IEEE Application-Specific Software and Systems Engineering & Technology (ASSET'99),
IEEE Computer Society Press, Los Alamitos, CA, Mar. 24-27, 1999, pp. 170-179.

Analytical Design of Reusable Software Components for Evolvable, Embedded Applications

Carol L. Hoover
Department of Electrical & Computer Engineering
clh@cs.cmu.edu

Pradeep K. Khosla
Institute for Complex Engineered Systems
pkk@ices.cmu.edu

Daniel P. Siewiorek
Human-Computer Interaction Institute
dps@cs.cmu.edu

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-1890

Abstract

Transforming software requirements into a software design involves the iterative partition of a solution into software components. The partition process starts with the identification of basic high-level design components and concludes with the definition of low-level design elements such as modules, packages, and library specifications. The process is human-intensive and does not guarantee that design objectives such as reusability, evolvability, and adaptable performance are satisfied. This paper overviews our analytical approach for partitioning basic elements of a software solution into reusable and evolvable software components. We discuss the process of generating basic components for an embedded control application using a representative object-oriented design technique. Then we outline our analytical approach and demonstrate its application to a class of search techniques which can be embedded into applications requiring polynomial-time search of a solution-space. Lastly, we discuss future research directions.

1. Introduction

An embedded system characteristically includes a computer that interfaces directly with physical equipment that perform functions critical to a particular application. Typically the computer monitors and controls the attached equipment and thereby serves as an information processing component within a larger engineering system. Real-time systems, those in which processing must be done within application-specific time constraints, are often embedded in larger hardware/software systems. Burns and Wellings use the terms real-time and embedded synonymously [7]. We will do so as well for the remainder of this paper.

The process of designing the software for an embedded system is a tedious and human-intensive process. Essentially, design is a transformation from software requirements into a blueprint for the modules to be built, the tasks to be created from these modules, and the assignment of these tasks to processors. Software design involves the repeated partition of a solution into components. The partition process starts with the identification of basic system and subsystem (high-level design) components and concludes with the identification of modules (low-level design components) to be implemented.

For our discussion, the term partition has two related but different definitions: (1) the process of dividing into parts, and (2) the process of grouping elements into disjoint sets. We specifically apply the first definition to the decomposition of a software solution and the second definition to the logical grouping of solution elements such as data and operations into components. Preferably these components are readily interchanged and reused to enable the evolution of an application.

Prevailing design methods provide guidelines but minimal automated support for the systematic partition of a software solution into components that satisfy the desired behavior as well as design constraints. Commercial tools support the documentation of software designs but provide only minimal support for analyzing the "goodness" of a design. The cost and error involved in upgrading embedded systems motivates the need for systematic and automatable methods and tools for generating designs whose implementations are reusable, evolvable, and adaptable [7]. As discussed in a seminal paper by [14], many design methods recommend that the designer localize parts of the system which are to be reused or changed but require the designer to determine how this is best done. This paper originated from the guidelines used by the U.S. Naval Research Laboratory (NRL) to modularize the A-7E avionics systems as reported originally in an NRL report by [6] and in a case study by [2].

In this paper, we overview our analytical method to partition basic elements of an embedded software solution into reusable and evolvable design components that can be mapped to implementable modules. Our approach directs the designer to carefully consider reuse and change and

uses this information to mathematically determine a partition. In Section 2, we briefly overview the relationship between partitioning and object-oriented design techniques. We show partitioning decisions that might be made for a representative embedded control system. In Section 3, we briefly review research related to our work. In Sections 4 and 5, we overview our approach and demonstrate its application to a class of search techniques used to solve complex problems such as scheduling the use of distributed resources. We conclude in Section 6 with a discussion of the results of applying our approach. In Section 7, we discuss future research directions.

2. Background

In this section, we use examples to emphasize the importance of partitioning (decomposition and grouping) in the design process. Figure 1 shows a macro-level view of the transformation from requirements to design using object-oriented development techniques. With these techniques, the designer is responsible for determining the appropriate level of decomposition and the grouping of solution elements to best satisfy design objectives. For our example, we reference the object-oriented analysis and design methods as defined by [5].

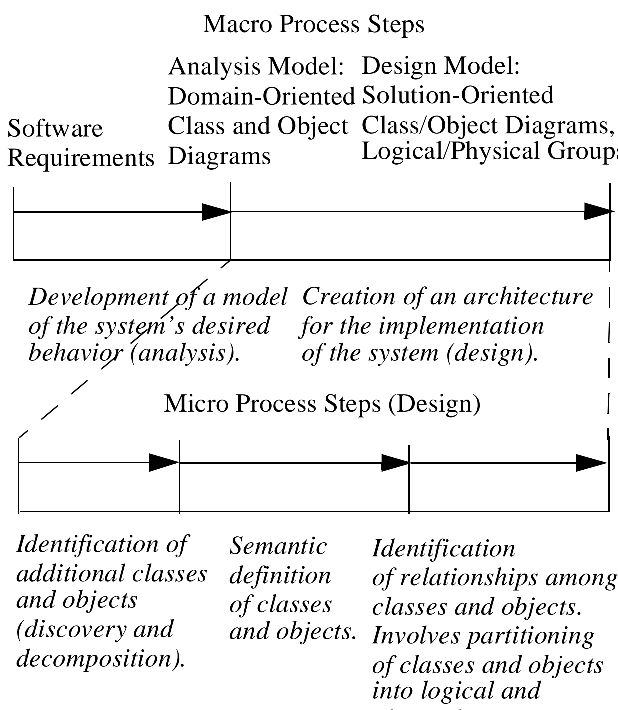


Figure 1: Transformation from requirements to design - Object-Oriented Approach.

The identification of user-oriented objects (classes) and solution-oriented objects (classes) involves abstraction as well as partitioning. In Figure 2 in Appendix A, we show the transformation from a user-oriented to a solution-oriented view of a generic control system. In the solution space, objects (classes) communicate with the external sensor and actuator objects shown in the user view. The solution-oriented sensor, actuator, and control objects are decomposed into generic super-objects and more specialized objects which hide the details of the specific sensors, actuators, and controllers. The designer may also decide to logically group the sensor, actuator, and controller objects that will collaborate in the real system.

Throughout the design process, the designer must repeatedly partition the design space and decide between alternative designs. In Figure 3, Part (a) in Appendix A, we show that a designer may alternatively decompose a communication component into subcomponents defined by performance characteristics or by functional use. A partition according to performance may simplify the mapping to execution-time tasks, whereas a partition according to functionality may best localize logic that differs for each type of device.

Figure 3, Part (b) in Appendix A, demonstrates three alternative levels of decomposition. A motion control system consists of three types of control: (1) program control to download, store, decode, and execute user programs that direct the motion of a particular device; (2) motion control to plan the path and perform the low-level servo-control of the motor that controls the movement of the target equipment; and (3) system control to handle system functions such as monitoring the proper operation of the system (watchdogs), start-up/shut-down, and task management. The designer must determine the appropriate level of decomposition into subsystem components. These components will be mapped into modules to be implemented and possibly into packages of modules for compilation. Components such as these may also represent building blocks to be reused across similar applications [15].

For an in-depth report on an analysis using the Booch method of the air traffic control domain, a type of embedded application, the reader should see [20]. This report outlines the results of a research project which focused on "Data Modeling for Advanced Flight Plan Processing Systems" and which was performed jointly by the Computer Information Systems group at the Technical University of Berlin and the EUROCONTROL Experimental Center.

IEEE Application-Specific Software and Systems Engineering & Technology (ASSET'99),
IEEE Computer Society Press, Los Alamitos, CA, Mar. 24-27, 1999, pp. 170-179.

3. Related Research

The research and development of patterns or styles of architecture are efforts to codify the knowledge of the expert designer. Novice designers would use a handbook of patterns or styles to guide them in the selection of a type of design that has been successively used to solve a similar problem [18], [11], and [8]. The description of the style or pattern includes a definition of the problem, the forces which guide or constrain the solution to the problem, and the solution. The solution or software design consists of components and their interactions. Some researchers are developing languages for describing patterns. For instance, the ROOM approach defines a model and language for documenting architectural patterns for real-time systems [17]. The styles or patterns approaches depend upon the expert designer to generate and codify good designs.

The SAAM, Software Architecture Analysis Method, is a technique for evaluating a candidate software architecture with respect to quality attributes such as modifiability and performance. The human evaluator ranks candidate architectural descriptions with respect to an agreed upon set of scenarios of how the system will be used. Like the styles or patterns approach, SAAM requires the designer to generate the candidate architectures [3].

Our goal is to make the process of generating a software architecture more systematic and automatable. We mathematically model the relationships between basic solution elements according to design objectives such as reuse, evolution, and adaptability. We use these relationships to determine a good partition of the solution elements. Similarly the hardware-software codesign community partitions basic functional units for implementation in hardware or software and for allocation to different processing elements [1]. The difference between their work and ours is that they start with predefined functional units that are to be mapped to hardware or software, while we start with requirements that are to be transformed to functional units or components.

Two other research efforts at Carnegie Mellon University have studied the design of reconfigurable and evolvable, real-time software. Chimera objects are reusable software packages that enable the rapid deployment of application processes activated by a real-time kernel [19]. The Simplex Architecture is a design approach for building software modules, called cells, that can be safely interchanged during run-time to enable the dynamic upgrade of system control [10]. In contrast to both Chimera and Simplex, our concern is with the static organization of software and the impact of this organization on the software engineer's ability to reuse, evolve, and adapt a software solution via static changes to the software design and resulting implementation.

4. Partitioning Data and Operations

Basic elements of a software solution include data, operations, and control flow. Following an object-oriented approach, the designer identifies objects, some or all of which encapsulate key data for the software solution. The problem is that the designer is solely responsible for determining the composition of the basic system components or objects. Prevalently used design methods do not guarantee that the designer will consider the appropriate level of reuse or group together those elements which change together.

Our approach has two primary features: (1) a manual but guided reuse and change analytic process and (2) a mathematical model and automatable algorithm for localizing solution features that change together. Our approach consists of the following six basic steps which were adapted from the process reported in [12].

1. Identify the basic data and operational features of the software solution.
2. Recursively decompose the large-effect operations and identify additional key data elements.
3. Enumerate the feasible or expected changes to the software solution.
4. Determine the change set of data and operations for each expected change.
5. Combine and componentize the overlapping change sets.
6. Add other necessary components.

The first step is to identify the basic data and operational features of the software solution. For our genetic algorithm example, the characteristic features are a population P of samples (with an initial state and a potential goal state) and a transition function T. The second step involves the recursive decomposition of the large-effect operations into non-trivial, smaller-effect operations. A non-trivially reusable operation encapsulates some logic whose reuse would reduce the cost to design, implement, or maintain an application or class of applications. The decomposition should also divide a large-effect operation into smaller-effect operations that are easier to understand and implement.

The third step is to enumerate the types of changes that the researcher would expect to make to the software solution. Prime candidates are changes to the data and operations that were identified in steps one and two. Determining expected or reasonable changes to a software solution requires the designer to think critically about both the problem and the solution domains. The designer should talk with the domain expert about changes in requirements that may be planned as well as those which are feasible though not specifically planned. The designer then considers the changes to the solution that would be necessary to support the changes in requirements.

The fourth step involves the partition of the data and small-effect operations into software components based on the analysis of anticipated changes to the software solution. The goal is to group data and operations affected by the same changes into the same components. We might think that we are done in that each set of data and operations affected by a change could be the contents of a software component. But such compositions could result in duplicate copies of data or operations. So in step five, we mathematically combine those components that intersect directly or transitively. Lastly in step six, we add software components to encapsulate those parts of the software solution not delineated in steps two through four.

Our analysis of a genetic algorithm resulted in the six software components shown in Figure 5. Each component has a boldface *change signature* which represents the anticipated changes associated with that component. Encapsulated data is shown on the next line of the component diagram. The operations contained within the component are displayed in italics. We were able to make the expected changes either by replacing or by modifying only the software component associated with a particular change.

We also applied our analysis technique to the simulated and genetic annealing algorithms. We found that we could reuse the software components across the three different search algorithms as indicated in Figure 6. The genetic and hybrid genetic annealing algorithms share the population software component, but the simulated annealing algorithm uses a configuration component. Each search algorithm requires its own Master Controller component. For performance improvements, the designer may want to execute the resulting components across several processors. For instance, the genetic solution could be distributed as shown in Figure 4. The designer could develop different versions for the components that are adaptable to the level of available resources.

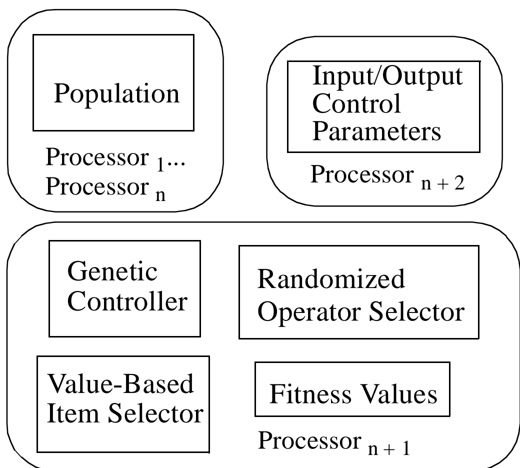


Figure 4: Distribution of genetic components on multiple processors.

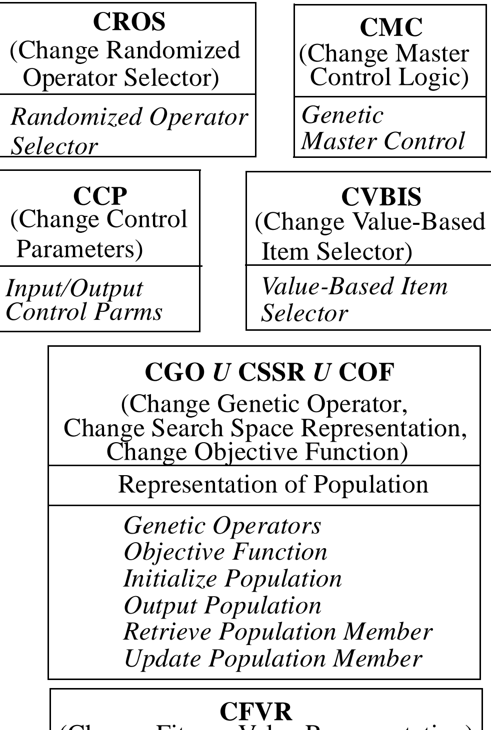


Figure 5: Componentized data and operations for the genetic search solution.

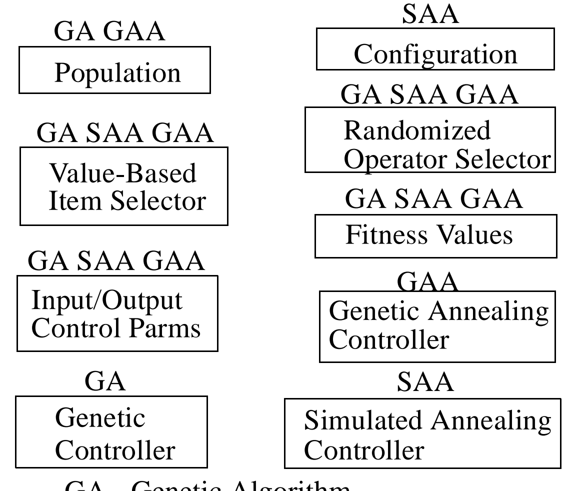


Figure 6: Component reuse across three search algorithms.

5. Partitioning Control Flow

In this section, we analyze change with respect to control flow. By control flow, we mean the order of execution of a set of tasks. For instance, in our genetic algorithm example, the Master Controller activates a sequence of tasks $<t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9>$ in which each task t_i is defined as shown below.

t_1 Initialize the population.
t_2 Retrieve the population member 1.
t_3 Retrieve the fitness value for member 1.
t_4 Randomly select the population member 2.
t_5 Randomly select a genetic operator.
t_6 Apply the genetic operator to generate a new member.
t_7 Determine the fitness value for the new member.
t_8 Select the most fit of member 1 or the new member.
t_9 Save the most fit member 1 or the new member.

Application requirements may vary over time thereby requiring changes to the original control flow. In some cases, the domain expert may specify a list of alternative activation sequences in the requirements specification such as $<t_1,t_4,t_5,t_2,t_3,t_6,t_7,t_8,t_9>$ and $<t_1,t_5,t_2,t_3,t_4,t_6,t_7,t_8,t_9>$ for the genetic algorithm. One approach to reduce the complexity of making changes to the control flow is to put the task activations in components separate from the tasks. The high-level flow of control would be embedded in control components that activate the task components. We used this style of architecture to design control components for the search algorithms discussed in Section 4.

The Master Controller components separated the details of the high-level sequence of task activations from the low-level details of the operations. The idea of decoupling the part of a software solution which may change (in this case the order of task activations) from the part not involved in the change (the tasks themselves) is similar to the design pattern in which *Mediator* objects coordinate the application-specific interactions between *Colleague* objects [11]. If the Master Controller logic becomes complex or if we want to distribute it over several processors, then we need a way to partition the Master Controller logic into separate components. Outlined below is our analytical process for optimally partitioning a sequence of task activations into components.

1. Develop a metric for determining the difficulty and "error-proneness" of modifying the control components.
2. Determine the way in which the control components would be modified by the maintainer and relate the modification steps to the counting that must be done for the metric.
3. Express the required control flow as a sequence and generate all partitions of this sequence.
4. For each partition and for each potential change to the required control flow, "walk-through" the necessary modifications to make the change and apply the metric. Save these values in a table that stores the change complexity value for each partition with respect to every potential change.
5. For each partition, sum the change complexity values to derive the total across all potential changes. The partition with the minimum total is the best design choice for the required control flow and potential changes.

In the first step, we select the summation of the number of task activations involved in changing the control flow to be our complexity metric. In the second step, we decide to localize the activation of control components in a "master" control component. We use "master" to distinguish this component from the Master Controller component discussed in Section 4. We relate our complexity metric to the summation of the size of each control component that is modified to accomplish a particular change.

The third step is to determine the allowable partitions of the required sequence $<t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9>$ (those which preserve the original task order) as shown below:

$<t_1><t_2><t_3><t_4><t_5><t_6><t_7><t_8><t_9>$
$<t_1><t_2><t_3><t_4><t_5><t_6><t_7><t_8,t_9>$
$<t_1><t_2><t_3><t_4><t_5><t_6><t_7,t_8><t_9>$
$<t_1><t_2><t_3><t_4><t_5><t_6><t_7,t_8,t_9>$...
$<t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8,t_9>$. In step four, we "walk-through" the modification of each partition for each change and determine the total number of task activations contained in the components affected by the change. Lastly in step five, we total the sums across all changes and determine the partition that minimizes this total.

The number of partitions of a sequence S increases exponentially with the length of the sequence: the cardinality of the set of all partitions is 2^{n-1} where n is the length of S. The execution time required to find the optimal partition is proportional to $O(2^n)$. For small values of n, this may not be a problem. But as n increases, the time to determine an optimal partition may become computationally expensive. One could apply a combinatorial optimization algorithm to determine a reasonably good partition in polynomial time. The change complexity metric would provide the rationale for an objective function, but automation of the process of "walking-through" the change to a control flow sequence to determine the change complexity value is an intricate process.

During our "walk-through" of making the alternative changes, we observed that locating each invariant subsequence a single component helps to reduce the overall change complexity value. We proved that in most partitions the location of invariant subsequences in single components does indeed reduce the complexity value. We

designed an $O(m*n)$ algorithm for locating the longest invariant subsequences, where m is the number of alternative sequences and n is the length of the required sequence [13]. We placed each invariant sequence $<t_2,t_3>$ and $<t_6,t_7,t_8,t_9>$ in a separate component and componentized the other subsequences. The resulting control components are shown in Figure 7.

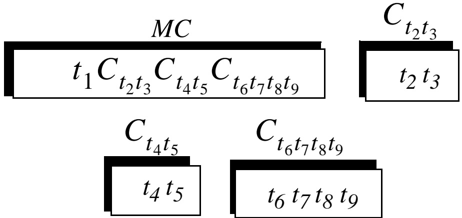


Figure 7: Control components which localize invariant subsequences.

6. Discussion of Results

By localizing the parts of our example genetic solution that change together, we could more easily modify our componentized solution than the original monolithic version. For example, we could change the representation of the population by modifying or replacing only the population component rather by reviewing the entire monolithic solution to find those parts that operate directly on the population members. We could more quickly and easily make the necessary changes in a smaller, more functionally coherent population component because there was less distance between related changes. In the componentized solution, we reused the parts not affected by the change and could thereby guarantee that they perform as predictably and reliably as before the change was made. In the monolithic program component, we had to be careful not to introduce errors into unrelated sections of the monolithic solution. In the componentized solution, we could readily improve the performance of a particular part of the solution by replacing the component which encapsulates that part. We were also able to execute the small-effect, reusable components concurrently as shown in Figure 4.

Our analytical approach supports the fundamental design principles of decomposition, partitioning, and encapsulation [14]. Our approach provides direction about how to organize both the population and fitness data as well as the non-data parts of the software solution such as the randomized operator selector and the Master Controller logic. While the definition of objects still depends largely on the creative thinking of the human with typical object-oriented approaches, our components originate from a systematic analysis of specific changes that we expect to make to software solutions or that we think are feasible. In conclusion, we think that our approach complements existing design methods by making more systematic and mathematical the process of partitioning a software solution into components.

7. Summary and Future Research

The purpose of this paper was to overview our approach for partitioning a software solution into components that reduce the impact of change, that promote component reuse, and that enable the solution to be adapted for performance. We discussed the role of partitioning in the transformation from requirements to design for a representative embedded control application and showed that this process is highly dependent on the expertise of the human designer. Then we outlined our change analytic approach for grouping together data and operations that are impacted by the same changes and applied it to the partition of a genetic algorithm into components.

We also discussed ways to partition a high-level sequence of task activations into evolvable control components and briefly demonstrated a process for determining the partition which minimizes the complexity of changing the sequence. We explained why the complexity of performing this process is exponential and noted our discovery that placing each invariant subsequence in its own component tends to minimize the complexity of change. Using a polynomial-time process for locating such invariant subsequences in the control flow, we then demonstrated the componentization of the task activations for the example genetic algorithm.

Three important features characterize our design approach: (1) the recursive decomposition of large-effect operations into small-effect operations, (2) the enumeration of anticipated or feasible changes to the software solution and the analytical grouping of solution elements impacted by the same changes, and (3) the identification of a heuristically good way to organize control flow elements. We note that the process of decomposing for reuse and the identification of the impact of change are qualitatively dependent on human judgment: at present, the best we can do is to program the computer to remember our decomposition decisions and identification of changes. We can mathematically model and therefore automate the process of grouping together change dependent data and operations. Likewise, we can formally model and automate the identification of invariant subsequences in control flow sequences: a fact which leads us to an important observation and direction for our future work.

The representation of change dependencies as a set of related elements and control flow as a "sequence of symbols" is analogous to the work of information theorists who

search for relationships between data in order to determine a "good" organization of the data. Similarly, we are researching ways to symbolically and mathematically represent what we know about "good" designs. Algorithms developed by researchers working in the areas of data analysis and clustering theory may help us to model and automate additional features of the design process. The reader should see [9] for an extensive collection of data clustering algorithms and [16] for a review of the literature on clustering theory. Though the idea of applying information-theoretical models to the analysis of software structural complexity is not new [4], our search of the literature shows that the systematic application of information theory to software design is an open area of study. We intend to further research and develop methods to systematically and semi-automatically generate designs that satisfy system design constraints such as evolvability and adaptability to available system resources.

8. References

[1] Adams, J. and D. Thomas (1997), "Design Automation for Mixed Hardware-Software Systems," In *Electronic Design*, Vol. 45, No. 5, pp. 64-66, 71-72.

[2] Bass, L., P. Clements, and R. Kazman (1998), "A-7E: A Case Study in Utilizing Architectural Structures," Chapter 3 in *Software Architecture in Practice*, Addison-Wesley Publishing Company, Reading, MA, pp. 45-71.

[3] Bass, L., P. Clements, and R. Kazman (1998), "Analyzing Development Qualities at the Architectural Level," Chapter 9 in *Software Architecture in Practice*, Addison-Wesley Publishing Company, Reading, MA, pp. 189-220.

[4] Belady, L. (1981), "Complexity of Large Systems," Chapter 13 in *Software Metrics: An Analysis and Evaluation*, A. Perlis, F. Sayward, and M. Shaw, Eds., MIT Press, Cambridge, MA, pp. 225-233.

[5] Booch, G. (1994), *Object-Oriented Analysis and Design: With Applications*, Second Edition, The Benjamin/Cummings Publishing Company, Redwood City, CA.

[6] Britton, K. and D. Parnas (1981), *A-7E Software Module Guide*, NRL Memorandum Report 4702, Dec.

[7] Burns, A. and A. Wellings (1990), Chapter 1: "Introduction to Real-Time Systems" and Chapter 2: "Designing Real-Time Systems," *Real-Time Systems and Their Programming Languages*, Addison-Wesley Publishing Company, Wokingham, England, pp. 1-39.

[8] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996), *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, Chichester, England.

[9] Diday, E. (1994), *New Approaches in Classification and Data Analysis*, Springer-Verlag, Berlin, Germany.

[10] Gagliardi, M., R. Rajkumar, and L. Sha (1996), "Designing for Evolvability: Building Blocks for Evolvable Real-Time Systems," In *Proceedings of the Real-Time Technology and Applications Symposium*, June 10-12, IEEE Computer Society, Los Alamitos, CA.

[11] Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, MA.

[12] Hoover, C. and P. Khosla (1996), "An Analytical Approach to Change for the Design of Reusable Real-Time Software", In *Proceedings of the Second Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 1-2, IEEE Computer Society Press, Los Alamitos, CA.

[13] Hoover, C. and P. Khosla (1997), "Analytical Design of Evolutionary Control Flow Components," In *Proceedings of the Second Workshop on High Assurance Systems Engineering*, Aug. 11-12, IEEE Computer Society Press, Los Alamitos, CA.

[14] Parnas, D., P. Clements, and D. Weiss (1984), "The Modular Structure of Complex Systems," In *Proceedings of the Seventh International Conference on Software Engineering*, March, pp. 408-417, Reprinted in *IEEE Transactions on Software Engineering*, SE-11, pp. 259-266, 1985.

[15] Ramachandran, M. and W. Fleisher (1996), "Design for Large Scale Software Reuse: An Industrial Case Study," In *Proceedings of the Fourth International Conference on Software Reuse*, April 23-26, IEEE Computer Society Press, Los Alamitos, CA, pp. 104-111.

[16] Reinke, R. (1991), *Symbolic Clustering*, Ph.D. Dissertation, Report No. UIUCDCS-R-91-1704, Department of Computer Science, University of Illinois, Urbana-Champaign, IL.

[17] Selic, B., G. Gullekson, and P. Ward (1994), *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, NY.

[18] Shaw, M. and D. Garlan (1996), "Architectural Styles," Chapter 2 in *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, NJ, pp. 19-32.

[19] Stewart, D. (1994), *Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems*, Ph.D. Dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA.

[20] Wortmann, J. (1996), *Object-Oriented Analysis for Advanced Flight Data Management*, Report No. 96-43, Technical University of Berlin, Berlin, Germany.

Appendix A

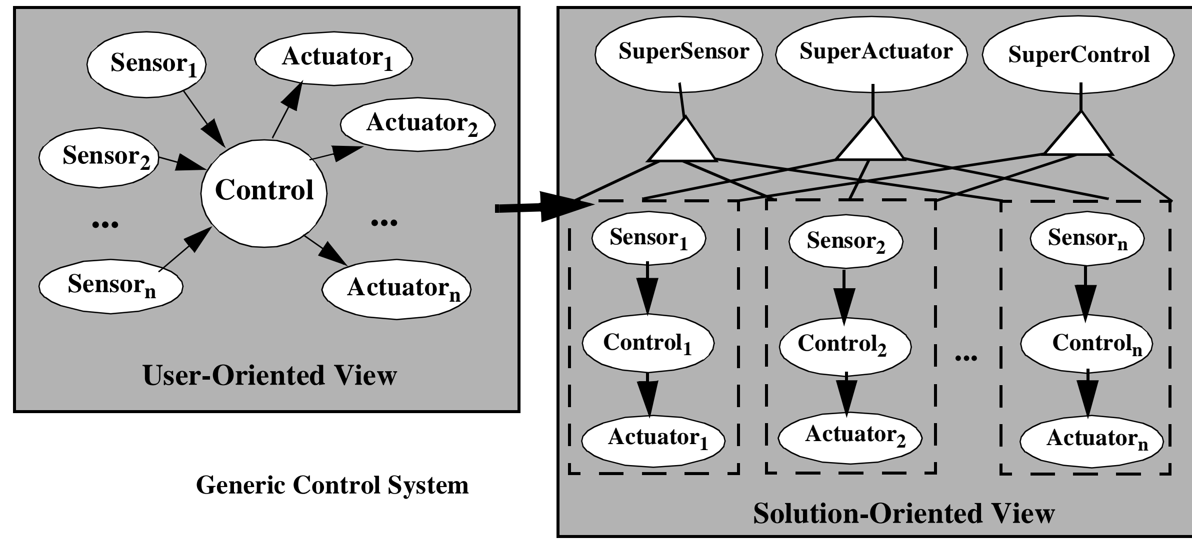
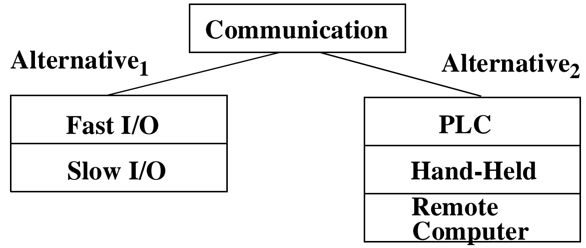
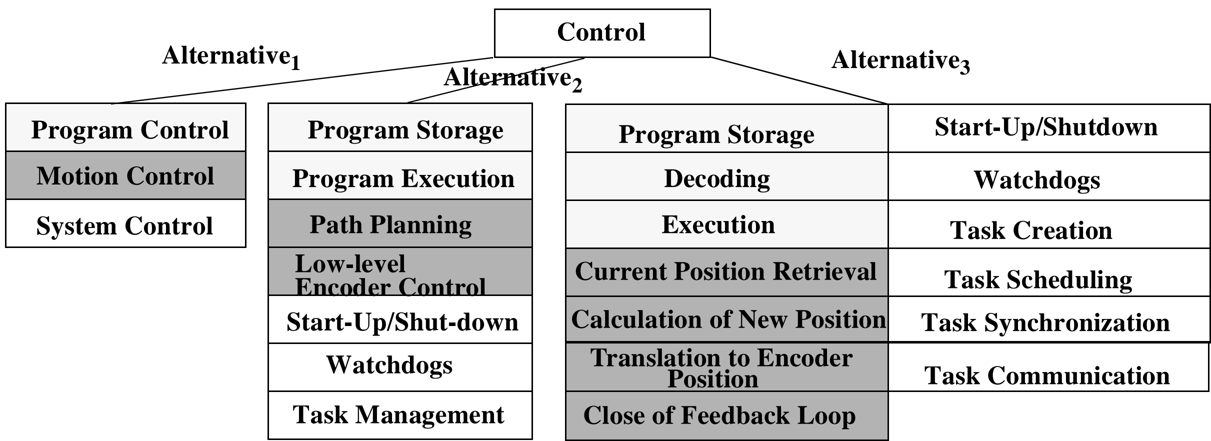


Figure 2: Transformation from user-oriented to solution-oriented objects and classes.



(a) Partition with respect to performance characteristics versus functional use.



(b) Functional partition with varying levels of decomposition.

Figure 3: High-Level Design Alternatives in a Motion Control System.