INTEGRATING SOFTWARE INTO SYSTEMS: AN AXIOMATIC DESIGN APPROACH

Jason D. Hintersteiner[1] and Amrinder S. Nain[2]

Massachusetts Institute of Technology[1]
77 Massachusetts Avenue, Rm. 31-261, Cambridge MA 02139
jdhinter@alum.mit.edu

SVG Lithography Systems, Inc.[2]
901 Ethan Allen Highway, Ridgefield, CT 06877
nain@svg.com

ABSTRACT

Today's increasingly complex electromechanical systems require extensive use of software control to achieve necessary functionality. However, software design efforts for complex systems tend to be made only after most, if not all, of the hardware has been defined. As a result, the software often bears the burden of achieving the system's desired functionality. While software is more flexible than hardware, the software design can often be greatly simplified with minor changes to the hardware design, if the software and hardware designs are done concurrently. Such unnecessary software complexity can have detrimental effects on the system in terms of safety and reliability under unusual operating conditions, as well as complicating upgrades and product redesigns.

This paper proposes a methodology, based upon Axiomatic Design, for facilitating the design of software control systems in conjunction with their corresponding hardware systems. In the Axiomatic Design framework, a system is defined in a hierarchical structure known as a *system architecture*, where the specifications for the command and control logic, which is typically implemented in software, appear at each level of the design hierarchy. Thus, the design of the system software is distributed throughout the design of the system hardware.

To apply this technique, programming terms are defined and their roles are explored in the Axiomatic Design framework. Next, a template is developed that represents system software and serves to highlight the functionality required to control and coordinate the various activities of the system hardware. A case-study example of a robot calibration routine is examined to illustrate these methods.

KEYWORDS

Axiomatic Design, System Architecture, Command and Control, Systems Engineering, Software Design

1. Introduction

While it is well accepted that electromechanical "systems" consist of both hardware and software, hardware engineering and software engineering are all too frequently treated as separate disciplines. It is therefore not uncommon to treat these tasks separately, with independent groups of designers, in product development. Since the design of the control logic for the hardware depends on what hardware must be controlled, software is typically developed after the hardware is mostly, if not completely, defined. Unfortunately, this often leads to very short development times for software, as well as the last-minute addition of functionality to the software because of hardware limitations or the perception that it will be "easier" to implement certain tasks in software. While software is more flexible than dedicated hardware, this philosophy often leads to undue software complexity.

> A computer's behavior can be easily changed by changing its software. In principle, this feature is good—major changes can be made quickly and at seemingly low cost. In reality, the apparent low cost is deceptive… and the ease of change encourages major and frequent change, which often increases complexity rapidly and introduces errors. [4 (pg. 34)]

As a result, the software can behave unpredictably, since it is impossible to test software under all possible operating conditions.

> Software may fail even after several years of satisfactory use and after thousands or even millions of copies have been installed… The failures are due to the fact that the number of possible execution paths is astronomically high even in software systems of moderate size; therefore only an extremely small percentage of all possible execution paths will ever be executed during testing and even during the whole lifetime of the system. [1 (pg. 93)]

Recent efforts have been made to perform more of the hardware and software design in parallel, in order to reduce overall development time. This cannot be done effectively, however, unless the intended functional requirements of hardware and software are well understood, and their interrelationships can be qualified. Accordingly, tools are needed to document the interrelationships between the hardware and software in a system. In Axiomatic Design, the software in electromechanical systems is represented by means of a command and control algorithm (CCA) at every level of the design hierarchy. At each level, the CCA captures the logic of the interactions among the hardware elements at that level, along with all of the communication protocols necessary to interact with its immediate parent and children CCAs. Thus, a software hierarchy emerges which mirrors the hardware hierarchy, and the software design is embedded within the hardware design [3]. Hence, the argument can be made that software and hardware design not only *should* be done concurrently, but that they *must* be done concurrently in order to provide a good system design.

Up until now, the CCA has been defined only as an abstract concept. This paper explicitly shows how the CCA is decomposed in the Axiomatic Design framework, based on the system architecture template. This includes an overview of Axiomatic Design, an overview of the traditional software design process, a definition of terms, and a case-study example of a robot calibration routine.

2. Background: Axiomatic Design Applied to Systems Engineering

Design is defined as the development and selection of a means (design parameters, or DPs) to satisfy objectives (functional requirements, or FRs), subject to constraints. Axiomatic Design provides a framework for describing design objects which is consistent for all types of design problems and at all levels of detail. Thus, different designers can quickly understand the relationships between the intended functions of an object and the means by which they are achieved. Additionally, the design axioms provide a rational means for evaluating the quality of proposed designs, and guides designers to consider alternatives at all levels of detail by making choices between these alternatives more explicit. The main concepts of Axiomatic Design include the following: (1) *domains*, which separate the functional and physical parts of the design; (2) *hierarchies*, which categorize the progress of a design in the functional and physical domains from a systemic level to more detailed levels; (3) *zigzagging*, which indicates that decisions made at one level of the hierarchy affect the problem statement at lower levels; and (4) *design axioms*, which dictate that the independence of the functional requirements must be maintained and that the information content (i.e. cost, complexity, etc.) must be minimized, in order to generate a design of good quality. Suh [5, 6] and Tate [7] provide more thorough explanations and detailed case-study examples of Axiomatic Design theory.

For large systems, a *system architecture* is developed which breaks down the design into individual systems and subsystems at each level of the design hierarchy. In this representation, a system is modeled as a series of interacting inputs and outputs, and its functions are broken down into three categories: process functions (i.e., functions that perform value-added activities), command and control logic, and support and integration functions (e.g., pneumatics, mechanical structure, etc.). A *system template* has been developed to maintain a consistent representation at all levels of the hierarchy [2]. This paper extends the representation by showing explicitly how the system template is applied to represent the software control logic in electromechanical systems.

3. The Software Design Process

Software can be defined as a set of instructions for changing the state of a computer, which, when implemented in a suitable environment, becomes an important part of a system. The overall process of designing software is very similar to the process followed in other forms of engineering, namely: (1) study of feasibility and problem analysis, (2) problem definition, (3) synthesis of design ideas, (4) analysis of the design, and (5) implementation. [1] Software design, however, is very different from hardware design in that there is no "physical" implementation of software. Accordingly, software is based on rules of logic, and not necessarily rules of physics. Software is based on mathematical and combinatorial logic, as opposed to analysis. There is also no "wear and tear" on software. [1] Several tools and techniques, such as object-oriented programming, have been developed for software design, and several programming languages have been developed and customized for particular applications.

The overwhelming majority of errors in software design emerge from errors in the software specification, meaning that the software addresses the specified requirements, but the requirements themselves are not what the customer had intended. These errors, naturally, are very difficult to detect, and often remain undetected until an incident or accident occurs. By comparison, implementation errors (i.e., errors in the code itself) are fairly easy to find, since compiler and software debugger tools exist to detect and fix such problems during development.

There are two common types of specification errors. The first are errors of omission, where key physical constraints are not satisfied because the software designers have little, if any, training in the hardware design, and therefore do not consider physical constraints which may be more obvious to a hardware engineer. [1, 4] The second, and far more onerous, are errors of added functionality. Because of the lack of physicality, there is a common misconception that, compared to changes to a hardware design, changes to a software design are easy to make. While software is very flexible because it is very simple to modify particular lines of code, understanding which lines should be changed and ascertaining the implications of those changes on other parts of the system is just as difficult as analogous changes in hardware. [1, 4] Unfortunately, functionality is often added to the software design to compensate for inherent problems in the hardware design. This functionality tends to emerge very late in the process, and has to be "slipped in" to the existing software design. Hence, even a software design which starts off "clean" will not necessarily remain that way.

> Flexibility also encourages the redefinition of tasks late in the development process in order to overcome deficiencies found in other parts of the system. During development of the C-17 [aircraft], for example—a project that has run into great difficulties largely because of software problems—the software was changed to cope with structural design errors in the aircraft wings that were discovered during wind tunnel tests. This case is typical. [4 (pg. 34)]

In order to improve the design of systems, a framework must be established to link hardware and software design. This framework must not only capture the complete functionality required by the software early in the design process, but also highlight the implications of making design changes to one part of the design to compensate for design deficiencies in other parts. The next section shows such a framework.

4. Application of the System Template to Software Systems

The software programs used to control systems satisfy key value-added tasks that are necessary for the entire system to run effectively. However, the software programs alone are not sufficient to provide control – the order in which the tasks are performed, the need to interface with a user (i.e. an operator, maintenance engineer, or an autonomous external computer), and the need to handle and recover from errors are also important tasks which direct and support the overall task to provide control functionality. Thus, it is important for the design of a CCA to capture all of these elements. Accordingly, it is convenient to consider each CCA as a system in its own right, and the system template should, and indeed does, apply to the representation of these software systems. In order to see this clearly and provide a meaningful basis for discussion with software engineers, an analogy is drawn between the terms commonly used in software and the system template.

- ***Programs:*** A program is a block of code which performs a desired value-added function or task. It may be a stand-alone executable or a subroutine embedded within a larger program. At the top level, tasks typically include providing for normal operation, initialization, and shutdown / servicing. At lower levels, tasks generally involve processing certain inputs to generate appropriate outputs. Thus, a program is analogous to a process subsystem (DP).

- ***Interfaces:*** The FR to provide an interface with the external environment (i.e., human operator, factory computer, or higher and lower levels of software control) emerges at each level of the software system hierarchy, after the key programs have been defined. This dependency results from the fact that the programs (DPs) define what kind of information will be available to and needed from the environment in order for the programs to run properly. (In some cases, constraints may be placed on the FRs for the programs, dictating the type of information that should be available to and from the user.) An interface at a particular level of the design hierarchy is composed of building blocks, which define all of the types of interfaces available between the system and the environment, as well as specific interfaces to correspond with each program at this level. Thus, the interfaces are analogous to transport subsystems, as they are responsible for transporting information.
- ***Control Logic Diagrams:*** This diagram (typically a state diagram or a functional block diagram) is used to show how the different programs in a software system interact with one another, both in terms of the information exchanged through variables as well as the order of execution, indicating if any tasks are running in parallel (i.e., multitasking). This is analogous to the CCA in the system template, since it defines how the various programs are controlled and coordinated. An individual diagram does not need to be decomposed further, though diagrams from multiple levels in the system architecture can be used to show a range of global to local views of the software logic.
- ***Support Programs:*** In order to support the value-added programs, interfaces, and control logic, certain supporting tasks must be performed. These tasks include garbage data collection, memory allocation, variable and timer initialization, log files, and so forth. In addition, the support functionality incorporates the need for libraries and error handling and recovery.

  A library is defined here as a block of code which has general functionality, and is typically accessed by multiple programs to perform a particular support task. An example of a library is a function contained in a header file to generate the square root of a number. As long as the function is accomplished to the desired accuracy, the particular algorithm implemented to generate the result is unimportant to the overall software design. (For example, in the case of square root, multiple algorithms exist which can produce results accurate to varying numbers of significant digits.) Each library is captured within the support program at the highest level of the system architecture hierarchy which utilizes it.

  The requirement to handle and recover from errors, both errors that can and cannot be predicted, is a complex issue in software system design. Errors that can be predicted are defined by the known constraints and limitations placed on the software design, and specific error handling and recovery logic can be developed. Unpredictable errors, however, tend to emerge from unusual and/or unintended operating conditions, as well as from mistakes in the control logic itself. For unpredictable errors, error handling and recovery is usually limited to an attempt to bring the system into a safe state. Furthermore, error handling software is itself prone to errors, and has a tendency to overcomplicate the software design and thus increase the information content. [4] The error handling captured within the support programs is generally concerned with predictable errors in the interactions between the programs, as defined by the control logic diagram. However, some error handling can also be built into the control logic itself, so there is not necessarily a clear one-to-one mapping of how error handling functions fit into the hierarchy. This is an area of continuing research.

Table 1: Comparison of the system architecture template for representing hardware and software.

	Functional Requirements (FRs)		**Design Parameters (DPs)**	
	Hardware Domain	*Software Domain*	*Hardware Domain*	*Software Domain*
1	Perform physical process #1	Perform control task #1	Process subsystem #1	Program #1
2	Perform physical process #2	Perform control task #2	(a) Process subsystem #2 (1st opt.) (b) Process subsystem #2 (2nd opt)	(a) Program #2 (1st opt.) (b) Program #2 (2nd opt.)
3	Perform physical process #3	Perform control task #3	Process subsystem #3	Program #3
4	Perform process #4 (Transport)	Perform control task #4 (Provide interface w/ user)	Process subsystem #4 (transport subsystem)	User interface
5	Schedule and coordinate all local process functions	Schedule and coordinate all local tasks	Command and control algorithm (CCA)	Control logic diagram
6	Integrate & support functionality	Integrate control tasks	Support systems	Support programs

$$\begin{bmatrix}\text{Perform task \#1}\\ \text{Perform task \#2}\\ \text{Perform task \#3}\\ \text{Provide interface}\\ \text{Control processes}\\ \text{Integrate functions}\end{bmatrix} = \begin{bmatrix} X & O & O & O & O & O\\ {}^{X}\!/_{O} & X & O & O & O & O\\ {}^{X}\!/_{O} & {}^{X}\!/_{O} & X & O & O & O\\ {}^{X}\!/_{O} & {}^{X}\!/_{O} & {}^{X}\!/_{O} & X & O & O\\ X & X & X & X & X & O\\ X & X & X & X & X & X\end{bmatrix} \begin{bmatrix}\text{Program \#1}\\ \text{Program \#2}\\ \text{Program \#3}\\ \text{User interface}\\ \text{Control logic diagram}\\ \text{Support programs}\end{bmatrix} \quad (1)$$

The system template, showing the application to both hardware and software systems, is provided in Table 1. The idealized design matrix to maintain a decoupled design is shown in Equation 1 for software systems. In this equation, an "X/O" means that a relationship may or may not exist between different programs, though the design of the software system still conforms to the system template.

5. Case Study Example: Robot Calibration Routine

The system template for hardware and software systems has been applied to the design of a photolithography tool manufactured by SVG Lithography Systems, Inc. The system uses one 6 DOF robot to move wafers between different wafer processing areas in a work cell, as well as moving the wafers into and out of the system. A second robot is also used in a similar fashion for transporting reticles (i.e., wafer field masks). The example described below outlines the design of the robot calibration routine for these robots. This routine is responsible for initializing and calibrating the robot with respect to the discrete locations in each work cell.

Constraints imposed on the design of the robot calibration routine include the use of a standard robot accessory (a teaching pendant with display, known as the MCP control pad) for the user interface, speed and trajectory limitations, restrictions on robot motions at each discrete location in the work cell, and implied constraints for minimizing the necessary time required to calibrate the locations. Efforts were made early on in the design process to establish and reconcile the functional requirements dictated by various departments, including engineering, assembly, field servicing, etc. For example, requirements from engineering emerged from the design of the work cell itself, while field service requirements focused more on ease of use and maintaining a short learning curve.

The top-level decomposition is shown in Table 2. The programs are the blocks of code which perform the value-added functions of selecting the locations (DP.1), moving the robot between locations (DP.2), calibrating the locations (DP.3), and recording the locations (DP.4). The only interface defined here is the user interface (DP.5), which displays information gathered by and given to the user during different phases of the calibration. The control logic (DP.6) is shown in Figure 1. The support programs (DP.7) constitute the elements required to maintain the continuity thread between the various programs and the control logic. These include global variables, continuous error recovery logic, library functions, and so forth.

Table 2: Decomposition of the robot calibration routine (top level).

	Functional Requirements (FRs)		**Design Parameters (DPs)**
	Name	*Description*	*Description*
		Calibrate robot locations	***Robot calibration system***
1	*Select locations*	Select location to calibrate	Location selection list
2	*Move robot*	Move robot between valid locations and reset position	Robot motion algorithm
3	*Calibrate*	Calibrate location	Calibration algorithm
4	*Record*	Record location	Record algorithm
5	*Interface*	Provide an interface to the user	MCP control pad interface
6	*Control*	Schedule and coordinate all local tasks	MCP operational control logic diagram
7	*Support*	Integrate and support tasks	Support programs & error handling

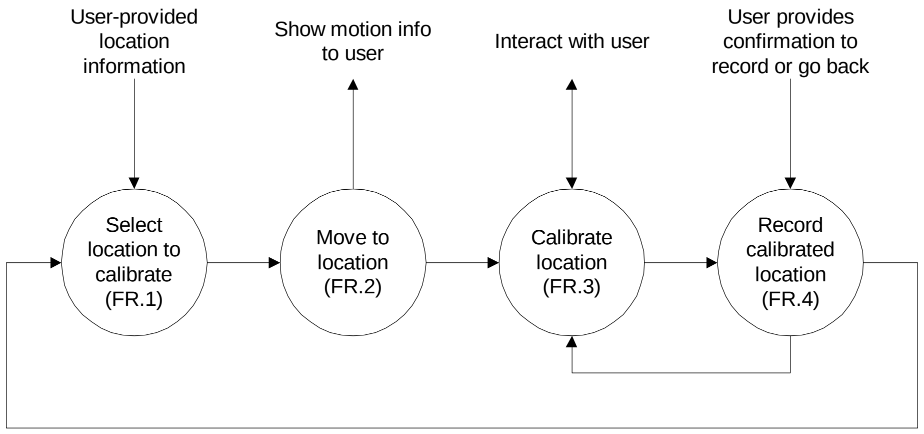
User-provided location information
Show motion info to user
Interact with user
User provides confirmation to record or go back
Select location to calibrate (FR.1)
Move to location (FR.2)
Calibrate location (FR.3)
Record calibrated location (FR.4)

Figure 1: Control logic diagram for the robot calibration system (DP.6).

The corresponding design matrix, shown in Equation 2, follows the system template and indicates that the robot calibration routine is a decoupled design. The off-diagonal "X" terms indicate that, for example, the locations to be calibrated must be established before the motion to the locations and the calibration and recording routines for those locations are designed. This has ramifications not only for how the programs interact, but also for the user interface. Similarities between the information exchanged with the user for each program give rise to the creation of basic building blocks for developing the interface. While not shown here, the decomposition has been performed to the low level design for this software, and the system representation for software holds at every hierarchical level.

$$\begin{bmatrix}\text{Select locations}\\ \text{Move robot}\\ \text{Calibrate location}\\ \text{Record location}\\ \text{Provide user interface}\\ \text{Control processes}\\ \text{Integrate and support}\end{bmatrix} = \begin{bmatrix} X & O & O & O & O & O & O\\ X & X & O & O & O & O & O\\ X & O & X & O & O & O & O\\ X & X & X & X & O & O & O\\ X & X & X & X & X & O & O\\ X & X & X & X & X & X & O\\ X & X & X & X & X & X & X\end{bmatrix}\begin{bmatrix}\text{Location selection list}\\ \text{Robot motion algorithm}\\ \text{Calibration algorithm}\\ \text{Record algorithm}\\ \text{MCP interface}\\ \text{Control logic diagram}\\ \text{Support programs}\end{bmatrix} \quad (2)$$

6. Conclusions

This paper has demonstrated how command and control algorithms (CCAs), which capture the software control logic in Axiomatic Design, are characterized in the system architecture. Specifically, CCAs are an important part of electromechanical systems at every level of the hierarchy, and thus their design cannot be separated from the hardware systems they control. Furthermore, it has been shown that CCAs are systems in and of themselves, consisting of software programs, control diagrams, and support programs, and thus can be represented by the system template. Terms from software engineering practice have been defined so that the analogy between software systems and the system template is preserved. In addition, a case study example of a robot calibration routine has been shown as a proof-of-concept that this technique is applicable to complex software control systems. Current work in this area includes application of this technique to more complex case-study examples, refinement of error handling and design constraint issues, and the integration of this technique into the system design review process.

ACKNOWLEDGEMENTS

The authors would like to acknowledge SVG Lithography Systems, Inc. for sponsoring this research, especially Matthew Van Doren, Peter Filosi, and Richard Zimmerman for their comments and support. The authors would also like to acknowledge the MIT Axiomatic Design Group and Axiomatic Design Software, Inc. for their input.

REFERENCES

1. Goos, G. and Aβmann, U. (1998) "Systematic Software Construction." Proceedings of the Universal Design Theory Workshop, Karlsruhe, Germany. May, 1998.
2. Hintersteiner, J. D. (1999). "A Fractal Representation for Systems." Proceedings of the 1999 International CIRP Design Seminar, Enschede, the Netherlands, March 24-26, 1999.
3. Hintersteiner, J. D. and Tate, D. (1998). "Command and Control in Axiomatic Design Theory: Its Role and Placement in the System Architecture." Proceedings of the 2nd International Conference on Engineering Design and Automation, Maui, Hawaii USA, August 9-12, 1998.
4. Leveson, N. G. (1995). *Safeware: System Safety and Computers.* Addison-Wesley Publishing Co., Inc. Reading, MA. ISBN 020-111972-2
5. Suh, N. P. (1990). *The Principles of Design*, Oxford University Press, New York. ISBN 019-504345-6.
6. Suh, N. P. (1999). *Axiomatic Design: Advances and Applications.* To be published by Oxford University Press, NY.
7. Tate, D. (1999). "A Roadmap for Decomposition: Activities, Theories, and Tools for System Design." Ph.D. Thesis, Dept. of Mechanical Engineering, Massachusetts Institute of Technology, Cambridge, MA.