Modeling Software Systems by Domains

Richard D'Ippolito and Kenneth Lee
Software Engineering Institute
Carnegie Mellon University

The Software Architectures Engineering (SAE) Project at the Software Engineering Institute (SEI) has developed engineering modeling techniques that both reduce the complexity of software for domain-specific computer systems and result in systems that are easier to build and maintain. These techniques allow maximum freedom for system developers to apply their domain expertise to software.

We have applied these techniques to several types of applications, including training simulators operating in real time, engineering simulators operating in non-real time, and real-time embedded computer systems. Our modeling techniques result in software that mirrors both the complexity of the application and the domain knowledge requirements. We submit that the proper measure of software complexity reflects neither the number of software component units nor the code count, but the locus of and amount of domain knowledge. As a result of using these techniques, domain knowledge is isolated by fields of engineering expertise and removed from the concern of the software engineer. In this paper, we will describe kinds of domain expertise, describe engineering by domains, and provide relevant examples of software developed for simulator applications using the techniques.

Separation of Concerns by Domain Expertise

We classify computer system developers by expertise and role using three categories: systems analyst, domain engineer, and software engineer. Systems analysts are responsible for defining the policy, strategy, and use of the application to be developed, e.g., the concept of operations, and the training requirements. Domain engineers are the modelers responsible for determining which real-world entities need to be modeled to satisfy the policy, strategy, and use defined by the systems analysts. Domain engineers determine if and how the entities selected to be modeled can be specified within the constraints imposed by the software engineers. Finally, they express the models in the language natural to their domain. Software engineers are responsible for defining a consistent software structure into which the domain expertise will go, and providing translations from the domain-specific natural languages into executable software.

It is not generally possible to reduce the amount of domain knowledge required to either develop or enhance a software-dependent system. To borrow a phrase from Albert Einstein, our system models should be as simple as necessary, but no simpler. If we can separate the design of the models from the design of the software, we can separate the tasks of the domain engineer from the tasks of the software engineer. This would allow the software engineer to make simplifications in the software packaging and execution structures which would not affect the way the domain engineer expresses the models. It would also allow the domain engineer the freedom to design model algorithms without requiring specialized software knowledge. In effect, each engineer is relieved of the burden of becoming an expert in other domains of expertise.

We have found that this separation of concerns by domain expertise is what enables us to simplify the overall design process and gain a more enhanceable (maintainable) computer system.

Engineering by Domain

In our vocabulary, a *domain* is a specific field of engineering expertise. Engineering expertise is classified by families of models and related sets of practices for applying the models, not by the problems to which the expertise is applied. Common classifications of engineering domains are: electrical, civil, nuclear, mechanical, chemical, and (the as yet undefined field of) software engineering. An *application area* consists of related problems that can be described using models from a variety of domains. Examples of application areas are command and control systems, factory automation

This work is sponsored by the U.S. Department of Defense. The SAE project members are Richard D'Ippolito, Kenneth Lee, Charles Plinta, and Jeffrey Stewart.

systems, embedded systems, and simulator systems[1]. Thus, a flight simulator application requires domain expertise in aeronautical engineering, electrical engineering, mechanical engineering, and so on.

Models are reusable, adaptable, engineering assets because they are patterns expressed in their most general form and are scalable, usually through *templates*. A good example of a templated model is a dress pattern, where all of the cut-lines are given by dress size.

We classify models using two major types, which we call *product models* and *practice models*[2]. The product model, when scaled, results in a component of the delivered product. The dress pattern is an example of a product model, as is the set of engineering drawings for an I-beam or a DC motor. Clearly, the dress pattern is no good without the practice know-how of fabric and thread selection, cutting, stitching, hemming, pleating, and all of the other activities needed to produce the final product. As a commercial venture, dress-making would require in addition to the product models the assembly-line models, materials-handling models, business and economic models, and so on. All of these models are what we call the practice models, because they define the established body of practice around the product models. Interestingly, the more mature an engineering discipline, the more the product and practice models will be public. In a mature discipline, the business enterprise seeks value added through system composition (model application), not model creation or refinement, which are seen as adjunct activities to be undertaken only when necessary to complete an application.

In the construction industry (civil engineering and architecture), for example, all engineering firms have access to the same materials, material costs, implementation practice (labor), and labor costs. In these cases, the firms compete on system composition, where success is meeting the customer's needs with a timely and economical design. Electrical engineers do not manufacture their own wire, integrated circuits, resistors, and other electrical and mechanical components, but compete on the basis of using these components efficiently to satisfy a need. The information on the components themselves is found in *engineering databooks* (usually manufacturer's publications), and *engineering handbooks* which are compendia of the practice knowledge. Both require an experienced practitioner with an in-depth education to interpret, however, as one cannot learn and practice an engineering discipline solely from the handbooks. With that education, however, the use of the handbooks will go a long way toward guaranteeing a successful routine (precedented) design. The use of the handbooks are not intended to support innovative design.

SAE has been very successful in applying models across various software application areas because our models have captured patterns of structure and behavior at the domain level. The *Object-Connection-Update (OCU)* model[3] is a good example of a building block that allows the domain engineer to capture the patterns of structure and behavior of the real-world subsystems being modeled[4]. Originally created for flight simulators, the OCU was immediately applied to the design of the seeker subsystem of an anti-tank missile and is now being used in the design of subsystems for engineering simulators. What made these applications of the model possible was the capturing of the basic pattern of subsystem operation into a few standardized architectural elements[5] (models), each responsible for a particular subsystem task. Complexity is reduced because any subsystem can (and must) be expressed using only these basic elements, thus constraining the choice of solution structures available for consideration. Systems analysts, domain engineers, and software engineers

[1] As an example, consider the domain of a rope where force is transmitted through tension in a flexible member (try using a rope under compression to push an object). Mechanical engineers have no problem applying the same rope design models, i.e., the domain expertise, to suspension bridges, elevators, cranes, and fishing rods, yet the application areas will seem quite unrelated to those not proficient in the domain.

[2] We have deliberately avoided the overloaded term *process*, preferring to reserve it for its traditional engineering reference to a controlled activity within a plant or machine. We use *practice* to refer to those engineering activities that support product development.

[3] The seminal report on the OCU is CMU/SEI-88-TR-30, *An OOD Paradigm for Flight Simulators, 2nd Edition*. This report, however, is dated relative to current SAE experience and is being updated. We are, also, in the process of writing a series of white papers that will fully describe the OCU and the engineering of software-dependent systems.

[4] In our terms, the total application is composed of *subsystems* so that those who wish may apply the term *system* to the whole.

are able to make use of the OCU as the basis for their separation of concerns; the OCU is the framework that ensures all activities will work together.

OCU Subsystem Examples

The OCU, produced by the software engineers, guides the systems analysts and domain engineers by providing the fundamental pattern of analysis and the structure for model capture. The systems analysts, with the foreknowledge that the ultimate software implementation will be subsystems captured by the OCU, will be guided to view the application as a collection of subsystems. The domain engineers, with the same foreknowledge, will be guided to compose models as collections of subsystems, each composed of objects organized by a controller. We will show in the following examples, taken from a simulator application, how the OCU provides this guidance.

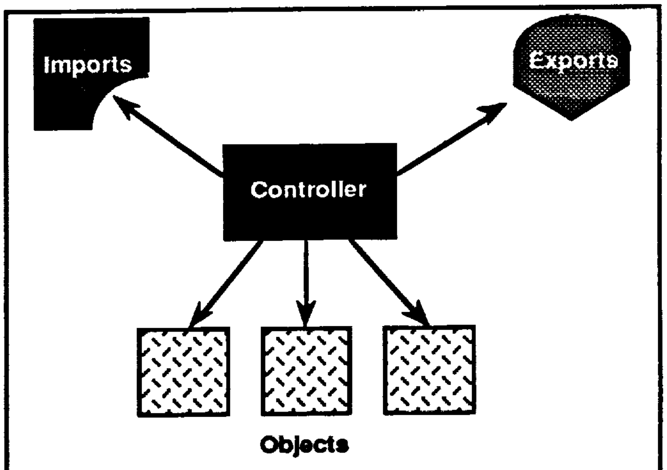
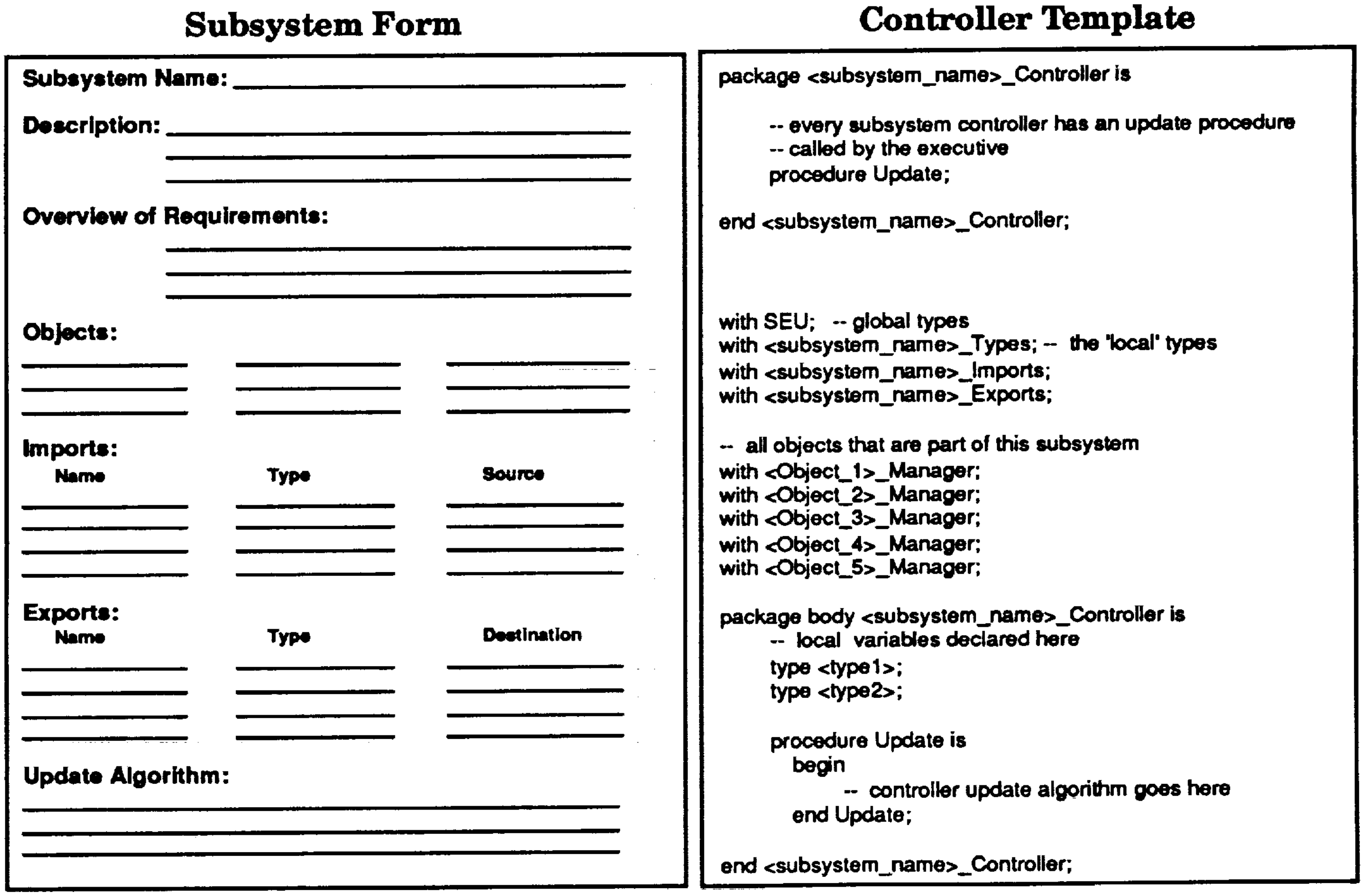


Figure 1: OCU Subsystem Diagram

Before we describe how the OCU provides this guidance, we will provide more detail about the OCU



Subsystem Form

Subsystem Name:

Description:

Overview of Requirements:

Objects:

Imports:
Name Type Source

Exports:
Name Type Destination

Update Algorithm:

Controller Template

```
package <subsystem_name>_Controller is

        -- every subsystem controller has an update procedure
        -- called by the executive
        procedure Update;

end <subsystem_name>_Controller;



with SEU;  -- global types
with <subsystem_name>_Types; --  the 'local' types
with <subsystem_name>_Imports;
with <subsystem_name>_Exports;

--  all objects that are part of this subsystem
with <Object_1>_Manager;
with <Object_2>_Manager;
with <Object_3>_Manager;
with <Object_4>_Manager;
with <Object_5>_Manager;

package body <subsystem_name>_Controller is
        --  local  variables declared here
        type <type1>;
        type <type2>;

        procedure Update is
          begin
               -- controller update algorithm goes here
          end Update;

end <subsystem_name>_Controller;
```

Figure 2: Subsystem Specification Form and Controller Template

[5] The basic elements are controllers, objects, import areas, export areas, surrogates, and device handlers. Controllers are the loci of subsystem connection and operation information; objects provide the subsystem services; import areas provide the subsystem with a view to the external world; export areas provide a window into the subsystem state for the external world; surrogates translate information from external formats to internal formats and back; and device handlers handle external-world communications. All instances of each of these elements are of the same form (implementation structure).

itself. We have found that the general patterns of operation of subsystems in any domain can be captured in a universal structure. These patterns involve separation of mission from operation, localization of state, activation and control of subsystems, and transfer of information. Separation of mission from operation is derived from a principle that is fundamental to all human and machine behavior: the mechanism of making decisions should be separate from the mechanisms used to carry out the decisions. Localization of state is derived from the fundamental software engineering principle of information hiding. In the OCU (Figure 1), the controller is the locus of decision making, and the objects provide the service mechanisms and the localization of state.

We knew that we could reduce the software complexity by repeated use of a small number of elements, a standard method of information transfer, and a standard method of control. We also knew that a maintainable system required closely related services be isolated from other, unrelated, services. In software engineering terms, this means coupling between unrelated entities is minimized, cohesion between related entities is maximized, and maintainability is enhanced by repeated use of the same patterns. In the OCU, isolation and information transfer is provided by the import and export areas. Cohesion among the objects in a subsystem is enforced by having the controller be the sole entity that implements connections to objects. We have found this set of elements: objects, controllers, export areas, and import areas, to be sufficient for describing any real-world subsystem.

We, as software engineers, have implemented the elements of the OCU in Ada. We have captured the patterns with a subsystem specification form and a set of element code templates.

The OCU is applied with the aid of the subsystem specification form and the element code templates, subsets of which are shown in Figure 2 (only the controller template is shown). The subsystem form provides a standard way for the systems analyst and domain engineers to record the specifications of subsystems in terms of the known compositional elements of subsystems, as shown in Figure 3. The subsystem templates provide a standard way for the

Sonar Subsystem Form

Subsystem Name: *Sonar*

Description:
The sonar subsystem is used to locate mine-like objects. Its transmit power level and pulse repetition rate are controlled by the console operator. The received signals are sent to the console.

Overview of Requirements:

References:	SW5TO-EO-MMO-020 pp. 3-5 pp. 7- all FO-8 FO-12 Telemetry Data Format MNV-Engineering Worksheet Schematic Slide

Objects:

Sonar Soundhead
Sonar Tilt Potentiometer
Flow Control Servo_Valve
Rotary Actuator

Imports:

Name	Type	Source
Rate_Cmd	Volts	Electronics Unit
Xmit_Level_Cmd	Xmit_Level	Electronics Unit
Slew_Rate_Limit_Cmd	Slew_Rate_Limit	Electronics Unit
Range_Reset_Cmd	Range_Reset	Electronics Unit
Sonar_Received_Signal	Sonar_Signal	Environment
Pulse_Repetition_Rate_Cmd	Pulse_Repetition_Rate	Electronics Unit
Hydraulic_Pressure_Available	Hydraulic_Pressure	Hydraulic System

Exports:

Name	Type	Destination
Sonar_Tilt_Potentiometer_Voltage	Volts	Electronics Unit
Composite Video	Sonar_Video_Signal	Electronics Unit
Sonar Transmitted Signal	Sonar_Signal	Environment

Sonar Controller Code

```
package Sonar_Controller is

     -- every subsystem controller has an update procedure
     -- called by the executive
     procedure Update;

end  Sonar_Controller;


with SEU;  -- global types
with Sonar_Types; --  the 'local' types
with Sonar_Imports;
with Sonar_Exports;

-- all objects that are part of this subsystem
with Flow_Control_Servo_Valve_Manager;
with Rotary_Actuator_ Manager;
with Sonar_Soundhead_ Manager;
with Sonar_Tilt_Potentiometer_ Manager;

package body Sonar_Controller is

    procedure Update is
     begin
       Flow_Control_Servo_Valve_Manager.Update(
              Sonar_Imports.Rate_Command,
              Sonar_Imports.Hydraulic_Pressure_Available,
              Sonar_Exports.Controlled_Pressure);

       Rotary_Actuator_Manger.Update(
              Sonar_Exports.Controlled_Pressure,
              Sonar_Exports.Controlled_Torque);
```

Figure 3: Completed Subsystem Specification Form and Controller Template (truncated)

software engineer to map the design of the models, captured on the forms, directly into an Ada implementation of the elements, also shown in Figure 3.

We can now describe how the OCU provides guidance to systems analysts and domain engineers. The systems analyst, in consultation with the customers and users, analyzes the application to identify subsystems consistent with the concept of operation and patterns of use of the application. Each of the subsystems is assigned a specification form and passed to the appropriate domain engineer for completion. In addition to the identification of subsystems, the systems analyst will provide the domain engineer with a mapping of the training requirements expressed in terms of model fidelity, operational modes, and malfunctions.

Figure 4 shows a sonar subsystem schematic from a Navy remote-controlled, minehunting, undersea vehicle. This diagram was constructed by sonar engineers and represents the real-world sonar subsystem. The schematic captures the knowledge needed by the domain engineer to model how the sonar subsystem is constructed. For the construction of a complete simulator, the systems analyst will gather representative schematics and provide them, with the specification forms, to domain engineers.

A domain engineer receives a partially completed form and some subsystem schematics from the systems analyst. The domain engineer then models the real-world subsystem to match the fidelity requirements expressed on the form. Each element of the model is mapped to an element of the OCU, the element models are parameterized to realize the specified operational modes and malfunctions, and the parameterized models are captured in a language natural to the domain engineer. The domain engineer completes the specification form by recording the mapping and forwarding the form, containing the natural language description of the parameterized models, to the software engineer.

Figure 5 shows a representation of the sonar subsystem as modeled by the domain engineer. The objects remaining are those sufficient to simulate the subsystem to match the fidelity requirements, modes, and malfunctions. Some connections to other

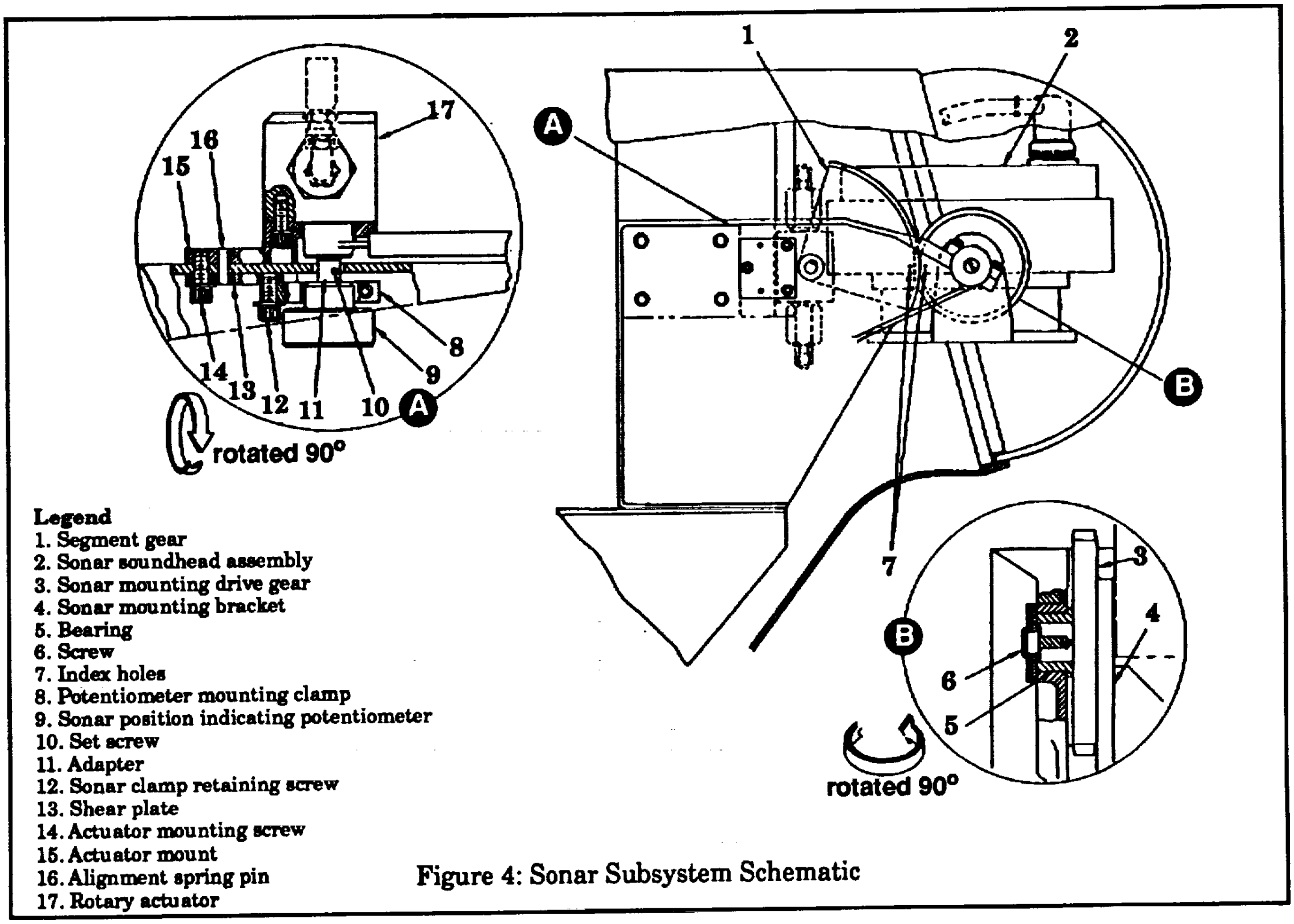

1
2
A
17
16
15
8
9
14
13
12
11
10
A
rotated 90°
B
7
3
4
B
6
5
rotated 90°
Legend
1. Segment gear
2. Sonar soundhead assembly
3. Sonar mounting drive gear
4. Sonar mounting bracket
5. Bearing
6. Screw
7. Index holes
8. Potentiometer mounting clamp
9. Sonar position indicating potentiometer
10. Set screw
11. Adapter
12. Sonar clamp retaining screw
13. Shear plate
14. Actuator mounting screw
15. Actuator mount
16. Alignment spring pin
17. Rotary actuator

Figure 4: Sonar Subsystem Schematic

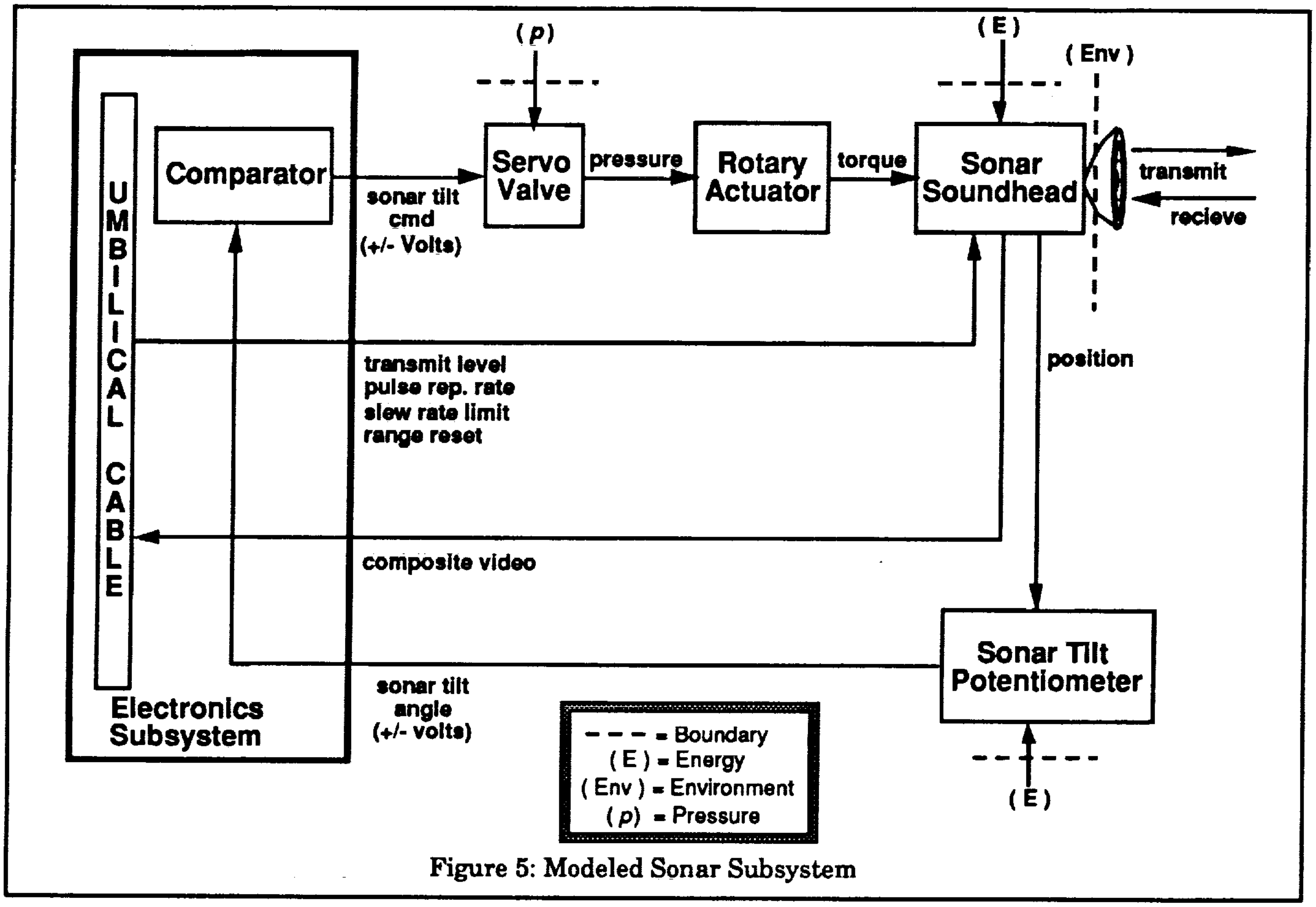


Figure 5: Modeled Sonar Subsystem

subsystems on the undersea vehicle are shown as well. Figure 6 shows an OCU diagram for the modeled sonar subsystem.

Conclusions

Using a fixed set of templates means that the interface mechanism between elements is known ahead of and independent of model design. All subsystems look (structurally) alike, and each subsystem can be made to lie within a single domain, with communication between subsystems also being handled by common structures. This means that the software engineer can proceed with executive and test harness design. It also means that the model specifiers can work independently in their own domains, knowing that their models will fit into the completed system.

Thus, a completed simulator application will consist of as many instances of the OCU subsystem model as required by the use and fidelity requirements. Space limitations prevent us from describing the additional elements used to compose the simulator executives, but the same techniques and the OCU are used there as well.

We conclude that composition by domain-specific subsystems allows maximum freedom for the systems analysts, domain engineers, and software engineers to apply their expertise, and that having common software structures results in software applications that are more easily understood and enhanced, i.e., systems which have reduced complexity.

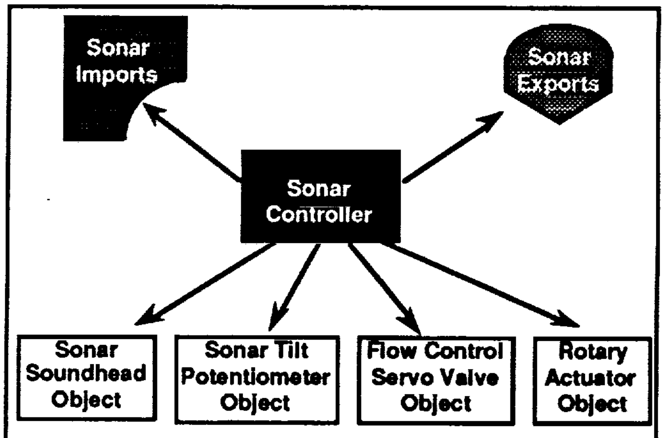


Figure 6: Sonar Subsystem Diagram