INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Software Modeling for Reconfigurable Machine Tool Controllers

by

Sushil Birla

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1997

Doctoral Committee:
Professor Kang G. Shin, Chair
Professor Yoram Koren
Assistant Professor Nandit Soparkar
Research Scientist C. V. Ravishankar

To my family

ACKNOWLEDGEMENTS

This treatise is a tribute to the community of manufacturing engineering researchers, developers, and practitioners, who recognized the issues and opportunities in machine tool controls integration, and confronted the architectural design challenges. In this community, key supporters of this work include the organizers and panelists of the Department of Energy Integrated Manufacturing Fellowship Program, administered by the National Research Council.

In 1986, Richard Jackson, the manager of Equipment Development at the Advanced Engineering Staff (AES) of General Motors (GM) asked me to find a common approach to the design of controllers for three challenging machines under development. Directors, John Schwiekert, Ed Burke, and Roger Hiembuch endorsed the assignment. In January 1987, Wayne Moore, CEO of Moore Special Tool Company, asked me for a requirements paper, identifying machine tool technology needs, for presentation at a conference in June 1987, co-sponsored by the Air Force Mantech Office, Dayton, Ohio and the National Center for Manufacturing Sciences (NCMS), Ann Arbor, Michigan. Ron Haas, Vice President of GM AES, endorsed my participation in this communal effort. During the conference, thought leaders, Wayne Moore, Robert Hocken, Ray McClure, Richard Kegg, Ralph Taylor, and Yoram Koren, encouraged me and joined hands in formulating a machine control architecture project that received the strongest recommendation from the conference. Ralph Taylor, Jim Korein, Steve Patterson and others worked with me to produce the first requirements document in November, 1987. A number of other industry representatives in the NCMS worked with John Wagner and me to produce the second Requirements Definition Document in 1990.

By then it had become clear that my assignment required profound knowledge that was scarce in this industry. At that time, GM AES had the vision to rejuvenate the half-life of engineers through a PhD program at the University of Michigan (UM). Coincidentally, during the Winter 1991 term Professor Shin at UM asked me to make a presentation on this subject in his graduate course, "Principles of real-time computing." In the quest for that knowledge, I proposed to GM AES management my participation in the GM-UM PhD program. James Caie, Roger Hiembuch, and Gary Cowger recognized the value to the business, and approved this venture. My wife, Pramila, and children, Jyoti, Asheesh, and Preeti, stepped up to the challenge, bearing my share of the householder's duties.

During my course work, my teachers, Professors Jahanian, Rundensteiner, Irani and others patiently worked with me to answer questions of practical application. During my research, my colleagues at GM Powertrain, Whitey Simon, Jerry Yen, Clark Bailo, Warren Stanford, and others, provided ongoing encouragement.

Members of the doctoral committee, Professors Yoram Koren, Chinya Ravishankar, and Nandit Soparkar, for their active discussions, questions, comments, suggestions, guidance, and encouragement, and Professor Kang Shin, for accepting the role of advisor in a situation

that presented many difficulties, involving multiple disciplines, an ill-defined field, and an ill-conditioned student.

Members of the TEAM ICLP API working group provided critical reviews and inputs. They include John Michaloski, Doug Sweeney, Rick Igou, Dave Uchida, George F. Weinert, and Jerry Yen.

Members of the UM open architecture controller project group cooperated in building the testbed, provided focused inputs, and conducted validation experiments with the machining process in the loop. They include Robert Landers, B. K. Min, Sung Chul Jee, J. J. Park, Zbigniew Pasek, Yan Song Shan, Mark H Wu, Lei Zhou, and Professors Yoram Koren and Galip Ulsoy.

UM students participating in the research experiment produced insights and encouragement through their prototyping work and extension of the testbed. They include Brett Bandsma, Krisztian Flautner, Kevin Gudeth, Aimee Holdwick, Christopher Holtz, Jennifer Kiessel, Yan Kit Lau, Aleksandr Oysgelt, Swee Ting Pan, Hiren Parikh, Jaehyun Park, Ofer Ronen, Jigney N. Shah, Chi-to Shiu, Gabriel Tewari, Shige Wang, Michael Washburn, Ka Fai Yau, and others.

Jennifer Braganza and Yasmin Ullah assisted in documentation to disseminate the information.

This architecture is still a child in its infancy. An unusually tenaceous community of engineers and scientists has helped nurture it, and an unusually tolerant wife has nurtured me. It takes more than a village ...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CLASS STRUCTURES

Class-structure

LIST OF APPENDICES

CHAPTER 1

Introduction

This thesis research is to establish that appropriate *software modeling* of an adequately defined *domain* can improve *reconfigurability* of machine tool control software for a given level of *development effort.* Reconfigurability considerations focus on hard real-time control functions of machine tools, composed of servo-controlled kinematic devices. Development effort considerations focus on issues that affect design complexity, especially factors that require cycles of trial and error. It relies upon a software model of the domain as a reusable resource in developing and reconfiguring machine control applications. The model is extensible to support changes resulting from better understanding of original requirements, upgrades in the physical equipment, extensions in functionality or versatility, and improvement in performance. Such changes are also known as the maintenance phase of the software lifecycle. The research is performed as a series of experimental steps in which the domain model is developed incrementally and a test application is prototyped at each step to evaluate the model and discover limitations.

1.1 Computer-organization of domain knowledge

The developed software model organizes application domain knowledge in a manner that is natural to the domain. Thus, it can be used by control system developers, not merely by experienced programmers.

Communication bridge: The software model bridges significant communication gaps among manufacturing engineers and computer programmers [15], providing them a universe of discourse.

Unifying basis for integration: The model is reusable across various types of automated functions for material processing, inspection, and material handling found in agile manufacturing workstations — applications that have been historically treated by industry as different. Thus, it provides a framework for their computer integration.

External validity – industrial relevance: The model is representative of real-world industrial applications. It has been developed from a case study [5] for agile machining of automotive powertrain components, and progressively improved through interactions with groups of people experienced in developing software to control industrial automation.

Relation to open modular architecture controllers: The software model defines an architectural framework and external interfaces for fine-grained objects to be integrated in a hard real-time control system. In contrast, recent and ongoing efforts to develop standards for open architecture controls have focused on larger granularity [3,13,35,36]. Some of these efforts [7,40,48] yielded a useful textual collection of needs and requirements, but the results were not in an implementable form. The latest project [3] has demonstrated working prototypes, but the degree of reconfigurability is limited to a standardization of data interfaces focused on numerically controlled machines. It does not address reconfiguration of hard real-time functions, and does not provide a unified programming paradigm for motion control and discrete logic [34]. The model resulting from this research is a computer-interpretable requirements specification — the interface specifications can be automatically transformed into code in an implementation language. It provides a unified specification for a sequence of service requests or operations given in a program written in the EIA RS 274-D standard for numerical control and a program written in a language conforming to the IEC 1131-3 standard for programming controller programming languages. The same specification paradigm may be used to specify the flow of control for hard real-time functions, e.g., servo control of motion.

1.2 Approach

A reusable requirements-model offers very high payoff, but its development is a very complex and difficult subject of long-range research. Freeman [19,46] identified two open research issues pertinent to this study: (1) ways to bound the domain to obtain stability in its requirements, and (2) a modeling technique that makes evolution easy.
This research explores the conjunction of the *domain bounding* and of the *evolution ease* issues, scoping the domain at each stage in consideration of the incremental effort involved in its evolution and the incremental gain in applications supported. Thus, it is a very early stage in a field of complex long-term research. It primarily focuses on principles of proper modularization. Secondarily, it develops rules and constraints to develop software for servo-control of motion and similar hard real-time applications.

Completed analytical and experimental work: We bounded the manufacturing automation domain [5] to which the proposed research applies, defined the domain modeling process [9], created a domain model including a number of subdomain models [8], and developed, as its context, a framework of a controller software development life cycle [6]. Objects based on this model have been used to construct prototype applications for controlling a multi-axis machine tool. An experimental testbed and development tools have been assembled.

The domain model has been built in stages following a defined process (Chapter 2). The experimental steps are performed iteratively until an adequate validation is obtained. Developer participants log effort applied and difficulties encountered at each stage. Other influencing factors are kept as stable as possible. All experimental work has been conducted on an industrial platform, consisting of well-supported components, e.g, a VMEbus-based PC-compatible computer and input-output (IO) modules, and a commercial real-time operating system, QNX. Software is developed with the aid of PC-compatible industrial-grade tools, including C++ programming language compilers, an object-oriented CASE tool, and a tool to build graphic user interfaces.

1.3 Organization of the dissertation

Chapter 2 introduces the reconfigurability problem through an example reconfigurable machining system (RMS), describing a scenario of reconfigurations required and gaps between implied general needs and current control system capabilities. Given that all the required reconfigurations cannot be described exhaustively, the architectural design must be such that various configurations — unknown in advance, but within some known bounds — may be realized. Chapter 2 describes a software development process to define and bound the domain and to evolve a domain-specific software architecture. It starts the process with a specific example case, representative of the needs to be met by this architecture, but not fulfilled in current practice. Chapter 2 also describes the research method used to validate this approach for its evolvability and reconfigurability. Chapters 3 and 4 describe the modeling of software for one axis motion control. During this process, we identify subdomains that are applicable beyond single-axis motion control. Chapter 3 focuses on the earlier stages of the software process, developing the static aspects of the model, whereas Chapter 4 describes its dynamic and execution aspects, including architectural constraints to facilitate application design. Chapter 5 shows how the same modeling principles, abstractions, and subdomains are reused and extended for software to coordinate multiple axes of motion. Chapter 6 discusses extensions to the model to cope with the diversity found in external inputs and outputs, including sensors, actuators, and the human machine interface. Chapter 7 discusses extension of the model to coordinate different process, motion, and logic control functions, again building on the same principles and framework. The concluding chapter presents the results and evaluation of the project, summarizes the contributions, and describes how this body of knowledge may be extended with future research.

CHAPTER 2

Domain-specific modeling of machine control software

How reconfigurable should a control system be? How is adequacy determined? How is it assured? To understand these questions, we consider a scenario of a workstation (Figure 2.1 and Table 2.2), which epitomizes a reconfigurable machining system (RMS). Reconfiguration issues are illustrated by tracing data flow through the most common case (Section 2.3). From the example scenario, we generalize needs and requirements for reconfigurability following Heuristics 1–5. It is not practical to anticipate all needed configurations, pre-design solutions for them, and build the solutions in the initially supplied control system — the system should be reconfigurable. However, there is little organized knowledge to describe or meet the requirements of configurations or specific applications not known explicitly in the beginning. Therefore, we have synthesized a process (Section 2.6) to develop a domain-specific software architecture, based on two paradigms (Section 2.7) - object-orientation (OO) and the finite state machine (FSM). The process is investigated experimentally, iterating a development cycle from requirements definition to a working prototype, demonstrated in the control of a machine tool. An organization for reconfigurability and efficient composability of machine tool control software is described in Section 2.4.

2.1 Example scenario

A user envisions a future automated machining workstation (Figure 2.1) which can process a prismatic workpiece completely to the required specifications, including reorientation and repositioning of all surfaces to be machined. However, lacking the funds, skills, and firm economic justification to exploit a technologically complex workstation, the user wishes to evolve it in stages (Table 2.2), while minimizing obsolescence of initial equipment.

2.1.1 Common difficulties in control retrofit and integration

Historically, users have experienced many problems in retrofitting controls when upgrading manufacturing automation in stages. Such retrofits are very costly. The retrofitter is unable to add functionality not designed into the initial controller. A separate controller has to be added for the additional functionality. The new functions cannot be closely coupled in timing relationships with functions in the original controller. Other difficulties are described in the example scenario (Section 2.1.2) in the form of general needs for reconfigurable controllers. These general needs are an early stage of evolving requirements, starting from the example case and utilizing Heuristics 1. Generalizations are in terms of the variety

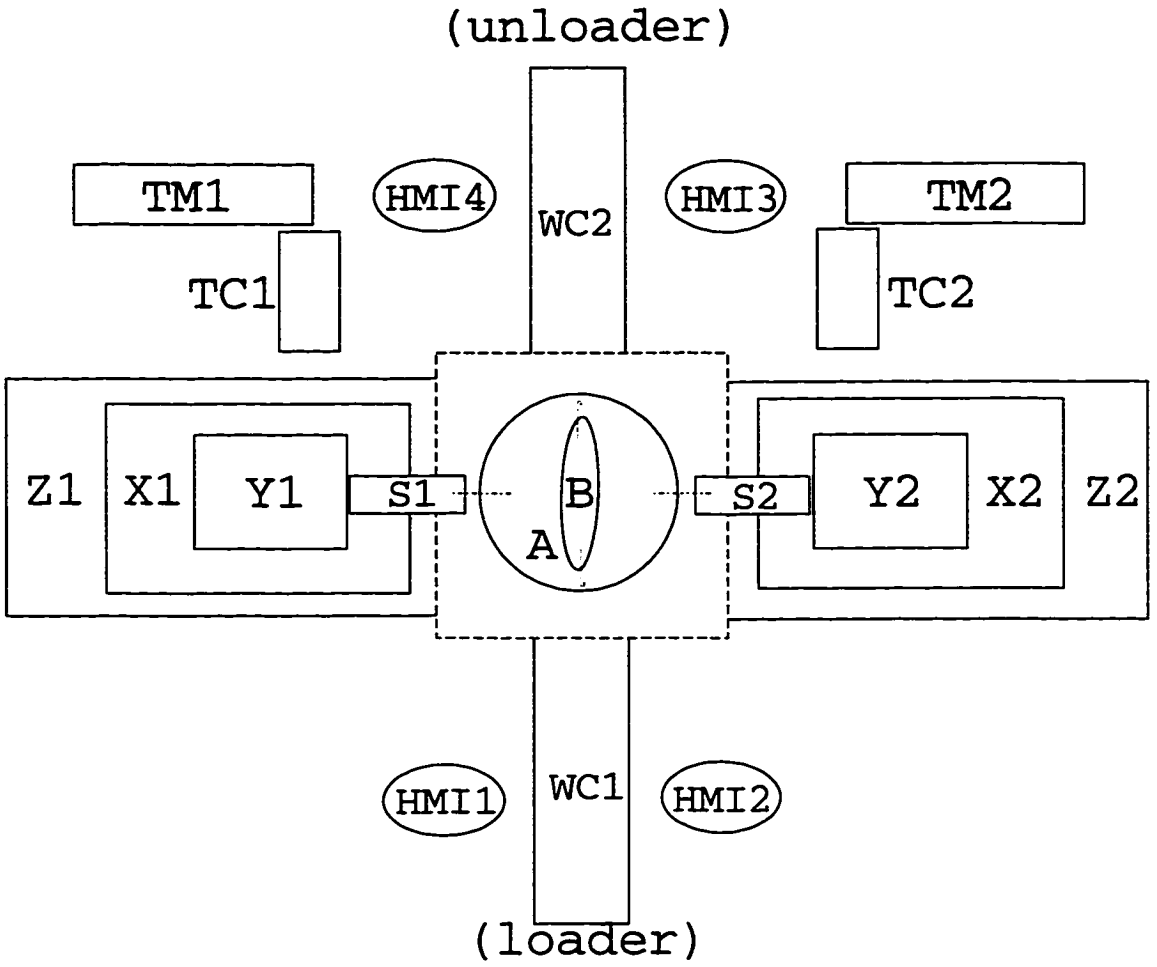

Figure 2.1: An evolvable machining workstation – an example RMS

and number of controlled objects, their interrelationships, and interactions, e.g., coordination, concurrency, interlock, and temporal coupling. The needs are over-generalized on purpose. Subsequent steps in the requirements definition process will address the extent of generalization that would be appropriate.

2.1.2 Example reconfigurations

The initial configuration (Table 2.2 Stage R1) includes MC1 and the mechanical resources (space, structure, interfaces, etc.) to allow physical expansion. MT1 operations include facing and hole-working of surfaces accessible within the combined reach of units X1, Y1, and Z1. *The user expects corresponding modularity (structure and interfaces) in the control system, with reconfigurability in software.*

Compound rotation module: Unit *A-B* is retrofitted (Table 2.2 Stage R2) with a provision for quick removal. Now the RMS can perform machining of faces and holes at different angles within the combined reach of units X1, Y1, Z1, A and B, and the machining of contoured surfaces curving in different planes. With current commercial control systems, a higher initial investment in Stage 1 would be required (e.g. 5-axis contouring control

Symbol	Description
X_i, Y_i, Z_i	Axes of translational motion (a type of machine tool element)
A	Axis for rotational feed motion (axis vertical)
B	Axis for rotational feed motion (mounted on A with axis horizontal)
S_i	Spindle (a type of machine tool element)
MT_i	Machine tool with elements X_i, Y_i, Z_1, S_i
TM_i	Tool magazine (a machine tool element that stores, retrieves and presents tools for access by TC_i)
TC_i	Tool changer (a machine tool element that exchanges tools between S_i and TM_i)
MC_i	Machining center consisting of MT_i, TM_i, TC_i
WC_n	Workchanger to load or unload workpiece (part) or other fixtured object
HMI_n	Human machine interface unit Basic usage: - Normal operation and setup of closest mechanisms Future usage: - Remote control of any part of workstation; - maintenance; - tuning; - programming control logic and other human interaction.

Table 2.1: Notation used in example of reconfigurable machining workstation.

system at the time of retrofit) or the initial 3-axis control system would have to be replaced with a 5-axis control system. A separate 2-axis control system is required to pre-test the compound rotation module before integration. If the original 3-axis system is retained, and a separate 2-axis control system is procured with the compound rotation module, their close coordination for contouring control is not feasible. Commercial CNC units allow only simple kinematic relationships amongst coordinated axes of motion. For example, eccentric rotations and non-orthogonal translational motions are either not controllable with high precision or require extensive custom engineering. There is no uniformity in specifying the kinematic relationships. The reconfiguration Stage R2 implies General Needs 1–3 for the control system.

General Need 1 *A control system must allow scalability in number of controlled objects, e.g, axes of motion, axis groups, and on-off devices, (0 ... N) with different types of control – discrete or continuous.*

General Need 2 *A control system must allow grouping of controlled objects by closeness of their interaction (time interval between information exchanges).*

General Need 3 *A control system must allow specification of a variety of kinematic configurations or relationships and apply the specification to generate the required workpoint trajectory.*

Peripheral automation: In order to reduce non-value-adding time-cost of changing tools, workpieces, or fixtures, high-speed peripheral automation (WC1, WC2, TC1, TM1) is added (Table 2.2 Stage R3). Although the peripheral motions need not be coordinated

Stage	Change in configuration	Implied General Need
R1	Initial configuration: - Basic MT1; - Mechanical structure for future additions.	-
R2	Retrofit compound rotary table (Unit A-B); Continuous path control with 1–5 axes	1-3
R3	Peripheral automation to reduce cycle time: - Tool changer TC1; - Tool Magazine TM1; - Workchangers WC1, WC2.	1-2; 4-5
R4	On-machine inspection to improve quality with minimum time loss: - compensation for true location of part, fixture. - high-speed, continuous path; - retrofit sensor;	4-5, 6
R5	Concurrent machining with multiple units to reduce cycle time: - retrofit MC2; - increase concurrency; - provide spatial mutual exclusion.	1, 4, 5
R6	Improved boring precision and flexibility: - replace servo control law or rule; - retrofit cross coupling control.	4-5, 7
R7	Retrofit broken tool detection: - stop motion in time; - monitoring such other process conditions.	2, 4, 5, 6
R8	Adaptive constraint control: - adjust feedrate to control process variable.	6, 7
R9	Chatter control: - adjust cutter path axially; - adjust cutter path radially.	8
R10	Geometric error correction: - specification of kinematic model including motion errors	3 extended

Table 2.2: Stages of evolution of a reconfigurable machine tool – an example.

with the machining motions as closely as for contouring motion, their work zones overlap, and thus, concurrent access to the same space must be restricted. Current practice does not exploit adequate concurrency in motion of peripheral mechanisms and machining mechanisms — it results in excessive idle time. The need to reduce this type of cycle time loss (General Needs 4–5) is common in automotive operations that employ interacting programmably controlled mechanisms. The example represents a class of applications that reinforce General Needs 1–2.

General Need 4 *A control system must allow specification of interactions across controlled objects, e.g., sequence of operations, concurrency, mutual exclusion, in close temporal coupling.*

General Need 5 *A control system must allow close interaction of continuous control and discrete control objects within the latency tolerable between related events or measurements, including discrete logic conditions.*

On-machine inspection: To improve and verify quality during the machining process, with minimal time loss, the user retrofits the ability to inspect dimensions of the workpiece, while it is on the machine, before and after processing steps (Table 2.2 Stage R4). To minimize idle time, measurement must be performed during continuous path motion, controlling acceleration and velocity within limits allowed for the specified measurement accuracy. On-machine inspection also allows compensation for variations in the workpiece – surfaces used as location references and surfaces to be machined. First, in current control systems, retrofit of new inspection devices or probes is very expensive. Second, run-time cost is also high, due to a limitation in the achievable speed-to-accuracy ratio. Alternatively, special hardware is required to capture probe-measurements and positions of machining axes within a bounded time difference. Stage R4 — on-machine high-speed continuous path measurement — adds General Need 6 and represents another class of applications reinforcing General Needs 4–5.

General Need 6 *A control system must allow retrofit of sensors for high-speed measurement of dimensions and other process variables during motion.*

Machining concurrently from multiple directions: The objective is to reduce work cycle time, by processing the workpiece from multiple directions concurrently, but without collision between the two motion mechanisms. Therefore, another processing module MC_2 is added on the opposite side of MC_1 (Table 2.2 Stage R5) – work zones of MC_1 and MC_2 overlap. However, in current systems, it is not easy to specify and achieve high concurrency of operation without interference. Stage R5 represents another class of applications reinforcing General Needs 1, 4, and 5.

Improved boring accuracy with flexibility: Servo-control of axes and their coordination must be improved for high-precision, flexible, boring operations. Planetary motions to generate bores extend the diametral range of holeworking with a single multi-bladed boring tool, without degrading accuracy (Table 2.2 Stage R6). The control algorithm may require a change in the servo-loop sampling interval, but this change is not easy to implement in a reconfigurable controller where multiple programs in execution (processes) share the same computing resources. Current practice also does not allow retrofit of algorithms for different tradeoffs in accuracy, performance, and cost. Stage R6 adds General Need 7 and represents another class of applications reinforcing General Needs 4–5. .

General Need 7 *A control system must allow retrofit or replacement of objects that specify and apply control laws and rules.*

Broken tool detection: When a tool fails, motion must be stopped before more damage can occur. However, current control systems do not facilitate interaction within a short time delay, commensurate with the physical reaction time. Without such protection, machining rates in practice are very conservative, increasing machining time and reducing resource utilization. If broken tool detection requires high-speed measurement of force or acceleration, then sensor-integration costs are high, because of the separate processor, separate user interface, and controller-specific communication interface. Table 2.2 Stage R7 represents class of real-time process monitoring applications that reinforce General Needs 2, 4, 5, and 6.

Adaptive constraint control: Machining rates can be adjusted to satisfy some processing constraint, e.g., within certain levels of machining forces, through change in feedrate, i.e., an "outer" control loop to override programmed feedrate. However, retrofit of the outer control loop is costly (as described above for the sensor) (Table 2.2 Stage R8). This example implies a class of adaptive rate control applications that reinforce General Needs 6 and 7.

Chatter control: A more advanced form of adaptive control provides chatter-free machining by modifying depth of cut in order to maximize material removal rate without chatter (Table 2.2 Stage R9). However, current numerical controllers do not support the needed path modification during execution (General Need 8).

General Need 8 *A control system must allow modification of motion path specifications, e.g., axial and radial adjustments of the tool path, during motion.*

Geometric error correction: Operational accuracy can be improved with the ability to calibrate and compensate for geometric errors of motion, e.g., errors in squareness and straightness of motion. (Table 2.2 Stage R10). However, there is no uniform way for the user to describe the geometric errors of motion to a controller and no uniform way in which a controller can integrate the information and apply the corresponding corrections during execution. Major geometric errors may be viewed as kinematic relationships that are arbitrarily different from the traditional kinematic relationships in machine tools (e.g., orthogonality; alignment) [17]. Thus, we treat this need as an extension of General Need 3 (Heuristics 1, Steps S5.1, S5.3, and S7).

Other upgrades that may be required: There are many other examples of useful upgrades that are too expensive to retrofit and not available in commercial off-the-shelf controllers. Unfortunately, users are not equipped to define and describe the required degree of reconfigurability. An exhaustive enumeration of all the identifiable cases is not economically usable — instead, there should be some compact requirements description that can ease the development of software. Traditional software engineering approaches and system design guidelines are inadequate. In Section 2.2, we introduce an approach to characterize requirements, such that a larger number of configurations are possible from a given set of specifications.

2.2 Characterizing reconfigurability requirements

Heuristic Heuristics 1–2 systematize the early steps of the process to develop architectural specifications. Common cases, as described in the example scenario, initialize this process. Steps S5–S6 yield a generalization that supports more configurations than the initial cases, providing a larger coverage for a given level of specification development effort. Step S7 seeks review guidance from experienced people.

Analyze each case as follows.

S1: Find whether the case is representative of many similar common cases.

S2: If commonalities are not obvious, defer search for genericity until more knowledge is acquired about the domain. If yes, proceed further.

S3: Identify specific similar cases, their collection being a domain, D_i where i= iteration number ($i = 1$ to denote the initial domain).

S4: Identify the common features (pertinent to the purpose) across D_i.

S5: Search for key parameters for describing these features, trying the following approaches.

- **S5.1:** Employ the underlying knowledge used for engineering the instances in D_i.
- **S5.2:** Search for analogies to specific examples outside D_i.
- **S5.3:** Search for structural similarities (isomorphic matches).
- **S5.4:** Employ domain-independent computer science knowledge.

S6: Use the aggregation of these parametric features for a description of the domain, D_{i+1} at the end of this iteration.

S7: Review with knowledgeable persons and improve understanding of the domain.

S8: Iterate **S3–S7**.

Heuristics 1: Method to identify genericity across specific cases.

S1: Consider common features across common configurations of reconfigurable machinery.

S2: Identify classes of components (typically physical) from which these features were composed while engineering the machinery.

S3: Recursively decompose these components, as in Step **S2**, until the conceptual primitives of the domain are discovered.

S5: Model the conceptual primitives.

S6: Successively build the model of the aggregation hierarchy discovered during the decomposition stage **S2-S3**.

Product of Steps **S5, S6** = Specifications of component classes.

Heuristics 2: Method to identify common component hierarchies.

The design of a system is the specification of its elements in terms of their functions and interfaces, their inter-relationships, given the functions of the whole system and its external interactions and interfaces. The scope of system design issues in this research is limited to application software systems, given a particular execution environment (hardware and software platform) and given a particular development environment and tool-set. This research is focused on issues of system design when a family of applications is not explicitly known. Design for such a family of systems is also known as designing an application framework

or a domain-specific software architecture (DSSA). In general, design is a combination of art, craft, and science. This research transforms a portion of that art and craft into an engineering science, in the domain of software for reconfigurable manufacturing automation. Its objective is to enable the realization of a variety of configurations, allowing for the integration of a new element, while exploiting prior investment through reuse or extension of existing elements.

Scope limited to resource-neutral cases: We exclude from the scope of this study changes made to real-time software that increase the load on given resources. We assume that either adequate resources are added, corresponding to the increased load, or the change is neutral relative to the demand on resources. Examples of the latter case are replacement (upgrade) of feedback sensors, drives, and other kinematic components, changes in control parameters, increase in sampling and update intervals, and changes in calibration parameter values.

2.3 Data flow in an example controller configuration

We describe the architectural approach and reconfigurability issues, explaining the rationale for software component boundaries, in the context of computer numerical controllers, which have certain long-standing well-understood conceptual components with fixed paths for the flow of data. Figure 2.2 shows a synthetic example data flow in a reconfigurable machine tool controller, generalized from the traditional CNC. The machine tool in the synthetic example includes an arbitrary number of objects, either under "continuous" control or discrete control. Their discussion of continuous control is limited to motion objects, e.g., axis objects (which control individual axes of motion), axis-group objects (which coordinate axes in that group), and objects that play a role in conveying instructions (service requests) to axis-group objects and receiving status information from them.

Initial conditions of the example: Assume that the user makes a sequence of selections which represent the most common, non-branching case of data flow. Furthermore, the controller and the controlled machine tool have already been powered up, and the startup, initialization, and diagnostic routines have been executed successfully. The Axis objects (Chapters 3–4 are operational, i.e., they are processing their servo-loops cyclically, holding their current positions.

Selections through user interface: The user selects a *User Process Program* to be (translated or "compiled" and) run. The *User Interface* object (Chapter 6) resorts to standard services of the *file manager* on a PC platform to retrieve the selected file object, and converts it into a *process program* object, independent of the file format.

Process program translation: No single universally-adopted standard exists for the input process program. The EIA RS274 standard is used for NC machining. A process program in this form, a part program is prepared specifically for a selected machine tool. To allow the scheduling flexibility of running the process program on any machine tool having the required capability, the process program should be in a machine-independent form. Therefore, the RS274 part program is transformed to a machine independent *ControlPlan* object (Chapter 7) through a *Translator*. Currently, there is no consensus view on the nature

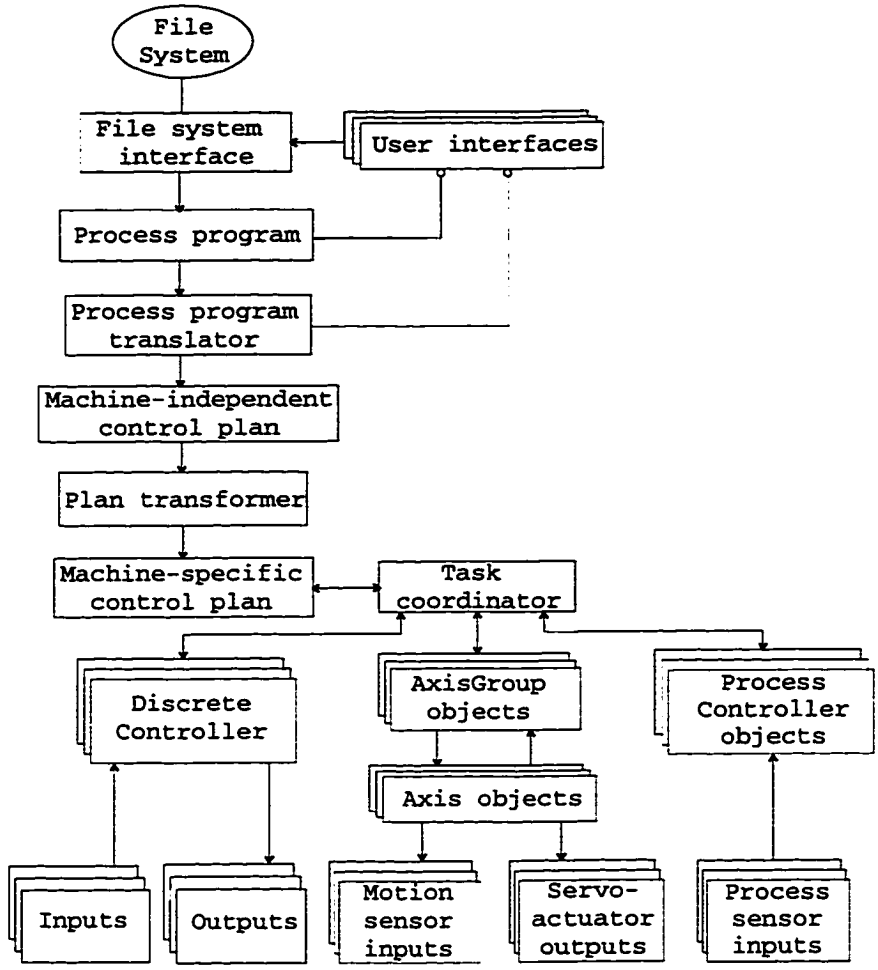

Figure 2.2: Data flow through an example configuration

of the ControlPlan, but in our concept, its generic structure allows for the specification of all functions of all objects available in the control system, and the specification of a flow of control through these functions. In the future, when a machine-independent ControlPlan standard becomes popular, this step may be external to the controller, or may not be needed at all.

Transformation from machine-independent to machine-specific plans: If the machine-independent ControlPlan is to be run on reconfigurable machine tools and control systems, there must be a transformation from the machine-independent ControlPlan to a machine-specific ControlPlan which refers to the objects and functions available on that machine. The *Plan Transformer* object performs this transformation. When multiple units of the same workpiece have to be made, this transformation does not have to be repeated. Therefore, the part program translation and transformation steps should be separated from its execution.

Machine-specific configuration: The Plan Transformer must also be machine-independent, but it must have access to the configuration of the machine on which the ControlPlan is to be run. The kinematic configuration is provided through the *MachineTool* object

(Chapter 7). The software configuration information, e.g., identities of objects to be used, is provided through the *TaskCoordinator* object (Chapter 7) at the time of startup and initialization. Suppose that a machining workstation consists of multiple machine tools, cycling independently and only loosely coordinated. Then, this coordination is provided by a *TaskCoordinator* object at the workstation level. Furthermore, suppose that the user provides each machine tool (loosely) coordinated under it with a separate RS274 program. Then, there would be separate machine-specific objects for the corresponding translation and transformation. On the other hand, suppose that the workstation shown in Figure 2.1 has motions that have to be coordinated more closely. The RS274 standard does not provide for the specification of closely-coupled motion of such a large number of axes. In the example scenario, there could be nine AxisGroup objects operational in a configuration at some instant in time, where the AxisGroup concept represents a group of axes whose motion must be closely coordinated. The ControlPlan allows for the specification of their motion parameters, the concurrency of their motion steps, and the conditions for the execution of these motion steps.

Execution of specified work cycle: When the user selects "Start Cycle" for a particular machine tool, execution of its machine-specific ControlPlan starts. This refers to a sequence of services requested from various motion or discrete or process control objects, e.g., an AxisGroup object (Chapter 5). An AxisGroup object in turn produces setpoint data for the servo control loops of the Axis objects in its group. The Axis object acquires information about the current state of the external world, e.g., position from an encoder object, and sends the next state setpoint to the corresponding actuation object, e.g., PWM amplifier.

2.4 Domain knowledge based organization of software

Table 2.3 portrays the conceptual organization of reusable component software in the form of class libraries from which machine control subsystems, and systems can be built. *Primitive data types* and general purpose language libraries are shown at the bottom of the hierarchy of architectural components — they provide the most generality, but building computer programs from such primitives is very time consuming. Therefore, we conceive an hierarchy of abstract data types with progressively increasing semantic content, but reusable in correspondingly narrower application domains. The layer above primitives is a category of classes for *measures* and *units* (consistent with ISO Standard 10103 Part 41 [22]), and *matrices* — it is useful in the domain of engineered electromechanical systems. These classes are reusable assets, deployed to build a category of classes for *space* and *kinematics* (consistent with ISO Standard 10103 Part 42 [22]), applicable to the domain of rigid mechanical systems, e.g., workholding fixtures and other tooling. These reusable assets are deployed to build class libraries for servo controlled motion components, e.g., sensors, actuators, and control components. From a relatively small number of classes, it becomes possible to compose a much larger variety of machine tool axis objects — the atomic, independently controllable units of motion, from which the kinematic model of a *machine* object is composed, consistent with ISO Standard 10103 Part 105 [33]. The same class assets are reused to build the models of peripheral mechanisms, e.g., tool changers and work changers and workstations with multiple machining mechanisms (e.g., Figure 2.1). Thus, software is organized to mirror concepts used by machinery engineers and the real objects of their creation. This organization localizes the effects of changes made by machinery

Application domain	Reusable resources (classes, categories)	
	Data and functions	Control flow
RMS with closely interacting motion mechanisms	Workstation Machinetool	ControlPlan FSM
Automated motion mechanisms	KinMechanism	ControlPlan FSM
Multi-axis coordinated motion	AxisGroup KinematicPath; PathElement	FSM
Servo-controlled axis	Axis	FSM
Axis components	AxisCompt AnalogIO; DigitalIO; CtrTimerIO ControlComponent AxisCtrl; PIDctrl	NA
Rigid mechanical systems (fixtures, cutters)	KinStructure CoordinateFrame Point, Direction Line, Circle	NA
Engineered systems	Measure; Unit; Matrix	NA
All	primitive data types standard language libraries container classes	NA
NA: Not applicable		

Table 2.3: Aggregation hierarchy of reusable class categories

engineers, whether to provide new functions to users, or to improve performance, or to replace obsolete components in the machine.

Integrating a controller from software library assets: Our architecture facilitates the composition of a wide range of controller configurations. Figure 2.4 is an example of a control system composed of objects instantiated from the classes described above, combined with platform software and hardware. The specifier or integrator of the system determines its configuration — in particular, choice of components not available in the domain model and choice of the implementation language. The system may consist of one or more cooperating tasks (shown in Figure 2.4 as the vertical bars, where a task represents the execution of a sequential program or a sequential component of a concurrent program [21].

We will also refer to them as *active objects*, when contrasting with passive objects, or as processes, where these programs run as OS-managed processes. They are reusable only in a very similar organization of computational work, e.g., the same type of operating system (OS) and platform configuration. Any of the active objects may have "glue" code that ties together the services invoked from any of the reusable software component objects. Thus, these active objects are not as reusable and not as portable as their components. The processing nodes may be interconnected with any medium, e.g., networks or buses chosen by the system specifier. Similarly, inputs and outputs (IO) may be interconnected to a processor through a bus or network.

Separation of integration model from component models: The architecture allows for the composition of an integrated workstation model in terms of its constituents and their inter-relationships, e.g., their kinematic arrangement. It may be composed of machine tools, axis groups, and axes. From a given number and type of constituent components, a much larger number of configurations can be composed. Only the configuration information containers are affected.

2.5 Research method

This work thesis has been validated experimentally, in a series of steps (Figure 2.3) to evolve the domain model primarily, and to refine the model evolution process secondarily. The initial process and model are based on existing body of knowledge, which by itself, not sufficient to ease the development of reconfigurable software, i.e., it requires much greater skills and effort than a "one-shot" application. Therefore, the purpose of the experiments is to identify a domain-specific subset of knowledge that improves the development efficiency, i.e., to identify the set of principles, rules, heuristics, procedures, and abstractions for evolving a domain-specific software architecture to ease the development and reconfiguration of applications in this domain. The software development process is designed for industrial use (Figure 2.5). It is synthesized and adapted from literature in real-time object-oriented software engineering, and relies on the concept of domain engineering [2, 46]. Model abstractions have been derived from a combination of scientific knowledge, current practice, and reconfiguration needs. Manufacturing-specific scientific knowledge has been obtained from literature and experts in the field of manufacturing automation. Knowledge of generic structures and interaction patterns has been synthesized and adapted from the field of computer science. The validation and refinement process includes prototyping and evaluation on a machine tool testbed, as well as reviews by domain experts (Figure 2.3, Groups A and B), and feedback from participating developers (Figure 2.3, Group C), at each iteration in the development process. In addition, this thesis shows how the developed model or its extensions provide the required reconfigurability.

2.5.1 The experiment – a typical development cycle

A typical experimental cycle (Figure 2.3) begins with a formulation, reformulation, revision or extension of requirements through systematic inquiry by the investigator (myself), involving interaction with the two domain expert groups, A and B. Iteration steps, implemented by Group C, are designed to evolve in scope and functionality, as elaborated in Chapters 3–7. The evolving work product is an architectural framework in the form of class graphs and interface specifications.

Group C members documented the class specifications using a CASE Tool, through which they generated skeleton code in C++. Code in the generated function bodies was added to the extent necessary to construct a prototype application for testing on the machine tool. This code was based on similar prior implementations in the C language, supplied by members of Groups A and B.

Based on these classes, Group C members constructed test applications for running test motions on the machine tool. A key test move that revealed effects of timing variations was a circular path. Results were compared in various task configurations, including a monolithic task, in order to understand the impact of timing variations resulting from multi-tasking at various levels of granularity.

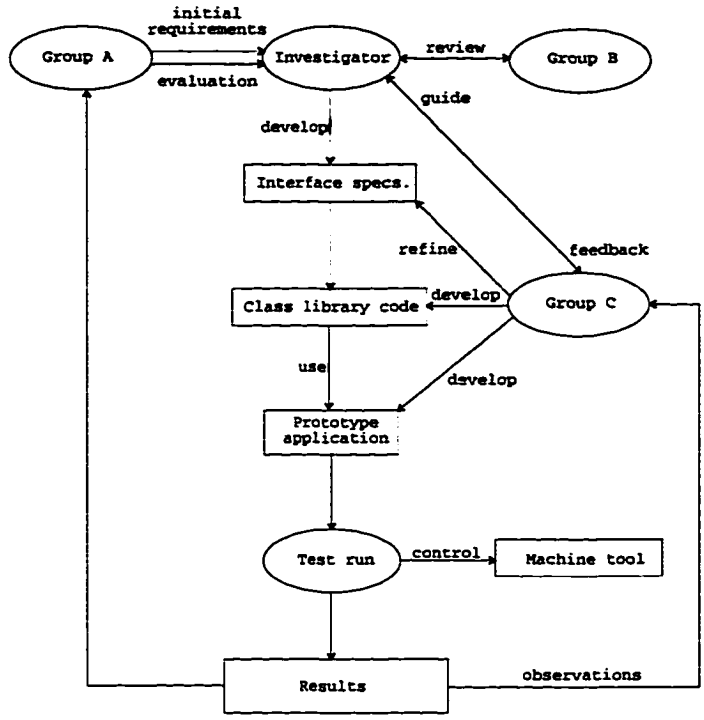

Figure 2.3: Experimental cycle of software development

Process for the software development experiment.

1. Obtain inputs from Groups A and B.
2. Model software for corresponding increment of functionality.
3. Review the software model with Groups A and B.
4. Incorporate the feedback in a revision of the model.
5. Prototype a test application from the model, adding test code.
6. Run test on a machine tool.
7. Analyze the results of the prototyping and test experience.
8. Improve the model.
9. Extend the model iterating the previous steps.
10. Review and evaluate the software development process with the participants.
11. Improve the software process and guidelines correspondingly.

Modeling the domain: The investigator assimilates and transforms Group A-B inputs into an information model, using the object and finite state machine paradigms (Section 2.7), supplemented with other modeling principles and rules (Heuristics 1–5; Rules & Constraints Set 4). The work product is a set of object class interface specifications that describe the corresponding concept, physical object, component, functionality, interaction, or clusters of these objects, including revisions to a prior iteration. For non-trivial behavior, a state machine is created or extended. The model is a combination of the traditional analysis and design phases of a software engineering process. It is developed, maintained, and

documented with the aid of a CASE tool, which also includes facilities to generate skeleton C++ code, including default constructors, destructors, and accessor functions. C++ header files are used as an alternate form of documentation of the class specifications. Group C members are participants in the development and maintenance of the model.

Prototyping a class: When a non-trivial function is encountered, a Group C member proceeds with a prototype implementation, by adding the body of the function to the C++ implementation file generated by the CASE tool. If it involves no additional domain knowledge, a Group C participant creates the code. If domain knowledge is required, the code for the body is derived from some prior implementation (typically a one-shot application) in the C programming language, obtained through Group A or B. This code typically requires significant reorganization to produce reconfigurable software. Group C participants provide feedback of their experience to the investigator and the model or the modeling process is revised as needed.

Prototyping and testing an application: When a logically self-contained cluster of classes is sufficiently described to warrant further testing representative of a common use case in a real-life controller, a participant from Group C develops a test application program including the classes to be used. When enough class-level software is developed to warrant testing with hardware in the loop, some prior offline test application program is extended and tested. For example, in an early cycle, classes to interface with external IO are tested with the external interfacing hardware in the loop. In a subsequent cycle, classes involved in the control of one axis of motion are tested. Classes for coordinating multi-axis motion are tested in a later cycle.

Evaluating test results: The investigator reviews the results and observations at the end of a cycle with Groups A and C, and the implications on the model, with Group B, obtaining feedback to plan the next cycle. The reviews address several general questions about the effectiveness of the software development approach (Heuristics 3).

Q1: Is the process of extracting domain knowledge effective?

Q2: Is the process of generalization from specific initial information effective?

- Q2.1: Is excessive genericity adding significant cost?
- Q2.2 Is the generality adequate?
- Q2.3 How is the degree of generality determined?
- Q2.4 Is the model extensible?

Q3: Does the modeling paradigm suit the domain?

Q4: Do observations fit in well-established theory underlying software engineering?

Note: Effectiveness is evaluated as the *degree* to which general needs and requirements for *reconfigurability* are met, relative to the associated *design complexity*, in comparison with current alternatives.

Heuristics 3: Questions addressed in the evaluation step of the experiment.

Subsequent chapters provide evaluations relative to these questions and other issues that were discovered to be significant during the experiment. Some evidence contradicts prevailing theory-based or intuition-based beliefs.

External validity of the experiment: The research experiment is designed to be representative of an industrial application and its development and implementation environment, in order to make relevant factors dominate over other variables — development conditions are controlled accordingly, extraneous variables, affecting the development effort, are controlled within a range representative of conditions in industry (Appendix A), or kept at more pessimistic levels, to ensure external validity. In this manner, sensitivity to extraneous variables is reduced relative to the effect of the architecture and process.

2.5.2 Participants in the experiment

Evolution of the domain model is a process involving the investigator with three working groups: A (motion and machining researchers at the University of Michigan), B (motion control and other software developers at industrial research institutions including several national laboratories), and C (computer science and engineering students at the University of Michigan).

Groups A and B: They provided general information in several forms, including broad statement of long-range goals in written form, objectives for a particular development cycle, immediate implementation needs or issues to be resolved, and pre-conceptions of the architecture in the form of references. Separate face-to-face meetings and discussions with members of each group yielded more specific conceptual information about the role of some software module and suggested interfaces.

Members of both groups had prior experience in prototyping control systems to meet research and development needs specific to a project. None of the participants had attempted to generalize software components or an architecture for use beyond a single application. The members had varying degrees of belief and confidence in the realizability of such genericity or reusablity. None of the original members was predisposed toward the object paradigm — some had serious reservations about practical difficulties in using it, e.g., availability of integrated suite of tools, availability of skill pool, execution time cost and uncertainty. When we proposed its use, one member with experience in applying the object paradigm in non-real time applications using the Smalltalk language, was added in Group B. Neither group was predisposed to adopt the FSM formalism for specifying behavior of applications or composing them.

Group C: Members of Group C were responsible for documenting and maintaining the class specifications, implementing the class libraries, constructing the test application programs, conducting the tests, and providing feedback of their development experiences, including effort applied and difficulties encountered. The feedback was in the form of an ongoing log, as well as interviews with the investigator. There were fourteen participants in the experiment, distributed across different cycles. All the participants were college students, not having prior professional work experience in the OO paradigm, in the C++ programming language, or in applying the FSM formalism to specify software behavior. Nine of them had taken a junior level course in data structures and algorithms, which exposed them to C++ but did not provide a formal foundation of the OO paradigm. Five

participants had no prior OO exposure, and three of them had learned the C programming language by themselves. The participants perfomed most of the experimental development during academic semesters, mostly applying 8–10 hours per week distributed across 3 or more work sessions. One participant applied an average of 15 hours per week. Five participants also worked in the summer break, applying an average of 40 hours per week over a 4-month interval. The experiment was performed over a 30-month interval. One participant was associated with the experiment for a duration of 24 months, one, for 20 months, and eight, for 8 months, and the rest for 4 months or less. Thus, in comparison to industrial conditions, the skill base and continuity factors in the experiment were lower, indicative of over-estimation in the effort and difficulty reported.

2.5.3 Experimental testbed

The testbed is designed to be representative of the industrial environment (Appendix A), in order to ensure external validity in the experiment. It consists of a hardware platform, a software platform, application development resources , and a machine tool. The runtime control testbed consists of elements used in industrial control systems. Each iteration of the prototyped test application is tested on a real machine tool.

Platform hardware

An on-line testbed, connected to a machine tool (Section 2.5.3), consists of three computers A, B, and C (Figure 2.4), interconnected with point-to-point ethernet links. A (Pentium P100 based) supports the user interface, and various preparatory functions related to the controlled machine and process. It is also used as a development platform and software repository. It has two ethernet links — one connected to the local area network using TCP/IP for non real-time communications outside the controller and one dedicated to B, for soft real-time communications, either using TCP/IP or QNX's network messaging service. B is a XYCOM XVME Intel-486 based processing module interconnected to various input and output modules through a VMEbus as follows: a timer-counter module for input from incremental position encoders, an analog input module for analog signals, e.g., tacho-generator feedback, a digital output module for output to (PWM type) servo-amplifiers, an ethernet link dedicated to connection with A and another ethernet link dedicated to connection with C. B has 16 MB of main memory and a hard disk from which Computers B and C may be booted. C is a XYCOM XVME Intel-486 based processing module, interconnected through a VMEbus backplane to an analog input module for signals from force dynamometers or accelerometers, and an ethernet link to B. C is diskless. The plug-in IO modules have been replaced, removed, or added in various ways to arrive at different test configurations.

In some current industrial applications, the roles of B and C in the run-time testbed are served with modules that plug into the motherboard of A. Since the testbed configuration is less efficient than a state-of-the-art industrial configuration, the resulting observations are an under-estimation of the performance and an over-estimation of the difficulty reported.

Platform software

B and C run under a real-time microkernel-type operating system, QNX. A is set up to run under one of several operating systems — QNX, Microsoft NT, Windows95, and DOS. A commercial C++ class library (Rogue Wave Tools.h++) is available on each computer.

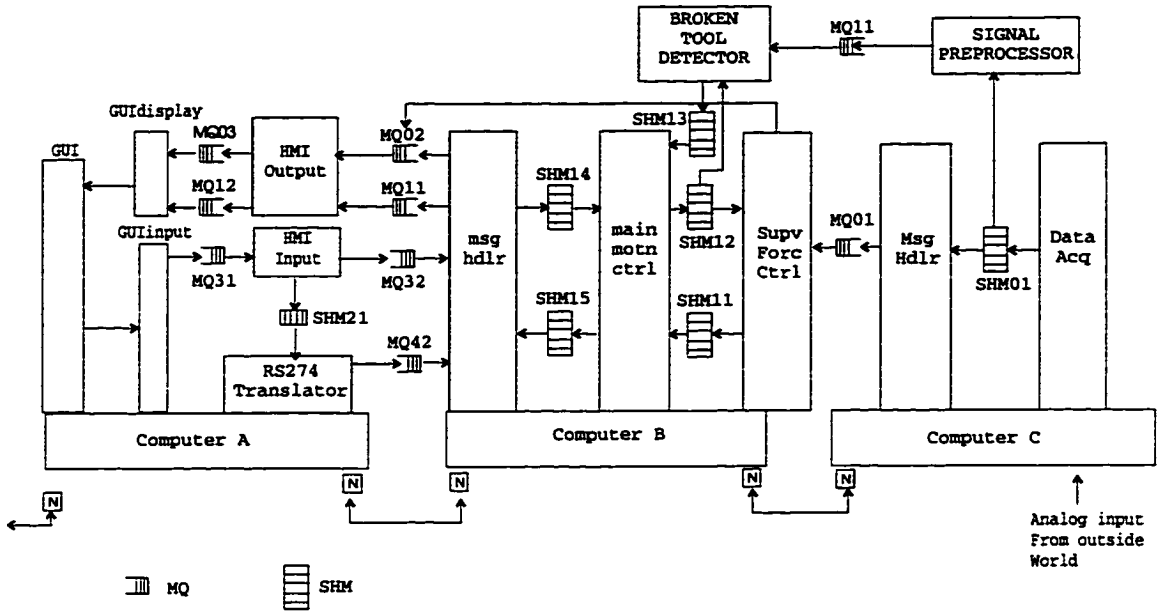

Figure 2.4: Schematic block diagram of a distributed control system testbed.

Resources for application development

Off-line development resources are similar to elements of the on-line testbed described above. The environment on A is replicated on several personal computers to reduce the dependence on availability of computing resources in the development process. The QNX partition in A and other similar development machines have the WATCOM/C++ compiler, development environment, and C++ class libraries, and the TILCON graphical user interface (GUI) development tool.

An object-oriented CASE tool, Rational/Rose/C++ is installed on each development platform, for use under Microsoft Windows or NT operating systems. The analysis, design, and part of the coding phase (generating C++ code header and body templates) of the software lifecycle are performed on the CASE tool. Final stages of code development, test-and-debug cycles, and tuning and testing of auxiliary programs, e.g., benchmarking, profiling, and testing programs are performed under QNX. User modifications made in the generated code can be reverse-engineered through this CASE tool, to update the design model and documentation, provided the code-level modifications are not structural. The graphic expression has helped participants understand class inter-relationships more easily than code listings. The static part of the model can be automatically transformed into the C++ language. This feature, combined with the dialog boxes and class diagrams, has helped reduce mistakes and shorten the time to repair, evolve, and revise the model. Third party C++ class libraries are integrated with our domain model, within the framework of this tool.

Validation on a real machine tool

To validate developed test applications, a small, simple machine tool (named *Robotool*) is used. It has been used by Mechanical Engineers in research on sensing and control. The mechanical researchers were faced with the problem that a different control program was required for each research project — it was not easy to integrate the ideas and results of

the various researchers. Thus, the *Robotool* machine was selected as a testbed with two objectives (i) to allow integration of local research results, and (ii) to evolve and validate the architecture for wider application in future research, as well as in industry.

2.6 The software engineering process

The process developed in this thesis is a synthesis of existing guidelines on domain engineering [52], iterative, incremental development [11], use case driven approach [26], and other modeling and design methods [12,21,47,49]. The foundation for the developed process is a set of organizing principles given in Heuristics 4. Additional rules, constraints, and procedures have been developed specific to the domain of hard real-time systems for the control of machine tools, e.g., Heuristics 1–5.

Rule 1 *Compose a system from orthogonal elements, localizing effects of change, including localization in time.*
Rule 2 *Correspond with objects and concepts in the application domain.*
Rule 3 *Use the scientific and engineering knowledge in the physical system.*
Rule 4 *Abstract commonalities in features across objects in the domain.*
Rule 5 *Organize commonalities into generalization hierarchies.*
Rule 6 *Organize concepts into larger logical units.*
Rule 7 *Seek the simplest representation of the real-world complexity.*
Rule 8 *Facilitate comprehension and learning, considering the available industrial skill pool.*

Heuristics 4: Machine control software organizing principles, rules, and constraints.

2.6.1 The perimeter of the domain

Rule 3 prescribes that the outer limits of the domain should be within the mature body of engineering knowledge about computer controlled motion. This knowledge is available from the perspectives of manufacturing [27], robotics [28], and computer science [21]. An early-stage domain analysis of automotive machining automation had been performed [5,8,9], serving as Step S1 of Heuristics 5. Results are also available from an earlier project to develop specifications for an open systems architecture standard [7,35,36,40,48], serving as a point of departure. However, those specifications were incomplete and not validated.

2.6.2 The initial iteration

The machine tool in the experimental testbed served as a specific initial case (Heuristics 5, Step S2). This step, along with the reference architectures mentioned above, serves as the "top down" aspect of a traditional software engineering cycle, seeking conceptual integrity. However, even this initial case is not a single application in the traditional sense,

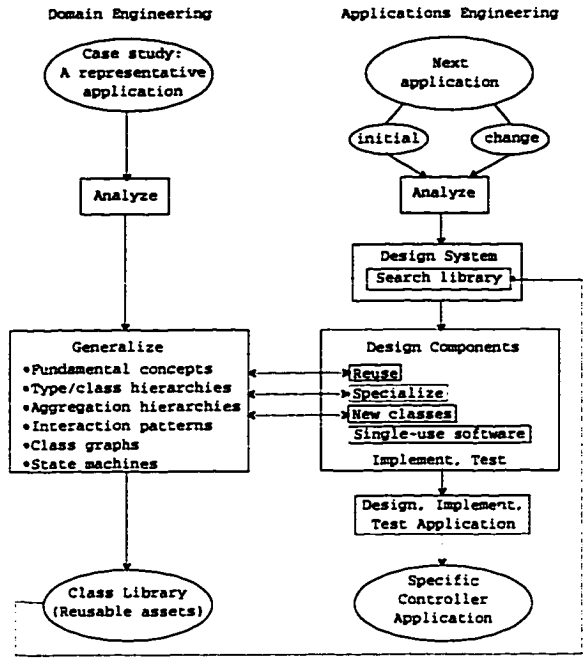

Figure 2.5: Domain model based application development process

S1: Proceed through the first iteration of identifying the domain, using Heuristics 1.

S1: Select a representative (possibly synthetic) application case.

S2: Reduce the multiplicity of similar objects.

S3: Select the most common use case [26].

S4: Focus on the essential or critical functions.

S5: Proceed through a software development cycle from analysis to prototyping.

Heuristics 5: Initial iteration of domain engineering.

because it could have different physical configurations and different control configurations. It is a case of "multiple tops" where all the tops cannot be identified initially. A strict "bottom up" approach would require starting from a set of basic data types and instructions of a typical computer system and building upward. However, how would a software engineer determine the necessity and sufficiency of the "language" and architecture thus built-up? Validation would require some concrete or abstract or simulated "top" (Heuristics 5, Step S2). Since the scope of the functionality of the chosen case is still too large to tackle in one iteration, a smaller initial iteration step is devised, using Heuristics 5, Steps S2-S4. The multiple-axis experimental machine tool case can be simplified to a single-axis system, since an axis can be operated and tested independently, without requiring other axes. This is not an unrealistic simplification, because a large number of real machines with only one axis of motion exist in automotive manufacturing, e.g., stations in a flexible transfer line that perform hole-working operations with a multiple-spindle head specific to each workpiece hole-pattern. Referring to the data flow shown in Figure 2.2, the first iteration is focused on developing the axis object. The other objects are rapid prototypes to obtain initial experience in the flow of data from the user interface to the axis — they could be thrown away.

One scaling-down criterion is to reduce the commitment of effort in the first iteration to the smallest possible amount from which the major architectural concerns could be resolved. Another scaling-down criterion is to avoid misleading distortion. An early concrete proof of concept is needed, e.g., a physical demonstration of motion control, where performance could be observed and discussed. With the concurrence of Group A, one axis of motion is selected as a goal for the first iteration. It allows for the modeling and representative prototyping of key issues in the overall architecture, as described in Chapters 3–4.

2.7 Underlying formal model

This project has used the object-oriented paradigm to express the domain model, based on the rules and constraints of the *OMG* object model [42] and the notation of the C++ programming language.

The choice of the modeling form has been controversial for this application domain, particularly for hard real-time controls. Criteria at one end of the spectrum, are *logical unambiguity and precision*, driving toward a formal model, and criteria at the other end of the spectrum are *ease of comprehension and implementation*, requiring a seamless transformation from a model to an implementation. There are no commercial tools available for meeting these criteria economically in this application domain. *Extensibility* of a specification is another important criterion for adaptability to changes in requirements. *Portablity of the specification*, or independence from an implementation language, or freedom to select different implementation languages for different modules is also a consideration. It may be evaluated in terms of the relative cost of porting the specification to different implementation languages. Next is a review of the alternatives considered (summarized in Tables 2.4–2.5) and a justification for selecting the object model for this application domain. Additional evaluative insights gained through using the object paradigm in the experiment are reviewed in Section 3.12.

2.8 Comparison of alternatives in modeling form & notation

The alternatives are narrowed down from both extremes — popularity and formality. The most common language for communicating requirements specifications – the English language – is inadequate, by itself, in meeting the criteria given above, because it is too ambiguous. The most common programming language in real-time controls, C, is also inadequate — it makes an architectural specification more obscure than the English language, because of the low-level, implementation-oriented expression, rather than application-oriented expression.

Formal specification languages leave semantic gaps between the real-world requirements and the formal model and between the formal model and the implementation language. In the first case, there is a loss of information and uncertainty in the transformation from human expression to the formal language, because a formal specification language, by itself, does not make it easier to capture and express concepts natural to the application domain. It is difficult to validate the formal specification against the real requirements of the application domain, without a number of implementation cycles. In the second case, uncertainty is introduced in the transformation from the formal model to an implementation language. It is difficult to establish that the transformation tool is more correct than the more popular language compilers in commercial use, particularly, in the implementation-specific aspects

of the tool. With its much lower usage volume than the leading language compilers, the transformation tool will inherently have a longer debug interval and higher amortized cost.

Reducing the semantic gap in modeling requirements

The object-oriented paradigm reduces the semantic gap between the model and the engineering concepts used in the design of manufacturing machines.As shown in Table 2.3, by applying this paradigm, we have built a domain model that provides more expressivity for the specification of more complex concepts in the application domain. Thus, the domain model serves as a specification language that requires fewer mental transformations and fewer manual entries than the most popular programming language (C) used in this domain.

How formal is the object model?

One weakness of the OO model is its lack of a sound mathematical foundation. We assess the relevance of this weakness by comparing our domain modeling paradigm with the Z notation [53] — one of the more commonly accepted formal specification languages for real-time software. The Z notation is based on first-order logic and set theory. It relies on a combination of mathematical notation and prose. The notation is used to express *types*, including a special type called *schema*, *predicates* on these types, a *schema calculus* for defining schema expressions, and a description of *state machines*. The Z schema corresponds to the C++ classes used to build our domain model. The Z schema inclusion corresponds to specialization in the object model. The Z type is an extensional concept (set membership of included elements), in contrast to the more compact and "pure" intensional concept in the object model. Our domain model specification of the dynamics of behavior (flow of control) follows an extended finite state machine model [24] (FSM). It identifies the state of an object symbolically, but we do not have as compact a notation for the value of an object after an operation. The object model also lacks as compact a notation as Z for expressing predicates.

Z Notation	Object model
Types, schema	Classes
Type definition is extensional	Intensional
Schema calculus	Generalization-specialization; aggregation
Predicates	No constructs above programming language
FSM notation	No constructs. Class structure built.

Table 2.4: Comparison of a formal modeling language with the object model

Other pertinent formal specification approaches include *SDS* [14] and *ROOM* [49]. Both of them follow a similar approach, although with different notations and tools. They use the object model for the static apsects and a finite state machine for the dynamic aspects of an application. They use a graphic notation for convenience of human understanding. They have the capability to transform the specification into C++. They have been used in building large real-time systems. They specify a number of explicit constraints on a system architecture, in order to simplify correct application design. In this respect, they offer more support than the Z notation for building practical systems.

Thus, considering the criterion of disambiguating requirement specifications, the foundation of our domain model is comparable to the Z notation. Therefore, we do not consider

other formal modeling alternatives in further evaluation.

Implementability considerations

Although there are a number of modeling languages, model libraries are in formats proprietary to the respective tool suppliers. They cannot be mixed in a single application. The *OMG's Interface Description Language* (IDL) is intended to be a vendor-neutral medium, but there are very few commercially available transformation tools. Rational Software Corporation has proposed to the *OMG* and a group of other industries a *Universal Modeling Language* (UML). We have converted to a recently released form of the UML in our project, using the Rational Rose/C++ object-oriented CASE tool. The UML model can also be transformed into Java and IDL. Unfortunately, the CASE tool lacks model libraries needed for building models of real-time systems similar to libraries available with SDS and ROOM. Thus in our project we had to build our own models for such generic entities as finite state machines, periodic processes, message ports on these processes, and interprocess communication mechanisms (IPCs) connecting ports between different processes. In exchanging information for experiments in our project, Group B members (spread across multiple distant sites) found it easier to understand and evaluate the specifications in the form of C++ header files rather than the modeling language of the CASE tool. Group B members at one site were able to transform the C++ class descriptions into IDL. Other group members at a third site were able to transform the specifications from IDL into Java.

Factor of comparison	C	C++	IDL	Z
Disambiguation of requirements	L	ML	ML	ML
Transformation into implementation language	H	H	M	L
Extensibility	L	H	H	H
Portability of specification	L	M	MH	MH
Current use in real-time controls	H	M	L	L
Common knowledge	H	M	ML	ML
Evaluation codes: (L) Low (ML) Medium low (MH) Medium-High (H) High.				

Table 2.5: Comparison of interface modeling languages

Extensibility considerations

Mature industrial practice in real-time controls relies upon the C programming language for interface specifications. Operating system and network services also define their interfaces through C. For compatibility and ease of integration with existing specifications, development environments, and implementation environments, C was considered. However, the extension and adaptation of a C interface is more costly than an interface described in the object model form, because the latter supports a generalization hierarchy. For example, a class can be specialized by restricting the domain of its members or the domain of its member function parameters. A class can also be specialized by adding members. C++, being a superset of C, serves as a good bridge between the legacy interfaces described in C and the object model.

2.9 Characterizing task interactions space

Based on existing knowledge, Figure 2.6 provides a characterization of the types of inter-dependence that may be required in an application, and how these characteristics affect design complexity. In the dimension of time-constraints on the completion of an interaction, the application requirements become simpler as the allowed time increases and the time-constraint becomes softer. Applications served by current controllers exhibit a wide spectrum of time constraints on various interactions within the same system. At one extreme are the servo-sensor loops with very short and strict sampling time intervals. At the other extreme are queries from the human or a remote computer system, which could be served on an "when time available" basis. This research is focused on the domain of hard, short time-constraint cases. It develops architectural constraints to meet the needs of these cases without unnecessary design complexity.

Considering the different types of interaction, we find that most of the functionality for automated operation in current machine tool controllers falls into a very simple pattern of single producer single consumer type of data transfer. Most of the other interactions can be handled asynchronously. In the few cases where a chained dependency may be involved, the dependency is sequential (serial). In this study, we have found that the general needs identified in Section 2.1.2 can also be met within these interaction constraints. Therefore, we exclude from the scope, more general interaction patterns that introduce more design complexity.

2.10 Architectural design granularity alternatives

Given the requirements for closeness of interaction, reconfigurability in scale, and behavior, what architectural granularity should be considered? Group B focused on application (passive object) class specifications, independent of any execution model. In our architecture, we considered multiple levels of granularity, described next, in order to meet the general needs identified earlier, without unnecessary increase in design complexity. It has been a controversial issue in both Groups A and B. We describe the choices, with examples from current practice, and clarify the issues next.

Command language interface: Examples of command language interfaces are the EIA RS274D standard for numerical controllers, the VAL+ language for robots, and the FADL being developed in the OSEC project in Japan. Command languages were intended for application to a complete control system, known as a *configuration* in IEC 1131. If a "complete" control system is viewed as a "component" of a larger multiple-controller system (Figure 2.7), the command language serves as the universe of discourse (communication). When the functionality of a system is decomposed into loosely coupled components, e.g., A, B, and C in Figure 2.7, each of these components would need to support only a subset of the total vocabulary of the system. Such subsets may be devised specific to the needs of an organization, a project, or an application.

- Type of time-constraint on completion of interaction.
  - Length
    - Shorter [More complex]
    - Longer [Simpler]
  - Strictness
    - Hard time limits [More difficult]
    - Soft time limits [Simpler]
- Type of interaction.
  - Asynchronous [Simplest]
  - Synchronous
    - Single round
    - Multiple rounds
      - Lineaer sequence
      - Acyclic graph
      - Cyclic graph [Most complex]
- Type of dependency.
  - Sequential
  - Acyclic graph
  - Cyclic graph
- Length of dependency chain, and variability in length. [Small => simplicity]
- Number of inter-dependent items. [Small => simplicity]
- Degree of exception handling and recovery.
  - Stop; save state; shutdown. [Easier]
  - Semi-automated return to system startup condition. [More difficult]

    ⋮
  - Automated return to resume operation. [Very difficult]

Figure 2.6: Characterization of task interactions by design complexity.

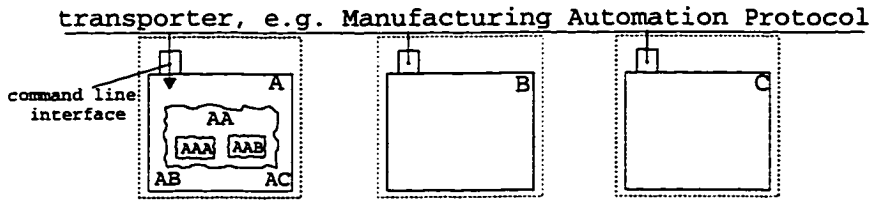

Figure 2.7: Command line style interaction across programs in execution

Data exchange interface: A component can produce or consume data in the specified format, e.g., the ISO STEP standard to exchange product description data, or the communication objects in the *OSACA* program. IEC 1131 describes it as a data acquisition interface, which maps into ISO/IEC 9506-5 Read service Variable Access Specification and Exchange Data service. However, components may be preprogrammed to alter their behav-

ior (known as parametric control in IEC 1131), based on the value of the data, as possible with the ISO/IEC 9506-5 Write service.

Major executable component interfaces: An example from the document processing domain is the exchange of information across a wordprocessor, a spreadsheet package, and a graphics tool. Each component is an independent thread of control. One component can exercise limited control on another, e.g., for starting it, making it find a file, and making it reformat and serve the information. The inter-component communication interfaces (Figure 2.8) are similar to the IEC/ISO 9506-5 Event Notification service and Program Invocation and Event Enrollment objects. In the example figure, AA, AB, and AC can inter-operate in closer temporal copuling than components A, B, and C of Figure 2.7.

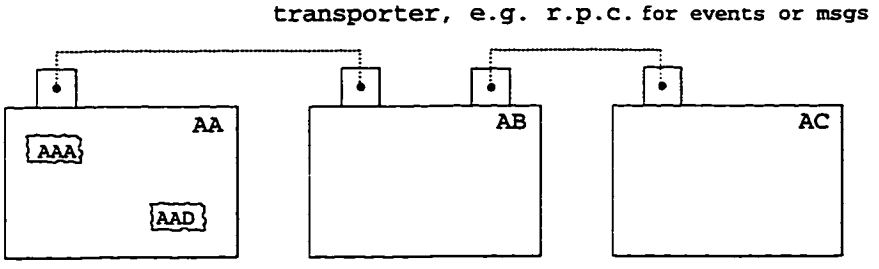

Figure 2.8: Modularization into coarse-grained executable components

Fine-grained executable components: In the extreme case of designing for change and maximizing scheduling flexibility, each function, particularly an IO function, is placed in a separate executable component, as in the Naval Research Laboratory Cost Reduction Method [18,43,44]. Events are used for inter-component synchronization. As each component, AAA–AAD does less work than the larger components, AA–AC, of the previous case, their execution times tend to be smaller, interleaving in an execution schedule is easier, and therefore temporal coupling can be closer.

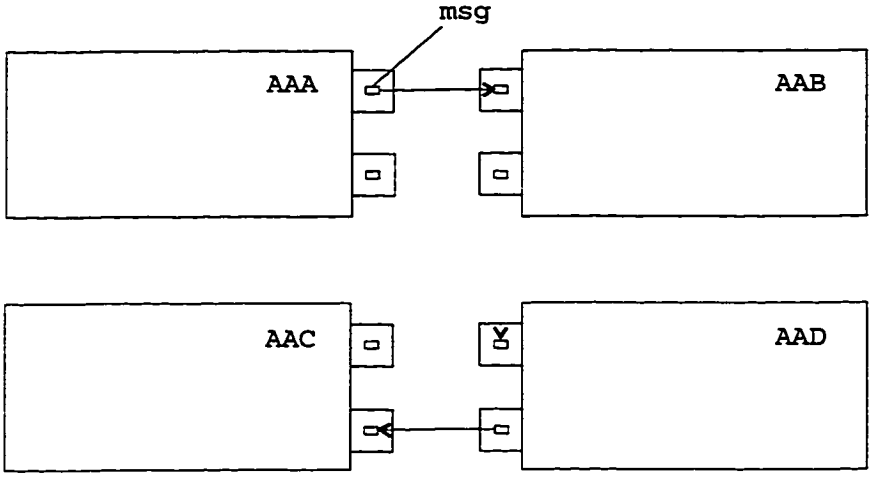

Figure 2.9: Modularization into fine-grained executable components

Passive application objects: When an executable component, such as AA–AC mentioned earlier, employs the services of other objects, e.g., AAA–AAD in Figure 2.10 internal

to it, the latter (known as passive objects) will be finer-grained than the executable component using them. The ISO/IEC 9506-5 **Named Variable** object is a simple example and the **Event Enrollment** object is a more complex example. As interaction amongst AAA–AAD does not require inter-process communication services from the OS, their interaction design process is simpler and their execution time is lower and predictable.

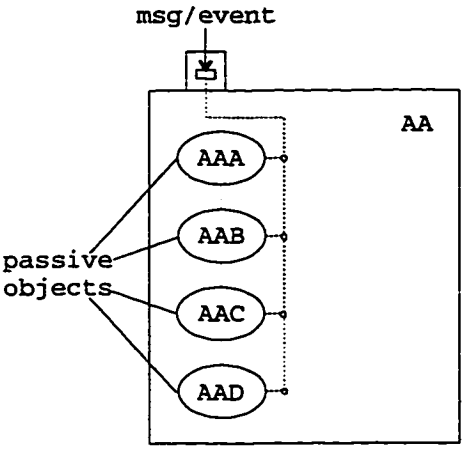

Figure 2.10: Modularization into passive components

Is the design freedom of structuring tasks detrimental, i.e., does the associated complexity and cost outweigh the benefits? This question is analyzed next. Without this scope of granularity, it would not be possible to meet General Needs 1–8 identified in Section 2.1.2. Specific examples are given below, with an explanation of the supporting theory.

2.10.1 Consideration of data transfer efficiency

General need 2 is satisfied most efficiently when functions with close temporal coupling, can be included in the same task. Consider three cases of data transfer across functions in programs allocated to run on the same computer — *intra-object; intra-process; inter-process.* Following is an order-of-magnitude comparison of the data transfer times and the variation in these times for the three cases. The comparison given above shows that the time-penalty of inter-process communication is significant. The secondary effect of variation in timing is an even worse penalty, as discussed next.

c: Best case time to access a variable within the same object. [Approximately 1 memory access cycle].

vc: Variation in c over all accesses. [Insignificant].

c1: Best case time, as a multiplier of c, for data transfer through an accessor function of another object in the same process. [Approximately 2].

vc1: Variation in $c1$, as a multiplier of c, over all accesses. [Approximately 1].

c2: Best case time, as a multiplier of c, for sending data as a message or event to another process ready to run on the same computer. [Approximately 100].

vc2: Variation in $c2$ over all accesses. [Approximately 100].

2.10.2 Consideration of application design complexity

When different objects interact in close temporal coupling, with their actions having the same priority and periodicity, placing the objects in the same process (address space) requires smaller and simpler interaction code — function calls within the same name space, as opposed to remote procedure calls (*rpc*) or proxy object services.

Timing uncertainty: Furthermore, when *temporal distance* is closely constrained, as in the case of servo control of high precision high speed motion, the additional timing variation introduced by cross-address space interaction increases the complexity of the design process.

Type-checking: The current technology of programming languages supports automatic checking and matching of user-defined data types within a single program, including pre-compiled objects used in that program. This is a very useful and effective protection against type mismatch errors, which are very common when modules are produced and supplied by a variety of sources. Commercial compilers for the more commonly used languages, e.g., C++, are relatively inexpensive. However, there are no such commercial tools for type checking and matching for data transferred across programs (i.e., name spaces or address spaces). The supporting system services transport messages merely as sequences of bytes. The application is responsible for the semantic validity of a transferred unit of information, rendering the application more error prone and increasing the development cost. Thus, the granularity approach in this architecture enlarges the configuration design space, for a given level of design complexity and cost.

Comparison factor	Level of granularity				
	1	2	3	4	5
Integratable?	yes	yes	yes	yes	yes
Function reconfigurability	low	no	medium	high	high
Design complexity	low	low	medium	high	medium-high
Communication efficiency	low	medium-low	medium-low	medium-high	high
Response speed	low	no	medium-low	medium-high	high
Level of granularity codes: (1) Command-line (2) Data exchange (3) Coarse-grained executable (4) Fine-grained executable (5) Passive object					

Table 2.6: Comparison of alternatives in architectural granularity

Application developers can limit granularity exposed: Suppose that the majority of industrial applications will be satisfied with a much narrower configuration space, not requiring interaction across modules in very close temporal coupling. Then, it would seem that the overhead of learning how to use this architecture would burden all these applications without adding value to them. However, users of this architectural model still have the option of prepackaging larger modules, executable modules, and configurations, and limiting the interface to suit their respective application domains.

2.11 Summary

We have synthesized a process to develop a domain-specific software architecture for reconfigurable machine tool controllers. The architecture-development process is based on an emerging software development technique known as domain engineering, and the architecture itself is based on two paradigms — OO, for modeling static aspects, and FSM, for modeling the dynamic aspects. We supplemented the known techniques with additional procedures, specific to the domain of machine tool control. Recognizing that the design of machine tools and their controllers are based on established engineering principles, our procedures systematize the process of extracting that knowledge and using it to organize machine control software. Early stages of the process are exercised on a specific case, synthesized to incorporate a number of long-standing needs. Our process deliberately over-generalizes the requirements manifested in the synthetic case, in order to expand the domain at the early stage and explore the implications on development effort and difficulty. This exploration is performed by exercising the process in a series of software experiment cycles, and recording experiential observations pertinent to evaluative questions (Heuristics 3).

Our generalization of needs and requirements is supported by studies conducted in past projects, elsewhere, to develop specifications for open system architecture standards. Several schools of thought in software research support the notion of starting a software project with a wider base of requirements than it specific target, e.g., investment in front-end analysis to identify the key classes or data types in a domain. The history of software development clearly attributes "missed or changing requirements" as the most dominant factor of cost and delay in the software lifecycle. Therefore, our approach attempts to stretch the envelope. We assume that cost of computing resources is rapidly decreasing relative to cost of reconfiguring control systems and lost-opportunity cost of delays (Appendix A). However, the cost-effectiveness of the resulting wider scope has been controversial. The software experiment is intended to identify the more significant issues underlying these controlversies and to clarify them. The following chapters describe the evolution of the architecture in several stages, and evaluate the development experience at each stage.

CHAPTER 3

Axis motion software — static aspects

The architecture for software to control a reconfigurable axis of motion is a microcosm of architectonics required for a RMS. It is broadly driven by current controller functionality and the reconfigurability needs identified in Chapter 2 (General Needs 1–8). Intermediate evolution steps are defined by derived needs and requirements identified in the course of evolving the axis software model. Prototyping cycles are designed for partial validation, or resolution of some uncertainty. The model is grown in increments of relatively small steps. The course of software development is reassessed and corrections made as a result of the learning experience in each cycle (Figure 2.3). Research experiments (Section 2.5) track the effort, difficulty, and progress in each cycle.

The early stages focus on the core function of an axis, but the model is designed for extension, e.g., different types of control, monitoring, diagnostics, ease of setup, calibration, tuning, and other forms of maintenance, without compromising the integrity and correctness of the core function.

In spite of the long history of numerical control of machine tools (introduced in 1952) and computer control of industrial robots (introduced in 1961) [27], this degree of extensibility, reconfigurability, and customizability in axis control has not been possible, largely due to concerns of losing integrity, timing correctness, and maintainability of the software.

Chapters 3 and 4 describe an evolvable software model for one axis motion control in two parts. Chapter 3 focuses on the earlier stages of the software process, developing the static aspects of the model, whereas Chapter 4 describes its dynamic and execution aspects, including architectural constraints to facilitate application design. Section 3.1 introduces the controlled axis of motion and a schematic of its basic control. A structural overview of the axis model (Section 3.2) is followed by interfaces to describe the device (setup, kinematics, and dynamics) and interfaces for the servo control cycle (sensor interface, actuator interface, and setpoints interface). Sections 3.10–3.11 describe subdomains (measures and units; space and kinematics) discovered in the process of modeling these interfaces. Being more generic, these subdomain models, are more widely applicable. Sections 3.12–3.13 evaluate the application of the object paradigm and the domain-specific abstractions, and Chapter 4 completes the evaluation and conclusion of axis software modeling.

3.1 Overview of the axis motion domain

An *axis* of motion is an element of a kinematic mechanism that provides motion with precise continuous control of position and velocity in a single degree of freedom (DOF),

where DOF refers to the minimum number of independent variables required to specify completely the motion of a mechanical system. In the conventions of the righthand rectangular coordinate system, the six independent variables consist of the three translations along the three axes of the coordinate system, X, Y, and Z, and rotation around each of these axes respectively. Each axis of motion is independently controllable. There are two kinds of motion commonly used in robots and machine tools — a *translational axis* provides and constrains motion to translation along a fixed axis, and a *rotary axis* provides and constrains motion to rotation about a fixed axis.

The axis definition given above is synthesized from the conventions of common usage in the field of programmably controlled kinematic mechanisms, since there is no explicit common definition across the different types of manufacturing equipment, (NC machines, various robotic mechanisms, inspection and measurement devices, handling mechanisms, and combinations and specializations of these devices). For example, precision requirement is left implicit in the usage context — it is assumed that a five-axis NC machine will follow a specified continuous path more accurately than a robot with five articulated joints, and this robot will move more accurately than a handling mechanism. Current standards make no provision for explicit specification of the required accuracy of motion.

Despite the existence of mature knowledge in the field of manufacturing engineering, there is no common basis to express this knowledge for an axis of motion in a form that lends itself to computer automated exchange of information with semantic content about its motion characteristics. Lack of a uniform language for describing motion in current practice adds to the cost and difficulty of computer integration of different manufacturing devices. Thereofore, an axis of motion must be described in a unified scheme that is more general than the view in any single subfield of manufacturing equipment mentioned above.

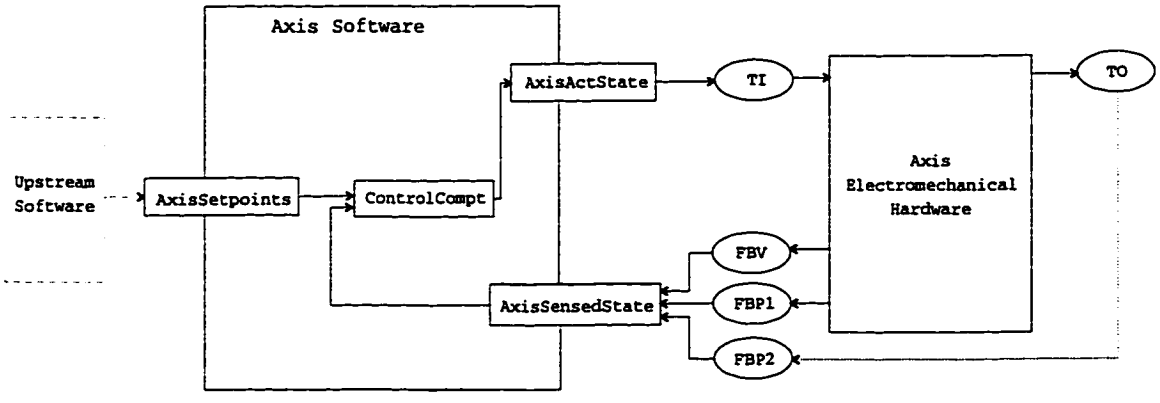

Figure 3.1: Overview of axis servo-control software model

Overview schematic of an axis: Figure 3.1 is a schematic block diagram of a servo-control model, showing the key abstractions in servo-control of motion. The final output of the axis is the translational motion of its work-point, depicted by the symbol, TO. The ControlCompt object produces an output (reflected at the workpoint) to the AxisActState object which converts it to the corresponding actuator signal units and sets the signal to the actuator at TI, a terminal for signal connection. Feedback signals from sensors, e.g., at terminals FBV, FBP1, and FBP2, are acquired into the AxisSensedState object which converts them into the corresponding workpoint units for use by the ControlCompt object.

3.2 Organization of axis control software

Axis software (Figure 3.2) is organized in accordance with the approach described in Section 2.4. The Axis model is not specific to any execution model. For example, an axis of control may be set up as an independent periodic process, or as part of a process that includes other axes, or as part of a process that also includes coordination of their motion, and coordination of other tasks. To facilitate formulation of execution structures, the specification of behavior and control flow, through a finite state machine (Figure 3.2, axisFSM), is separated from the specification of the various service-providing objects (Figure 3.2, Resources) of an axis. Axis resources are organized around the axis object (Class Structure 3.1), which aggregates software components that perform the control (controlCompt), components that provide the device (physical axis) descriptions and settings, and components that provide the interfaces for setpoints (axisSetpoints), feedback (axisSensedState), and output to the actuator (axisActState). The jogHome component transforms user functions for jogging and homing moves into the corresponding sequence of axisSetpoint settings.

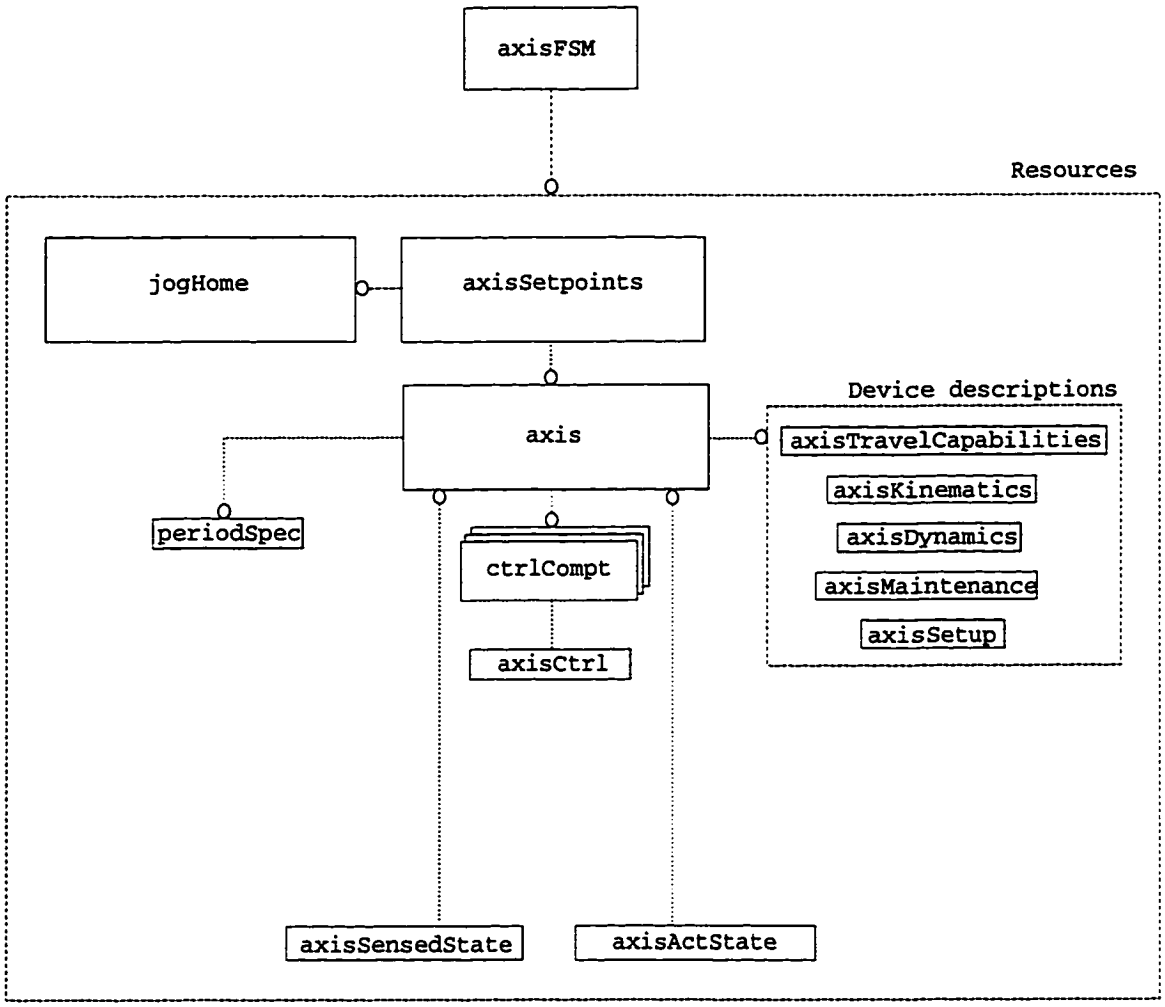

Figure 3.2: Organization structure of the Axis model

The Axis class is abstract — its members describe features common to all axes of motion

in machine tools, robots and other programmable, servo-controlled motion devices, but no instance is created from it.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
CtrlCompt *ctrlCompt
AxisCtrl *axisCtrl
AxisActState *axisActState
AxisSensedState *axisSensedState
AxisSetup *axisSetup
AxisTravelCapabilities *axisTravelCap
AxisKinematics axisKinematics
AxisDynamics axisDynamics
AxisMaintenance axisMaintenance
AxisFSM *axisFSM

Accessor functions for following data members:
CNTRL_MODE controlMode //enum - position,pos-vel; direct-force
MOT_OBJ motionObjective //enum - min-jerk, min-time

Other member functions:
void processServoLoop()
Boolean checkPrecondition()
Boolean checkInPosition()
void resetEmergency()
void resetHold()

Class-structure 3.1: Interface of the Axis class

The responsibilities common to all types of programmably controlled axes (Class Structure 3.1) are the flow of control through various modes (specified in AxisFSM), and, under continuous automatic control, the most commonly used mode, to process the servo motion control loop.

Classes Axis, CtrlCompt, AxisCtrl, AxisSetpoints, AxisSensedState, AxisActState (or their specializations) participate in the continuous control of motion. AxisSensedState, which is the designated state information server for the axis, also saves state information for precondition-checking and recovery from an interruption. Other classes (Figure 3.2, device descriptions) support the main responsibility of the Axis class for some preparatory or recovery step, as follows. Classes AxisSetup and AxisTravelCapabilities support the setup of an axis. Classes AxisKinematics and AxisDynamics suppport configuration and initialization of axis mechanics related information, including the plant model for axis control. The AxisMaintenance class is intended for future extension of the model. External loads and disturbances are not included in this axis model, but are treated to be a part of the external process model. By distributing the responsibilities of the axis across members focused on different aspects of preparation and operation, the effect of a change is localized, reducing the effort and likelihood of errors resulting from the change (Rule 1).

There are two main specializations of Axis — TranslationalAxis (Class Structure B.2) for translational axes of motion and RotaryAxis for rotational axes. This specialization is based on restricting the domain of the data supplied to, returned by, or processed by various axis functions (Section 3.10). Instances are created from one of these two subclasses or their specializations.

3.2.1 The servo control loop function

The main function of an Axis is as follows: Given some setpoint(s), i.e., input(s) and the current state of the controlled axis of motion, perform a transformation, which produces some output (setpoint) for the actuator, to reach the setpoint. At this level of generality, there is good agreement amongst users of various forms of single-axis motion control about the core function of an axis, identified as processServoLoop() in the Axis class. It gets the desired setpoints from axisSetpoints and the specified feedback, from axisSensedState, supplies these parameters to the transform function of the selected CtrlCompt object (Class Structure 3.2), receives the output from it and sets it in the axisActState object. In order to perform continuous motion control, this function must be executed every servo loop interval.

Novelty: The novelty of the model lies in its modularity – the function processServoLoop() is a composition of five services, each of which isolates a change that can occur independently. For example, if a different control mode is required, the corresponding CtrlCompt is replaced. If a new servo-control algorithm is to be retrofitted, the axisCtrl object is replaced. No change in processServoLoop() is needed. No such modularity exists in current practice. Most commercial motion controllers do not even allow this level of access. The few that allow changes at this level require more programming effort because they are not adequately modularized.

3.2.2 Checking preconditions

The function checkPrecondition() is provided to test if all the preconditions for execution of the next servo loop are satisfied, e.g., the axis must not have violated its travel limits. An integrator-user may specify the condition to be evaluated by this function.

Contrast with current practice of precondition checking: CNCs have built in monitoring of certain abnormalities such as excess following error. Some of them allow the system integrator to adjust the threshold values. However, no motion control product allows the flexibility to define the preconditions that must be checked at every cycle of the servo loop. This axis model is novel in isolating the function checkPrecondition() and supporting its customization.

3.2.3 Status reporting

The function checkInPosition() tests if the axis has reached the position specified in the previous setpoint or met the condition specified for successful completion of the move. Position need not be explicitly specified. For example, if the move is specified as *touch*, i.e, "move until contact is sensed", then for that move inPosition is interpreted to mean that the condition of completion of the move has been satisfied. It sets the result in the data member inPosition in axisSensedState. The caller or client of the axis object can get the inPosition state at the appropriate time to test if the axis has moved as specified.

Architectural advantage: The Boolean value reported by this function is compact, processed knowledge about the state of the axis, in contrast to the traditionally reported *actual position*, thus reducing the burden on the axis client. For example, in a point-to-

point move involving multiple axes, the motion coordinator checks this variable after issuing the last setpoints of the move. When all axes under its coordination have reported that they are in position, the coordinator deduces that the specified move has been completed. If the motion coordinator were to compare actual position and setpoint (desired position) for each axis, it would increase the volume of data and code handled by the coordinator. Furthermore, this code would have to change when the coordinated axes change. In our axis model, the responsibility is delegated to individual axes, where the code can be reusable as part of the class library code. The function is provided in the Axis class rather than in one of its members, because only the Axis class has visibility of setpoint information. The result is set in the axisSensedState object.

Architectural issue: The function checkInPosition() and data member inPosition are not useful in all types of motion, e.g., when continuously following an edge. Yet the cost of carrying the interface definition and storing the data member is built into the architecture. We justify their inclusion in the basic architecture with a qualitative evaluation of three cost-benefit factors. First, motion in which position is controlled explicitly or implicitly is by far the most common mode of usage — structural support is built in for the most common use cases (without limiting extensibility to meet other needs), because the accumulated benefits are large. Secondly, after the initial architectural design, the cost of carrying the design definition and the data member is primarily the cost of on-line space and the effort required to understand the design. These costs are insignificant in this case. Thirdly, maintaining differences in architectures for different cases is much more costly, in off-line space, as well as effort required to understand the architecture.

3.3 Modularization of servo control software

There are a number of control schemes and modes, e.g., *position, velocity, acceleration, and direct-force*, in common use for axis motion control. The architectural support must be quite generic to accommodate the more common cases. Yet, it should provide sufficient constraints to minimize integration-stage effort, e.g., errors of interface mismatch. Servo-level functions are also very sensitive to execution time costs and time variation. Thus, there is a need to balance the tradeoffs across control design flexibility, integration cost, and execution-time cost. The Axis class includes a polymorphic control component (Class Structure 3.2) that can be specialized to provide the various control modes or schemes mentioned above. It uses AxisCtrl — more reusable and polymorphic (Class Structure 3.3) — to implement the control law, algorithm, or rules. Through specialization of the Axis class multiple control components may be included. Figure 3.3 shows a simple case, with one ctrlCompt and one axisCtrl object active in a particular configuration of axis software. The axis– >processServoLoop() function calls the ctrlCompt– >transform(...) function, supplying it references to the axisSetpoints, axisSensedState, and axisActState objects. The ctrlCompt– >transform(...) function calls the axisCtrl– >calControlCmd(...) function, supplying it references to the relevant objects, axisSetpoint, actualPosition, actualVelocity, and axisOutput.

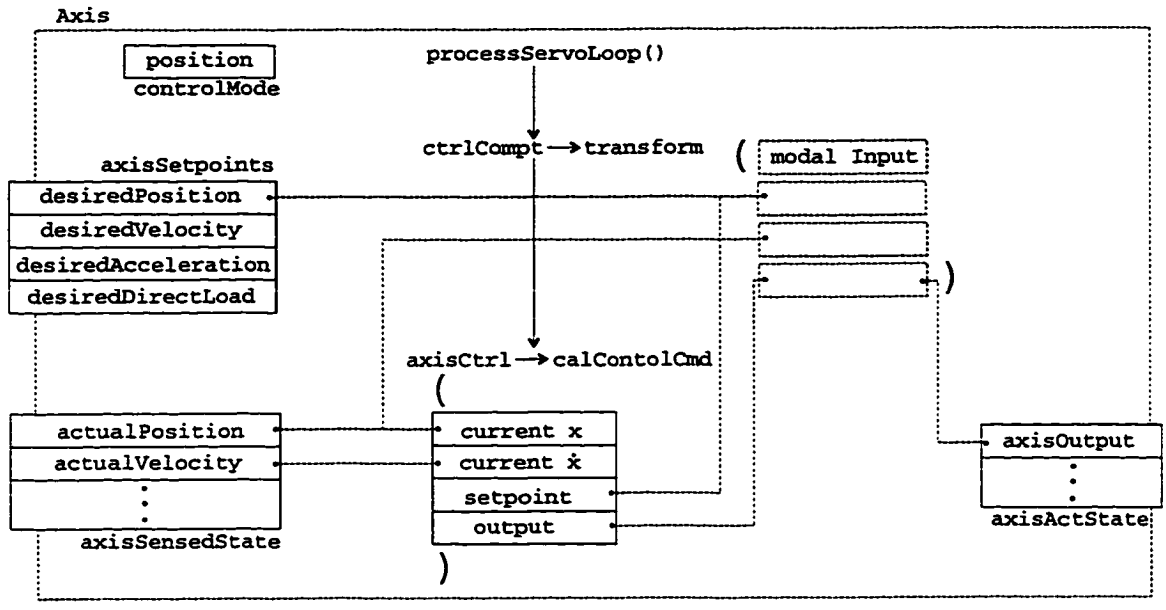

Figure 3.3: Object interactions in processing a servo loop

3.3.1 Reconfiguring control strategies

The CtrlCompt class is abstract. To create an instance, the CtrlCompt class must be specialized, by restricting the domains of its *transform* function parameters. For example, in a control system requiring only a simple direct-force control component, the class ForceCtrlCompt is created by specializing the parameters inputVector, stateVector, and outputVector of the function transform — each parameter would have only one element of class Force. A user selection of a particular control mode is mapped into the selection of the appropriate ctrlCompt object.

The behavior of a servo control component may be adapted, tuned, or modified through *modalInput*, which consists of a code number and a reference to the data associated with that number, so that the CtrlCompt class can select and provide the appropriate response using the given data. An example of using the modalInput object is to update the code number to indicate a transition to the feedHold state; there is no data associated with feedHold. Then, the ctrlCompt object can adjust control parameters (internal or in its associated axisCtrl component) to stop motion expeditiously.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
AxisCtrl * axisctrl
PeriodSpec * periodSpec

Main member function:
virtual void transform(ModalInput &modalInput, MeasureVector &
inputVector, MeasureVector & stateVector, MeasureVector & outputVector)

Class-structure 3.2: Interface of the CtrlCompt class

3.3.2 Reconfiguring control laws and rules

The AxisCtrl class (Class Structure 3.3) models a basic single-input single-output servo-controller to control a single variable, X. In the case of an axis of motion, X is its position. However, the genericity of this class supports many other common single-input, single-output servo-control applications. It provides access to data used or set by tools for system identification or tuning. Although these tools are not specified in this architecture, the data interfaces allow their on-line integration. The main function of this class is calControlCmd(...). Its caller supplies, as inputs, references to the feedback variables, setpoints, and the output variable nextStateX in which to receive the result.

A reference to the member periodSpec (Class Structure 4.9) is set in CtrlCompt for use by the control algorithm in the AxisCtrl class function calControlCmd(...).

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object member:
PeriodSpec * periodSpec

Accessor functions for following members:
double feedFwdFirstDerivGain
double feedFwdSecondDerivGain
double gainSetpoint
double gainSetpointFirstDeriv
double gainSetpointSecondDeriv
double offsetSetpointX
double offsetCurrentStateX
double offsetNextState
double gainSetpointOffset
double gainCurrentStateOffset
double gainNextStateOffset
double gainFeedforwardOffset
double compensatedXFollowingError

Other member functions:
void calControlCmd(
double& currentStateX,
double& currentVelocityX,
Setpoints& setpoints,
double& outputNextState)
double calcClosedLoopGain()
void closedLoopOperation(Boolean b)
void enableFeedforwardGains(Boolean b)
void enableOffsetSetpoint(Boolean b)
void enableOffsetNextState(Boolean b)
void enableOffsetCurrentState(Boolean b)

Class-structure 3.3: Interface of class AxisCtrl.

The AxisCtrl class is abstract. It must be specialized according to a chosen control algorithm, e.g., *pid* control or *fuzzy logic* control, for which the respective subclasses, AxisCntrlPID and AxisCntrlFuzzy, have been provided as examples, where the only change is the implementation of the function calControlCmd. Further specializations of AxisCtrl could include real-time system identifiers and tuners as mentioned above.

Setting gains and offsets: The gains of the setpoint and its derivatives (gainSetpoint, gainSetpointFirstDeriv, gainSetpointSecondDeriv) may be individually set. There is pro-

vision to set offsets (offsetSetpointX, offsetCurrentStateX, offsetNextStateX) and offset gains (gainSetpointOffset, gainCurrentStateOffset, gainNextStateOffset) for setpointX, currentStateX, and nextStateX respectively. Ideally, scaling or offset of output and feedback should not be needed in a properly engineered and tuned servo-control system. However, these adjustments have been available in hardware traditionally – primarily to help in the initial startup and tuning or to compensate for changes in characteristics of the controlled system. The AxisCtrl class provides the software-analog of those hardware adjustments. It also provides functions to enable or disable offset of the setpoint (enableOffsetSetpoint), offset of the output (enableOffsetNextState), and offset of the feedback (enableOffsetCurrentState). The need for such adjustments may be minimized by utilizing the information provided online about the kinematics and dynamics of the axis. The dynamic characteristics may be determined experimentally, utilizing the functions provided with its software set to the appropriate mode of usage.

Initialization: A default initial state of the axisCtrl object is provided to suit the most common cases, i.e., applications that do not use feedforward and applications that take advantage of the facilities described above for adjusting gains and offsets in control parameters. The axisCtrl object has closed loop operation disabled, to avoid unexpected startup action — the application program determines the sequence in which closed loop operation is enabled. The default initial state may be provided by invoking the following functions in the constructor of the axisCtrl object: enableFeedforwardGains(FALSE), enableOffsetSetpoint(FALSE), enableOffsetNextState(FALSE), enableOffsetCurrentState(FALSE), closedLoopOperation(FALSE). Values of the following data members are initialized to zero: feedFwdFirstDerivGain, feedFwdSecondDerivGain, gainSetpointFirstDeriv, gainSetpointSecondDeriv, offsetSetpointX, offsetCurrentStateX, offsetNextState, gainSetpointOffset, gainCurrentStateOffset, gainNextStateOffset, gainFeedforwardOffset, compensatedXFollowingError.

Open or closed loop option: Modal settings are provided to reconfigure axisCtrl. Closed loop operation may be enabled or disabled through the function closedLoopOperation(...). For example, in the case of a machine tool spindle, preparatory to generate a helical path, closedLoopOperation(TRUE) sets spindle for closed loop operation. To prepare for ordinary machining, closedLoopOperation(FALSE) sets it for open loop operation. Feed forward control may be enabled or disabled with the function enableFeedforwardGains(...). The data member compensatedXFollowingError is informational output from *AxisCtrl*, calculated as shown in Equation 3.1.

$$
\begin{aligned}
compensatedXFollowingError \;=\; & \\
& (SetpointX \\
& + offsetSetpointX * gainSetpointOffset) \\
& - (CurrentStateX \\
& + offsetCurrentStateX \\
& * gainCurrentStateOffset)
\end{aligned}
$$

3.4 Software to facilitate axis setup

The AxisSetup class provides services preparatory to normal operation of an axis, e.g., setting the home position, travel limits (TravelLimits class), and various operational limits (OperationalLimits class) to be used by motion planning and coordinating clients of an axis. Preparatory settings are separated from software used in continuous control of axis motion (applying Rules & Constraints Set 4 Rule 1), for simplification. These facilities are not available in current practice. Client software will be able to perform certain operational setting changes automatically.

3.4.1 Setting operating limits of an axis

The OperationalLimits class (Class Structure 3.4) provides for online availability of key operating limits of an axis, to facilitate the following types of functions, as examples. A near real-time process planner or a motion control algorithm could optimize the performance-cost relationship, monitoring software could detect "out of limit" conditions, exception handlers could prevent damage from malfunctions, and maintenance and diagnostic software could detect degradation trends. The operating limits are clustered in five attributes, three of which (overshoot, underReach and followingError) provide monitoring limits, and the remaining two objects (tolerances, dynamicLimits) provide operational limits for use by axis clients. The data member "overshoot" is the amount by which the axis travels beyond the specified target position), "underReach" is the amount by which the axis travel falls short of the specified target position, and "followingError is the maximum amount by which actual position is lagging the setpoint. These three attributes are of class AxisError (Class Structure 3.5), which has three attributes, *warningLimit* that an application may use to generate a warning, *violationLimit* at which the application may interrupt processing, and, if this limit has been violated, then the *amount* by which the limit was exceeded. A violation indicates that the axis is out of control. In normal operation of a properly tuned system, the warning level would not be exceeded. The attribute *dynamicLimits* (Class Structure 3.6) provides the limits of velocity, acceleration, deceleration, and jerk allowable for a particular type of move.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
AxisError overshoot
AxisError underReach
AxisError followingError
AxisTolerances tolerances
DynamicLimits dynamicLimits

Class-structure 3.4: Interface of class OperationalLimits.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
measure warningLimit
measure violationLimit
measure amount

Class-structure 3.5: Interface of class AxisError.

Constructor-destructor functions: Omitted for brevity **Accessor functions for following object members:** measure loadLimit measure jerkLimit measure accelerationLimit measure decelerationLimit measure velocityLimit

Class-structure 3.6: Interface of class DynamicLimits.

Most current motion control systems facilitate settings for following error only, either preventing or ignoring "overshoot" and "under reach" during operation. However, a flexible machine tool may be used for different types of motion, requiring different constraints. Current controllers over-constrain motion for less demanding moves or do not meet the requirements for more demanding moves. We discuss how this architecture allows an application to reduce the compromise.

Switching operating limits specific to type of motion process

Most moves in a machining application may be characterized as shown in Table 3.1 for the purpose of constraining parameters of axis motion. There is significant performance benefit in adjusting the operating limits corresponding to each type of move. The architecture facilitates this adjustment as follows. The application developer establishes the operating limits for each type of move ahead of time. The application creates an object of the OperatingLimits class corresponding to each type of move, initializes it with the pre-established values, and makes these objects visible to the axis client. Since the client is aware of the sequence of moves, the client sets the operatingLimits object in AxisSetup to the object corresponding to the next move, before initiating the next move. Thus, motion planning and control algorithms can use operating limits specific to the type of move to optimize the performance-cost relationship.

Motion process	Path deviation	Velocity deviation	Accel limit	Jerk limit
Rapid traverse (rt)	H	+0	H	MH
$rt \rightarrowtail rm$	M	+0	MH	M
Rough machining (rm)	M	$+0, -\delta V_{rm}$	MH	M
Finish machining (fm)	ML	$+0, -\delta V_{fm}$	ML	ML
$(rt \cup rm) \rightarrowtail fm$	ML	+0	ML	ML
Finished inspection (fi)	L	+0	L	L
$(rt \cup rm \cup fm) \rightarrowtail fi$	L	+0	L	L
Jog	M	+0	L	L
Feed hold	M	+0	M	M
Emergency stop	H	+0	H	H

Symbols:
(L) Low (ML) Medium low (MH) Medium-High (H) High.
$\rightarrowtail$: Transition to
$-\delta V_{rm}$: Decrease allowed for rough machining
$-\delta V_{fm}$: Decrease allowed for finish machining

Table 3.1: Motion process specific operating limits

Traverse: A traverse is a motion process used for repositioning the axis to the next working position rapidly. Since it is a non-value adding move, it is desirable to minimize the time cost of this move. Path accuracy is not a consideration. In flexible machining systems for automotive parts, the repositioning moves are a very large proportion of total move times. The moves are relatively short, so that acceleration and deceleration times in these moves dominate. The time cost of short traverses may be reduced by applying higher acceleration and deceleration (switching to a corresponding dynamicLimits object), or by not requiring precise arrival at the destination point before initiating the next move request (switching to a corresponding tolerances object), or by tolerating "overshoot" and "underReach" and possibly allowing a larger "followingError" (switching to the corresponding objects). An operationalLimits object corresponding to traverse motion facilitates the change in settings.

Transitions to more constrained motion: When traverse is followed by machining or machining is followed by inspection, the following move is sensitive to the resulting uncertainty in the position of the end point. This problem has prevented the use of more aggressive motion parameters in less constrained moves. To facilitate the solution of this problem, a corresponding operatingLimits object may be selected for each type of transition.

Processing moves: Moves in which some process is being performed on the workpiece have to be more constrained than idle traverses. For example, in a finish machining move, no overshoot would be tolerated, precise positioning accuracy would be needed, and following error may have to be limited. During rough machining, the operating limits could be looser. In contrast, for inspection, the tolerances may be tighter and accelerations may be limited to reduce structural vibrations.

3.4.2 Travel limits

The data member travelLimits (Class Structure 3.7) specifies three travel limits in each direction of travel (forward and reverse) — the positions of the software overtravel limits, "hardwired" overtravel limits, the absolute limit ("hard stop") of the travel stroke in the reverse direction (mechRevOTravelLim), and the total travel (maxTravel). Additionally, a homePosition is defined. The value of mechRevOTravelLim is used as a reference for all other values. Violation of the "hardwired" travel limits indicates an "out of control" condition, triggering an emergency stop. The software limits may be used by some exception handler to trigger a "hard stop", i.e., use of maximum deceleration. In a properly programmed control system, "watchdog" software should prevent a programmed move that violates the "software" limits. The "hardwired" and "hard stop" travel limits may be measured experimentally. With this information, the "soft" travel limits may be determined, by calculating the needed stopping (deceleration) distance (to avoid reaching a "hardwired" limit). The deceleration available may be derived using data available from other objects in an axis model (TransTravelCapabilities, AxisDynamics, LowerKinematicModel). This example illustrates that the availability of relevant axis data online allows better monitoring and exception handling, and the modular organization eases the programming effort.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
length_measure maxTravel
length_measure mechRevOTravelLim
length_measure homePosition
length_measure softFwdOTravelLim
length_measure softRevOTravelLim
length_measure hardFwdOTravelLim
length_measure hardRevOTravelLim

Class-structure 3.7: Interface of class AxisTravelLimits.

3.4.3 Capacity and accuracy capabilities of an axis

The class TravelCapabilities serves information about the velocity, acceleration and load capacity of an axis, and its accuracy capabilities. Referring to Figure 3.4, quasiStaticLoadLimit is the maximum force at TO — typically generated when the axis is standing still and the input at TI is at the maximum value, zeroVelAccLim is the corresponding maximum acceleration at TO, provided there is no external force applied at TO, and maxVelAccLim is the maximum acceleration available at TO at the maximum velocity of the axis given by the member velocityLimit in the class DynamicLimits. These four data members are the basic capacity limits of the axis. They can be derived from the lower kinematic and dynamic models of the axis.

The basic accuracy capability of the axis is characterized in industrial practice by *repeatability* of position, measured in point-to-point moves from zero-velocity to zero-velocity, using normal operating limits of acceleration and jerk. Often, this value of repeatability alone is not sufficiently indicative of the accuracy obtainable under actual operating conditions. Therefore, we propose a set of metrics for positioning error, defined at the extreme operating points of velocity and load fluctuation. These accuracy metrics are expressed as ratios of the worst-case positioning error to the conventionally measured repeatability. The data member posErrRatioIdleStationary is the error at zero-velocity and no external load, but in the presence of internal disturbance (static or quasi-static). The data member posErrRatioIdleMoving is the error at maximum velocity and no external load, but in the presence of internal disturbance (static or quasi-static). The data member posErrRatioCutStationary is the error at zero velocity, when subjected to a unit step load at the work point. The data member posErrRatioCutMoving is the error at maximum velocity, when subjected to a unit step load at the work point. From these ratio measures, an application program can estimate accuracy at a given operating velocity and expected load change. By having this information on line as a part of the axis model, the user can use the information during operation to adapt the work processing parameters. The accuracy metrics are determined experimentally; they also serve as metrics to monitor degradation, e.g., due to wear and tear. The user can perform the experiments, at selected intervals and events, compare the results with past performance, and store the maintenance history on line.

Constructor-destructor functions: Omitted for brevity **Accessor functions for following object members:** measure quasiStaticLoadLimit measure zeroVelAccLim measure maxVelAccLim measure velocityLimit measure repeatability ratio_measure posErrRatioIdleStationary ratio_measure posErrRatioIdleMoving ratio_measure posErrRatioCutStationary ratio_measure posErrRatioCutMoving

Class-structure 3.8: Interface of class TravelCapabilities.

3.5 Axis kinematics

Kinematic information about an axis is maintained in the object axisKinematics (Class Structure B.7), in two parts – the lowerKinematicModel object and the upperKinematicModel object. The former represents kinematics internal to the physical axis, for the purpose of servo control of the axis actuator (Figure 3.4). This model also supports information needed for incorporating dynamics in the control of axis motion. The upperKinematicModel (Class Structure 3.14) represents the workspace view of axis kinematics, for the purpose of multi-axis coordination to control the workpoint trajectory, served in the form of a kinStructure object (Class Structure B.6) — useful in building the kinematic model of a multiaxis mechanism. In this manner, the information model is built to serve the axis-specific data needed to transform motion in the workspace into motion needed from a single axis, and to transform that requirement into input to its actuator. The domain abstractions for both parts of the kinematic model are based on mature mechanical engineering knowledge (Procedure 1, S5.1).

Architectural design rationale: In the most common cases, this type of information is considered "constant", and in current practice, much of this information is not accessible on line in time to support control computations – some data is supplied at the time of configuration, in the form of a "machine constants" file or equivalent. However, certain kinematic attributes may change during operation, e.g., thermal deformation of the leadscrew. It is not sufficient to ignore this variability when higher performance is required. In such cases, axisKinematics also serves a role during operation. We discuss this further in the following sections.

3.5.1 Lower kinematics model of an Axis

The model of an axis from the perspective of servo control of its actuator is labeled "lower kinematic model." It was introduced earlier (Figure 3.4) in terms of terminal TI for its actuation input (the *manipulated variable*) and terminals FBV, FBP1, FBP2 for its feedbacks, and a terminal TO for its output (the controlled variable).

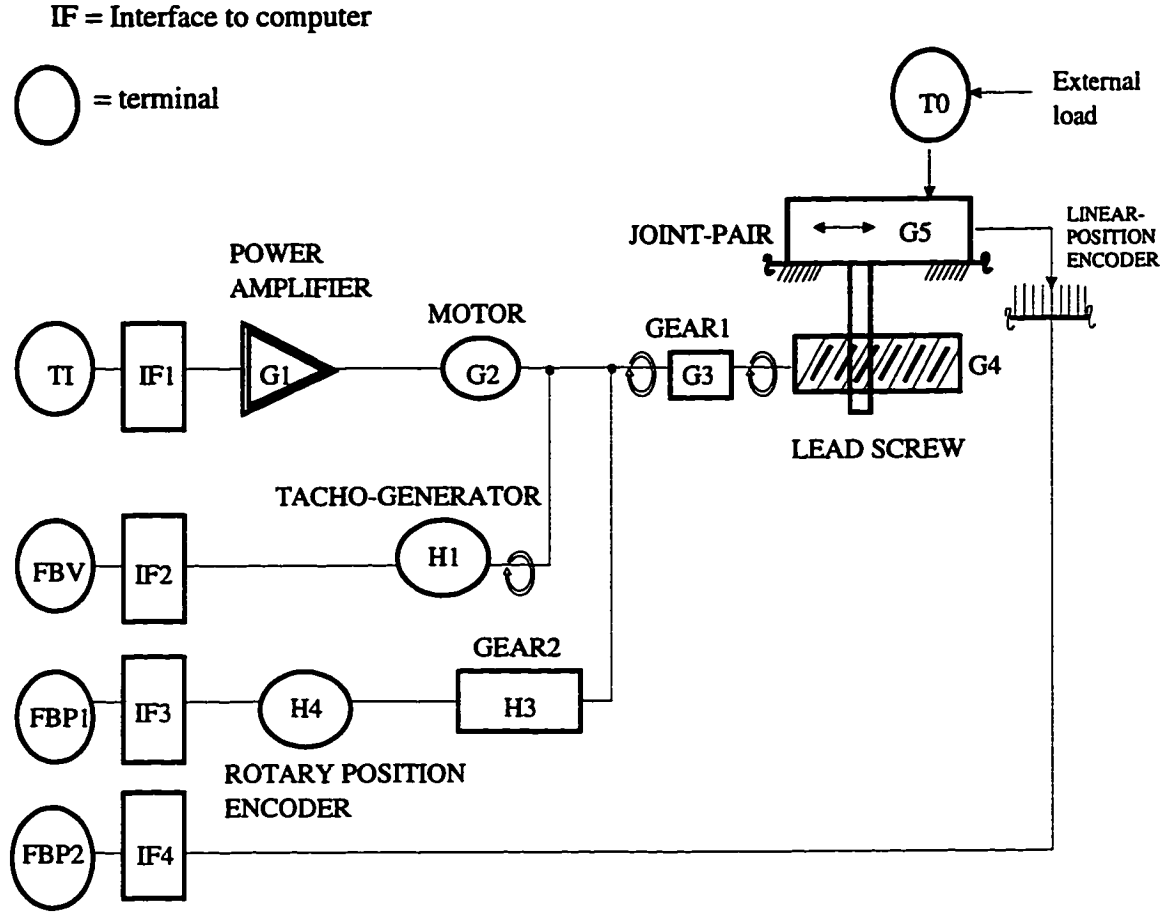

Figure 3.4: Schematic block diagram of a typical translational axis.

Referring to Class Structure 3.9, LowerKinematicModel serves the forward gain and the feedback gains for position and velocity, and the associated offsets at terminals TI, FBV, FBP1, FBP2 — the values that map to zero values of the corresponding variables at the axis output (terminal TO). This technique reduces the effort and skill required to update parameters of an axis affected by a change in some axis component, e.g., retrofit of a drive or a motion feedback sensor.

We represent the "lower kinematic model" as a directed acyclic graph (DAG) (Procedure 1, S5.4), where the components (axisKinCompts) are the nodes and their interconnections (componentConnections) are the directed arcs. AxisKinCompts and ComponentConnections are container classes, providing the services of insertion, removal, and update for each contained member (Procedure 1, S5.4). The container axisKinCompts has objects derived from class AxisCompt (Class Structure B.8), so that the property of the respective component can be obtained from each object. Each component has a single input and a single output connection. The connection of the output of one component to the input of another component is an identity transformation, i.e., the two quantities are of the same type and have the same value. The container componentConnections has objects of class ComponentConnection (Class Structure 3.10). Each object represents a directed arc in terms of the connected components, which the user identifies by user-supplied integers, fromNodeNum and toNodeNum, and a computer program identifies by references, fromNodeRef and toNodeRef. The user supplies the DAG information to create the objects

for the lower kinematic model ahead of startup time, as persistent information, identifying each component with an integer code. At startup and initialization, the function setComponentReferences() of the LowerKinematicModel class sets the component object references in each of the objects in componentConnections.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
ComponentConnections componentConnections
AxisKinCompts axisKinCompts
gain_measure openLoopFwdGain
gain_measure velFeedBackGain
gain_measure posFeedBackGain1
gain_measure posFeedBackGain2
int inputOffset
int velFeedbackOffset
int posFeedbackOffset1
int posFeedbackOffset2

Other member functions:
setComponentReferences()
void calcOpenLoopGain()
void calcVelocityFeedbackGain()
void calcPositionFeedbackGain1()
void calcPositionFeedbackGain2()
measure output inOutTransform(int input)
int input outInTransform(measure output)

Class-structure 3.9: Interface of class LowerKinematicModel.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
int fromNodeNum
int toNodeNum
StringL32 connectionType
int connectionNum
AxisKinCompt fromNodeRef
AxisKinCompt toNodeRef

Class-structure 3.10: Interface of class ComponentConnection.

3.5.2 Abstracting kinematic relations as gains

The transformation $\frac{outputChange}{inputChange}$ of a component, a subsystem of components, or the whole axis is known as its *gain*, and in the case of a static input, it is known as *static gain.* This well-known concept in control systems (Procedure 1–S5.1) provides a unifying abstraction for the variety of components described in the example above. From the viewpoint of the controller of an axis, the corresponding abstractions are its open loop forward gain and feedback gains under static conditions, defined in Equations 3.1–3.4 for the example of Figure 3.4. Their values may be supplied manually or derived from the lower kinematic model, using the respective functions.

$$openLoopForwardGain = \frac{outputChange_{TO}}{inputChange_{TI}} \tag{3.1}$$

$$velocityFeedbackGain = \frac{valueChange_{FBV}}{velocityChange_{TO}} \tag{3.2}$$

$$positionFeedbackGain1 = \frac{valueChange_{FBP1}}{displacement_{TO}} \tag{3.3}$$

$$positionFeedbackGain2 = \frac{valueChange_{FBP2}}{displacement_{TO}} \tag{3.4}$$

The forward gain and feedback gains can be different from one physical configuration of an axis to another. These relationships affect how computer software must interpret the various signals. Over the useful life of a machine tool, servo drives and feedback sensors may be replaced several times. Software integration corresponding to these retrofits must be made easy. As these changes can occur independently, their effects must be isolated (Rule 1). A software model of the interconnections of these components would allow isolation of a change in any component. Such a model would also be useful in providing on-line support for calibration, tuning and diagnosis.

Example application to a translational axis

Consider the example lower kinematic model of a translational axis, shown in Figure 3.4 and described next, beginnning from terminal TI and following the components and interconnections. Each component performs a linear transformation of its single input to its single output. Power amplifier (G1) transforms the input signal at TI into an electromotive force (emf), which is connected to the input of the motor (G2), which transforms the emf input into mechanical rotation (angular velocity). Gear1 (G3) transforms this angular velocity into another angular velocity, which is transformed by the leadscrew (G4) into a linear velocity. The output of G4 is transformed by the joint-pair (G5) – a unity transformation – into linear velocity of the workpoint (TO). The output of G2 is also connected to sensing devices that feed back velocity and position for servo-control of axis motion. A tacho-generator (H1) transforms the angular velocity of the motor output into an analog emf. The interfacing A/D converter (IF2), transforms this analog signal into a digital number output at FBV. Gear2 (H3) transforms motor output rotation (plane-angle) into another rotation (plane-angle). The output of H3 is transformed by a rotary position encoder (H4) into a sequence of electrical pulses. The interfacing counter (IF3) transforms this sequence of pulses into a digital number output at FBP1. Similarly, the output of H2 is transformed into a digital number output at FBP2.

3.5.3 Example of velocity measurement

Referring to Figure 3.4, the computer control system sees velocity measurement at terminal FBV as an integer value, but a client of velocity measurement needs the value in SI units (meters per second) as observable at terminal TO. Equation 3.2 describes the transformation relationship, velocityFeedbackGain. In an application not subject to change in this relationship, manually input value of velocity feedback gain could suffice. However, if an axis is reconfigured with progressive upgrades, the information model should make it easier to change the velocityFeedbackGain. The objects, componentConnectiosn and axisKiCompts and their constituents allow the user-integrator to describe the appropriate axis component characteristics and their interconnections. The function, calcVelocityFeedbackGain(), derives the velocity feedback gain automatically. When an axis component is changed, the user-integrator changes its characteristic values in the corresponding object, and invokes this function to update the velocity feedback gain.

3.6 Axis dynamics

There is mature engineering knowledge to abstract the dynamic properties of an axis of motion, in the case of linear systems, and these abstractions have been used in robotic control [37]. However, there is no uniformity for the purpose of automated data exchange. Even if the main properties of a particular type of axis component, e.g., D. C. servo motor, could be described in a uniform way for data exchange, there are variations in the properties of a complete axis, dependent on the types of components in it and their interconnection topology. We isolate this dependency by referring to the lower kinematic model (Class Structure 3.11), and model the dynamics-specific attributes, in a manner that allows the same interface to be applied to all axis topologies. We arrive at this uniformity in the abstractions by reflecting all quantities to the interface with the digital computer, using the equivalent circuit notion from electrical engineering (Procedure 1–S5.2). These values are interpreted through the referenced lowerKinematicModel object. The dynamics model (Class Structure 3.11) is described next.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
time_measure timeConstant
time_measure riseTime
int eqvtOvershoot
DampingCoefficient damping
int eqvtRunFriction
int eqvtStaticFriction
int eqvtDeadband
int eqvtBacklash
ratio_measure eqvtWorstStiffness
ratio_measure eqvtSteadyStiffness
LowerKinematicModel lowerKinematicModel

Other member functions:
void injectStepInput(int stepsize)
void measureEqvtStiction()
void measureBacklash()
void stepResponse(int stepsize)

Class-structure 3.11: Interface of class AxisDynamics.

Key dynamic characteristics: The attributes for modeling axis dynamics are explained as follows, referring to Figure 3.4 and Class Structure 3.11. The data member timeConstant is the time taken for the output, axis velocity, at terminal TO to reach 63.21 percent of its maximum value corresponding to a step input at terminal TI; and riseTime is the time taken for the output to change from 10 percent to 90 percent of its final value corresponding to a step input. These members reflect the inertia effects in an axis. The data member eqvtOvershoot is the equivalent of the overshoot (i.e., the maximum amount by which the output exceeds its final value, in response to a step input), as calculated at the terminal for feedback of velocity FBV. The value at FBV is an integer, related to the value at TO by Equation 3.2. The data member eqvtRunFriction is the value at TI required to sustain steady state velocity of the axis, when there is no external load. It is a measure of the steady state "resistance" to motion, including electrical resistance in the electrical components and friction in the mechanical components. Since this abstraction captures the damping properties of an axis, it suffices for normal operation. However, if mechanical

friction need be isolated for diagnostics, it can be derived from eqvtRunFriction, given the relevant values of the electrical component parameters (e.g., electrical resistance) in their corresponding software objects (e.g., motor). Similarly, eqvtStaticFriction is the minimum value at TI required to start motion of the axis — it reflects the combined "stiction" or "breakaway" friction effect resulting from all parts of an axis, typically observed when attempting to move an axis after it has been standing still. Defining backlash or lost motion in an axis as the maximum displacement at its actuator output upon reversal of its motion, before motion occurs at TO, its equivalent, as calculated at FBP1, is labeled eqvtBacklash, in the case where FBP1 is the feedback for servo-control. The data member eqvtDeadband is the change in input at TI, an integer value, upon reversal of its direction, for which no change is observed in the output at TO. It includes the effects of stiction and backlash. Defining stiffness of an axis as the change in actuator load corresponding to unit displacement at its output when motion at TO is prevented, let the load change at TI be *deltaInput*, an integer, and the corresponding displacement change at FBP1 be *deltaPosition*, also an integer; then $\frac{deltaInput}{deltaPosition}$ is the equivalent stiffness in the model. Its worst case value eqvtWorstStiffness is typically found at the reversal of actuation (the value could be as low as zero). Its steady value eqvtSteadyStiffness is found upon unidirectional increase of actuator load until stiffness is observed to be constant.

Functions to characterize axis dynamics: The function injectStepInput(stepsize) sets the value at TI to stepsize — it is equivalent to "*force* an output to a given value", as specified in IEC 1131. The user may invoke this function in a test program, along with other functions to collect position and velocity data. The function measureEqvtStiction() injects a sequence of step inputs starting from zero, in unit increments, until measurements at the feedback terminals indicate the start of motion. Then, stepsize is decremented to zero. It calculates and sets the value of eqvtStaticFriction. The function measureBacklash() invokes measureEqvtStiction() and upon stoppage of motion, invokes it again with stepsizes in the opposite direction, while measuring response at the feedback terminals. It calculates and sets the value of eqvtBacklash. If there is backlash, eqvtWorstStiffness is set to zero. The function stepResponse(stepsize) invokes injectStepInput(stepsize) and captures the response at the available feedback points, e.g., FBV, FBP1, FBP2, and calculates riseTime, timeConstant, eqvtOvershoot, and eqvtRunFriction.

Generalization achieved: Thus, we have generalized representations for key attributes of axis dynamics, e.g., friction (force or torque), stiffness (torsional or linear), inertia (mass or rotational moment), backlash (plane angle or length), overshoot (linear velocity or angular velocity). The data types of these quantities are integer or ratio_measure, thereby allowing use of the same interface for all kinds of axes that behave as linear systems.

Extensibility: The functionality of AxisDynamics may be extended by specialization. For example, the function stepResponse(...) may be specialized to calculate and set eqvtSteadyStiffness also. Other test functions, e.g., for sinusoidal response may be added.

3.7 Command setpoint inputs to an axis

The responsibility of AxisSetpoints (Class Structure 3.12) is to hold client-supplied setpoints of controlled variables, for normal repetitive operation, e.g., desiredPosition, desired-

Velocity, desiredAcceleration, desiredForce. Here, the term "client" refers to the software unit supplying the setpoints, e.g., in this architecture, an AxisGroup object that coordinates motion of one or more axes. The rationale for clustering client-supplied setpoints in one object is to facilitate communication. A client needs to be given closer visibility only to this object, rather than the whole Axis object. If a client and the Axis object were in different address spaces and had to be closely coupled (a common case), the axisSetpoints object could be mapped in shared memory.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following members:
measure desiredPosition
measure desiredVelocity
measure desiredAcceleration
measure desiredLoad

Class-structure 3.12: Interface of class AxisSetpoints.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
measure actualPosition
measure actualVelocity
measure positionFeedback1
measure positionFeedback2
measure velocityFeedback
measure interruptedPosition
AxisKinematics * axisKin (private)
AxisSetup * axisSetup (private)

Accessor functions for following members:
Boolean inPosition
Boolean enablingPrecondition
Boolean softFwdOTravel
Boolean hardFwdOTravel
Boolean softRevOTravel
Boolean hardRevOTravel
Boolean followingErrorWarn
Boolean followingErrorViolation
Boolean overShootViolation
Boolean overShootWarn
Boolean underReachViolation
Boolean underReachWarn

Other member functions:
calcActualPosition()
calcActualVelocity()

Class-structure 3.13: Interface of class AxisSensedState.

3.8 Sensed inputs and status variables

AxisSensedState (Class Structure 3.13) is responsible for updating real-time sensed data about the state of its axis and serving corresponding derived data in terms meaningful to the axis of motion. The class serves axis position (actualPosition) and axis velocity (actualVelocity) as calculated by its functions calcActualPosition() and calcActualVelocity(), respectively, at the output of the axis (Figure 3.4, TO). These calculations require values of

the respective feedback gains, positionFeedbackGain1, positionFeedbackGain2, and velocityFeedbackGain, obtained from the axisKin object. The axisSensedState object receives a reference to the axisKin object at the time of construction. It obtains the values of the various feedback gains from axisKin during the startup and initialization procedure. Thus, it is able to serve axis state data in terms most meaningful to clients external to the axis. Feedback gain information has to be maintained at only one place.

3.8.1 Other axis state information for monitoring

In addition to current position and velocity, AxisSensedState also holds interruptedPosition – the position of the axis when hold or emergency is activated. This is useful for recovery, e.g., resumption of motion after a hold from the point at which motion was interrupted, or withdrawal of the axis to a safe position after an emergency. It is also useful for analysis and diagnosis, as in the case of flight recorder type data. AxisSensedState is also the server for the state of various logical and physical switches associated with the axis described next.

Overtravel

The typical servo-controlled machine tool axis has limit switches to detect overtravel in the forward and reverse directions. Their states are maintained in the data members, hardFwdOTravel and hardRevOTravel, respectively. Actuation of either limit switch is an indication that the system is out of control, i.e., further intervention by the computer control system cannot be trusted. The required response is a hard-wired emergency stop, which also removes power from the actuators. Recovery from this emergency response is costly in time and skill required. Therefore, it is also customary to provide soft travel limits. The class AxisTravelLimits holds these limits – the values are determined manually at setup time. AxisSensedState is given a reference to the corresponding object, travelLimits, at the time of its construction. During the startup and initialization procedure, it obtains the values of the soft overtravel limits in the forward and reverse directions. In every execution cycle, it checks currentPosition against these limits. If there is a violation, it sets softFwdOTravel or softRevOTravel respectively.

Following error and overshoot

AxisSensedState monitors *following error*, the difference between currentPosition and desiredPosition, and *overshoot*, the amount by which currentPosition exceeds desiredPosition in the direction of travel. It obtains the limits from the operationalLimits object. In every execution cycle it performs the necessary checks against these limits and sets followingErrorWarn or followingErrorViolation or overShootViolation accordingly. If any of the limits mentioned above is violated, AxisSensedState sets enablingPrecondition FALSE. AxisCtrl checks this value in every execution cycle, and proceeds accordingly. One of its actions may be to send an appropriate message to its client, and in this manner, information about the malfunction is propagated upward in the client chain until it reaches the user interface.

3.8.2 Novelty

There are several novelties in the design of this class. First, all the state information about one axis is clustered in one object, including on-off hardware switches and similar logical variables, to facilitate valuable quick interaction amongst these variables. In traditional practice, the limit switch signals of an axis are not connected to its servo-control software. The discrete signals are processed in a separate discrete logic control system, which does not have close interaction with the servo-control subsystem, and therefore, these signals do not lead to quick response. Secondly, the services of the axisSensedState object may be used by other clients too, e.g., for near real-time monitoring, post-failure diagnosis, or setup time instrumentation, calibration, and tuning. These clients do not have to access the hardware interfaces directly. AxisSensedState encapsulates details of a particular physical configuration of an axis of motion, specific to its feedback devices, and the manner in which the information is transported from the feedback devices. Thirdly, where performance warrants it, a reference to the whole axisSensedState object may be shared with a client, without placing other parts of an Axis object in jeopardy or contention by such sharing. Thus, this organization of information simplifies the serving or accessing of commonly used axis state information in ways not foreseen nor explicitly engineered in the initial application. Application developers may extend the AxisSensedState class through specialization, to add other less commonly used sensors, e.g., the axis drive current.

3.9 Axis output to its actuator

AxisActState is a root class, providing an interface to the axis actuator, while encapsulating details specific to the physical configuration of the axis, including the characteristics of its actuator. AxisActState is given a reference to the object lKin at startup, so that it may obtain the axis-specific transformation data during the initialization procedure. Thus, this root class is applicable to all types of servo-controlled axes. Its main function output(...) takes, as a parameter, the next state of the controlled variable, sets the data member axisOutput to that value, computes the corresponding value of actuatorSetpoint using the function lKin− >outInTransform(...), and assigns this value to actuatorSetpoint. The transfer of the value from actuatorSetpoint to the external actuator is implementation-specific. The transferring function may be included within the responsibility of a specialization of the AxisActState class. For example, a reference to the appropriate device driver may be given to it, and the function output(...), redefined so that it also performs the transfer to the device driver by calling its appropriate function. Alternatively, the system integrator may choose an independent scheme to transfer the signal to the external actuator.

Flexibility by decoupling source and destination: By setting the data members axisOutput and actuatorSetpoint, we allow decoupling of the source from the destination, thereby providing flexibility of choices in data communication and system integration. Taking an example from the preparatory stages of a system, one may replace the connection to the external actuator with a software unit that performs simulation or diagnosis or assists in setup and tuning. To take an example from run-time configurations, consider the case where the hardware interfaces for the inputs and outputs are interconnected to the computing platform processing axis control software through a serial time-division multiplexing network. In that case direct access to the actuator hardware interface is not possible. Secondly, the moment at which a fresh value of actuatorSetpoint is available may not coincide

with the time window in which the network can receive it, necessitating decoupling in timing. Therefore, some software intermediary is needed. The AxisActState class provides the communication decoupling. The subject of external communication is further discussed in Chapter 6.

3.10 Subdomain of measures and units

Since the specification of axis setpoints should also include, in some way, the unit of measurement, we apply Procedure 1 and discover the domain of measures. Applying Step S1, it can be seen that a single axis of motion has other measurement data. Applying Step S3, within the scope of the case of Figure 3.4, the search results in the discovery that there could be more data of the same types of measurements resulting from conversions of the raw measurements. The collection of these measurements, D_1, forms an initial domain of measurements. Applying Step S4 (intuitively applied so far), it is observed that the specific measurements in D_1 are instances of several types of measurement: length, plane angle, and angular velocity. In accordance with Step S5.1, we discover the underlying engineering knowledge, namely, the knowledge of measures and units. The equivalent of Steps S5.2 and S5.3 were subsumed in the organization of this knowledge in the field of physics, and this organization has matured in the form of an international standard, the *SI* system. It is readily found that all the standardized measures are organized into a common structure, with the common features of a unit of measurement, dimensions of the measure in terms of fundamental units of measure, a label or name, and a symbol. Thus, from the initially identified domain, D_1, we identify a generalized domain, D_2, consisting of all measures standardized in the *SI* system. A second iteration of Procedure 1 reveals at Steps S1 and S5.1 that there may be other measures, not included in the *SI* system. Some of these measures may apply to other data in the example of Figure 3.4. However, we defer the effort of further search, in accordance with Step S2 of the next iteration of Procedure 1. Thus, we observe that the investigation about the data type in a specific case leads to the discovery of a domain of measures. We summarize this discussion by formulating General Need 9 in the software requirements modeling process.

General Need 9 *Specify measurement type and unit for data that represent measurements.*

Within the first iteration of modeling one data input item for one use case of a single axis of motion, a sub-domain of measures and units is discovered. From Rule 3, it is evident that this sub-domain has wider application than a single axis of motion — it is likely that this sub-domain will pervade other parts of the machine control software. Table 2.3 shows the concept of using measures and units to support higher level constructs.

3.10.1 Evolving a software model of measures and units

A search for engineering information about software standards concerning measures and units reveals that a scheme has already been organized by the International Standards Organization (ISO) in STEP [22], the *St*andard for *E*xchange of *P*roduct information. This scheme is used as a guideline to organize software to specify measures and units. However, only some parts of this scheme are in common use in the engineering of mechanical products — other aspects are not familiar to the machine control industry. Thus, Rules 2 and 8 are only partly satisfied.

Additional modeling uncertainty is expressed in terms of the following questions:

Question 3.10.1 *Does additional semantics about measures and units impose significant run-time overhead in accessing the measurement in a servo-loop?*

Question 3.10.2 *If so, can the run-time cost of the semantics of measures and units be avoided without compromising compile-time type-checking?*

Deferring search for the answers to these questions to a later iteration, only a small part of this scheme is implemented initially, with the expectation that the overall structure would be reusable and the model would be extensible later. Parameters of motion (e.g., length, velocity) are expressed as the amount or magnitude of the measured quantity (type-defined as double, instead of the whole measure object). Thus, we bypass strict compile-time type checking in the initial iteration. This incremental iterative process raises Questions 3.10.3–3.10.4.

Question 3.10.3 *Would the effort in the initial increments of model development be wasted upon later discoveries of data types?*

Question 3.10.4 *If so, could improvements in the modeling process be devised through this experience of incremental development?*

Metric	Number
Classes	49
Data members	70
Function members	381
(excl accessors, default constructors, destructor)	
Functions with more than two parameters	0
Functions implemented	365
Class graphs	3
Depth of generalization-specialization tree	3
Embedded pointers to objects	28

Table 3.2: Size statistics of measures subdomain

3.10.2 Observations in developing software for measures

The model of the measures subdomain (size statistics summarized in Table 3.2) was developed and implemented in seven stages M1–M7, as shown in Table 3.3, with the expenditure of almost 400 hours of effort, applied by five participants, spread over a 20-month duration, interspersed with several changes in the tools and the development environment. Since it was a foundation domain, typically a revision in measures was the first assignment to a new participant. Therefore, this effort also includes initial costs of learning and familiarization with the development tools and environment. Most of the participants had no direct interaction related to this subdomain.

Observations about skill requirements:

The fundamentals of the OO paradigm were easy to understand, and the mechanical aspects of using the CASE tool to model a simple class were also easy to learn — it was found that high-school level knowledge of science and Pascal programming were sufficient.

Stage	Description	Item hours	Stage hours
M1	Structural model of physical quantities: - performed by non-engineering student; - includes time to learn OO modeling.	-	80
M2	Partial alignment with ISO model: - only amount data member - units not implemented	-	60
M3	Implemented units using polymorphism: - only si_unit implemented; - overloaded operator "=" for si_measure classes.	-	61
M4	Familiarization time for a new worker. Learning time to use static members Using static instead of polymorphic unit members; copy constructors not implemented; warnings added.	4 4 22 3	33
M5	Implemented ISO derived_unit_elements, reusing library class for data structure reorganization of library files	30 12	42
M6	Transition to new workers prior worker assisting new workers; new worker familiarization; fixing modeling errors and issues adding converted units model adding gain measures	4 12 34 7 13	70
M7	Learning time for new worker Overloaded arithmetic operators; implemented copy constructors; implemented destructors	30	130
ALL	Total spread across 5 workers		476

Table 3.3: Evolution of model for measures subdomain

These observations were confirmed through separate experiments (not included in Table 3.3) with two high school students and two business school students. Participants were given an opportunity to learn the concepts through tutorials. However, typically after an average of eight hours of tutorial, they felt ready to do some useful work. Of course, their learning process continued throughout the work assignments. The high-school skill level was also sufficient to understand the basic entity model of individual measures and units defined in the ISO standard, and to construct an OO model using the CASE tool. However, there were several cases where transformation of the entity model into an OO model could not be performed with the direct application of the transformation guidelines given to the participants. The difficulty increased with more involved entity relationships. The skill level of a computer science senior was needed to resolve these difficulties. A comparison of the effort in Stage M6 with the overall effort, and interviews with the participants show that the OO paradigm and tool were very helpful to a new participant in understanding the model, repairing its weaknesses, and extending it. Although previously generated code files were also available to the participant, working with code only was more difficult and costly.

Observations about efficiency of evolution:

To address Question 3.10.3, we analyze the total effort in three components — learning, familiarization, and transition to different workers; extension; improving genericity, and rework of previously modeled elements (primarily units of measure). The successive improvements and corrections at each stage also included learning experiences. Given the skill level of a novice at the start, some form of learning iteration would have been required even if we had set the goal to arrive at the Stage M7 model directly. We make several observations relevant to Question 3.10.4. First, six transitions were made successfully, without requiring any stage to "start from scratch", i.e., there was significant reuse of prior results. Secondly, the evolution inefficiency is attributed mostly to changes in personnel and long breaks between iterations. Thirdly, the bigger cost of evolution in the measures model was its impact on the classes that used measures — this cost is discussed in the respective model evolution sections.

3.11 Subdomain of space and kinematics

Further application of Procedure 1 Steps S1 and S4 to axis software modeling reveals that the concept of a coordinate frame is fundamental to kinematics in computer controlled motion of a manufacturing machine — useful well beyond the scope of controlling a single axis. Applying Step S5, we find that there is mature domain knowledge well-documented in the form of an international standard [22]. Thus, in the process of modeling a single axis, we discover a subdomain of geometry and spatial relationships.

3.11.1 Reusing resources for modeling a point in space

Adopting the abstractions from ISO STEP and building on the model of measures and a commercial class library class for a vector, we begin modeling the subdomain of space with the model of a cartesian point (Class Structure B.4). A mismatch between the STEP specification and the commercial vector class in the indexing convention requires additional work for adaptation. The cartesian_point class provides an interface to get or set the coordinates of the point in the form of a three-element vector of pointers to three length_measure objects, accessed where the indices 1, 2, and 3 correspond to the x, y, and z coordinates, respectively. If the application is restricted to 2-dimensional space in the XY-plane, the z-coordinate is initialized to 0. Functions are also provided to get or set a single coordinate, consistent with the STEP standard. The function distance(...) returns the distance of the point specified in the argument from the point on which the function is invoked.

3.11.2 Representing frames for modeling kinematics

Applying Step S5.1, we find that machine tool design does not employ coordinate transformation mathematics explicitly to describe motion or spatial relationships, and there is no uniformity in the modeling of data for exchange of information. Applying Step S5.2, we find that the related field of robotics describes motion in terms of coordinate frame transformations, and searching further, we find that the concepts are more commonly used in software for simulation of kinematic motion, but there is no uniformity for data exchange. The model of space must include a coordinate frame, but there are several modeling alter-

natives, discussed next.

Coordinate frame modeling choices and design tradeoff: A search of international standards [33] reveals several choices of abstractions for modeling a coordinate frame. However, the alternative of the homogeneous transform matrix representation (Class Structure B.5) is chosen for the following reasons, exemplifying salient architectural design principles. First, the homogeneous transform matrix representation is more commonly understood (Rule 8). Secondly, it allows incorporation of emerging knowledge for representing errors of motion, including thermal deformation [17]. Thirdly, the generality allows application to both, translational and rotational axes. Fourthly, commercially available software for matrix mathematics could be reused. The tradeoff for these benefits is that the entries of the matrix have to be of the data type double which is not sufficiently specific and descriptive in itself – additional semantics about the entries must be specified outside the matrix. Therefore, in addition to the usual matrix operations inherited from HomogeneousTransformMatrix, we specify operations rotate, translate, and transform, including more specific and restrictive parameters data types, length_measure and plane_angle_measure.

Tradeoff in reuse through a layered architecture: Thus, we have a model of cartesian space, built up to the extent needed to support the modeling of kinematic transformations. Almost all of the code for the constructor, destructor, and accessor functions has been generated automatically. Most of the code for matrix and vector mathematics has been inherited from commercial class libraries. The result is a solid foundation framework consistent with the *STEP* standard. Such a layered architecture approach often results in a tradeoff of execution time for reusability and composability of foundation resources. Run time cost deters the adoption of such approaches in real-time systems. However, these specifications only affect interfaces for information exchange — the implementation internal to a module may use a different organization of the software. Secondly, on closer examination of an individual axis application, we find that the functions dependent on this model are required only at the time of startup, initialization, and setup, when there is no servo controlled motion. Thus, there is no penalty of time cost in the servo loop. The nominal increase in space cost is justified through more efficient setup.

3.11.3 Modeling a physical kinematic structure

The science of kinematics (Procedure 1 Step S5.1) describes kinematic relationships in terms of coordinate frames and their transformations. We model this concept in the KinStructure class (Class Structure B.6). It has two CoordinateFrame attributes (Class Structure B.5) — the baseFrame is the reference (or base) coordinate frame of the physical kinematic entity, relative to which the placementFrame is defined, and the placementFrame is the coordinate frame relative to which we may define the location and orientation of some physical object attached to it. For example, it may be used as a reference for specifying the location of TO (Figure 3.4). The data in the axis placementFrame follow the D-H convention from robotics [16], used to model a robotic joint pair. In the case of a machine tool spindle — treated as a special type of rotational axis of motion — its placementFrame is located at the intersection of the spindle axis with the gage-plane of the surface on which the tool or chuck would be mounted. Kinematic errors of motion may be included as a part of this model, consistent with the scheme devised at NIST [17].

3.11.4 Upper Kinematics model of an Axis

Continuing with the application of the layered architecture paradigm illustrated in Table 2.3, we build the external or upper kinematic model of an axis (Class Structure 3.14) to the extent needed for describing the kinematics of a multi-axis machine. The properties of an axis or joint (Figure 3.4– G5), are abstracted using the D-H convention from robotics [16] (Procedure 1, S5.1). The key parameters are a, d, alpha, and theta. The abstraction holds for translational, as well as rotational axes — the (controlled) joint variable is d in the case of a translational axis, and theta in the case of a rotational axis, with the other parameters being nominally constant. UpperKinematicModel also includes models of the joint-pair components, fixedLink and movingLink. Their data type, Link. has only one attribute, frame, of class CoordinateFrame. However, application developers may specialize Link to add other attributes, e.g., to model inertia related properties, as needed.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
KinStructure kinStructure
Link fixedLink
Link movingLink
length_measure a
length_measure d
plane_angle_measure alpha
plane_angle_measure theta

Class-structure 3.14: Interface of class UpperKinematicModel.

3.11.5 Experimental observations in modeling space and kinematics

The model of the space and kinematics subdomain was developed and implemented in four stages SK1–SK4 with the expenditure of almost 70 hours of effort, applied by five participants, spread over a 20-month duration, as shown in Table 3.5. Most of the participants had no direct interaction related to this subdomain. Stages SK1 and SK2 produced only draft specifications – no impementation – through separate effort by two workers who did not continue with the project. Stage SK3 produced usable results. Table 3.2 shows size statistics of the model.

Metric	Number
Classes	26
Data members	22
Function members	20
(excl accessors, default constructors, destructor)	
Functions with more than two parameters	4
Functions implemented	9
Class graphs	10
Maximum depth of generalization-specialization tree	10
Embedded pointers to objects	10

Table 3.4: Size statistics of space and kinematics subdomain

Observations about the development effort: Although the semantic content of the classes in space is much greater than a typical class in measures, the development effort

Stage	Description	Item hours	Stage hours
SK1	Initial model of space		
	- worker previously trained in using tools		10
SK2	Kinematics domain (new worker)		
	- worker not skilled in tools or domain		
	- study, search commercial classes for reuse	10	
	- add KinStructure class	1	
	- other classes started		15
SK3	Space model revised by 2 different workers:		
	Workers previously trained in tools		
	Study of ISO standard (one worker)	8	
	Revision to use revised measures model	8	
	Implement consistent with ISO STEP	34	
	- reuse of commercial container classes	16	41
SK4	Kinematics domain (new worker)		
	- worker trained in tools, not in domain		
	- corrections to previous model		
	- upper kinematic model for axis		1
ALL	Total spread across 5 workers		66

Table 3.5: Evolution of model for space and kinematics subdomain

is smaller, primarily because of layering (from length_measure and plane_angle classes to CoordinateFrame class to KinStructure class to UpperKinematicModel of an axis). Secondly, addressing Question 3.10.3 the change in the measures model is costly — 20 percent of the effort in Stage SK2 and 12 percent of the total development effort. Although in itself this is not a large amount of labor, we observe that in a layered architecture the cost of a change at a lower layer is magnified by the number of instances of its use in a higher layer.

Lesson learned about the modeling process: The space modeling experiment also indicates a change in the order of model development steps, addressing Question 3.10.4, formalized in Rule 9. For a given amount of resources, the implication is a delay in the first working prototype, which might deprive the developers the learning benefit of insights obtained from a working prototype, i.e., the second order benefits of applying effort in a sequence that yields a working prototype earliest.

Rule 9 *To minimize cumulative model development effort, lower layers in an architecture should be developed more thoroughly before implementing higher level layers — a "bottom-up" approach.*

3.12 OO modeling of axis software — evaluation

We experienced difficulties in using the object model, when interactions with other objects was involved. However, the associated cost and comlexity appear to be a worthwhile tradeoff for the many well-recognized advantages of organizing functions and knowledge in the object model: modularization, encapsulation, uniformity of interfaces, and compaction and extensibility of the model through generalization-specialization hierarchies and polymorphism.

3.12.1 Problems with over-decentralization and autonomy to objects

Modularization and decentralization allow reconfiguration of applications with reduced modification to some "central" program, but indiscriminate decentralization can lead to unwanted side effects and loss of integrity. Consider an object A of TransaxisSensedState class, intended to be a server of axis state to an unknown number of clients There could be several sources to update values of a data member, say velocityFeedback, in A. In addition to a device driver for a tachogenerator signal, there could be other objects in an axis application, say B, C ...that derive axis velocity from axis position, using different algorithms for different conditions. Normally, in the OO paradigm, each of B, C ...would have a reference to A. Then, B can affect the state of A without the awareness of C and vice versa. To prevent against unexpected state changes, some architectural constraint is needed, e.g., in a particular system configuration, only one object should be allowed to update the state of a particular member, say, A.velocityFeedback. The accessor function to set_velocityFeedback has to be accessible to a source, say B, of its updates. If set_velocityFeedback is publicized in the external interface of A, then any other object in the system, e.g. C, having a reference to A could also use the function A->set_velocityFeedback. This is possible in a decentralized style of OO application design, where objects are may be added to work autonomously without central control. To prevent C from updating velocityFeedback directly, OO experts recommend that A be equipped with different interfaces to for B,C .... There are n different sources of update, this scheme would require n interfaces of A. As the number of interfaces increases, the base cost of creating and maintaining the architectural model increases. Secondly, this scheme (with n external interfaces of A) introduces a coupling between the design of A's class and how it will be used, defeating the very purpose of "decentralization", and limiting the durability and stability of A's class design. Therefore, the n external interfaces should not be a part of A's class for highly reconfigurable systems with low volume of usage of each configuration. Within a single program, the flow of control, e.g., for updates, should be explicitly specified in its FSM, and the program designer should assure the correctness of updates. For interaction with other programs, access to a "set_" function should be limited to only one source, e.g., using the restrictive external interfaces, mentioned earlier. For example, if the device driver for axis overtravel limit switches is executing in a separate process, it may be "connected" to AxisSensedState through an external interface object consisting of the Boolean data members hardFwdOTravel and hardRevOTravel with only this device driver performing thr'eir "set_" operations. The utility of such external interface objects may be limited to a particular application.

3.12.2 Difficulty with intra-process object interactions

Although interaction of an object with changeable objects in the same address space is facilitated by providing it their identity (references), a change in the referent is required, whenever the identity of any of those objects changes. These changes have to be performed manually, leaving opportunities for lapses. Resulting inconsistencies become difficult to trace and debug. This problem was a significant source of unpredictable delays and costs in our project.

3.12.3 Difficulty with inter-process object interactions

When object interaction, communication, or transfer occurs in a distributed system, particularly across heterogeneous computer systems, there is significant development effort

and difficulty associated with objects that contain references to other objects, particularly when polymorphism is used. This issue is exacerbated due to lack of compatibility and integration of development tools across platforms. This problem added significant cost and delay to this project when interfacing the real-time axis motion control software with a user interface running on another computer.

3.12.4 Polymorphism affects execution efficiency and repeatability

Polymorphism across many levels of inheritance introduces corresponding levels of indirection, and requires corresponding number of dereferencing operations. For example, our project employed polymorphism in device drivers, applying a set of common abstractions to interface different types of buses, networks, and sensor and actuator IO from different sources. As a result, the average time for each access was 100 microseconds approximately. By dereferencing at the time of initialization and storing the pointers, the access time was reduced to 35 microseconds approximately. The reduction was significant to the application, considering that the total execution time of a *pid* control algorithm was less than 100 microseconds. This approach required trading off dynamic reconfigurability in exchange for execution efficiency.

3.12.5 Encapsulation adds cost of indirection

Another efficiency question about the OO paradigm was the cost of inter-object "messaging." The increase in execution time due to the indirection added when the state of an object is accessed through its accessor functions – corresponding to a single indirection – is constant in the order of a microsecond on the platform used in the testbed –insignificant relative to the total execution time of a servo-loop (in the order of 100 microseconds). In contrast, the time-cost and variation associated with OS services was much larger, e.g., inter-process communication (IPC) using "messaging" in the conventional sense. The variability and cost associated with IPC via the OS' messaging service (POSIX mq service) affected the servo loop interval beyond tolerance. Therefore, it became necessary to devise a technique to prevent the cost penalty without compromising the encapsulation provided by the messaging paradigm. The AxisTask was equipped with conceptual Port objects that encapsulate CommPort (IPC) objects, which can be specialized into POSIX mq or shared memory. An object in each communicating process is mapped into the shared memory. Within each process, accessor functions of the object are used, e.g., in a producer process, to set_ values, and in a consumer process, to get_ values. Thus, the time-cost is the same as for object interaction within the same process. The IPC-configuration required excessive effort (4 hours per instance) and training (3 days), in earlier development stages. With the support of the polymorphic CommPort class, an instance of IPC may be set up in an hour or so, after a day of training in its use.

3.12.6 Difficulty in specialization by restricting the domain of members

In a generalization-specialization hierarchy, a very common need is to specify a larger domain of a member of a superclass, deriving various specialized classes by restricting the domain of that member. In the modeling and implementation languages used in this project, there was no compact way of specifying specialization by domain-restriction. For example, consider AxisSetpoints (Class Structure 3.12) – it is a vector of measure objects. In a special-

ization for translational axes, the domains of the setpoints are restricted to length_measure, LinearVelocity, LinearAcceleration, and Force, and in a specialization for rotational axes, the domains are restricted to plane_angle_measure, AngularVelocity, AngularAcceleration, and Torque. In each subclass, at each occurrence of an input within the producer and consumer of AxisSetpoints, the domain of the variable must be manually specified, i.e., all the affected data members and function members must be explicitly redefined. A conceptually simple domain restriction (which could be compactly specified in first-order logic or the Z notation) explodes into a much larger number of specification change items, increasing the labor and the risk of error in manual entry. In the superclass and each of the subclasses, the affected data members must be specified as pointer variables, which makes it more difficult to review the code, and to transfer or copy objects. The application-building labor is also increased when such objects are members of larger objects. The pointers to these object members must be manually assigned to the corresponding pointers in the superclass, increasing the application-building labor, and making the application code less intuitive and more difficult to understand.

3.13 Evaluation of abstractions in the domain

In almost every field of software application, one of the most severe problems with quality (as reflected in satisfaction of requirements), cost (as reflected in effort) and delivery (as reflected in duration) is rooted in inadequate requirements modeling. Although formal specification languages provide the mathematical constructs to describe the requirements *unambiguously*, these languages provide little assistance in formulating the abstractions that represent the real requirements *faithfully*. The focus of this project has been to reduce this semantic gap in the specific domain of control of a machine tool axis of motion. We evaluate the resulting domain model through the question, "Are the abstractions an improvement on existing ones (published or in use)?" Interface improvement criteria are as follows: reusability across many applications (size of the domain of applicability), support for checking correctness of the interface, extensibility and adaptability of the interface, composability of different applications, effort required to use the interface correctly, and skill level and effort required to familiarize with the model. We begin with an evaluation of the axis servo control model by comparing it with various generic process control models found in literature.

3.13.1 Alternatives in process control abstractions

There are many alternatives to organize the servo-control loop software. If continuous control were the only mode of using an axis, the need for explicit user access to the function processServoLoop() might have been argued, i.e., this function could have been hidden internally. However, other modes of operation might require other services of an axis. For example, in the maintenance mode, for diagnosis or calibration, it would be helpful to have a function to test the open-loop step response of the axis, known from existing practice (Rule 3). In that case, the servo loop is not required. Therefore, processServoLoop() would not be the appropriate function. By providing a named function processServoLoop() corresponding to continuous control operation, the model allows for the possibility of adding functions to support other modes of usage which are distinct and different from the continuous control mode, and are not required at the same time (Rule 1). A second alternative is to provide a higher-level user interface in place of processServoLoop(), e.g., a selection at

some human-machine interface (HMI) to *start* or to *run* the servo-loop of an axis. However, it is well-known that the form of the HMI might vary across applications and might change in successive revisions to a particular application, even if the axis were the same in all cases. Applying Rule 1, a selection from a HMI must be mapped into a function specific to an axis, say *run()* or *start()*. That function would be equivalent to processServoLoop().

Shaw [50] treats *process control* as one of the major architectural patterns of organizing software, characterized by the types of components and the special relations amongst them. At a macro-level, [50] observes a data flow architectural pattern, with an asymmetry in the data flow (controller and the controlled system), a cyclic topology (feedback of the monitored variable from the controlled system to the controller), and continuous update of the setpoints and monitored variables (we interpret "continuous" as its sampled-data system equivalent). Our axis model (Figure 3.2) corresponds to this architectural pattern, at the interface objects axisSetpoints, axisSensedState, and axisActState, where axisSensedState includes the feedback of the monitored variables and axisActState includes the manipulated variable. The *process definition* in [50] approximates to the device description in Figure 3.2, and the *control algorithm* corresponds to the controlComponent objects including the axisCtrl objects.

The axis domain model (Figure 3.2) adds precision to the architectural pattern identified in [50] by including a timing specification, periodSpec, for discretizing the continuous control system. By being specific to the domain of axis motion control, it is a larger reusable resource. The model in [50] includes *sensors*, but does not mention the counterpart *actuators*. The Booch model [11] shows both sensors and actuators, explicitly, including the signal interfacing devices. Our model clearly excludes both from the basic axis control software, limiting its scope to the sensed variables in axisSensedState and the controlled variable in axisActState. By this exclusion, the basic axis control software is sheltered from changes in the external IO. Secondly, it makes it easier to run the external IO functions in a separate process.

In the RCS architecture [1], its level 1 corresponds to axis control in function and timing specification. The RCS world model includes many types of state information and device knowledge placed in global memory, whereas our architecture subdivides the different types of information into different objects axisSensedState, axisActstate, axisDynamics, axisKinematics ..., based on usage relationships with other objects, for better maintainability of the software. RCS specifies access paths to world models at each level broadly in terms of levels in the architecture corresponding to time granularity. The axis model refines that specification for the RCS level 1 functionality, to assist in meeting the hard real-time requirements.

3.13.2 Reconfigurability of servo control software

Given that the external interface of an axis provides the function processServoLoop(), the control algorithm could be embedded in this function (as done in our evolution stage C1). However, different control algorithms or laws would be required to meet different requirements (e.g., robustness, run-time cost, accuracy) and different applications would require different types of control (e.g., *position, velocity, acceleration, direct-force* or even some combination of these controlled variables). Therefore, software affected by such changes should be replaceable without affecting other software (Rule 1) — we facilitate this reconfigurability by isolating the software in separate objects.

Ability to alter or replace control law: Several means are provided to alter the behavior of the servo motion controller, offering different points in the design space. The tradeoff factors (Table 3.6) are as follows: software skill needed, control engineering skill needed, and functional flexibility.

For a given set of *ctrlCompt* and *axisCtrl* objects, an axis control application developer can program the function axis.processServoLoop() to select the appropriate object and set appropriate parameters, e.g., various gains, based on values of the various axis mode or state parameters, e.g., aggressive gains when hold is TRUE, very aggressive gains when emergency is TRUE, lower gains when in manual mode and jogging. Thus, a set of pre-engineered solutions are packaged to respond to the most common cases requiring alteration of servo control behavior. Modal parameters can also be extended by specialization, to extend the range of pre-engineered solutions offered. This approach provides moderately high functional flexibility, without requiring specialized skills in the field — it is a relatively simple and safe way to alter servo control behavior.

A very commonly occurring need is the adjustment of the servo control behavior to match changing characteristics of the controlled axis (plant), either due to wear or due to change in the axis components. The needed adjustments may be performed by directly adjusting the servo control parameters. Technicians rely on experience, intuition, trial and error in such adjustments. By making the axis dynamics and internal kinematics model available on line, we provide systematic means of deriving, calibrating, and applying the plant model information, reducing the need for intuition, trial and error.

Any of the pre-constructed CtrlCompt and AxisCtrl objects brought on line at startup can be programmably replaced. It does not require additional software skills, provided the design value of the worst case execution time is honored in all cases. However, correct replacement requires extra-ordinary control engineering skills, because the theoretical foundation for dynamic switching of control laws is not strong.

Additional control objects can be introduced by further specialization of the Axis class, e.g., feedforward control, and filtering of the feedback. However, it requires a higher level of skill to develop the software components, and to perform the capacity validation analysis.

Behavior altering feature	Engineering skills needed in field		Functional flexibility
	Software engrg	Ctrl engrg	
Mode or state based	None	None	Moderately high
Plant model calibration	None	Minor	Limited
Adjusting control params	None	Moderate	Limited
Replace control law	Minor	High	High
Add new components	High	High	Very high

Table 3.6: Tradeoff points in design space to alter servo control behavior

Control law replacement experiments: The feasibility and ease of replacement of the AxisCtrl object has been established in two different experiments. In one experiment, the AxisCtrl interface was used to design axis control independently on two different sites by different personnel — one at the University of Michigan, Real-Time Computing Lab (UM/RTCL), and the other at the National Institute of Standards and Technology, Manufacturing Engineering Lab, Intelligent Systems Division (NIST/MEL/ISD). An implementation of the AxisCtrlPID subclass, developed at UM/RTCL was supplied to NIST and successfully integrated in NIST's axis control prototype without any communication be-

tween the UM developer of the component and its NIST integrator. Software design issues discovered in the experiment are reported in a later section.

3.13.3 Controlled access to object members in an axis

Members that are instances of non-primitive classes are known as complex objects — their access is more involved than for variables of primitive or simple data types. Accessor functions to *set* (assign values) to the complex object members are given *read-only* references of the corresponding objects. Accessor functions to *get* (obtain values) from the complex object members return *read-only* references to the objects. The read-only access through references provides safety, reduces the information exchanged, and, in cases where the same data is forwarded many times, it also provides efficiency in time and space usage.

3.13.4 Homing and jogging functions and axis boundary

Should the boundary of an axis model be the axisSetpoints interface between higher functions in the control system and the servo control loop? Or, should it also include jogging and homing functions? The latter functions may be abstracted as transformation of motion requested between two points in in one-axis space into a series of axisSetpoints. This transformation requires the generation of a velocity profile. Thus, the jogging and homing functions are similar to multi-axis motion coordination functions constrained to one-axis space. The common abstraction and the common output interface (axisSetpoints) suggest that jogging and homing functions be grouped with multi-axis motion coordination functions (satisfies Rules 1–5). This grouping partitions the domain knowledge needed in designing the components — a specialist in servo control can focus on the servo control components sensitive to timing variations, and a specialist in motion profile or trajectory generation can focus on the jogging, homing, and multi-axis coordination functions, which are not as sensitive to timing variations. Thus, modularization along domain knowledge boundaries allows focused leveraging of specialized knowledge, potentially realizing better value in the components. In the overall aim of the architecture – *ease of reconfiguration*, Rule 2 suggests that reconfiguration will be easier if software objects correspond to physical objects and their inter-relationships. Applying this principle to an axis, we find that the design of reconfigurable machine tools and robots is typically modularized at the physical axis (or joint) level — the objective is to facilitate the assembly, testing, maintenance, and upgrade of each physical axis independently. In support, the control system should be correspondingly modular. Testing, maintenance, and setup of an axis requires the functions of jogging and homing. Therefore, these functions should be available even when multi-axis coordination is not needed. This requirement provides the rationale to separate jogging-homing functions from multi-axis coordination and including them in single axis control software. Thus, we have a design conflict. The two lines of argument, applying the same principles, seemingly suggest that a JogHome object be in two different groupings. Our architectural approach resolves this conflict through its finer granularity, separating design decisions of the definition of individual objects from their grouping into executables (task structures). The JogHome class can be supplied by the same specialist that supplies multi-axis coordination software, using the same velocity profile generating knowledge. A JogHome object is packaged with axis servo control software (Figure 3.2), as a later stage decision, satisfying the requirements of standalone axis control.

3.13.5 Expanding domain boundary with extra features – tradeoffs

We have chosen to include as many of the well-known requirements of axis motion control in machine tools as fit well in a basic organization of functions and data. For example, AxisSetpoints includes setpoints for position, velocity, acceleration, and direct-force control, and a provision is made for ControlComponent objects to manipulate the corresponding variable. However, in most common applications, only one or two of these setpoints are used. Similarly, AxisSensedState includes data members holding values sensed by two position sensors, a velocity sensor, and overtravel signals, even though not all members will be needed in most applications. Our approach burdens all applications with the space cost of carrying the extra "baggage." A component provider and application buider are also burdened with the responsibility to document the members that are supported, and to ensure that the unused members do not result in any undesirable side effect. The IEEE 1003.1b and 1003.1c standards have set a precedent for this style of specification. The approach of embedding extra features not needed in all applications has become common in electronic products, because the cost of extra materials is considered to be insiginificant in comparison to the cost of maintaining engineering changes and specializations to accommodate a variety of requirements, or in comparison to the cost of application opportunities lost. We also considered the alternative of establishing a base class with only the most essential members, and various subclasses to add various members. However, the extra members are needed in various combinations in actual industrial applications. If we include all combinations known to be useful, we are faced with a combinatorial explosion of subclasses, increasing the time required to comprehend the architecture. If we include only a few cases initially, there will be a number of followon architectural changes or specializations, adding to the cost of maintaining the architecture, and rendering early adopters obsolete. The first alternative was chosen in keeping with the cited precedents, which indicate lower expected life cycle cost.

Extensible boundary - an example: The Axis class includes "place-holder" provisions for objects that will evolve over time, e.g., AxisMaintenance, which is modeled as a container of containers whose contents will be defined at the time of specialization. When reusable patterns emerge, it is expected that features will be generalized and moved up from a specialized class to a more general class.

3.13.6 Contribution to requirements modeling process

We systematized a process to model requirements for a domain of axis motion control applications, where the initial requirements and potential future reconfigurations and extensions are only partially specified at the beginning of the software life cycle. Requirements specification is not a well-developed science even for single-shot applications. For reconfigurable, extensible families of applications, it is an area of long-term research.

Evolving from specific case to general model: Application of the developed process has demonstrated that a software model for a general domain of one-axis motion control applications could be evolved, starting from a specific case study. Although very experienced engineers who are very knowledgeable in the application domain, as well as in software engineering, intuitively apply various elements of their knowledge, the intellectual process is considered to be an art. The experiment has shown, through three cycles of evolution,

that systematic analysis of a specific common case (not the most comprehensive case), yielded a general model applicable to a family of potential applications.

Discovery of subdomains: The procedure suggests decomposition of the case under study into conceptual primitives. Identification of the conceptual primitives of the representative case study and their parameterization leads to the discovery of various subdomains, serving as a foundation for a general model of the application domain. The process of analyzing a single-axis control application leads to the discovery of several subdomains, e.g., measures and units, kinematics of motion components, dynamics of motion components, and closed-loop control of a continuous process. The discovery process and the discovered subdomains are not unique to the specific case studied. The conventional software engineering process does not lead to the systematic modeling of these subdomains. On the contrary, conventional software project management, supported with recommendations from experts [51] guards against the inclusion of requirements beyond the specific, concrete requirements of a particular application.

Mining domain knowledge: We have demonstrated that a software modeler can acquire subdomain knowledge without extensive domain expertise. For example, in the subdomains of measures and units and coordinate geometry, only "high school level" academic preparation is needed to extract the pertinent knowledge from available international standards documents. It is recognized that acquiring knowledge about a domain is one of the most difficult and challenging activities in software engineering — sometimes described as "mining." Ordinarily, the study of a previous implementation, without additional knowledge about the application domain, does not lead to the needed generalization. Although domain experts are an obvious resource, they are not readily available. In the case of software to control machinery, the knowledge used in engineering the machinery is a key resource, which our procedure utilizes for parameterization of variables found in a specific case. Utilizing published knowledge , e.g., textbooks and standards, knowledge about the subdomains is acquired with reduced dependency on domain experts.

Conclusions about the iterative process to evolve the software model: Although software engineering practice accepts that the first few prototypes of an application have to be "thrown away", we did not have to abandon concepts captured in an earlier version of the model. As different participants in the experiment performed different stages of revision of a class hierarchy, they reported that the previous stage model made it easier to learn about the application subdomain described in the class hierarchy and the modeling issues.

Renaming classes and their members is a commonly occurring activity. Changes within the class graph could be made efficiently. In a case where a class name was not referenced in any member, the routine aspect of the effort averaged two minutes per name change — the database of the CASE tool assured consistency in all parts of the model. In cases where the name was referenced in some member, and the length of the chain of references was one, average effort required in a name change was within fifteen minutes per change — additional effort was required to check consistency and correct for oversights. With a longer chain of references, the effort was greater and less predictable. *Moving members from one class to another* in the CASE model took an average within two minutes per change where a member was not referenced anywhere. *Changes in function signatures* required an average effort within two minutes per parameter, in a case where the change was a name

change or a data type augmentation. However, if the change involved the *creation of a new data type* (enriched semantics), the effort could exceed an hour.

Type of change	Relative effort
Class name	L
Member name	L
Member data type	L
Function parameter name	L
Function parameter data type	L
Function return value type	L
Creation of new data type	H
Secondary effect on embedded reference	H
Eval codes: (L) Low (ML) Medium low (MH) Medium-High (H) High.	

Table 3.7: Analysis of effort in evolutionary changes to a class

3.13.7 Contribution to requirements model

A significant result of this research is a software architecture specification for the domain of axis motion control. The architecture describes an organization of software services that typefy a library of software objects, modules, building blocks, or components, their inter-relationships, objects that inter-connect software components, and rules and constraints on their configuration, interaction and execution. Conformance to this architecture will ease integration, reconfiguration, upgrade, extension, and adaptation of applications (different types of axis motion control subsystems), where *ease* is defined as relative reduction in required skill level, effort, and duration to produce a correct application.

The Axis domain model covers all levels of granularity described in Section 2.10, as illustrated in the following examples. Functions provided in the JogHome class correspond to a *command language interface* (Figure 2.7) applicable to an axis. The AxisSetpoints class corresponds to a *data exchange interface* for data flowing into the axis, and AxisSensedState class, for data flowing out of the axis. The eventPort of an AxisTask corresponds to an *executable component interface.* Our model does not preclude other compositions of a task, e.g., multiple axes and their coordination being controlled in the same task, an example of a larger grain-size (Figure 2.8), or separating the servo IO activity into a separate task, an example of a smaller grain size (Figure 2.9). To provide this flexibility, the domain model specifies *passive application object* classes, e.g., CtrlCompt and system service selection classes, e.g., Port, CommPort which can be specialized and custom-composed to suit the needs of specific applications.

Although the architecture is domain-specific, the axis motion control domain services are specified as compositions of more general subdomains, e.g., measures, servo control, sensors, actuators, and other kinematic transformers or transducers. The genericity and soundness of the architecture results from its scientific underpinnings — it integrates knowledge from a number of disciplines, including servo control of motion, mechanical engineering, machine tool design, real-time computing systems, and computer science. The scientific foundation of its structural specifications is the object model, and of its behavioral specifications, the finite state machine model. The scientific foundation of domain abstractions is the knowledge used in the engineering of the controlled system.

Elements of the architecture have been prototyped and tested in a series of evolution-

ary steps, integrated into prototype axis control subsystems at various stages, and successively refined. Multiple axes have been concurrently operated in a laboratory machine tool testbed. Control abstractions and the architectural pattern for the control loop have been reviewed with a group of domain experts from a variety of research institutions. In an inter-changeability experiment between this project and a remote site, where two different axis control applications were being developed, our implementation of the most critical software element, the servo control law, was successfully integrated into the application at the remote site by an individual not involved in our implementation.

3.14 Recapitulation

The basic functionality required in axis control, including monitoring, setup, and configuration, is well covered in our model. The primary purpose of modularization was to provide controlled access to data produced in the control of an axis, or data used by it, including parameter settings, so that the application could be extended with less time-critical functions requiring access to that data. It is expected that the organization of these functions will continue to evolve with usage experience. Further studies will be needed to evaluate how well the initial architecture facilitates reorganization.

Excessive application level data typing?

A key premise of our software modeling approach is that data exchanged between software objects should be semantically self-sufficient (self-defining) through user-defined (domain-oriented) data types. The class descriptions make the semantics explicit, so that different interacting objects developed by different sources at different times in different places may be designed to interact with the same intent. Data exchanged by interacting objects in the same program are checked for type-match at compile-time, allowing early detection of mismatch problems. On the other hand, strict data typing increases effort in maintaining the architectural specifications and effort required in the early stage of application development. A near term study is needed to validate the efficacy of the data type restrictions specified in this architecture.

Extra members in classes an excessive burden?

The extra members in classes, i.e., members not needed in all applications, require some form of automated configuration control. The life cycle cost implications need further study. There is a similar issue in data packaging (aggregation) to suit the needs of various applications, i.e., the combinatorial explosion in aggregation combinations.

Exceptions and their handling

Additions are needed in the architecture to specify the most common exceptions and their treatment. OMG CORBA [41] provides a reference model and major implementation languages (C++, ADA) provide exception handling mechanisms. However, additional study is needed to document the commonly occurring errors or exception conditions, and new exception conditions that will arise as a result of the reconfigurability features of this architecture, and to develop a uniform style and state transitions to handle these conditions.

CHAPTER 4

Axis motion software — dynamic aspects

The static aspects of the axis model introduced in Chapter 3 correspond to a conventional object-oriented information model. Class libraries conforming to those specifications serve as a foundation of reusable resources for application builders. The conventional information model does not concern itself with execution aspects, tasking models, process scheduling, and interprocess communication. The OMG [42] architectural approach is also based on the premise that a distributed application may be viewed as a single program in which remote objects are represented by local proxies, hiding externalization-internalization issues and interprocess communication. This view, a form of orthogonal decomposition (Rule 1, Section 2.6), simplifies the design process *internal* to a particular program in a distributed application. However, it does not reduce the skilled effort required in composing multi-tasking hard real-time applications. The general needs identified in (Section 2.1) require *close* temporal coordination of a cooperating set of tasks, most of which perform computations for continuous processes that are discretized at *short* intervals. The closeness of interaction and shortness of task periods are relative to the response time of basic system services. When below some critical limits, they cause extraordinary complexity in the application design process, involving many trial and error cycles. These observations were confirmed in early iterations of the experiment (Section 2.5.1). In subsequent iterations, the static model of an axis was extended to include dynamic aspects in this chapter. Section 4.1 identifies general execution patterns for various real-time axis control features, as characterization of the dynamic aspects of the requirements space. At this level of generality, this characterization also applies to other aspects of real-time monitoring and control functions in machine tools. Section 4.2 maps these execution patterns (or the execution requirements space) into a set of constraints on the architectural design space for controlling concurrent units of execution, including constraints on interactions with other software in a control system. Section 4.3 provides a structure to the architectural design space for periodic tasks, which account for most of the work and most of the design issues in axis motion control. It develops a complement of classes for structuring periodically executable tasks, and specifying their control flow, scheduling parameters, and inter-task communications. Section 4.8 describes a group of classes for a period timing service to improve timing accuracy of a task with reduced overhead of system services (Section 4.8). To further simplify the design of axis control tasks, constraints are specified on the allocation and interaction of system resources (Section 4.10). Given the domain model and architectural specifications, we describe the process of developing an application (Section 4.11).

Section 4.12 evaluates the model. We show that the model addresses the general needs

described in Chapter 2, and that it is extensible. It also discusses controversial and unresolved issues. Section 4.13 summarizes the contributions made in the software process and the axis model, remaining issues to be resolved, and promising directions of future research in modeling axis software.

4.1 Patterns of execution

The various monitoring and control functions of an axis (Section 3.2) require execution at different times in the axis lifecycle, with different frequencies or intervals and different timing constraints. Therefroe, these functions may be teated as being decoupled in the temporal dimension. By decomposing and analyzing software functions in this manner (Rule 1, Section 2.6), we find that execution sequences fall into the following five patterns, assimilated from [50,54] and specialized to develop the architecture for machine tool control in the OO paradigm: A startup procedure, aperiodic event-driven execution, periodic execution, state-based event driven responses to abnormalities, and a shutdown procedure. This analysis approach was used in [54]; we have developed it further for machine tools.

Creation: Upon startup of the application, including restart after an emergency shutdown, each participating object is created, in an orderly sequence, by invocation of its constructor, thus allocating memory for the functions and state for each object. Objects that may be repeatedly required over the life of the application (i.e., until shutdown) are created at startup and destroyed only at shutdown. Using the knowledge that these objects may be required later, we minimize dynamic allocation and deallocation of memory to minimize risk of errors and ill side effects in a time-critical and safety-critical concurrent unit of execution, trading off some extra memory. During early software development, without using this guideline, we have repeatedly experienced two types of errors in software that is executed periodically. One type of mistake is the repeated creation of objects serving the same purpose, but not followed by it sdestruction at the end of every usage, resulting in failure due to excessive consumption of memory. A second type of mistake is the unexpected destruction of objects created in certain copy operations, resulting in failure due to pointers left invalid. The sources of these failures were difficult to diagnose. These experiences support the value of static memory allocation for repeatedly used objects.

Configuration: After the creation of all the basic objects in the startup process, the specified configuration is created. It may include locating objects that must interact, and establishing their "connections", e.g., setting up the inter-process communication (IPC). The architecture does not preclude reconfiguration at some point in the work cycle when useful external work may be safely interrupted.

Initialization: Either at the time of creation or upon configuration "constants" and other initial values associated with the object may be supplied explicitly in its constructor or through a persistent object, or through the execution of an auxiliary program, possibly with user interaction.

Setup and other manual interaction: Most of the setup activities result in setting data values, some of which are determined through a (re)calibration procedure. Setup may be performed interactively and may involve reconfiguration. It is performed at some point in

the work cycle when directly productive work may be interrupted, i.e., a more time-critical activity is not competing with the setup activity for resources. Setup activities are infrequent and manually initiated; so we treat these activities, and other manual interactions, as *aperiodic event-driven* patterns of execution.

Initial calibration: Functions associated with initial calibration, e.g., determination of the various travel limits of an axis, are typically performed interactively. Some functions, e.g., determination of the home position may include a semi-automated sequence of functions, involving motion and measurement. Initial calibration includes activities that are performed after some maintenance activity that affects the original calibration, or when the electromechanical system is first placed in operation. The results of initial calibration are treated as persistent data, because they hold valid across multiple startups, executions, and shutdowns of the control system.

Recalibration: Certain initially calibrated data may vary over time due to change in operating temperatures, wear, etc. These changes are slow and the resulting kinematic and dynamic characteristics of an axis hold valid for some interval much longer than the current execution cycle. These changes may be compensated through recalibration procedures, which may be (semi-)automated. Recalibration is performed when directly productive work may be interrupted, i.e., a more time-critical activity is not competing with it for resources. In view of the long and often non-uniform intervals between its activations, a recalibration procedure is treated as *event-driven.*

Cyclic periodic activities: The core responsibility of an axis, namely, servo control of motion, requires an execution pattern that meets the constraints on the discretization of the continuous control process. The function, axis– >processServoLoop(), and the functions on which it is dependent, must be run repeatedly at the specified time interval. The time interval between successive readings of the feedback sensor(s) must be uniform and the time interval between successive setpoint outputs to the actuator must be uniform. These requirements are formalized in Section 4.2.1. The periodic activity of reporting the axis state is treated at a lower priority than servo control of motion. There may be other periodic activities, e.g., measurement of axis load through its drive current, that are not required for primary servo control of axis motion, but ar eneeded for some monitoring activity.

Abnormal events: The axis model provides two modal states, hold and emergency, a transition into which provides appropriate responses. In these states, the requirement for quick stoppage overrides normal constraints on spatial accuracy of motion.

Shutdown: A procedure associated with shutdown is invoked only once in a "lifetime" of a concurrent unit of execution. It results in a systematic release of resources, including unlinking. The destructors of all the existing objects are invoked. The shutdown procedure also saves persistent state information.

4.2 Mapping execution patterns into tasks

Focusing on the main responsibility of an axis, servo control of motion, we examine how other application functions may be accommodated without disruption of the servo

control task. In the process of mapping application functions into executable tasks, we limit the architectural design space to choices in real-time operating system (RTOS) services conforming to IEEE standards 1003.1b and 1003.1c for portable operating system interfaces, henceforth labeled POSIX [25]. The available choices are evaluated in Section 4.2.6 for suitability in the subdomain of axis control and related tasks.

4.2.1 A representative periodic task – the servo loop

In order to arrive at a model of a servo control loop task, we consider the general case of a continuous process, discretized for execution in a periodic task. We consider the general model of timing aspects of a periodic task, p, developed by Xu and Parnas [55] as a quadruple (r_p, c_p, d_p, prd_p),
where, referring to the task p but dropping the suffix p
prd = Task period; prd_{s_i} = Start time of the ith perio;.
es = Earliest time at which execution of the task can start;
le = Latest allowable time by which execution of the task must be completed;
$r = es - prd_s$;
c is the worst case execution time of the task;
$d = le - prd_s$;
$prd_{s_1} = 0$, i.e., the start of absolute time measurement;

We extend this model for application to continuous process control by adding an explicit constraint (Equation 4.1) on the uniformity required in the task period. Then, we define the servo control loop as a specialization of continuous process control as follows. A servo control loop is a special class of continuous processes whose discretization is a task, sl, described as a recurring sequence of steps (or cycle) in Procedure 6:

S1	Acquire the setpoints.
S2	Acquire the current state from the feedback sensors.
S3	Convert the raw readings into typed data.
S4	Check for enabling preconditions.
S5	Process the control law.
S6	Convert the output to match the actuator interface.
S7	Set the output to the actuator.

Heuristics 6: Servo control cycle

Atomicity of this cycle can simplify the design and maximize its performance for a given algorithm. However, there may be other system constraints requiring that direct accesses to the IO be in another process, e.g., the IO may be batch-accessed and batch-transferred along with other IO, possibly from a remote location, using a network. Since atomicity of the whole cycle could be too restrictive, excluding many practical cases, we limit it to Steps S3–S6 requiring *run to completion* semantics, and specify explicit constraints on the timing relationships with Steps S2 and S7, through Rule 10, which includes Equations 4.1–4.3. Limits on variations of the timing relationships ($var, varAdj, dis$) are specified as ratios of the period, prd, to the respective variation. These limits are related to the accuracy performance required from the control loop, the robustness of the control algorithm, the

variability in the controlled motion and external process, and the sensitivity of the controlled system to external variations. Ratiometric parameters are chosen to allow reuse of the parameter values when the same axisCtrl object is used in different applications, in which the value of prd could vary with the value of axisDynamics.timeConstant (Class Structure 3.11).

Rule 10 *Execution of a servo loop task must assure timing uniformity within specified limits (normalized to the nominal servo loop period) on the variation in the interval at which the output is updated (Equation 4.1), the variation in the interval between successive samplings of the feedback (Equation 4.2), and the variation in the time distance between sampling the input and setting the output (Equation 4.3).*

$$\forall i : done_{sl_i} - done_{sl_1} - prd_{sl} * (i-1) < prd_{sl}/var \tag{4.1}$$

$$\forall i : sampled_{sl_i} - sampled_{sl_{i-1}} < prd_{sl}/var_{adj} \tag{4.2}$$

$$\forall i : done_{sl_i} - sampled_{sl_i} < prd_{sl}/dis \tag{4.3}$$

where
sl_i: ith iteration of the servo loop task sl,
$done_{sl_i}$: time when the output to the actuator is set,
$sampled_{sl_i}$: time when the feedback is sensed,
var: the ratio of the period, prd_{sl}, to its variation across all iterations,
var_{adj}: the ratio of the period, prd_{sl}, to its variation between successive sampling times,
dis: the ratio of the period, prd_{sl}, to the variation in the time distance between sampling the feedback and setting the next output.

Figure 4.1 illustrates the time distance variation problem if these constraints are not specified, as in conventional deadline-based scheduling approaches. It shows three successive servo-loop cycles over intervals $i-1$, i, and $i+1$, of nominal length prd. Cycle $i-1$ starts at time $ts(i-1)$, runs to completion, and ends at time $te(i-1)$. The next cycle, i, starts at $ts(i)$, but some higher priority task pre-empts it. It is resumed later and completed at time $te(i)$, still meeting the traditional deadline for cycle i. The following cycle, $(i+1)$, starts soon afterwards, at time $ts(i+1)$, runs to completion, and ends at time $te(i+1)$. The time elapsed between the output of cycle $i-1$ and the input of cycle i, $oid(i)$, is much larger than $oid(i+1)$. The variation could be as large as the difference between the period, prd, and the best-case execution time of the servo loop processing cycle. Similarly, ood, the time-distance between two successive outputs, could vary by the same amount. Thus, the traditional deadline constraints do not assure conformance to Rule 10.

Stringency of servo loop period: The constraints of Rule 10 are difficult because the servo loop period is short, relative to the execution of basic system services. These timing constraints become even more stringent when such execution patterns run in a reconfigurable, flexible, multi-tasking computing environment. To our knowledge, these constraints for servo loop tasks have not been formalized nor are they applied explicitly, in practice. Research to develop solutions satisfying such constraints has been limited [23] because of difficulties in the general case. Systematic application of these constraints in practice has been limited because the cost in relation to immediate benefit has been very high. Commercial control systems do not provide reconfigurability at the level of the servo loop task, except for tuning of certain gain parameters to match the characteristics of the

```
i = iteration number
ts = start time
te = end time
ood = distance between successive outputs
oid = distance between  input and output
```

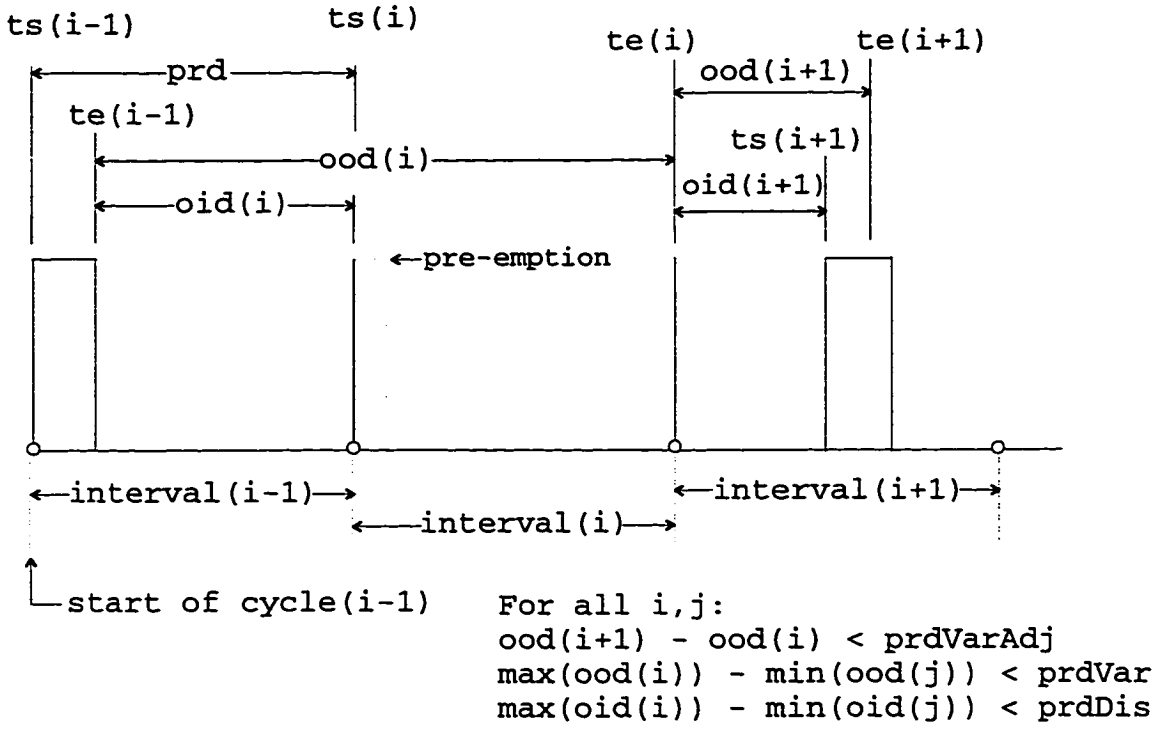

Figure 4.1: Specification of a period and time distance variation constraints

controlled system. The resulting achievable processing rate, accuracy, and rate-to-accuracy relationship are accepted as the operating limits of the system. To achieve and assure a certain level of performance (in speed and accuracy), servo loop tasks are typically run in an embedded environment with little variation in the control flow and little reconfigurability.

4.2.2 Relation of servo loop to other periodic activities

Following are some other common periodic tasks associated with monitoring and control of axis motion:

1. Acquisition of additional sensory information, e.g., axis load. The period may range from 0.05 to 1.0 of prd_{sl}. Values of var, dis, var_{adj} may be of same order as for the servo task if injected directly in the servo loop, or less stringent, if data acquisition is followed by filtering.

2. Filtering of sensed data. The period may be of same order as prd_{sl}. Values of var, dis, var_{adj} may be comparable to or less stringent than for the servo task.

3. Parameter estimation. The period may be $N * prd_{sl}$, where N is a positive integer. Values of var, dis, var_{adj} may be less stringent than for the servo task.

4. Reporting. The period may be $N*prd_{sl}$, where N is a positive integer. Values of var, dis, var_{adj} are less stringent than for the filtering task.

5. Logging. The period is $N * prd_{sl}$, where N, an integer, is larger than N for reporting. Values of var, dis, var_{adj} are much larger than for the reporting task.

Allowances of execution time and variations in the respective communication and data transfer are treated as part of the time and variation budget for each task.

4.2.3 Mapping required periods into execution periods

Multiple periodic tasks on the same computer with period values having non-integer relationships result in a long scheduling cycle and may result in unacceptable irregularities in the execution intervals from one period to the next. This problem can be alleviated with some modification to the specified periods. By specifying shorter periods than required, the application designer can choose period values prd_j for all tasks j in the application such that are integer multiples of the shortest interval prd_T timed by the basic timing service in the system (Equation 4.4). This reduces the execution timing error resulting from "rounding effects". In the case of tasks k in a dependency chain, the overall scheduling cycle is simplified if task periods are the same or if the period of a subsequent task is an integral multiple of the period of the task on which it is immediately dependent (Equations 4.5–4.6). Further simplification of the scheduling cycle is possible by applying a similar constraint on the periods of all tasks (Equations 4.7–4.8). When execution periods are different from specified task periods, parameters of the associated algorithms, e.g., gains in the control law, must be adjusted accordingly by the controls engineer. For example, consider a simple case of an axis object, x (Class Structure 3.1), with the x · ctrlCompt object, set to operate under the *position control* mode, using the x · ctrlCompt · axisctrl object, specialized from the AxisCntrlPID class, to process its control law. Then, in the simple case, for small adjustments in the value of the period, the new value may be determined by maintaining the same value of the *gain/period* ratio, R, determined from the previous values of the pertinent data members as follows:

$R = \frac{x.ctrlCompt.axisctrl.gainSetpoint}{x.periodSpec.period.amount}$

The constraints on task period relationships trade off execution time for timing accuracy, repeatability, determinism, and scheduling simplification. We formalize these constraints in Rules 11–12.

Rule 11 *Periods of tasks with dependencies should be chosen to reduce non-uniformity across successive cycles, by satisfying the conditions in Equations 4.5–4.6*

Rule 12 *Periods of other tasks should also be chosen to reduce non-uniformity across successive cycles, by satisfying the conditions in Equations 4.7–4.8.*

$$\forall j : prd_j / prd_T = N \tag{4.4}$$

$$\forall k : prd_k / prd_{k-1} = N \tag{4.5}$$

where $\forall k : prd_{k_{req}} \prec prd_{k-1_{req}}$ (dependency order)

$$\forall k : prd_k \leq prd_{k_{req}} \tag{4.6}$$

$$\forall j : prd_j / prd_{j-1} = N \tag{4.7}$$

where $\forall j : prd_{j_{req}} \geq prd_{j-1_{req}}$ (ascending order)

$$\forall j : prd_j \leq prd_{j_{req}} \tag{4.8}$$

where
N, j, k are positive integers,
prd_T is the shortest interval of the basic timing service,
prd_k is the chosen period for task k,
$prd_{k_{req}}$ is the required period for task k,
$prd_{j_{req}}$ is the required period for task j.

4.2.4 Responses requiring interruption of motion

Abnormal events requiring interruption of motion receive attention at the periodic interval axisCtrl.timestep — the response is *state-based* rather than event-driven (transitions to state hold or emergency). There are three reasons to justify not responding to the event any sooner. First, the axis may not respond significantly faster, because of the relatively large value of its axisDynamics.timeConstant. Secondly, it would require significant additional engineering effort to develop a response algorithm and its parameter values to avoid damage to the axis from shock. Thirdly, it would require significant additional engineering effort in scheduling the response correctly, i.e., without detriment to other functions in the system. Since the ratio of the incremental benefit to the incremental cost is not favorable, we justify the design simplification, and formulate Rule 13.

Rule 13 *Attend to an asynchronous event affecting change in control flow of the servo loop task at the next occurrence of the task.*

Although experts do not hold this belief in the general case of hard real-time systems [20], Xu and Parnas [55] support this position. Rule 13 does not preclude application developers from pursuing quicker responses to abnormal events affecting an axis. It only guards against increased design complexity.

4.2.5 Relation of servo loop to non-periodics

Other executions requiring responses that do not affect the servo control of motion immediately, are treated at a lower priority than the servo loop task, and they may be pre-empted. The startup and shutdown cases are not discussed, because the system is not responsible for any external action or service during those stages.

4.2.6 Architectural design space for task control mechanisms

The design space for controlling the various concurrent units of execution associated with an axis consists of a number of choices adapted from [29], as shown in Table 4.1.

Control mechanism	Disturbance in Timing
No execution control mechanism	Low
Standard operating system process	Medium-low
POSIX thread	Medium-low
Event handler	Medium-high
Interrupt service routine	High

Table 4.1: Task control mechanisms and their impact on timing disturbances

No execution control mechanism: An application with no control over the various concurrent units of execution described above is a simple feasible solution under the limited conditions where it is possible to construct a fixed sequence of executions such that there is never any contention for any resource. The application need not be a single module — it can be a collection of cooperating modules that are run at a fixed period in a fixed sequence. It may be possible to construct such a sequence if the processor utilization, τ (also known as the time-loading factor [30, pg 14]), is low. Providing excess capacity for this purpose may be an economic alternative in an application which requires little reconfigurability and which will have a small number of copies used over its life, such that the amortized cost per use may be more significant than the cost of processing and communication hardware. However, such programs become more difficult to maintain, in case many reconfigurations are needed, where tasks contend for resources and differ in activation frequencies and timing accuracies, as in the application domain targeted in this research. Therefore, we consider other options that provide control over various concurrent units of execution.

Interrupt servicing routines and event handlers: As discussed above, no application level task or function identified in the axis model directly uses an interrupt service. External IO for servo control is accessed periodically, in accordance with the requirement for discretizing a continuous process. Other external inputs, e.g., state of an overtravel limit switch, are also sampled periodically in time for use during the following execution of the servo loop task. There is no added benefit of acquiring the state-change information any sooner. Therefore, interrupt-driven execution is not needed. Acquisition of such inputs is part of a preplanned schedule, simplifying the scheduling and improving temporal determinism in the application. An interrupt from a hardware timer, or equivalent external source, is needed for the basic interval timing service. From the basic interval timer, multiples can be derived for timing larger intervals without requiring additional hardware interrupts. Interval timers trigger their client processes through POSIX (real-time) signals — this is the most frequent use of events in axis control. Interactions with the user for setting various data values, when an axis is disabled or not ready for motion, are treated as event driven. An event handler could be used for each type of interaction, e.g., corresponding to each object in the axis, but it is not necessary. The application developer could also use POSIX threads or a single process with a switch case statement (one case for each type of input event).

Processes and threads: One or more aperiodic processes or threads could be used for user interactions during configuration, initialization, and setup. Setting data values in various objects through independent processes requires the programming burden of transferring the object to the process that will do the actual work, e.g., execution of the servo loop or IO.

Threads reduce this burden by sharing the address space. Our architectural design reduces the likelihood of conflicting accesses to same addresses in several ways. First, only threads cooperating toward a common integrated goal need to share the same address space. Secondly, objects shared across threads are encapsulated, i.e., access is provided only through accessor functions. Thirdly, the design minimizes the exposure for concurrent access of an object by appropriate modularization, e.g., a thread accessing an object for setup would not be active during motion control. The application domain architecture does not force the selection between a process and a thread, deferring the choice to a specific application design. For example, most of the user interactions could be incorporated in the same process that the servo control task is running, but with state-based and event-based flow control preventing conflicts. The domain architecture focuses on design criteria for execution of the periodic tasks, several of which may be concurrent. The requirements and constraints specified in Rules 10, 11, and 12 drive the choices or selections in the management of a periodic task, including the source of its stimulus, its interactions with other tasks, the system services, and communication with external input and output. Therefore, the architectural design space must be constrained appropriately to minimize variations in the servo loop task execution timings. Since the choices in system support services are limited to services specified in the IEEE standards 1003.1b and 1003.1c, the only available scheduling policy pertinent to hard real-time systems is the priority-based preemptive scheduling, with a first-in-first-out (FIFO) policy within a particular priority level. We will refer to this as the FIFO scheduling policy, for brevity. As discussed in Section 4.2.1, this policy is not adequate for meeting the strict timing relationships required in a servo loop. Therefore, specifications must be developed for additional support to the application.

Example sources of timing variations are asynchronous events and tasks, contention for resources required by the servo loop, a high processor utilization, τ, the resulting context switches, and variations in sizes of various queues in which data or processes wait. These issues are addressed in the next section.

4.3 Structure of a periodic task for axis control

There is considerable repetitious effort involved in setting up a task even after being given a complete complement of implemented classes described in Section 3.2. The recurring patterns in a periodic computational task for continuous process control are modeled as the PeriodicTask class (Class Structure 4.1) to reduce and subdivide the effort involved in creating applications for the most common cases. A PeriodicTask class is organized as an aggregation of software objects in four groups, as follows: the resources that perform useful work in the application the fsm (Section 4.5) that specifies state-based and event-based flow of control through various functions of the resources, the ports (Section 4.4) that specify the protocol and provide the mechanism for inter-task communication, and schedParam (Section 4.6), that stores and provides parameters and data for scheduling. This decomposition allows the effort to be spread and decisions to be committed at different phases in the design process. By decoupling decisions that have to be made at different phases, we reduce the complexity of the design process. The decomposition also localizes the effects of changes in any of these decisions to separable units of program code. In this manner, we facilitate reconfiguration of control software.

For brevity of explanation, generalizations in the hierarchy above the PeriodicTask class are omitted. Functions of the PeriodicTask class, e.g., init(), execute() ... are specialized for

each application; other data members may also be added to meet the specific needs of that application.

Constructor-destructor functions:
PeriodicTask(const StringL32 "taskName")
virtual PeriodicTask()

Accessor functions for following object members:
StringL32 taskName
Resources resources
SchedParam schedParam
FSM fsm
finite state machine Ports ports

Other member functions:
virtual void init()
virtual void clear()
virtual void execute()
virtual void finish()
virtual void reset()
void setPriority(const int prio)
const StringL32 getTaskName() const return(taskName);

Private member:
int nameID // The ID assigned by the (QNX) OS name server

Class-structure 4.1: Interface of class PeriodicTask.

Objects performing useful application-level work, covered in the term, resources, were introduced in Chapter 3. For example, consider a task with the responsibility for only the control of one translational axis of motion. Its resources might be the objects axisX, positionCtrlCompt, axisCtrlPID, jogHome, axisSensedState, and axisSetpoints.

4.4 Communication between processes

Communication between processes is much more complex than communication between objects in the same name space or address space. Object and function identifiers made meaningful through a programming language compiler in one name space do not hold meaning in another name space. Similarly, the facilities of the OO paradigm to add semantic content to an object and to control state change within one program are not available across programs in execution. Setting up IPC mechanisms correctly requires experience. In early stages of our software development, we found this to be a significant problem. Participants (see Section 2.5.2, Group C) who had taken a senior-level course in operating systems required approximately three days to become familiar with OS services for shared memory and message queues, after which it took them approximately three hours to set up an IPC instance. The resulting code added clutter in the program, and made it more difficult to understand. According to their consensus, the repeated effort of setting up IPCs could be reduced with the aid of classes encapsulating the IPC code. Furthermore, we needed a technique to preserve, communicate, and reuse the semantic content of an object when communicating across address spaces, described by OMG [42] as the process of externalization-internalization. In order to reduce these repetitious elements of effort, we adopted the well-known motif, Port (Class Structure 4.2), as an abstraction for communication between processes, in parallel to the semantic constraints defined in ROOM [49].

We use the term **ports** to denote the (zero or more) **port** objects that provide the communication interfaces with other tasks. Typically, the number of ports will be specific to the needs of a particular application. A port is either designated for incoming communication (**direction**=INCOMING) or for outgoing communication (**direction**=OUTGOING). The set of messages that can be sent or received through a port, i.e., its protocol, is defined in the member **msgCodes**. All messages in the system are assigned an integer code, which is associated with a priority number and an integer code for the class of the data in the message. Interacting tasks are given at least a relevant subset of these encoding objects, thus eliminating th need to pass all the information explicitly withe each message, and thereby improving execution efficiency. Given the data class, the data object can be unmarshalled. The structure of a data class parallels the sequence of parameters of the function that will use the data, thus simplifying the transfer of the extracted variables to parameter sequences of the responding functions. Given the message code objects (Class Structure 4.4) in the container, **msgCodes**, the size of each message, and thus the largest message size, can be determined for buffer sizing.

The data member, **bufferingOption**, is an application level abstraction of the type of IPC, selected as a system design decision, and set as part of a system initializing step. When bufferingOption value *immediate* is selected, the receiver shares the sender's data structure directly — this is only possible if both are in the same address space. When the value is *shared*, the receiver shares the sender's buffer directly — possible only if both are on the same processor. The option of *direct* buffering means that the buffer is transferred directly from the sender to the receiver. The option of *buffered* transfer means that the supporting service provides the intermediate buffer — this is necessary if transferring across different nodes. In the *allocated* option of buffering, the sender allocates the buffer dynamically, and the receiver frees it after consuming the transferred data. We strive to avoid this option in a hard real-time task with a short time period, in order to minimize the variability in timing associated with dynamic allocation and release of buffers. Corresponding to the selected value of **bufferingOption**, the **bind(...)** function links the port to one of the three communication mechanisms specified in the standard IEEE 1003.1. Shared-memory is used for the *shared* buffering option and a message queue for the options *direct* and *buffered*. When the communication is only a signal without data, a real-time signal may be used. We refer to each communication mechanism as a **commPort**. Thus, depending on the selected buffering option, a corresponding type of commPort object is created and supplied as a parameter to the **bind** function. Thus we have shown that this abstraction simplifies the design process and allows reuse of the encapsulated code. The abstraction holds for tightly coupled communication between tasks within the same address space, by selecting the *immediate* option, as well as for communication across different processors, by selecting the *buffered* option.

The experiment in constructing the CommPortMQ, CommPortSM, and CommPort classes have confirmed that subtle errors could creep in under different usage conditions. The classes have required several iterations of improvements, indicated by tests under different usage conditions.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
MsgCodes msgCodes
Accessor functions for following data members:
DIRECTION direction //enum, values: INCOMING, OUTGOING
BUFF_OPT bufferingOption //enum, values: IMMEDIATE, SHARED, DIRECT, ALLOCATED, BUFFERED
CommPort * commPort

Other member functions:
bind(CommPort & commPort)
unlink(CommPort & commPort)
connect(...)
disconnect(...)
init()
transfer(...) //if direction==OUT, send; if direction==IN, receive.

Class-structure 4.2: Interface of class Port.

Example abstraction of an IPC mechanism — message queues: Class Structure 4.3 describes the interface to the IEEE 1003.1 message queue service. The user specifies the parameters in the public data members explicitly or indirectly through a program that calculates the parameter values from data provided by the user elsewhere in the system, e.g., in the definition of the ports and the message codes.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following public data members:
StringL32 connectionName
int maxMsgs //program assigns as needed.
int maxMsgsize //program assigns, calculating from msgCode
Boolean flagO_RDONLY //init TRUE
Boolean flagO_CREAT //init TRUE
Boolean NONBLOCK //flag; init FALSE
int accessPermissions //init to 0664
int exceptionCode
int msgPriority //init to 0; assign value for msgCode
int msgSize //program calculates from the msgCode

Private data members:
struct mq_attr mqAttributes
mqd_t mqID

Other member functions:
connect()
disconnect()
init()
send(...)
receive(...)

Class-structure 4.3: Interface of class CommPortMQ.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following data members:
int eventCode
ParameterListStructure *data //the structure containing the data.

Class-structure 4.4: Interface of class MsgCode.

4.5 Specifying control flow in the FSM paradigm

The task is modeled as an extended finite state machine [24], consisting of a state transition table, **stTable**, specialized to suit each application, and an application-independent engine, **fsmsupp**, to process external events and corresponding transitions using **stTable**. The **fsmsupp** object updates the current state, **currState**, of the task after procesing each event and transition.

Constructor-destructor functions:
FSM(&port)
FSM()

Accessor functions for following object members:
FSMsupport fsmsupp
STtable stTable
ProcessState currState //in top level *reset, preparing, ready, executing.*

Other member functions:
init()
reset()
getEvent(Port &port) //port->receive(...) ...
void processEvent(int eventNum)

Class-structure 4.5: Interface of class FSM.

A task has only one top-level **fsm**, constructed at the time the task is created, with a parameter, **port**, from which it will fetch events, initially. The task starts in the *reset* state, with the **fsm** accessing the given **port** at the beginning of every period. The only eligible event in the **reset** state is the one that requests the **init** function, which it initializes the task, i.e., creates objects needed in the task but not previously created in the task constructor, assigns initial values, locates (finds) interacting software units external to the task, establishes the necessary connections with them, and exchanges initial data needed. During this process, the task is in the *preparing* state. Upon successful completion of the **init** function, the task transitions to the *ready* state, when a subordinate fsm object becomes effective, with its designated set of ports. In the case of an axis to be run as an independent task, this subordinate fsm is the **axisFSM** which uses the **axisSTtable**, specialized from the **STtable** class through the specification of certain states required in all axes. The top level fsm transitions to the *executing* state when its **execute** function is envoked. It performs an execution cycle (Procedure 7), with *run to completion* semantics, as required for atomicity of the servo loop (Section 4.2.1). In the top level fsm's *executing* state, an event for the **finish** function may be called, when the subordinate **fsm** is in its appropriate state. The **finish** function is used for orderly unlinking and release of various resources (e.g., connections and internal objects) that were secured for the task. The **reset** function may be called in any state, to be used only when some error or exception is encountered for which no action is defined. It places the task in the *reset* state.

4.5.1 Reconfigurability in behavior

The **axisFSM** specifies behavior through the **axisSTtable** which is specialized for a particular application, through the addition or modification of a **transition**. Reconfiguration through the **axisFSM** allows change in behavior at a granularity of the task period, because an event is sampled only at the start of each periodic cycle. For example, some axis motion

S1 Fetch an event from its designated port;

S2 Check if a transition for that event is defined in the current state;

S3 Check if the precondition is satisfied;

S4 Perform the corresponding transition;

S5 Evaluate and update preconditions for the next cycle;

S6 Set the next state in the fsm as the currentState.

Heuristics 7: Execution cycle of a finite state machine controlling a periodic task

inhibiting event, e.g., feedHold directs the axis to stop motion expeditiously. A change in behavior at a finer granularity of time requires a change in a setting in some component or a replacement of the component — these provisions are explained next.

Switching to different behavior modes: Lozano, in his view of a closed loop model [32], characterizes automotive welding robot control as open-loop, because of its alleged inability to switch to different behavior modes, when confronted with disparate discrete events. The Axis model corrects this deficiency by providing for mode change explicitly. It provides for the input of discrete events in several ways. If sensed through hardware switches, e.g., overtravel, the signal is stored in axisSensedState. It also detects abnormal conditions (internal discrete events), deriving the information from various sensed variables and thresholds. If an axis control is running as an independent OS-managed process and some other process generates a discrete event signal, e.g., emergency stop or feed hold, the signal is introduced through the eventPort shown in Figure 4.2. To respond to such discrete events, the Axis-FSM class includes the transition which specifies the appropriate stopping action and mode change.

Closely coupled interaction across discrete-continuous control: The architecture provides two ways to introduce discrete event signals into the continuous control of axis motion. In both cases, event handling is delayed till the next occurrence of the servo loop cycle, per C4 in Heuristics 8, following from the justification given in Section 4.2.6. If the response required from the axis is a change of state, e.g., hold or emergency, to stop motion aggressively, the corresponding message is set at the eventPort of the task in which the axis control is being executed. The axisSensedState object includes data members for some common events corresponding to malfunctions within the axis, e.g., overtravel and excess following error. Other axis-related events may be added by class specialization. If further motion is to be prevented, the expression for the axisSensedState.enablingPrecondition is modified to include the additional variable.

4.5.2 Development effort for FSM class structure

The top-level FSM (size statistics summarized in Table 4.2) was developed with a total effort of 225 person-hours (Table 4.3), spread over an eleven week period, involving three

participants. Its development was initially assigned to one participant, who prototyped and tested the idea in three weeks, constructing a minimal prototype application. The participant experienced difficulty implementing the transformation of an action indentifier (used in a state transition) into a pointer to the function to be executed. For ease of reconfiguration, it was expected that the action identifier encoding-decoding would be performed through some commercial-off-the-shelf (COTS) container such as a pointer list. Due to the difficulty experienced, two other participants were assigned to work on this aspect of the problem. However, the issue could not be resolved by the conclusion of this phase of the experiment.

Metric	Number
Classes	10
Data members	7
Function members	30
(excl accessors, default constructors, destructor)	
Functions with more than two parameters	0
Functions implemented	30
Class graphs	1
Depth of generalization-specialization tree	2
Embedded pointers to objects	7

Table 4.2: Size statistics of fsm class graph

The prototyped FSM was used in several versions of a multi-axis motion control program, in which the motion control process was running periodically at an interval of 0.010 seconds, in a multitasking environment, using message-passing IPC. Execution time for the same application functionality was measured before and after applying the FSM. Although, in theory, we expected an increase in execution time due to the multiple level of indirection introduced through the FSM, we could not isolate any significant increase in execution time or its variability attributable to the FSM — probably because it was masked by the large timing-variability resulting from the IPC using message queues.

Limitation on reconfigurability of transitions: The current implementation of the FSM requires application-specific encoding-decoding of the action identifiers (packaged in the TaskActionSet class), used in the transition objects. A near-term development effort is required to improve the reconfigurability of the TaskActionSet.

Reconfigurability of a state transition table: A user interface program was developed to assist the application developer in specifying the states and transitions for a FSM. The user inputs are converted into a form that can be used in the initialization stage of a task to create the STtable object (including objects of the classes StateRecord, ProcessState, and Transition). Concept-evaluation experiments indicate that this process requires approximately two hours to build a set of ten or less state transitions. Alternatively a spreadsheet form has also been investigated to obtain user inputs of initialization data for objects. The spreadsheet form of input is also workable and practicable.

Effort to build an application-specific FSM: An experiment was conducted to evaluate the effort required (Table 4.4) for developing a small machine control application using

Description	Hours
Familiarization	17
Modeling	33
Trying pre-existing building blocks	19
Coding FSMsupport	20
Testing FSMsupport	5
Coding action code to function decoder	7
Testing action code decoder	15
Miscellaneous coding and testing	44
Upgrade of FSM	40
User interface to create stTable	24
Test usage of user interface program	2
Documentation	10
Total spread across 3 workers	225

Table 4.3: Effort to develop FSM related classes

the FSM to modularize flow of control in the program. The measured effort excludes conceptual design of the FSM. Approximately 24 person-hours of effort are required to implement a program where the number of functions is less than eighteen and the number of states is less than ten. All the functions used in the action of a transition are invocations upon pre-existing objects. We can conclude that this form of modularization of control flow is workable and practicable. However, we also conclude that the amount of effort required in building transitions should be reduced as discussed above.

4.6 Scheduling parameters

The scheduling of a hard real-time task is based on *run to completion* semantics [49]. Class Structure 4.6 describes the scheduling parameters as an aggregation of the scheduling policy, specified through the enumerated variable, schedPolicy, (default value = FIFO), the priority of the task relative to other tasks, specified through the object, priority (Class Structure 4.7), and the timing requirements and requests, specified through the object, timereq (Class Structure 4.8).

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
SCHED_POLICY schedPolicy //typedef SCHED_POLICY enum; values
FIFO, ROUND_ROBIN, USER-POLICY. Default FIFO
Priority priority
ContProcTimeReq timeReq

Class-structure 4.6: Interface of class SchedParam.

The total effort required to develop the SchedParam and Priority classes is approximately 20 hours, including integration into an application task and testing.

Priority: In a priority-based pre-emptive system, the servo loop task and secondary activities upon which it is dependent, should be assigned a sufficiently high priority level to assure allocation of resources when needed. However, events that require interruption of motion and tasks that require repetition at shorter time periods must also be executed

Phase	Description of effort element	Hours
Task-control level (5 functions):		
Design	Get states, transitions, eventCodes and actionList	2
	Draw state transition diagram	2
Implem	Create state-transition table	4
	Revise code	4
Test		4
Overall machine tool control logic level (18 functions):		
Design	Get states, transitions, eventCodes and actionList	6
	Draw state transition diagram	2
Implem	Create state-transition table	2
	Revise code	4
Test	Regular test (test this FSM alone)	4
	Integration test (test with other level FSM)	4
Process program level FSM (18 functions):		
Design	Get states, transitions, eventCodes and actionList	6
	Draw state transition diagram	2
Implem	Create state-transition table	2
	Revise code	4
Test	Regular test (test this FSM alone)	4
	Integration test (test with other level FSM)	4

Table 4.4: Effort to develop a machine tool control application using FSM for control flow.

satisfactorily. One common example is a periodic data acquisition activity with a shorter sampling interval than axisCtrl.timestep (Class Structure 3.3). If executing on the same processor, it requires a higher priority, in order to assure its timely execution [31].

Constructor-destructor functions: Omitted for brevity
Accessor functions for following data members:
int priorityOffset
OFFSET_FROM offsetFrom //typedef OFFSET_FROM enum values (MAX,MIN)

Other member functions:
int getMaxPriorAvail()
int getMinPriorAvail()

Private data members:
int maxPriority
int minPriority

Class-structure 4.7: Interface of class Priority.

Referring to Class Structure 4.7, the user should specify the priority for each O.S.-managed process, as an offset priorityOffset from the minimum value available or from the maximum value available, selecting the former case by setting offsetFrom = MIN, or se-

lecting the latter case by setting **offsetFrom** = MAX. The value of the minimum priority available, **minPriority** is obtained through the function **getMinPriorAvail()**, and the value of the maximum priority available, **maxPriority**, is obtained through the function **getMaxPriorAvail()**. The range may be the full range of the O.S.-supported priority levels for the selected scheduling policy, except that the highest limit may be restricted, in order to reserve higher priorities for critical O.S.-management processes. The priority specified by the user is checked for validity with respect to the allowable values, e.g., $1 < priority < 29$ for the QNX operating system.

4.7 Timing requirements and requests

It is difficult to assure uniformity of the servo loop period when implementing the control in a multi-tasking computer system, as illustrated from the following example. Consider the common case of current automotive machining applications,
where $0.040 < axisDynamics \cdot timeConstant < 0.10$ seconds.
When less common cases of smaller time constants are considered, and allowing for stability margins, values of the sampling interval, **periodSpec.period**, found in practice are in the range $0.002 < periodSpec.period < 0.020$ seconds.
In comparison, basic system services such as context switching and interrupt handling times are in the range of 5–10 microseconds and the scheduler consumes 25–35 microseconds per cycle, when using a commercial real-time operating system on an Intel 486-50 MHz. Considering the worse values in the ranges, **periodSpec.period** values could be as low a multiple as 200 of basic system service timings. In order to keep the period variation, dis, (Equation 4.3) insignificant (i.e., less than 1% in common design practice), the allowable variation is of the same order as basic system service execution timings. This observation indicates a need for a thorough analysis of the application requirements and a need for predictability in system services called by the application. It poses a much higher level of design complexity requiring prototyping and testing for validation.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following data members:
PeriodSpec prdSpec
TICK execTimeWC //execution time - worst case.
TICK execTimeAvg //execution time - average.
TICK offset //offset of the start of execution.

Other member functions:
Boolean countDown() // increments and tests (prd==+currentTicks)
int arm()
inline int disarm()
void resetCurrentTicks() //sets currentTicks to zero.

Private member functions:
pid_t proxy
TICK currentTicks
Boolean armed

Class-structure 4.8: Interface of class ContProcTimeReq.

Specifying time periods: One of the sources of variation controllable in application design is the mapping of many task time periods – *real* numbers – into *ordinals* for scheduling. A second source of error is the *rounding* effect when the specified periods are not integral multiples of the timing resolution. In common applications where the task time period is much larger than the period timing resolution, these errors are assumed to be insignificant, and the tasks and their designers are unaware of them. However, in axis motion control applications, these errors result in a mismatch between the actual time period at which a task is executing and the nominal time period (servo update interval) for which its control algorithm is designed. The errors from these sources can be reduced by making the mapping of *real* numbers to *integers* explicit when the timing requirements are specified, as shown in Class Structures 4.8–4.10. Initially, the system designer makes a global decision about the resolution, *sysRes*, of the timing signal from the hardware timer that will be used for the set of tasks, *taskSet*, that have a close temporal inter-relationship. The task can obtain the value of *sysRes* through the function tickCounter->getSysResolution() or by directly using a system service. The least count or resolution, prdRes, of the software timing service TickCounter is set to be an integral multiple, n, of *sysRes*, such that it is smaller than the smallest variation limit amongst the constraints of all the cooperating tasks, specified as follows:

$$\forall k : prdDis_k = m * n \mid m \geq 2; taskSet = \{Task_k\} \tag{4.9}$$

where m, n, k are positive integers.
The value of periodSpec.period is adjusted to the closest integral multiple of n, typically resulting in a period, prd, smaller than the original specified value. Then, the user adjusts the control law parameters, e.g., gains, to match this period. Thus, this technique eliminates the two systemic sources of timing error mentioned above. The time period and its accuracy requirements are specified through the class PeriodSpec. Execution time and offset are specified in the class ContProcTimeReq, as integral multiples of *prdRes*. The offset parameter is provided for a potential application-level pre-scheduler, to compute and store the task release time offsets that meet the task sequencing requirements with the shortest overall repeatable scheduling cycle. In the future, other conditions of sequencing may be added. Values of execTimeWC, execTimeAvg are obtained from the task, typically determined through offline tests prior to start of operations. The timing information may be used by an application-level pre-scheduler or by some simulation tool that tests whether the requirements can be met. The function countDown() is used by the TickCounter to test if the interval count has reached the specified period. If so, it signals the client task. The function, arm() is used to start the interval timing and signaling cycle for a particular client task, and the function, disarm(), to stop it. The request is initiated by the client task and executed in the TickCounter class.

Constructor-destructor functions:
PeriodSpec()
PeriodSpec (const time_measure period, const time_measure periodResolution, const int var, const int varAdj, const int dis)
PeriodSpec(const TICK prd, const TICK prdVar, const TICK prdVarAdj, const TICK prdDis)
PeriodSpec()

Accessor functions for following data members:
time_measure period
int var //ratio of period to allowed variation
int varAdj //ratio of period to allowed variation in adjacent periods
int dis //ratio of period to allowed variation in distance between input and output
TICK prd //typedef TICK int where its unit = *prdRes*. Conversion of period into prdRes units.
TICK prdVar //*prd* · *amount*/*varRatio* rounded down to nearest prdRes unit.
TICK prdVarAdj //*prd* · *amount*/*varAdjRatio* rounded down to nearest prdRes unit.
TICK prdDis //*prd* · *amount*/*disRatio* rounded down to nearest prdRes unit.

Class-structure 4.9: Interface of class PeriodSpec.

4.8 Period timing service for a group of tasks

Many hard real-time tasks such as a servo-motion control loop require close temporal relationship with other tasks such as servo-sensor IO accesses, and their interaction patterns fall into repeating sequences. It is very difficult to provide the required temporal relationship when the tasks are released to run periodically under individual timers, as in traditional RTOS management, as explained next.

Weakness in traditional scheduling of periodic tasks: In traditional real-time systems, every periodic task acquires its own software interval timer from the operating system services. The timer has the same priority as the task. Over the course of time, as the OS interleaves tasks of higher priority, including its own services, the signal of interval-expiration is handled at a non-uniform interval. In one technique used to circumvent the OS-induced variations, each task sets up its own timer, triggered by an interrupt from the hardware timer. However, this technique increases the number of interrupts in the system in proportion to the number of periodic tasks. Interrupts increase the system overhead. Moreover, since this overhead occurs whenever the interrupt occurs, it consumes its time slice irregularly, introducing more variability in the task timings.

Design approach to coordinate period timing service: By taking advantage of the predictable arrival pattern of these closely related tasks [55], denoted as a *taskSet* here, we can control their release more accurately and efficiently. As an early stage step in that direction, we have developed a coordinated period timing service, denoted as the TickCounter class (Class Structure 4.10) to serve the *taskSet*. It supplies the tasks in the set with release time signals at the intervals and offsets specified in their respective timeReq objects. Upon receipt of the signal, each task in the group starts its execution cycle (execute function). The TickCounter uses only one hardware timer interrupt for a taskSet, reducing its overhead without loss of accuracy. By setting the TickCounter process at a sufficiently high priority, we also reduce variability in the time it triggers a signal. Thirdly, as mentioned earlier, by explicit specification of timing in integer multiples of sysRes, rounding-off error is eliminated.

Setting up the timing service: The system startup procedure makes the TickCounter object operational in a special independent process before any of its client tasks, supplying it the values of the maximum number of clients for which it must be setup and the number of sysRes units in a prdRes unit. The TickCounter provides an incoming message queue (mq) communication interface to be used by all clients for initial registration, and an outgoing interface to signal the release time to each client.

Client registration: When the function registerClients(...) is invoked, the TickCounter prepares and waits to receive registrations from a number of tasks specified in clients, for a time limit specified in sysResCnt. If it reaches the time limit before all the clients are registered, it throws an exception. Each client registers by sending a copy of its timeReq object, which includes its timing requirements. When all the clients are registered (clientsRegistered==clients), the TickCounter is ready.

Starting and stopping the timing cycle: When the function, startCountCycle(), is invoked, TickCounter *arms* all the timeReq objects, i.e., enables their signaling, and starts its counting cycle for the group of tasks. The function stopCountCycle() stops the counting and signaling cycle. This facility allows the starting and stopping of tasks in a taskSet in the proper sequence.

Constructor-destructor functions:
TickCounter(int maxClients, int prdRes2sysRes)
TickCounter()

Accessor functions for following object members:
CommPortMQ mq4Clients

Accessor functions for following data members:
int prdRes2sysRes //number of sysRes units in one prdRes
int maxClients //maximum number of clients registers.

Other member functions:
registerClients(int clients, int sysResCnt)
connectSysResCounter() //attach interrupt handler
disconnectSysResCounter()
startCountCycle() //countDown for all clients
stopCountCycle()
getSysResolution()

Other private members:
sysResCounter() //interrupt handler: counts sysRes interrupts.
setPrdResSignal() //obtains ID of signal from sysResCounter
ContProcTimeReq timeReq[maxClients]
int clientsRegistered //init 0; increment as each client

Data members in space shared with interrupt handler:
int sysResCnt //current count of sysRes units; init 0
int prdRes2sysRes //number of sysRes units in one prdRes
pid_t prdResSignal //ID of signal from sysResCounter.

Class-structure 4.10: Interface of class TickCounter.

Connecting to the hardware timer: When the function connectSysResCounter() is invoked, the hardware timer in the computer is set up to provide an interrupt at every sysRes

interval, which is received and counted by the interrupt handler named sysResCounter() The function disconnectSysResCounter() stops the hardware timing service and releases its connection with the sysResCounter(). The function setPrdResSignal() acquires from the OS the identifier of the signal to be sent to it by sysResCounter().

4.9 Task structure impact on reconfiguration effort

The organization of software to structure a task results in a decomposition of application-reconfiguration effort elements. This decomposition and associated relative costs are summarized in Table 4.5. Only changes within one program are considered. There are four types of configuration changes in the table, sectioned in order of increasing difficulty. Reconfiguration effort elements are subdivided in five groups, corresponding to the whole program and its four parts, introduced in Figure 4.2: resources, fsm, schedParam, and ports. Each effort element is rated for its contribution to the cost of reconfiguration relative to other effort elements. The rating is based upon a combination of traditional effort metrics such as lines of code, number of object interfaces involved, observations during the development experience, and interviews with the developer participants. Next we describe these task elements and explain the cost rating assigned to each element.

Reconfiguring control flow: A change in the flow of control is most commonly expected during the early stages of a new physical configuration of the controlled system, as initial usage provides learning experience. Such changes may be viewed as "rewiring the logic", while all external inputs and outputs remain unchanged. The response to a particular event may be a different sequence of functions, which is defined in the action object, or it may be a different enabling condition, which is defined in a condition object. Their object identifiers, coded as integers, have to be added to the corresponding decoding tables. The response to an event may also be a transition to a different pre-defined state. The changes are specified in a transition object. These additions and changes require recompilation of the affected decoding tables and the stTable object (affected state transition table). Since the change may alter the task execution time, the processor utilization, τ, must be rechecked. The re-engineering effort depends upon the magnitude of τ and system overhead, Toh_t, relative to critical limits.

Replacing resource object: In response to general needs identified in Section 2.1.2, suppose that the control strategy has to be changed and several new CtrlCompt objects and AxisCtrl objects have to be added in the application program, given that these components exist in the class library. Then, the code in this application program must be modified to include these classes and to instantiate these objects — relatively easy activities. Then, the new object must be initialized to the proper values — an activity that may require some study. Availability of persistent objects would allow this effort to be applied ahead of time, reducing the intellectual work-load at integration time. The object-function encoding-decoding tables have to be updated, requiring a number of well-defined activity steps.

Adding external service: If the function of some object existing in the program, or some composition of such functions, has to be made visible as an external service of the program, then a corresponding msgCode object is added to the protocol of the port through which the service is offered. If the service requires a new data structure, then it must be derived

from the ParameterListStructure class, instantiated, and initialized, requiring a number of well-defined activity steps. Secondly, the fsm · stTable is extended with the needed event, condition, action, and transition objects. These steps involve higher intellectual activity than the extension of a port protocol.

Adding IPC: If an additional inter-process communications port is required, e.g., to connect to a new program, then the Port class is specialized and instantiated, and the object is initialized. This includes the creation and initialization of a CommPort object and the msgCodes that define the protocol of that port. The intellectual effort is greater than for extension of the fsm described above. The addition of an IPC could increase communication workload, increasing the design difficulty, depending upon the type of IPC used.

4.10 Assigning a task to processing resources

Task assignment is treated as part of an implementation stage in a traditional application development cycle. However, a new task assignment involves significant development cost, and implies significant recurring implementation and maintenance costs. Much of the development cost in a new task assignment occurs after a prototype is constructed. One of the main cost factors is the failure to meet timing requirements, forcing late-stage changes in the design and "implementation." The high cost and associated uncertainties discourage changes in the task assignment. In a hard real-time subsystem, it is fixed for a narrow family of applications in a particular organization, limiting reconfigurability.

4.10.1 Weakness in traditional software development

The classic software engineering approach favors *hardware independence* and *software flexibility*, and defers "optimization" for performance to a later stage. When poor performance results in failure to meet the timing and ordering required in a hard real-time subsystem, the design is also a *failure in correctness.* We need constraints on the "flexibility" in the system design space to avoid late-stage failures in timing correctness. The constraints must not increase hardware dependencies, i.e., not require services beyond those specified in IEEE 1003, and must also not require code optimization. It should also be possible to apply some of these constraints in early stages of system design. Experienced system architects constrain their design space intuitively. Some of these constraints are made explicit next.

4.10.2 Architectural constraints on axis control environments

Heuristic rules and constraints on the execution environment of an axis control related taskSet reduce problems in the later stages of a software development cycle. These constraints reduce timing uncertainty by excluding from the hard real-time subsystem those activities that introduce unpredictable timing variations. We justify this approach from Section 4.2.1, deriving Rule 14 from it (also used as an architectural design premise in ROOM [49]).

Rule 14 *Support run to completion semantics.*

However, the IEEE 1003 standard does not specify *run to completion semantics.* Therefore, the priority pre-emptive scheduling policy of the OS must be supplemented with constraints in the appication design, i.e., the OS should not be required to pre-empt a hard real-time

task. Since one reason for pre-emption is non-availability of some needed resource, Rule 15 is formulated to eliminate this source of pre-emption.

Rule 15 *Secure all necessary resources before releasing a task for execution.*

Rule 15 implies C1–C4 in Rules & Constraints Set 8 on the design and assignment of a taskSet. It also implies need to minimize dependence on OS services that introduce significant timing variations, as expressed in Rule 16.

Rule 16 *When the processor utilization, τ, is high, limit dependencies on OS services, such that the task management and synchronization overhead (e.g., scheduling, context switching, synchronization, mutual exclusion), Toh_t, affecting a closely coupled task set is below a significant (critical) proportion, R_{cr}, of the total execution time, c_t, of all tasks in this set, i.e., $Toh_t/c_t < R_{cr}$.*

For an application that falls within the constraints described in Section 2.10, a static task execution and communication pattern emerges during normal across. operationhen, $Toh_t/c_t < 0.01$, is insignificant. When Toh_t/c_t is insignificant, the design decision to use the OS services may be made within the design of the taskSet subsystem. If not, an application-wide analysis is needed. Increased design effort and skill are required, as the value of Toh_t/c_t increases, because Toh_t will also increase as other tasks are added, even though these tasks may be of a lower priority. It extends the dependency chain and introduces cycles in the design process, thus increasing the complexity, skill requirements, cost, and duration of the design process.

Heuristics 8 are specified to limit the complexity and cost of the design process. Rule 16 implies C5–C6 in Rules & Constraints Set 8. In traditional design of computer implementations of axis control, these constraints are honored by dedicating a set of processing resources to axis control only — interactions with other subsystems is very limited and use of OS services is minimal. Our architectural constraints allow static reconfiguration. However, the processor utilization, τ, must be within safe design limits, as discussed next.

C1	The taskSet should be serializable.
C2	Execution in the taskSet should not be dependent upon retrieval of data from mass storage devices.
C3	No paging should be required (needed pages should be locked in memory).
C4	Asynchronous requests to a task, e.g., keyboard inputs, should not be serviced until the task is activated at its normal period.
C5	The task configuration and assignment should be *static*. There should be no dynamic creation and destruction of tasks within the time critical task set.
C6	Other types of dynamic allocation and release of memory should be minimized, e.g., all objects required in a cycle of the task set should be created in the initialization stage, and destroyed only in the finishing stage. The design should strive toward a static object map in the execution stage.

Heuristics 8: Constraints on an axis control task set and its environment

4.10.3 Constraint on processor utilization

The total workload of all tasks and OS activities on a processor should be well within the capacity of the processor. If all tasks are periodic, their schedulability to meet their

period deadlines can be estimated by using rate monotonic analysis [31]. If some task is not periodic, its shortest (worst-case) inter-arrival time is used as its period for the purpose of this analysis. At the stage of architectural design, neither the application execution timings nor the platform characteristics are known well enough to perform a definitive analysis. However, by making pessimistic assumptions about load and capacity, feasibility questions can be addressed. A preliminary rate monotonic analysis is useful, even though POSIX does not specify a rate monotonic algorithm and it is not available on commercial real-time operating systems for the Intel X86-family processers. Based on this anaylsis, a feasibility criterion for the limiting processor utilization, τ_{cr}, is chosen (Rule 17).

Rule 17 *Limit the processor utilization:* $\tau < \tau_{cr}$
where $\tau_{cr} = 0.69$ *under a rate monotonic scheduling algorithm, with all tasks may be treated as periodic, and no other resource constraint exists. If worst-case execution times cannot be determined accurately at the time of conceptual design, then apply a reserve factor in proportion to the uncertainty.*

4.11 Procedure to develop an application

Given the domain model for software to control an axis of motion, Procedure 9 is applied to design a specific axis control application. This procedure is derived from the overall domain engineering based process shown in Figure 2.5. We will use the example introduced in Section 3.1 and Figure 3.4 to define the requirements. In this example, the domain architecture will be specialized (Procedure 9), primarily by specializing the constituents.

1. Define specific requirements of the application.
2. Specialize domain architecture to derive architecture specific to the application.
3. Identify components reusable from the class library.
4. Adapt library components, as needed.
5. Create the needed instances of class library components.
6. Initialize or configure the created components.
7. Design and construct additional application-specific components as needed.
8. Evaluate application-specific components for broader use across the domain. If promising, propose their evolution into a class library asset.
9. Test new components individually.
10. Integrate components into subsystems (tasks and task sets) and test.
11. Integrate system and test.

Heuristics 9: Procedure to develop an axis control application.

4.11.1 Defining requirements specific to an application

The axis control application is analyzed in terms of the controlled plant, the servo control scheme, and interactions with other software units that use its services (*clients*) or provide services to it (*servers*).

Controlled plant: An analysis of this example shows that the controlled axis (in general, known as the *plant*) is a translational axis driven by input to a pulse-width modulated power

amplifier, equipped with multiple feedback sensors (a linear incremental position encoder, a rotary incremental position encoder, and a tacho-generator). There are no overtravel limit switches. The time constant of the axis is in the range of 0.040–0.050 seconds.

Servo control scheme: The axis employs the *position control* mode using only a position reference setpoint, along with a *pid control* algorithm. The motion (acceleration) objective is *minimum jerk*. There is no feedforward control. A servo loop interval of 0.010 second is specified. Only one position feedback is used for servo control. There is no filtering, correction, compensation, or other preprocessing of the feedback signals. This represents a common case in industrial applications.

Computation to communication time ratio: In this baseline case, the pure computational time for the *pid control* is of the same order of magnitude as the time to transfer data, e.g., to access external IO associated with its computation.

Requirements of client software units: It is acceptable to respond to mode change requests, e.g., emergency stop, at the beginning of the next servo loop cycle, but further delay is not acceptable. Reporting of position and velocity is required for display at intervals of approximately 0.1 seconds in which a variation of 0.050 seconds is quite acceptable and occasional larger variations are also acceptable.

Future requirements of clients: Reporting of position history (sequence) will be required for analysis after a motion segment is completed. In a future reconfiguration, the feedback may be used closer to real time, e.g., in an outer control loop that adjusts setpoints to improve multi-axis coordination, or in a parameter estimating procedure, or in a monitoring procedure. Feedback signals from the two position sensors, the velocity sensor, and, later, feedback of current may be required in a temporally correlated manner, for more advanced monitoring and control. The control scheme may require modification for quicker response to signals that require mostion stoppage.

Interactions with related functions: During earlier stages of the development cycle, it should be possible to test the physical axis by itself with minimal dependence on the rest of the control system. This requirement implies collocation of axis control software and the device drivers to interface its sensors and actuators. The position sensors produce pulse trains that are counted by a pulse counting device, **counter**, the velocity sensor produces an analog voltage signal that is sampled and digitized by an analog input device, **analogIn**, and the power amplifier requires an 8-bit digital signal, interfaced through a digital output device, **digitalOut**. The devices are interconnected to the processor through a shared bus, **transferDevice**, employing memory-mapped IO to transfer data between the IO-interfacing devices and the processor on which the axis control software will be executed. Setpoints for motion control are updated at every servo loop interval.

Other reconfigurations required: Later, several coordinated axes may be in the same subsystem. The motion coordinating software and outer control loops may also be in the same subsystem. The IO-interfacing devices may be in a different subsystem for efficiency through batching multiple IO accesses. The interfacing and transfer hardware may be

replaced in the future. The processor may also be replaced in the future; however, the OS and development tools will present the same software interfaces to applications.

4.11.2 Reuse and adaptation of library components

We review and select library classes in an outside-to-inside sequence. The PeriodicTask class is specialized to the AxisTask class (Figure 4.2). The functions are specialized to refer to the axis-specific objects. Amongst its constituents, the SchedParam class is used without specialization, and the classes, Resources, FSM, and Ports, are specialized to the classes, AxisResources, AxisFSM, and AxisPorts, respectively, as described next.

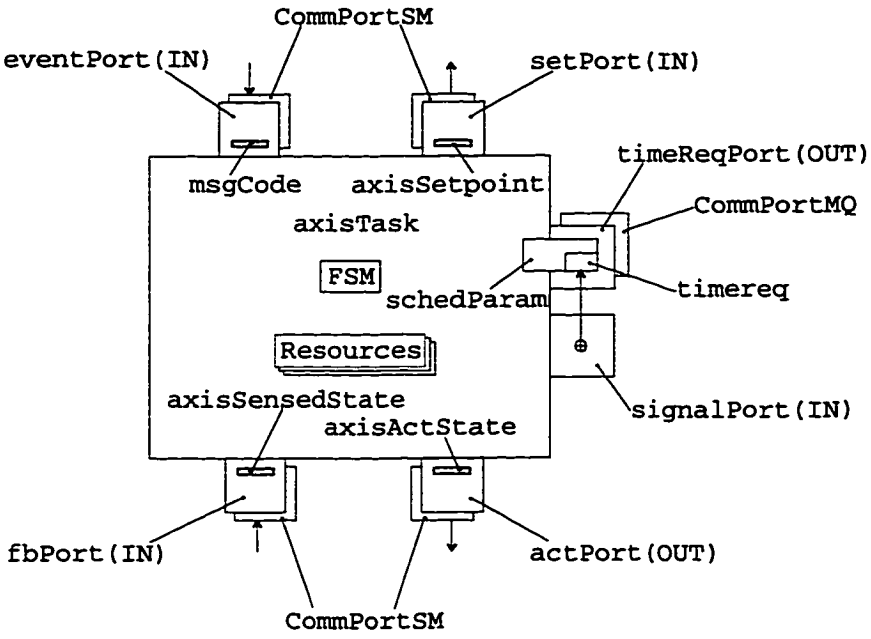

Figure 4.2: Structure of a hard real-time task with short periodicity — one-axis example

Axis resources: The main work-performing objects included in the AxisResources class are axisX (TranslationalAxis class), pidCtrl, counter, analogIn, digitalOut, and transferDevice.

Ports: A single axis task requires four types of interactions, supported with a Port object dedicated to each of these interactions — an incoming port, fbPort, for axis feedback, an outgoing port, actPort, for axis output, an incoming port, setptPort, for setpoints to the axis, and an incoming port, eventPort, for events affecting control flow in the axis. The fbPort may also be used to communicate axis sensed state to some monitoring, logging, or status reporting tasks, running at a lower priority with *read-only* access, so that these secondary tasks do not interfere with the primary work of the axis.

Finite state machine to control an axis: The axisTask · fsm structure remains unchanged, because the functions and their control flow logic are the same as for the Period-

icTask class. The next-level fsm (within the AxisTask->execute() function) requires a state transition table specific to the functions of an axis.

4.11.3 Creating the needed instances of class library components

The TranslationalAxis constructor is also responsible for creating objects directly referenced in it, namely, axisSetpoints (TransaxisSetpoints class), axisSensedState (TransaxisSensedState class), and axisActState (AxisActState class). Other objects in TranslationalAxis, included by value, are created without further explicit coding.

The fbPort object is set up with the *shared* buffering option, and its commPort is instantiated from a specialized class, CommPortSM, which provides a *shared memory* mechanism for interprocess communication. The msgCode objects correspond to the functions, get_axisSensedState, get_actualPosition, get_actualVelocity... to be exposed.

The actPort object is set up similarly, where the msgCode objects correspond to the functions, set_axisActState, set_axisOutput ...A similar shared memory link is set up for transferring motion setpoints to the axis control task, through an incoming port, setptPort, with the msgCode objects corresponding to the functions, get_axisSetpoints ...These three ports provide an efficient interprocess communication for the main operation processServoLoop(). A fourth port, eventPort, is used for events affecting control flow in the axis software. This port is also used for various setup and configuration changing functions.

4.11.4 Initializing the created components

Since most of the created objects are configurable by assignment of data member values, the next step is to provide the values for these assignments.

Assignment of pointers to abstract objects: Polymorphism has been used in several cases in the Axis model through inheritance hierarchies, in order to defer the selection of the specific object subclass that will provide the service offered in the superclass interface. For example, the pointer to pidCtrl is assigned to the axisCtrl object pointer which was automatically generated within the axisX object. Similarly, we assign pointers to the IO-interfacing device drivers.

Means of value assignment: Application-specific "constants" may be assigned to the corresponding data members in several ways. A technique that maximizes reuse is to initialize the created objects from corresponding persistent objects. In the absence of persistent object services in the suite of development tools, an alternative technique is to store the persistent data in a file. This technique requires additional programming specific to each application. In case the number of reuse applications is too small to justify the creation of persistent objects, a third technique is to include the data in the code for the AxisResources class. This technique requires the least development cost, but the initialization code is the least reusable.

Axis control software may be configured by data value assignments. For example, when the motion objective, xAxis · motionObjective, is assigned the value, *minimum_jerk*, a corresponding algorithm is selected. Similarly, when xAxis *cdot* controlMode is assigned the value, *position*, the position control strategy is selected.

Limited reusability of an AxisTask object: The xisTask object is reusable only for a particular configuration of the software components in it. Its class will be changing frequently because it is affected whenever any of its constituents is changed, even though changes may be minor. Secondly, its reuse is limited to the case when axis control is an independent task. Similarly, its constituent classes, AxisPorts, AxisResources, and AxisFSM, also have limited reuse. A future axis control may require a different number and type of ports, thus affecting the AxisPorts class, or may have a different AxisCtrl object or different device drivers, affecting the AxisResources class. The stTable component for the fsm within the execute function of AxisTask is slightly more reusable than AxisResources because of polymorphism (Section 4.11.4).

In general, we make two observations. First, larger scale aggregations will be less widely reusable than their component classes. Secondly, specialized component classes will be less widely reusable than their generalizations.

The model is evaluated relative to General Needs 1–8 identified in Chapter 2 and derived General Need 9.

4.12 Evaluation of model

The developed model is evaluated for its effectiveness relative to general needs identified earlier, considering the following general evaluative questions. Are these needs supported satisfactorily? How does it compare with alternatives considered? Is this model an improvement over existing models? If there are tradeoffs, what is the rationale (justification) for the selected approach? The general needs are clustered and discussed in groups — reconfigurability in scale, closeness of interaction and reconfigurability in behavior. Correctness of operation, including specified *timing behavior* or ordering of events and actions are treated as directly implied requirements. *Ease of (re)configuration and integration* is an indirect requirement.

A key architectural strategy for reconfigurability is to defer and separate the design specification of building blocks (framework of reusable classes) from system design and integration decisions such as the following:

1. Organizing function sequences into concurrent units of execution by identifying the independence or degree of coupling from the application domain requirements.

2. Packaging these units into OS-managed tasks.

3. Assigning tasks to processing resources.

Next, we show that the architecture is scalable in number of similar axes and sensors with incremental effort. The significant engineering cost in scaling is the validation analysis for timing requirements.

4.12.1 Scalability in number of axes

General Needs 1 and 6 require that a control system be scalable in number of objects, e.g., axes and sensors. Some closely implied requirements are *scalability without degradation of the required run-time performance* of the system and its *performance-to-cost ratio.*

Consider the scale-up in number of similar axes of a system originally designed to implement one axis of control on dedicated processing resources. We evaluate the reusabiliy and

adaptability of the software in the following cases, and show that the reusability is primarily at the level of the building block classes, which are reusable in various reconfigurations for scale-up.

Replication of physical resources: If adding identical processing resources for each axis, there is no change in the software, originally packaged in an object of the AxisTask class, except assignment of values specific to each axis, and specific to the communication interfaces with the rest of the system. Analysis of inter-processor communication resources required and available is also required. The latter is the larger engineering expense.

Replication of task on same processor: If replicating AxisTask objects running on the same processor, the software changes are similar to the previous case, whether each axis is in a separate OS process or whether each axis is in a separate thread with coordinated threads running in the same OS process. In the latter case, the buffering option of *immediate* may be used. However, an analysis of the processor utilization is required as described in Section 4.10.3. Additional analysis of resource requirements in comparison to available capacity is also required. The analysis is application-specific and its results are not widely reusable. If the reserve capacity is low, this analysis may be a significant engineering expense.

Coordinated axes in same task: If all the axes are in the same task (whether it is a thread or a process), some modularity is given up for efficiency of system resource utilization when motion of the axes is coordinated. There is slight additional software packaging cost to add the resource, port and msgCode objects for the additional axes, and to augment the state transition table in the fsm. However, no change is required in the software of the Axis class hierarchy, except assignment of values specific to each axis. Analysis of resource requirements and capacity is similar to the previous case. The expense of packaging is application-specific and not reusable, except for some narrow family of applications in one organization.

Mixed types of axes: If the original axis was a translational axis and a new axis is a rotational axis, then the AxisTask object cannot be reused identically. If the classes for rotational axes had not been constructed earlier, these classes have to be implemented in addition to the changes noted in the previous case. However, the implementation of rotational axes is similar to the translational axes. Once implemented, these classes are reusable assets, amortizable over future applications.

4.12.2 Scalability in sensors

The model provides a generic class for the three basic types of transduced signals, an analog waveform, a train of pulses, and a digital logic level (bit, or group of bits), and the classes to transform the raw inputs into data types for every type of physical quantity defined in the *ISO* standard for measures and units. Kinematic transform models are also included for the common types of motion feedback sensors. In the case that the scale-up employs the same types of device drivers, no additional code is needed to create instances of common sensors. An instance of a previously modeled case is added by selecting and instantiating the corresponding classes. We consider two common cases of adding sensors.

Axis state sensed at servo update interval: In order to add sensors closely coupled to one axis, e.g., axis drive current, and sampled at the axis servo update interval, the axisSensedState object is instantiated from a specialization of the TransaxisSensedState or the RotaxisSensedState class. Since no new functions are needed, the effort is primarily the labor of aggregation and assignment of values specific to the "constants" of the added sensor. The software for the rest of the Axis classhierarchy remains the same. The software for the AxisTask class is changed to add a new axisSensedState object and the device driver object associated with the new sensor. The fsm object is also changed for the additional functions (accesses and conversion) to be performed. The analysis of the processor utilization must be revised to reflect the time-costs of the added functions. However, the re-calibration effort is not significant when the device drivers are of the same type, because the time-costs are predictable.

Sensing on same processor at period shorter than servo period: If the purpose is data acquisition only, the basic building block classes are reusable. However, this type of sensing function must be placed in a separate instance of the PeriodicTask class. Since its time period is shorter than the servo control task, the sensing task must be assigned a higher priority. A time-loading analysis must be performed. Since the data acquisition functions are similar to other data acquisition in the axis software, the effort to obtain execution time-cost of these functions is a predictable activity and not a significant cost. However, if the processor utilization, τ, is high, the schedulability analysis becomes more complicated. For example, sensors for force, acceleration, and acoustic emissions may require sampling at an interval as low as 0.0001 seconds, and the execution time to acquire data may be in the order of 0.000035 seconds on a common computer platform, in our experience, utilizing 35% of the capacity of an Intel 486-50 Hz processor, without considering system overhead.

Signal processing: Often, the data acquisition is followed by some processing of the acquired signal. If the signal processing period, *prd_sp*, is smaller than or comparable to the axisTask period, then the case is similar to the previous cases. If *prd_sp* is some large multiple, N, of the axisTask period, then it is implemented in a separate instance of PeriodicTask. The engineering effort may be reduced by decoupling it as follows. The axisTask object is extended to produce a sequence (length $M \gg N$) of successive data values and to make the sequence available in a separate port. Corresponding changes are required in its constituents as follows: add a sequence building object, include its sequence-building functions in the fsm · stTable object, add the port, and map the sequence object into that port. The signal processing task is run at a lower priority. Since the acquired data is buffered, its task period may be allowed a larger variation. The cost of the new sequence building object, if designed as a class, is amortizable over future applications. The cost of the processor utilization analysis is still application-specific, as in earlier cases.

4.12.3 Closeness of interaction

The architectural design separates the code for building blocks from the composition of their functions into execution sequences, and subsequent packaging into executable tasks. Thus, objects requiring interaction with axis control at the servo loop period may be included in the same task as the axis or axes. This assures the desired execution order. It requires a change in the composition of that task, but not in any of the building block objects.

Ability to specify close interactions across objects: The axisSensedState object is updated at every servo loop interval for every axis of motion in a system. The architecture allows other software objects *read-only* access to it. If a software object, e.g., *cross-axis coupling control* must not only read the sensed state, but also adjust the axisSetpoints or change its state, that object may be included in the same task. If the servo controllers for the axes were prepackaged into an executable, the *cross-axis coupling control* would have to run in a separate task at the same period — a less efficient alternative for several reasons. First, its output would be applied with a delay of at least one servo loop cycle. Secondly, a separate task would incur additional system services overhead.

4.13 Conclusion

The execution model that was developed for one-axis motion control yielded abstractions (class structures) that are applicable to continuous process monitoring and control in general. The execution control flow structure (fsm) supports the integration of discrete events with continuous process control. It may be applied to control the flow of any program that fits in the fsm model. Secondly, it provides an organization of these abstractions that facilitates reconfiguration and adaptation. The process used in developing this model facilitates incremental evolution, as demonstrated through three iterations of development and evaluation. Experiments with the use of this model showed that the associated execution overhead was insignificant in comparison to system services overhead. It did not increase execution time variations. It was acceptable for implementing the hard real-time, short-duration, short time-period servo-motion control applications investigated.

Improvements in software productivity, timeliness, and quality in the domain of reconfigurable hard real-time axis motion subsystems is a field of long-term research. Further investigations in the single-axis application domain will help answer a number of research questions in domain-specific software architectures for integration of multi-axis motion and process control subsystems. Some promising near-term research issues are reviewed below.

Configuration rules: While the configuration rules and constraints given in our model make application development easier, further study is needed in two directions. In the near-term, additional rules should be developed to further simplify the application design process. Longer term research is also needed to develop less constraining rules and a well-defined design process for applications that can tolerate larger timing variations. The architecture does not support dynamic installation of new software components. In addition to unknowns in computer science, there are also unknowns in control theory when discontinuities arise due to dynamic reconfigurations.

This architecture provides the basic framework for timing specifications and studies which will be needed for the suggested research.

Application development tools: After the design process and configuration rules are well developed, tools should be developed to automate the routine aspects of the procedure, in order to reduce the cost of configuration, integration, and quality assurance. Setting up an application composed of a number of processes distributed across several computers has been much more difficult than constructing the passive objects and running them in a single program. By difficulty, we mean that the required skill level, effort, and duration were relatively higher. The difficulty could be reduced with domain-specific tools for configuration,

integration, and verification of distributed control systems.

Persistence: A number of parameters, e.g., equipment characteristics remain unchanged for relatively long periods of operation. The traditional approach is to specify their values in a file containing configuration parameters. Additional coding is required to transform the ASCII contents of files into data for a program. This cost could be reduced by providing persistent objects to hold these values across different runs of the application.

Software Group	Reconfiguration effort element	Cost
Type of change: Modify the control flow logic in an existing program; *No new state, condition, event, object, function.*		
FSM	Create, add condition or action object	M
	Create transition object; add to stTable	ML
	Remove unwanted transition from stTable	ML
schedParam	Calibrate and update execution time	M
Whole system	Analyze time-loading factor	ML
	If increased, re-engineer as needed.	?
	Example cases:	-
	- $\tau < 0.25$, $Toh_t/c_t < 0.01$	L
	- $\tau > 0.6$, $Toh_t/c_t > 0.3$	H
Type of change: Substitute or replace work-performing resources.		
Resources	Include class; add instance of class	L
	Initialize the object	M
FSM	Add object functions to decoding table	M
Type of change: Add external service; no new state, object, function.		
Ports	Create msgCode object; add to protocol of existing port	ML
	Create, add ParameterListStructure object	ML
FSM	Create, add event, condition, actions objects	M
Type of change: Add IPC to existing program.		
Ports	Create, add Port objects,	M
	- incl. CommPort objects	MH
FSM	(Top-level) modify initialization to apply new port	M
	(Executing-level) add fsm/state/transition to use new port	M
Whole system	Analyze communication overhead	-
	-shared memory	L
	-event signaling	M
	-messaging	MH

Notes:
Effort elements add to previously listed elements incrementally for each type of change.
Cost is rated relatively as follows.
(L)=low; (ML)=medium low; (M)=medium; (MH)=medium high; (H)=high.
(?)=unknown.
Assumptions:
The needed classes exist.
Application conforms to the given architectural rules.
Development and execution environments are less constraining
than the limits described in Appendix A.

Table 4.5: Reconfiguration effort elements for common software changes.

CHAPTER 5

Multi-axis motion coordination

The most common form of coordinated motion in conventional machine tools and robots is the coordination of independently driven axes of motion which are in a kinematic configuration that controls the motion of a work point (e.g., center of a cutting tool). We denote the group of axes "bound together" to produce a specified work point trajectory as an AxisGroup.

As in the case of single-axis subsystems (Chapters 3–4), the terminology in industry is inconsistent across different types of manufacturing automation due to differences in their dominant applications and nature of motion coordination. Coordination of motion must be described in a unified scheme that is more general than the view in any single field. The AxisGroup model (Class Structure 27) provides the needed generality, reusablity, and extensibility as explained in the next section.

The model builds on the subdomains identified and formalized in Chapters 3–4. In conclusion, we summarize the reconfigurability and reusability achieved, and further research issues identified.

5.1 Motion coordination by an axis group

The primary function of an AxisGroup is to transform the input motion specification into a series of setpoints (AxisSetpoint objects) for each of the axes in that group, equispaced at the update intervals of the axis control loops. This transformation is performed by the move(...) function (Section 5.1.2), supported with a number of preparatory or setup functions.

5.1.1 Strategy for reusability of motion specification software

An executable specification of motion to process a part (workpiece) is prepared significantly in advance of execution time. This specification is known as a part program (in numerically controlled machine tools), a robot program, a process program, or a motion process plan. As a result of executing this specification, a manufacturing system should be able to produce a part within the specified tolerance at the programmed rates within allowable variation, and, if needed, correcting or compensating for the dynamics and kinematics specific to the kinematic mechanisms involved in its execution.

Traditionally, a (non-executable) motion process plan (typically specification of a path and rate along that path, e.g., output of APT or APT-like output from CAD/CAM sys-

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
VelocityProfileGenerator * VPG
CoordinateFrame * baseFrame //in which moves are specified.
CoordinatedAxes coordinatedAxes
KinStructure * kinStructure //transformation between
axis coordinates and workspace coordinates.
ToolPartTransforms * toolPartTransforms
AxisSetpointsArray *axisSetpointsArray //size=num of axes in group.

Accessor functions for following data members:
int qSize
LinearVelocity feedrate //Tangential feedrate for linear moves.
ratio_measure feedrateVariationHi
ratio_measure feedrateVariationLo
LinearVelocity traverseRate
double feedrateOverride
LinearJerk jerkLimit
Boolean inPosition //Set when inPosition TRUE for all axes
Boolean newMotionRequest //Set TRUE when client supplies new ...
ACC_MODE accMode //enum
time_measure axisUpdateInterval

Other member functions:
void feedHold()
void feedResume()
void estop()
void move(KinematicPath *kinematicPath, LinearVelocity *vel)
void traverse(KinematicPath *kinematicPath)
void setNextSetPoint(length_measure *dist)

Class-structure 5.1: Interface of class AxisGroup.

tems) is prepared independent of the machine tool on which it may be executed. A post processor is used off line, ahead of time, to transform the machine-independent process plan to a machine-specific part program. This early binding or commitment limits late-stage flexibility in its allocation to processing equipment, or, conversely, renders the post-processing investment obsolete if the required equipment is not available and the work has to be assigned to some other machine. The accumulated lifetime cost of application software obsolescence is far greater than the lifetime cost of the corresponding machine tool controllers. Secondly, it forces use of conservative equipment parameters, or conversely adds near-run time cost to "fine tune" the part program. Thirdly, it increases the risk of producing a specification that exceeds the capabilities of the selected machine, because similar machine tools differ in performance due to differences in aging and in the variety of controllers typically retrofitted at different stages in the life of these machine tools. Again, the cumulative lifetime costs of these compromises is far greater than the lifetime cost of the controllers. Therefore, it is desirable to perform machine-specific transformations of a machine-independent process plan at the machine tool itself. For example, to maintain the specified tolerances, "overshoot" and "underReach" (Class Structure 3.4) must be limited e.g., by modifying feedrates, calculating the distances for acceleration and deceleration.

This need was recognized decades ago, initially with the idea of performing post-processing of APT or APT-like motion process plans at the processing equipment itself, culminating in efforts to establish a new standard. However, its adoption and implementation was hampered, in part, by the limitation of the prevailing standards and conventions of

input interface to a numerical controller (EIA RS-274), the lack of a more suitable motion specification interface, and the inability of prevalent machine tool controllers to execute such specifications. The AxisGroup model specifies the needed interfaces between the controller and the motion process plan. We introduce this specification with the move(...) function of the AxisGroup class.

5.1.2 The primary function - move

The move(...) function (Class Structure 5.1) may be invoked after the appropriate setup or preparatory functions have been executed. It sets its computational result in the member axisSetpointsArray. It uses the KinematicPath parameter (Section 5.3) for geometric interpolation and the vpg object (Section 5.2), for calculating the velocity profile. The traverse() function calls the move() function with the traverse-rate as the LinearVelocity argument. The move() function also causes invocation of the initAccDecProfile() function specific to the type of kinematicPath.

The data members to be set before using the move(...) function are as follows:

coordinatedAxes. The identifiers of the coordinated axes, set as part of the initial configuration.

baseFrame. The coordinate frame relative to which the KinematicPath is specified, given through the motion specification program or through a user interface.

kinStructure. The transformation between axis coordinates and workspace coordinates, set as part of the initial configuration, and obtained from the machine tool model.

toolPartTransforms. Pointers to the three kinematic transforms (nominal, quasi-statically calibrated, and dynamically compensated) between the tool center (work point) and the baseFrame.

VPG. A pointer to the VPG object to be used, set as part of the initial configuration.

axisSetpointsArray. The size of the array is equal to the number of axes in the group to hold a pointer to the axisSetpoints object for each axis — these pointers are set as part of the initial configuration.

Rate parameters. The AxisGroup object must be initialized with safe or common-case default values of jerkLimit, accMode (trapezoidal), traverseRate (0.0), feedrate (0.0), feedrateOverride (1.0) feedrateVariationHi (0.0), and feedrateVariationLo (1.0). After initialization, values specific to a move must be set before the move is initiated — the values persist until explicitly changed.

Status variables. Various status type data members must also be initialized — inPosition (TRUE), newMotionRequest (FALSE).

5.1.3 Flexibility through proper modularization

The role of the historic interpolators used in robots and machine tools is distributed across a number of objects, in order to support reconfigurability through the least number of changes necessary for a particular configuration. This reduction in number of items to be changed is afforded through the use of Rule 1 (Chapter 2). Although a simple

numerically controlled machine tool treats all axes to be coordinated at all times, in the case of reconfigurable machine tools, the particular axes of a kinematic mechanism that are coordinated for a move are not always the same. For this reason, we have modeled the *kinematic mechanism* and the *coordinated axes* as two separate concepts. For example, consider three cases of a machine tool with a reconfigurable spindle axisS as follows. In case 1 (most common use case), axisS may be performing motion under velocity control only, independent of other axes participating in the same tool-to-workpiece mechanism. The spindle may be considered to be a group by itself. In case 2 (helix machining case), axisS may be performing motion under position control and velocity control coordinated with other axes in the same mechanism. The spindle is a part of this group only for the interval that helical path generation is required. In case 3 (relocatable spindle), a spindle may be indexed to a different location and become part of another group of axes moving in coordination. At appropriate safe points in the work cycle, the group membership of axisS may be changed by manipulating the coordinatedAxes member, and setting proper values of other data members in the corresponding AxisGroup object. The application of Rule 1 to other forms of modularization is discussed in Sections 5.3–5.5.

5.1.4 Modeling issue of move parameters

Should a move(...) function have self-sufficiency in its parameters, so that it does not depend upon prior state information (statelessness strategy)? This approach is taken in the NIST RCCL. Or, should it include the minimum number of parameters (only the ones that change), depending upon prior specified values as default values (statefulness strategy)? EIA RS-274 economizes on data transmission using the latter approach. These questions are representative of a generic modeling issue.

The *statelessness strategy* makes it easier to implement the function — it makes the external data used by the function explicit, minimizes the dependency on the class design, and makes it easier to switch to other objects dynamically. The parameter data items have to be stored and managed outside the AxisGroup object somewhere else in the application, or they have to be retransmitted with each function invocation. The *statefulness strategy* reduces the size of the function signature, which makes it easier to understand, and reduces the likelihood of data entry mistakes in programming the application. It also reduces the traffic volume in function parameter objects or their references. In both approaches, the number of data items used by the function and the transformation performed by it remain the same. In that respect, the design of the function is equivalent and there is no significant difference in its execution efficiency.

We adopted the *statefulness strategy* for the following reasons. It clearly simplifies the development of a common case application which does not require dynamic change of AxisGroup objects. The parameterization used in the AxisGroup class design is based on a mature motion engineering foundation (Rule 3), as discussed below. Therefore, we expect these concepts to be stable. The immature aspect of the model is the grouping of these concepts in the particular classes in our model. The model needs further validation through actual usage. Further refinement and reorganization may be required. However, the cost of moving data members from one class to another is not significant, as shown in the development of the single-axis model (Chapter 3).

5.2 Velocity profile generation

Given the velocities at the start and end of a path element and the required nominal velocity and acceleration profile (ACC_MODE) along the path element, the velocity profile generator (Class Structure 5.2), computes the target tangential velocity at each point P_i along a path element $\forall i : 1 < i < n$, where n is the number of axis servo loop intervals occurring during the motion across the path element, $i = 1$ corresponds to the start of the path element, and $i = n$ corresponds to its end. A vpg object uses services and data from a number of other objects in performing this computation, as described in the following sections. Following Rule 1, it does not have within it any hard-coded computational knowledge of various types of path elements, acceleration profiles, axis dynamic limits, and the kinematic model of the machine.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
AccDecProfile * accDecProfile
LinearVelocity Vi //initial velocity along the KinematicPath
LinearVelocity Vf //desired nominal velocity along the KinematicPath
LinearVelocity Ve //final velocity along the Kinematic Path

Private members:
length_measure instantX
length_measure instantY
length_measure instantZ
LinearVelocity instantVelocity
int step
time_measure samplingTime //Sampling time of the servo loops.

Class-structure 5.2: Interface of class VelocityProfileGenerator.

Motion rate parameters and limits: The FSM executing the functions of the AxisGroup object sets the parameters corresponding to the type of process being executed, prior to each move. The move function uses its velocity parameter and the preset values (e.g., feedrate override), and computes the parameters for the vpg. Recall that the Axis setup model provided for setting dynamic limits, e.g., force, jerk, acceleration, deceleration, velocity, (Chapter 3, Class Structures 3.4–3.6, B.3). Section 3.4.1 described how these limits may be changed corresponding to the type of process being performed in a particular move or sequence of moves. The FSM that executes functions of the AxisGroup object assigns references to the appropriate// dynamicLimits, tolerances, and axisError objects for each axis, thus applying motion constraints corresponding to the needs of the process, e.g., liberal limits for idle traverse, conservative limits for machining, more conservative limits for inspection, and so on. The vpg has references to these objects as well as the axisTravelCap object (Class Structure 3.8), and uses the limits specified in them to calculate the velocity profile of a move. The AxisGroup object must also check that the travel limits are not violated in the requested move (remain within the soft travel limits in reverse and forward directions). For this purpose, it also has a reference to the travelLimits object for each axis (Class Structure 3.7).

Evolution of velocity profile generation The model has been prototyped and tested for the trapezoidal acceleration mode. Other acceleration modes have to be programmed and tested to evaluate the reusability of existing functions. The base class, VelocityProfile-

Generator, is directly usable for the one-axis case, as well as a multi-axis case. For example, the same class is used in the JogHome class to jog or home an individual axis. Our prototype implementation has tested the model only for the case where velocity can be reduced to zero. Blending velocity across path elements requires looking ahead to process the next path element, an additional velocity profile generator object, changing the association of a path element and a velocity profile generator object, and requiring additional exchange of information across velocity profile generator objects. The model provides these capabilities. However, ease of application has to be evaluated.

5.3 Path specification from client

The path of the work point is specified in the work space coordinate system, in a manner consistent with the STEP entity *kinematic path* [22] (KinematicPath class in our model), where the path is a chain of path elements, each of which includes its shape and accuracy specification. As specified in STEP, a kinematic path may a composite path (CompositePath class) or a single path element (PathElement class), and the latter may be specialized to describe a point to point move (PointToPointPath class), a straight line move with velocity in each dimension proportioanl to the distance moved along the line (LinearPath class), or a circular arc (CircularPath class). Other specializations of PathElement may be added in the future, for other path shapes. KinematicPath may be specialized for non-uniform rational B-splines (NURBS) by adding the additional parameters specific to NURBS. Then the function move(...) will receive a reference to a NURBS object as a parameter, and perform the computations for the complete NURBS path specified. Thus, the move function specification allows a very compact specification of motion along a free form path or a section through a sculptured surface, e.g., in machining or inspecting dies, molds, aerofoil surfaces, and blends across different surfaces of a turbine blade.

Special features of the path model: We extend the STEP model of a kinematic path by adding functions for path-geometry related calculations along the path, so that it can perform the geometric aspects of traditional interpolation. The function length() calculates the length of the path. Given the total length of the path, the vpg (Section 5.2) calculates the number of servo loop intervals, n, required for the move, and the displacement along the path in each interval. The nextPoint(length_measure dist) function of the path element object computes P_i, $\forall i : 1 < i < n$, given the cumulative displacement dist. This modularization localizes shape-specific knowledge to the respective path class. For example, when a new path shape is to be specified, a corresponding subclass is developed, with the functions length() and nextPoint(...) specialized to that subclass. In the same manner, functions related to velocity profile do not have to be implemented in the path class hierarchy.

Blending In the case where a sequence of moves consists of discontinuous path elements and the nominal velocities in each element are different, compromises between the velocity and path specifications are required at the juncture of adjacent path elements. These adjustments in the move specification require a look ahead at the specification of the next move. Consider the example of Figure 5.1 showing two successive moves: path element, lp1, from point, P1, to point, P2, at nominal velocity vf1, and path element, lp2, from, P2, to point, P3, at nominal velocity, vf2. A separate vpg object, vpg1 and vpg2, is associated with each path element, lp1 and lp2, respectively — vpg1 an dvpg2 use the respective

specification of allowable deviation, d1 and d2, from the nominal path to compute the initial velocities, vi1 and vi2, and final velocities, ve1 and ve2, respectively. The objects, vpg1 and vpg2, exchange information to arrive at the same values for the final velocity, vel, for lp1, and the initial velocity vi2, for lp2. If specified constraints cannot be met, an error is returned.

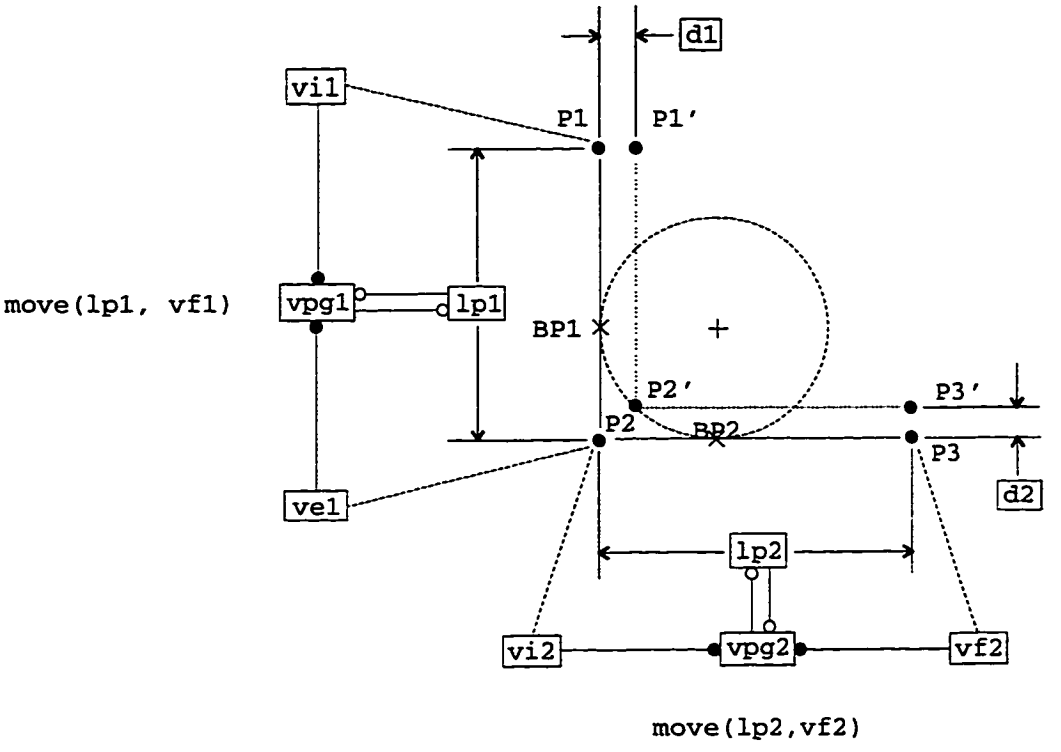

Figure 5.1: Velocity profile blending across two path elements.

Buffering input There are several reasons for buffering input to the AxisGroup, e.g., to provide look-ahead for blending, to prevent starvation, and to loosen the temporal coupling. The data member, qsize, is used to specify the buffer size.

Source of base coordinate frame for kinematic path In conventional numerical control, there is no unified way to specify the reference coordinate frame. The value of a variable, *part origin* (or equivalent name), is supplied through the user interface. EIA RS 274 provides the format to specify three planes of rotation of the work coordinates. Extensions also exist to specify lateral inversion (mirror-image) of the work coordinates. In our model, the data member, baseFrame, is provided to capture and combine the results of applying the specification of an origin, plane rotation, and lateral inversion. Thus, software conforming to this model has a uniform interface with more generality through specification of any combination of rotation and translation. The transformations from the different forms of user input to the baseFrame (Figure 2.2) are external to this model. This uniformity improves the reusability of user-specified process programs and plans and makes them more compact. For example, when processing multiple workpieces in the same setup, but in different locations, or multiple recurring patterns on the same workpiece, the corresponding repeating sequence of moves may be packaged as a routine that may be repeated simply with the change in the baseFrame value.

5.4 Transforming workspace to axis space coordinates

The information for the transformation of workspace coordinates to axis coordinates is encapsulated in a KinStructure object (Class Structure B.6) for the kinematic mechanism in which the axes of this AxisGroup are kinematically related. Once again it is shown that basic concepts, represented in class structures identified in developing the Axis model, are reused in the coordinated motion of multiple axes. In the simple case of a machine with three orthogonal translational axes X, Y, and Z, the motion specification coordinate frame (the baseFrame) may be aligned with the machine tool coordinate frame, i.e., when the axes are at their home positions, the workpoint is at the origin of the workspace coordinates. Then kinStructure is an identity transform matrix.

Offsets of part or tool: Even in the simple case described above, often the actual location of the workpiece or the effective work point of a tool is slightly offset from the nominal specification — traditionally known as part offset and tool offset respectively. If multiple workpieces are processed in the same setup or if several different tools are used in processing a workpiece, then an offset is associated with each of them. Information about the particular part and tool offset involved in the process at any time is brought into use through the ToolPartTransform object — the offsets are treated as a Coordinate transform and described through a reuse of the CoordinateFrame class. Basic kinematic errors and thermally induced errors of motion are treated in the same manner, consistent with the scheme devised at NIST [17].

5.5 Output to the axes

Given a P_i, $\forall i 1 < i < n$, computed by the KinematicPath object, the setNextSetPoint() function of AxisGroup applies the inverseTransform function of the (compensated) KinStructure object to produce a set of values for axisSetpoints objects, corresponding to each servo loop update of each axis.

Buffering output: The AxisGroup has the information to produce a sequence of axisSetpoints objects for each axis well ahead of the axes' consumption cycle. The decision about the amount of buffering is left to the application design. When the AxisGroup setNextSetPoint() function sets the respective axisSetpoints object of each axis directly, the data is used within one servo loop interval. When AxisGroup transfers axisSetpoints to Axis through a *shared* buffering option, it introduces a delay of one servo loop interval. Larger buffering costs extra space, and incurs a higher risk of wasted computation in case a discrete event requires a change in plan. Larger buffering loosens the temporal coupling between the producer and the consumers. However, extra software is required to transfer from the buffer(s) to the axes synchronously.

Interruption of motion: An interruption of motion results in setting the effective feedrate to zero, forcing the AxisGroup to produce the next setpoints for the axes to be the same as the current setpoints. The FSM executing the AxisGroup sets the appropriate state, e.g., hold or emergency, associated with the interruption. A corresponding state changing event is also sent to the coordinated axes.

5.6 Issues in exception handling

The AxisGroup object should also check for status, e.g., inPosition or exceptions, posted by each axis under its coordination. We created the AxisSensedState class for making axis status information available to interested clients, leaving it to the client to check the state information and favoring design simplification in the hard real-time part of the system by minimizing the reporting burden and variability in the axis task. Exception signals may be generalized as discrete events that the client FSM should process. In our model, we have provided one FSM for each task, and in our prototypes, we have mapped each task into an OS process. Each FSM has an Exception state. In our tests, the AxisGroup object has not been run as an independent task — it was run in a program that also has overall coordination software. Thus, there is only one FSM for the task – there is no distinct FSM for the AxisGroup. There is no software within the AxisGroup class to handle exception type discrete events.

Currently, our model does not have an assigned responsibility to handle specific exceptions generated by the axes — exception handling is left outside the scope of the AxisGroup class. A common general rule is that the client of a service should receive and handle status and exception information. Although an AxisGroup object is a source of data to the axes, in a reconfigurable system, it may not be the only source of axisSetpoints data, e.g., a cross-coupling controller may be the immediate source of data to the axes. At times, special monitors may be assigned the responsibility of checking status and handling exceptions. If the immediate source of data is to receive and handle the exceptions, then the exception handling changes every time the immediate source of data to the axes changes. If left to be application-specific, exception handling could amount to significant level of effort, and also result in non-uniformity, which, in turn, could cause significant integration labor.

5.7 Integration of cross-coupling control

Control involving cross-axis coupling (CCC) requires input of an AxisSetpointsArray object from the AxisGroup and AxisSensedState objects from the respective axes.The CCCs output is a corrected set of axisSetpoint objects for the coordinated axes. The structural support for the data input and output exists in the model. However, there are several unresolved questions in ease of integration of CCC. One issue concerns ease of turning CCC on or off — representative of switching any auxiliary controller in or out. It involves a change in dataflow, which could be time-consuming, as discussed next.

CCC in separate program

If the AxisGroup is in a separate program from the axes and CCC is added as an additional program, then dataflow connections across processes have to be changed. In our model, each of these processes would be an instance of some class derived from a PeriodicTask class, equipped with Port objects to facilitate set up of inter-process communications. Port connections with hard real-time tasks, e.g., axes, are established as part of a static configuration step performed at startup, in a special state for configuration. We have not attempted a switch to a configuration state from an execution state. In addition to a change in port connections, it will require processing through an initialization step. In our experimental prototyping, we have found these steps to be error-prone — trial and error was required, and the resulting solution was hard coded. If each axis is running in a separate

process, another task management issue is created — the CCC task (a producer) and the axis tasks (multiple consumers) must run in a cyclic sequence. There are two aspects to the task management issue. First, switching the CCC in or out changes the task graph — some task management outside the OS is required to make this easy. Secondly, correctness of timing must be assured, when such controllers are switched in and out.

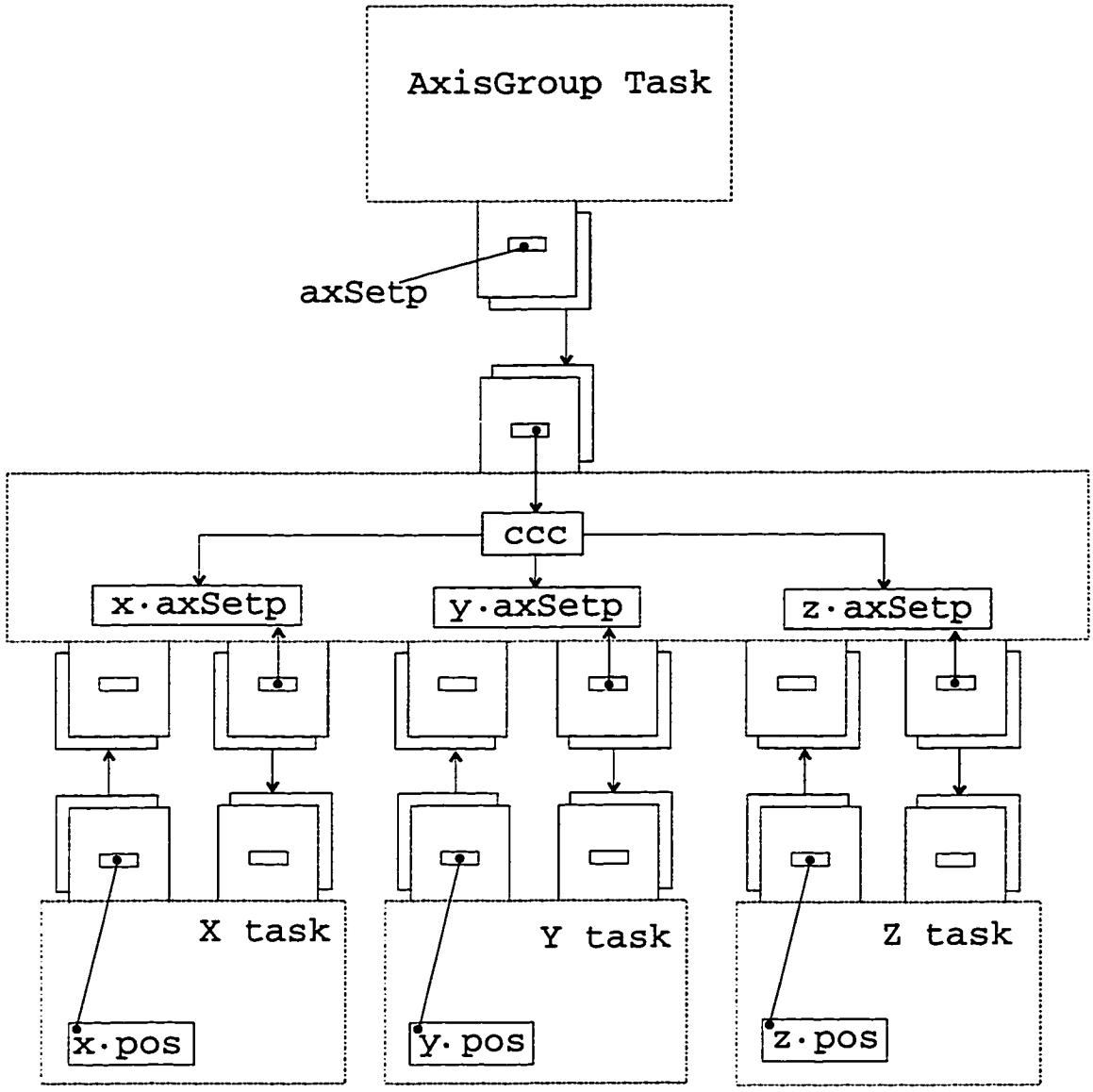

Figure 5.2: Retrofit of cross coupling control as a separate process.

CCC in same program as axes

If the CCC and all the axes are to run in the same program, the task management issue does not arise — timing and synchronization issues are simplified. However, a change is

required to an existing, working program, when the CCC is initially added. The disadvantage of touching an existing program is that errors may creep in when the change is made. In this case, the change consists of two parts (a) adding the CCC object,which is simplified when CCC is a predesigned prechecked class, the style used in our model, and (b) modifying the state machine, which is with our FSM model simpler than with conventional programming, but it still requires testing after a change.

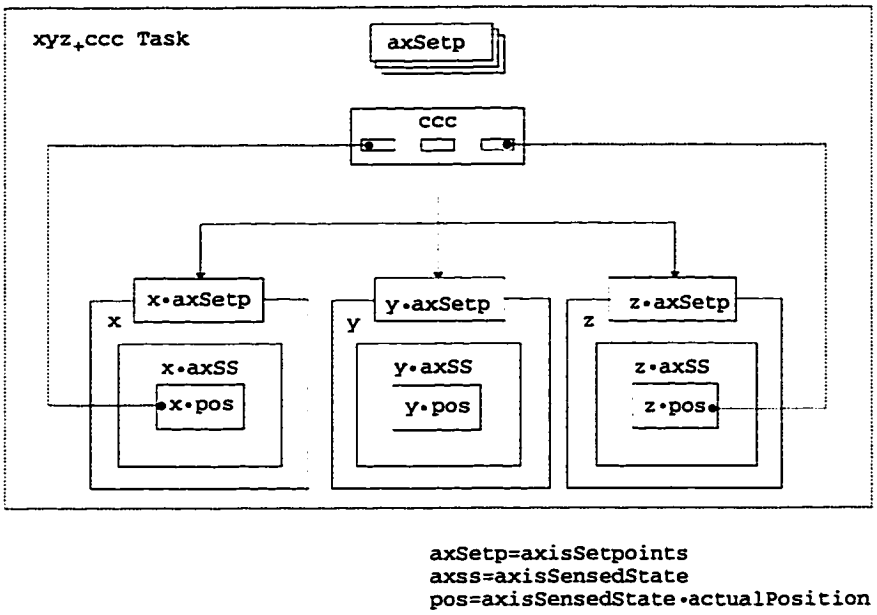

Figure 5.3: Safe, efficient, tightly-coupled integration of cross coupling control.

5.8 Recapitulation

We have shown that the AxisGroup model supports a larger number of configurations for a limited number of concepts (modeled in object classes) than the alternatives currently available. It provides much more generality, reusability, and extensibility than prior approaches.

The software architecture for multi-axis coordinated motion needs further evaluation for ease of application development, particularly in the synthesis of an application-specific FSM incorporating a variety of discrete events, e.g., dynamic swapping of objects and exception handling.

Concepts reused from axis model: We have shown that our domain model is evolvable. It reuses and builds on concepts established in evolving the single-axis model. The mechanical engineering foundation of measures was formally modeled while developing a single axis model. It was used to develop the model of space, in combination with classes for fundamental mathematical structures of matrices and vectors. Based upon the model of space, we built classes for kinematic structures and kinematic mechanisms.

Variety of kinematic configurations: It allows the formulation of new configurations of coordinated axes — not possible in current practice.

Ability to modify motion path specs: The model specifies a uniform way of modifying motion specifications, more general than the approaches used in practice.

Velocity profile generation: The model needs to be tested for ease of application to cases where blending across path elements is specified with smaller allowable deviations in path and velocity. Evaluation studies are also needed for programming various acceleration modes, to understand the adaptability of software classes developed earlier.

Kinematic transformations: The model needs evaluation for ease of application to more complex cases involving offsets and geometric and thermally induced errors of motion. This investigation is coupled with a related task of integrating the calibration of errors of motion and correction for thermally induced deformations.

Integration of other controller tasks: Research is needed to make it easier to integrate, switch in, and switch out controller tasks closely coupled with axes and intervening between the axes and their coordinating AxisGroup object, as discussed above. It requires investigation in the state transition path from the execution state through the configuration state, eventually returning to the execution state. It also requires development of a configuration tool that sets up the appropriate task graph, and application level task management, including pre-scheduling, which uses the task graph, for proper sequencing of task releases.

Exceptions and other discrete events: Where should exceptions generated by axes (and, in general, other objects) be checked and handled? How should such discrete events be integrated without causing detrimental effects on the timing of any task in the system? How can the associated programming and integration effort be minimized in a reconfigurable system? These questions are not easy to answer in a reconfigurable control system, as discussed above. However, if investigated specific to motion coordination in the near term, it is more likely that an economic solution will be found — it could be followed by a longer term investigation to generalize the approach to a wider domain.

CHAPTER 6

External Inputs and Outputs

Machine tool controllers have a wide variety of external inputs from sensors and switches and outputs to actuators, e.g., solenoids and motor drives. There is also a wide range in the required sampling and update rates or timing constraints, and the degree of coupling with various tasks and data in the controller. In the quest for maximum reconfigurabilty for a given level of cost and complexity, our architectural objective is to cover the most common cases by organizing software in a manner that reduces the volume and variety of information needed by a developer. Our approach toward this objective is to search for abstractions that allow a compact organization of information favoring the common cases, while allowing extension to cover other cases. The model should also make explicit and support automated reasoning about the sensed parameters. It should also ease reconfiguration through reduction of errors discovered at integration time, the amount of effort and level of skill required, and the duration of the development.

There are conflicts and tradeoffs between degree and ease of reconfiguration, as in other software discussed earlier. However, these issues are exacerbated in external IO. First, most external inputs arrive as untyped data, i.e., the data has not been organized and typed in the same paradigm with compatible tools. Secondly, current practice leaves the semantic interpretation of external inputs implicit; it is typically hard-coded in the application program using the inputs: Therefore, it is not easy to use the same input information for other purposes in later configurations. User-defined abstract data types alleviate the issue, but tend to increase the initial learning and application effort. The use of abstract data types and encapsulaton might increase the execution time or variation in timing also.

We encountered all these difficulties in our experimental investigation. We review our model and discuss observations on the development process in Sections 6.1–6.3, expand the application of the same concepts to user interfaces in Sections 6.4—6.7, summarize the evaluation in Section 6.8, and present the status and further research needs in Section 6.9.

6.1 Servo sensors and actuators

Providing flexibility and reconfigurability in machine tool control software is most challenging in external IO for a servo control loop, because of the short time period (Section 4.2.1), constraints on variation in the period (Figure 4.1), time-distance constraints coupling the input, control loop processing, and the output (Rule 10), the relatively large amount of processor time consumed cumulatively due to the high frequency of sampling and update, and the associated overhead of task switching (Rule 16).

6.1.1 Overview of IO interfacing software

There are three common types of Axis IO interfaces: a continuous analog signal (e.g., output of a tachogenerator sensing angular velocity), a pulse train (e.g., output of an incremental position encoder), and a digital bit-pattern, byte, or word (e.g., input to a pulse-width modulating amplifier). A trivial case of the bit pattern is a single bit signal, e.g., output of a limit switch or input to a bistable solenoid. In each case, some signal interfacing device converts an incoming signal into a group of bits that forms the minimum transferable unit, e.g., byte or word or double word. Similar signal interfacing devices convert the output from the computer control system to a signal of the electrical characteristics needed by an external output device.

The interconnection between the processor and the external IO device may also vary with the hardware implementation, e.g., the ISA bus, the VME bus, the PCI bus. The interfacing software must allow easy adaptation to any of the common existing buses, and extension to accommodate any future communication path to a passive device. Here we confine the scope of our model to passive devices of the type that map IO into memory. Thus, the core function of the interfacing software is to translate variable names and addresses between the application process and the interfacing device via the protocol of the interconnection medium.

To preserve the correctness properties obtained by encapsulation, the application software must not use memory addresses directly to access the external IO. However, the indirection must not introduce variability in execution timing, and should not compromise execution efficiency. There is a conflict between the objectives of data protection and ease of reconfiguration on the one hand and, execution efficiency, on the other hand.

6.1.2 Device models

The interfaces to external devices hide vendor-specific and device-specific differences; therefore, they may be called device drivers. The interfaces are modeled in three parts: a master device (Class Structures 6.1), a slave device (Class Structures 6.2) and an interconnecting bus. A master device can acquire the bus (become bus master) to access another device, whereas a passive device can only be interconnected through the bus, but cannot actively access its services.

The MasterDevice class hides implementations of device accesses specific to a processor, operating system, and interconnection medium. A subclass, CPUboard, specializes it to processors using bus interconnections with other devices. The class hierarchy to this level is abstract. The CPUboard class is further specialized to a vendor-product specific concrete subclass, e.g., XVME674, which also inherits from the subclass for the type of interconnecting bus, e.g., the VMEbus class. The function GetBus(...) is used to acquire the bus at a priority level specified by nLevel (the data type byte is the same as an unsigned char). The enumerated variable mMode may alternately be assigned the value for privileged bus access. The function ReleaseBus() releases the bus for use by another master device, if any. Since the access and release of a bus are time-consuming operations, a single master system may be initialized to acquire the bus — a release during operation is not necessary. The functions Peek(...) read and the functions Poke(...) write a byte, word, or dword from (or to) a specified memory location. Their implementation is defined in the appropriate subclass, e.g., XVME674, specific to the platform access restrictions and permission protocol. The function Inport(...) reads a byte of data from the specified hardware input port, and

the function Outport(...) writes a byte of data to the specified hardware output port. The function SetTimeConstant(word nValue) sets the duration after which the access operation is treated as a failure. The function SetBerrTC(word nTC) is used to flag the error. The data types word and dword are the same as unsigned short and long, respectively.

```
Constructor-destructor functions:
MasterDevice();
MasterDevice();

Public member functions:
virtual void GetBus(byte nLevel = 0, BUS_ACCESS_MODE mMode = STAN-
DARD_BUS_ACCESS) = 0;
void SetBerrTC(word nTC);
virtual void ReleaseBus() = 0;
virtual byte Peek(long nAddr) = 0;
virtual void Poke(long nAddr, byte nValue) = 0;
virtual word WPeek(long nAddr) = 0;
virtual void WPoke(long nAddr, word nValue) = 0;
virtual dword DWPeek(long nAddr) = 0;
virtual void DWPoke(long nAddr, dword nValue) = 0;
virtual byte Inport(long nAddr) = 0;
virtual void Outport(long nAddr, byte nValue) = 0;

Private member functions:
word GetTimeConstant();
void SetTimeConstant(word nValue);
```

Class-structure 6.1: Interface of class MasterDevice.

SlaveDevice, an abstract class, provides accessor functions for a pointer to the MasterDevice object that will access the slave device and the base address assigned to the slave device, corresponding to the hardware setting. It is specialized into three abstract subclasses: AnalogIO, CntTmrIO, and DigitalIO. As in the case of master devices, each is specialized to a vendor-product specific subclass which also inherits from the subclass for its interconnecting bus.

```
class SlaveDevice
{ public:
SlaveDevice();
SlaveDevice();
long GetBaseAddress();
void SetBaseAddress(long nValue);
void SetCPUPointer(CPUboard *ptr);
CPUboard * GetCPUPointer();
};
```

Class-structure 6.2: Interface of class SlaveDevice.

The AnalogIO (Class Structure B.13) has additional functions to configure and initialize it, accessor functions for the software gain of each port on the board, and accessor functions for the digital value in a port. There are accessor functions for the number of ports, word length, hardware gain, range, maximum value, and conversion time constant, but the hardware-dependent settings have protected access only.

6.1.3 Device model development experiment

In the first stage of the device modeling experiment, we developed a set of abstract classes and a starter set of concrete classes for a VMEbus family of master and slave devices: MasterDevice, CPUboard, and their concrete class, XVME674, the class, Bus and its concrete class VMEbus, the class, AnalogIO and its concrete classes, XVME500 for input, and XVME530 for outputs, the class, CntTmrIO and its concrete class , XVME203, and the class, DigitalIO and its concrete class, XVME201.

Purpose of developing device interfacing software

In addition to the original purpose of providing a vendor-neutral, platform-independent interface to access slave devices, a performance-improvement reason emerged after trying out the vendor-supplied library for our testbed platform running under QNX. In the vendor-implemented client-server model, the time consumed in each access was in the order of 1 millisecond, whereas our objective was to support data acquisition at 100 microsecond intervals. *Would the object-oriented model allow the execution efficiency required?* We needed an investigation to answer this question. The first implementation, using virtual functions with several levels of indirection, resulted in an average read-access time of 100 microseconds. Although an improvement over the vendor-supplied implementation, it was still too high for the project needs. Peek and Poke functions were revised to use lower level bus access functions — the result is an average read-access time less than 35 microseconds.

Evaluation of device model

Although this access time is still larger than direct access under DOS, it satisfied the more common data acquisition needs. We conclude that it is a reasonable compromise between execution efficiency and flexibility, in view of the trend of continually lowering processing time costs and continually increasing software change costs. One aspect of the compromise is the dependence of the XVME674 class on certain hardware addresses specific to a version of the processor module — the hardware may change in the future. The cost of the change is limited to a change in the XVME674 class. Slave devices are protected against such changes. The XVME class also provides an alternative to use the vendor-supplied library. If the lower performance is acceptable, no change to the XVME674 class is required — only the vendor-supplied library has to be updated.

The total effort in developing the classes mentioned above, including integrated testing and performance improvement, was approximately 400 person hours spread over a 3-month duration, interspersed with other non-related work. The developer, PA, had considerable experience in device driver software, the VME bus system, programming in C and C++, and UNIX, but not in the QNX operating system.

6.1.4 Device model extension experiment

In the second stage of the prototyping experiment, another concrete specialization, IP320 class of the AnalogIO class, was implemented. Its primary developer, PB, had no prior experience or familiarity with device driver software, the VME bus system, the operating system, and the application. However the person had good programming skills in C and introductory working knowledge of C++. Due to difficulties encountered, a second person, PC, with prior education and experience similar to PB, was assigned to assist in the

development of the IP320 class.

Purpose of model extension experiment

The purpose of the experiment was to evaluate the ease of extension of the initial model, recognizing that the first concrete class XVME500 had influenced the generalization in the AnalogIO class. The original developer, PA, was not available during the development of the IP320 class.

There were several differences in the hardware. The IP320 hardware does not support assignment of a different gain value to each channel (labeled port in our model) ahead of time; however, a gain value can be assigned for the port currently selected for reading. The issue is circumvented by pre-storing a gain value for each channel in the IP320 object. When a port is selected through the function GetPort(...), this function calls an IP320-specific private function SetPresentGain(...) to access the pre-stored value for that port; it sets the gain through the public function SetGain(...) included in the AnalogIO specification. Similarly, the utility of the GetGain(...) function is limited to the currently selected port. A second difference arose in byte swapping (an issue peculiar to the use of Intel processors in a VME bus system) — the XVME500 required byte swapping in the bus master enabled, whereas the IP320 required it to be disabled. Since this option was selected in the BIOS settings, the difference did not become explicit in the class design.

In the process of developing the IP320 class, some other weaknesses of the AnalogIO class structure were also discovered and noted for future improvement. The analog input channels can be set up as single-ended inputs or differential inputs. This difference had not been captured explicitly in a data member of the AnalogIO class. It had to be hard-coded. The voltage range of the input was also not explicitly modeled in the AnalogIO class. Although these data do not affect the coding to read channels on the card, their omission does leave the information implicit and unavailable for future automated reasoning about properties of the incoming signal. The code was found to be inconsistent with the class design model.

Development cost

Participants,PB and PC, developing the IP320 class were given time to familiarize with the original device model, separate it from other software models, and make the model consistent with the code. This time was approximately 50 hours for the two participants (approximately equally distributed). The total time specific to the IP320 class was approximately 130 hours (approximately equally distributed) of which over 80 hours were spent in integrated testing — mainly in the discovery of the byte swapping option setting in the BIOS. The class design time was 2 hours and the initial coding and unit testing time was approximately 14 hours.

Evaluation of the model extension

The model and implementation for the master device and the bus were totally reusable (the Peek and Poke functions were reused with no modifications). Although PA had spent significant effort in the platform-specific details hidden within these functions, PB and PC did not have to spend time learning about these details. The reuse of this software will be very helpful to future developers too. The application software previously using the XVME500 class required little change when the XVME500 was replaced with the IP320

object (declaration change). The time used in the IP320 class design, coding, and unit testing was reasonable, considering the developer's inexperience. The extraordinarily large time spent in integrated testing and debugging was attributable primarily to the obscurity of the byte swapping option set in the BIOS, and secondarily to implicit differences in the hardware of the two analog input devices. The class design model and documentation were not sufficient in avoiding this cost. Implicit hardware dependencies must be documented explicitly — the only means we have is textual documentation.

The device model described above provides interfaces for passive IO devices, typically used in a servo-motion control loop or a high speed data acquisition process. The passive IO interfaces may also be used for other applications.

6.2 Model of discrete control (on-off) devices

There is a category of discrete (on-off) devices, e.g., limit switches of class Switch, where the state changes much more slowly than in servo-IO devices; thus, the sampling or update interval is large. Often, especially in automotive machine tools, the number of such devices is large. Therefore, a satellite processor (or multiple processors) may be used to access the IO. If this processor is on the same IO bus as the main processor, it may be treated as a master device in the model shown in the next section. If the IO-interfacing processor is connected to the main processor through a network, the case is discussed in Section 6.3, where the IO interfacing model is integrated into a general networked interface model.

6.3 Generalization of external IO

When an external IO device is interfaced to the main part of the application through a network service, the main application does not communicate with the real IO device directly. A protocol is needed to identify the device explicitly and associate the data with it. This protocol must be economical in its use of run-time resources. Proprietary systems that supply the IO bundled with the main application often utilize implicit assumptions to economize on the amount of information transferred. This approach is not practical in RMSs, where IO and application components must be integrated from multiple sources at different times in the life of the RMS. At the other extreme is the totally self-contained self-explanatory message packet, as in general-purpose networked communications. This becomes unnecessarily voluminous and consumptive of resources. Given that in a real-time RMS, all communicating objects and their class specifications are known at configuration time, we use this knowledge to devise a compact yet explict and complete scheme, based upon the idea of externalization-internalization of objects when communicating across processors. Inputs from sensors or outputs to actuators fit in this scheme.

6.3.1 The messaging scheme

We describe the messaging aspect of the domain architecture with an example of a simple external input update of some sensor state. (The messaging of outputs is symmetrically opposite). Then, we show how the scheme is applicable to generalized external inputs and outputs.

Referring to Figure 6.1, an application process, R, is configured to receive inputs from remote sensors and switches at an incoming port, P, (Class Structure 4.2) with buffered

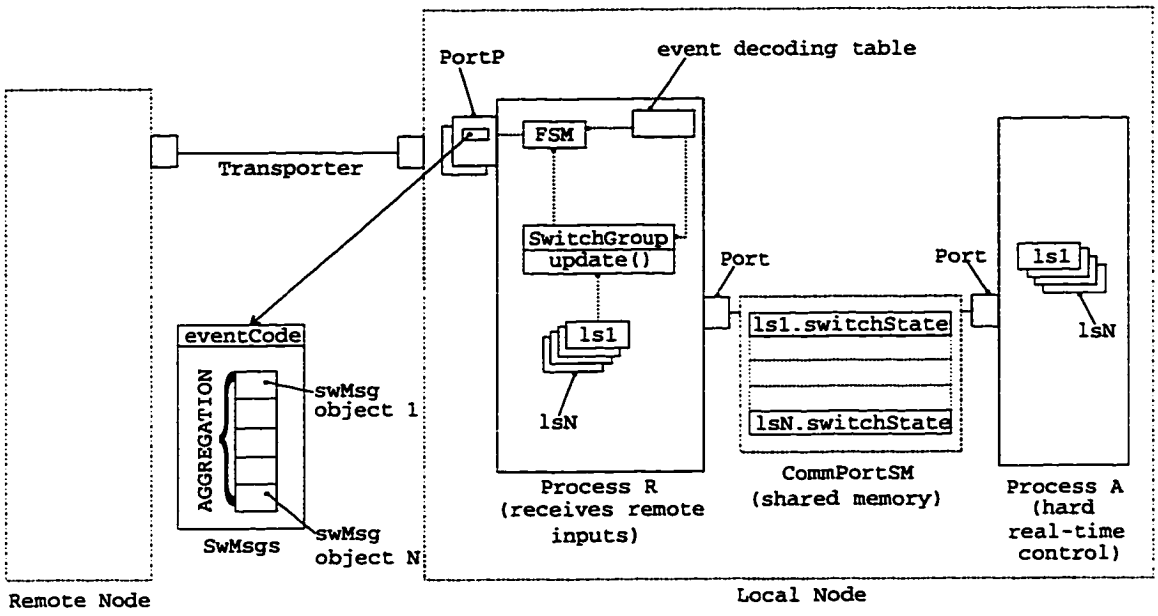

Figure 6.1: Message structuring and handling for efficiency

option, which causes a message queue mechanism to be set up through the CommPortMQ object (Class Structure 4.3). The remote process sending the sensed signals connects into it. If the remote node is not compliant with the system architecture, its proxy agent performs these roles on its behalf. Process R has an object corresponding to each sensed input it is designed to receive. Consider the example of an overtravel limit switch on an axis of motion, for which the object, ls1, of the Switch class is created. As a minimum, ls1 has the accessor functions for the last value of the sensed variable, say setSwitchState(Boolean) and getSwitchState() which returns a Boolean value. The object, ls1, is the server of the value of switchState to other objects in Process R requiring the information.

The message structure

Port P is initialized with the MsgCode objects (Class Structure 4.4) corresponding to each external input intended for it. Each message transferred to Process R has a corresponding structure. Let the MsgCode object, corresponding to ls1– >setSwitchState(Boolean) be named swMsg1. Recall that MsgCode has two data members — an eventCode and a pointer to an object derived from the class ParameterListStructure. Process R has a data object corresponding to each msgCode defined in Process A.

Encoding event identifiers: Every incoming message must map into some function of some object in Process R. Therefore, every <object identifier><function identifier> pair in Process R is encoded as an eventCode. For example, <ls1><setSwitchState> may be encoded as 4501.

Structures to receive data: The architecture specifies a specialization of the ParameterListStructure class for the data structure corresponding to the sequence of the parameters of each function of each class. In the case of the Switch class, for the function setSwitchState(...), which has only one parameter of type Boolean, the structure (say

ls1State) is of class BooleanStructure. Thus, swMsg1 has a pointer to a data structure of type BooleanStructure. The message receiving function in Process R (eventPort->transfer) writes the value of switchState at the pointed location for this BooleanStructure object.

Processing a message

The FSM of Process R examines the eventCode (4501 in our example) and maps it into ls1− >setSwitchState(ls1State). In the general case of functions with multiple parameters, each member of the ParameterListStructure is mapped into the corresponding parameter, in order. The function ls1->setSwitchState(ls1State) is executed in accordance with the state transitions specified in the FSM of Process R. Although we chose a trivial example to illustrate this procedure, it is is a general event processing scheme.

6.3.2 Improving message handling performance

If there is a large volume of simple data updates and the ultimate receiver of the data is a time-critical process that requires economy in execution time, steps in the procedure may be distributed across intermediary processes, employing two types of performance-improvement techniques — batching or aggregating updates and mapping updates into shared memory. These techniques fit in the pre-existing software framework, illustrated in Figure 6.1, as described next.

Batching data transfer and update

Batching updates implies the processing of a "batch" of messages, say swMsgs, in one step of the FSM of the destination process, say A. We offload pre-processing of the batch to a subordinate FSM in a helper process R. Consider a case with a number of objects, N, of class Switch that are logically, geographically, and temporally correlated (same sampling and update interval, which is much larger than the update interval of A). Furthermore, their readings are taken, and collected by a common processing node, which transmits the readings in a batch, at every sampling interval, to the same receiving node. We illustrate how this pattern fits into the general messaging scheme with the following example. At the destination node, a receiving process R (of priority lower than that of process A) is set up with a port having a MsgCode object, say processMsgList, an object, batchProcessor, of class, BatchProcessor, specialized from the class FSM, and a swMsgs object of a specialization of the ParameterListStructure class.

Referring to Class Structure 4.4, let
processMsgList·eventCode = 0701, and
processMsgList− >data = swMsgs,
where the event code 0701 corresponds to the function, process(), of the BatchProcessor class. This function processes the event 0701 by processing each message in swMsgs as in the example of swMsg1 given earlier. As a result, the switchState member of each Switch object is updated. Since the execution time for receiving and processing the processMsgList object occurs in a lower-priority process R, process A is not directly affected with the extra overhead incurred in communicating with a remote node or traversing through a container object. This example was a simple case of batching updates of Boolean objects. The same scheme is readily usable for updating other types of objects, e.g., integer or double. The framework described above also supports the batching of dissimilar messages. Since different applications may require different types of batching, the number and variety of aggregations

could be large, and the reuse of each aggregated structure, relatively small. Therefore, such aggregations for external communication, e.g., swMsgs object are not a part of the domain model.

Shared memory for updating sensed state

To transfer the updates from process R to process A efficiently, a pair of port objects is set up between R and A with the *shared* buffering option. This results in the set up of a CommPortSM object (shared memory) between them. The Boolean data member, switchState, of each Switch object in process R is mapped into shared memory. A corresponding switchState object in process A is mapped into the same shared memory. A shared Boolean variable is also used as an application-level semaphore — process R sets it indicating "refreshed" and process A resets it indicating "stale." If process R finds it reset, it will update the shared object.

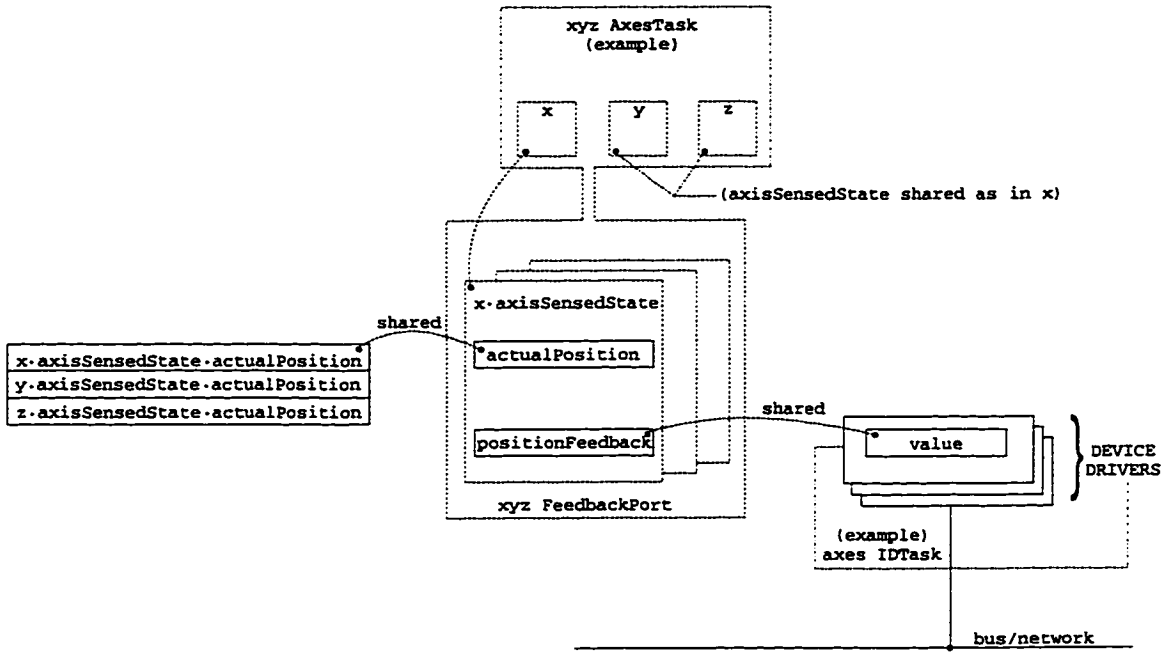

Figure 6.2: Efficient update of shared, yet encapsulated, objects in hard real-time control

6.4 Inputs from and display to users

Our generalization of external inputs and outputs applies to human interaction as follows. First we consider simple manual signaling devices such as pushbuttons. We model them as specializations of the Switch class, to allow for differences in processing the signals from different types of pushbuttons, e.g., "momentary-type", "maintained-position type", etc. In all cases, the signal from the switch is transformed into an event that maps into the function setSwitchState(...), as for limit switches discussed above. There are different types of manual signaling devices with similar semantic intent — our model focuses on the common semantics, e.g., the Switch class hierarchy. Still, a controller must be interfaced

with the different signaling devices and must localize and isolate their differences. So we apply the same concept of device drivers as for other IO devices.

Let us consider a case where the equivalent of the pushbutton signal comes from a graphical user interface (GUI). Treating the GUI as a device, the device-specific software is isolated in a class named GUIinput (Figure 2.4). It encapsulates the functions for receiving the signals from the GUI and for transforming them into local msgCode objects (Class Structure 4.4). The HMIinput object transforms the local msgCode objects into global msgCode objects. The task in which it runs has the port through which the msgCode objects are sent out. Thus, the HMIinput object has no knowledge of the GUI device-specific signaling, and the GUIinput object is not aware of the rest of the system (Rule 1). Historically, it is common to customize a user interface to the application. This customization is localized between the GUIinput and the HMIinput objects.

Since the application is also reconfigurable in RMSs, the set of services available in a particular application, i.e., machine, will vary, even for the same machine over its life. These changes cause a change in the global msgCode objects. In order to shelter the HMIinput object from these variations, the encoding of the global msgCodes is maintained in a "lookup" resource, equivalent to a table. A configuration-specific "translation table" is created for translating a local msgCode into a global msgCode. The difference between the local and global msgCode objects is the eventCode — the ParameterListStructure object is the same between the local and the global msgCode objects. The local eventCode objects are specific to the particular choices made between the GUI and the HMIinput. Thus, the GUI component design decision is decoupled from the application system design for a particular RMS (machine).

Simple displays, e.g., values of variables, are handled in a similar manner through the HMIoutput and GUIdisplay objects. However, not all user inputs transform into simple one-to-one mappings onto objects and functions defined in the domain model. One such case is discussed next.

6.5 Manual data input for numerical controllers

One form of user input provided in CNC machines is called "manual data input" (MDI), a command line (ASCII string) conforming to the EIA RS274D format. When the user makes the preparatory selections for MDI, the HMIinput is correspondingly prepared to send the ASCII string to the RS274Translator (Figure 2.4), which in turn uses the services of a "SingleLineTranslator" to transform the command line into a sequence of objects understandable in the domain model similar to the local msgCode objects mentioned above. We assign a special name to this type of a MsgCode object, "ControlPlanStep." At this stage, the eventCode is "local", i.e., machine-independent. The sequence may be of length one, in the simpler cases. The SingleLineTranslator hides details specific to the RS274 grammar (Rule 1). Then the local eventCode in a ControlPlanStep is translated into a global code using the translation table mentioned above, resulting in a machine-dependent ControlPlanStep. The task in which the RS274Translator runs has the port through which the sequence of ControlPlanStep objects is sent out.

User inputs may be batched in a "script", which may then be viewed as a stored sequence of user instructions, known as a "part program." Next we describe the transformation of the part programming format specified in the EIA RS274D specification. Assume that the user has made a sequence of selections that places the controller in the state appropriate

for selection and translation of a RS274D part program.

6.6 Part program translation

Referring to Figures 2.2 and 2.4, when the user specifies a filename, the HMIinput recognizes that the msgCode from the GUIinput is a request for a local service. The file system is accessed and the contents (the part program) are transformed into a user ProcessProgram object, which is a sequence (doubly linked list) of STRING80 (80-character string) objects. Thus, subsequent processing steps do not use any code specific to the file system (reuse of the principle of device independence and Rule 1). The ProcessProgram and STRING80 classes are derived from a third party class library — navigation and manipulation functions are inherited and reused.

If the next user input is a request for the translation of the whole part program, the HMIinput invokes the services of the RS274Translator, supplying it the reference to the ProcessProgram object and the local to global translation table mentioned above. Using its SingleLineTranslator, it produces a sequence (doubly linked list) of ControlPlanStep objects, as in the case of MDI.

The RS274Translator builds a machine-independent ControlPlan, consisting of the ControlPlanStep objects and other data members that are "constants" for the whole ProcessProgram. The ControlPlan serves as a FSM subordinate to the FSM invoking its services Thus, the architectural pattern matches batched updates of external inputs described above. The ControlPlan provides functions for controlling the flow through the ControlPlanStep objects, e.g., single-stepped sequence, iteration, and branching.

A ControlPlan may include subordinate ControlPlan objects, generalizing the "canned cycle" feature in EIA RS274.

The combination of nesting and iteration compacts the specification of processing identical workpieces (parts) sequentially at different locations in the workspace, or the specification of multiple machining passes on the same workpiece at different depths of cut, by interjecting a change in the baseFrame value. Branching allows on-machine adaptation of the ControlPlan. A branching condition may be based on the current state of the manufacturing process, which, in its simplest form, could be the value of some sensed variable, and, more practically, some value derived from sensed variables and other data in the RMS controller. First we consider some quasi-static applications of this feature. The decision about the number of passes to mill a surface could be based upon the amount of excess stock determined by sensing the surface before processing it. Similarly, the final pass specification (depth of cut) could be determined by sensing the surface before the finishing pass.

There is no uniform means of specifying these manufacturing processes in currently available formats for user inputs. However, since the specifications are static or quasi-static, users devise indirect techniques either through additional machine control logic given in the language of the programmable controller and executed by it, or through macros for an EIA RS274 program that are pre-processed in another computer, or a combination of these techniques. The ControlPlan allows users to specify the manufacturing process in a uniform manner in one "program." It also allows the specification of branching conditions to be applied dynamically during machining, e.g., stoppage of motion upon sensing a broken tool. In general, the ControlPlan is a construct that allows the specification of the order and conditions for execution of any of the functions supported in a given RMS, i.e., all the functions of all the objects in a particular controller. Its FSM allows the specification of

state-based constraints on allowable functions.

The nesting feature of the FSM also allows the specification of upper levels of the FSM at earlier stages of application-development, progressively reducing the number of choices available (and thus the likelihood of mistakes) at lower ControlPlan objects, specified at the operational stages. At the operational stages, this feature may be used for "family of parts progamming" — a techniques in which the structure of a part program for a family of parts, is specified in an earlier stage ControlPlan and part-specific values are provided in a later stage ControlPlan. A higher level FSM may also serve as the machine control logic for the RMS, as discussed next.

6.7 Specifying control logic

It is common practice in CNC machining centers, turning centers, workstations, and cells to execute the control logic for basic discrete on-off functions in a programmable controller, commonly an entity distinct and separate from the numerical controller. Some reasons for the historic separation were mentioned above. The discrete logic is "fixed" across all NC part programs that will be executed on the machine. The traditional part program does not require very close dynamic interaction between motion control functions and discrete logic functions on the machine. However, the traditional separation has required two different forms of programming paradigms, translation software, and execution hardware on the same machine. The structure of the FSM does not distinguish between discrete logic functions and continuous motion functions. Specifications in the IEC 1131-3 standard for programmable controllers can be translated into the FSM, although we have not developed an IEE 1131-3 Translator.

Additionally, motion control and discrete logic can be integrated more closely. For example, consider the case of the "tool is broken" signal S. It is integrated by adding the necessary transition in the EXECUTING state for the event S.The effect of S is to execute the specified action sequence and change state to BROKEN_TOOL. At the early stages of evolution, the action sequence may be as simple as a single function of AxisGroup feedHold(). A later upgrade may replace it with a function that executes more aggressive stoppage of motion.

6.8 Evaluation

Experiments in the development of software for interfacing external IO addressed several questions. Does the OO paradigm and its use with fine granularity assist in the reconfigurability and extensibility of IO-interfacing software? Can this approach support hard real-time specifications of the servo-sensor loop?

6.8.1 Execution overhead of OO IO-interfaces

The experiments for the development of software to access signal-interfacing hardware showed that the approach yielded reusable IO-access software, which allowed less experienced personnel to develop extensions. The experiments also showed that there was an execution time penalty. Further experiments were conducted with various task configurations on the project testbed (an Intel486-50 MHz processor running under the QNX 4.23 operating system in a VME bus) to understand the impact of the larger execution time.

Software for external IO access was not the major contributor to the time-loading factor. In one experimental setup, motion control software for three axes of control, including their IO, was run in one process at a period of 10 milliseconds, with the system timing resolution set at 50 microseconds. The timer interrupt handling accounted for a time-loading factor of 0.35 approximately. The process execution time was between 150 and 300 microseconds per axis depending upon the control algorithm used, accounting for a time-loading factor under 0.1. Thus the total known time-loading factor was approximately 0.45. The load from various OS processes could not be measured. In contrast, the time-load contributed by the IO accesses (6 inputs, 3 outputs) was estimated to be under 300 microseconds per 10 millisecond period, i.e., a time-loading factor of 0.03 — less than 8 percent of the total known load on the processor. Under these conditions, the variation in the periodicity of the task was limited to 1 unit of resolution almost half of the time, to 2 units, most of the time, and to 3 units in the worst case. Thus, this experiment showed that a basic three-axis motion control could be run successfully, utilizing well encapsulated, fine granularity software for accessing external IO. The tradeoff between reconfigurability and extensibility on the one hand and execution time penalty on the other hand was reasonable.

6.8.2 Feasibility of OO multitasking

In an extension of this experiment, a data acquistion process was added, running at a period of 1 millisecond, with execution time estimated to be at least 50 microseconds, i.e., a time-loading factor of 0.05 at least. A supervisory force-constraining control process running at a 40 millisecond process was also added — its time-loading factor was significantly lower than the other processes on the computer. The worst case variation in the periodicity of the motion control process remained the same; however, the distribution became a little worse. We estimate that the time-loading factor was approaching a critical limit. This experiment demonstrated the feasibility of running multiple application tasks, composed with fine-grained object-oriented software, cycling with different time periods, and managed by a general-purpose real-time operating system.

6.8.3 Generalization of external IO

The concept of external IO is generalized to treat any external event as an external input. A messaging scheme adapts more general techniques to provide timing economy and predictability. This messaging scheme and the concept of device independence in software are then applied to human machine interfaces (HMI). It shows scalability and flexibility in the HMI application and reuse of the same architectural constructs.

6.8.4 Generalization of user inputs

The notion of HMI is extended to include scripts, part programs, and machine control logic specifications provided by different types of users at different times in the lifecycle of a RMS. A unified construct, the ControlPlan, is introduced for the various forms of user programs. An RS274Translator has been developed and demonstrated. The resulting ControlPlan has been used as input to the motion control subsystem.

6.9 Status of architecture for interfacing external IO

We have developed specifications and demonstrated prototype implementations of interfaces to the basic types of external input and output signals. Applications designed to these specifications are protected against changes in the underlying operating system, processing hardware, communication bus, and the signal-interfacing hardware. Software for the processing and communication hardware and software for a signal-interfacing device are independent of each other, i.e., the effect of a change is localized. This modularization increased execution time cost; however, it was still possible to run a number of cooperating processes in different task configurations, with task periods ranging from 1 millisecond to 10 milliseconds. We have demonstrated extension of the passive device model — inexperienced developers were able to reuse prior software and add a specialization, without interaction with an expert in device driver development.

Efficiency and flexibility in messaging: A messaging specification has been developed for interfacing remote IO through a "foreign" IO-process. This specification is generic — it may be used to invoke any function defined in the domain model and the application. The novelty in the messaging scheme is the balance between flexibility and execution efficiency appropriate for the domain of machine tool control.

Improvements in the abstraction of signal-interfacing hardware are needed. First, dependencies on hardware and BIOS configurations need to be made explicit. Secondly, emerging patterns in on-board functions should be investigated and corresponding abstractions of data and function members should be developed, so that cost of future specializations may be reduced.

CHAPTER 7

Overall work coordination and distribution

Should a machine tool controller have a central overall source of coordination? The answer is not clear from general software organization theory or practice. At one extreme is the "decentralization school", which claims benefits of maximum flexibility, reconfigurability, and extensibililty. A new program may be added to the system without modifying any other program. The new program in execution (we will call it an active object) follows some protocols specified in the architecture that allows it to find other active objects and exchange messages with any of them. If the new active object cannot find or obtain the resources it needs, it will not run. Each active object is designed to maintain its own correctness, integrity, and communication with its activator (ultimately, the user). This architectural concept is very attractive for RMSs.

However, can correctness of the overall operation be assured, even if each active object assures its own correctness? Here, we define correctness as meeting overall system requirements (which, by its very nature, is a centralized unit of information). How should the overall system requirements be specified so that conformance can be evaluated? How should these requirements be transformed into an overall system design so that conformance to these requirements may be assured or evaluated? Similarly, how can a system design be transformed into component design, e.g., design of the active objects, to assure conformance? These are long-standing long-term research questions.

Since the addition of a new active object amounts to a change in the system, it must be preceded by some change in the overall system requirements (a centralized unit of information). We argue that the overall system design process is simplified by transforming the centralized requirements into a centralized system design. The latter may be transformed into decentralized components that conform to the centralized system specifications, including protocols, and operate under central coordination, i.e., a single root of control.

We have formalized this concept in the form of a TaskCoordinator class — Section 7.1 introduces its role and responsibilities. It balances centralization and decentralization. Sections 7.2–7.5 describe its role in the control, coordination, and integration of a multi-machine workstation, a simple machine, a user interface local to a machine, remote control, process control, and discrete control. These coordination roles show how the task coordination concept supports extensibility of the architecture. Then, we evaluate this coordination concept by comparing it with the total decentralization alternative (Section 7.6). We will show through counter-examples how the total decentralization concept increases design complexity. In conclusion, we summarize the discoveries made in the software development process (Section 7.7), and issues yet to be resolved (Section 7.7).

7.1 Role and responsibilities of the task coordinator

The TaskCoordinator, a specialization of the Task class, coordinates all other tasks in a single machine tool controller, e.g., motion, sensing, and various event-driven control processes. Its main role is to manage overall control flow, through its FSM (Class Structure 4.5). In order to help in the integration of a control system, the TaskCooordinator provides a unified systematic means of specifying and interleaving a *discrete control action* and a *continuous motion control action* through its FSM. Procedure 10 summarize the role of a top-level task coordinator (TC_{top}).

- Startup (possibly delegating to an agent).
  Directly or indirectly:
  - Create the next level of objects needed in the integrated control system, including other tasks.
  - Cause inter-task connections to be established.
  - Set up all software resources needed in the system.
  - Initialize objects, e.g., from persistent objects.
  - Start all other tasks.
  - Maintain overall system state and control flow:
    * Receive "external" events:
      · From human-machine interface (HMI) directly, or
      · From user script or control plan initiated from a HMI.
    * Process the events and transitions.
    * Track system state.
  - Control scheduling of tasks indirectly through their release.
- Shutdown (possibly delegating to an agent)
- Handle exceptions as a handler of last last resort.

Heuristics 10: Responsibilities of a top-level task coordinator.

Relationship to IEC 1131: A task coordinator corresponds to the root of a *configuration* in the IEC1131-3 software model [39, Section 1.4.1, Figure 1]. In an application consisting of very loosely coupled subsystems, corresponding to a *resource* in IEC1131-3, one or more such subsystems may have their own task coordinator, TC_sub. Some TC_sub may have no motion control under it; it may have only discrete inputs, discrete outputs, and Boolean conditions under its sphere of control. Then, it corresponds to the root of a *signal processing unit* that specializes in *logic control* [38, Section 4.2].

Scalability and evolvability of a task coordinator: When the control system is simple, e.g., it consists of only one task, it is constructed as a TC_{top}. All the software resources for the system are created, initialized, and placed into execution within TC_{top}.

When another task is added to the system, adding new functions (not moving any functions from TC_{top} to it), then two types of modifications to TC_{top} are needed – additional resources, e.g., a port or addition of msgCode objects to an existing port, and corresponding transitions in its FSM.

A rudimentary task coordinator was added to the experimental prototype in Stage S6 (Table 7.1). It received external events from the user interface, as well as from a translated user part program. In Stage S7 of the prototype, TC_{top} includes the capability to startup other tasks and to cause inter-task connections to be established. With the improved Task class, its FSM provides the framework to initiate orderly stoppage (pause) or shutdown and to handle exceptions passed to it by other tasks. Other tasks, derived from the same Task class, also have the same FSM structure, with an EXCEPTION state to handle exceptions locally. If a particular task is unable to handle some exception, it notifies the exception to the task coordinator as a message with the event code for that exception. When the task coordinator receives this message, it performs the necessary action(s) and enters the EXCEPTION state.

Stage	Change in configuration	Effort (hours)
S1	One-axis motion; Wrapper around old pid control law; Objectified IO interfaces; 3 persons learning, prototyping, debugging	300
S2	Multiple one-axis motions; - Periodic task, incl Posix timer; - GUI on separate computer, linked with ftp. Learning time for new participant Understand, port code from prior program Code and debug motion program Code GUI program; connect to motion program	145 60 90 44
S3	Multi-axis coordinated linear motion; Fine-grained separate processes; Wrapper around old interpolator; IO drivers in separate processes; shared memory IPC. Total effort	325
S4	All motion-related functions in one MC process; Multi-axis coordinated motion objectified - wrappers around old interpolators; Total effort	150
S5	Rudimentary task coordinator with FSM added; Force data acquisition process integrated; Force-constraining outer control loop integrated; POSIX mq atop QNX messaging service inter-node; Shared memory IPC intra-node. Total effort	140
S6	Force data acq process on third computer; FSM classes improved.	16 110
S7	Changes resulting from changes in class lib (Not run on machine tool successfully).	220

Table 7.1: Stages of developing test applications from class libraries.

7.2 Coordinating a workstation

Referring to the example workstation (Figure 2.1) and its stages of evolution (Section 2.1.2), we show how the various functions in this workstation are coordinated. The physical functions of the workstation, in its last configuration, are decomposed into five parts, considering the degree of coupling, namely MC1, MC2, A-B, WC1, and WC2. A-B is treated as a separate unit, because it may be coordinated with either MC1 or MC2 or WC1 or WC2.

7.2.1 The organization of task coordinating software

Task coordinators are set up in a hierarchy as follows, so that each coordinated unit may function by itself in some earlier-stage configuration. The whole workstation has a top-level task coordinator, TC_w, which has "direct software connections" with task coordinators, TC_{mc1}, for MC1, TC_{mc2}, for MC2, TC_{ab}, for A-B, TC_{mc1}, for WC1, and TC_{mc1}, for WC2. At the next level, TC_{mc1} has "direct software connections" with task coordinators TC_{mt1} for MT1, TC_{tc1} for TC1, and TC_{tm1} for TM1. Task coordination for MC2 is similarly organized. The concept of static task configuration implies that, after proper initialization, a task coordinator has information about the task configuration "downstream" of it — when it receives a request for service (event), it does not have to perform an external search for the server (provider of the service).

7.2.2 The flow of control

Our concept of control hierarchy means that the task coordinator controls the flow of control for responses to received requests for service (events), through its FSM. For example, in a particular state, it may cause redirection of all further events from a source to some other task coordinated by it (delegation). Thus, a control hierarchy of chain-length L does not necessarily imply a data flow path of length L (number of transmissions in the flow of every message).

The delegate (task) can service only those events defined in its eventPort. If the delegate receives an unrecognized message, its FSM raises or throws an exception which is caught by its coordinating TC. The delegate's FSM must have an exit state, in which control is returned to its coordinating TC through a message.

7.3 Coordinating user interface

Let us consider how user inputs fit into the concept of organizing work in a machine tool MT1. An example task configuration is shown in Figure 2.4, where the controller has one task coordinator TC_{mt1} running in the process named "mainMotnCtrl." In Figure 2.4, HMIinput on Computer A performs the transformation of a user input into a message as described in Section 6.4. The message is received at the eventPort of TC_{mt1} causing the occurrence of an "event." The FSM of TC_{mt1} processes the event. In certain states, e.g., configuration of a coordinated task "SupvForcCtrl", the FSM of TC_{mt1} redirects the subsequent messages to the eventPort of SupvForcCtrl. When the user signals termination of this configuration mode, an event occurs at SupvForcCtrl, which sends a corresponding message to TC_{mt1} whose FSM exits that configuration state.

7.4 Specification of coordination logic

Suppose that the workstation integrator provides its overall coordination logic conforming to the IEC 1131-3 standard. Then, an IEC1131-3 Translator converts it to a state transition table (STtable object) for the workstation-level FSM, say, ST_w. If the control logic for each of the coordinated machining centers and mechanisms is also provided in the same form, then, the same translator converts that logic to respective STtable objects, say, ST_{mc1}, ST_{mc2}, ST_{ab}, ST_{wc1}, and ST_{wc2}. Similarly, the control logic for the machine tools and mechanisms in MC1 and MC2 is transformed into STtable objects ST_{mt1}, ST_{mt2}, ST_{tc1}, ST_{tc2}, ST_{tm1}, and ST_{tm2}. Starting from ST_w, the integrator may define a transition that directs the flow of control to the next immediate subordinate task coordinator.

7.4.1 Reconfiguration for an evolving workstation.

Since the STtable is an aggregation of StateRecord objects, each of which is an aggregation of Transition objects, a static reorganizaton of functions and responsibilities is effected by moving the respective Transition objects from one task coordinator to another. In the example of the workstation (RMS) shown in Figure 2.1, in an early stage of its evolution, there is only one task coordinator, TC_{mt1}, with the machine control logic defined in ST_{mt1} — it serves as TC_{top}. When the RMS is upgraded from a simple MT1 to MC1, TC_{mc1} is added, and the transitions associated with the role of TC_{top} are removed from ST_{mt1}, modified as necessary, and added to ST_{mc1}. Upgrade of the RMS to the level of the complete workstation involves similar changes.

7.4.2 Kinematic reconfigurations

An evolving RMS not only requires additional objects for sensors, actuators, controllers, and coordinators, but also knowledge or description of its kinematic configuration. However, there is no accepted definition of the functional, spatial, and kinematic boundaries of a machine tool or workstation or cell. The term machine tool has been used for a complete multi-station transfer line, as well as for a single-spindle, three-axis mechanism.

In the context of this research, we restrict the scope of a RMS to an automated machine tool defined as a kinematic arrangement of devices that cooperate to process one or more workpieces (parts) autonomously, using one or more tools to shape surfaces of the workpiece, and, optionally, measure them to assure conformance to specifications, through unified relative motion between a set of tools and a set of workpieces, as specified in a unified process plan or program.

By "process ...autonomously", we mean that, under normal circumstances, an automated machine tool performs the assigned processing of a part independently. Interaction or coupling with other units of automation in the factory is loose. Due to the immense variation in such units of automation — the degree of "intelligence" in them, the capabilities aggregated in any unit, and the kinematic configurations — there is limited commonality across organizations and applications. As a starting point, this information model provides a base class **Machine** (Class Structure 7.1), with a minimal "device description level" interface, from which various specialized classes may be derived, without losing the base commonality.

Modeling a machine tool: We show how the knowledge about a Machine is represented, building on concepts introduced earlier in Class Structure B.7. Referring to the definition of an automated machine tool, given above, the ControlPlan is the "unified process plan or program", and the "kinematic arrangement" is formalized in the class KinMechanism. By "a kinematic arrangement of devices that cooperate ..." and "unified relative motion ...", we mean that a single "tool to part relationship" is being transformed. If *toolA* works on *partA* and *toolB* works on *partB* and the two relationships ${}_{toolA}T^{partA}$ and ${}_{toolB}T^{partB}$ are being transformed independently of each other, the relationships are represented as two objects of the basic Machine class. For more complex cases, the Machine class is extended to add the members necessary to describe the kinematic relationships of the physical system.

The Constituents object is a container of objects in a machine (other than kinematic objects included in kinMechanism), whose services may be invoked in a (machine-specific) control plan. As a root of an aggregation tree, it allows compact identification of all the needed objects.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
Constituents constituents
KinMechanism kinMechanism
ToolPartTransforms toolPartTransforms

Class-structure 7.1: Interface of class Machine.

Modeling a kinematic mechanism: The class KinMechanism (Class Structure 7.2) models a kinematic mechanism as a network of other kinematic mechanisms identified in the container kinMechanisms and their interconnections are modeled as Connection objects (Class Structure 7.3), identified in the container connections. A connection identifies the two connected KinMechanism objects (nodeFrom, nodeTo) and the kinematic transform (placement) through the connection object. This network model is a generalization of the D-H model [16] which is limited to a serial chain. The purpose of providing this detail on line is to allow the specification of on-line kinematic reconfiguration. In the example workstation of Figure 2.1, the unit A-B may be part of four different kinematic mechanisms at different intervals of the workcycle: (X1, Y1, Z1, A, B) (X2, Y2, Z2, A, B), (WC1, A, B), and (WC2, A, B). In each configuration, the resultant kinStructure model (Class Structure B.6) may be derived from the connectivity model. The container kinMechanisms also includes the workholding and toolholding devices, the workpiece and the tool, so that the same model is used to account for their offsets, when any of these objects in a kinematic relationship is changed during the workcycle. These dynamic reconfigurations are specified through a control plan for execution by the FSM of a particular task coordinator.

Corrections and compensations through the kinematic model: The ToolPart-Transforms class allows for additional compensations to the kinematic model, e.g., when the true position of the tool or the workpiece is known.

7.4.3 Coordination of remote control interface

A machine tool may be controlled remotely by a human or higher level automation in the factory. The remotely accessible functionality is a subset of the functionality available at the local user interface. TC_{top} (representing the machine tool) must be in a state in

Constructor-destructor functions: Omitted for brevity **Accessor functions for following object members:** KinMechanisms kinMechanisms Connections connections Connection connection KinStructure kinStructure

Class-structure 7.2: Interface of class KinMechanism.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
KinMechanism nodeFrom
KinMechanism nodeTo
CoordinateFrame placement //transform
//from nodeFrom.kinStructure.placementFrame
//to nodeTo.kinStructure.baseFrame

Class-structure 7.3: Interface of class Connection.

which "remote input" is allowed. The machine tool can be placed in this state only from a local HMI, which is also authorized to remove the machine tool from that state.

Consider a case where the remote control unit is not conforming to this information model. Then, remote input is represented by a task (OS-level process), say RCI, which may be connected to TC_{top} only in the proper state. If TC_{top} is in the state to accept remote input, RCI checks the validity of the remote input, transforms it into a message conforming to the application architecture, and sends it to TC_{top}. If TC_{top} is not in the state to accept remote input, RCI sends the appropriate reply to the remote control unit. TC_{top} accepts only one source of user input for initiating action at a time. Stopping commands are accepted from a local source at any time.

7.5 Coordination of process control

We consider two cases of process control (Figure 2.4) where some computations are performed on continuous signals at periods different from the motion control task period. Therefore, these computations are set up in separate OS-processes.

7.5.1 Data acquisition:

The external signals (analog of one or more components of Force or Torque) are sampled in Process DataAcq on Computer C at very short intervals in the order of a millisecond. The raw data is given to two different data reduction computations. One computation is very short and requires the current and previous readings only; therefore, it is performed in Process DataAcq itself. The other computation requires a sequence of readings and its processing time is longer; therefore, it is performed in a separate process named SignalPre-processor. Process DataAcq places the raw values and the result of the quick computation in shared memory to minimize the communication load on it and the resulting variation in its timing.

7.5.2 Computation for force-constraint control:

The result of the quick computation is used by a force-constraint controller that has to be run periodically at a longer interval; therefore, it is set up in a separate process named

ForcCtrl. At every execution period, Process ForcCtrl has access to state information from the Process MotionCtrl, and supplies to it the overriding motion parameter, e.g., feedrate override, to maintain the machining forces within the process control limits. Processes ForcCtrl and MotionCtrl communicate through shared memory to minimize the communication load and resulting timing variation in the time-critical Process MotionCtrl. This computation may be viewed as a continuous process control loop operating as an outer loop for motion control. It modifies a motion parameter, but does not alter the flow of control.

7.5.3 Computations for broken tool detection:

Process SignalPreprocessor reduces sequences of raw data to some compact form, e.g., parameters of a time series model and forwards the results to the Process BrokenToolDetector. The latter process uses this data in combination with other state information obtained from the motion control process (through shared memory) and performs a computation to determine if the tool is broken. If so, it produces a resulting message, e.g., STOP. This message has a very high priority. The motion control process checks for this message at the beginning of every execution period. If the message is found, it executes the corresponding transition to stop motion quickly and switches to the appropriate state. The broken tool detection and motion stoppage may be viewed as a form of discrete event control — it alters the flow of control in the MotionCtrl process.

7.5.4 Work distribution

The decision about placing various computations in the same task or different tasks is left to the particular application. However, we can draw some guidelines applicable to the domain, using the example shown in Figure 2.4. These guidelines are applications of the workload characterization in Section 4.2.2, Rules 11, 12, 16, and 17, and Constraints 8

In the example of Figure 2.4, Rule 16 is applicable. Task periods are small relative to the system services overhead Toh_t. The time-loading factor is high. Allowable timing variation is low. In this context, we derive the following Rule 18.

Rule 18 *When the time-loading factor and system services overhead Toh_t are significantly high, distribute computational work across tasks in a manner that reduces the overhead Toh_t, while meeting other application requirements and constraints.*

All periodic closely-interacting computations that require the same time period should be executed in the same task. Where computation intervals differ only slightly, application re-engineering should be explored to commonize them.

Time periods of repeating work: Discretization of continuous signals or control processes results in specifications of sampling and update intervals, primarily dependent on the characteristics of the physical process or phenomena (Section 4.2.1, and secondarily on the algorithms. If the application requirements thus derived allow some range on the choice of the time interval, for computational economy, the control engineer may specify the longest intervals possible, without compromising robustness. The computer implementation cannot increase the time intervals. However, if several different computations have intervals that are only slightly different, it may be possible to select the shortest interval as the common interval for those computations, provided the algorithms or their parameters can be adjusted to match the change.

IPCs and task allocation: Applying Rule 16, time-critical tasks interact with other tasks through shared objects, and closely interacting tasks are located on the same processor. Message-handling across processors is delegated to helper processes — even though process-switching overhead is increased, it reduces the timing-variation of a time-critical task.

7.6 Evaluation

One aspect of this research investigated difficulties in developing, configuring, integrating and reconfiguring a multi-task application, composed from a given set of classes. The purpose of this investigation was to discover the effectiveness of the chosen software development approach, and to identify the gaps in reusable resources, e.g., class libraries. The investigation was conducted through a series of experimental prototyping stages (Table 7.1).

The testbed evolved from a system that had only one application task (as an OS-process) running while motion was operational (Table 7.1 Stage S1). One axis was moved under a pid control algorithm, for which the code was obtained from a previous program used by Group A, and "wrapped" with a C++ interface. The IO hardware and software was new.

In Stage S2, axes IO operations were executed in separate OS-processes. However, timing relationships varied beyond acceptable limits (Rules 10). Also, a GUI task was added on a separate computer (Figure 2.4, Computer A) because the capacity of one computer was not sufficient for the additional GUI workload. Each task had to be started manually separately.

In Stages S3 and S4, POSIX mq service was used for communication between the GUI and the motion control subsystem. However, mq was not accessed during motion.

In Stage S5, processes for force data acquisition and force-constrained control were added on Computer B with shared memory for IPC. Priority of the force data acquisition process had to be lowered (contrary to RMA guidelines) to allow the motion process to run properly. The larger variation in the sampling interval of force data acquisition was more tolerable than variation in servo loop intervals. A reconfigurable FSM was introduced in the main motion control process.

In Stage S6, force data acquisition was located on Computer C. POSIX mq service was used for all IPC. However, mq accesses were associated with unacceptable period timing variations in the motion control process and in the force data acquisition process. A rudimentary task coordinator was added, and the FSM was improved. However, other tasks were still started manually. The additional run-time overhead of adding a reconfigurable FSM was not significant in relation to the system services overhead and total time for IO accesses.

In Stage S7, there were significant revisions in the classes related to measures, space, kinematics, and axis control. Coding defects in measures and space related classes resulted in memory leaks, which prevented a successful run on the machine tool.

Setting up tasks and IPCs is costly: The development experience in Stages S3–S6 showed that setting up each task and the IPC involved much more effort than the coding of the functions that performed the main work. Learning to set up a task took several days. Learning to use each IPC (mq or shared memory or signaling) took several days. Setting up an instance of an IPC mechanism took half a day. The development effort was measured as follows. After several participating developers had already performed these steps, and reported acceptable "ease of use", a new participant was given the assignment of setting

up additional tasks and IPCs (mq and shared memory). The participant was a computer engineering senior undergraduate student, ranking high academically, who had previously taken courses in operating systems, data structures and algorithms and C++/C programming languages. The participant had been programming in the testbed environment of this project for several months. The participant was supplied all the documentation and reference software produced previously. After several rounds of setting up different periodic and aperiodic tasks and instances of IPCs, the participant was asked to note the development effort required in setting up a correctly working task and an IPC instance. Then, the participant was asked to analyze the main issues requiring effort, and suggest improvements in the development process, resources, etc. Two leading consumers of effort were identified: (1) setting up the IPCs, and (2) investigating timing problems. As a result, the Task class structure was revised, classes were added for IPCs, and period timing specification and service. The task coordinator was evolved to the next stage to assist in setting up IPCs.

Defects in classes for measures and space: Group A attributed the defects to inadequate, off-line pre-testing of the changed code. This pre-testing would have also required test suites simulating the real applications. Further analysis revealed that objects were multiplying unnecessarily, i.e., a new object would be created in every cycle unnecessarily. The code was revised to eliminate this problem. In consideration of safe design, and in the interest of eliminating timing variations due to memory allocation and deallocation, a guideline was given to create all the needed objects at startup and initialization, limiting dynamic creation and destruction only where necessary for the application.

Comparison with the decentralized alternative: It may be argued that an addition to an application would be simplified if a top-down, centralized style were not used. We contend that this "saving" only pushes the effort downtream, and support our contention with the following example case.

There is a style of organizing software (from the Smalltalk model), in which every object in the system has the functions to display or "present" itself and interact with the user. This style reduces the labor of building and maintaining a user interface. Whenever a new object is added, it "knows" how to present itself — no effort is required to build a "central" user interface program. This style is not suitable in a hard real-time control system for a number of reasons, discussed next.

Objects executing in hard real-time cannot be burdened with arbitrary addition of GUI workload — a premise well accepted in machine tool controllers (this project has supporting evidence too).

Secondly, the hard real-time processes and the user interaction processes may be operating under different environments on different platforms, requiring externalization and internalization of objects.

Thirdly, all functions of an object should not be made accessible to all users. Many functions are provided for diagnosis and debug of the control system, reconfiguration, setting servo-control parameters, etc. intended only for qualified personnel under restricted modes of access. Application-developers may wish to restrict access further. Our messaging scheme and protocols defined for each port of each process address these needs.

Fourthly, the user requires a presentation limited to the focus of attention, an arrangement that follows the flow of control, and timely intervention in the display by critical events. Self-displaying objects do not assure that the user's requirements will be met. An

integrated (top-down) user interface design is needed.

Event registration service: Another aspect of the decentralized style is dynamic event announcement and registration. Every process "advertises what it has to offer." "Interested" processes register with the advertiser for delivery of the produced information or notification of future events. This approach simplifies IPC design significantly. However, it imposes an unpredictable workload. Therefore, this architecture does not use this style during hard real-time operation. An event registration service is provided for startup and initial setup. The same building blocks may be used for future consumers of information produced by the hard real-time system, to modify static configurations. However, after all clients are registered the performance of the system should still be evaluated to assure that time-critical operations meet their speciifcations.

7.7 Status of architecture for coordinating tasks

When computations in a machine tool controller are distributed across several programs, the application system design becomes a much more complex activity, requiring significant application-specific engineering effort, much of which is spent in testing, "tuning", reworking, and debugging issues associated with inter-process communication, correctness of process sequencing, and timing. In order to support a variety of requirements (functionality, scalability in performance and cost ...), the application-design stage should have the flexibility of distributing work across a variable number of processes and processors.

7.7.1 Findings

General guidelines of work distribution are not adequate in reducing the effort and duration required to develop a hard real-time system, such as the one described in Figure 2.4. The use of "prefabricated" components (class libraries) for machine control domain-specific functions in building applications has also not been adequate in reducing the application development effort, because a larger proportion of the development time was spent in building, testing, tuning, and debugging the application programs.

When the control system is in the execution state, ready for regular operation, the inter-process communication protocols are simple — mostly of the single producer-single consumer pattern, and in some cases, of the single-producer multiple-consumer pattern. Yet, the programming of interprocess communication functions and the debugging of timing problems has been a time-consuming process, especially when the time-loading factor is high and a significant proportion of this load is associated with the use of system services.

During normal motion and machining operations in traditional machine tool controllers, there is little change in control flow. The domain architecture developed in this project provides systematic means of intervention in the execution of process programs when warranted by process state conditions sensed in real time, e.g., tool breakage. This intervention avoids the use of disruptive system services such as interrupts, in order to simplify the timing issues of application design.

7.7.2 Future work

Additional research is required to develop rules, guidelines, and tools that make it easier to assure correct timing and ordering of execution in a multi-tasking system.

Determination of event-processing order: When external inputs arrive into a process from multiple sources, how should their order of processing be determined? The underlying IEEE 1003.1b services only provide priority-based FIFO ordering for message queues and for scheduling processes. Domain-specific rules should be developed to specify the ordering. An application-level service should be developed to enforce the specification.

Priority assignment for periodic processes on same node: A related issue is the assignment of priorities to periodic processes with different time periods. RMA suggests that the process with the shortest period be assigned the highest priority and progressively assign lower priorities to other processes in the ascending order of their periods. Although this guideline simplifies process scheduling, it does not always reflect application requirements properly. We found a counter-example. The servo-process must run at a period of 10 milliseconds, with very little tolerable variation. A force data acquisition process has to run at a 0.5 millisecond period, but a larger variation is tolerable. Under RMA guidelines, we should assign a higher priority to the data acquistion process, but it causes unacceptable variation in the servo-control loop timing. RMA does not allow us to take advantage of the larger timing tolerance of the data acquisition process. This is a common case where the data acquisition process serves secondary monitoring needs, or the data is subjected to filtering, smoothing, or other reduction process, or where the reduced data is used in an outer longer-period processing loop.

Assuring run-to-completion semantics: IEEE 1003.1b does not explicitly specify a scheduling policy that will assure run-to-completion semantics. Some applicaton-level service should be developed that can work with commercially supplied POSIX-compliant scheduling policies, to assure run-to-completion semantics for time-critical processes, including processes that use shared memory resources.

Time cost model of OS-services: For specific (static) task configurations and repetitive (cyclically constant) workload, a time-cost model of OS-services would help in reducing the time to develop and validate a particular controller configuration.

The time cost of a service determined experimentally in isolation is not adequate, because it tends to be the best-case value, which increases substantially when the OS has multiple requests pending for service. A general model is not feasible, because of a multitude of dependencies on the application conditions. However, a characterization is possible for a particular configuration pattern. In this manner several common patterns for machine control could be characterized.

Exception handling: Even after developing the technology to design and validate systems for the normal case, a major gap remains in designing for the impact of exception conditions. When changes in control flow are required in multiple FSMs in multiple processes on multiple computers, assuring correctness of results is very difficult. These problems require research in several time horizons. In the near-term laboratory experiments are needed to identify the dominant factors. In the mid-term, field research is needed to characterize common cases and patterns. In the longer term, theoretical models should be developed and validated for use in future system design.

Software lifecycle economics: Long-term studies are needed to evaluate the efficacy of an architecture that supports reconfiguration, and to determine the boundary of the domain of applications or configurations that can be economically supported in a domain architecture.

CHAPTER 8

Conclusion

Emerging reconfigurable manufacturing systems require reconfigurable control systems. With the technology of controller software design in practice today, the needed reconfiguration is very difficult, i.e., the process is error-prone and the required effort, skill level, and duration amount to a prohibitively large and unpredictable cumulative cost. Most commercial control systems would require substantial re-engineering to satisfy the general needs identified in this project. We investigated an extensible domain-specific architectural model to *ease of reconfiguration*. Application of this approach to machine tool controllers is novel, and to hard real-time systems, in general, is a very early stage effort in a field requiring long-term research.

8.1 Research contributions

This project demonstrates a process of developing requirements specification for an application domain by generalization from a specific case. Chapter 2 introduces a synthetic case study to describe reconfigurations required in the evolution of a highly automated machining workstation. The case is synthesized to be representative of similar needs in many manufacturing systems in current use. Each reconfiguration need is over-generalized to include many other similar applications.

Section 2.2 formalizes a method to systematically derive reconfigurability requirements from a specific case. The method takes advantage of the property that machine tools are engineered systems, whose design is based on a mature body of engineering knowledge. The method draws upon domain-independent mathematical and computational structures and organizing principles for analysis, abstractions, and clustering. These generalizations are evolved iteratively into subdomain models, which are reviewed with domain experts. The domain and subdomain models are expressed in the object-oriented paradigm, in the form of classes and class graphs that capture interaction patterns. A key architectural modeling decision is to separate task structuring as a later stage of the software engineering cycle. Chapters 3–4 apply the requirements and modeling principles to an axis of motion in several iterations. In the process, several generic subdomains are discovered and modeled. Chapter 5 builds upon these models and applies the same principles to organize software for coordinating multi-axis motion. Chapter 6 focuses on external inputs and outputs for machine control and generalizes the model to other human machine interaction and information exchanges across independent programs. Chapter 7 extends earlier organizing principles and models and applies them to distribution and coordination of computational

work in an integrated control system.

Interface specifications for software components are developed on the scientific foundations of the object model, for the structural and static aspects, and the finite state machine, for dynamic aspects. Software conforming to these specifications has been prototyped and demonstrated in software development experiments with over twelve participants in the construction of the class design model, its implementation, unit testing, integrated application testing, and demonstration of motion control on a machine tool testbed.

It was demonstrated that participants with very limited domain knowledge and software development experience could develop and maintain software for the library classes. The class-level modularization and organization of information reduced the learning time for new participants. However, the class model or application programming interfaces (APIs) by themselves were not sufficient information for other participants to apply the classes independently. Additional textual documentation and example applications were required.

Test applications were structured as a set of cooperating tasks for ease of reconfiguration. However, significant amount of effort was required in setting up the task structures, inter-process communications, message structures, and timing relationships. Although the OO and FSM model of software organization made application reconfiguration easier, these paradigms and general software engineering principles were not sufficient in making the application development process simple and predictable. Additional heuristic guidelines, rules, and constraints developed in this research were helpful in further reducing the difficulties of application development and reconfiguration. Considerable more effort is needed before advanced hard real-time machine tool control applications can be developed and reconfigured reliably and economically from software modules sourced from diverse providers at different stages in the application lifecycle. Some research requirements found in this project, representative of the early stages of a development cycle, are described next.

8.2 Future directions

Our experiments in the development of software modules and applications using these modules in various configurations have revealed a number of opportunities for improvements in the efficiency of the development cycle. A combination of near-term, mid-term, and long-term research endeavors are needed to reduce the cost of composition, customization, reconfiguration, and integration. Two common threads run through the various aspects of further research. First, the research testbed should be upgraded with incremental extensions and refinements of the software class library, to evaluate ideas that reduce application development effort, increase reconfigurability through multi-sourced software modules and platform services, and scalability in functions, performance, and cost. Secondly, there should be careful record-keeping of the development experience at each stage – effort, duration, and difficulties encountered. The logged records should be analyzed to evaluate the efficacy of various modularization ideas in providing end-use economic benefits, reusability and extensibility of the modules, the necessity, sufficiency, and testability of their specifications, and the transferability of the knowledge required to build reconfigurable machine tool controllers. This is a long-term, multi-disciplinary research thrust, including field research to continue the monitoring and evaluation of reconfigurable controllers over their technologically and economically useful life. Some specific research issues are summarized next.

Initialization of information specific to a machine tool: Support for persistent objects for data that is constant from run to run can reduce the amount of effort required to initialize an application. Current practice uses a "configuration file." The user supplies the persistent data as ASCII strings in this file. The application code converts the strings into values for its corresponding data variables. A modest development effort is needed in string-to-object and object-to-string conversion functions. *Ease of configuration* research issues in the near-term include determination and validation of the most cost-effective approach to provide persistence — types of tools and forms of user input (e.g., spread sheet templates), tools to convert the user input into persistent objects (e.g., custom stand-alone program or an object-oriented database management system with an application program), and tools to transfer the objects from the user-friendly environment to the hard real-time subsystem (e.g., classes encapsulating rpc services specific to this domain, or more general services such as CORBA, or environment-specific services such as OLE).

Automated accessibility to interface specifications: A related near-term research issue is the accessibility of the class models (interface specifications) to application software, e.g., the OMG Interface Repository. It would help reduce the effort required in building message structures — our architecture specifies inter-process message structures that correspond to object-function signatures. The process of building the message structures requires user entries of information that is substantially a duplication of information provided in the class descriptions. In addition to the expenditure of effort, the process also leaves room for inadvertent errors in mismatches in the order or type of the parameters. This process could be semi-automated if class descriptions created in our model were accessible to message-structure building software — in current technology, class specifications (e.g., C++ header files) are economically accessible only by the compiler. Such a database could also be used by configuration tools. The research issue is the determination and validation of the most cost-effective form of this database — some candidate alternatives are the database of a CASE tool with an application program to query the database and an OODBMS with application programs to create, update and query the database.

Completeness of configuration specifications: What are the specifications on a configuration that can assure an implementation with correct performance under the management of OS services conforming to the IEEE 1003.1b? Although the specifications developed in this thesis reduce the configuration development effort, this question is not answered. Within the capabilities specifiable through IEEE 1003.1b, OS services do not guarantee the semantics of time-distance relationships required by data acquisition and continuous process control applications. Existing techniques to analyze schedulability focus on meeting deadlines — they do not assure time-distance relationships (upper, as well as lower bounds on latencies).

Domain-specific pre-scheduling: A near-term research project should develop the capability to construct an application-level precomputed fixed schedule and task release-control, focused on the inter-process interaction patterns identified in this thesis.

Interprocess interaction patterns: Iteratively and incrementally, this capability should be extended, with related research that identifies more inter-process interaction patterns commonly found in the hard real-time subsystems of this application domain and provision

of correspondingly specialized IPC objects. While our basic producer-consumer pattern of a fixed sequence cycle is required in many cases, it is overly restrictive in other cases where slight latency is tolerable. To take advantage of the decoupling afforded, one enabling near-term development is a circular buffer in shared memory with a monitor on the consumer's lag, which could allow efficient IPC and catch a violation of the latency limits. The cost of creating such an object could be amortized over many applications by designing it as a reusable, adaptable resource, i.e., a generalized circular buffer class with a built-in monitor.

Time cost model of basic OS services: Unfortunately, monitoring the lag of a consumer process for latency only catches a failure — it does not prevent the problem. The fixed sequence precomputed schedule and release time control can only take limited advantage of the allowable latency. To take advantage of any more scheduling flexibility, a performance model of the OS services (e.g., time cost modeling of interrupt servicing, signaling, messaging, semaphores, process scheduling) is needed. When focused on the configuration and communication patterns specific to this application domain, the OS performance characterization is a mid-term research endeavor. In general, broadening the domain to allow for more involved communication patterns, extends the OS performance characterization problem into a long-term research issue.

Composing and reconfiguring state machines: A key factor in easing reconfiguration is the user-specifiability of control flow. A number of open architecture controller projects have focused on APIs, and some include fixed state machine specifications for the behavior of certain functions. However, we have discovered that considerably more effort is spent in composing and reconfiguring applications, even with a full complement of library components in hand. To facilitate the composition of applications, we have developed library classes to build user-reconfigurable state machines. Further research is needed to evaluate this approach for development efficiency, execution efficiency, and testability for conformance to user requirements. Accomplishment of the *ease of reconfiguration* objectives at the motion coordination level is a mid-term research effort, and, at the servo-sensor level, it is a longer-term effort.

Reconfiguration studies in the field: While a laboratory research testbed is very useful in controlling conditions of the experiment, they offer limited utility in external validation. Therefore, instrumented prototype control systems should be placed in the field in working conditions more representative of a real controller life cycle, so that costs and benefits may be understood better, and new requirements may be discovered.

APPENDICES

APPENDIX A

Assumptions about the industrial environment

Following are the assumptions made about the industrial context in which the control development process and architecture fits. Factors affecting machine control software productivity in industry are characterized in terms of Boehm's cost drivers [10], i.e, computer (development platform) attributes, project attributes, and personnel attributes (profile of the developers), the number of potential applications, product attributes (typical application functions).

A.1 Computer attributes

Boehm uses this term for the development platform. The PC is the predominant platform for development, formerly under DOS, predominantly under Microsoft Windows, and rapidly being replaced by NT. The hardware platform for development is not a constraining factor. Therefore, we assume that a developer has access to a PC with ample memory, when needed. The main issue in development is the software environment, discussed next.

A.2 Project attributes

Following is an assessment of the state of practice in software development, given in terms of software process maturity, programming language, tools, and work scheduling patterns.

A.2.1 Modern programming practices

The software process in the controller supplier industry is at Level 2 of the Capability Maturity Model (CMM) [45]. Additions and integration in the field, e.g., to retrofit sensors, are at Level 1 of the CMM.
Software targeted for specialized processors is typically developed under correspondingly specialized tools. The C programming language has been the de facto standard in this industry. Specialized libraries are used for software to run on specialized processors. Device drivers are typically interfaced in an assembler language. C++ is being used increasingly, at least as a better C. Java is being introduced in the non real time functions of a controller. Little investment is made beyond the engineering of the current project — software is typically designed for the current product only. Due to budgetary constraints, requirements for the future are not well-engineered into the design for the current product.

Issue: Compilers that run on the PC and produce code to run on a specialized processor under a specialized operating system are available from limited sources only; their capability is limited.

A.2.2 Use of software tools

Typically CASE tools are not used in this industry. There is little support beyond the programming language and associated standard libraries.

A.2.3 Required development schedule

Short delivery times continue to be a serious constraint on the software developers. Product enhancements are implemented over a 3–6 month period. New products are typically targeted for an 9–18 month development period. Frequent interruptions of work are common during every phase of program-development, an influencing factor not isolated in Boehm's model.

A.3 Personnel attributes

The control system developer is typically an organization and work team distinct and distanced from the machine builder, who, in turn, is typically distinct and distanced from the user and maintainer. Thus there is very poor communication of end-user needs to the control developer. As a result, costs and lead time to satisfy new control-related user requirements are very high. The control system supplier organization also partitions its work force into hardware, platform software, and application software teams. This compartmentalization further reduces the quality and rate of communication, and thus, innovation. The current generation of software developers has grown under pressure to minimize the use of run-time hardware resources. As a result, their software is closely optimized for the target hardware with significant dependencies on it. In general the mindset of the developers does not favor encapsulation, isolation, modularization, portability, extensibility, or flexibility. "Real-time executives" have been "home-made" in this personnel-environment. On the other hand, the commercial hardware-independent real-time operating systems typically have high licensing costs and yet do not provide the specialized features the developers seek. The developer of the home-made "real-time executives" in automation controllers is less skilled in software engineering than the modern operating system developer, but is more knowledgeable about the application, although not adequately informed (see *Application experience* below).

Analyst capability: Control software development activities are not partitioned along Boehm's factors. Typically, the person is designated as a machine-control-engineer, automation-control-engineer, or equivalent, responsible for all phases of development. The person has many assignments in various phases of the project lifecycle. A senior engineer typically performs the role of requirements analysis and high-level design.

Applications experience: There is also a very large variation in the individual's understanding of the application domain. The industrial control engineer is typically viewed as a limited-service-provider to the overall mission of the manufacturing system or machine. The person seldom has a direct understanding of the physical processes and system, but rather depends on others to provide the system requirements and specifications. This

communication is typically incomplete, ambiguous, and iteratively clarified.

Programmer capability: There is a wide variation in education and experience. Older employees may have experience up to 30 years, but may not have a college degree. Newer employees are degreed, typically in mechanical or electrical engineering. Only a small fraction of the workforce has formal education in computer engineering. An even smaller fraction is formally educated and trained in software engineering. A person formally educated or trained in real-time systems is rare.

Programming language experience: The industrial control software developer is not a full-time professional computer programmer. Coding is a small and infrequent part of the person's activities. C is the most commonly understood language. Very few people know C++ and even fewer know how to use it properly.

Virtual machine experience: Most people perform their development work on a PC-compatible machine. In the past, the target machine has been some vendor-specific product and its programming interface (equivalent of virtual machine) is a vendor-specific higher-level language for programming machine control logic or motion control. Thus, user application programs are written in some vendor-specific higher-level language for programming machine control logic or motion control.

A.4 Number of potential applications:

The total number of industrial automation systems being developed in the U.S. is estimated to be a few thousand units a year, of which only a few hundred demand agility, i.e., high flexibility, extensibility, reconfigurability, and performance. Thus, relative to commercial application of computer systems, the number of potential applications is very small. As a result, the cost of supporting the less common requirements has not been affordable.

A.5 Product attributes

The application domain targeted in this study is servomotion-based equipment for manufacturing a mix of relatively high volume components for the automotive industry. Normal automatic operation of equipment is highly cyclic and predictable. The scope of the typical application functions is bounded within the functions of a manufacturing cell [5]. Table A.1 shows these functions, organized by levels of spatial span, which correspond to a reference model architecture, shown in Figures A.1–A.2 [1].

A.5.1 Required data reliability

Very high reliability is needed in timely updates at the servo-sensor level (Chapter 4, Figures A.1–A.2, Level 1). If the sensory data stream at this level is interrupted, physical operation must be stopped immediately. The consequential costs of such failures are many times greater than the cost of the control system. Such failures could endanger human safety. Most of the application software cost lies in handling exceptions, error conditions, and failures of various types and degrees. These failures present opportunities for advanced sensors to monitor condition of machine elements and cutting tools [4].

Level	Role	Spatial span
6	Cell functions	Several workstations
5	Workstation functions	Intra-workstation
4	Equipment functions	Multiple basic-machines
3	Machine-moves	Multiple axis-groups
2	Tool/workpiece moves	Individual axis-group
1	Servo-commands	Individual joint/axis

Table A.1: Spatial span of control levels in a manufacturing cell

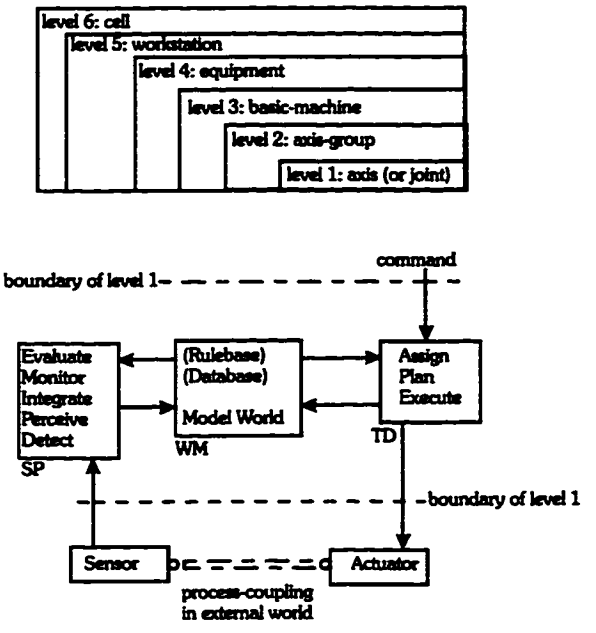

Figure A.1: Levels of control and tasks within a level

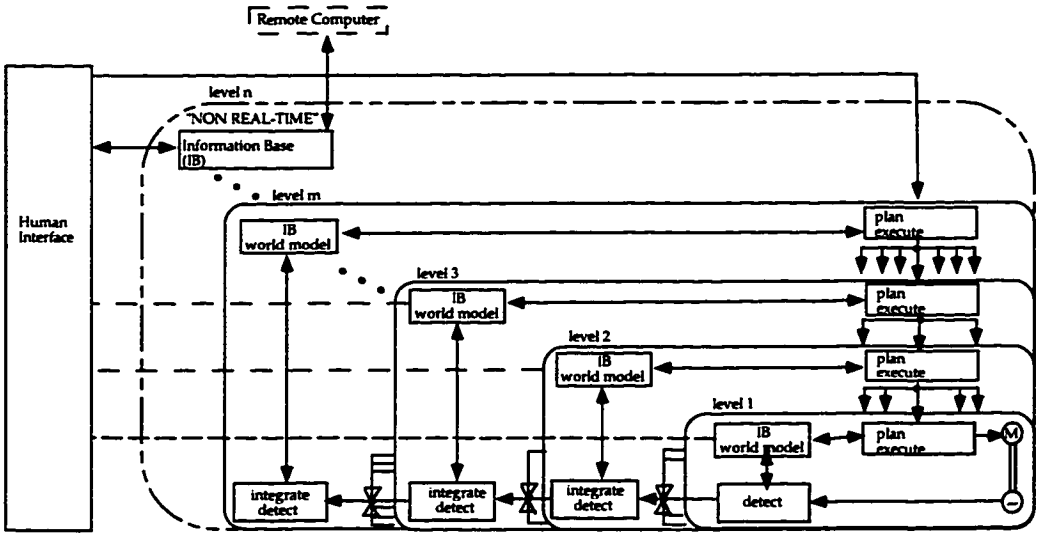

Figure A.2: Control hierarchy for an integrated manufacturing cell

A.5.2 Database size

Real-time data for control of motion and related discrete functions is small enough to keep the last sixty milli-seconds of data in main memory, when implemented on a PC platform (midrange, mainstream). For example, the number of objects involved in one axis of servo-motion control (Chapter 3, Figures A.1–A.2, Level 1) is in the order of ten. Each axis may has a number of data members – in the order of thirty. The number of axes in an axis group (Figures A.1–A.2, Level 2) may be in the order of ten, but over 90% of the applications have three axes or less, and over 40% have only one axis per group. The number of data items exchanged per period within an axis group is in the order of ten. The period at Level 2 may be longer than at Level 1 (multiple greater than 2). Time periods and their accuracy constraints at higher levels are progressively looser in a similar manner. Historic data can be condensed into abstracted parameters. The history-abstracts can be saved to persistent storage at intervals of thirty milli-seconds or larger. Real-time data may be forwarded to a different processor for persistence service. No such history is saved in current controllers.

A.5.3 Product complexity

Section A.5.7 describes the scope of functionality. Machine tools requiring multi-axis coordinated motion predominantly use computer numeric controllers (CNCs). Historically, CNCs and more sophisticated multi-axis motion controllers have been complex because of tight coupling: use of shared memory, global variables, pointer-access to data, and lack of adequate modularity, especially within the real-time executive. These practices stemmed from the concern to limit cost and size of hardware (processing capacity, storage, power supply), and cost and overhead of commercial real-time operating systems. Controllers were not designed for flexibility due to limited budgets and time. On the contrary, functionality was limited to serve the mainstream market, precluding customization to lower-volume customer needs.

As industry emerges from this environment, at the initial stage, industry will not be seeking dynamic reconfiguration of tasks or other runtime changes that change the workload significantly.

Multi-axis precision motion controllers are typically systems with multiple, often dissimilar, processors, interconnected with different buses. As an example, we describe a three-processor system, consisting of a platform for servo-motion control, a platform for other real-time functions, and a platform for non-real-time functions. Note that, in contrast to this *heterogeneous* environment, much of the research in multi-computer systems has been in a regular *homogeneous* environment, i.e., a group of identical or very similar processors or computers.

A.5.4 Platform for servo-motion control

Referring to Figures A.1–A.2, functions of Level 1 are mapped onto this platform. Outputs to power-amplifiers and inputs from position sensors are typically interfaced on a specialized processor, e.g., a digital signal processor (DSP) board — 8 sets per board is common. The DSP runs cyclic tasks, with no operating system on it — it provides hard real time guarantees to these tasks. Cycle times may be in the order of a millisecond.

Trend: Hardware that integrates a processor and interfaces to sensors and actuators is emerging for applications in the automobile (e.g., the Intel 80196CA chip), wireless communication, and multi-media consumer electronics. The lowest level of processing may be embedded in such hardware. Its output will be interconnected to higher levels in the control system through a real-time network, e.g., the Controller Area Network (CAN). Application-level protocol standards are emerging for interfacing to such networks.

Issue: Due to the limited and specialized software development environment for specialized hardware, only stable or relatively fixed functions are likely to be embedded on specialized processors. This environment is not conducive to rapid innovation in software.

A.5.5 Platform for other real-time functions

Referring to Figures A.1–A.2, functions of Level 2 are mapped onto this platform. Typically, it is a conventional processor on the DSP board (Section A.5.4), with a dual-ported memory shared between the two processors. Often, a "home-made" real-time executive manages the tasks and resources on this conventional processor. The real-time subsystem is often tight in resources. Modifications are very difficult. It is difficult to add new devices, e.g., sensors, within the real-time subsytem. Only the original equipment manufacturer (OEM) is able to provide such modifications, and, then, at great expense. Users could add sensors on the more accessible bus of the non-real-time platform (Section A.5.6), but their temporal correlation with events on the real-time subsystem is not close enough. This limits the rates of innovation in user factories.

Trend: Economics are driving these functions on to a target environment that enjoys high volume of usage in other fields, e.g., the PC platform.

Issues:

1. Developers desire some flexibility in the placement of real-time functions between the specialized servo-control platform and the conventional real-time platform.

2. There is no real-time operating system that has the maturity and popularity of DOS or UNIX. Some users and developers are hoping that NT will suffice. However, it is unclear if NT can provide adequate determinacy. The most mature real-time operating systems for the PC (Intel x86 hardware set) are Intel's iRMX and Quantum's QNX. Software development tools on these operating systems are not as abundant and cheap as tools available on Microsoft Windows/DOS, and not as powerful as tools available on Unix. Thus, the iRMX and QNX environments are also somewhat limited and not very conducive to rapid innovation.

A.5.6 Platform for non-real-time functions

The real-time subsystem is connected, through another more accessible bus, to a third processor that performs such functions as interfaces to the user and to a network external to the local machine control system. These more general services are provided under a more general-purpose operating environment, e.g. DOS, UNIX or Microsoft Windows or NT. Additional I/O, e.g., storage devices, when needed, are interconnected through the

I/O bus on the PC. Additional non-real-time computing capacity is loosely coupled and interconnected through the external network, e.g., a "thin-wire ethernet".

Trends: Certain soft-real-time functions can be placed on the non-real-time platform under NT or on the real-time platform. Some users and developers are seeking a uniform platform-interface for real-time functions and non-real-time functions and some flexibility in placement of tasks/processes between these platforms. More users are seeking a uniform software interface between the (non-real-time) controller platform and other shop-floor computers in the typical computer-integrated-manufacturing (CIM) environment. These shop-floor computers are typically UNIX workstations or PCs with NT gradually replacing DOS.

Issues:

1. Commercial off the shelf software packages, e.g, GUI tools, CASE tools, and compilers, available for DOS, Unix, and Windows cannot be run on current real-time operating systems.
2. It is costly and difficult to keep personnel proficient in multiple development environments.

A.5.7 Scope of functionality required in the future

The next generation of manufacturing automation requires more agility, i.e., changeover to new products and processes should take less time. Changing products and processes will introduce run to run variability and varying mix of products made on the same equipment will also increase cycle to cycle variability. To cope with this variability, more sensor-based initelligence will be required. Controller tasks may be viewed in the categories, as follows. *Monitoring tasks* are the most common. *Control tasks* are tasks required for control, in addition to the monitoring tasks. *Cognitive tasks* are additional tasks required to exhibit intelligent behavior.

Monitoring tasks

1. Acquire value of some variable sensed in the controlled system.
2. Collect a prescribed time-history of such values.
3. Reduce this time history to some meaningful parameter, in accordance with some prescribed procedure, possibly using equipment models.
4. Compute the expected value of the sensed or derived parameter, possibly using equipment models.
5. Compare the sensed or derived value with its corresponding expected value.
6. Compare the difference or deviation with allowable or prescribed limit.
7. Trigger prescribed action upon reaching or crossing such limit.
8. Store the intermediate computational results as prescribed. The prescription may include further reduction procedures and the maintenance of a time history. These reduction procedures may use equipment models.

Control tasks

1. Acquire value of some controlled variable or parameter from prescribed plan of execution (typically decomposed from a program for processing workpieces).

2. Decompose or transform this acquired value to values of variables or parameters to be controlled by execution agents. These transformations would use equipment models.

3. Distribute or transfer the values to these execution agents. The ultimate resulting values are set as outputs to some controlled actuators in the manufacturing equipment.

Cognitive tasks

Current manufacturing automation controllers provide little support for cognitive tasks, i.e., the less structured knowledge-acquiring computational tasks. One type of such tasks is *machine learning.* Here the scope of machine learning is limited to the fitting of parameter values in previously prescribed models, using prescribed model-fitting procedures. The purpose is to support the tasks of monitoring, control, prognostics, preventive maintenance, diagnostics, corrective maintenance, and enhancement or engineering improvements. The data for such machine learning may come from operational data or from controlled calibration tests, performance-evaluation tests, or other engineering experiments. The domain model and architecture should provide the structure of the models needed for such machine learning. Generally the exact values of parameters in the causal laws of engineering are not known. However, these values can be estimated with a combination of controlled calibration experiments and operational data. The given causal laws also allow estimation of the reaction time needed for each physical function to be serviced and the response time needed by the physical serving mechanism.
The domain models and architecture should also support learning about perception. Most of the time the perception is not at the point of interest, but at some remote location. Thus, feedback is correspondingly distorted and contains systemic errors, uncertainty and noise of measurement, in addition to similar deviations from the monitored process. Established causal laws and quantitative information do not allow clear isolation of these factors. The scope of the cognitive processes is to learn about the perception-model-parameters from operational data by applying available knowledge, and to generate and signal an alarm when it crosses some prescribed threshold of "ability to learn". Over the course of time, human review of the history of automated operation and learning is expected to yield improvements in these processes, thus changing the limits of "ability to learn".
Cognitive processes also support simplification and selection of models of other processes by association with the contexts and by limiting the accuracy to what is needed for the purpose. For example, consider the case of heating or cooling. The following different reasons for cooling impose progressively more difficult modeling and computation:

1. Maintain safe temperatures (easiest).

2. Avoid degradation of machine elements and fluids.

3. Maintain operational accuracy (most difficult).

Appropriate estimations of heat-generation and heat-transfer rates are needed to design cooling capacity in the system. For operational accuracy, on-line control of temperature

would also be beneficial. Feedback control is not sufficient, because of the long time-constants in temperature-changes. Therefore, some degree of predictive control is required. Some of the same engineering information that is needed for the design of the cooling system capacity could be used in predictive control, along with on-line information about the different operational speeds and loads that affect the heat-generation rate in various parts of the machine. This collective information makes it possible to regulate cooling and active heating to minimize the fluctuation of temperatures. Merely balancing heat-gain and heat-removal could reduce much of the fluctuation. The initial models should provide the knowledge to predict and account for such first-order effects. The modeling framework should also provide the facilities to improve upon these models as experience is gained. While there are other examples of "higher intelligence" in maufacturing automation, there is little reusability of the real-time control software. Therefore, in this project we have limited the scope to the basic cognitive tasks described above.

Modularity of a control system

Modularity should support a desired change, in any external characteristic of the overall system, at a low enough cost to allow the change to be performed economically. The change and its effects should be localized, i.e., there should be no ripple effects and side effects in other parts of the control system. It implies that modularity in the control system should support and map the modularity of the controlled system.

1. It should be easy to reconfigure, extend, and enhance the initial system. The smaller the cost-threshold or granularity of improvement allowed to the user, the higher will be the value or degree of openness of the system.

2. The roles of the replaceable components of a system and their inter-relationships should be clearly and completely documented and verifiable, through well-known procedures.

3. Compatibility and compliance information must be well-defined, at the level of component granularity necessary to implement the desired reconfiguration, enhancement, or extension. For examples, see cost factors enumerated in Section A.5.7. This information must include the following:

    - the services of each replaceable component,
    - the performance-specification of the component,
    - its external interface, and
    - verification procedures

4. The compatibility and compliance information should be sufficient for third party vendors to become component-providers economically, without further dependence on the original control system supplier.

5. Costs of licensing components should be insignificant, i.e., not preclude the economics of the desired enhancement or extension of the manufacturing system.

Some example situations follow.

Example 1: A portion of a user-display screen is to be revised due to some change in a sensor. Can this change be made without causing any other change in the user interface?

Example 2: A user discovers that scanning certain inputs more frequently and others less frequently would increase benefit for the same computational resources. How many software units will require a change? The user attaches value to products that allow such improvements in small increments of cost and that allow users to perform such improvements by themselves or through third parties. The user envisions a supply chain in which its direct supplier is a system integrator, who, in turn, would procure and integrate component wares, analogous to building a personal computer system. A modular architecture would enlarge the pool of suppliers , especially for after-market supply of retrofittable components.

Cost drivers for modular control systems

Users are seeking open control systems to minimize life cycle cost-to-benefit ratio associated with controls, while obtaining initial installed cost lower than current costs for equivalent functionality. Some factors affecting lifecycle economics, roughly ordered by importance, are as follows:

1. Cost of training personnel in operation, maintenance, programming, updating and upgrading. Costs are spread across the life in service. These costs are increased significantly, because these personnel have to cope with differences across controllers in use. By the time personnel return to work with the same controller, they have forgotten necessary details about it. Such costs should be minimized, e.g., through standard interfaces.

2. Cost of wiring, interconnections, associated errors. These costs occur initially and also at time of equipment reconfiguration and changes in sensors and actuators.

3. Cost of IO devices, their installation, and their integration other than wiring. These devices include discrete IO as well as power-amplifiers and feedback devices for servo-drives.

4. Associated cost of opportunity lost when users avoid upgrading economically obsolete components.

5. Ability to reconfigure the control system as easily as reconfiguring a PC (personal computer):

   (a) Ability to reuse and integrate easily and efficiently previously developed and proven components. For example, such IO devices as sensors and actuators and such human interface components as status indications, command inputs, icons.

   (b) Ability to use and integrate or temporarily deploy previously developed and proven tools and aids. For example, screen-building tools, programming interfaces, debugging aids, performance monitoring tools.

APPENDIX B

Supporting class structures

B.1 Axis related class structures

Constructor-destructor functions: Omitted for brevity
Accessor functions for following members:
measure axisOutput
int actuatorSetpoint
LowerKinematicModel *lKin (private)

Other member functions:
void output(measure outputNextState)

Class-structure B.1: Interface of class AxisActState.

Superclass: Axis
Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
TransaxisSensedState *axisSensedState
TransaxisSetpoints *axisSetpoints
TransaxisSetup *axisSetup
TransTravelCapabilities *axisTravelCap
length_measure retractPos

Other member functions:
Boolean checkOTravel()
Boolean checkInPosition()

Class-structure B.2: Interface of the TranslationalAxis class

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
OperationalLimits operationalLimits
AxisTravelLimits travelLimits

Class-structure B.3: Interface of class AxisSetup.

B.2 Class structures pertaining to space and kinematics

Constructor-destructor functions: Omitted for brevity
Other member functions:
const CoordinatesPtrVector get_coordinates() const
void set_coordinates(const CoordinatesPtrVector value)
length_measure * get_coordinates(int index)
void set_coordinates(int index, length_measure *value)
length_measure * distance(cartesian_point p)

Class-structure B.4: Interface of class cartesian_point.

Superclass: HomogeneousTransformMatrix
Constructor-destructor functions: Omitted for brevity
Other member functions:
void rotate(
plane_angle_measure roll,
plane_angle_measure pitch,
plane_angle_measure yaw,
CoordinateFrame& newFrame)
void translate(
length_measure x,
length_measure y,
length_measure z,
CoordinateFrame& newFrame)
void transform(
plane_angle_measure roll,
plane_angle_measure pitch,
plane_angle_measure yaw,
length_measure x,
length_measure y,
length_measure z,
CoordinateFrame& newFrame)

Class-structure B.5: Interface of class CoordinateFrame.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
CoordinateFrame baseFrame
CoordinateFrame placementFrame

Class-structure B.6: Interface of class KinStructure.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
UpperKinematicModel upperKinematicModel
LowerKinematicModel lowerKinematicModel

Class-structure B.7: Interface of class AxisKinematics.

B.3 Modeling axis components

Section 3.5.1 had introduced the primary purpose for an on-line model of axis components — deriving the lower kinematic model of an axis. Therefore, each component is viewed as a kinematic transformer. We also include related attributes for dynamics, operating limits, and life expectancy. All axis components are derived from the root class AxisCompt.

As in the case of the lower kinematic model of an axis, the main abstractions are the input, output, and the corresponding domain of the input and output, their transformation relationship (staticGain) and the functions inOutTransform(...), outInTransform(...) (Class Structure B.8). The component model also includes operating limits on the environment of the component, e.g., temperature range, relative humidity, acceleration, and jerk. It also provides a data member for the current or estimated operating temperature, which affects operating characteristics, as well as life.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
measure input
measure inputLowerLimit
measure inputUpperLimit
measure output
measure outputLowerLimit
measure outputUpperLimit
gain_measure staticGain
LinearAcceleration vibrationLimit
measure jerkLimit
thermodynamic_temperature_measure operatingTemperature
measure relativeHumidity
thermodynamic_temperature_measure temperatureUpperLimit
thermodynamic_temperature_measure temperatureLowerLimit
time_measure ratedLife

Other member functions:
measure output inOutTransform(int input)
int input outInTransform(measure output)

Class-structure B.8: Interface of class AxisCompt.

AxisCompt is specialized into four subclasses of axis components — FeedbackSensor, Actuator, Drivetrain, JointPair. We explain the specialization approach next, using feedback sensors as an example.

B.3.1 Modeling feedback sensors

The most common feedback sensors for controlling precise motion are sensors for position and velocity Figure 3.4). Using (Procedure 2–1), we generalize to a common superclass FeedbackSensor (Class Structure B.9). As the model evloves in the future, other features common to all feedback sensors may be added to this class.

Constructor-destructor functions: Omitted for brevity
Other member functions:
void configureInterface(byte nPort, SlaveDevice *value, int mode)

Class-structure B.9: Interface of class FeedbackSensor.

Position sensors. Position measurement in most computer controlled machine tools for machining is done with incremental pulse encoders (Figure 3.4). However, occasionally, e.g., in the case of rotational axes limited to one revolution of rotation, an absolute encoder may be used. Other types of position measurement devices, e.g., laser interformeters, ultrasonic sensors, may also be used, depending upon the precision required, ease of retrofit, and other application-specific considerations. Therefore, we set up a generalized class for

position sensors (Class Structure B.10). The function configureInterface(...) inherited from FeedbackSensor is redefined to allow application-specific implementation. The function getValue() is provided to access the device driver and obtain the current sensor reading from it.

Constructor-destructor functions: Omitted for brevity
Other member functions:
void configureInterface(byte nPort, SlaveDevice *value, int mode)
virtual measure getValue()

Class-structure B.10: Interface of class PositionSensor.

Angular position sensors. Position sensors may be rotary or linear — we find both cases in Figure 3.4. Rotary sensors have some common characteristics. They require some means of signaling the start or completion of a rotation, i.e., an angular reference marker. They can be applied to rotary axes directly. Since they measure a plane angle, the data type can be restricted to plane_angle_measure. Therefore, we provide accessor functions for the position measured by the sensor, served in *radians* – the SI unit of plane_angle_measure. Through specialization of this function, corrections and compensations specific to an implementation may be added. Other features common to angular position sensors may be added in the future as the model evolves.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following object members:
plane_angle_measure position

Other member functions:
void configureInterface(byte nPort, SlaveDevice *value, int mode)
virtual measure getValue()

Class-structure B.11: Interface of class AngularPositionSensor.

Incremental rotary encoders. The incremental rotary encoder is the most common type of position measurement device in an axis for precision machinery. It generates a pulse corresponding to a certain increment of motion between two elements of the transducer. The size of the increment varies with specific devices. In order to sense direction of motion, typically there are two such transduction channels within the same device, producing waveforms, say A and B, at quadrature to each other. If the condition " A leading B" corresponds to the positive direction, then the condition " B leading A" corresponds to the opposite direction. The pulses from channel A are counted as additive (counting up) and the pulses from channel B as subtractive (counting down) from some initial reference count corresponding to the designated origin or home position of the axis. Some devices come equipped with this up and down counting and provide a net count, corresponding to the position of the axis, i.e., they have built-in quadrature decoding. Other devices require external quadrature decoding. This distinction can be specified in the parameter mode of the function configureInterface, which is redefined in this class.

Constructor-destructor functions: Omitted for brevity
Accessor functions for following data members:
int port

Other member functions:
void configureInterface(byte nPort, SlaveDevice *value, int mode)
virtual measure getValue()

Class-structure B.12: Interface of class IncrementalRotaryEncoder.

B.4 Interfaces to external IO

```
class AnalogIO : public SlaveDevice
{
public:
AnalogIO();
AnalogIO();
virtual void SetGain(byte nPort, word nGain) = 0;
virtual word GetGain(byte nPort) = 0;
virtual void Reset() = 0; //resets HW into power-on state.
virtual void Configure(byte nPort) = 0;//cfg 1 port/channel
virtual void Initialize(byte nPort) = 0;//init if output
virtual word GetPort(byte nPort) = 0;//reads dig val if input
virtual void SetPort(byte nPort, word nValue) = 0;//writes if output
double GetValue(byte nPort);
void SetValue(byte nPort, double dValue);
byte GetNumPort();//num of ports on board
byte GetWordLength();//word length
word GetTimeConstant();
word GetHardGain();
double GetRange();
protected: //hardware specific settings
void SetNumPort(byte nValue);
void SetWordLength(byte nValue);
void SetTimeConstant(word nValue);
void SetHardGain(word nValue);
void SetRange(double dValue);
void SetMaxValue(word nValue);
private:
byte nNumOfPort;
byte nWordLength;
word nTimeConstant;
word nHardGain;
word nMaxValue;
double dRange;
};
```

Class-structure B.13: Interface of class AnalogIO.

```
class XVME500 : public VMEbus, public AnalogIO

public:
XVME500();
XVME500(CPUBoard *Cptr);
XVME500(CPUBoard *Cptr, long nBase);
XVME500();

void CheckBoard(long nBase = 0);
void Configure(byte nPort);
void Initialize(byte nPort);
void Reset();
word GetPort(byte nPort);
void SetPort(byte nPort, word nValue);
word GetGain(byte nPort);
void SetGain(byte nPort, word nGain);
;
```

Class-structure B.14: Interface of class XVME500.

```
// Driver for the Industrial I/O Pack (IP) Series IP320 module.
// 12-bit, 20 differential or 40 single ended analog input channels.
class IP320 : public AnalogIO, public VMEbus
{ public:
IP320();
//Cptr: pointer to the cpu board which uses ip320 services.  //nBase:  base address of ip320
IP320(CPUBoard *Cptr, long nBase = 0);
IP320();
void Configure(byte nPort);
void Initialize(byte nPort);
void Reset();
word GetPort(byte nPort);//read converted value from nPort
word GetGain(byte nPort);//nPort must be currently selected for read.
void SetGain(byte nPort, word nGain);//SW gain (val 1,2,4,8 only).

private:
void ControlWrite(word value);//Write to the control register.
word ControlRead(); //Read the control register.
void TriggerConversion();//start conversion.
void SetPresentGain(word value);
//set presentGain' to 'value' 1, 2, 4 or 8; recalibrate ip320
//called by GetPort only when necessary; no intended for user.
//Data integrity is maintained by the function SetGain.

// Set control register for next read to be from 'port' with gain 'gain'.
void SetControl(byte port, word gain);

word CorrectData(word data); // Correct read value with calibration info.

private:
int gainArray[40];
int presentChannel;
double voltCalHi; //Used in data correction. Dependent on 'presentGain'
double voltCalLo; //Used in data correction. Dependent on 'presentGain'

//Count values calculated during calibration; used for data correction.
word countCalHi; // Count value for 'voltCalHi'.
word countCalLo; // Count value for 'voltCalHi'.
```

Class-structure B.15: Interface of class IP320.

```
class DigitalIO : public SlaveDevice
{
public:
DigitalIO();
DigitalIO();

//reads byte value from the input port specified by nPort
//and returns the boolean value of the bit position
//specified by nBitNo - an integer from 1 to 8
virtual Boolean GetBitValue(byte nPort, byte nBitNo) = 0;

//writes bValue to the bit position nBitNo of output port nPort.
virtual void SetBitValue(byte nPort, byte nBitNo, Boolean bValue) = 0;

virtual void Reset() = 0;//resets HW to power-on state.

virtual void Initialize(byte nPort, byte nValue) = 0; //init nPort with nValue.

// reads byte from input port specified by nPort, masked bit-wise masked by nMask, and result returned.
virtual byte GetByteValue(byte nPort, byte nMask) = 0;

// writes nVlaue to nPort; nMask specifies the bit masked position
virtual void SetByteValue(byte nPort, byte nMask, byte nValue) = 0;

//configures port identified by nPort to mDirection enumerated as follows. IN_DIR: input direction.
OUT_DIR: output direction. BI_DIR: bidirectional.
virtual void Configure(byte nPort, DIRECTION mDirection) = 0;

virtual byte GetByteValue(byte nPort) = 0; //reads byte from nPort.
virtual void SetByteValue(byte nPort, byte nValue) = 0; //writes to nPort.

word GetNumPort(); //returns the number of ports on the board.
word GetPortWidth(); //returns the number of bits on a port.

protected: //hardware specific settings
void SetNumPort(word nValue);
void SetPortWidth(word nValue);

private:
word nNumOfPort;
word nPortWidth;
};
```

Class-structure B.16: Interface of class DigitalIO.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. S. Albus, "RCS: A reference model architecture for intelligent control," *Computer*, vol. 25, no. 5, pp. 56–59, May 1992.

[2] G. Arango, "Domain analysis methods," in *Software Reusability*, W. Schafer, R. Prieto-Diaz, and M. Matsumoto, editors, Ellis Horwood, 1994.

[3] J. Baer, J. Barg, E. Beck, C. Hardenbusch, P. Lutz, J. Muller, M. Novo, J. Pietchmann, M. Sozi, and W. Sperling, "Open system architecture for controls within automated systems, phase ii, access to man machine control, motion control, and logic control," Technical Report EP 9115 Work package 4 Deliverable D 2422, Esprit III, April 1996.

[4] S. K. Birla, "Sensors for adaptive control and machine diagnostics," in *Technology of Machine Tools — Machine Tool Controls*, R. V. Miskell, editor, number UCRL-52960-4 in Technology of Machine Tools, chapter 7.12, pp. 1–70, Lawrence Livermore Laboratory, University of California, Livermore, California 94550, October 1980. A survey of the state of the art by the Machine Tool Task Force.

[5] S. K. Birla, "Conceptual modeling of manufacturing automation," Technical Report CSE-TR-220-94, University of Michigan, Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, College of Engineernig, 2220 EECS Building, Ann Arbor, Michigan 48109-2122, 1994.

[6] S. K. Birla, H. Egdorf, R. Igou, J. Michaloski, D. Sweeney, G. Weinert, D. Uchida, and C. J. Yen, "An open architecture model of system development," in *Proceedings of the ASME Dynamic Systems and Controls Division*, K. Danai, editor, volume DSC-Vol. 58, pp. 277–282, 1996.

[7] S. K. Birla, J. Korein, et al. *Next Generation Workstation/Machine Controller (NGC)*, November 1987.

[8] S. K. Birla and K. G. Shin, "Intelligent control of manufacturing automation: Making it affordable and maintainable," in *Proceedings of the 27th CIRP International Seminar on Manufacturing Systems Design, Control And Analysis of Manufacturing Systems*, pp. 49–55, Ann Arbor, Michigan, May 1995.

[9] S. K. Birla and K. G. Shin, "Software engineering of control systems for agile machining: An approach to lifecycle economics," in *Proceedings of the 1995 IEEE International Conference on Robotics and Automation, Nagoya, Japan*, pp. 1086–1092, May 1995.

[10] B. Boehm, *Software Engineering Economics*, Englewood Cliffs, 1981.

[11] G. Booch, *Object Oriented Analysis and Design with Applications*, The Benjamin/Cummings Publishing Company, 390 Bridge Parkway, Redwood City, California 94065, 1993.

[12] R. Braek and O. Haugen, *Engineering Real Time Systems*, Prentice Hall, UK, 1993.

[13] J. Bruhl, "Open system architecture for controls within automated systems, phase ii, support for application configuration," Technical Report EP 9115 Work package 3 Deliverable D 2322, Esprit III, April 1996.

[14] *Specification and Description Language SDL Recommendation Z.100*, CCITT, Geneva, 1993.

[15] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1286, 1988.

[16] J. Denavit and R. Hartenberg, "A kinematic notation for lower-pair mechanisms based on matrices," *Journal of Applied Mechanics*, pp. 215–221, June 1955.

[17] M. A. Donmez, C. R. Liu, and M. M. Barash, "A generalized mathematical model for machine tool errors," in *Modeling, Sensing, and Control of Manufacturing Processes*, K. Srinivasan et al., editors, ASME Press, 1988.

[18] S. Faulk and D. Parnas, "On synchronization in hard real-time systems," *Communications of the ACM*, March 1988.

[19] P. Freeman, "Reusable software engineering: A statement of long-range research objectives," Technical Report TR-159, University of California, Irvine, ICS Dept., 1980.

[20] B. O. Gallmeister, *Programming for the Real World*, O'Reilly Associates, Inc, Sebastopol, CA, U.S.A., 1995.

[21] H. Gomaa, *Software design methods for concurrent and real-time systems*, Addison Wesley, Reading, Mass, U.S.A., 1993.

[22] H. Mason and PMAG, "ISO 10303 Industrial automation systems and integration - Product data representation and exchange – Part 1 Overview and fundamental principles," Technical report, International Standards Organization TC 184/SC4/PMAG, September 1992.

[23] C. C. Han, K. J. Lin, and C. J. Hou, "Distance-constrained scheduling and its applications to real-time systems," *IEEE Transactions on Computers*, vol. 45, no. 7, pp. 814–826, July 1996.

[24] D. Harel et al., "On the formal semantics of statecharts," in *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pp. 54–64, New York, 1987, IEEE Press.

[25] *Portable operating system interface (POSIX)—Part 1: Application Program Interface (API) [C Language]—Amendment: Realtime Extensions*, Institute of Electrical and Electroics Engineers, 1994.

[26] I. Jacobson et al., *Object-oriented software engineering - a use case driven approach*, Addison-Wesley, 1992.

[27] Y. Koren, *Computer Control of Manufacturing Systems*, McGraw-Hill, 1983.

[28] Y. Koren, *Robotics for Engineers*, McGraw-Hill, 1985.

[29] T. Lane, "Studying software architecture through design spaces and rules," Technical Report CMU/SEI-90-TR-18, Carnegie Mellon University, 1990.

[30] P. A. Laplante, *Real-Time Systems Design and Analysis*, IEEE Press, 445 Hoes Lane, PO 1331, Piscataway, NJ 08855-1331, 1993.

[31] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[32] T. Lozano-Perez, *Autonomous Robot Vehicles*, chapter Foreword, Springer-Verlag, 1990.

[33] A. Ludwig and WG3, "ISO 10303 Industrial automation systems and integration - Product data representation and exchange - Part 105 Integrated application resource: Kinematics," Technical Report 265, International Standards Organization TC 184/SC4/WG3, November 1993.

[34] P. Lutz. *Presentation and discusssion of OSACA*. Private communication with P. Lutz, ISW, Stuttgart, Germany, March 1997. OSACA/TEAM-API Workshop – see [3,13].

[35] *Next Generation Controller Specification for an Open Systems Architecture Standard–Overview*, Manufacturing Technology Directorate Wright Laboratory, September 1994. Wl-TR-94-8032.

[36] *Next Generation Controller Specification for an Open Systems Architecture Standard*, Manufacturing Technology Directorate Wright Laboratory, September 1994. Wl-TR-94-8033.

[37] D. J. Miller and R. C. Lennox, "An object-oriented environment for robot system architecture," *IEEE Control Systems*, vol. 11, no. 2, pp. 14–23, 1991.

[38] *Programmable controllers – Part 1: General information*, National Electrical Manufacturers Association, December 1994. approved as an ANSI Standard, December, 1994.

[39] *Programmable controllers – Part 3: Programming languages*, National Electrical Manufacturers Association, December 1994. approved as an ANSI Standard, December, 1994.

[40] *Next Generation Workstation/Machine Controller (NGC) Requirements Definition Document (RDD)*, 1989.

[41] Object Management Group, *CORBA: Common Object Request Broker Architecture and Specification*, Object Management Group, Framingham, MA, 1995.

[42] Object Management Group, *The Object Management Architecture Guide*, Object Management Group, Framingham, MA, June 1995.

[43] D. Parnas and P. Clements, "A rational design process: how and why to fake it," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 251–257, February 1986.

[44] D. Parnas, P. Clements, and D. Weiss, "The modular structure of complex systems," *Proceedings of the Seventh IEEE International Conference on Software Engineering*, pp. 408–417, March 1984.

[45] M. Paulk et al., "Capability maturity model for software version 1.1," Technical Report CMU/SEI-93-TR-24, Software Engineering Institute, Pittsburgh, 1993.

[46] R. Prieto-Diaz and G. Arango, *Domain Analysis and Software Systems Modeling*, The IEEE Computer Society Press, Los Alamitos, California, 1991.

[47] J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[48] R. Schappell et al. *Next Generation Workstation/Machine Controller (NGC) Needs Analysis*, February 1990.

[49] B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-oriented modeling*, Wiley, 1994.

[50] M. Shaw and D. Garlan, *Software architecture - Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[51] C. U. Smith, *Performance engineering of software systems*, Addison-Wesley, 1988.

[52] *Reuse-driven software process guidebook*, Software Productivity Consortium, 1993.

[53] J. Spivey, *The Z notation: A reference Manual*, Prentice Hall, 1989.

[54] D. Stewart, R. Volpe, and P. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," Technical Report CMU-RI-TR-93-11, Carnegie Mellon University, July 1993.

[55] J. Xu and D. Parnas, "On satisfying timing constraints in hard real-time systems," *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 70–84, January 1993.