Reconfigurability, extensibility, and reusability aspects of open-architecture control systems for intelligent manufacturing

Sushil Birla Kang Shin

Real-time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122, U.S.A.

Abstract

We report our experience in evolving an open-architecture design for computer-controlled motion of machine tools, building on available scientific and engineering knowledge. The economic goal of openness in this project was to minimize the cost of software over the lifecycle of an application. These costs impede innovation in intelligent controls, particularly associated with mechatronics and peripherals sourced from different suppliers at different times—costs typically discovered at the debug and integration stages. Provisions for these changes and add-ons cannot be predesigned always—the system design and architecture must have flexibility and adaptability in anticipation. We describe how our approach to the architecture and the software process facilitates reuse, adaptation, and reconfiguration, through our experience in a laboratory testbed.

1 Introduction

We report our experience in evolving an open-architecture design for computer-controlled motion of machine tools, building on available scientific and engineering knowledge [1–4]. Lacking a well-accepted or clear definition of "openness", we oriented ourselves to the goal of minimizing life cycle cost-to-benefit ratio associated with controls in an environment of ongoing changes in the manufactured product, process, and equipment, either due to change in needs or due to technological obsolescence (non-competitiveness). As reported in various previous studies [16], machine controllers become obsolete most frequently (often three retrofits over the life of the "iron" part of the machine tool). Next in frequency of changes are the mechatronics, e.g., drives and sensors. Addition of peripheral automation, e.g., for inspection or handling, is also common. The dominating cost factors are interfacing and integration. The largest cost-overruns are in software, especially the phase that is euphemistically known as "debug". Much of this cost could be attributed to lack of appropriate interface specifications at the time of procurement. A study of experiences in other applications of software [18, 19] revealed that it is unrealistic, at the time of initial speciifcation, to predict changes precisely and define interfaces specific to those changes. The field of software engineering has concluded that it pays to anticipate the nature of change, to plan for these changes as a class of partially-defined applications (domain), and to develop and follow a domain-specific architecture. Then, generic building blocks conforming to such an architecture can be reused and adapted at less cost than development from scratch. The lifecycle economics of this approach is promising but not proven. Adoption of such an architecture requires a high capability and maturity-level of the software development process in the community of its users. In contrast, we find that the state of software practice amongst researchers, developers, and practitioners in the manufacturing automation field is much below the state of the art.

In Section 2, we describe how our approach to the architecture and the software process facilitates reuse, adaptation, and reconfiguration.

Project background. This study is being performed in cooperation with several collaborative projects. At the University of Michigan, an open architecture testbed is evolving to support research in motion control, supervisory process control (ref Pasek imece96 paper), and real-time systems. Researchers in Mechanical Engineering and Computer Engineering are collaborating, with seed funding from the National Science Foundation, and with the support of several industrial organizations. The architecture is evolving in collaboration with the "Technologies Enabling Agile Manufacturing" (TEAM) program (see Michaloski's imece96 paper). It is part of an effort to develop application program interfaces (APIs) for the TEAM "Intelligent Closed Loop Processing" (ICLP) project.

ICLP. Here, we use the term in a general sense. By *Intelligence* we mean the ability to bring to bear all the knowledge that one has in the service of one's goals [15]. By *closed loop*, we mean having any kind of sensory feedback utilized in the control system. *ICLP* may apply to any manufacturing process [2, 4].

2 The Information Model

We describe the underlying principles, considerations in bounding the scope of the domain, the concept of a reusable assets library, and how we applied it to the ICLP domain.

Relationship to existing standards. This information model was developed upon principles consistent with existing and emerging international standards [9, 11, 12].

2.1 General principles underlying the information model

Following are the general principles used in developing the information model for the ICLP domain (henceforth described as the ICLP domain model).

2.1.1 Facilitation of comprehension.

It is almost certain that programs that are the most comprehensible will also be the ones easiest to modify [6]. Since the maintenance phase of software development costs possibly three times as much as the initial development or more, its reduction is an important objective. The ICLP domain model will bridge significant communication gaps among technology specialists, manufacturing engineers and computer programmers—it could serve as the universe of discourse across these communities.

2.1.2 Correspondence with concepts in the application domain.

Lifecycle effort can be reduced if software units correspond to physical objects, logical units, concepts, and primitives in the application domain. It minimizes effect of physical changes on software. It minimizes unnecessary communication across software units. It also facilitates comprehension within the community of its users.

2.1.3 Orthogonal decomposition.

Lifecycle effort will be reduced if the domain is decomposed into independent characteristics. It facilitates comprehension. It reduces the number of features modeled. A relatively small complement of orthogonal building blocks can yield a relatively large number of composed applications.

2.1.4 Abstraction of commonalities.

Abstractions are based on the underlying scientific and mathematical principles and concepts upon which applications are engineered. This approach mitigates the traditional problem of not knowing enough about applications ahead of time.

2.1.5 Organizing commonalities into hierarchies.

It facilitates comprehension and reuse. The value of organizing knowledge in a generalization-specialization hierarchy is well-acknowledged [5].

2.1.6 Organizing concepts into larger logical units.

Lifecycle effort will be reduced by organizing fundamental concepts into larger logical units, through clustering, aggregation, composition, and other forms of organization. Such organization reduces complexity and facilitates comprehension by reducing the volume of information within the user's focus. However, if this aggregation is taken too far, the number of such composed units could be even larger than the number of composing units! In the extreme, if we composed units into executables equivalent to popular word processors and spreadsheet processors, there would be a huge variety, and reuse of any one executable would be more limited than reuse of its constituent building blocks. The large variety in applications and relatively small number of users presents challenges in the ICLP domain, not found in developing word processors and spreadsheets.

2.1.7 Incremental and iterative development within an overall reference architecture.

Lifecycle cost-benefit ratio can be improved in this manner. It also allows people and organizations the time they need to grow into the new paradigm.

2.2 Additional principles for bounding the ICLP Domain

Certain significant principles of the domain-bounding technique are described next.

2.2.1 Existence of mature engineering knowledge.

The scope is limited to a domain of commercial manufacturing automation that is designed with well-established principles of engineering, a known technology base, known trends, and that knowledge has a large enough following of its users, so that there is continuity, endurance and stability of design practices.

2.2.2 Comprehensibility of abstractions.

The abstractions in the domain that follow from the engineering knowledge mentioned above must have a large enough following of its users, in order to have enough applications to make the software economical. In this respect, trade practice will have to grow more comfortable with abstractions that may be natural to the research community.

2.2.3 Commonality-variability relationship across domain.

To obtain favorable lifecycle economics, the domain should have sufficient commonality across various applications, in relation to the application-specific features. Thus, the current ICLP domain model provides more support to the more common cases. It provides structural support for less common cases to the extent of the underlying mathematical and scientific abstractions.

2.2.4 Value-to-investment relationship.

The scope of the domain should be such that there is a large enough potential benefit in relation to the investment in the reusable asset, such that the net present amortized cost is less than the cost with current practice. As an example, potential benefit can be estimated in terms of the number of applications, if all applications were equally rewarding. In the case of flexible manufacturing automation, almost every axis or joint of motion is servo-controlled. Axis position and velocity are commonly sensed variables, using a variety of sensors, based on different principles, providing different performance characteristics, with differences in software. However, there are less commonly used properties of these measurements that pertain to compensation or correction, timing, access control, etc. Another sensed variable is current of electrical servo-drives, but it is less commonly used; and extraction of load-related information from the measurement of current or temporally correlating it with position or velocity measurement is even less common. As a result, the net present value of the less common measurements and properties would be limited. However, if their initial modeling investment could be reduced, they could become more affordable. Therefore, in our evolutionary domain-modeling approach, illustrated in Figure 3, we will identify, plan, and execute a sequence of incremental extensions that would balance initial commitment and incremental investment.

2.3 Concept of reusable assets library

To illustrate our approach to reusable assets, let us consider libraries associated with the *C* Programming Language, e.g., for input-output and mathematical functions. Reuse of such library software has become commonplace, reducing programming labor. However, the library code is limited in its adaptability. In several modern implementation languages [13, 14, 17, 20], the notion of libraries has been extended to a more adaptable concept—class libraries. Through specialization of a given class, its user may modify or add functions, in a reliable way. General purpose class libraries are commercially available through the major compiler suppliers. Matrix-math class libraries are available from other sources serving a more specialized market. In the ICLP domain, the APIs are the specifications for ICLP domain-specific class libraries. We envision an infrastructure of specialized suppliers to grow in this market.

2.4 ICLP Domain Model

Figure 1 portrays the conceptual organization of class libraries useful in building machine control systems. Consider the analogy of building materials. The *primitive data types*, shown at the bottom of Figure 1, are akin to such raw materials as sand, gravel, and clay. Building computer programs from such primitives is too time consuming. Therefore, we resort to various kinds of "bricks and cinder blocks" (foundation class libraries) for the world of engineered systems. Examples are the various *units* and *measures*, *arrays*, *matrices*, and such other *containers* of information. With the help of such building blocks, we build classes for *geometry*, which,

in turn, are used to build classes for *kinematic structures* of rigid mechanical systems, e.g. workholding fixtures and other tooling. This hierarchy of aggregating structural components is based upon the ISO Standard 10103–Part 105 [11]. Using the reusable assets mentioned so far, we build class libraries for such motion components as *sensors*, *actuators*, and *pid control laws*. From a relatively smaller number of such components, we can compose a much larger variety of *machine tool axes* or *robotic joints*—the atomic, independently controllable units of motion, from which motions of a complete *machine* can be composed. We can see that software is organized in classes that mirror concepts used by machinery engineers and the real objects of their creation. This organization of software localizes the effects of changes made by machinery engineers, whether to provide new functions to users, or to improve performance, or to upgrade components in the machine.
Not all software components have physical equivalents. For example, an axis group is only a logical entity.

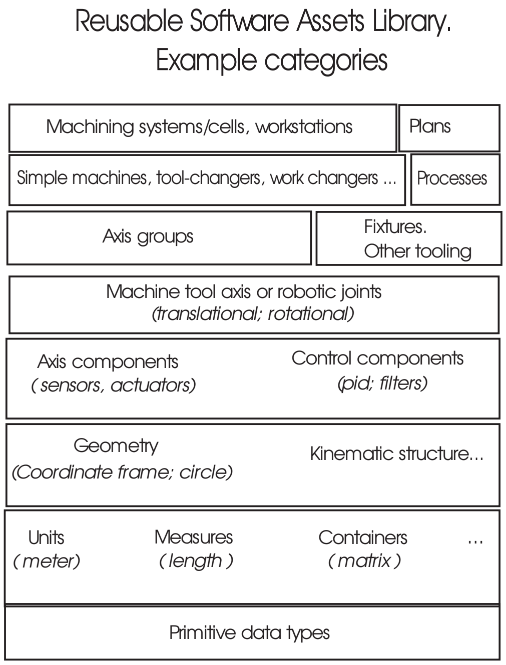

Figure 1: Organization of reusable assets library

2.5 Integrating a controller from assets in library

Figure 2 is an example of a control system consisting of software components that are instances of the class libraries described above. The specifier or integrator of the system determines its configuration The system may consist of one or more cooperating programs—tremendous flexibility is obtained from orthogonality in decisions about component-selection and programs (the "stovepipes" in Figure 2) composed from the component-services (the ovals in Figure 2). These programs may run on one or more processors (labeled P in Figure 2. Any of these programs may have "glue" code that ties together the services invoked from any of the software objects (the ovals in Figure 2).

2.6 ICLP Domain-independent services

Real-time control software must eventually be integrated with platform services in order to be executable in a way that satisfies specified requirements. The variety of requirements, possibly satisfying platform configurations, and performance-cost spread in the solution space is very large. Thus the demand (in number of units) for any single integrated solution will be much smaller than the potential market (in number of units) for the constituent services.

2.6.1 System software services.

The various cooperating processes that comprise a control system may be utilizing (platform) software services not yet specified in the ICLP APIs, e.g., resource management, low-level interface to hardware, and interprocess communication. To the extent that abstractions for these services have been defined in emerging standards [9, 10], we expect that the ICLP APIs will build on those abstractions. Additional effort may be required to bridge the gap, e.g., in resource models.

2.6.2 Hardware services.

The processing nodes may be interconnected with any medium, e.g., one or more networks (labeled N in Figure 2) or bus) chosen by the system specifier. Similarly, IO may be interconnected to a processor through a bus (labeled B in Figure 2) or through networks. In this respect, openness in real-time control systems is much more challenging than openness in non-real-time data processing systems. The ICLP community needs a suite of device interfaces at a finer level of granularity than the non-real-time data processing systems. The good news is that there is an adequate technology base to define a suite of abstract device interfaces for the commonly used hardware services (e.g., communication, interrupt, timing). The bad news is that the variety in hardware is large and the volume (number) of applications is so small that the unit amortized cost will be large, relative to such consumer devices as "sound boards". Again, abstractions defined at a fine granularity will provide wider coverage, and thus help reduce the unit amortized cost.

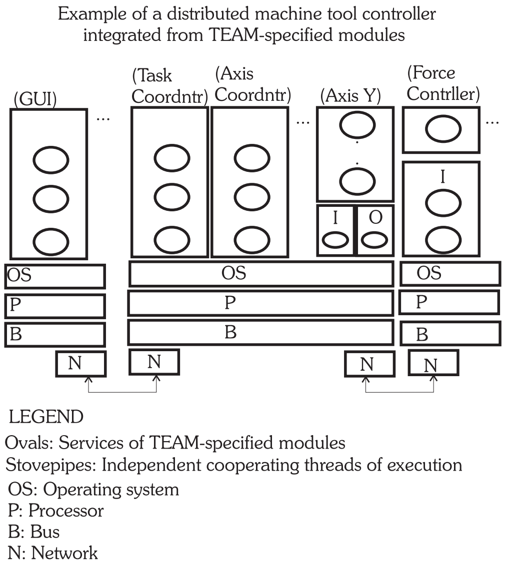

Figure 2: Example integrated control system

2.6.3 Implementation dependencies.

The various levels of abstractions discussed above help localize dependencies on various services (third party application software components, supporting software system services, processor services, network services), so that the cost of change is contained. However, the system specifier and integrator has to make some vendor-independent implementation decisions that limit the selection and interchangeability of software components, e.g.,

- Processor: Will the component run on a PC-clone?
- Language: If the component is to be purchased in source code form, in what programming language should it be?
- Bus/Network: If real-time functions are distributed, what interconnection scheme will be used—PCI, ISA, EISA, CAN, SERCOS, SP50 …?
- Message format: If the interconnection is a serial network such as the Controller Area Network, how is the information in each transmittal encoded?

2.6.4 How to assure interchangeability?

For the duration over which one wishes to assure interchangeability of components conforming to the APIs, the system specifier must commit to a particular implementation language and interface to platform resources. (See Michaloski imece96 paper for more information on the development cycle and choices to be constrained). The paying organization may find it more economic to extend these constraints to all projects over a certain period. It is up to the system specifiers (projects; organizations) when to break such constraints and give up future interchangeability. In any case, the ICLP APIs will help reduce the application-software cost of innovation in intelligent manufacturing systems, in comparison to the alternatives available today. When the ICLP community proceeds to its first prototype based upon these APIs, it is expected that the project team will propose or request additions to the APIs, from the needs and experiences of the project. Such iteration and evolution will be a continuing process, as illustrated in Figure 3. Nevertheless, the underlying architectural principles will endure and support the incremental evolutionary process.

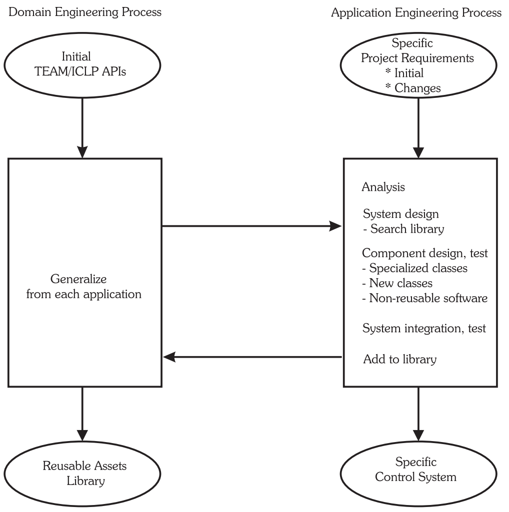

Figure 3: Growth cycle of reusable assets for ICLP domain

3 Object interactions in the operational mode of a machine tool

Figure 4 gives a view of interactions amongst the major software components in the traditional operational mode of a numerically controlled machine tool. Here we describe the generality supported by the architecture.

3.1 User interfaces

There may be many different kinds of user interfaces, even in a single implementation, e.g., to meet the needs of different types of users [16] ranging from the software system developer to the machine operator. Access may be correspondingly restricted according to the policy of the system owner. Figure 4 shows a user interface in the operational mode, i.e., the controller and the machine tool are ready to run—the setup necessary to machine a workpiece has already been performed. Suppose that the operator wants to process the part in the automatic operation mode from stored instructions, known as a part program in the tradition of numerical control. The interface to the storage (shown as a file system in Figure 4 should be independent of implementation.

3.2 User process program

The traditional part program format for numerical control (EIA RS274) is too restrictive. A "BCL" version overcomes some of its shortcomings. However, the community using coordinate measurement machines was not satisfied with numerical control programming standards—it has developed its own standard, "DMIS". The community using robots is seeking even more generality—several different vendor-specific "languages", e.g., *KAREL, V+, etc.*, are popular in industry. Programming for discrete control is performed in yet another group of vendor-specific languages, with IEC 1131-3 as the only vendor-neutral standard on the horizon. The term user process program applies to any type of process or machine discussed above. We assume that industry-specific formats will continue to exist. However, none of them provide an integrated solution needed for machines that integrate material-processing, material-handling, dimensional measurement, other types of inspection, and discrete control functions. There is a need for a canonical form of processing instructions—we call it a control plan—that provides the expressivity needed by the ICLP community.

3.3 Translator

In view of the wide range of user process programming forms, there will be a need for a translator specific to each form. However, all translators will produce a unified canonical form, consistent with and mappable into IEC 1131-specified forms. As an initial step in the evolutionary process, we have defined a RS274Translator. It translates motion commands into more general forms of continuous motion functions, based on ISO 10303 [11, 12]. Similarly, it maps "M codes" for miscellaneous functions into the more general model of discrete control and other services.

3.4 Control plan

Our information model provides a unified paradigm for control of discrete and continuous processes and devices. A control plan specifies the automatic sequence of operations of a complete machinetool. It contains a sequence of control plan steps or units, each specifying the operation to be performed in that step. These specifications are independent of the implementing device, so that the control plan may be executed on any machine with the necessary capabilities. In Figure 4, we label such a control plan as machine-independent. A workpiece-specific control plan may be nested under a higher-level control plan that does not change from one workpiece-processing program to another, e.g., logic for startup, shutdown, and emergency. When a machine-independent control plan is sent to be run on a machine, it must be tranformed into a machine-specific control plan, using information about the configuration and the kinematic model of the machine.

3.5 Machine tool

This software component holds information about the machine tool constituents, kinematic model, and services provided directly, e.g., selection of usage modes, startup and shutdown of the machine tool. This model applies to a wide range of machine tools, coordinate measurement machines, other types of dimension measuring equipment, inspection equipment, and robotic equipment used in manufacturing.

The basic machine tool is modeled as a physical resource with a kinematic structure to support or drive the workpiece and a kinematic structure to support or drive the tool or end effector. In its simplest form, the kinematic structure may be a connection (including the coordinate transform) to GROUND (the coordinate frame of reference for this machine tool). In a more general form, the kinematic structure may be composed of other kinematic structures through connections that define the placement and coordinate transform from one kinematic structure to the next. The interconnections between kinematic structures are based on the same D-H convention [7] that is used to model any one kinematic structure. Kinematic errors of motion are included as a part of this model, consistent with the scheme devised at NIST [8].

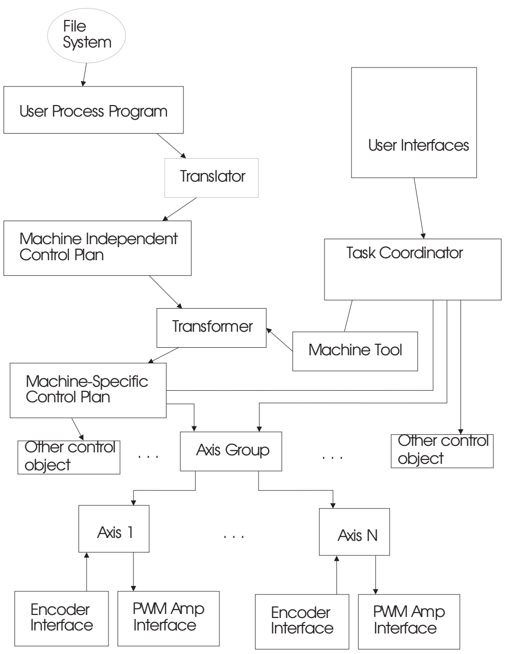

Figure 4: Interactions of software objects—operational mode

3.6 Axis Group

The role of an axis group is to coordinate servo-controlled motion of a group of machine tool axes (or robotic joints) to a specified trajectory produced from their concurrent, coordinated motion. The trajectory specification usually includes the path of the designated workpoint (e.g. tool center) in space and velocity along that path. It may also include other constraints, e.g. path-deviation, velocity-deviation, allowable jerk, allowable processing force or torque, or some other process parameter. The services of an axis group include interpolation and generation of the velocity profile for each axis. User-specified motion in the control plan maps into services of an axis group. Outputs to the axes, e.g., desired position and desired velocity are spaced at their servo-loop update intervals. It also checks for status, e.g., in position or some exception posted by each axis. The information needed about the characteristics, capabilities, and constraints of the axes is obtained from the respective axis objects and "cached" in the axis group at the time of configuration and setup, with a request to the axis for notification of change. Upon such notification, the axis group would update the "cached" information.

A machine may have more than one axis group, e.g., for each arm of a two-armed robotic machine. An axis may be part of several different axis groups. In a series of user-specified motions, certain moves may specify one form of axis coordination, requiring the services of one axis group, e.g., a straight-line move involving three translational axes simultaneously, and subsequent moves may specify other forms of coordination, requiring the services of another axis group, e.g., a helical move involving one of the translational axes and a spindle. Thus, our architecture provides a consistent interface for a variety of situations which, historically, were treated differently.

3.7 Other control object

It may control an on-off device (traditionally performed in a programmable logic controller) or it may perform a "supervisory" computation for some set point, e.g., feedrate override by a force-constraint controller, or the computation for some alarm indicator, e.g., detection of a broken tool. Thus, the model provides a unified framework for integrating a producer of a discrete event and a producer of a continuous variable. The system designer has the flexibility to select and "wire up" the various software components needed in a particular system.

3.8 Axis

This software component encapsulates the data and operations pertaining to a single axis in a (possibly multi-axis) control system. It includes data and operations for operation, setup, maintenance (test, calibration, diagnosis), etc. An axis may be rotational or translational. The properties of an axis of motion may be derived from its constituent model (kinematic and dynamic), at the time of configuration, setup, or calibration. Such a modeling framework allows uniform integration of design information into control information. An axis contains a separately replaceable control component—given the desired state (e.g., position, velocity) and the current state, it computes the next state. The "control component" receives all information in consistent units of the parameters being controlled. It does not have to deal with conversions. The axis maintains current state in a sensed state component, which receives information from the sensors and converts it into the proper units for the state of the axis. The conversion factors are obtained from the kinematic model of the axis. Similarly, an actuator state component receives the "next state" information from the "control component" and converts it into the units for the actuator.

3.9 Interfaces to sensors and actuators

Figure 4 depicts a particular kind of sensor (incremental position encoder) and a particular kind of actuator (pulse-width modulated power-amplifier drive). Our information model provides a generalization-specialization hierarchy for device interfaces, in order to reduce the effort required to interface a new device.

3.10 Tasks and Task Coordinator

As mentioned in Section 2.5, the services of the software components described above may be organized in several concurrent cooperating threads of execution, which we label as *tasks*. These tasks may be operating system processes on the same processor, or more likely distributed across several processors, possibly procured as packaged subsystems from different sources. The designer or integrator of the system uses the task coordinator for application-specific customization of the way the selected software components are to be run. In the continuous automatic operation mode, the task coordinator regulates or paces the execution of the control plan. Under normal conditions, there is no need for its intervention, since our architecture decomposes and delegates all normal work to the various software components described above. The task coordinator intervenes in the event of an exception that none of the other software components is able to handle. When a user selects some mode of operation other than the continuous automatic operation mode, the task coordinator helps avoid inconsistencies.

The task coordinator also plays a controlling role in the startup and shutdown of the control system.

4 Conclusion

Status. The architecture has evolved to the stage that our open-architecture testbed has generic building blocks for internal test and evaluation to provide three axes of coordinated linear motion. For a given hardware configuration, the defined interfaces allow specification of the axes to be coordinated, the nature of the coordination, the accuracy of a move, choice of servo-control components, gain parameters, and sampling intervals.

Issues. The state of software practice amongst researchers, developers, and practitioners in the manufacturing automation field is much below the state of the art. An open environment at the real-time level will require significant upgrade in the skill base of the industry. On the other hand, software professionals do not understand the subtle issues and cost-sensitivities of the manufactuirng automation field. Solutions that worked for defense and aerospace applications do not fit the cost-performance needs of this industry.

Future work. Our plan for future work is to continue evolving the architecture incrementally and iteratively. Its effectiveness will be tested through building working prototypes: extension of motion control to five coordinated axes, fusion of sensed position, velocity, current, and cutting forces, addition of process control, e.g., force-constrained adaptive control, protection against cutter damage, recognition and avoidance of chatter, and accuracy enhancement. The infrastructure of the control system will also be extended to support distributed control, deploying and evaluating several different media for communication, e.g., Controller Area Network applied at servo-motion update rates and switched ethernet applied to other real-time commnication. Through a series of such evolutionary steps, we will evaluate the cost-effectiveness of our approach to open-architecture control systems for intelligent manufacturing.

References

[1] 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.

[2] 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*, Ann Arbor, Michigan, May 1995.

[3] 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*, May 1995.

[4] S. K. Birla, K. G. Shin, and J. Albus, "Incorporating intelligence in real-time control of manufacturing automation: an architectural approach," in *Proceedings of the SIGAda Artificial Intelligence Working Group, Washington Ada Symposium, 1994*, McClean Hilton, Virginia, June 1994.

[5] R. J. Brachman, *On the epistemological status of semantic networks, associative networks: Representation and use of knowledge by computers*, pp. 3–50, Academic Press, New York, 1979.

[6] S. Conte, A. Dunsmore, and V. Shen, *Software engineering metrics and models*, Benjamin Cummings, Reading, Mass, 1986.

[7] J. Denavit and R. Hartenberg, "A kinematic notation for lower-pair mechanisms based on matrices," *Journal of Applied Mechanics*, pp. 215–221, june 1955.

[8] 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.

[9] O. M. Group, *The Object Management Architecture Guide*, Object Management Group, Framingham, MA, June 1995.

[10] *Portable operating system interface (POSIX)—Part 1: Application Program Interface (API) [C Language]—Amendment: Realtime Extensions*, Institute of Electrical and Electroics Engineers, 1994.

[11] 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.

[12] 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.

[13] B. Meyer, *Object-Oriented Software Construction (Eiffel Language)*, Englewood Cliffs, New Jersey., 1988.

[14] H. Mossenbock and N. Wirth, "The programming language oberon-2," *Structured Programming*, vol. 12, no. 4, , 1991.

[15] A. Newell, *Unified Theories of Cognition*, Harvard University Press, Cambridge, MA, U.S.A., 1990.

[16] *Next Generation Workstation/Machine Controller (NGC) Requirements Definition Document (RDD)*, 1989.

[17] M. Odersky, "Extending modula-2 for object-oriented programming (modula-90 language)," in *Proceedings of the First International Modula-2 Conference*, Bled, Yugoslavia, oct 1989, AIAA.

[18] D. L. Parnas, "On the design and development of program families," *IEEE Trans. Software Engineering*, vol. SE-2, pp. 1–9, March 1976.

[19] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. Software Engineering*, vol. SE-5, no. 2, pp. 128–138, March 1979.

[20] B. Stroustrup, *The C++ Programming Language*, Addison Wesley, Reading, MA, 1991.