.

ASM has been implemented both in C on top of VxWorks on the MPC555 embedded microprocessor, and in Java on PC, using Controller Area Network (CAN) as the communication medium. It could be equally implemented on any real-time operating system using any communication media.

This paper is organized as follows: Section 2 explains the attribute interface and service interface. Section 3 presents the ASM communication pattern: its client/server structure and communication protocol. Section 4 discusses some extended design patterns of ASM commonly used in control systems. Section 5 concludes the paper.

2. ASM Interface Pattern

There are two common elements in any computational system: *computation resources* and *computation routines*. For distributed and/or concurrent systems, resources are shared by many routines in one or more processors across a network. ASM takes this view to an extreme: communication between two routines resided anywhere in the network is by setting and getting values from a shared resource. Therefore, ASM is essentially the *shared variable model* in distributed computation. The shared variable model has the advantage over the message passing model, in that no explicit messages need to be defined at the application level, all the communication are performed transparently. ASM has two types of components: attributes and services. Attributes represent computation resources and services represent computation routines.

2.1 Attribute Interface

Attributes are abstractions of shared resources. An example of an attribute is a desired value of some device, e.g., temperature, which may be set by a high level task, such as a temperature profile generator. The low level linear controller then uses this value to drive the system to the desired state. Another example of an attribute is storage, where producers are putting products, and the consumers are getting products.

Each attribute is associated with set(), get() and reset() methods. These methods are executed in the same thread of control as the calling routine. Structures built into the attributes protect the shared resources as well as supporting synchronization between multiple services. In particular, each attribute is associated with a block of data that has multi-thread protection, i.e., at any given time, only one thread can access the data through the set(), get() or reset() methods that are provided for all attributes. This ensures the integrity of the shared data. In some sense, the attribute interface pattern is an instantiation of the "Monitor Object" pattern [1].

There is a Boolean variable valid indicating the validity of the data. Every time a set() operation is performed, valid is set to be **true** and every time a get() operation is performed, valid is set to be **false**. There are two synchronization flags associated with the block of data, syncGetFlag and syncGetFlag. When syncGetFlag is set, the routine getting data using get() will block until the data is set by another routine. When syncSetFlag is set, the routine setting data using set() will block until the data is used by another routine before setting a new value. The combination of these two produces four possible ways for a particular attribute to synchronize with services that access the data. For example, a routine that tracks desired settings will block until a new desired setting comes in; a routine that generates commands will block until the previous command has completed execution.

A typical implementation for the attribute set() and get() function looks like:

```
set(data) {
   mutex_lock();        //protection starts
   while (syncSetFlag and valid) wait();

   copy_in(data);
   valid = true;

   if (syncGetFlag) signal();

   mutex_unlock();    //protection ends
}

get(data) {
   mutex_lock();      //protection starts
   while (syncGetFlag and not valid) wait();

   copy_out(data);
   valid = false;

   if (syncSetFlag) signal();

   mutex_unlock();   //protection ends
}
```

where mutex_lock() protects the data from concurrent accessing, wait() is a conditional block and signal() is a corresponding notification method.

2.2 Service Interface

Services are abstractions of hardware or software routines. All services have associated methods start(), stop(), suspend(), continue() and reset() methods. Services can be realized in software, firmware or hardware. In general, a firmware or hardware service corresponds to setting some registers which control system devices; while a software service is a thread that is programmed to perform a particular task or behavior.

All services are considered to have their own thread of control with start() and stop() commands, even though some services may only be a couple of lines of code. A routine calling start() for a service will not block; many services can run concurrently. In some sense, the service interface pattern is related to the "Active Object" pattern [4], but simpler. A service can also be associated with a set of parameters that shall be initialized when the service is started. For example, if the service is a state-based system, the initial state of the service can be passed with the start() function. Since a service represents a thread of control, then it must be stopped before it is started again.

A service can also be *suspended*, and then be *continued* again. The difference between start/stop and suspend/continue is that start()/stop() would create/terminate a new software or hardware process, while suspend()/continue() would only pause/continue to run the same process. There are two Boolean flags indicating the state of a service: started meaning the service has been started and running meaning the service is not suspended. However the service could not be running until it is started. The reset() method may only be applied to a service that is not running.

An example of a hardware service might be as simple as turning on a power switch, or starting a signal generation; a software service might be a linear controller tracking a desired setting, or a nonlinear optimization routine solving a set of constraints over time.

3. ASM Communication Pattern

For a networked embedded system, it is a challenging problem to coordinate and synchronize services in different processors. ASM simplifies this at the application level by making the location of where attributes are stored or where services are run transparent to the user and by using a unified protocol for communication. All attributes and services are accessible both *locally* and *remotely* where remote is defined as accesses requiring inter-processor communication. Local and remote objects are created differently but have the same interface described in the previous section. The creation of a local object involves a local registration, which assigns the object with a class ID and an object ID. The creation of a remote object involves a remote lookup operation, which also associates the object with the class ID and the object ID of the object it refers to. A local object is called a *server object*, which runs a service or stores the attribute data, and a remote object is called a *client object*, which refers to a service or an attribute in a remote location. Every client object is created with a *client port*, which is obtained when a successful connection is established. A port can be either one-to-one or one-to-many communication. This port becomes a handle for communication with the remote processor(s). More than one client object can share the same client port if they are all running in the same thread of operation.

In addition to using attribute get() to obtain remote values, the *Publish/Subscribe* pattern is also integrated into ASM. In Publish/Subscribe, a local attribute can subscribe to another local attribute of the same class located in a remote processor in the network. Whenever a local attribute publishes its current data, all the subscribers of that attribute will receive it. For example, a monitoring service in one processor may hold a list of readings, each of which subscribes to the current reading of some sensor in the network. Whenever a reading is published, the corresponding reading in the monitoring service will be updated. The rate of the publication is set by the server, not the client. Some common patterns are: publish at a fixed rate, publish when a value changes significantly, or the combination of both. Note that functions subscribe()/publish() apply to local attributes only. The Publish/Subscribe mechanism is more efficient than the remote attribute get() when the client and the server need to be synchronized frequently and changes in the server are unable to be predicted by the client.

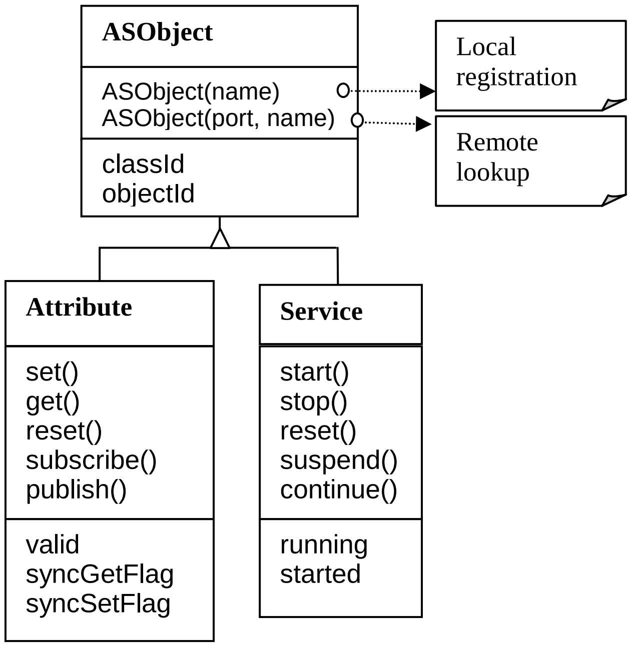


Figure 2. Class diagram of ASM

Figure 2 shows the class diagram of ASM. The notation used in the class diagrams in this paper is adopted from the design pattern book [3]. A class is depicted by a box with the class name bolded, followed by operations and variables of the class. A dashed arrow starting with an empty circle indicates the implementation of the operation. A solid line connection with an empty triangle indicates the subclass relation.

Every processor supporting ASM has a *gateway daemon* whose job is to accept connection requests from other processors and then spawn a new *server daemon* for each such connection. Each ASM server daemon is responsible for dispatching and invoking the correct actions for remote attributes and services associated with its connection (Figure 3). A client object calling an interface function will generate a request message through its *proxy* and send the message through the port to the remote server. The remote server will execute the request and send the result or status back to the client through the same port.

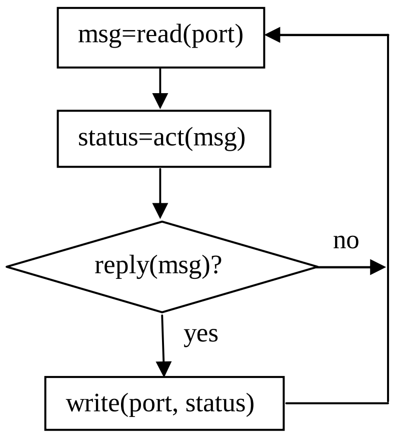


Figure 3: Flowchart of ASM server daemon

The underlying communication protocol for remote access of attributes and services is simple and uniform. A message header contains method ID, class ID and object ID; the reminder of the message is the actual data. If the method ID indicates a request of finding the class ID or object ID by name, the data field would be the string representing the class or object name. The server shall search the lookup table to find the corresponding attribute or service according to the IDs or the names. A flag in the method ID indicates whether the client needs a reply or not. For methods finding IDs or getting attribute values, the reply flag would be set by the system, otherwise the client can set the flag to indicate whether a reply is needed or not. If a return message is demanded, the client may choose to block while waiting for the message. For example, if a client wants to make sure a remote service has been started as requested; it should choose to wait for the reply from the remote server. The functions of client proxies and server daemons make local and remote objects transparent at the application level.

ASM has been implemented both on the MPC555 microprocessor and on PC, using Controller Area Network (CAN) as the communication medium. CAN has proven that it fits very well into the suite of sensor/actuator buses because of its low price, multiple sources, highly robust performance and widespread acceptance [5]. However the CAN protocol is a low-level one, with a maximum of eight bytes of data per frame. A higher-level protocol developed on top of CAN, named MDCN (Massively Distributed Control Net) [10,11], has been used. MDCN handles messages of large sizes, supports three types of communication (individual, group and broadcast), has eight priority levels, and can address up to 255 nodes. MDCN has been implemented both in C on the RTOS VxWorks and in Java for PC. ASM is then implemented on the top of MDCN at both ends. The implementation is such that Java and C objects can be used interchangeably, i.e., a remote Java ASM object can refer to a C ASM object and vice versa.

4. Extensions and Examples of ASM

Both attributes and services in ASM can be extended with data types and operations. This section presents three fundamental extensions that are useful in distributed control.

4.1 Event/Trigger Pattern

For embedded systems with sensors, actuators and tasks, one important type of coordination is to react to the changes of system or in environmental conditions. The event/trigger pattern is made for that purpose. A *trigger* is associated with a particular type of condition change. A trigger shall be fired whenever that condition change occurs. The change can be either a logical or a physical signal, such as timer expiration or sensor value changes. An *event* can be associated with a set of triggers; whenever one of the triggers associated with an event is fired, the event is activated.

Figure 4 displays the class diagram of the event/trigger pattern, in which the dot with a solid triangle indicates possibly more than one reference. In this pattern, an event is a subclass of an attribute, with the syncGetFlag set to be **true** and the syncSetFlag set to be **false**. An event can wait for a trigger with waiting(). A process calling the event waiting function will be blocked until the event is triggered. A trigger can be operated as an interrupt service routine, where the hardware interrupt can cause the firing of a trigger, or as a software thread that continuously polls the state of the device and fires the trigger whenever the trigger condition is satisfied.

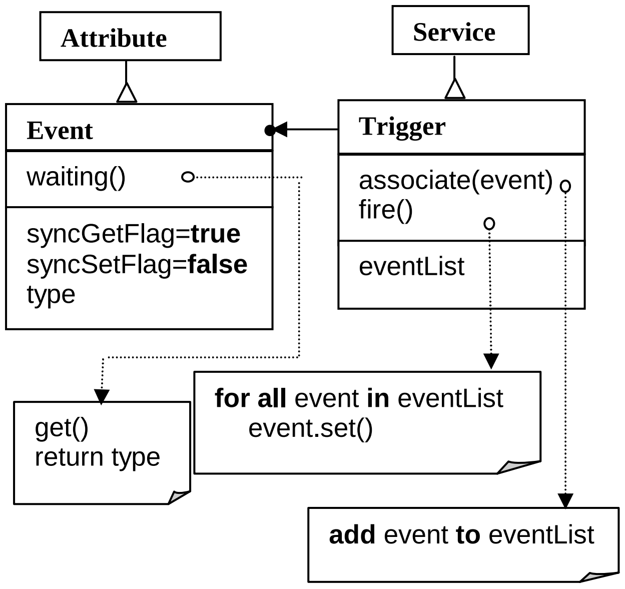
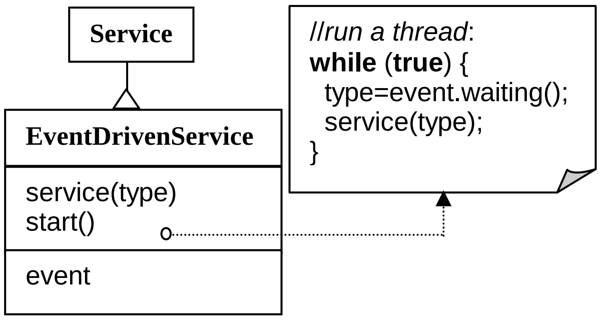


Figure 4: Class diagram of event/trigger pattern

4.2 Event-Driven Service Pattern

In many situations, software services are not running continuously, but rather are triggered by events. For example, a service can be triggered periodically by timers or by sensor reading passed the threshold. The event-driven service pattern is made for this purpose. An event-driven service is a subclass of a service that is associated with an event. The operation of the service corresponds to a thread running on a periodic software task, triggered by the event. Figure 5 shows the class diagram of the event-driven service pattern. A service can be synchronized with a local or remote event by waiting for that event using function waiting(). Whenever a trigger is fired, any service waiting for the event associated with that trigger is activated. The exactly what action follows varies with the trigger type. For example, if the event is triggered by a timer, then a regular service may be called; if it is triggered by an exception, then an exception handler may be called.



```
//run a thread:
while (true) {
  type=event.waiting();
  service(type);
}
```

Figure 5: Class diagram of event-driven service

4.3 Phase Automata Pattern

Most discrete control mechanisms can be represented by state machines. *Phase automata* are generally event-driven discrete state machines with periodical behaviors. The phase of each indicates that machine's particular starting point in the automata in a continuous time domain. Phase automata are efficient representations of hybrid systems with both higher-level discrete event-driven and lower-level continuous characteristics. A phase automaton extends an event-driven service with: a state, a direction, an initialization routine, and an event handler as a service function. The event handler in general consists of two parts: an action function and a next state function. Figure 6 shows the class diagram of the phase automaton.

The initialization routine will be executed at the beginning of start(), which is inherited from the event-driven service. The input argument to the initialization routine indicates the initial delay phase of the automaton. The initialization routine is responsible for setting the initial state and the initial actions.

In addition to having a persistent state like all automata, phase automata have direction variables; so that phase automata can run forward or backward.

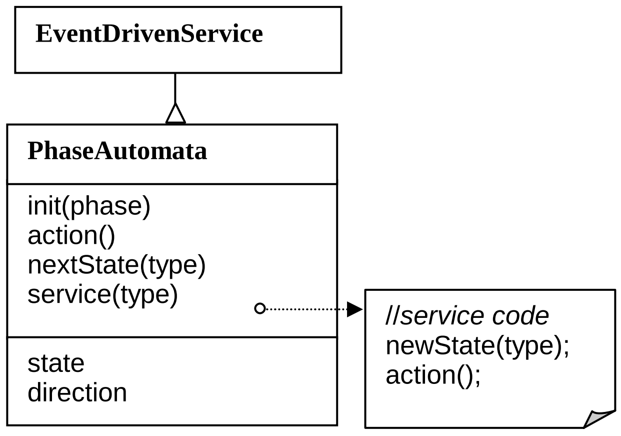


Figure 6: Class diagram of phase automata

Phase automata provide a general framework for controlling coordinated behaviors, such as locomotion gaits [9]. They can represent time driven or sensor driven, periodic or non-periodic, local or global, and hierarchical behaviors.

5. Conclusions

This paper has presented the Attribute/Service Model (ASM) and related design patterns for coordination of multiple sensors, actuators and tasks in a networked embedded control system. In addition to supporting useful design patterns, ASM provides the structural bricks for component-based architectures and serves as a middleware residing between real time operating systems and applications. ASM extracts basic features that are widely used for multi-threaded coordination. It enables the efficient software design for massively distributed, largely scalable, deeply embedded, tightly coupled, and highly responsive control systems by: using a simple unified protocol for communication; allowing the access to attributes or the running of services to be independent of where such attributes or services reside; protecting shared resources, and simplifying the synchronization of multiple processes in multiple processors. Design patterns derived from the basic model, such as Event/Trigger, Event-Driven Services and Phase Automata demonstrate the generality of ASM. ASM can be implemented in embedded systems efficiently with small footprints.

Acknowledgements:

This work is funded in part by the Defense Advanced Research Project Agency (DARPA) contract # MDA972-98-C-0009. Thanks to Markus Fromherz for valuable comments and suggestions.

References

[1] D.C. Schmidt, “Monitor Object: an Object Behavior Pattern for Concurrent Programming”, *C++ Report*, vol. 12, Mar. 2000.

[2] D.C. Schmidt, D.L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Brokers”, *Computer Communications*, vol. 21, pp. 294-324, Apr. 1998.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Weslеy Professional Computing Series, 1995.

[4] R.G. Lavender and D.C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming”, *Proc. 2nd Annual Conference on Pattern Languages and Programs*, pp. 1-7, September 1995.

[5] W. Lawrenz, CAN System Engineering: From Theory to Practical Applications, Springer, 1997.

[6] Object Management Group, Real-time CORBA Joint Revised Submission, OMG Document orbos/99-02-12 ed., March 1999.

[7] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addision-Wesley Publishing Company, 1997.

[8] M. Yim, D. Duff, K. Roufas, “PolyBot: a Modular Reconfigurable Robot” *Proc. of the IEEE Int. Conf. on Robotics and Automation*, April 2000.

[9] M. Yim, Y. Zhang, D. Duff, “Modular Robots”, *IEEE Spectrum*, Feb. 2002.

[10] Y. Zhang, K. Roufas, M. Yim, “Software Architecture for Modular Self-Reconfigurable Robots”, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Hawaii, 2001.

[11] Y. Zhang, K. Roufas, M. Yim, “Massively Distributed Control Nets for Modular Self-Reconfigurable Robots”, *AAAI Spring Symposium for Intelligent Distributed and Embedded Systems*, 2002.

[12] Sun’s JavaSpace: http://java.sun.com/products/javaspaces/

[13] IBM’s TSpace: http://www.icc3.com/ec/tspace/